1 Fundamentals of Computer Design

And now for something completely different.

Monty Python's Flying Circus

1.1	Introduction	1
1.2	The Task of a Computer Designer	8
1.3	Technology Trends	11
1.4	Cost, Price and their Trends	14
1.5	Measuring and Reporting Performance	25
1.6	Quantitative Principles of Computer Design	40
1.7	Putting It All Together: Performance and Price-Performance	49
1.8	Another View: Power Consumption and Efficiency as the Metric	58
1.9	Fallacies and Pitfalls	59
1.10	Concluding Remarks	69
1.11	Historical Perspective and References	70
	Exercises	77

1.1 Introduction

Computer technology has made incredible progress in the roughly 55 years since the first general-purpose electronic computer was created. Today, less than a thousand dollars will purchase a personal computer that has more performance, more main memory, and more disk storage than a computer bought in 1980 for $1 million. This rapid rate of improvement has come both from advances in the technology used to build computers and from innovation in computer design.

Although technological improvements have been fairly steady, progress arising from better computer architectures has been much less consistent. During the first 25 years of electronic computers, both forces made a major contribution; but beginning in about 1970, computer designers became largely dependent upon integrated circuit technology. During the 1970s, performance continued to improve at about 25% to 30% per year for the mainframes and minicomputers that dominated the industry.

The late 1970s saw the emergence of the microprocessor. The ability of the microprocessor to ride the improvements in integrated circuit technology more closely than the less integrated mainframes and minicomputers led to a higher rate of improvement—roughly 35% growth per year in performance.

This growth rate, combined with the cost advantages of a mass-produced microprocessor, led to an increasing fraction of the computer business being based on microprocessors. In addition, two significant changes in the computer marketplace made it easier than ever before to be commercially successful with a new architecture. First, the virtual elimination of assembly language programming reduced the need for object-code compatibility. Second, the creation of standardized, vendor-independent operating systems, such as UNIX and its clone, Linux, lowered the cost and risk of bringing out a new architecture.

These changes made it possible to successfully develop a new set of architectures, called RISC (Reduced Instruction Set Computer) architectures, in the early 1980s. The RISC-based machines focused the attention of designers on two critical performance techniques, the exploitation of instruction-level parallelism (initially through pipelining and later through multiple instruction issue) and the use of caches (initially in simple forms and later using more sophisticated organizations and optimizations). The combination of architectural and organizational enhancements has led to 20 years of sustained growth in performance at an annual rate of over 50%. Figure 1.1 shows the effect of this difference in performance growth rates.

The effect of this dramatic growth rate has been twofold. First, it has significantly enhanced the capability available to computer users. For many applications, the highest performance microprocessors of today outperform the supercomputer of less than 10 years ago.

Second, this dramatic rate of improvement has led to the dominance of microprocessor-based computers across the entire range of the computer design. Workstations and PCs have emerged as major products in the computer industry. Minicomputers, which were traditionally made from off-the-shelf logic or from gate arrays, have been replaced by servers made using microprocessors. Mainframes have been almost completely replaced with multiprocessors consisting of small numbers of off-the-shelf microprocessors. Even high-end supercomputers are being built with collections of microprocessors.

Freedom from compatibility with old designs and the use of microprocessor technology led to a renaissance in computer design, which emphasized both architectural innovation and efficient use of technology improvements. This renaissance is responsible for the higher performance growth shown in Figure 1.1—a rate that is unprecedented in the computer industry. This rate of growth has compounded so that by 2001, the difference between the highest-performance microprocessors and what would have been obtained by relying solely on technology, including improved circuit design, is about a factor of fifteen.

In the last few years, the tremendous imporvement in integrated circuit capability has allowed older less-streamlined architectures, such as the x86 (or IA-32) architecture, to adopt many of the innovations first pioneered in the RISC designs. As we will see, modern x86 processors basically consist of a front-end that fetches and decodes x86 instructions and maps them into simple ALU, memory access, or branch operations that can be executed on a RISC-style pipelined pro-

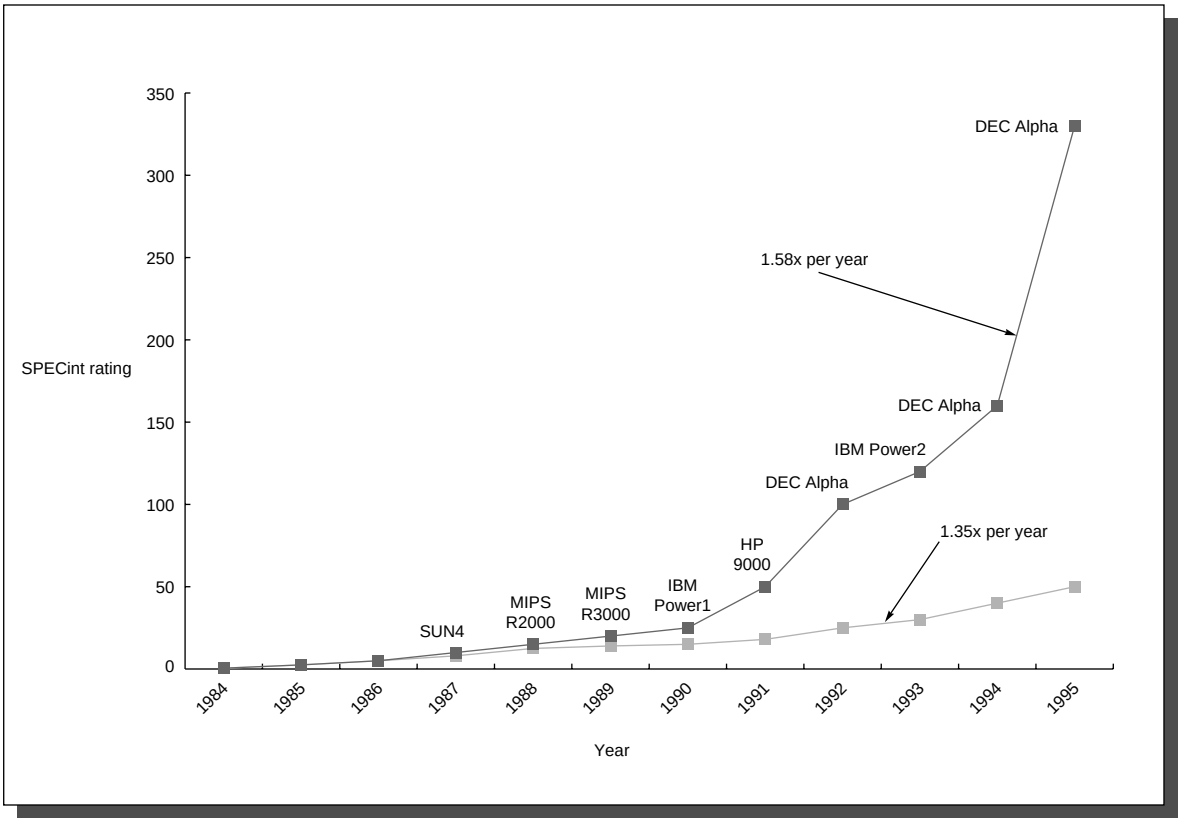


FIGURE 1.1 Growth in microprocessor performance since the mid 1980s has been substantially higher than in earlier years as shown by plotting SPECint performance. This chart plots relative performance as measured by the SPECint benchmarks with base of one being a VAX 11/780. (Since SPEC has changed over the years, performance of newer machines is estimated by a scaling factor that relates the performance for two different versions of SPEC (e.g. SPEC92 and SPEC95.) Prior to the mid 1980s, microprocessor performance growth was largely technology driven and averaged about 35% per year. The increase in growth since then is attributable to more advanced architectural and organizational ideas. By 2001 this growth leads to about a factor of 15 difference in performance. Performance for floating-point-oriented calculations has increased even faster.

Change this figure as follows:
!1. the y-axis should be labeled "Relative Performance."
2. Plot only even years
3. The following data points should changed/added:
a. 1992 136 HP 9000; 1994 145 DEC Alpha; 1996 507 DEC Alpha; 1998 879 HP 9000; 2000 1582 Intel Pentium III
4. Extend the lower line by increasing by 1.35x each year

cessor. Beginning in the end of the 1990s, as transistor counts soared, the overhead in transistors of interpreting the more complex x86 architecture became neglegible as a percentage of the total transistor count of a modern microprocessor.

This text is about the architectural ideas and accompanying compiler improvements that have made this incredible growth rate possible. At the center of this dramatic revolution has been the development of a quantitative approach to computer design and analysis that uses empirical observations of programs, experimentation, and simulation as its tools. It is this style and approach to computer design that is reflected in this text.

Sustaining the recent improvements in cost and performance will require continuing innovations in computer design, and the authors believe such innovations will be founded on this quantitative approach to computer design. Hence, this book has been written not only to document this design style, but also to stimulate you to contribute to this progress.

1.2 The Changing Face of Computing and the Task of the Computer Designer

In the 1960s, the dominant form of computing was on large mainframes, machines costing millions of dollars and stored in computer rooms with multiple operators overseeing their support. Typical applications included business data processing and large-scale scientific computing. The 1970s saw the birth of the minicomputer, a smaller sized machine initially focused on applications in scientific laboratories, but rapidly branching out as the technology of timesharing, multiple users sharing a computer interactively through independent terminals, became widespread. The 1980s saw the rise of the desktop computer based on microprocessors, in the form of both personal computers and workstations. The individually owned desktop computer replaced timesharing and led to the rise of servers, computers that provided larger-scale services such as: reliable, long-term file storage and access, larger memory, and more computing power. The 1990s saw the emergence of the Internet and the world-wide web, the first successful handheld computing devices (personal digital assistants or PDAs), and the emergence of high-performance digital consumer electronics, varying from video games to set-top boxes.

These changes have set the stage for a dramatic change in how we view computing, computing applications, and the computer markets at the beginning of the millennium. Not since the creation of the personal computer more than twenty years ago have we seen such dramatic changes in the way computers appear and in how they are used. These changes in computer use have led to three different computing markets each characterized by different applications, requirements, and computing technologies.

Desktop Computing

The first, and still the largest market in dollar terms, is desktop computing. Desktop computing spans from low-end systems that sell for under $1,000 to high-end, heavily-configured workstations that may sell for over $10,000. Throughout this range in price and capability, the desktop market tends to be driven to optimize *price-performance*. This combination of performance (measured primarily in terms of compute performance and graphics performance) and price of a system is what matters most to customers in this market and hence to computer designers. As a result desktop systems often are where the newest, highest performance microprocessors appear, as well as where recently cost-reduced microprocessors and systems appear first (see section 1.4 on page 14 for a discussion of the issues affecting cost of computers).

Desktop computing also tends to be reasonably well characterized in terms of applications and benchmarking, though the increasing use of web-centric, interactive applications poses new challenges in performance evaluation. As we discuss in Section 1.9 (Fallacies, Pitfalls), the PC portion of the desktop space seems recently to have become focused on clock rate as the direct measure of performance, and this focus can lead to poor decisions by consumers as well as by designers who respond to this predilection.

Servers

As the shift to desktop computing occurred, the role of servers to provide larger scale and more reliable file and computing services grew. The emergence of the world-wide web accelerated this trend due to the tremendous growth in demand for web servers and the growth in sophistication of web-based services. Such servers have become the backbone of large-scale enterprise computing replacing the traditional mainframe.

For servers, different characteristics are important. First, availability is critical. We use the term availability, which means that the system can reliably and effectively provide a service. This term is to be distinguished from reliability, which says that the system never fails. Parts of large-scale systems unavoidably fail; the challenge in a server is to maintain system availability in the face of component failures, usually through the use of redundancy. This topic is discussed in detail in Chapter 6.

Why is availability crucial? Consider the servers running Yahoo!, taking orders for Cisco, or running auctions on EBay. Obviously such systems must be operating seven days a week, 24 hours a day. Failure of such a server system is far more catastrophic than failure of a single desktop. Although it is hard to estimate the cost of downtime, Figure 1.2 shows one analysis, assuming that downtime is distributed uniformly and does not occur solely during idle times. As we can see, the estimated costs of an unavailable system are high, and the estimated costs in

Figure 1.2 are purely lost revenue and do not account for the cost of unhappy customers!

Application	Cost of downtime per hour (thousands of $)	Annual losses (millions of $) with downtime of		
		1% (87.6 hrs/yr)	0.5% (43.8 hrs/yr)	0.1% (8.8 hrs/yr)
Brokerage operations	$6,450	$565	$283	$56.5
Credit card authorization	$2,600	$228	$114	$22.8
Package shipping services	$150	$13	$6.6	$1.3
Home shopping channel	$113	$9.9	$4.9	$1.0
Catalog sales center	$90	$7.9	$3.9	$0.8
Airline reservation center	$89	$7.9	$3.9	$0.8
Cellular service activation	$41	$3.6	$1.8	$0.4
On-line network fees	$25	$2.2	$1.1	$0.2
ATM service fees	$14	$1.2	$0.6	$0.1

FIGURE 1.2 The cost of an unavailable system is shown by analyzing the cost of downtime (in terms of immediately lost revenue), assuming three different levels of availability. This assumes downtime is distributed uniformly. This data is from Kembel [2000] and was collected an analyzed by Contingency Planning Research.

A second key feature of server systems is an emphasis on scalability. Server systems often grow over their lifetime in response to a growing demand for the services they support or an increase in functional requirements. Thus, the ability to scale up the computing capacity, the memory, the storage, and the I/O bandwidth of a server are crucial.

Lastly, servers are designed for efficient throughput. That is, the overall performance of the server–in terms of transactions per minute or web pages served per second–is what is crucial. Responsiveness to an individual request remains important, but overall efficiency and cost-effectiveness, as determined by how many requests can be handled in a unit time, are the key metrics for most servers. (We return to the issue of performance and assessing performance for different types of computing environments in Section 1.5 on page 25).

Embedded Computers

Embedded computers, the name given to computers lodged in other devices where the presence of the computer is not immediately obvious, are the fastest growing portion of the computer market. The range of application of these devices goes from simple embedded microprocessors that might appear in a everyday machines (most microwaves and washing machines, most printers, most networking switches, and all cars contain such microprocessors) to handheld digital devices (such as palmtops, cell phones, and smart cards) to video games and digital set-top boxes. Although in some applications (such as palmtops) the comput-

ers are programmable, in many embedded applications the only programming occurs in connection with the initial loading of the application code or a later software upgrade of that application. Thus, the application can usually be carefully tuned for the processor and system; this process sometimes includes limited use of assembly language in key loops, although time-to-market pressures and good software engineering practice usually restrict such assembly language coding to a small fraction of the application. This use of assembly language, together with the presence of standardized operating systems, and a large code base has meant that instruction set compatibility has become an important concern in the embedded market. Simply put, like other computing applications, software costs are often a large factor in total cost of an embedded system.

Embedded computers have the widest range of processing power and cost. From low-end 8-bit and 16-bit processors that may cost less than a dollar, to full 32-bit microprocessors capable of executing 50 million instructions per second that cost under $10, to high-end embedded processors (that can execute a billion instructions per second and cost hundreds of dollars) for the newest video game or for a high-end network switch. Although the range of computing power in the embedded computing market is very large, price is a key factor in the design of computers for this space. Performance requirements do exist, of course, but the primary goal is often meeting the performance need at a minimum price, rather than achieving higher performance at a higher price.

Often, the performance requirement in an embedded application is a real-time requirement. A *real-time* performance requirement is one where a segment of the application has an absolute maximum execution time that is allowed. For example, in a digital set-top box the time to process each video frame is limited, since the processor must accept and process the next frame shortly. In some applications, a more sophisticated requirement exists: the average time for a particular task is constrained as well as the number of instances when some maximum time is exceeded. Such approaches (sometimes called *soft real-time*) arise when it is possible to occasionally miss the time constraint on an event, as long as not too many are missed. Real-time performance tend to be highly application dependent. It is usually measured using kernels either from the application or from a standardized benchmark (see the EEMBC benchmarks described in Section 1.5). With the growth in the use of embedded microprocessors, a wide range of benchmark requirements exist, from the ability to run small, limited code segments to the ability to perform well on applications involving tens to hundreds of thousands of lines of code.

Two other key characteristics exist in many embedded applications: the need to minimize memory and the need to minimize power. In many embedded applications, the memory can be substantial portion of the system cost, and memory size is important to optimize in such cases. Sometimes the application is expected to fit totally in the memory on the processor chip; other times the applications needs to fit totally in a small off-chip memory. In any event, the importance of memory size translates to an emphasis on code size, since data size is dictated by

the application. As we will see in the next chapter, some architectures have special instruction set capabilities to reduce code size. Larger memories also mean more power, and optimizing power is often critical in embedded applications. Although the emphasis on low power is frequently driven by the use of batteries, the need to use less expensive packaging (plastic versus ceramic) and the absence of a fan for cooling also limit total power consumption.We examine the issue of power in more detail later in the chapter.

Another important trend in embedded systems is the use of processor cores together with application-specific circuitry. Often an application's functional and performance requirements are met by combining a custom hardware solution together with software running on a standardized embedded processor core, which is designed to interface to such special-purpose hardware. In practice, embedded problems are usually solved by one of three approaches:

1. using a combined hardware/software solution that includes some custom hardware and typically a standard embedded processor,
2. using custom software running on an off-the-shelf embedded processor, or
3. using a digital signal processor and custom software. (Digital signal processors are processors specially tailored for signal processing applications. We discuss some of the important differences between digital signal processors and general-purpose embedded processors in the next chapter.)

Most of what we discuss in this book applies to the design, use, and performance of embedded processors, whether they are off-the-shelf microprocessors or microprocessor cores, which will be assembled with other special-purpose hardware. The design of special-purpose application-specific hardware and the detailed aspects of DSPs, however, are outside of the scope of this book.

Figure 1.3 summarizes these three classes of computing environments and their important characteristics.

The Task of a Computer Designer

The task the computer designer faces is a complex one: Determine what attributes are important for a new machine, then design a machine to maximize performance while staying within cost and power constraints. This task has many aspects, including instruction set design, functional organization, logic design, and implementation. The implementation may encompass integrated circuit design, packaging, power, and cooling. Optimizing the design requires familiarity with a very wide range of technologies, from compilers and operating systems to logic design and packaging.

In the past, the term *computer architecture* often referred only to instruction set design. Other aspects of computer design were called *implementation*, often

Feature	Desktop	Server	Embedded
Price of system	$1,000–$10,000	$10,000–$10,000,000	$10–$100,000 (including network routers at the high-end)
Price of microprocessor module	$100–$1,000	$200–$2000 (per processor)	$0.20–$200
Microprocessors sold per year (estimates for 2000)	150,000,000	4,000,000	300,000,000 (32-bit and 64-bit processors only)
Critical system design issues	Price-performance Graphics performance	Throughput Availability Scalability	Price Power consumption Application-specific performance

FIGURE 1.3 A summary of the three computing classes and their system characteristics. The total number of embedded processors sold in 2000 is estimated to exceed 1 billion, if you include 8-bit and 16-bit microprocessors. In fact, the largest selling microprocessor of all time is an 8-bit microcontroller sold by Intel! It is difficult to separate the low end of the server market from the desktop market, since low-end servers–especially those costing less than $5,000–are essentially no different from desktop PCs. Hence, up to a few million of the PC units may be effectively servers.

insinuating that implementation is uninteresting or less challenging. The authors believe this view is not only incorrect, but is even responsible for mistakes in the design of new instruction sets. The architect's or designer's job is much more than instruction set design, and the technical hurdles in the other aspects of the project are certainly as challenging as those encountered in doing instruction set design. This challenge is particularly acute at the present when the differences among instruction sets are small and at a time when there are three rather distinct applications areas.

In this book the term *instruction set architecture* refers to the actual programmer-visible instruction set. The instruction set architecture serves as the boundary between the software and hardware, and that topic is the focus of Chapter 2. The implementation of a machine has two components: organization and hardware. The term *organization* includes the high-level aspects of a computer's design, such as the memory system, the bus structure, and the design of the internal CPU (central processing unit—where arithmetic, logic, branching, and data transfer are implemented). For example, two processors with nearly identical instruction set architectures but very different organizations are the Pentium III and Pentium 4. Although the Pentium 4 has new instructions, these are all in the floating point instruction set. *Hardware* is used to refer to the specifics of a machine, including the detailed logic design and the packaging technology of the machine. Often a line of machines contains machines with identical instruction set architectures and nearly identical organizations, but they differ in the detailed hardware implementation. For example, the Pentium II and Celeron are nearly identical, but offer different clock rates and different memory systems, making the Celron more effective for low-end computers. In this book the word *architecture* is intended to cover all three aspects of computer design—instruction set architecture, organization, and hardware.

Computer architects must design a computer to meet functional requirements as well as price, power, and performance goals. Often, they also have to determine what the functional requirements are, and this can be a major task. The requirements may be specific features inspired by the market. Application software often drives the choice of certain functional requirements by determining how the machine will be used. If a large body of software exists for a certain instruction set architecture, the architect may decide that a new machine should implement an existing instruction set. The presence of a large market for a particular class of applications might encourage the designers to incorporate requirements that would make the machine competitive in that market. Figure 1.4 summarizes some requirements that need to be considered in designing a new machine. Many of these requirements and features will be examined in depth in later chapters.

Functional requirements	Typical features required or supported
Application area	Target of computer
General purpose desktop	Balanced performance for a range of tasks, including interactive performance for graphics, video, and audio (Ch 2,3,4,5)
Scientific desktops and servers	High-performance floating point and graphics (App A,B)
Commercial servers	Support for databases and transaction processing, enhancements for reliability and availability. Support for scalability. (Ch 2,7)
Embedded computing	Often requires special support for graphics or video (or other application-specific extension). Power limitations and power control may be required. (Ch 2,3,4,5)
Level of software compatibility	Determines amount of existing software for machine
At programming language	Most flexible for designer; need new compiler (Ch 2,8)
Object code or binary compatible	Instruction set architecture is completely defined—little flexibility—but no investment needed in software or porting programs
Operating system requirements	Necessary features to support chosen OS (Ch 5,7)
Size of address space	Very important feature (Ch 5); may limit applications
Memory management	Required for modern OS; may be paged or segmented (Ch 5)
Protection	Different OS and application needs: page vs. segment protection (Ch 5)
Standards	Certain standards may be required by marketplace
Floating point	Format and arithmetic: IEEE 754 standard (App A), special arithmetic for graphics or signal processing
I/O bus	For I/O devices: Ultra ATA, Ultra SCSI, PCI (Ch 6)
Operating systems	UNIX, PalmOS, Windows, Windows NT, Windows CE, CISCO IOS
Networks	Support required for different networks: Ethernet, Infiniband (Ch 7)
Programming languages	Languages (ANSI C, C++, Java, Fortran) affect instruction set (Ch 2)

FIGURE 1.4 Summary of some of the most important functional requirements an architect faces. The left-hand column describes the class of requirement, while the right-hand column gives examples of specific features that might be needed. The right-hand column also contains references to chapters and appendices that deal with the specific issues.

Once a set of functional requirements has been established, the architect must try to optimize the design. Which design choices are optimal depends, of course, on the choice of metrics. The changes in the computer applications space over the last decade have dramatically changed the metrics. Although desktop computers remain focused on optimizing cost-performance as measured by a single user, servers focus on availability, scalability, and throughput cost-performance, and embedded computers are driven by price and often power issues.

These differences and the diversity and size of these different markets leads to fundamentally different design efforts. For the desktop market, much of the effort goes into designing a leading-edge microprocessor and into the graphics and I/O system that integrate with the microprocessor. In the server area, the focus is on integrating state-of-the-art microprocessors, often in a multiprocessor architecture, and designing scalable and highly available I/O systems to accompany the processors. Finally, in the leading edge of the embedded processor market, the challenge lies in adopting the high-end microprocessor techniques to deliver most of the performance at a lower fraction of the price, while paying attention to demanding limits on power and sometimes a need for high performance graphics or video processing.

In addition to performance and cost, designers must be aware of important trends in both the implementation technology and the use of computers. Such trends not only impact future cost, but also determine the longevity of an architecture. The next two sections discuss technology and cost trends.

1.3 Technology Trends

If an instruction set architecture is to be successful, it must be designed to survive rapid changes in computer technology. After all, a successful new instruction set architecture may last decades—the core of the IBM mainframe has been in use for more than 35 years. An architect must plan for technology changes that can increase the lifetime of a successful computer.

To plan for the evolution of a machine, the designer must be especially aware of rapidly occurring changes in implementation technology. Four implementation technologies, which change at a dramatic pace, are critical to modern implementations:

- *Integrated circuit logic technology*—Transistor density increases by about 35% per year, quadrupling in somewhat over four years. Increases in die size are less predictable and slower, ranging from 10% to 20% per year. The combined effect is a growth rate in transistor count on a chip of about 55% per year. Device speed scales more slowly, as we discuss below.
- *Semiconductor DRAM* (dynamic random-access memory)—Density increases by between 40% and 60% per year, quadrupling in three to four years. Cycle time has improved very slowly, decreasing by about one-third in 10 years. Bandwidth per chip increases about twice as fast as latency decreases. In addi-

tion, changes to the DRAM interface have also improved the bandwidth; these are discussed in Chapter 5.

- *Magnetic disk technology*—Recently, disk density has been improving by more than 100% per year, quadrupling in two years. Prior to 1990, density increased by about 30% per year, doubling in three years. It appears that disk technology will continue the faster density growth rate for some time to come. Access time has improved by one-third in 10 years. This technology is central to Chapter 6, and we discuss the trends in greater detail there.
- *Network technology*—Network performance depends both on the performance of switches and on the performance of the transmission system, both latency and bandwidth can be improved, though recently bandwidth has been the primary focus. For many years, networking technology appeared to improve slowly: for example, it took about 10 years for Ethernet technology to move from 10 Mb to 100 Mb. The increased importance of networking has led to a faster rate of progress with 1 Gb Ethernet becoming available about five years after 100 Mb. The Internet infrastructure in the United States has seen even faster growth (roughly doubling in bandwidth every year), both through the use of optical media and through the deployment of much more switching hardware.

These rapidly changing technologies impact the design of a microprocessor that may, with speed and technology enhancements, have a lifetime of five or more years. Even within the span of a single product cycle for a computing system (two years of design and two to three years of production), key technologies, such as DRAM, change sufficiently that the designer must plan for these changes. Indeed, designers often design for the next technology, knowing that when a product begins shipping in volume that next technology may be the most cost-effective or may have performance advantages. Traditionally, cost has decreased very closely to the rate at which density increases.

Although technology improves fairly continuously, the impact of these improvements is sometimes seen in discrete leaps, as a threshold that allows a new capability is reached. For example, when MOS technology reached the point where it could put between 25,000 and 50,000 transistors on a single chip in the early 1980s, it became possible to build a 32-bit microprocessor on a single chip. By the late 1980s, first-level caches could go on-chip. By eliminating chip crossings within the processor and between the processor and the cache, a dramatic increase in cost/performance and performance/power was possible. This design was simply infeasible until the technology reached a certain point. Such technology thresholds are not rare and have a significant impact on a wide variety of design decisions

Scaling of Transistor Performance, Wires, and Power in Integrated Circuits

Integrated circuit processes are characterized by the *feature size*, which is the minimum size of a transistor or a wire in either the *x* or *y* dimension. Feature siz-

es have decreased from 10 microns in 1971 to 0.18 microns in 2001. Since a transistor is a 2-dimensional object, the density of transistors increases quadratically with a linear decrease in feature size. The increase in transistor performance, however, is more complex. As feature sizes shrink, devices shrink quadratically in the horizontal dimensions and also shrink in the vertical dimension. The shrink in the vertical dimension requires a reduction in operating voltage to maintain correct operation and reliability of the transistors. This combination of scaling factors leads to a complex interrelationship between transistor performance and process feature size. To first approximation, transistor performance improves linearly with decreasing feature size.

The fact that transistor count improves quadratically with a linear improvement in transistor performance is both the challenge and the opportunity that computer architects were created for! In the early days of microprocessors, the higher rate of improvement in density was used to quickly move from 4-bit, to 8-bit, to 16-bit, to 32-bit microprocessors. More recently, density improvements have supported the introduction of 64-bit microprocessors as well as many of the innovations in pipelining and caches, which we discuss in Chapters 3, 4, and 5.

Although transistors generally improve in performance with decreased feature size, wires in an integrated circuit do not. In particular, the signal delay for a wire increases in proportion to the product of its resistance and capacitance. Of course, as feature size shrinks wires get shorter, but the resistance and capacitance per unit length gets worse. This relationship is complex, since both resistance and capacitance depend on detailed aspects of the process, the geometry of a wire, the loading on a wire, and even the adjacency to other structures. There are occasional process enhancements, such as the introduction of copper, which provide one-time improvements in wire delay. In general, however, wire delay scales poorly compared to transistor performance, creating additional challenges for the designer. In the past few years, wire delay has become a major design limitation for large integrated circuits and is often more critical than transistor switching delay. Larger and larger fractions of the clock cycle have been consumed by the propagation delay of signals on wires. In 2001, the Pentium 4 broke new ground by allocating two stages of its 20+ stage pipeline just for propagating signals across the chip.

Power also provides challenges as devices are scaled. For modern CMOS microprocessors, the dominant energy consumption is in switching transistors. The energy required per transistor is proportional to the product of the load capacitance of the transistor, the frequency of switching, and the square of the voltage. As we move from one process to the next, the increase in the number of transistors switching and the frequency with which they switch, dominates the decrease in load capacitance and voltage, leading to an overall growth in power consumption. The first microprocessors consumed tenths of watts, while a Pentium 4 consumes between 60 and 85 watts, and a 2 GHz Pentium 4 will be close to 100 watts. The fastest workstation and server microprocessors in 2001 consume between 100 and 150 watts. Distributing the power, removing the heat, and prevent-

ing hot spots have become increasingly difficult challenges, and it is likely that power rather than raw transistor count will become the major limitation in the near future.

.

1.4 Cost, Price and their Trends

Although there are computer designs where costs tend to be less important—specifically supercomputers—cost-sensitive designs are of growing importance: more than half the PCs sold in 1999 were priced at less than $1,000, and the average price of a 32-bit microprocessor for an embedded application is in the tens of dollars. Indeed, in the past 15 years, the use of technology improvements to achieve lower cost, as well as increased performance, has been a major theme in the computer industry.

Textbooks often ignore the cost half of cost-performance because costs change, thereby dating books, and because the issues are subtle and differ across industry segments. Yet an understanding of cost and its factors is essential for designers to be able to make intelligent decisions about whether or not a new feature should be included in designs where cost is an issue. (Imagine architects designing skyscrapers without any information on costs of steel beams and concrete.)

This section focuses on cost and price, specifically on the relationship between price and cost: price is what you sell a finished good for, and cost is the amount spent to produce it, including overhead. We also discuss the major trends and factors that affect cost and how it changes over time. The Exercises and Examples use specific cost data that will change over time, though the basic determinants of cost are less time sensitive. This section will introduce you to these topics by discussing some of the major factors that influence cost of a computer design and how these factors are changing over time.

The Impact of Time, Volume, Commodification, and Packaging

The cost of a manufactured computer component decreases over time even without major improvements in the basic implementation technology. The underlying principle that drives costs down is the *learning curve*—manufacturing costs decrease over time. The learning curve itself is best measured by change in *yield*—the percentage of manufactured devices that survives the testing procedure. Whether it is a chip, a board, or a system, designs that have twice the yield will have basically half the cost.

Understanding how the learning curve will improve yield is key to projecting costs over the life of the product. As an example of the learning curve in action, the price per megabyte of DRAM drops over the long term by 40% per year. Since DRAMs tend to be priced in close relationship to cost–with the exception

of periods when there is a shortage–price and cost of DRAM track closely. In fact, there are some periods (for example early 2001) in which it appears that price is less than cost; of course, the manufacturers hope that such periods are both infrequent and short. Figure 1.5 plots the price of a new DRAM chip over its lifetime. Between the start of a project and the shipping of a product, say two years, the cost of a new DRAM drops by a factor of between five and ten in constant dollars. Since not all component costs change at the same rate, designs based on projected costs result in different cost/performance trade-offs than those using current costs. The caption of Figure 1.5 discusses some of the long-term trends in DRAM price. .

Microprocessor prices also drop over time, but because they are less standardized than DRAMs, the relationship between price and cost is more complex. In a period of significant competition, price tends to track cost closely, although microprocessor vendors probably rarely sell at a loss. Figure 1.6 shows processor price trends for the Pentium III.

Volume is a second key factor in determining cost. Increasing volumes affect cost in several ways. First, they decrease the time needed to get down the learning curve, which is partly proportional to the number of systems (or chips) manufactured. Second, volume decreases cost, since it increases purchasing and manufacturing efficiency. As a rule of thumb, some designers have estimated that cost decreases about 10% for each doubling of volume. Also, volume decreases the amount of development cost that must be amortized by each machine, thus allowing cost and selling price to be closer. We will return to the other factors influencing selling price shortly.

Commodities are products that are sold by multiple vendors in large volumes and are essentially identical. Virtually all the products sold on the shelves of grocery stores are commodities, as are standard DRAMs, disks, monitors, and keyboards. In the past 10 years, much of the low end of the computer business has become a commodity business focused on building IBM-compatible PCs. There are a variety of vendors that ship virtually identical products and are highly competitive. Of course, this competition decreases the gap between cost and selling price, but it also decreases cost. Reductions occur because a commodity market has both volume and a clear product definition, which allows multiple suppliers to compete in building components for the commodity product. As a result, the overall product cost is lower because of the competition among the suppliers of the components and the volume efficiencies the suppliers can achieve. This has led to the low-end of the computer business being able to achieve better price-performance than other sectors, and yielded greater growth at the low-end, albeit with very limited profits (as is typical in any commodity business).

Cost of an Integrated Circuit

Why would a computer architecture book have a section on integrated circuit costs? In an increasingly competitive computer marketplace where standard

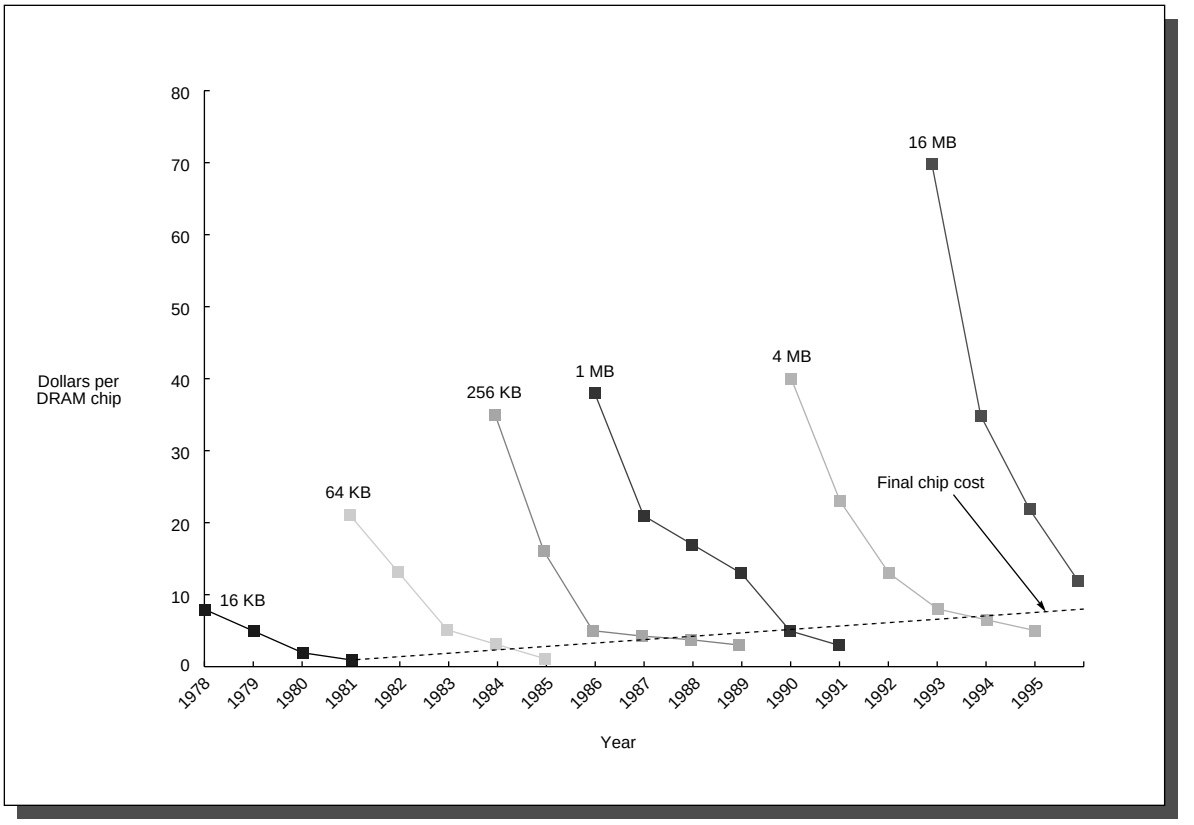


FIGURE 1.5 Prices of six generations of DRAMs (from 16Kb to 64 Mb) over time in 1977 dollars, showing the learning curve at work. A 1977 dollar is worth about $2.95 in 2001; more than half of this inflation occurred in the five-year period of 1977–82, during which the value changed to $1.59. The cost of a megabyte of memory has dropped *incredibly* during this period, from over $5000 in 1977 to about $0.35 in 2000, and an amazing $0.08 in 2001 (in 1977 dollars)! Each generation drops in constant dollar price by a factor of 10 to 30 over its lifetime. Starting in about 1996, an explosion of manufacturers has dramatically reduced margins and increased the rate at which prices fall, as well as the eventual final price for a DRAM. Periods when demand exceeded supply, such as 1987–88 and 1992–93, have led to temporary higher pricing, which shows up as a slowing in the rate of price decrease; more dramatic short-term fluctuations have been smoothed out. In late 2000 and through 2001, there has been tremendous oversupply leading to an accelerated price decrease, which is probably not sustainable.

- Add 64Mb data Change MB to Mb in labels and KB to Kb.
- Remove the final chip cost line and the label on it.
- Extend x-axis: change 1996 data point to $6.00; add to the 16Mb line: 1997: 3.78; 1998: $1.30
- Add a new line labeled 64Mb: 1999: $4.36; 2000: $2.78; 2001: $0.68

parts—disks, DRAMs, and so on—are becoming a significant portion of any system's cost, integrated circuit costs are becoming a greater portion of the cost that varies between machines, especially in the high-volume, cost-sensitive portion of the market. Thus computer designers must understand the costs of chips to understand the costs of current computers.

Although the costs of integrated circuits have dropped exponentially, the basic procedure of silicon manufacture is unchanged: A *wafer* is still tested and

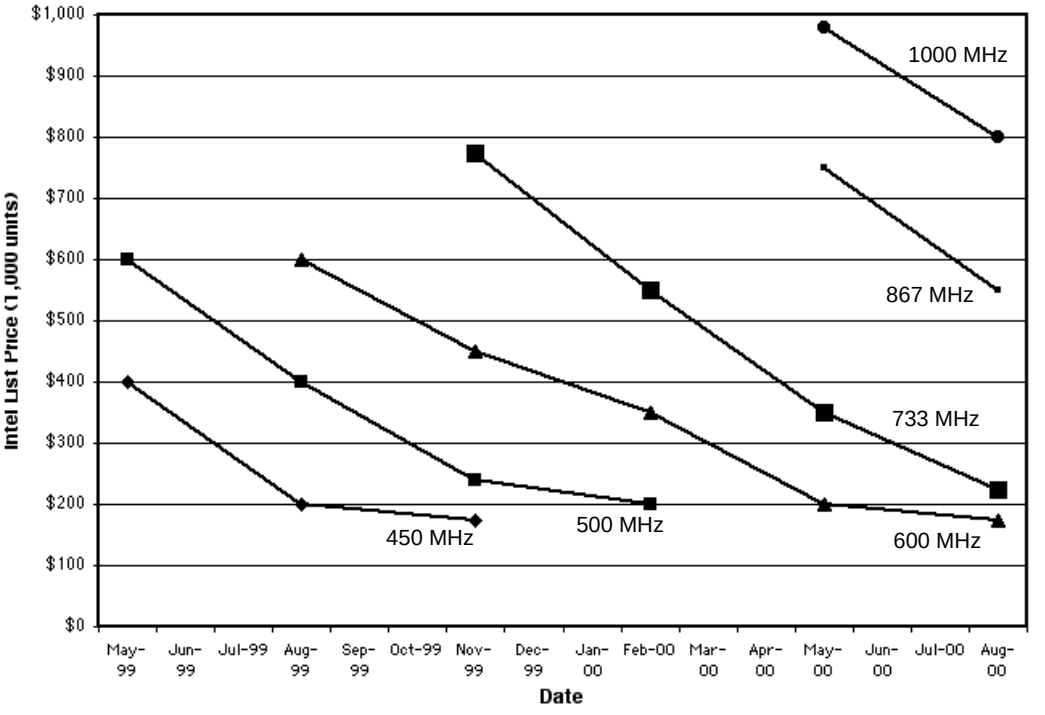


FIGURE 1.6 The price of an Intel Pentium III at a given frequency decreases over time as yield enhancements decrease the cost of good die and competition forces price reductions. Data courtesy of Microprocessor Report, May 2000 issue. The most recent introductions will continue to decrease until they reach similar prices to the lowest cost parts available today ($100-$200). Such price decreases assume a competitive environment where price decreases track cost decreases closely.

chopped into *dies* that are packaged (see Figures 1.7 and 1.8). Thus the cost of a packaged integrated circuit is

$$\text{Cost of integrated circuit} = \frac{\text{Cost of die} + \text{Cost of testing die} + \text{Cost of packaging and final test}}{\text{Final test yield}}$$

In this section, we focus on the cost of dies, summarizing the key issues in testing and packaging at the end. A longer discussion of the testing costs and packaging costs appears in the Exercises.

To learn how to predict the number of good chips per wafer requires first learning how many dies fit on a wafer and then learning how to predict the percentage of those that will work. From there it is simple to predict cost:

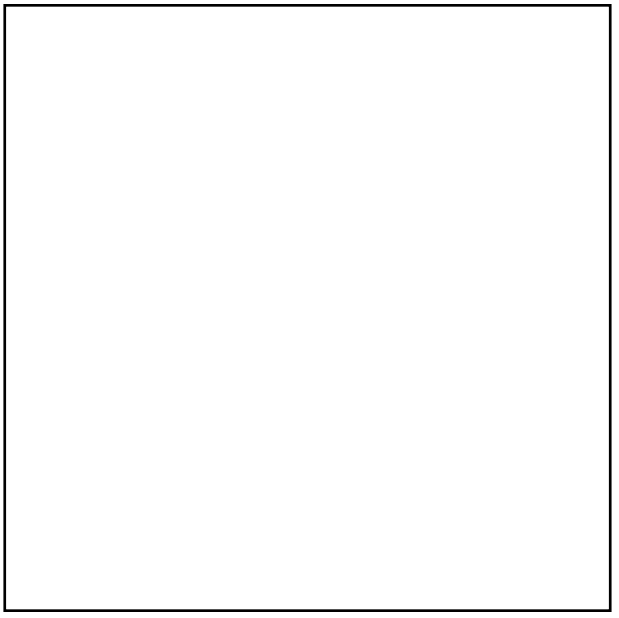


FIGURE 1.7 Photograph of an 12-inch wafer containing Intel Pentium 4 microprocessors. (Courtesy Intel.)

Get new photo!

$$\text{Cost of die} = \frac{\text{Cost of wafer}}{\text{Dies per wafer} \times \text{Die yield}}$$

The most interesting feature of this first term of the chip cost equation is its sensitivity to die size, shown below.

The number of dies per wafer is basically the area of the wafer divided by the area of the die. It can be more accurately estimated by

$$\text{Dies per wafer} = \frac{\pi \times (\text{Wafer diameter}/2)^2}{\text{Die area}} - \frac{\pi \times \text{Wafer diameter}}{\sqrt{2 \times \text{Die area}}}$$

The first term is the ratio of wafer area (πr^2) to die area. The second compensates for the "square peg in a round hole" problem—rectangular dies near the periphery of round wafers. Dividing the circumference (πd) by the diagonal of a square die is approximately the number of dies along the edge. For example, a wafer 30 cm (≈ 12 inch) in diameter produces $\pi \times 225 - (\pi \times 30 / 1.41) = 640$ 1-cm dies.

EXAMPLE Find the number of dies per 30-cm wafer for a die that is 0.7 cm on a side.

ANSWER The total die area is 0.49 cm^2. Thus

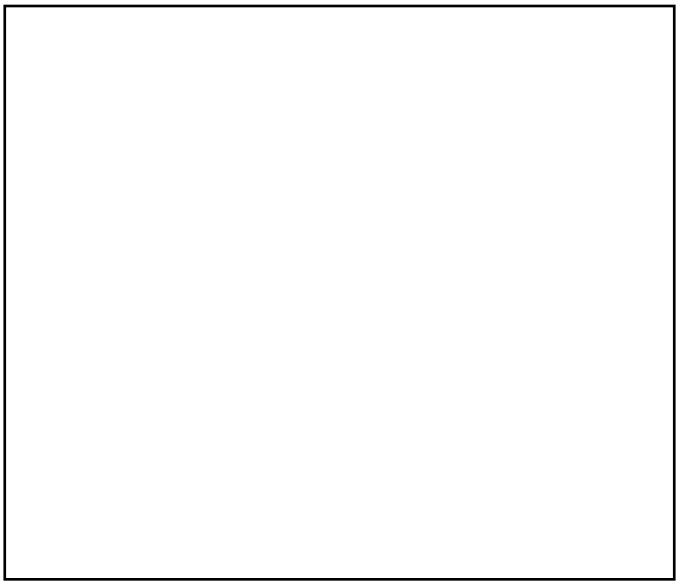


FIGURE 1.8 Photograph of an 12-inch wafer containing NEC MIPS 4122 processors.

Get new photo

$$\text{Dies per wafer} = \frac{\pi \times (30/2)^2}{0.49} - \frac{\pi \times 30}{\sqrt{2 \times 0.49}} = \frac{706.5}{0.49} - \frac{94.2}{0.99} = 1347$$

n

But this only gives the maximum number of dies per wafer. The critical question is, What is the fraction or percentage of good dies on a wafer number, or the *die yield*? A simple empirical model of integrated circuit yield, which assumes that defects are randomly distributed over the wafer and that yield is inversely proportional to the complexity of the fabrication process, leads to the following:

$$\text{Die yield} = \text{Wafer yield} \times \left(1 + \frac{\text{Defects per unit area} \times \text{Die area}}{\alpha}\right)^{-\alpha}$$

where *wafer yield* accounts for wafers that are completely bad and so need not be tested. For simplicity, we'll just assume the wafer yield is 100%. Defects per unit area is a measure of the random manufacturing defects that occur. In 2001, these values typically range between 0.4 and 0.8 per square centimeter, depending on the maturity of the process (recall the learning curve, mentioned earlier). Lastly,

α is a parameter that corresponds inversely to the number of masking levels, a measure of manufacturing complexity, critical to die yield. For today's multilevel metal CMOS processes, a good estimate is $\alpha = 4.0$.

EXAMPLE Find the die yield for dies that are 1 cm on a side and 0.7 cm on a side, assuming a defect density of 0.6 per cm^2.

ANSWER The total die areas are 1 cm^2 and 0.49 cm^2. For the larger die the yield is

$$\text{Die yield} = \left(1 + \frac{0.6 \times 1}{2.0}\right)^{-4} = 0.35$$

For the smaller die, it is

$$\text{Die yield} = \left(1 + \frac{0.6 \times 0.49}{2.0}\right)^{-4} = 0.58$$

n

The bottom line is the number of good dies per wafer, which comes from multiplying dies per wafer by die yield (which incorporates the effects of defects). The examples above predict 224 good 1-cm^2 dies from the 30-cm wafer and 781 good 0.49-cm^2 dies. Most 32-bit and 64-bit microprocessors in a modern 0.25µ technology fall between these two sizes, with some processors being as large as 2 cm^2 in the prototype process before a shrink. Low-end embedded 32-bit processors are sometimes as small as 0.25 cm^2, while processors used for embedded control (in printers, automobiles, etc.) are often less than 0.1 cm^2. Figure 1.34 on page 81 in the Exercises shows the die size and technology for several current microprocessors.

Given the tremendous price pressures on commodity products such as DRAM and SRAM, designers have included redundancy as a way to raise yield. For a number of years, DRAMs have regularly included some redundant memory cells, so that a certain number of flaws can be accomodated. Designers have used similar techniques in both standard SRAMs and in large SRAM arrays used for caches within microprocessors. Obviously, the presence of redundant entries can be used to significantly boost the yield.

Processing a 30-cm-diameter wafer in a leading-edge technology with 4-6 metal layers costs between $5000 and $6000 in 2001. Assuming a processed wafer cost of $5500, the cost of the 0.49-cm^2 die is around $7.04, while the cost per die of the 1-cm^2 die is about $24.55, or more than three times the cost for a die that is two times larger.

What should a computer designer remember about chip costs? The manufacturing process dictates the wafer cost, wafer yield, α, and defects per unit area, so the sole control of the designer is die area. Since α is around 4 for the advanced

processes in use today, die costs are proportional to the fifth (or higher) power of the die area:

$$\text{Cost of die} = f(\text{Die area}^5)$$

The computer designer affects die size, and hence cost, both by what functions are included on or excluded from the die and by the number of I/O pins.

Before we have a part that is ready for use in a computer, the die must be tested (to separate the good dies from the bad), packaged, and tested again after packaging. These steps all add significant costs. These processes and their contribution to cost are discussed and evaluated in Exercise 1.9.

The above analysis has focused on the variable costs of producing a functional die, which is appropriate for high volume integrated circuits. There is, however, one very important part of the fixed cost that can significantly impact the cost of an integrated circuit for low volumes (less than one million parts), namely the cost of a mask set. Each step in the integrated circuit process requires a separate mask. Thus, for modern high density fabrication processes with four to six metal layers, mask costs often exceed $1 million. Obviously, this large fixed cost affects the cost of prototyping and debugging runs and, for small volume production, can be a significant part of the production cost. Since mask costs are likely to continue to increase, designers may incorporate reconfigurable logic to enhance the flexibility of a part, or choose to use gate arrays (that have fewer custom mask levels) and thus, reduce the cost implications of masks.

Distribution of Cost in a System: An Example

To put the costs of silicon in perspective, Figure 1.9 shows the approximate cost breakdown for a $1,000 PC in 2001. Although the costs of some parts of this machine can be expected to drop over time, other components, such as the packaging and power supply, have little room for improvement. Furthermore, we can expect that future machines will have larger memories and disks, meaning that prices drop more slowly than the technology improvement.

Cost Versus Price—Why They Differ and By How Much

Costs of components may confine a designer's desires, but they are still far from representing what the customer must pay. But why should a computer architecture book contain pricing information? Cost goes through a number of changes before it becomes price, and the computer designer should understand how a design decision will affect the potential selling price. For example, changing cost by $1000 may change price by $3000 to $4000. Without understanding the relationship of cost to price the computer designer may not understand the impact on price of adding, deleting, or replacing components.

System	Subsystem	Fraction of total
Cabinet	Sheet metal, plastic	2%
	Power supply, fans	2%
	Cables, nuts, bolts	1%
	Shipping box, manuals	1%
	Subtotal	**6%**
Processor board	Processor	23%
	DRAM (128 MB)	5%
	Video card	5%
	Motherboard with basic I/O support, and networking	5%
	Subtotal	**38%**
I/O devices	Keyboard and mouse	3%
	Monitor	20%
	Hard disk (20 GB)	9%
	DVD drive	6%
	Subtotal	**37%**
Software	OS + Basic Office Suite	**20%**

FIGURE 1.9 Estimated distribution of costs of the components in a $1,000 PC in 2001. Notice that the largest single item is the CPU, closely followed by the monitor. (Interestingly, in 1995, the DRAM memory at about 1/3 of the total cost was the most expensive component! Since then, cost per MB has dropped by about a factor of 15!) Touma [1993] discusses computer system costs and pricing in more detail. These numbers are based on estimates of volume pricing for the various components.

The relationship between price and volume can increase the impact of changes in cost, especially at the low end of the market. Typically, fewer computers are sold as the price increases. Furthermore, as volume decreases, costs rise, leading to further increases in price. Thus, small changes in cost can have a larger than obvious impact. The relationship between cost and price is a complex one with entire books written on the subject. The purpose of this section is to give you a simple introduction to what factors determine price and typical ranges for these factors.

The categories that make up price can be shown either as a tax on cost or as a percentage of the price. We will look at the information both ways. These differences between price and cost also depend on where in the computer marketplace a company is selling. To show these differences, Figure 1.10 shows how the dif-

ference between cost of materials and list price is decomposed, with the price increasing from left to right as we add each type of overhead.

Direct costs refer to the costs directly related to making a product. These include labor costs, purchasing components, scrap (the leftover from yield), and warranty, which covers the costs of systems that fail at the customer's site during the warranty period. Direct cost typically adds 10% to 30% to component cost. Service or maintenance costs are not included because the customer typically pays those costs, although a warranty allowance may be included here or in gross margin, discussed next.

The next addition is called the *gross margin*, the company's overhead that cannot be billed directly to one product. This can be thought of as indirect cost. It includes the company's research and development (R&D), marketing, sales, manufacturing equipment maintenance, building rental, cost of financing, pretax profits, and taxes. When the component costs are added to the direct cost and gross margin, we reach the *average selling price*—ASP in the language of MBAs—the money that comes directly to the company for each product sold. The gross margin is typically 10% to 45% of the average selling price, depending on the uniqueness of the product. Manufacturers of low-end PCs have lower gross margins for several reasons. First, their R&D expenses are lower. Second, their cost of sales is lower, since they use indirect distribution (by mail, the Internet, phone order, or retail store) rather than salespeople. Third, because their products are less unique, competition is more intense, thus forcing lower prices and often lower profits, which in turn lead to a lower gross margin.

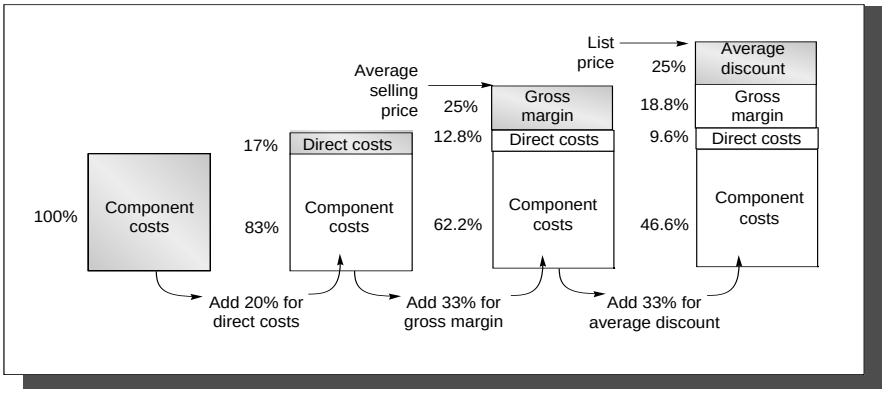


FIGURE 1.10 The components of price for a $1,000 PC. Each increase is shown along the bottom as a tax on the prior price. The percentages of the new price for all elements are shown on the left of each column.

List price and average selling price are not the same. One reason for this is that companies offer volume discounts, lowering the average selling price. As person-

al computers became commodity products, the retail mark-ups have dropped significantly, so list price and average selling price have closed.

As we said, pricing is sensitive to competition: A company may not be able to sell its product at a price that includes the desired gross margin. In the worst case, the price must be significantly reduced, lowering gross margin until profit becomes negative! A company striving for market share can reduce price and profit to increase the attractiveness of its products. If the volume grows sufficiently, costs can be reduced. Remember that these relationships are extremely complex and to understand them in depth would require an entire book, as opposed to one section in one chapter. For example, if a company cuts prices, but does not obtain a sufficient growth in product volume, the chief impact will be lower profits.

Many engineers are surprised to find that most companies spend only 4% (in the commodity PC business) to 12% (in the high-end server business) of their income on R&D, which includes all engineering (except for manufacturing and field engineering). This well-established percentage is reported in companies' annual reports and tabulated in national magazines, so this percentage is unlikely to change over time. In fact, experience has shown that computer companies with R&D percentages of 15-20% rarely prosper over the long term.

The information above suggests that a company uniformly applies fixed-overhead percentages to turn cost into price, and this is true for many companies. But another point of view is that R&D should be considered an investment. Thus an investment of 4% to 12% of income means that every $1 spent on R&D should lead to $8 to $25 in sales. This alternative point of view then suggests a different gross margin for each product depending on the number sold and the size of the investment.

Large, expensive machines generally cost more to develop—a machine costing 10 times as much to manufacture may cost many times as much to develop. Since large, expensive machines generally do not sell as well as small ones, the gross margin must be greater on the big machines for the company to maintain a profitable return on its investment. This investment model places large machines in double jeopardy—because there are fewer sold *and* they require larger R&D costs—and gives one explanation for a higher ratio of price to cost versus smaller machines.

The issue of cost and cost/performance is a complex one. There is no single target for computer designers. At one extreme, *high-performance design* spares no cost in achieving its goal. Supercomputers have traditionally fit into this category, but the market that only cares about performance has been the slowest growing portion of the computer market. At the other extreme is *low-cost design*, where performance is sacrificed to achieve lowest cost; some portions of the embedded market, for example, the market for cell phone microprocessors, behaves exactly like this. Between these extremes is *cost/performance design*, where the designer balances cost versus performance. Most of the PC market, the worksta-

tion market, and most of the server market (at least including both low-end and midrange servers) operate in this region. In the past 10 years, as computers have downsized, both low-cost design and cost/performance design have become increasingly important. This section has introduced some of the most important factors in determining cost; the next section deals with performance.

1.5 Measuring and Reporting Performance

When we say one computer is faster than another, what do we mean? The user of a desktop machine may say a computer is faster when a program runs in less time, while the computer center manager running a large server system may say a computer is faster when it completes more jobs in an hour. The computer user is interested in reducing *response time*—the time between the start and the completion of an event—also referred to as *execution time*. The manager of a large data processing center may be interested in increasing *throughput*—the total amount of work done in a given time.

In comparing design alternatives, we often want to relate the performance of two different machines, say X and Y. The phrase "X is faster than Y" is used here to mean that the response time or execution time is lower on X than on Y for the given task. In particular, "X is n times faster than Y" will mean

$$\frac{\text{Execution time}_{\text{Y}}}{\text{Execution time}_{\text{X}}} = n$$

Since execution time is the reciprocal of performance, the following relationship holds:

$$n = \frac{\text{Execution time}_{\text{Y}}}{\text{Execution time}_{\text{X}}} = \frac{\dfrac{1}{\text{Performance}_{\text{Y}}}}{\dfrac{1}{\text{Performance}_{\text{X}}}} = \frac{\text{Performance}_{\text{X}}}{\text{Performance}_{\text{Y}}}$$

The phrase "the throughput of X is 1.3 times higher than Y" signifies here that the number of tasks completed per unit time on machine X is 1.3 times the number completed on Y.

Because performance and execution time are reciprocals, increasing performance decreases execution time. To help avoid confusion between the terms *increasing* and *decreasing*, we usually say "improve performance" or "improve execution time" when we mean *increase* performance and *decrease* execution time.

Whether we are interested in throughput or response time, the key measurement is time: The computer that performs the same amount of work in the least time is the fastest. The difference is whether we measure one task (response time) or many tasks (throughput). Unfortunately, time is not always the metric quoted in comparing the performance of computers. A number of popular measures have been adopted in the quest for a easily understood, universal measure of computer

performance, with the result that a few innocent terms have been abducted from their well-defined environment and forced into a service for which they were never intended. The authors' position is that the only consistent and reliable measure of performance is the execution time of real programs, and that all proposed alternatives to time as the metric or to real programs as the items measured have eventually led to misleading claims or even mistakes in computer design. The dangers of a few popular alternatives are shown in *Fallacies and Pitfalls*, section 1.9.

Measuring Performance

Even execution time can be defined in different ways depending on what we count. The most straightforward definition of time is called *wall-clock time*, *response time*, or *elapsed time*, which is the latency to complete a task, including disk accesses, memory accesses, input/output activities, operating system overhead—everything. With multiprogramming the CPU works on another program while waiting for I/O and may not necessarily minimize the elapsed time of one program. Hence we need a term to take this activity into account. *CPU time* recognizes this distinction and means the time the CPU is computing, *not* including the time waiting for I/O or running other programs. (Clearly the response time seen by the user is the elapsed time of the program, not the CPU time.) CPU time can be further divided into the CPU time spent in the program, called *user CPU time*, and the CPU time spent in the operating system performing tasks requested by the program, called *system CPU time*.

These distinctions are reflected in the UNIX time command, which returns four measurements when applied to an executing program:

```
90.7u 12.9s 2:39 65%
```

User CPU time is 90.7 seconds, system CPU time is 12.9 seconds, elapsed time is 2 minutes and 39 seconds (159 seconds), and the percentage of elapsed time that is CPU time is (90.7 + 12.9)/159 or 65%. More than a third of the elapsed time in this example was spent waiting for I/O or running other programs or both. Many measurements ignore system CPU time because of the inaccuracy of operating systems' self-measurement (the above inaccurate measurement came from UNIX) and the inequity of including system CPU time when comparing performance between machines with differing system codes. On the other hand, system code on some machines is user code on others, and no program runs without some operating system running on the hardware, so a case can be made for using the sum of user CPU time and system CPU time.

In the present discussion, a distinction is maintained between performance based on elapsed time and that based on CPU time. The term *system performance* is used to refer to elapsed time on an *unloaded* system, while *CPU performance* refers to *user* CPU time on an unloaded system. We will focus on CPU performance in this chapter, though we do consider performance measurements based on elapsed time.

Choosing Programs to Evaluate Performance

Dhrystone does not use floating point. Typical programs don't …

Rick Richardson, *Clarification of Dhrystone* (1988)

This program is the result of extensive research to determine the instruction mix of a typical Fortran program. The results of this program on different machines should give a good indication of which machine performs better under a typical load of Fortran programs. The statements are purposely arranged to defeat optimizations by the compiler.

H. J. Curnow and B. A. Wichmann [1976], Comments in the Whetstone Benchmark

A computer user who runs the same programs day in and day out would be the perfect candidate to evaluate a new computer. To evaluate a new system the user would simply compare the execution time of her *workload*—the mixture of programs and operating system commands that users run on a machine. Few are in this happy situation, however. Most must rely on other methods to evaluate machines and often other evaluators, hoping that these methods will predict performance for their usage of the new machine. There are five levels of programs used in such circumstances, listed below in decreasing order of accuracy of prediction.

1. *Real applications*—Although the buyer may not know what fraction of time is spent on these programs, she knows that some users will run them to solve real problems. Examples are compilers for C, text-processing software like Word, and other applications like Photoshop. Real applications have input, output, and options that a user can select when running the program. There is one major downside to using real applications as benchmarks: Real applications often enocunter portability problems arising from dependences on the operating system or compiler. Enhancing portability often means modifying the source and sometimes eliminating some important activity, such as interactive graphics, which tends to be more system-dependent.

2. *Modified (or scripted) applications*—In many cases, real applications are used as the building block for a benchmark either with modifications to the application or with a script that acts as stimulus to the application. Applications are modified for two primary reasons: to enhance portability or to focus on one particular aspect of system performance. For example, to create a CPU-oriented benchmark, I/O may be removed or restructured to minimize its impact on execution time. Scripts are used to reproduce interactive behavior, which might occur on a desktop system, or to simulate complex multiuser interaction, which occurs in a server system.

3. *Kernels*—Several attempts have been made to extract small, key pieces from real programs and use them to evaluate performance. Livermore Loops and Linpack are the best known examples. Unlike real programs, no user would run kernel programs, for they exist solely to evaluate performance. Kernels are best used to isolate performance of individual features of a machine to explain the reasons for differences in performance of real programs.

4. *Toy benchmarks*—Toy benchmarks are typically between 10 and 100 lines of code and produce a result the user already knows before running the toy program. Programs like Sieve of Eratosthenes, Puzzle, and Quicksort are popular because they are small, easy to type, and run on almost any computer. The best use of such programs is beginning programming assignments.

5. *Synthetic benchmarks*—Similar in philosophy to kernels, synthetic benchmarks try to match the average frequency of operations and operands of a large set of programs. Whetstone and Dhrystone are the most popular synthetic benchmarks. A description of these benchmarks and some of their flaws appears in section 1.9 on page 59. No user runs synthetic benchmarks, because they don't compute anything a user could want. Synthetic benchmarks are, in fact, even further removed from reality than kernels because kernel code is extracted from real programs, while synthetic code is created artificially to match an average execution profile. Synthetic benchmarks are not even *pieces* of real programs, although kernels might be.

Because computer companies thrive or go bust depending on price/performance of their products relative to others in the marketplace, tremendous resources are available to improve performance of programs widely used in evaluating machines. Such pressures can skew hardware and software engineering efforts to add optimizations that improve performance of synthetic programs, toy programs, kernels, and even real programs. The advantage of the last of these is that adding such optimizations is more difficult in real programs, though not impossible. This fact has caused some benchmark providers to specify the rules under which compilers must operate, as we will see shortly.

Benchmark Suites

Recently, it has become popular to put together collections of benchmarks to try to measure the performance of processors with a variety of applications. Of course, such suites are only as good as the constituent individual benchmarks. Nonetheless, a key advantage of such suites is that the weakness of any one benchmark is lessened by the presence of the other benchmarks. This advantage is especially true if the methods used for summarizing the performance of the benchmark suite reflect the time to run the entire suite, as opposed to rewarding performance increases on programs that may be defeated by targeted optimizations. Later in this section, we discuss the strengths and weaknesses of different methods for summarizing performance.

One of the most successful attempts to create standardized benchmark application suites has been the SPEC (Standard Performance Evaluation Corporation), which had its roots in the late 1980s efforts to deliver better benchmarks for workstations. Just as the computer industry has evolved over time, so has the need for different benchmark suites, and there are now SPEC benchmarks to cover different application classes, as well as other suites based on the SPEC model.

Although we focus our discussion on the SPEC benchmarks in the many of the following sections, there are also a large set of benchmarks that have been developed for PCs running the Windows operating system. These cover a variety of different application environments, as Figure 1.11 shows.

Benchmark Name	Benchmark description
Business Winstone 99	Runs a script consisting of Netscape Navigator, and several office suite products (Microsoft, Corel, WordPerfect). The script simulates a user switching among and running different applications.
High-end Winstone 99	Also simulates multiple applications running simultaneously, but focuses on compute intensive applications such as Adobe Photoshop.
CC Winstone 99	Simulates multiple applications focused on content creation, such as Photoshop, Premiere, Navigator, and various audio editing programs.
Winbench 99	Runs a variety of scripts that test CPU performance, video system performance, disk performance using kernels focused on each subsystem.

FIGURE 1.11 A sample of some of the many PC benchmarks with the first four being scripts using real applications and the last being a mixture of kernels and synethetic benchmarks. These are all now maintained by Ziff Davis, a publisher of much of the literature in the PC space. Ziff Davis also provides independent testing service. For more information on these benchmarks, see: http://www.zdnet.com/etestinglabs/filters/benchmarks/.

Desktop Benchmarks

Desktop benchmarks divide into two broad classes: CPU intensive benchmarks and graphics intensive benchmarks (although many graphics benchmarks include intensive CPU activity). SPEC originally created a benchmark set focusing on CPU performance (initially called SPEC89), which has evolved into its fourth generation: SPEC CPU2000, which follows SPEC95, and SPEC92. (Figure 1.30 on page 64 discusses the evolution of the benchmarks.) SPEC CPU2000, summarized in Figure 1.12, consists of a set of eleven integer benchmarks (CINT2000) and fourteen floating point benchmarks (CFP2000). The SPEC benchmarks are real program modified for portability and to minimize the role of I/O in overall benchmark performance. The integer benchmarks vary from part of a C compiler to a VLSI place and route tool to a graphics application. The floating point benchmarks include code for quantum chromodynmics, finite element modeling, and fluid dynamics. The SPEC CPU suite is useful for CPU benchmarking for both desktop systems and single-processor servers. We will see data on many of these programs throughout this text.

In the next subsection, we show how a SPEC 2000 report describes the machine, compiler, and OS configuration. In section 1.9 we describe some of the pitfalls that have occurred in attempting to develop the SPEC benchmark suite, as well as the challenges in maintaining a useful and predictive benchmark suite.

Benchmark	Type	Source	Description
gzip	Integer	C	Compression using the Lempel-Ziv algorithm
vpr	Integer	C	FPGA circuit placement and routing
gcc	Integer	C	Consists of the GNU C compiler generating optimized machine code.
mcf	Integer	C	Combinatorial optimization of public transit scheduling.
crafty	Integer	C	Chess playing program.
parser	Integer	C	Syntactic English language parser
eon	Integer	C++	Graphics visualization using probabilistic ray tracing
perlmbk	Integer	C	Perl (an interpreted string processing language) with four input scripts
gap	Integer	C	A group theory application package
vortex	Integer	C	An object-oriented database system
bzip2	Integer	C	A block sorting compression algorithm.
twolf	Integer	C	Timberwolf: a simulated annealing algorithm for VLSI place and route
wupwise	FP	F77	Lattice gauge theory model of quantum chromodynamics.
swim	FP	F77	Solves shallow water equations using finite difference equations.
mgrid	FP	F77	Multigrid solver over 3-dimensional field.
apply	FP	F77	Parabolic and elliptic partial differential equation solver
mesa	FP	C	Three dimensional graphics library.
galgel	FP	F90	Computational fluid dynamics.
art	FP	C	Image recognition of a thermal image using neural networks
equake	FP	C	Simulation of seismic wave propagation.
facerec	FP	C	Face recognition using wavelets and graph matching.
ammp	FP	C	molecular dynamics simulation of a protein in water
lucas	FP	F90	Performs primality testing for Mersenne primes
fma3d	FP	F90	Finite element modeling of crash simulation
sixtrack	FP	F77	High energy physics accelerator design simulation.
apsi	FP	F77	A meteorological simulation of pollution distribution.

FIGURE 1.12 The programs in the SPECCPU2000 benchmark suites. The eleven integer programs (all in C, except one in C++) are used for the CINT2000 measurement, while the fourteen floating point programs (six in Fortran-77, five in C, and three in Fortran-90) are used for the CFP2000 measurement. See http://www.spec.org/osg/cpu2000/ for more on these benchmarks.

Although SPEC CPU2000 is aimed at CPU performance, two different types of graphics benchmarks were created by SPEC: SPECviewperf (see http://www.spec.org/gpc/opc.static/opcview.htm) is used for benchmarking systems supporting the OpenGL graphics library, while SPECapc (http://www.spec.org/gpc/apc.static/apcfaq.htm) consists of applications that make extensive use of graphics. SPECviewperf measures the 3D rendering performance of systems running under OpenGL using a 3-D model and a series of OpenGL calls that transform the model. SPECapc consists of runs of three large applications:

1. Pro/Engineer: a solid modeling application that does extensive 3-D rendering. The input script is a model of a photocopying machine consisting of 370,000 triangles.
2. SolidWorks 99: a 3-D CAD/CAM design tool running a series of five tests varying from I/O intensive to CPU intensive. The largetest input is a model of an assembly line consisting of 276,000 triangles.
3. Unigraphics V15: The benchmark is based on an aircraft model and covers a wide spectrum of Unigraphics functionality, including assembly, drafting, numeric control machining, solid modeling, and optimization. The inputs are all part of an aircraft design.

Server Benchmarks

Just as servers have multiple functions, so there are multiple types of benchmarks. The simplest benchmark is perhaps a CPU throughput oriented benchmark. SPEC CPU2000 uses the SPEC CPU benchmarks to construct a simple throughput benchmark where the processing rate of a multiprocessor can be measured by running multiple copies (usually as many as there are CPUs) of each SPEC CPU benchmark and converting the CPU time into a rate.This leads to a measurement called the SPECRate.

Other than SPECRate, most server applications and benchmarks have significant I/O activity arising from either disk or network traffic, including benchmarks for file server systems, for web servers, and for database and transaction processing systems. SPEC offers both a file server benchmark (SPECSFS) and a web server benchmark (SPECWeb). SPECSFS (see http://www.spec.org/osg/sfs93/) is a benchmark for measuring NFS (Network File System) performance using a script of file server requests; it tests the performance of the I/O system (both disk and network I/O) as well as the CPU. SPECSFS is a throughput oriented benchmark but with important response time requirements. (Chapter 6 discusses some file and I/O system benchmarks in detail.) SPECWEB (see http://www.spec.org/osg/web99/ for the 1999 version) is a web-server benchmark that simulates multiple clients requesting both static and dynamic pages from a server, as well as clients posting data to the server.

Transaction processing benchmarks measure the ability of a system to handle transactions, which consist of database accesses and updates. An airline reserva-

tion system or a bank ATM system are typical simple TP systems; more complex TP systems involve complex databases and decision making. In the mid 1980s, a group of concerned engineers formed the vendor-independent Transaction Processing Council (TPC) to try to create a set of realistic and fair benchmarks for transaction processing. The first TPC benchmark, TPC-A, was published in 1985 and has since been replaced and enhanced by four different benchmarks. TPC-C, initially created in 1992, simulates a complex query environment. TPC-H models ad-hoc decision support meaning that the queries are unrelated and knowledge of past queries cannot be used to optimize future queries; the result is that query execution times can be very long. TPC-R simulates a business decision support system where users run a standard set of queries. In TPC-R, pre-knowledge of the queries is taken for granted and the DBMS system can be optimized to run these queries. TPC-W web-based transaction benchmark that simulates the activities of a business oriented transactional web server. It exercises the database system as well as the underlying web server software. The TPC benchmarks are described at: http://www.tpc.org/.

All the TPC benchmarks measure performance in transactions per second. In addition, they include a response-time requirement, so that throughput performance is measured only when the response time limit is met.To model real-world systems, higher transaction rates are also associated with larger systems, both in terms of users and the data base that the transactions are applied to. Finally, the system cost for a benchmark system must also be included, allowing accurate comparisons of cost-performance.

Embedded Benchmarks

Benchmarks for embedded computing systems are in a far more nascent state than those for either desktop or server environments. In fact, many manufacturers quote Dhrystone performance, a benchmark that was criticized and given up by desktop systems more than 10 years ago! As mentioned earlier, the enormous variety in embedded applications, as well as differences in performance requirements (hard real-time, soft real-time, and overall cost-performance), make the use of a single set of benchmarks unrealistic. In practice, many designers of embedded systems devise benchmarks that reflect their application, either as kernels or as stand-alone versions of the entire application.

For those embedded applications that can be characterized well by kernel performance, the best standardized set of benchmarks appears to be a new benchmark set: the EDN Embedded Microprocessor Benchmark Consortium (or EEMBC–pronounced embassy). The EEMBC benchmarks fall into five classes: automotive/industrial, consumer, networking, office automation, and telecommunications. Figure 1.13 shows the five different application classes, which include 34 benchmarks.

Although many embedded applications are sensitive to the performance of small kernels, remember that often the overall performance of the entire application, which may be thousands of lines) is also critical. Thus, for many embedded

systems, the EMBCC benchmarks can only be used to partially assess performance.

Benchmark Type	# of this type	Example benchmarks
Automotive/industrial	16	6 microbenchmarks (arithmetic operations, pointer chasing, memory performance, matrix arithmetic, table lookup, bit manipulation), 5 automobile control benchmarks, and 5 filter or FFT benchmarks.
Consumer	5	5 multimedia benchmarks (JPEG compress/decompress, filtering, and RGB conversions).
Networking	3	Shortest path calculation, IP routing, and packet flow operations.
Office automation	4	Graphics and text benchmarks (Bezier curve calculation, dithering, image rotation, text processing).
Telecommunications	6	Filtering and DSP benchmarks (autocorrelation, FFT, decoder, and encoder)

FIGURE 1.13 The EEMBC benchmark suite, consisting of 34 kernels in five different classes. See www.eembc.org for more information on the benchmarks and for scores.

Reporting Performance Results

The guiding principle of reporting performance measurements should be *reproducibility*—list everything another experimenter would need to duplicate the results. A SPEC benchmark report requires a fairly complete description of the machine, the compiler flags, as well as the publication of both the baseline and optimized results. As an example, Figure 1.14 shows portions of the SPEC CINT2000 report for an Dell Precision Workstation 410. In addition to hardware, software, and baseline tuning parameter descriptions, a SPEC report contains the actual performance times, shown both in tabular form and as a graph. A TPC benchmark report is even more complete, since it must include results of a benchmarking audit and must also include cost information.

A system's software configuration can significantly affect the performance results for a benchmark. For example, operating syustems performance and support can be very important in server benchmarks. For this reason, these benchmarks are sometimes run in single-user mode to reduce overhead. Additionally, operating system enhancements are sometimes made to increase performance on the TPC benchmarks. Likewise, compiler technology can play a big role in CPU performance. The impact of compiler technology can be especially large when modification of the source is allowed (see the example with the EEMBC benchmarks on page 63) or when a benchmark is particularly suspectible to an optimization (see the example from SPEC described on 61). For these reasons it is important to describe exactly the software system being measured as well as whether any special nonstandard modifications have been made.

Another way to customize the software to improve the performance of a benchmark has been through the use of benchmark-specific flags; these flags often caused transformations that would be illegal on many programs or would

slow down performance on others. To restrict this process and increase the significance of the SPEC results, the SPEC organization created a *baseline performance* measurement in addition to the optimized performance measurement. Baseline performance restricts the vendor to one compiler and one set of flags for all the programs in the same language (C or FORTRAN). Figure 1.14 shows the parameters for the baseline performance; in section 1.8, *Fallacies and Pitfalls*, we'll see the tuning parameters for the optimized performance runs on this machine.

Hardware		Software	
Model number	Precision WorkStation 410	O/S and version	Windows NT 4.0
CPU	700 MHz, Pentium III	Compilers and version	Intel C/C++ Compiler 4.5
Number of CPUs	1	Other software	See below
Primary cache	16KBI+16KBD on chip	File system type	NTFS
Secondary cache	256KB(I+D) on chip	System state	Default
Other cache	None		
Memory	256 MB ECC PC100 SDRAM		
Disk subsystem	SCSI		
Other hardware	None		

SPEC CINT2000 base tuning parameters/notes/summary of changes:

+FDO: PASS1=-Qprof_gen PASS2=-Qprof_use

Base tuning: -QxK -Qipo_wp shlW32M.lib +FDO

shlW32M.lib is the SmartHeap library V5.0 from MicroQuill www.microquill.com

Portability flags:

176.gcc: -Dalloca=_alloca /F10000000 -Op

186.crafy: -DNT_i386

253.perlbmk: -DSPEC_CPU2000_NTOS -DPERLDLL /MT

254.gap: -DSYS_HAS_CALLOC_PROTO -DSYS_HAS_MALLOC_PROTO

FIGURE 1.14 The machine, software, and baseline tuning parameters for the CINT2000 base report on a Dell Precision WorkStation 410. This data is for the base CINT2000 report. The data is available online at: http://www.spec.org/osg/cpu2000/results/cpu2000.html.

In addition to the question of flags and optimization, another key question is whether source code modifications or hand-generated assembly language are allowed. There are four broad categories of apporoaches here:

1. No source code modifications are allowed. The SPEC benchmarks fall into this class, as do most of the standard PC benchmarks.
2. Source code modification are allowed, but are essentially difficult or impossible. Benchmarks like TPC-C rely on standard databases, such as Oracle or Microsoft's SQL server. Although these third party vendors are interested in the overall performance of their systems on important industry-standard bench-

marks, they are highly unlikely to make vendor- specific changes to enhance the performance for one particular customer.TPC-C also relies heavily on the operating system, which can be change, provided those changes become part of the production version.

3. Source modifications are allowed. Several supercomputer benchmark suites allow modification of the source code. For example, the NAS benchmarks specify the input and output and supply the source, but vendors are allowed to rewrite the source, including changing the algorithms, as long as the result is the same. EEMBC also allows source-level changes to its benchmarks and reports these as "optimized" measurements, versus "out-of-the-box" measurements that allow no changes.

4. Hand-coding is allowed. EEMBC allows assembly language coding of its benchmarks. The small size of its kernels makes this approach attractive, although in practice with larger embedded applications it is unlikely to be used, except for small loops.Figure 1.31 on page 65 shows the significant benefits from handcoding on several different processors.

The key issue that benchmark designers face in deciding to allow modification of the source is whether such modifications will reflect real practice and provide useful insight to users, or whether such modifications simply reduce the accuracy of the benchmarks as predictors of real performance.

Comparing and Summarizing Performance

Comparing performance of computers is rarely a dull event, especially when the designers are involved. Charges and countercharges fly across the Internet; one is accused of underhanded tactics and the other of misleading statements. Since careers sometimes depend on the results of such performance comparisons, it is understandable that the truth is occasionally stretched. But more frequently discrepancies can be explained by differing assumptions or lack of information.

We would like to think that if we could just agree on the programs, the experimental environments, and the definition of *faster*, then misunderstandings would be avoided, leaving the networks free for scholarly discourse. Unfortunately, that's not the reality. Once we agree on the basics, battles are then fought over what is the fair way to summarize relative performance of a collection of programs. For example, two articles on summarizing performance in the same journal took opposing points of view. Figure 1.15, taken from one of the articles, is an example of the confusion that can arise.

Using our definition of faster than, the following statements hold:

A is 10 times faster than B for program P1.

B is 10 times faster than A for program P2.

A is 20 times faster than C for program P1.

	Computer A	Computer B	Computer C
Program P1 (secs)	1	10	20
Program P2 (secs)	1000	100	20
Total time (secs)	1001	110	40

FIGURE 1.15 Execution times of two programs on three machines. Data from Figure I of Smith [1988].

C is 50 times faster than A for program P2.

B is 2 times faster than C for program P1.

C is 5 times faster than B for program P2.

Taken individually, any one of these statements may be of use. Collectively, however, they present a confusing picture—the relative performance of computers A, B, and C is unclear.

Total Execution Time: A Consistent Summary Measure

The simplest approach to summarizing relative performance is to use total execution time of the two programs. Thus

B is 9.1 times faster than A for programs P1 and P2.

C is 25 times faster than A for programs P1 and P2.

C is 2.75 times faster than B for programs P1 and P2.

This summary tracks execution time, our final measure of performance. If the workload consisted of running programs P1 and P2 an equal number of times, the statements above would predict the relative execution times for the workload on each machine.

An average of the execution times that tracks total execution time is the *arithmetic mean:*

$$\frac{1}{n}\sum_{i=1}^{n}\text{Time}_i$$

where Time_i is the execution time for the ith program of a total of n in the workload.

Weighted Execution Time

The question arises: What is the proper mixture of programs for the workload? Are programs P1 and P2 in fact run equally in the workload as assumed by the arithmetic mean? If not, then there are two approaches that have been tried for summarizing performance. The first approach when given an unequal mix of programs in the workload is to assign a weighting factor w_i to each program to indi-

cate the relative frequency of the program in that workload. If, for example, 20% of the tasks in the workload were program P1 and 80% of the tasks in the workload were program P2, then the weighting factors would be 0.2 and 0.8. (Weighting factors add up to 1.) By summing the products of weighting factors and execution times, a clear picture of performance of the workload is obtained. This is called the *weighted arithmetic mean*:

$$\sum_{i=1}^{n} \text{Weight}_i \times \text{Time}_i$$

where Weight_i is the frequency of the ith program in the workload and Time_i is the execution time of that program. Figure 1.16 shows the data from Figure 1.15 with three different weightings, each proportional to the execution time of a workload with a given mix.

	Programs			Weightings		
	A	**B**	**C**	**W(1)**	**W(2)**	**W(3)**
Program P1 (secs)	1.00	10.00	20.00	0.50	0.909	0.999
Program P2 (secs)	1000.00	100.00	20.00	0.50	0.091	0.001

Arithmetic mean:W(1)	500.50	55.00	20.00
Arithmetic mean:W(2)	91.91	18.19	20.00
Arithmetic mean:W(3)	2.00	10.09	20.00

FIGURE 1.16 Weighted arithmetic mean execution times for three machines (A, B, C) and two programs (P1 and P2) using three weightings (W1, W2, W3). The top table contains the original execution time measurements and the weighting factors, while the bottom table shows the resulting weighted arithmetic means for each weighting. W(1) equally weights the programs, resulting in a mean (row 3) that is the same as the unweighted arithmetic mean. W(2) makes the mix of programs inversely proportional to the execution times on machine B; row 4 shows the arithmetic mean for that weighting. W(3) weights the programs in inverse proportion to the execution times of the two programs on machine A; the arithmetic mean with this weighting is given in the last row. The net effect of the second and third weightings is to "normalize" the weightings to the execution times of programs running on that machine, so that the running time will be spent evenly between each program for that machine. For a set of n programs each taking Time_i on one machine, the equal-time weightings on that machine are $w_i = \dfrac{1}{\text{Time}_i \times \sum_{j=1}^{n} \left(\frac{1}{\text{Time}_j}\right)}$.

Normalized Execution Time and the Pros and Cons of Geometric Means

A second approach to unequal mixture of programs in the workload is to normalize execution times to a reference machine and then take the average of the normalized execution times. This is the approach used by the SPEC benchmarks,

where a base time on a SPARCstation is used for reference. This measurement gives a warm fuzzy feeling, because it suggests that performance of new programs can be predicted by simply multiplying this number times its performance on the reference machine.

Average normalized execution time can be expressed as either an arithmetic or *geometric* mean. The formula for the geometric mean is

$$\sqrt[n]{\prod_{i=1}^{n} \text{Execution time ratio}_i}$$

where Execution time ratio$_i$ is the execution time, normalized to the reference machine, for the *i*th program of a total of *n* in the workload. Geometric means also have a nice property for two samples X_i and Y_i:

$$\frac{\text{Geometric mean}(X_i)}{\text{Geometric mean}(Y_i)} = \text{Geometric mean}\left(\frac{X_i}{Y_i}\right)$$

As a result, taking either the ratio of the means or the mean of the ratios yields the same result. In contrast to arithmetic means, geometric means of normalized execution times are consistent no matter which machine is the reference. Hence, the arithmetic mean should *not* be used to average normalized execution times. Figure 1.17 shows some variations using both arithmetic and geometric means of normalized times.

Because the weightings in weighted arithmetic means are set proportionate to execution times on a given machine, as in Figure 1.16, they are influenced not only by frequency of use in the workload, but also by the peculiarities of a particular machine and the size of program input. The geometric mean of normalized execution times, on the other hand, is independent of the running times of the individual programs, and it doesn't matter which machine is used to normalize. If a situation arose in comparative performance evaluation where the programs were fixed but the inputs were not, then competitors could rig the results of weighted arithmetic means by making their best performing benchmark have the largest input and therefore dominate execution time. In such a situation the geometric mean would be less misleading than the arithmetic mean.

	Normalized to A			Normalized to B			Normalized to C		
	A	B	C	A	B	C	A	B	C
Program P1	1.0	10.0	20.0	0.1	1.0	2.0	0.05	0.5	1.0
Program P2	1.0	0.1	0.02	10.0	1.0	0.2	50.0	5.0	1.0
Arithmetic mean	1.0	5.05	10.01	5.05	1.0	1.1	25.03	2.75	1.0
Geometric mean	1.0	1.0	0.63	1.0	1.0	0.63	1.58	1.58	1.0
Total time	1.0	0.11	0.04	9.1	1.0	0.36	25.03	2.75	1.0

FIGURE 1.17 Execution times from Figure 1.15 normalized to each machine. The arithmetic mean performance varies depending on which is the reference machine—in column 2, B's execution time is five times longer than A's, although the reverse is true in column 4. In column 3, C is slowest, but in column 9, C is fastest. The geometric means are consistent independent of normalization—A and B have the same performance, and the execution time of C is 0.63 of A or B (1/1.58 is 0.63). Unfortunately, the total execution time of A is 10 times longer than that of B, and B in turn is about 3 times longer than C. As a point of interest, the relationship between the means of the same set of numbers is always harmonic mean ≤ geometric mean ≤ arithmetic mean.

The strong drawback to geometric means of normalized execution times is that they violate our fundamental principle of performance measurement—they do not predict execution time. The geometric means from Figure 1.17 suggest that for programs P1 and P2 the performance of machines A and B is the same, yet this would only be true for a workload that ran program P1 100 times for every occurrence of program P2 (see Figure 1.16 on page 37). The total execution time for such a workload suggests that machines A and B are about 50% faster than machine C, in contrast to the geometric mean, which says machine C is faster than A and B! In general there is *no workload* for three or more machines that will match the performance predicted by the geometric means of normalized execution times. Our original reason for examining geometric means of normalized performance was to avoid giving equal emphasis to the programs in our workload, but is this solution an improvement?

An additional drawback of using geometric mean as a method for summarizing performance for a benchmark suite (as SPEC CPU2000 does) is that it encourages hardware and software designers to focus their attention on the benchmarks where performance is easiest to improve rather than on the benchmarks that are slowest. For example, if some hardware or software improvement can cut the running time for a benchmark from 2 seconds to 1, the geometric mean will reward those designers with the same overall mark that it would give to designers that improve the running time on another benchmark in the suite from 10,000 seconds to 5000 seconds. Of course, everyone interested in running the second program thinks of the second batch of designers as their heroes and the first group as useless. Small programs are often easier to "crack," obtaining a large but unrepresentative performance improvement, and the use of geometric mean rewards such behavior more than a measure that reflects total running time.

The ideal solution is to measure a real workload and weight the programs according to their frequency of execution. If this can't be done, then normalizing so that equal time is spent on each program on some machine at least makes the rel-

ative weightings explicit and will predict execution time of a workload with that mix. The problem above of unspecified inputs is best solved by specifying the inputs when comparing performance. If results must be normalized to a specific machine, first summarize performance with the proper weighted measure and then do the normalizing.

Lastly, we must remember that any summary measure necessarily loses information, especially when the measurements may vary widely. Thus, it is important both to ensure that the results of individual benchmarks, as well as the summary number, are available. Furthermore, the summary number should be used with caution, since the summary–as opposed to a subset of the individual scores–may be the best indicator of performance for a customer's applications.

1.6 Quantitative Principles of Computer Design

Now that we have seen how to define, measure, and summarize performance, we can explore some of the guidelines and principles that are useful in design and analysis of computers. In particular, this section introduces some important observations about designing for performance and cost/performance, as well as two equations that we can use to evaluate design alternatives.

Make the Common Case Fast

Perhaps the most important and pervasive principle of computer design is to make the common case fast: In making a design trade-off, favor the frequent case over the infrequent case. This principle also applies when determining how to spend resources, since the impact on making some occurrence faster is higher if the occurrence is frequent. Improving the frequent event, rather than the rare event, will obviously help performance, too. In addition, the frequent case is often simpler and can be done faster than the infrequent case. For example, when adding two numbers in the CPU, we can expect overflow to be a rare circumstance and can therefore improve performance by optimizing the more common case of no overflow. This may slow down the case when overflow occurs, but if that is rare, then overall performance will be improved by optimizing for the normal case.

We will see many cases of this principle throughout this text. In applying this simple principle, we have to decide what the frequent case is and how much performance can be improved by making that case faster. A fundamental law, called *Amdahl's Law,* can be used to quantify this principle.

Amdahl's Law

The performance gain that can be obtained by improving some portion of a computer can be calculated using Amdahl's Law. Amdahl's Law states that the performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used.

Amdahl's Law defines the *speedup* that can be gained by using a particular feature. What is speedup? Suppose that we can make an enhancement to a machine that will improve performance when it is used. Speedup is the ratio

$$\text{Speedup} = \frac{\text{Performance for entire task using the enhancement when possible}}{\text{Performance for entire task without using the enhancement}}$$

Alternatively,

$$\text{Speedup} = \frac{\text{Execution time for entire task without using the enhancement}}{\text{Execution time for entire task using the enhancement when possible}}$$

Speedup tells us how much faster a task will run using the machine with the enhancement as opposed to the original machine.

Amdahl's Law gives us a quick way to find the speedup from some enhancement, which depends on two factors:

1. *The fraction of the computation time in the original machine that can be converted to take advantage of the enhancement*—For example, if 20 seconds of the execution time of a program that takes 60 seconds in total can use an enhancement, the fraction is 20/60. This value, which we will call $\text{Fraction}_{\text{enhanced}}$, is always less than or equal to 1.
2. *The improvement gained by the enhanced execution mode; that is, how much faster the task would run if the enhanced mode were used for the entire program*—This value is the time of the original mode over the time of the enhanced mode: If the enhanced mode takes 2 seconds for some portion of the program that can completely use the mode, while the original mode took 5 seconds for the same portion, the improvement is 5/2. We will call this value, which is always greater than 1, $\text{Speedup}_{\text{enhanced}}$.

The execution time using the original machine with the enhanced mode will be the time spent using the unenhanced portion of the machine plus the time spent using the enhancement:

$$\text{Execution time}_{\text{new}} = \text{Execution time}_{\text{old}} \times \left((1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} \right)$$

The overall speedup is the ratio of the execution times:

$$\text{Speedup}_{\text{overall}} = \frac{\text{Execution time}_{\text{old}}}{\text{Execution time}_{\text{new}}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

EXAMPLE Suppose that we are considering an enhancement to the processor of a server system used for web serving. The new CPU is 10 times faster on computation in the web serving application than the original processor. Assuming that the original CPU is busy with computation 40% of the time

and is waiting for I/O 60% of the time, what is the overall speedup gained by incorporating the enhancement?

ANSWER

$$\text{Fraction}_{\text{enhanced}} = 0.4$$

$$\text{Speedup}_{\text{enhanced}} = 10$$

$$\text{Speedup}_{\text{overall}} = \frac{1}{0.6 + \frac{0.4}{10}} = \frac{1}{0.64} \approx 1.56$$

n

Amdahl's Law expresses the law of diminishing returns: The incremental improvement in speedup gained by an additional improvement in the performance of just a portion of the computation diminishes as improvements are added. An important corollary of Amdahl's Law is that if an enhancement is only usable for a fraction of a task, we can't speed up the task by more than the reciprocal of 1 minus that fraction.

A common mistake in applying Amdahl's Law is to confuse "fraction of time converted to use an enhancement" and "fraction of time after enhancement is in use." If, instead of measuring the time that we *could use* the enhancement in a computation, we measure the time *after* the enhancement is in use, the results will be incorrect! (Try Exercise 1.2 to see how wrong.)

Amdahl's Law can serve as a guide to how much an enhancement will improve performance and how to distribute resources to improve cost/performance. The goal, clearly, is to spend resources proportional to where time is spent. Amdahl's Law is particularly useful for comparing the overall system performance of two alternatives, but it can also be applied to compare two CPU design alternatives, as the following Example shows.

EXAMPLE

A common transformation required in graphics engines is square root. Implementations of floating-point (FP) square root vary significantly in performance, especially among processor designed for graphics. Suppose FP square root (FPSQR) is responsible for 20% of the execution time of a critical graphics benchmark. One proposal is to enhance the FPSQR hardware and speed up this operation by a factor of 10. The other alternative is just to try to make all FP instructions in the graphics processor run faster by a factor of 1.6; FP instructions are responsible for a total of 50% of the execution time for the application. The design team believes that they can make all FP instructions run 1.6 times faster with the same effort as required for the fast square root. Compare these two design al-

ternatives.

ANSWER We can compare these two alternatives by comparing the speedups:

$$\text{Speedup}_{\text{FPSQR}} = \frac{1}{(1 - 0.2) + \frac{0.2}{10}} = \frac{1}{0.82} = 1.22$$

$$\text{Speedup}_{\text{FP}} = \frac{1}{(1 - 0.5) + \frac{0.5}{1.6}} = \frac{1}{0.8125} = 1.23$$

Improving the performance of the FP operations overall is slightly better because of the higher frequency. n

In the above example, we needed to know the time consumed by the new and improved FP operations; often it is difficult to measure these times directly. In the next section, we will see another way of doing such comparisons based on the use of an equation that decomposes the CPU execution time into three separate components. If we know how an alternative affects these three components, we can determine its overall performance effect. Furthermore, it is often possible to build simulators that measure these components before the hardware is actually designed.

The CPU Performance Equation

Essentially all computers are constructed using a clock running at a constant rate. These discrete time events are called *ticks*, *clock ticks*, *clock periods*, *clocks*, *cycles*, or *clock cycles*. Computer designers refer to the time of a clock period by its duration (e.g., 1 ns) or by its rate (e.g., 1 GHz). CPU time for a program can then be expressed two ways:

$$\text{CPU time} = \text{CPU clock cycles for a program} \times \text{Clock cycle time}$$

or

$$\text{CPU time} = \frac{\text{CPU clock cycles for a program}}{\text{Clock rate}}$$

In addition to the number of clock cycles needed to execute a program, we can also count the number of instructions executed—the *instruction path length* or *instruction count* (IC). If we know the number of clock cycles and the instruction count we can calculate the average number of *clock cycles per instruction* (CPI). Because it is easier to work with and because we will deal with simple processors

in this chapter, we use CPI. Designers sometimes also use *Instructions per Clock* or IPC, which is the inverse of CPI.

CPI is computed as:

$$\text{CPI} = \frac{\text{CPU clock cycles for a program}}{\text{Instruction Count}}$$

This CPU figure of merit provides insight into different styles of instruction sets and implementations, and we will use it extensively in the next four chapters.

By transposing instruction count in the above formula, clock cycles can be defined as $\text{IC} \times \text{CPI}$. This allows us to use CPI in the execution time formula:

$$\text{CPU time} = \text{Instruction Count} \times \text{Clock cycle time} \times \text{Cycles per Instruction}$$

or

$$\text{CPU time} = \frac{\text{Instruction Count} \times \text{Clock cycle time}}{\text{Clock rate}}$$

Expanding the first formula into the units of measurement and inverting the clock rate shows how the pieces fit together:

$$\frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}} = \frac{\text{Seconds}}{\text{Program}} = \text{CPU time}$$

As this formula demonstrates, CPU performance is dependent upon three characteristics: clock cycle (or rate), clock cycles per instruction, and instruction count. Furthermore, CPU time is *equally* dependent on these three characteristics: A 10% improvement in any one of them leads to a 10% improvement in CPU time.

Unfortunately, it is difficult to change one parameter in complete isolation from others because the basic technologies involved in changing each characteristic are interdependent:

- *Clock cycle time*—Hardware technology and organization
- *CPI*—Organization and instruction set architecture
- *Instruction count*—Instruction set architecture and compiler technology

Luckily, many potential performance improvement techniques primarily improve one component of CPU performance with small or predictable impacts on the other two.

Sometimes it is useful in designing the CPU to calculate the number of total CPU clock cycles as

$$\text{CPU clock cycles} = \sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i$$

where IC_i represents number of times instruction i is executed in a program and CPI_i represents the average number of instructions per clock for instruction i. This form can be used to express CPU time as

$$\text{CPU time} = \left(\sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i \right) \times \text{Clock cycle time}$$

and overall CPI as:

$$\text{CPI} = \frac{\sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i}{\text{Instruction count}} = \sum_{i=1}^{n} \frac{\text{IC}_i}{\text{Instruction count}} \times \text{CPI}_i$$

The latter form of the CPI calculation uses each individual CPI_i and the fraction of occurrences of that instruction in a program (i.e., $\text{IC}_i \div \text{Instruction count}$). CPI_i should be measured and not just calculated from a table in the back of a reference manual since it must include pipeline effects, cache misses, and any other memory system inefficiencies.

Consider our earlier example, here modified to use measurements of the frequency of the instructions and of the instruction CPI values, which, in practice, is obtained by simulation or by hardware instrumentation.

EXAMPLE Suppose we have made the following measurements:

Frequency of FP operations (other than FPSQR) = 25%
Average CPI of FP operations = 4.0
Average CPI of other instructions = 1.33
Frequency of FPSQR= 2%
CPI of FPSQR = 20

Assume that the two design alternatives are to decrease the CPI of FPSQR to 2 or to decrease the average CPI of all FP operations to 2.5. Compare these two design alternatives using the CPU performance equation.

ANSWER First, observe that only the CPI changes; the clock rate and instruction count remain identical. We start by finding the original CPI with neither enhancement:

$$\begin{aligned} \text{CPI}_{\text{original}} &= \sum_{i=1}^{n} \text{CPI}_i \times \left(\frac{\text{IC}_i}{\text{Instruction count}} \right) \\ &= (4 \times 25\%) + (1.33 \times 75\%) = 2.0 \end{aligned}$$

We can compute the CPI for the enhanced FPSQR by subtracting the cycles saved from the original CPI:

$$\text{CPI}_{\text{with new FPSQR}} = \text{CPI}_{\text{original}} - 2\% \times (\text{CPI}_{\text{old FPSQR}} - \text{CPI}_{\text{of new FPSQR only}})$$
$$= 2.0 - 2\% \times (20 - 2) = 1.64$$

We can compute the CPI for the enhancement of all FP instructions the same way or by summing the FP and non-FP CPIs. Using the latter gives us

$$\text{CPI}_{\text{new FP}} = (75\% \times 1.33) + (25\% \times 2.5) = 1.625$$

Since the CPI of the overall FP enhancement is slightly lower, its performance will be marginally better. Specifically, the speedup for the overall FP enhancement is

$$\text{Speedup}_{\text{new FP}} = \frac{\text{CPU time}_{\text{original}}}{\text{CPU time}_{\text{new FP}}} = \frac{\text{IC} \times \text{Clock cycle} \times \text{CPI}_{\text{original}}}{\text{IC} \times \text{Clock cycle} \times \text{CPI}_{\text{new FP}}}$$
$$= \frac{\text{CPI}_{\text{original}}}{\text{CPI}_{\text{new FP}}} = \frac{2.00}{1.625} = 1.23$$

Happily, this is the same speedup we obtained using Amdahl's Law on page 42. It is often possible to measure the constituent parts of the CPU performance equation. This is a key advantage for using the CPU performance equation versus Amdahl's Law in the above example. In particular, it may be difficult to measure things such as the fraction of execution time for which a set of instructions is responsible. In practice this would probably be computed by summing the product of the instruction count and the CPI for each of the instructions in the set. Since the starting point is often individual instruction count and CPI measurements, the CPU performance equation is incredibly useful. n

Measuring and Modeling the Components of the CPU Performance Equation

To use the CPU performance equation as a design tool, we need to be able to measure the various factors. For an existing processor, it is easy to obtain the execution time by measurement, and the clock speed is known. The challenge lies in discovering the instruction count or the CPI. Most newer processors include counters for both instructions executed and for clock cycles. By periodically monitoring these counters, it is also possible to attach execution time and instruction count to segments of the code, which can be helpful to programmers trying to understand and tune the performance of an application. Often, a designer or programmer will want to understand performance at a more fine-grained level than what is available from the hardware counters. For example, they may want to know why the CPI is what it is. In such cases, simulation techniques like those used for processors that are being designed are used.

There are three general classes of simulation techniques that are used. In general, the more sophisticated techniques yield more accuracy, particularly for more recent architectures, at the cost of longer execution time The first and simplest technique, and hence the least costly, is profile-based, static modeling. In this technique a dynamic execution profile of the program, which indicates how often each instruction is executed, is obtained by one of three methods:

1. By using hardware counters on the processor, which are periodically saved. This technique often gives an approximate profile, but one that is within a few percent of exact.
2. By using instrumented execution, in which instrumentation code is compiled into the program. This code is used to increment counters, yielding an exact profile. (This technique can also be used to create a trace of memory address that are accessed, which is useful for other simulation techniques.)
3. By interpreting the program at the instruction set level, compiling instruction counts in the process.

Once the profile is obtained, it is used to analyze the program in a static fashion by looking at the code. Obviously with the profile, the total instruction count is easy to obtain. It is also easy to get a detailed dynamic instruction mix telling what types of instructions were executed with what frequency. Finally, for simple processors, it is possible to compute an approximation to the CPI. This approximation is computed by modeling and analyzing the execution of each basic block (or straightline code segment) and then computing an overall estimate of CPI or total compute cycles by multiplying the estimate for each basic block by the number of times it is executed. Although this simple model ignores memory behavior and has severe limits for modeling complex pipelines, it is a reasonable and very fast technique for modeling the performance of short, integer pipelines, ignoring the memory system behavior.

Trace-driven simulation is a more sophisticated technique for modeling performance and is particularly useful for modeling memory system performance. In trace-driven simulation, a trace of the memory references executed is created, usually either by simulation or by instrumented execution. The trace includes what instructions were executed (given by the instruction address), as well as the data addresses accessed.

Trace-driven simulation can be used in several different ways. The most common use is to model memory system performance, which can be done by simulating the memory system, including the caches and any memory management hardware using the address trace. A trace-driven simulation of the memory system can be combined with a static analysis of pipeline performance to obtain a reasonably accurate performance model for simple pipelined processors. For more complex pipelines, the trace data can be used to perform a more detailed analysis of the pipeline performance by simulation of the processor pipeline.

Since the trace data allows a simulation of the exact ordering of instructions, higher accuracy can be achieved than with a static approach. Trace-driven simulation typically isolates the simulation of any pipeline behavior from the memory system. In particular, it assumes that the trace is completely independent of the memory system behavior. As we will see in Chapters 3 and 5, this is not the case for the most advanced processors–a third technique is needed.

The third technique, which is the most accurate and most costly, is execution-driven simulation. In execution-driven simulation a detailed simulation of the memory system and the processor pipeline are done simultaneously. This allows the exact modeling of the interaction between the two, which is critical as we will see in Chapters 3 and 5.

There are many variations on these three basic techniques. We will see examples of these tools in later chapters and use various versions of them in the exercises.

Locality of Reference

Although Amdahl's Law is a theorem that applies to any system, other important fundamental observations come from properties of programs. The most important program property that we regularly exploit is *locality of reference*: Programs tend to reuse data and instructions they have used recently. A widely held rule of thumb is that a program spends 90% of its execution time in only 10% of the code. An implication of locality is that we can predict with reasonable accuracy what instructions and data a program will use in the near future based on its accesses in the recent past.

Locality of reference also applies to data accesses, though not as strongly as to code accesses. Two different types of locality have been observed. *Temporal locality* states that recently accessed items are likely to be accessed in the near future. *Spatial locality* says that items whose addresses are near one another tend to be referenced close together in time. We will see these principles applied in Chapter 5.

Take Advantage of Parallelism

Taking advantage of parallelism is one of the most important methods for improving performance. We give three brief examples, which are expounded on in later chapters. Our first example is the use of parallelism at the system level. To improve the throughput performance on a typical server benchmark, such as SPECWeb or TPC, multiple processors and multiple disks can be used. The workload of handling requests can then be spread among the CPUs or disks resulting in improved throughput. This is the reason that scalability is viewed as a valuable asset for server applications.

At the level of an individual processor, taking advantage of parallelism among instructions is critical to achieving high performance. One of the simplest ways

to do this is through pipelining. The basic idea behind pipelining, which is explained in more detail in Appendix A and a major focus of Chapter 3, is to overlap the execution of instructions, so as to reduce the total time to complete a sequence of instructions. Viewed from the perspective of the CPU performance equation, we can think of pipelining as reducing the CPI by allowing instructions that take multiple cycles to overlap. A key insight that allows pipelining to work is that not every instruction depends on its immediate predecessor, and thus, executing the instructions completely or partially in parallel may be possible.

Parallelism can also be exploited at the level of detailed digital design. For example, set associative caches use multiple banks of memory that are typical searched in parallel to find a desired item. Modern ALUs use carry-lookahead, which uses parallelism to speed the process of computing sums from linear in the number of bits in the operands to logarithmic.

There are many different ways designers take advantage of parallelism. One common class of techniques is parallel computation of two or more *possible* outcomes, followed by late selection. This technique is used in carry select adders, in set associative caches, and in handling branches in pipelines. Virtually every chapter in this book will have an example of how performance is enhanced through the exploitation of parallelism.

1.7 Putting It All Together: Performance and Price-Performance

In the *Putting It All Together* sections that appear near the end of every chapter, we show real examples that use the principles in that chapter. In this section we look at measures of performance and price-performance first in desktop systems using the SPEC CPU benchmarks, then at servers using TPC-C as the benchmark, and finally at the embedded market using EEMBC as the benchmark.

Performance and Price-Performance for Desktop Systems

Although there are many benchmark suites for desktop systems, a majority of them are OS or architecture specific. In this section we examine the CPU performance and price-performance of a variety of desktop systems using the SPEC CPU2000 integer and floating point suites. As mentioned earlier, SPEC CPU2000 summarizes CPU performance using a geometric mean normalized to a Sun system with larger numbers indicating higher performance.

Each system was configured with one CPU, 512 MB of SDRAM (with ECC if available), approximately 20 GB of disk, a fast graphics system, and an 10/100 Mb Ethernet connection. The seven systems we examined and their processors and price are shown in Figure 1.18. The wide variation in price is driven by a number of factors, including system expandability, the use of cheaper disks (ATA versus SCSI), less expensive memory (PC memory versus custom DIMMs), software differences (Linux or a Microsoft OS versus a vendor specific OS), the cost

of the CPU, and the commoditization effect, which we discussed on page 14. (See the further discussion on price variation in the caption of Figure 1.18.)

Vendor	Model	Processor	Clock Rate (MHz)	Price
Compaq	Presario 7000	AMD Athlon	1,400	$2,091
Dell	Precision 420	Intel Pentium III	1,000	$3,834
Dell	Precision 530	Intel Pentium 4	1,700	$4,175
HP	Workstation c3600	PA 8600	552	$12,631
IBM	RS6000 44P/170	IBM III-2	450	$13,889
Sun	Sunblade 100	UltraSPARC II-e	500	$2,950
Sun	Sunblade 1000	UltraSPARC III	750	$9.950

FIGURE 1.18 Seven different desktop systems from five vendors using seven different microprocessors showing the processor, its clock rate, and the selling price. All these systems are configured with 512 MB of ECC SDRAM, a high-end graphics system (which is *not* the highest performance system available for the more expensive platforms), and approximately 20 GB of disk. Many factors are responsible for the wide variation in price despite this common elements. First, the systems offer different levels of expandability (with the Prescario system being the least expandable, the Dell systems and Sunblade 100 being moderately expandable, nd the HP, IBM, and Sunblade 1000 being very flexible and expandable). Second, the use of cheaper disks (ATA versus SCSI) and less expensive memory (PC memory versus custom DIMMs) has a significant impact. Third the cost of the CPU varies by at least a factor of two. In 2001 the Athlon sells for about $200, The Pentium III for about $240, and the Pentium 4 for about $500. Fourth, software differences (Linux or a Microsoft OS versus a vendor specific OS) probably affect the final price. Fifth, the lower end systems use PC commodity parts in others areas (fans, power supply, support chip sets), which lower costs. Finally, the commoditization effect, which we discussed in page 14, is at work especially for the Compaq and Dell systems. These prices are as of July 2001.

Figure 1.19 shows the performance and the price-performance of these seven systems using SPEC CINT2000 as the metric. The Compaq system using the AMD Athlon CPU offers both the highest performance and the best price-performance, followed by the two Dell systems, which have comparable price-performance, although the Pentium 4 system is faster. The Sunblade 100 has the lowest performance, but somewhat better price-performance than the other UNIX-based workstation systems.

Figure 1.20 shows the performance and price-performance for the SPEC floating point benchmarks. The floating point instruction set enhancements in the Pentium 4 give it a clear performance advantage, although the Compaq Athlon-based system still has superior price-performance. The IBM, HP, and Sunblade 1000 all outperform the Dell 420 with a Pentium III, but the Dell system still offers better price-performance than the IBM, Sun, or HP workstations.

Performance and Price-Performance for Transaction Processing Servers

One of the largest server markets is online transaction processing (OLTP), which we described earlier. The standard industry benchmark for OLTP is TPC-C, which relies on a database system to perform queries and updates. Five factors

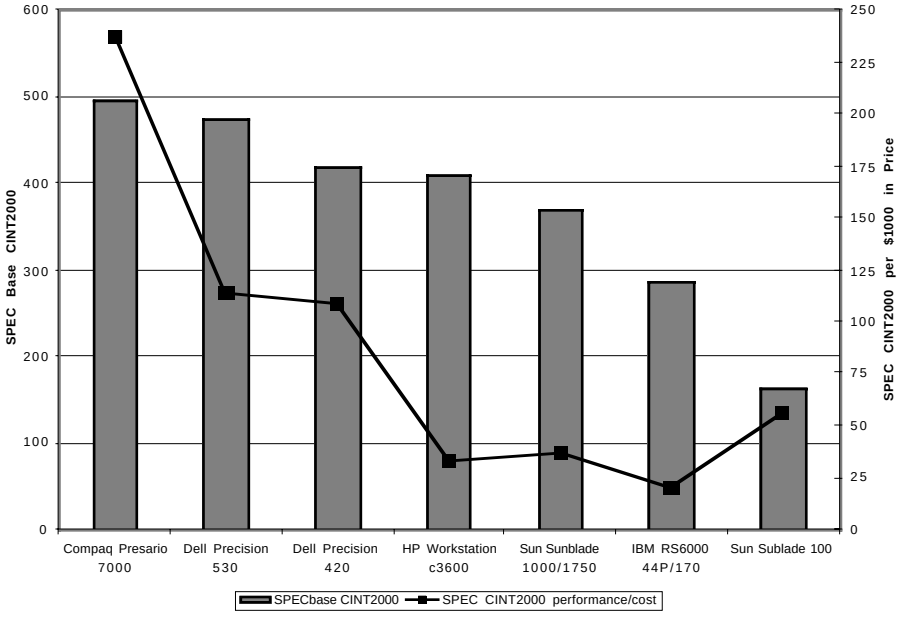


FIGURE 1.19 Performance and price-performance for seven systems are measured using SPEC CINT2000 as the benchmark. With the exception of the Sunblade 100 (Sun's low-end entry system), price-performance roughly parallels performance, contradicting the conventional wisdom–at least on the desktop–that higher performance systems carry a disproportionate price premium. Price-performance is plotted as CINT2000 performance per $1,000 in system cost. These performance numbers and prices are current as of July 2001.The measurements are available online as http://www.spec.org/osg/cpu2000/.

make the performance of TPC-C particularly interesting. First, TPC-C is a reasonable approximation to a real OLTP application; although this makes benchmark set-up complex and time consuming, it also makes the results reasonably indicative of real performance for OLTP. Second, TPC-C measures total system performance, including the hardware, the operating system, the I/O system, and the database system, making the benchmark more predictive of real performance. Third, the rules for running the benchmark and reporting execution time are very complete, resulting in more comparable numbers. Fourth, because of the importance of the benchmark, computer system efforts devote significant effort to making TPC-C run well. Fifth, vendors are required to report both performance and price-performance, enabling us to examine both.

Because the OLTP market is large and quite varied, there is an incredible range of computing systems used for these applications, ranging from small single processor servers to midrange multiprocessor systems to large-scale clusters

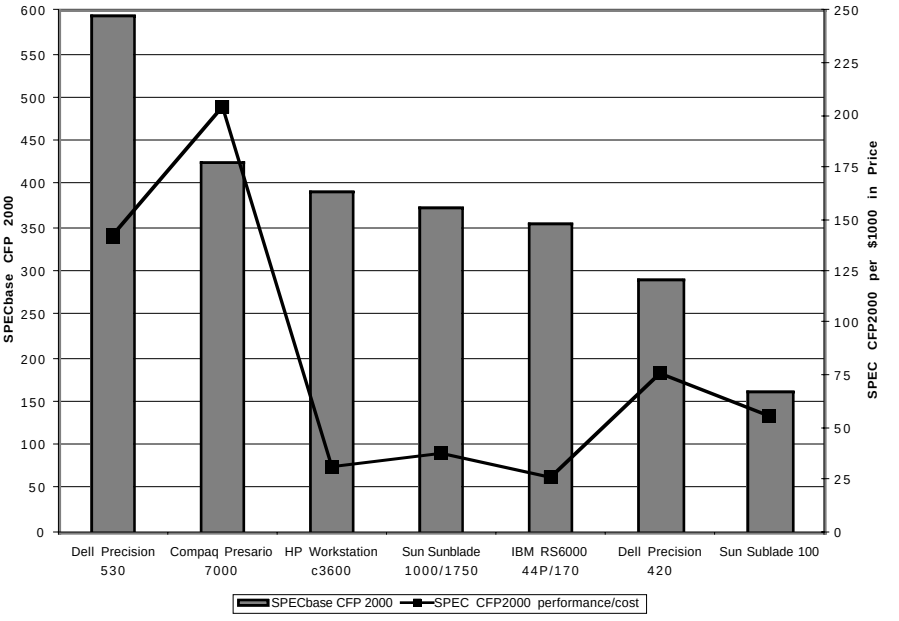


FIGURE 1.20 Performance and price-performance for seven systems are measured using SPEC CFP2000 as the benchmark. Price-performance is plotted as CFP2000 performance per $1,000 in system cost. The dramatically improved floating point performance of the Pentium 4 versus the Pentium III is clear in this figure. Price-performance partially parallels performance but not as clearly as in the case of the integer benchmarks. These performance numbers and prices are current as of July 2001. The measurements are available online as http://www.spec.org/osg/cpu2000/.

consisting of tens to hundreds of processors. To allow an appreciation for this diversity and its range of performance and price-performance, we will examine six of the top results by performance (and the comparative price-performance) and six of the top results by price-performance (and the comparative performance). For TPC-C performance is measured in transactions per minute (TPM), while price-performance is measured in TPM per dollar. Figure 1.21 shows the characteristics of a dozen systems whose performance or price-performance is near the top in one measure or the other.

Figure 1.22 charts the performance and price-performance of six of the highest performing OLTP systems described in Figure 1.21.The IBM cluster system, consisting of 280 Pentium III processors, provides the highest overall performance beating any other system by almost a factor of three, as well as the best price-performance by just over a factor of 1.5. The other systems are all moderate-scale multiprocessors and offer fairly comparable performance and similar

Vendor & System	CPUs	Database	OS	Price
IBM exSeries 370 c/s	280 x Pentium III @ 900 Mhz	Microsoft SQL Server 2000	Microsoft Windows Adv. Server	$15,543,346
Compaq Alpha server GS 320	32 x Alpha 21264 @ 1GHz	Oracle 9i	Compaq Tru64 UNIX	$10,286,029
Fujitsu PRIMEPOW-ER 20000	48 x SPARC64 GP @ 563 MHz	SymfoWARE Server Enterpr.	Sun Solaris 8	$9,671,742
IBM eServer 680 7017-S85	24 x IBM RS64-IV 600 MHz	Oracle 8 8.1.7.1	IBM AIX 4.3.3	$7,546,837
HP 9000 Enterprise Server	48 x HP PA-RISC 8600 552 MHz	Oracle8 v8.1.7.1	HP UX 11.i 64-bit	$8,522,104
IBM eServer 400 840-2420	24 x iSeries400 Model 840	IBM DB2 for AS/400 V4	IBM OS/400 V4	$8,448,137
Dell PowerEdge 6400	3 x Pentium III 700MHz	Microsoft SQL Server 2000	Microsoft Windows 2000	$131,275
IBM eserver xSeries 250 c/s	4 x Pentium III 700 MHz	Microsoft SQL Server 2000	Microsoft Windows Adv. Server	$297,277
Compaq Proliant ML570 6/700 2	4 x Intel Pentium III @ 700 MHz	Microsoft SQL Server 2000	Microsoft Windows Adv. Server	$375,016
HP NetServer LH 6000	6 x Pentium III @ 550 MHz	Microsoft SQL Server 2000	Microsoft Windows NT Enterprise	$372805
NEC Express 5800/180	8 x Pentium III 900 MHz	Microsoft SQL Server 2000	Microsoft Windows Adv. Server	$682,724
HP 9000 / L2000	4 x PA-RISC 8500 440MHz	Sybase Adaptive Server	HP UX 11.0 64-bit	$368,367

FIGURE 1.21 The characteristics of a dozen OLTP systems with either high total performance (top half of the table) or superior price-performance (bottom half of the table). The IBM exSeries with 280 Pentium IIIs is a cluster, while all the other systems are tightly coupled multiprocessors. Surprisingly, none of the top performing systems by either measure are uniprocessors! The system descriptions and detailed benchmark reports are available at: http://www.tpc.org/.

price-performance to the others in the group. Chapters 7 and 8 discuss the design of cluster and multiprocessor systems.

Figure 1.23 charts the performance and price-performance of the six OLTP systems from Figure 1.21 with the best price-performance. These systems are all multiprocessor systems, and, with the exception of the HP system, are based on Pentium III processors. Although the smallest system (the 3-processor Dell system) has the best price-performance, several of the other systems offer better performance at about a factor of 0.65 of the price-performance. Notice that the systems with the best price-performance in Figure 1.23 average almost four times better in price-performance (TPM/$ = 99 versus 27) than the high performance systems in Figure 1.22.

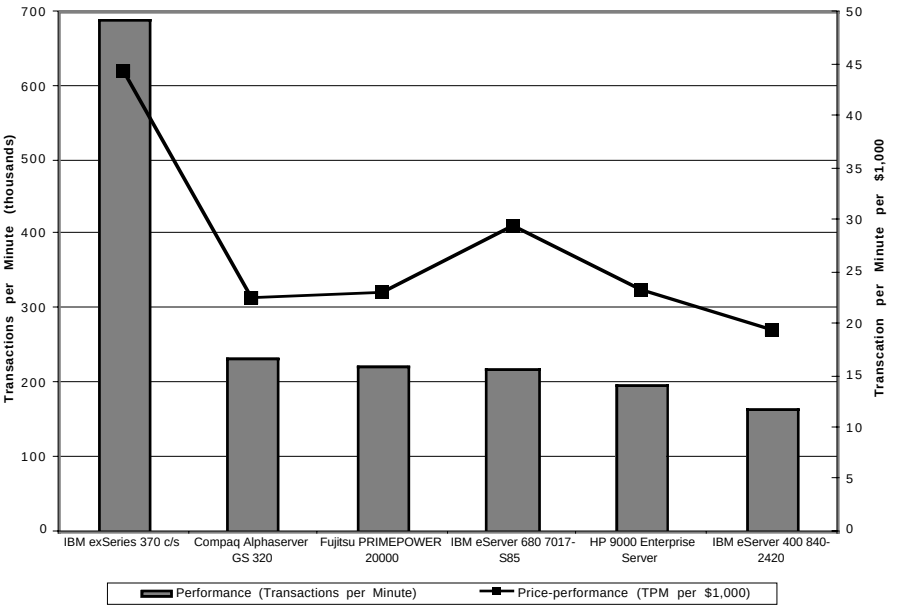


FIGURE 1.22 The performance (measured in thousands of transactions minute) and the price-performance (measured in transactions per minute per $1,000) are shown for six of the highest performing systems using TPC-C as the benchmark. Interestingly, IBM occupies three of these six positions, with different hardware platforms (a cluster of Pentium IIIs, an Power III based multiprocessor, and an AS 400 based multiprocessor.

Performance and Price-Performance for Embedded Processors

Comparing performance and price-performance of embedded processors is more difficult than for the desktop or server environments because of several characteristics. First, benchmarking is in its comparative infancy in the embedded space. Although the EEMBC benchmarks represent a substantial advance in benchmark availability and benchmark practice, as we discussed earlier, these benchmarks have significant drawbacks. Equally importantly, in the embedded space, processors are often designed for a particular class of applications; such designs are often not measured outside of their application space and when they are they may not perform well. Finally, as mentioned earlier cost and power are often the most important factors for an embedded application. Although we can partially measure cost by looking at the cost of the processor, other aspects of the design can be critical in determining system cost. For example, whether or not the memory controller and I/O control are integrated into the chip affects both power and cost of the system. As we said earlier, power is often the critical constraint in embed-

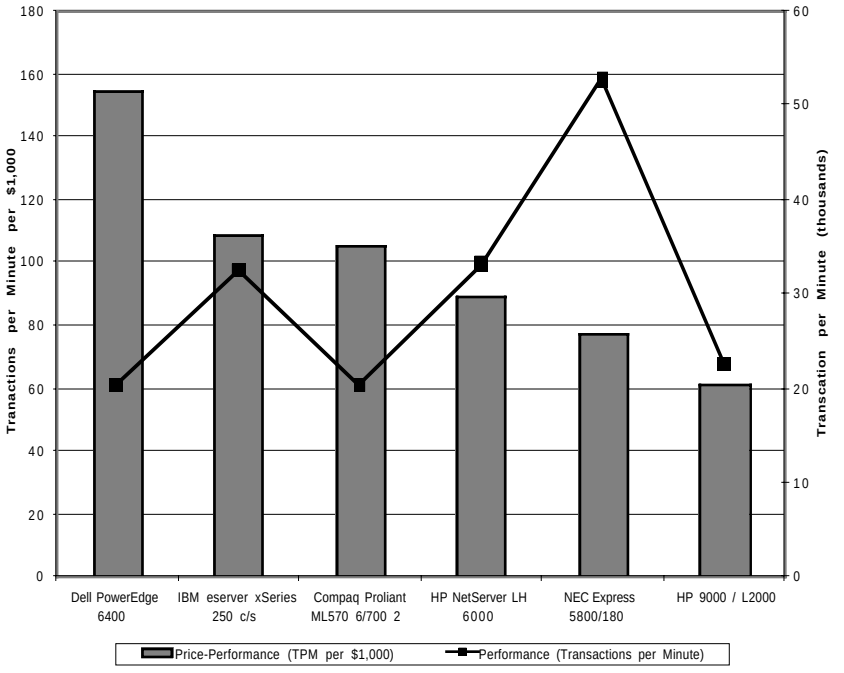


FIGURE 1.23 Price-performance (plotted as transactions per minute per $1000 of system cost) and overall performance (plotted as thousands of transactions per minute).

ded systems, and we focus on the relationship between performance and power in the next section.

Figure 1.24 shows the characteristics of the five processors whose price and price-performance we examine. These processors span a wide range of cost, power, and performance and thus are used in very different applications. The high-end processors, such as the PowerPC 650 and AMD Elan are used in applications such as network switches and possibly high-end laptops. The NEC VR 5432 series is a newer version of the VR 5400 series, which is one of the most heavily used processors in color laser printers. In contrast, the NEC VR 4121 is a low-end, low-power device used primarily in PDAs; in addition to the core computing

functions, the 4121 provides a number of system functions, reducing the cost of the overall system.

Processor	Instr. Set	Processor Clock Rate (MHz)	Cache Instr./Data On-chip Secondary cache	Processor organization	Typical power (in mW)	Price ($)
AMD Elan SC520	x86	133	16K/16K	Pipelined: single issue	1600	$38
AMD K6-2E+	x86	500	32K/32K 128K	Pipelined: 3+ issues/clock.	9600	$78
IBM PowerPC 750CX	PowerPC	500	32K/32K 128K	Pipelined 4 issues/clock	6000	$94
NEC VR 5432	MIPS-64	167	32K/32K	Pipelined: 2 issues/clock	2088	$25
NEC VR 4122	MIPS-64	180	32K/16K	Pipelined: single issue	700	$33

FIGURE 1.24 Five different embedded processors spanning a range of performance (more than a factor of ten, as we will see) and a wide range in price (roughly a factor of four and probably 50% higher than that if total system cost is considered). The price does not include interface and support chips, which could significantly increase the deployed system cost. Likewise, the power indicated includes only the processor's typical power consumption (in milliWatts); These processors also differ widely in terms of execution capability from a maximum of four instructions per clock to one! All the processors except the NEC VR4122 include a hardware floating point unit.

Figure 1.25 shows the relative performance of these five processors on three of the five EEMBC benchmark suites. The summary number for each benchmark suite is proportional to the geometric mean of the individual performance measures for each benchmark in the suite (measured as iterations per second). The clock rate differences explain between 33% and 75% of the performance differences. For machines with similar organization (such as the AMD Elan SC520 and the NEC VR 4121), the clock rate is the primary factor in determining performance. For machines with widely differing cache structures (such as the presence or absence of a secondary cache) or different pipelines, clock rate explains less of the performance difference.

Figure 1.26 shows the price-performance of these processors, where price is measured only by the processor cost. Here, the wide range in price narrows the performance differences, making the slower processors more cost effective. If our cost analysis also included the system support chips, the differences would narrow even further, probably boosting the VR 5432 to the top in price-performance and making the VR 4132 at least competitive with the high-end IBM and AMD chips.

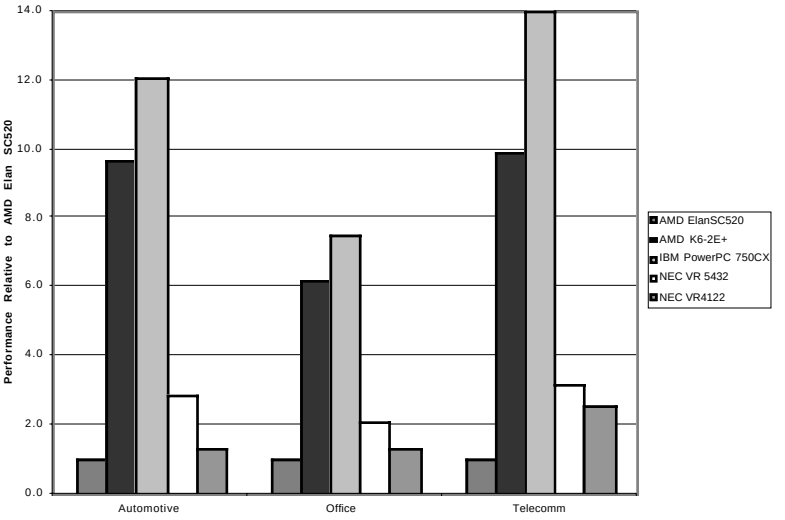


FIGURE 1.25 Relative performance for three of the five EEMBC benchmark suites on five different embedded processors. The performance is scaled relative to the AMD Elan SC520, so that the scores across the suites have a narrower range.

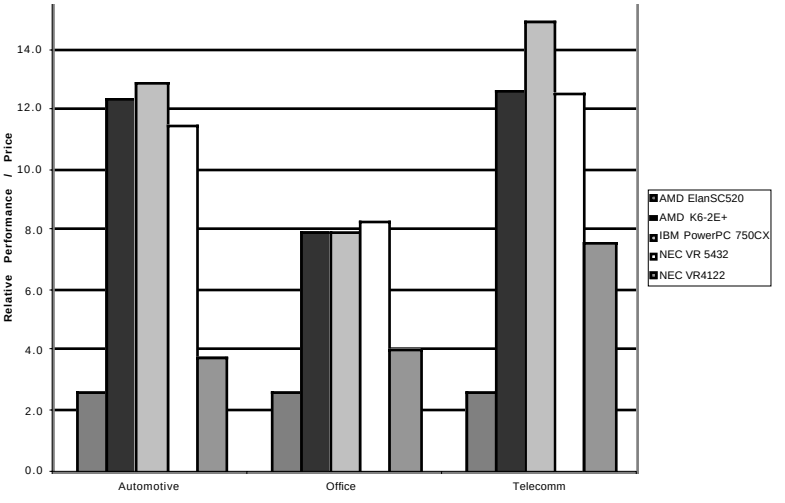


FIGURE 1.26 Relative price-performance for three of the five EEMBC benchmark suites on five different embedded processors, using only the price of the processor.

1.8 Another View: Power Consumption and Efficiency as the Metric

Throughout the chapters of this book, you will find sections entitled: *Another View.* These sections emphasize the way in which different segments of the computing market may solve a problem. For example, if the *Putting It All Together* section emphasizes the memory system for a desktop microprocessor, the *Another View* section may emphasize the memory system of an embedded application or a server. In this first Another View section, we look at the issue of power consumption in embedded processors.

As mentioned several times in this chapter, cost and power are often at least as important as performance in the embedded market. In addition to the cost of the processor module (which includes any required interface chips), memory is often the next most costly part of an embedded system. Recall that, unlike a desktop or server system, most embedded systems do not have secondary storage; instead, the entire application most reside in either FLASH or DRAM (as described in Chapter 5). Because many embedded systems, such as PDAs and cell phones, are constrained by both cost and physical size, the amount of memory needed for the application is critical. Likewise, power is often a determining factor in choosing a processor, especially for battery-powered systems.

As we saw in Figure 1.24 on page 56, the power for the five embedded processors we examined varies by more than a factor of 10. Clearly, the high performance AMD K6, with a typical power consumption of 9.3 W, cannot be used in environments where power or heat dissipation are critical. Figure 1.27 shows the relative performance per watt of typical operating power. Compare this figure to Figure 1.25 on page 57, which plots raw performance, and notice how different the results are. The NEC VR4122 has a clear advantage in performance per watt, but is the second lowest performing processor! From the viewpoint of power consumption the NEC VR4122, which was designed for battery-based systems, is the big winner. The IBM PowerPC displays efficient use of power to achieve its high performance, although at 6 watts typical, it is probably not be suitable for most battery-based devices.

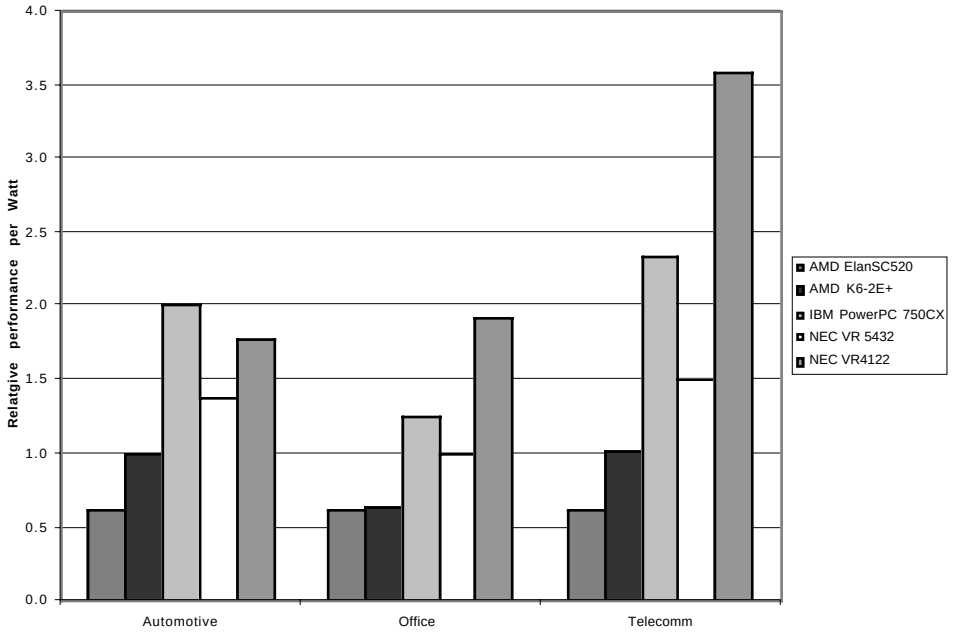


FIGURE 1.27 Relative performance per watt for the five embedded processors. The power is measured as typical operating power for the processor, and does not include any interface chips.

1.9 Fallacies and Pitfalls

The purpose of this section, which will be found in every chapter, is to explain some commonly held misbeliefs or misconceptions that you should avoid. We call such misbeliefs *fallacies.* When discussing a fallacy, we try to give a counterexample. We also discuss *pitfalls*—easily made mistakes. Often pitfalls are generalizations of principles that are true in a limited context. The purpose of these sections is to help you avoid making these errors in machines that you design.

Fallacy: The relative performance of two processors with the same ISA can be judged by clock rate or by the performance of a single benchmark suite.

As processors have become faster and more sophisticated, processor performance in one application area can diverge from that in another area. Sometimes the instruction set architecture is responsible for this, but increasingly the pipeline structure and memory system are responsible. This also means that clock rate is

not a good metric, even if the instruction sets are identical. Figure 1.28 shows the performance of a 1.7 GHz Pentium 4 relative to a 1 GHz Pentium III. The figure also shows the performance of a hypothetical 1.7 GHz *Pentium III* assuming linear scaling of performance based on the clock rate. In all cases except the SPEC floating point suite, the Pentium 4 delivers less performance per MHz than the Pentium III. As mentioned earlier, instruction set enhancements (the SSE2 extensions), which significantly boost floating point execution rates, are probably responsible for the better performance of the Pentium 4 for these floating point benchmarks.

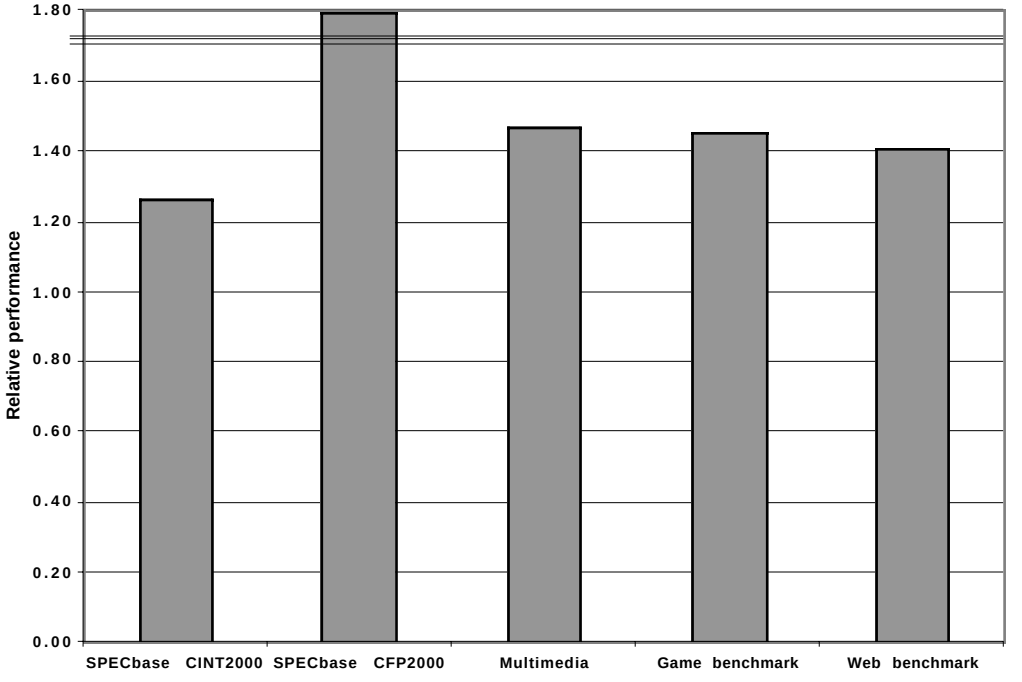


FIGURE 1.28 A comparison of the performance of the Pentium 4 (P4) relative to the Pentium III (P3) on five different sets of benchmark suites. The bars show the relative performance of a 1.7 GHz P4 versus a 1 GHz P3. The triple vertical line at 1.7 shows how much faster a Pentium 4 at 1.7 GHz would be than a 1 GHz Pentium III assuming performance scaled linearly with clock rate. Of course, this line represents an idealized approximation to how fast a P3 would run. The first two sets of bars are the SPEC integer and floating point suites. The third set of bars represents three multimedia benchmarks. The fourth set represents a pair of benchmarks based on the Game Quake, and the final benchmark is the composite Webmark score, a PC-based web benchmark

Performance within a single processor implementation family (such as Pentium III) usually scales slower than clock speed because of the increased relative cost of stalls in the memory system. Across generations (such as the Pentium 4 and Pentium III) enhancements to the basic implementation usually yield a performance that is somewhat better than what would be derived from just clock rate scaling. As Figure 1.28 shows, the Pentium 4 is usually slower than the Pentium III when performance is adjusted by linearly scaling the clock rate. This may partly derive from the focus on high clock rate as a primary design goal. We discuss both the differences between the Pentium III and Pentium 4 further in Chapter 3 as well as why the performance does not scale as fast as the clock rate does.

Fallacy: Benchmarks remain valid indefinitely.

Several factors influence the usefulness of a benchmark as a predictor of real performance and some of these may change over time. A big factor influencing the usefulness of a benchmark is the ability of the benchmark to resist "cracking," also known as benchmark engineering or "benchmarksmanship." Once a benchmark becomes standardized and popular, there is tremendous pressure to improve performance by targeted optimizations or by aggressive interpretation of the rules for running the benchmark. Small kernels or programs that spend their time in a very small number of lines of code are particularly vulnerable.

For example, despite the best intentions, the initial SPEC89 benchmark suite included a small kernel, called matrix300, which consisted of eight different 300 $\times$ 300 matrix multiplications. In this kernel, 99% of the execution time was in a single line (see SPEC [1989]). Optimization of this inner loop by the compiler (using an idea called blocking, discussed in Chapter 5) for the IBM Powerstation 550 resulted in performance improvement by a factor of more than 9 over an earlier version of the compiler! This benchmark tested compiler performance and was not, of course, a good indication of overall performance, nor of this particular optimization.

Even after the elimination of this benchmark, vendors found methods to tune the performance of individual benchmarks by the use of different compilers or preprocessors, as well as benchmark-specific flags. Although the baseline performance measurements requires the use of one set of flags for all benchmarks, the tuned or optimized performance does not. In fact, benchmark-specific flags are allowed, even if they are illegal in general and could lead to incorrect compilation!

Allowing benchmark and even input-specific flags has led to long lists of options, as Figure 1.29 shows. This list of options, which is not significantly different from the option lists used by other vendors, is used to obtain the peak performance for the Compaq AlphaServer DS20E Model 6/667. The list makes it clear why the baseline measurements were needed. The performance difference between the baseline and tuned numbers can be substantial. For the SPEC CFP2000 benchmarks on the AlphaServer DS20E Model 6/667, the overall performance (which by SPEC CPU2000 rules is summarized by geometric mean) is

1.12 times higher for the peak numbers. As compiler technology improves, the achieves closer to peak performance using the base flags. Similarly, as the benchmarks improve in quality, they become less suspectible to highly application specific optimizations. Thus, the gap between peak and base, which in early times was often 20%, has narrowed.

Peak: -v -g3 -arch ev6 -non_shared ONESTEP plus:

168.wupwise: f77 -fast -O4 -pipeline -unroll 2
171.swim: f90 -fast -O5 -transform_loops
172.mgrid: kf77 -O5 -transform_loops -tune ev6 -unroll 8
173.applu: f77 -fast -O5 -transform_loops -unroll 14
177.mesa: cc -fast -O4
178.galgel: kf90 -O4 -unroll 2 -ldxml RM_SOURCES = lapak.f90
179.art: kcc -fast -O4 -ckapargs='-arl=4 -ur=4' -unroll 10
183.equake: kcc -fast -ckapargs='-arl=4' -xtaso_short
187.facerec: f90 -fast -O4
188.ammp: cc -fast -O4 -xtaso_short
189.lucas: kf90 -fast -O5 -fkapargs='-ur=1' -unroll 1
191.fma3d: kf90 -O4
200.sixtrack: f90 -fast -O5 -transform_loops
301.apsi: kf90 -O5 -transform_loops -unroll 8 -fkapargs='-ur=1'

FIGURE 1.29 The tuning parameters for the SPEC CFP2000 report on an AlphaServer DS20E Model 6/667. This is the portion of the SPEC report for the tuned performance corresponding to that in Figure 1.14 on page 34. These parameters describe the compiler options (four different compilers are used). Each line shows the option used for one of the SPEC CFP2000 benchmarks. Data from: http://www.spec.org/osg/cpu2000/results/res1999q4/cpu2000-19991130-00012.html.

Ongoing improvements in technology can also change what a benchmark measures. Consider the benchmark gcc, considered one of the most realistic and challenging of the SPEC92 benchmarks. Its performance is a combination of CPU time and real system time. Since the input remains fixed and real system time is limited by factors, including disk access time, that improve slowly, an increasing amount of the runtime is system time rather than CPU time. This may be appropriate. On the other hand, it may be appropriate to change the input over time, reflecting the desire to compile larger programs. In fact, the SPEC92 input was changed to include four copies of each input file used in SPEC89; although this increases runtime, it may or may not reflect the way compilers are actually being used.

Over a long period of time, these changes may make even a well-chosen benchmark obsolete. For example, more than half the benchmarks added to the 1992 and 1995 SPEC CPU benchmark release were dropped from the next gener-

ation of the suite! To show how dramatically benchmarks must adapt over time, we summarize the status of the integer and FP benchmarks from SPEC 89, 92, and 95 in Figure 1.30.

Pitfall: Comparing hand-coded assembly and compiler generated high level language performance.

In most applications of computers, hand-coding is simply not tenable. A combination of the high cost of software development and maintenance together with time-to-market pressures have made it impossible for many applications to consider assembly language. In parts of the embedded market, however, several factors have continued to encourage limited use of hand coding, at least of key loops. The most important factors favoring this tendency are the importance of a few small loops to overall performance (particularly real-time performance) in some embedded applications, and the inclusion of instructions that can significantly boost performance of certain types of computations, but that compilers can not effectively use.

When performance is measured either by kernels or by applications that spend most of their time in a small number of loops, hand coding of the critical parts of the benchmark can lead to large performance gains. In such instances, the performance difference between the hand-coded and machine-generated versions of a benchmark can be very large, as shown in for two different machines in Figure 1.31. Both designers and users must be aware of this potentially large difference

Benchmark name	Integer or FP	SPEC 89	SPEC 92	SPEC 95	SPEC 2000
gcc	integer	**adopted**	modified	modified	modified
espresso	integer	**adopted**	modified	**dropped**	
li	integer	**adopted**	modified	modified	**dropped**
eqntott	integer	**adopted**	**dropped**		
spice	FP	**adopted**	modified	**dropped**	
doduc	FP	**adopted**		**dropped**	
nasa7	FP	**adopted**		**dropped**	
fpppp	FP	**adopted**		modified	**dropped**
matrix300	FP	**adopted**	**dropped**		
tomcatv	FP	**adopted**		modified	**dropped**
compress	integer		**adopted**	modified	**dropped**
sc	integer		**adopted**	**dropped**	
mdljdp2	FP		**adopted**	**dropped**	
wave5	FP		**adopted**	modified	**dropped**
ora	FP		**adopted**	**dropped**	
mdljsp2	FP		**adopted**	**dropped**	
alvinn	FP		**adopted**	**dropped**	
ear	FP		**adopted**	**dropped**	
swm256 (aka swim)	FP		**adopted**	modified	modified
su2cor	FP		**adopted**	modified	**dropped**
hydro2d	FP		**adopted**	modified	**dropped**
go	integer			**adopted**	**dropped**
m88ksim	integer			**adopted**	**dropped**
ijpeg	integer			**adopted**	**dropped**
perl	integer			**adopted**	modified
vortex	integer			**adopted**	modified
mgrid	FP			**adopted**	modified
applu	FP			**adopted**	**dropped**
apsi	FP			**adopted**	modified
turb3d				**adopted**	**dropped**

FIGURE 1.30 The evolution of the SPEC benchmarks over time showing when benchmarks were adopted, modified and dropped. All the programs in the 89, 92, and 95 releases are show. Modified indicates that either the input or the size of the benchmark was changed, usually to increase its running time and avoid perturbation in measurement or domination of the execution time by some factor other than CPU time.

and not extrapolate performance for compiler generate code from hand coded benchmarks.

Machine	EEMBC benchmark set	Performance Compiler generated	Performance Hand coded	Ratio hand/ compiler
Trimedia 1300 @166 MHz	Consumer	23.3	110.0	4.7
BOPS Manta @ 136 MHz	Telecomm	2.6	225.8	44.6
TI TMS320C6203 @ 300MHz	Telecomm	6.8	68.5	10.1

FIGURE 1.31 The performance of three embedded processors on C and hand-coded versions of portions of the EEMBC benchmark suite. In the case of the BOPS and TI processor, they also provide versions that are compiled but where the C is altered initially to improve performance and code generation; such versions can achieve most of the benefit from hand optimization at least for these machines and these benchmarks.

Fallacy: Peak performance tracks observed performance.

The only universally true definition of peak performance is "the performance level a machine is guaranteed not to exceed." The gap between peak performance and observed performance is typically a factor of 10 or more in supercomputers. (See Appendix B on vectors for an explanation.) Since the gap is so large and can vary significantly by benchmark, peak performance is not useful in predicting observed performance unless the workload consists of small programs that normally operate close to the peak.

As an example of this fallacy, a small code segment using long vectors ran on the Hitachi S810/20 in 1.3 seconds and on the Cray X-MP in 2.6 seconds. Although this suggests the S810 is two times faster than the X-MP, the X-MP runs a program with more typical vector lengths two times faster than the S810. These data are shown in Figure 1.32.

Measurement	Cray X-MP	Hitachi S810/20	Performance
A(i)=B(i)∗C(i)+D(i)∗E(i) (vector length 1000 done 100,000 times)	2.6 secs	1.3 secs	Hitachi 2 times faster
Vectorized FFT (vector lengths 64,32,…,2)	3.9 secs	7.7 secs	Cray 2 times faster

FIGURE 1.32 Measurements of peak performance and actual performance for the Hitachi S810/20 and the Cray X-MP. Note that the gap between peak and observed performance is large and can vary across benchmarks. Data from pages 18–20 of Lubeck, Moore, and Mendez [1985]. Also see *Fallacies and Pitfalls* in Appendix B.

Fallacy: The best design for a computer is the one that optimizes the primary objective without considering implementation.

Although in a perfect world where implementation complexity and implementation time could be ignored, this might be true, design complexity is an important factor. Complex designs take longer to complete, prolonging time to market. Given the rapidly improving performance of computers, longer design time means that a design will be less competitive. The architect must be constantly aware of the impact of his design choices on the design time for both hardware and software. The many postponements of the availability of the Itanium processor (roughly a two year delay from the initial target date) should serve as a topical reminder of the risks of introducing both a new architecture and a complex design. With processor performance increasing by just over 50% per year, each week delay translates to a 1% loss in relative performance!

Pitfall: Neglecting the cost of software in either evaluating a system or examining cost-performance.

For many years, hardware was so expensive that it clearly dominated the cost of software, but this is no longer true. Software costs in 2001 can be a large fraction of both the purchase and operational costs of a system. For example, for a medium size database OLTP server, Microsoft OS software might run about $2,000, while the Oracle software would run between $6,000 and $9,000 for a four-year, one-processor license. Assuming a four-year software lifetime means a total software cost for these two major components of between $8,000 and $11,000. A midrange Dell server with 512MB of memory, Pentium III at 1 GHz, and between 20 and 100 GB of disk would cost roughly the same amount as these two major software components. Meaning that software costs are roughly 50% of the total system cost!

Alternatively, consider a professional desktop system, which can be purchased with 1 GHz Pentium III, 128 MB DRAM, 20 GB disk, and a 19 inch monitor for just under $1000. The software costs of a Windows OS and Office 2000 are about $300 if bundled with the system and about double that if purchased separately, so the software costs are somewhere between 23% and 38% of the total cost!

Pitfall: Falling prey to Amdahl's Law.

Virtually every practicing computer architect knows Amdahl's Law. Despite this, we almost all occasionally fall into the trap of expending tremendous effort optimizing some aspect of a system before we measure its usage. Only when the overall speedup is unrewarding, do we recall that we should have measured the usage of that feature before we spent so much effort enhancing it!

Fallacy: Synthetic benchmarks predict performance for real programs.

This fallacy appeared in the first edition of this book, published in 1990. With the arrival and dominance of organizations such as SPEC and TPC, we thought perhaps the computer industry had learned a lesson and reformed its faulty practices, but the emerging embedded market, has embraced Dhrystone as its most quoted benchmark! Hence, this fallacy survives.

The best known examples of synthetic benchmarks are Whetstone and Dhrystone. These are not real programs and, as such, may not reflect program behavior for factors not measured. Compiler and hardware optimizations can artificially inflate performance of these benchmarks but not of real programs. The other side of the coin is that because these benchmarks are not natural programs, they don't reward optimizations of behaviors that occur in real programs. Here are some examples:

- Optimizing compilers can discard 25% of the Dhrystone code; examples include loops that are only executed once, making the loop overhead instructions unnecessary. To address these problems the authors of the benchmark "require" both optimized and unoptimized code to be reported. In addition, they "forbid" the practice of inline-procedure expansion optimization, since Dhrystone's simple procedure structure allows elimination of all procedure calls at almost no increase in code size.
- Most Whetstone floating-point loops execute small numbers of times or include calls inside the loop. These characteristics are different from many real programs. As a result Whetstone underrewards many loop optimizations and gains little from techniques such as multiple issue (Chapter 3) and vectorization (Appendix B).
- Compilers can optimize a key piece of the Whetstone loop by noting the relationship between square root and exponential, even though this is very unlikely to occur in real programs. For example, one key loop contains the following FORTRAN code:

```
X = SQRT(EXP(ALOG(X)/T1))
```

It could be compiled as if it were

```
X = EXP(ALOG(X)/(2×T1))
```

since

$$\texttt{SQRT(EXP(X))} = \sqrt[2]{e^X} = e^{X/2} = \texttt{EXP(X/2)}$$

It would be surprising if such optimizations were ever invoked except in this synthetic benchmark. (Yet one reviewer of this book found several compilers that performed this optimization!) This single change converts all calls to the square root function in Whetstone into multiplies by 2, surely improving performance—if Whetstone is your measure.

Fallacy: MIPS is an accurate measure for comparing performance among computers.

This fallacy also appeared in the first edition of this book, published in 1990. Your authors initially thought it could be retired, but, alas, the embedded market

not only uses Dhrystone as the benchmark of choice, but reports performance as "Dhrystone MIPS", a measure that this fallacy will show is problematic.

One alternative to time as the metric is MIPS, or *million instructions per second*. For a given program, MIPS is simply

$$\text{MIPS} = \frac{\text{Instruction count}}{\text{Execution time} \times 10^6} = \frac{\text{Clock rate}}{\text{CPI} \times 10^6}$$

Some find this rightmost form convenient since clock rate is fixed for a machine and CPI is usually a small number, unlike instruction count or execution time. Relating MIPS to time,

$$\text{Execution time} = \frac{\text{Instruction count}}{\text{MIPS} \times 10^6}$$

Since MIPS is a rate of operations per unit time, performance can be specified as the inverse of execution time, with faster machines having a higher MIPS rating.

The good news about MIPS is that it is easy to understand, especially by a customer, and faster machines means bigger MIPS, which matches intuition. The problem with using MIPS as a measure for comparison is threefold:

- MIPS is dependent on the instruction set, making it difficult to compare MIPS of computers with different instruction sets.
- MIPS varies between programs on the same computer.
- Most importantly, MIPS can vary inversely to performance!

The classic example of the last case is the MIPS rating of a machine with optional floating-point hardware. Since it generally takes more clock cycles per floating-point instruction than per integer instruction, floating-point programs using the optional hardware instead of software floating-point routines take less time but have a *lower* MIPS rating. Software floating point executes simpler instructions, resulting in a higher MIPS rating, but it executes so many more that overall execution time is longer.

MIPS is sometimes used by a single vendor (e.g. IBM) within a single set of applications, where this measure is less hamrful since relative differences among MIPS ratings of machines with the same architecture and the same benchmarks are reasonably likely to track relative performance differences.

To try to avoid the worst difficulties of using MIPS as a performance measure, computer designers began using relative MIPS, which we discuss in detail on page 75, and this is what the embedded market reports for Dhrystone. Although less harmful than an actual MIPS measurement, relative MIPS have their shortcomings (e.g., they are not really MIPS!), especially when measured using Dhrystone!

1.10 Concluding Remarks

This chapter has introduced a number of concepts that we will expand upon as we go through this book. The major ideas in instruction set architecture and the alternatives available will be the primary subjects of Chapter 2. Not only will we see the functional alternatives, we will also examine quantitative data that enable us to understand the trade-offs. The quantitative principle, *Make the common case fast,* will be a guiding light in this next chapter, and the CPU performance equation will be our major tool for examining instruction set alternatives. Chapter 2 concludes an examination of how instruction sets are used by programs.

In Chapter 2, we will include a section, *Crosscutting Issues,* that specifically addresses interactions between topics addressed in different chapters. In that section within Chapter 2, we focus on the interactions between compilers and instruction set design. This *Crosscutting Issues* section will appear in all future chapters.

In Chapters 3 and 4 we turn our attention to instruction level parallelism (ILP), of which pipelining is the simplest and most common form. Exploiting ILP is one of the most important techniques for building high speed uniprocessors. The presence of two chapters reflects the fact that there are two rather different approaches to exploiting ILP. Chapter 3 begins with an extensive discussion of basic concepts that will prepare you not only for the wide range of ideas examined in both chapters, but also to understand and analyze new techniques that will be introduced in the coming years. Chapter 3 uses examples that span about 35 years, drawing from one of the first modern supercomputers (IBM 360/91) to the fastest processors in the market in 2001. It emphasizes what is called the dynamic or runtime approach to exploiting ILP. Chapter 4 focuses on compile-time approaches to exploiting ILP. These approaches were heavily used in the early 1990s and return again with the introduction of the Intel Itanium. Appendix G is a version of an introductory chapter on pipelining from the 1995, Second Edition of this text. For readers without much experience and background in pipelining, that appendix is a useful bridge between the basic topics explored in this chapter (which we expect to be review for many readers, including those of our more introductory text, Computer Organization and Design: The Hardware/Software Interface) and the advanced topics in Chapter 3.

In Chapter 5 we turn to the all-important area of memory system design. We will examine a wide range of techniques that conspire to make memory look infinitely large while still being as fast as possible. As in Chapters 3 and 4, we will see that hardware-software cooperation has become a key to high-performance memory systems, just as it has to high-performance pipelines.

In Chapters 6 and 7, we move away from a CPU-centric view and discuss issues in storage systems and interconnect. We apply a similar quantitative approach, but one based on observations of system behavior and using an end-to-

end approach to performance analysis. Chapter 6 addresses the important issue of how to efficiently store and retrieve data using primarily lower-cost magnetic storage technologies. As we saw earlier, such technologies offer better cost per bit by a factor of 50–100 over DRAM. Magnetic storage is likely to remain advantageous wherever cost or nonvolatility (it keeps the information after the power is turned off) are important. In Chapter 6, our focus is on examining the performance of disk storage systems for typical I/O-intensive workloads, which are the counterpart to the CPU benchmarks we saw in this chapter. We extensively explore the idea of RAID-based systems, which use many small disks, arranged in a redundant fashion to achieve both high performance and high availability. Chapter 7 discusses the primary interconnection technology used for I/O devices. This chapter explores the topic of system interconnect more broadly, including wide-area and system-area networks used to allow computers to communicate. Chapter 7 also describes clusters, which are growing in importance due to their suitability and efficiency for database and web server applications.

Our final chapter returns to the issue of achieving higher performance through the use of multiple processors, or multiprocessors. Instead of using parallelism to overlap individual instructions, multiprocessing uses parallelism to allow multiple instruction streams to be executed simultaneously on different processors. Our focus is on the dominant form of multiprocessors, shared-memory multiprocessors, though we introduce other types as well and discuss the broad issues that arise in any multiprocessor. Here again, we explore a variety of techniques, focusing on the important ideas first introduced in the 1980s and 1990s.

.

1.11 Historical Perspective and References

If... history... teaches us anything, it is that man in his quest for knowledge and progress, is determined and cannot be deterred.

John F. Kennedy, Address at Rice University (1962)

A section of historical perspectives closes each chapter in the text. This section provides historical background on some of the key ideas presented in the chapter. The authors may trace the development of an idea through a series of machines or describe significant projects. If you're interested in examining the initial development of an idea or machine or interested in further reading, references are provided at the end of the section. In this historical section, we discuss the early development of digital computers and the development of performance measurement methodologies. The development of the key innovations in desktop, server, and embedded processor architectures are discussed in historical sections in virtually every chapter of the book.

The First General-Purpose Electronic Computers

J. Presper Eckert and John Mauchly at the Moore School of the University of Pennsylvania built the world's first fully-operational electronic general-purpose computer. This machine, called ENIAC (Electronic Numerical Integrator and Calculator), was funded by the U.S. Army and became operational during World War II, but it was not publicly disclosed until 1946. ENIAC was used for computing artillery firing tables. The machine was enormous—100 feet long, 8 1/2 feet high, and several feet wide. Each of the 20 10-digit registers was 2 feet long. In total, there were 18,000 vacuum tubes.

Although the size was three orders of magnitude bigger than the size of the average machines built today, it was more than five orders of magnitude slower, with an add taking 200 microseconds. The ENIAC provided conditional jumps and was programmable, which clearly distinguished it from earlier calculators. Programming was done manually by plugging up cables and setting switches and required from a half-hour to a whole day. Data were provided on punched cards. The ENIAC was limited primarily by a small amount of storage and tedious programming.

In 1944, John von Neumann was attracted to the ENIAC project. The group wanted to improve the way programs were entered and discussed storing programs as numbers; von Neumann helped crystallize the ideas and wrote a memo proposing a stored-program computer called EDVAC (Electronic Discrete Variable Automatic Computer). Herman Goldstine distributed the memo and put von Neumann's name on it, much to the dismay of Eckert and Mauchly, whose names were omitted. This memo has served as the basis for the commonly used term *von Neumann computer.* Several early inventors in the computer field believe that this term gives too much credit to von Neumann, who conceptualized and wrote up the ideas, and too little to the engineers, Eckert and Mauchly, who worked on the machines. Like most historians, your authors (winners of the 2000 IEEE von Neumann Medal) believe that all three individuals played a key role in developing the stored program computer. von Neumann's role in writing up the ideas, in generalizing them, and in thinking about the programming aspects was critical in transferring the ideas to a wider audience.

In 1946, Maurice Wilkes of Cambridge University visited the Moore School to attend the latter part of a series of lectures on developments in electronic computers. When he returned to Cambridge, Wilkes decided to embark on a project to build a stored-program computer named EDSAC, for Electronic Delay Storage Automatic Calculator. (The EDSAC used mercury delay lines for its memory; hence the phrase "delay storage" in its name.) The EDSAC became operational in 1949 and was the world's first full-scale, operational, stored-program computer [Wilkes, Wheeler, and Gill 1951; Wilkes 1985, 1995]. (A small prototype called the Mark I, which was built at the University of Manchester and ran in 1948, might be called the first operational stored-program machine.) The EDSAC was an accumulator-based architecture. This style of instruction set architecture re-

mained popular until the early 1970s. (Chapter 2 starts with a brief summary of the EDSAC instruction set.)

In 1947, Eckert and Mauchly applied for a patent on electronic computers. The dean of the Moore School, by demanding the patent be turned over to the university, may have helped Eckert and Mauchly conclude they should leave. Their departure crippled the EDVAC project, which did not become operational until 1952.

Goldstine left to join von Neumann at the Institute for Advanced Study at Princeton in 1946. Together with Arthur Burks, they issued a report based on the 1944 memo [1946]. The paper led to the IAS machine built by Julian Bigelow at Princeton's Institute for Advanced Study. It had a total of 1024 40-bit words and was roughly 10 times faster than ENIAC. The group thought about uses for the machine, published a set of reports, and encouraged visitors. These reports and visitors inspired the development of a number of new computers, including the first IBM computer, the 701, which was based on the IAS machine. The paper by Burks, Goldstine, and von Neumann was incredible for the period. Reading it today, you would never guess this landmark paper was written more than 50 years ago, as most of the architectural concepts seen in modern computers are discussed there (e.g., see the quote at the beginning of Chapter 5).

In the same time period as ENIAC, Howard Aiken was designing an electromechanical computer called the Mark-I at Harvard. The Mark-I was built by a team of engineers from IBM. He followed the Mark-I by a relay machine, the Mark-II, and a pair of vacuum tube machines, the Mark-III and Mark-IV. The Mark-III and Mark-IV were being built after the first stored-program machines. Because they had separate memories for instructions and data, the machines were regarded as reactionary by the advocates of stored-program computers. The term *Harvard architecture* was coined to describe this type of machine. Though clearly different from the original sense, this term is used today to apply to machines with a single main memory but with separate instruction and data caches.

The Whirlwind project [Redmond and Smith 1980] began at MIT in 1947 and was aimed at applications in real-time radar signal processing. Although it led to several inventions, its overwhelming innovation was the creation of magnetic core memory, the first reliable and inexpensive memory technology. Whirlwind had 2048 16-bit words of magnetic core. Magnetic cores served as the main memory technology for nearly 30 years.

Important Special-Purpose Machines

During the Second Wold War, there were major computing efforts in both Great Britain and the United States focused on special-purpose code-breaking computers. The work in Great Britain was aimed at decrypting messages encoded with the German Enigma coding machine. This work, which occurred at a location called Bletchley Park, led to two important machines. The first, an electromechanical machine, conceived of by Alan Turing, was called BOMB [see Good in

Metropolis 1980]. The second, much larger and electronic machine, conceived and designed by Newman and Flowers, was called COLOSSUS [see Randall in Metropolis 1980]. These were highly specialized cryptanalysis machines, which played a vital role in the war by providing the ability to read coded messages, especially those sent to U-boats. The work at Bletchley Park was highly classified (indeed some of it is still classified), and, so, its direct impact on the development of ENIAC, EDSAC and other computers is hard to trace, but it certainly had an indirect effect in advancing the technology and gaining understanding of the issues.

Similar work on special-purpose computers for cryptanalysis went on in the United States. The most direct descendent of this effort was a company Engineering Research Associates (ERA [see Thomash in Metropolis 1980], which was founded after the war to attempt to commercialize on the key ideas. ERA build several machines, which were sold to secret government agencies, and was eventually purchased by Sperry Rand, which had earlier purchased the Eckert Mauchly Computer Corporation.

Another early set of machines that deserves credit was a group of special-purpose machines built by Konrad Zuse in Germany in the late 1930s and early 1940s [see Bauer and Zuse in Metropolis 1980]. In addition to producing an operating machine, Zuse was the first to implement floating point, which von Neumann claimed was unnecessary!. His early machines used a mechanical store that was smaller than other electromechanical solutions of the time. His last machine was electromechanical but, because of the war, never completed.

An important early contributor to the development of electronic computers was John Atanasoff, who built a small-scale electronic computer in the early 1940s [Atanasoff 1940]. His machine, designed at Iowa State University, was a special-purpose computer (called the ABC: Atanasoff Berry Computer) that was never completely operational. Mauchly briefly visited Atanasoff before he built ENIAC and several of Atansanoff's ideas (e.g. using binary representation) likely influenced Mauchly. The presence of the Atanasoff machine, together with delays in filing the ENIAC patents (the work was classified and patents could not be filed until after the war) and the distribution of von Neumann's EDVAC paper, were used to break the Eckert-Mauchly patent [Larson 1973]. Though controversy still rages over Atanasoff's role, Eckert and Mauchly are usually given credit for building the first working, general-purpose, electronic computer [Stern 1980]. Atanasoff, however, demonstrated several important innovations included in later computers. Atanasoff deserves much credit for his work, and he might fairly be given credit for the world's first special-purpose electronic computer and for possibly influencing Eckert and Mauchly.

Commercial Developments

In December 1947, Eckert and Mauchly formed Eckert-Mauchly Computer Corporation. Their first machine, the BINAC, was built for Northrop and was shown

in August 1949. After some financial difficulties, the Eckert-Mauchly Computer Corporation was acquired by Remington-Rand, later called Sperry-Rand. Sperry-Rand merged the Eckert-Mauchly acquisition, ERA, and its tabulating business to form a dedicated computer division, called UNIVAC. UNIVAC delivered its first computer, the UNIVAC I in June 1951. The UNIVAC I sold for $250,000 and was the first successful commercial computer—48 systems were built! Today, this early machine, along with many other fascinating pieces of computer lore, can be seen at the Computer Museum in Mountain View, California. Other places where early computing systems can be visited include the Deutsches Museum in Munich, and the Smithsonian in Washington, D.C., as well as numerous online virtual museums.

IBM, which earlier had been in the punched card and office automation business, didn't start building computers until 1950. The first IBM computer, the IBM 701 based on von Neumann's IAS machine, shipped in 1952 and eventually sold 19 units [see Hurd in Metropolis 1980].In the early 1950s, many people were pessimistic about the future of computers, believing that the market and opportunities for these "highly specialized" machines were quite limited. Nonetheless, IBM quickly became the most successful computer company. The focus on reliability and a customer and market driven strategy was key. Although the 701 and 702 were modest successes, IBM's next machine the 704/705, first delivered in 1954, greatly exceeded its initial sales forecast of 50 machines, thanks in part to the inclusion of core memory.

Several books describing the early days of computing have been written by the pioneers [Wilkes 1985, 1995; Goldstine 1972], as well as [Metropolis, Howlett, and Rota 1980], which is a collection of recollections by early pioneers. There are numerous independent histories, often built around the people involved [Slater 1987], as well as a journal, *Annals of the History of Computing*, devoted to the history of computing.

The history of some of the computers invented after 1960 can be found in Chapter 2 (the IBM 360, the DEC VAX, the Intel 80x86, and the early RISC machines), Chapters 3 and 4 (the pipelined processors, including Stretch and the CDC 6600), and Appendix B (vector processors including the TI ASC, CDC Star, and Cray processors).

Development of Quantitative Performance Measures: Successes and Failures

In the earliest days of computing, designers set performance goals—ENIAC was to be 1000 times faster than the Harvard Mark-I, and the IBM Stretch (7030) was to be 100 times faster than the fastest machine in existence. What wasn't clear, though, was how this performance was to be measured. In looking back over the years, it is a consistent theme that each generation of computers obsoletes the performance evaluation techniques of the prior generation.

The original measure of performance was time to perform an individual operation, such as addition. Since most instructions took the same execution time, the timing of one gave insight into the others. As the execution times of instructions in a machine became more diverse, however, the time for one operation was no longer useful for comparisons. To take these differences into account, an *instruction mix* was calculated by measuring the relative frequency of instructions in a computer across many programs. The Gibson mix [Gibson 1970] was an early popular instruction mix. Multiplying the time for each instruction times its weight in the mix gave the user the *average instruction execution time*. (If measured in clock cycles, average instruction execution time is the same as average CPI.) Since instruction sets were similar, this was a more accurate comparison than add times. From average instruction execution time, then, it was only a small step to MIPS (as we have seen, the one is the inverse of the other). MIPS had the virtue of being easy for the layman to understand.

As CPUs became more sophisticated and relied on memory hierarchies and pipelining, there was no longer a single execution time per instruction; MIPS could not be calculated from the mix and the manual. The next step was benchmarking using kernels and synthetic programs. Curnow and Wichmann [1976] created the Whetstone synthetic program by measuring scientific programs written in Algol 60. This program was converted to FORTRAN and was widely used to characterize scientific program performance. An effort with similar goals to Whetstone, the Livermore FORTRAN Kernels, was made by McMahon [1986] and researchers at Lawrence Livermore Laboratory in an attempt to establish a benchmark for supercomputers. These kernels, however, consisted of loops from real programs.

As it became clear that using MIPS to compare architectures with different instructions sets would not work, a notion of relative MIPS was created. When the VAX-11/780 was ready for announcement in 1977, DEC ran small benchmarks that were also run on an IBM 370/158. IBM marketing referred to the 370/158 as a 1-MIPS computer, and since the programs ran at the same speed, DEC marketing called the VAX-11/780 a 1-MIPS computer. Relative MIPS for a machine M was defined based on some reference machine as

$$\text{MIPS}_{\text{M}} = \frac{\text{Performance}_{\text{M}}}{\text{Performance}_{\text{reference}}} \times \text{MIPS}_{\text{reference}}$$

The popularity of the VAX-11/780 made it a popular reference machine for relative MIPS, especially since relative MIPS for a 1-MIPS computer is easy to calculate: If a machine was five times faster than the VAX-11/780, for that benchmark its rating would be 5 relative MIPS. The 1-MIPS rating was unquestioned for four years, until Joel Emer of DEC measured the VAX-11/780 under a time-sharing load. He found that the VAX-11/780 native MIPS rating was 0.5. Subsequent VAXes that run 3 native MIPS for some benchmarks were therefore called

6-MIPS machines because they run six times faster than the VAX-11/780. By the early 1980s, the term MIPS was almost universally used to mean relative MIPS.

The 1970s and 1980s marked the growth of the supercomputer industry, which was defined by high performance on floating-point-intensive programs. Average instruction time and MIPS were clearly inappropriate metrics for this industry, hence the invention of MFLOPS (Millions of FLoating-point Operations Per Second), which effectively measured the inverse of execution time for a benchmark. . Unfortunately customers quickly forget the program used for the rating, and marketing groups decided to start quoting peak MFLOPS in the supercomputer performance wars.

SPEC (System Performance and Evaluation Cooperative) was founded in the late 1980s to try to improve the state of benchmarking and make a more valid basis for comparison. The group initially focused on workstations and servers in the UNIX marketplace, and that remains the primary focus of these benchmarks today. The first release of SPEC benchmarks, now called SPEC89, was a substantial improvement in the use of more realistic benchmarks.

References

AMDAHL, G. M. [1967]. "Validity of the single processor approach to achieving large scale computing capabilities," *Proc. AFIPS 1967 Spring Joint Computer Conf. 30* (April), Atlantic City, N.J., 483–485.

ATANASOFF, J. V. [1940]. "Computing machine for the solution of large systems of linear equations," Internal Report, Iowa State University, Ames.

BELL, C. G. [1984]. "The mini and micro industries," *IEEE Computer* 17:10 (October), 14–30.

BELL, C. G., J. C. MUDGE, AND J. E. MCNAMARA [1978]. *A DEC View of Computer Engineering*, Digital Press, Bedford, Mass.

BURKS, A. W., H. H. GOLDSTINE, AND J. VON NEUMANN [1946]. "Preliminary discussion of the logical design of an electronic computing instrument," Report to the U.S. Army Ordnance Department, p. 1; also appears in *Papers of John von Neumann*, W. Aspray and A. Burks, eds., MIT Press, Cambridge, Mass., and Tomash Publishers, Los Angeles, Calif., 1987, 97–146.

CURNOW, H. J. AND B. A. WICHMANN [1976]. "A synthetic benchmark," *The Computer J.*, 19:1.

FLEMMING, P. J. AND J. J. WALLACE [1986]. "How not to lie with statistics: The correct way to summarize benchmarks results," *Comm. ACM* 29:3 (March), 218–221.

FULLER, S. H. AND W. E. BURR [1977]. "Measurement and evaluation of alternative computer architectures," *Computer* 10:10 (October), 24–35.

GIBSON, J. C. [1970]. "The Gibson mix," Rep. TR. 00.2043, IBM Systems Development Division, Poughkeepsie, N.Y. (Research done in 1959.)

GOLDSTINE, H. H. [1972]. *The Computer: From Pascal to von Neumann*, Princeton University Press, Princeton, N.J.

JAIN, R. [1991]. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*, Wiley, New York.

LARSON, E. R. [1973]. "Findings of fact, conclusions of law, and order for judgment," File No. 4–67, Civ. 138, *Honeywell v. Sperry Rand and Illinois Scientific Development*, U.S. District Court for the State of Minnesota, Fourth Division (October 19).

LUBECK, O., J. MOORE, AND R. MENDEZ [1985]. "A benchmark comparison of three supercomputers: Fujitsu VP-200, Hitachi S810/20, and Cray X-MP/2," *Computer* 18:12 (December), 10–24.

METROPOLIS, N., J. HOWLETT, AND G-C ROTA, EDITORS [1980], *A History of Computing in the Twentieth Century*, Academic Press, N.Y.

MCMAHON, F. M. [1986]. "The Livermore FORTRAN kernels: A computer test of numerical performance range," Tech. Rep. UCRL-55745, Lawrence Livermore National Laboratory, Univ. of California, Livermore (December).

REDMOND, K. C. AND T. M. SMITH [1980]. *Project Whirlwind—The History of a Pioneer Computer*, Digital Press, Boston.

SHURKIN, J. [1984]. *Engines of the Mind: A History of the Computer*, W. W. Norton, New York.

SLATER, R. [1987]. *Portraits in Silicon*, MIT Press, Cambridge, Mass.

SMITH, J. E. [1988]. "Characterizing computer performance with a single number," *Comm. ACM* 31:10 (October), 1202–1206.

SPEC [1989]. *SPEC Benchmark Suite Release 1.0*, October 2, 1989.

SPEC [1994]. *SPEC Newsletter* (June).

STERN, N. [1980]. "Who invented the first electronic digital computer," *Annals of the History of Computing* 2:4 (October), 375–376.

TOUMA, W. R. [1993]. *The Dynamics of the Computer Industry: Modeling the Supply of Workstations and Their Components*, Kluwer Academic, Boston.

WEICKER, R. P. [1984]. "Dhrystone: A synthetic systems programming benchmark," *Comm. ACM* 27:10 (October), 1013–1030.

WILKES, M. V. [1985]. *Memoirs of a Computer Pioneer*, MIT Press, Cambridge, Mass.

WILKES, M. V. [1995]. *Computing Perspectives*, Morgan Kaufmann, San Francisco.

WILKES, M. V., D. J. WHEELER, AND S. GILL [1951]. *The Preparation of Programs for an Electronic Digital Computer*, Addison-Wesley, Cambridge, Mass.

EXERCISES

Each exercise has a difficulty rating in square brackets and a list of the chapter sections it depends on in angle brackets. See the Preface for a description of the difficulty scale.

still a good exercise

1.1 [20/10/10/15] <1.6> In this exercise, assume that we are considering enhancing a machine by adding a vector mode to it. When a computation is run in vector mode it is 20 times faster than the normal mode of execution. We call the percentage of time that could be spent using vector mode the *percentage of vectorization.* Vectors are discussed in Appendix B, but you don't need to know anything about how they work to answer this question!

a. [20] <1.6> Draw a graph that plots the speedup as a percentage of the computation performed in vector mode. Label the *y* axis "Net speedup" and label the *x* axis "Percent vectorization."

b. [10] <1.6> What percentage of vectorization is needed to achieve a speedup of 2?

c. [10] <1.6> What percentage of vectorization is needed to achieve one-half the maxi-

mum speedup attainable from using vector mode?

d. [15] <1.6> Suppose you have measured the percentage of vectorization for programs to be 70%. The hardware design group says they can double the speed of the vector rate with a significant additional engineering investment. You wonder whether the compiler crew could increase the use of vector mode as another approach to increasing performance. How much of an increase in the percentage of vectorization (relative to current usage) would you need to obtain the same performance gain? Which investment would you recommend?

still a good exercise

1.2 [15/10] <1.6> Assume—as in the Amdahl's Law Example on page 41—that we make an enhancement to a computer that improves some mode of execution by a factor of 10. Enhanced mode is used 50% of the time, measured as a percentage of the execution time *when the enhanced mode is in use*. Recall that Amdahl's Law depends on the fraction of the original, *unenhanced* execution time that could make use of enhanced mode. Thus, we cannot directly use this 50% measurement to compute speedup with Amdahl's Law.

a. [15] <1.6> What is the speedup we have obtained from fast mode?

b. [10] <1.6> What percentage of the original execution time has been converted to fast mode?

1.3 [15] <1.6> Show that the problem statements in the Examples on page 42 and page 45 are the same.

this exercise has been known to cause confusion, thought the concept is good

1.4

1.5 [15] <1.6> Suppose we are considering a change to an instruction set. The base machine initially has only loads and stores to memory, and all operations work on the registers. Such machines are called *load-store* machines (see Chapter 2). Measurements of the load-store machine showing the *instruction mix* and clock cycle counts per instruction are given in Figure 1.32 on page 69.

Let's assume that 25% of the *arithmetic logic unit* (ALU) operations directly use a loaded operand that is not used again.

We propose adding ALU instructions that have one source operand in memory. These new *register-memory instructions* have a clock cycle count of 2. Suppose that the extended instruction set increases the clock cycle count for branches by 1, but it does not affect the clock cycle time. (Chapter 3, on pipelining, explains why adding register-memory instructions might slow down branches.) Would this change improve CPU performance?

cache exercises should be tossed since we eliminated that section, we need some simple pipelining exercises. Feel free to take some from the old chapter 3

1.6 [15] <1.7> Assume that we have a machine that with a perfect cache behaves as given in Figure 1.32.

With a cache, we have measured that instructions have a miss rate of 5%, data references have a miss rate of 10%, and the miss penalty is 40 cycles. Find the CPI for each instruction type with cache misses and determine how much faster the machine is with no cache misses versus with cache misses.

still a good exercise;

1.7 [20] <1.6> After graduating, you are asked to become the lead computer designer at Hyper Computers, Inc. Your study of usage of high-level language constructs suggests that procedure calls are one of the most expensive operations. You have invented a scheme that reduces the loads and stores normally associated with procedure calls and returns. The first thing you do is run some experiments with and without this optimization. Your experiments use the same state-of-the-art optimizing compiler that will be used with either version of the computer. These experiments reveal the following information:

- The clock rate of the unoptimized version is 5% higher.
- Thirty percent of the instructions in the unoptimized version are loads or stores.
- The optimized version executes two-thirds as many loads and stores as the unoptimized version. For all other instructions the dynamic execution counts are unchanged.
- All instructions (including load and store) take one clock cycle.

Which is faster? Justify your decision quantitatively.

still a good exercise, although dated. I wonder if it can be salvaged.

1.8 [15/15/8/12] <1.6,1.9> The Whetstone benchmark contains 195,578 basic floating-point operations in a single iteration, divided as shown in Figure 1.33.

Operation	Count
Add	82,014
Subtract	8,229
Multiply	73,220
Divide	21,399
Convert integer to FP	6,006
Compare	4,710
Total	195,578

FIGURE 1.33 The frequency of floating-point operations in the Whetstone benchmark.

Whetstone was run on a Sun 3/75 using the F77 compiler with optimization turned on. The Sun 3/75 is based on a Motorola 68020 running at 16.67 MHz, and it includes a floating-point coprocessor. The Sun compiler allows the floating point to be calculated with the coprocessor or using software routines, depending on compiler flags. A single iteration of Whetstone took 1.08 seconds using the coprocessor and 13.6 seconds using software. Assume that the CPI using the coprocessor was measured to be 10, while the CPI using soft-

ware was measured to be 6.

a. [15] <1.6,1.9> What is the MIPS rating for both runs?

b. [15] <1.6> What is the total number of instructions executed for both runs?

c. [8] <1.6> On the average, how many integer instructions does it take to perform a floating-point operation in software?

d. [12] <1.9> What is the MFLOPS rating for the Sun 3/75 with the floating-point co-processor running Whetstone? (Assume all the floating-point operations in Figure 1.21 count as one operation.)

a good exercise, but needs some updating of costs and the data used--newer processors, e.g.

1.9 [15/10/15/15/15] <1.3,1.4> This exercise estimates the complete packaged cost of a microprocessor using the die cost equation and adding in packaging and testing costs. We begin with a short description of testing cost and follow with a discussion of packaging issues.

Testing is the second term of the chip cost equation:

$$\text{Cost of integrated circuit} = \frac{\text{Cost of die} + \text{Cost of testing die} + \text{Cost of packaging}}{\text{Final test yield}}$$

Testing costs are determined by three components:

$$\text{Cost of testing die} = \frac{\text{Cost of testing per hour} \times \text{Average die test time}}{\text{Die yield}}$$

Since bad dies are discarded, die yield is in the denominator in the equation—the good must shoulder the costs of testing those that fail. (In practice, a bad die may take less time to test, but this effect is small, since moving the probes on the die is a mechanical process that takes a large fraction of the time.) Testing costs about $50 to $500 per hour, depending on the tester needed. High-end designs with many high-speed pins require the more expensive testers. For higher-end microprocessors test time would run $300 to $500 per hour. Die tests take about 5 to 90 seconds on average, depending on the simplicity of the die and the provisions to reduce testing time included in the chip.

The cost of a package depends on the material used, the number of pins, and the die area. The cost of the material used in the package is in part determined by the ability to dissipate power generated by the die. For example, a *plastic quad flat pack* (PQFP) dissipating less than 1 watt, with 208 or fewer pins, and containing a die up to 1 cm on a side costs $2 in 1995. A ceramic *pin grid array* (PGA) can handle 300 to 600 pins and a larger die with more power, but it costs $20 to $60. In addition to the cost of the package itself is the cost of the labor to place a die in the package and then bond the pads to the pins, which adds from a few cents to a dollar or two to the cost. Some good dies are typically lost in the assembly process, thereby further reducing yield. For simplicity we assume the final test yield is 1.0; in practice it is at least 0.95. We also ignore the cost of the final packaged test.

This exercise requires the information provided in Figure 1.34.

Microprocessor	Die area (mm^2)	Pins	Technology	Estimated wafer cost ($)	Package
MIPS 4600	77	208	CMOS, 0.6μ, 3M	3200	PQFP
PowerPC 603	85	240	CMOS, 0.6μ, 4M	3400	PQFP
HP 71x0	196	504	CMOS, 0.8μ, 3M	2800	Ceramic PGA
Digital 21064A	166	431	CMOS, 0.5μ, 4.5M	4000	Ceramic PGA
SuperSPARC/60	256	293	BiCMOS, 0.6μ, 3.5M	4000	Ceramic PGA

FIGURE 1.34 Characteristics of microprocessors. The technology entry is the process type, line width, and number of interconnect levels.

a. [15] <1.4> For each of the microprocessors in Figure 1.34, compute the number of good chips you would get per 20-cm wafer using the model on page 18. Assume a defect density of one defect per cm^2, a wafer yield of 95%, and assume $\alpha = 3$.

b. [10] <1.4> For each microprocessor in Figure 1.34, compute the cost per projected good die before packaging and testing. Use the number of good dies per wafer from part (a) of this exercise and the wafer cost from Figure 1.34.

c. [15] <1.3> Both package cost and test cost are proportional to pin count. Using the additional assumption shown in Figure 1.35, compute the cost per good, tested, and packaged part using the costs per good die from part (b) of this exercise.

Package type	Pin count	Package cost ($)	Test time (secs)	Test cost per hour ($)
PQFP	<220	12	10	300
PQFP	<300	20	10	320
Ceramic PGA	<300	30	10	320
Ceramic PGA	<400	40	12	340
Ceramic PGA	<450	50	13	360
Ceramic PGA	<500	60	14	380
Ceramic PGA	>500	70	15	400

FIGURE 1.35 Package and test characteristics.

d. [15] <1.3> There are wide differences in defect densities between semiconductor manufacturers. Find the costs for the largest processor in Figure 1.34 (total cost including packaging), assuming defect densities are 0.6 per cm^2 and assuming that defect densities are 1.2 per cm^2.

e. [15] <1.3> The parameter α depends on the complexity of the process. Additional metal levels result in increased complexity. For example, α might be approximated by the number of interconnect levels. For the Digital 21064a with 4.5 levels of interconnect, estimate the cost of working, packaged, and tested die if $\alpha = 3$ and if $\alpha = 4.5$. Assume a defect density of 0.8 defects per cm^2.

1.10 [12] <1.5> One reason people may incorrectly average rates with an arithmetic mean is that it always gives an answer greater than or equal to the geometric mean. Show that for any two positive integers, a and b, the arithmetic mean is always greater than or equal to the geometric mean. When are the two equal?

we ditched the harmonic mean, so if we keep this (it's not bad), we need to define it here--this would be fine, since it uses the exercises to expound on a topic

1.11 [12] <1.5> For reasons similar to those in Exercise 1.10, some people use arithmetic instead of the harmonic mean. Show that for any two positive rates, r and s, the arithmetic mean is always greater than or equal to the harmonic mean. When are the two equal?

good exercise, if simple exercise, but needs new data for spec (use spec2000)

1.12 [15/15] <1.5> Some of the SPECfp92 performance results from the SPEC92 Newsletter of June 1994 [SPEC 94] are shown in Figure 1.36. The SPECratio is simply the runtime for a benchmark divided into the VAX 11/780 time for that benchmark. The SPECfp92 number is computed as the geometric mean of the SPECratios. Let's see how a weighted arithmetic mean compares.

Program name	VAX-11/780 Time	DEC 3000 Model 800 SPECratio	IBM Powerstation 590 SPECratio	Intel Xpress Pentium 815\100 SPECratio
spice2g6	23,944	97	128	64
doduc	1,860	137	150	84
mdljdp2	7,084	154	206	98
wave5	3,690	123	151	57
tomcatv	2,650	221	465	74
ora	7,421	165	181	97
alvinn	7,690	385	739	157
ear	25,499	617	546	215
mdljsp2	3,350	76	96	48
swm256	12,696	137	244	43
su2cor	12,898	259	459	57
hydro2d	13,697	210	225	83
nasa7	16,800	265	344	61
fpppp	9,202	202	303	119
Geometric mean	**8,098**	**187**	**256**	**81**

FIGURE 1.36 SPEC92 performance for SPECfp92. The DEC 3000 uses a 200-MHz Alpha microprocessor (21064) and a 2-MB off-chip cache. The IBM Powerstation 590 uses a 66.67-MHz Power-2. The Intel Xpress uses a 100-MHz Pentium with a 512-KB off-chip secondary cache. Data from SPEC [1994].

a. [15] <1.5> Calculate the weights for a workload so that running times on the VAX-

11/780 will be equal for each of the 14 benchmarks (given in Figure 1.36).

b. [15] <1.5> Using the weights computed in part (a) of this exercise, calculate the weighted arithmetic means of the execution times of the 14 programs in Figure 1.36.

still a decent exercise

1.13 [15/15/15] <1.6,1.9> Three enhancements with the following speedups are proposed for a new architecture:

$\text{Speedup}_1 = 30$

$\text{Speedup}_2 = 20$

$\text{Speedup}_3 = 10$

Only one enhancement is usable at a time.

a. [15] <1.6> If enhancements 1 and 2 are each usable for 30% of the time, what fraction of the time must enhancement 3 be used to achieve an overall speedup of 10?

b. [15] <1.6,1.9> Assume the distribution of enhancement usage is 30%, 30%, and 20% for enhancements 1, 2, and 3, respectively. Assuming all three enhancements are in use, for what fraction of the reduced execution time is no enhancement in use?

c. [15] <1.6> Assume for some benchmark, the fraction of use is 15% for each of enhancements 1 and 2 and 70% for enhancement 3. We want to maximize performance. If only one enhancement can be implemented, which should it be? If two enhancements can be implemented, which should be chosen?

1.14 [15/10/10/12/10] <1.6,1.9> Your company has a benchmark that is considered representative of your typical applications. One of the older-model workstations does not have a floating-point unit and must emulate each floating-point instruction by a sequence of integer instructions. This older-model workstation is rated at 120 MIPS on this benchmark. A third-party vendor offers an attached processor that is intended to give a “mid-life kicker” to your workstation. That attached processor executes each floating-point instruction on a dedicated processor (i.e., no emulation is necessary). The workstation/attached processor rates 80 MIPS on the same benchmark. The following symbols are used to answer parts (a)–(e) of this exercise.

I—Number of integer instructions executed on the benchmark.

F—Number of floating-point instructions executed on the benchmark.

Y—Number of integer instructions to emulate a floating-point instruction.

W—Time to execute the benchmark on the workstation alone.

B—Time to execute the benchmark on the workstation/attached processor combination.

a. [15] <1.6,1.9> Write an equation for the MIPS rating of each configuration using the symbols above. Document your equation.

b. [10] <1.6> For the configuration without the coprocessor, we measure that $F = 8 \times 10^6$, $Y = 50$, and $W = 4$. Find I.

c. [10] <1.6> What is the value of B?

d. [12] <1.6,1.9> What is the MFLOPS rating of the system with the attached processor board?

e. [10] <1.6,1.9> Your colleague wants to purchase the attached processor board even though the MIPS rating for the configuration using the board is less than that of the workstation alone. Is your colleague's evaluation correct? Defend your answer.

1.15 [15/15/10] <1.5,1.9> Assume the two programs in Figure 1.15 on page 36 each execute 100 million floating-point operations during execution.

a. [15] <1.5,1.9> Calculate the MFLOPS rating of each program.

b. [15] <1.5,1.9> Calculate the arithmetic, geometric, and harmonic means of MFLOPS for each machine.

c. [10] <1.5,1.9> Which of the three means matches the relative performance of total execution time?

OK exercise, but needs updating

1.16 [10/12] <1.9,1.6> One problem cited with MFLOPS as a measure is that not all FLOPS are created equal. To overcome this problem, normalized or weighted MFLOPS measures were developed. Figure 1.37 shows how the authors of the "Livermore Loops" benchmark calculate the number of normalized floating-point operations per program according to the operations actually found in the source code. Thus, the *native MFLOPS* rating is not the same as the *normalized MFLOPS* rating reported in the supercomputer literature, which has come as a surprise to a few computer designers.

Real FP operations	Normalized FP operations
Add, Subtract, Compare, Multiply	1
Divide, Square root	4
Functions (Expo, Sin,...)	8

FIGURE 1.37 Real versus normalized floating-point operations. The number of normalized floating-point operations per real operation in a program used by the authors of the Livermore FORTRAN Kernels, or "Livermore Loops," to calculate MFLOPS. A kernel with one Add, one Divide, and one Sin would be credited with 13 normalized floating-point operations. Native MFLOPS won't give the results reported for other machines on that benchmark.

Let's examine the effects of this weighted MFLOPS measure. The spice program runs on the DECstation 3100 in 94 seconds. The number of floating-point operations executed in that program are listed in Figure 1.38.

Floating-point operation	Times executed

FIGURE 1.38 Floating-point operations in spice.

addD	25,999,440
subD	18,266,439
mulD	33,880,810
divD	15,682,333
compareD	9,745,930
negD	2,617,846
absD	2,195,930
convertD	1,581,450
Total	109,970,178

FIGURE 1.38 Floating-point operations in spice.

a. [10] <1.9,1.6> What is the native MFLOPS for spice on a DECstation 3100?

b. [12] <1.9,1.6> Using the conversions in Figure 1.37, what is the normalized MFLOPS?

1.17 [30] <1.5,1.9> Devise a program in C that gets the peak MIPS rating for a computer. Run it on two machines to calculate the peak MIPS. Now run the SPEC92 gcc on both machines. How well do peak MIPS predict performance of gcc?

1.18 [30] <1.5,1.9> Devise a program in C or FORTRAN that gets the peak MFLOPS rating for a computer. Run it on two machines to calculate the peak MFLOPS. Now run the SPEC92 benchmark spice on both machines. How well do peak MFLOPS predict performance of spice?

update

1.19 [Discussion] <1.5> What is an interpretation of the geometric means of execution times? What do you think are the advantages and disadvantages of using total execution times versus weighted arithmetic means of execution times using equal running time on the VAX-11/780 versus geometric means of ratios of speed to the VAX-11/780

2 Instruction Set Principles and Examples

A n *Add the number in storage location* n *into the accumulator*.

E n *If the number in the accumulator is greater than or equal to zero execute next the order which stands in storage location* n; *otherwise proceed serially*.

Z *Stop the machine and ring the warning bell*.

Wilkes and Renwick
Selection from the List of 18 Machine Instructions for the EDSAC (1949)

2.1	Introduction	99
2.2	Classifying Instruction Set Architectures	101
2.3	Memory Addressing	105
2.4	Addressing Modes for Signal Processing	111
2.5	Type and Size of Operands	114
2.6	Operands for Media and Signal Processing	116
2.7	Operations in the Instruction Set	118
2.8	Operations for Media and Signal Processing	118
2.9	Instructions for Control Flow	122
2.10	Encoding an Instruction Set	127
2.11	Crosscutting Issues: The Role of Compilers	130
2.12	Putting It All Together: The MIPS Architecture	140
2.13	Another View: The Trimedia TM32 CPU	151
2.14	Fallacies and Pitfalls	152
2.15	Concluding Remarks	158
2.16	Historical Perspective and References	160
	Exercises	172

2.1 Introduction

In this chapter we concentrate on instruction set architecture—the portion of the computer visible to the programmer or compiler writer. This chapter introduces the wide variety of design alternatives available to the instruction set architect. In particular, this chapter focuses on five topics. First, we present a taxonomy of instruction set alternatives and give some qualitative assessment of the advantages and disadvantages of various approaches. Second, we present and analyze some instruction set measurements that are largely independent of a specific instruction set. Third, we discuss instruction set architecture of processors not aimed at desktops or servers: digital signal processors (DSPs) and media processors. DSP and media processors are deployed in embedded applications, where cost and power are as important as performance, with an emphasis on real time performance. As discussed in Chapter 1, real time programmers often target worst case performance rather to guarantee not to miss regularly occurring events. Fourth, we address the issue of languages and compilers and their bearing on instruction set architecture. Finally, the *Putting It All Together* section shows how these ideas are reflected in the MIPS instruction set, which is typical of RISC architectures, and *Another View* presents the Trimedia TM32 CPU, an example of a media processor. We conclude with fallacies and pitfalls of instruction set design.

To make the illustrate the principles further, appendices B through E give four examples of general purpose RISC architectures (MIPS, Power PC, Precision Architecture, SPARC), four embedded RISC processors (ARM, Hitachi SH, MIPS 16, Thumb), and three older architectures (80x86, IBM 360/370, and VAX). Before we discuss how to classify architectures, we need to say something about instruction set measurement.

Throughout this chapter, we examine a wide variety of architectural measurements. Clearly, these measurements depend on the programs measured and on the compilers used in making the measurements. The results should not be interpreted as absolute, and you might see different data if you did the measurement with a different compiler or a different set of programs. The authors believe that the measurements in this chapter are reasonably indicative of a class of typical applications. Many of the measurements are presented using a small set of benchmarks, so that the data can be reasonably displayed and the differences among programs can be seen. An architect for a new computer would want to analyze a much larger collection of programs before making architectural decisions. The measurements shown are usually *dynamic*—that is, the frequency of a measured event is weighed by the number of times that event occurs during execution of the measured program.

Before starting with the general principles, let's review the three application areas from the last chapter. *Desktop computing* emphasizes performance of programs with integer and floating-point data types, with little regard for program size or processor power consumption. For example, code size has never been reported in the four generations of SPEC benchmarks. *Servers* today are used primarily for database, file server, and web applications, plus some timesharing applications for many users. Hence, floating-point performance is much less important for performance than integers and character strings, yet virtually every server processor still includes floating-point instructions. *Embedded applications* value cost and power, so code size is important because less memory is both cheaper and lower power, and some classes of instructions (such as floating point) may be optional to reduce chip costs.

Thus, instruction sets for all three applications are very similar; Appendix B <RISC> takes advantage of the similarities to describe eight instruction sets in just 43 pages. In point of fact, the MIPS architecture that drives this chapter has been used successfully in desktops, servers, and embedded applications.

One successful architecture very different from RISC is the 80x86 (see Appendix C). Surprisingly, its success does not necessarily belie the advantages of a RISC instruction set. The commercial importance of binary compatibility with PC software combined with the abundance of transistor's provided by Moore's Law led Intel to use a RISC instruction set internally while supporting an 80x86 instruction set externally. As we shall see in section 3.8 of the next chapter, recent Intel microprocessors use hardware to translate from 80x86 instructions to RISC-like instructions and then execute the translated operations inside the chip. They maintain the illusion of 80x86 architecture to the programmer while allowing the computer designer to implement a RISC-style processor for performance.

DSPs and media processors, which can be used in embedded applications, emphasize real-time performance and often deal with infinite, continuous streams of data. Keeping up with these streams often means targeting worst case performance to offer real time guarantees. Architects of these computers also have a tradition of identifying a small number of important kernels that are critical to success, and hence are often supplied by the manufacturer. As a result of this heritage, these instruction set architectures include quirks that can improve performance for the targeted kernels but that no compiler will ever generate.

In contrast, desktop and server applications historically do not to reward such eccentricities since they do not have as narrowly defined a set of important kernels, and since little of the code is hand optimized. If a compiler cannot generate it, desktop and server programs generally won't use it. We'll see the impact of these different cultures on the details of the instruction set architectures of this chapter.

Given the increasing importance of media to desktop and embedded applications, a recent trend is to merge these cultures by adding DSP/media instructions to conventional architectures. Hand coded library routines then try to deliver DSP/media performance using conventional desktop and media architectures, while compilers can generate code for the rest of the program using the conventional instruction set. Section 2.8 describes such extensions. Similarly, embedded applications are beginning to run more general-purpose code as they begin to include operating systems and more intelligent features.

Now that the background is set, we begin by exploring how instruction set architectures can be classified.

2.2 Classifying Instruction Set Architectures

The type of internal storage in a processor is the most basic differentiation, so in this section we will focus on the alternatives for this portion of the architecture. The major choices are a stack, an accumulator, or a set of registers. Operands may be named explicitly or implicitly: The operands in a *stack architecture* are implicitly on the top of the stack, and in an *accumulator architecture* one operand is implicitly the accumulator. The *general-purpose register architectures* have only explicit operands—either registers or memory locations. Figure 2.1 shows a block diagram of such architectures and Figure 2.2 shows how the code sequence `C = A + B` would typically appear in these three classes of instruction sets. The explicit operands may be accessed directly from memory or may need to be first loaded into temporary storage, depending on the class of architecture and choice of specific instruction.

As the figures show, there are really two classes of register computers. One class can access memory as part of any instruction, called *register-memory* architecture, and the other can access memory only with load and store instructions, called *load-store* or *register-register* architecture. A third class, not found in com-

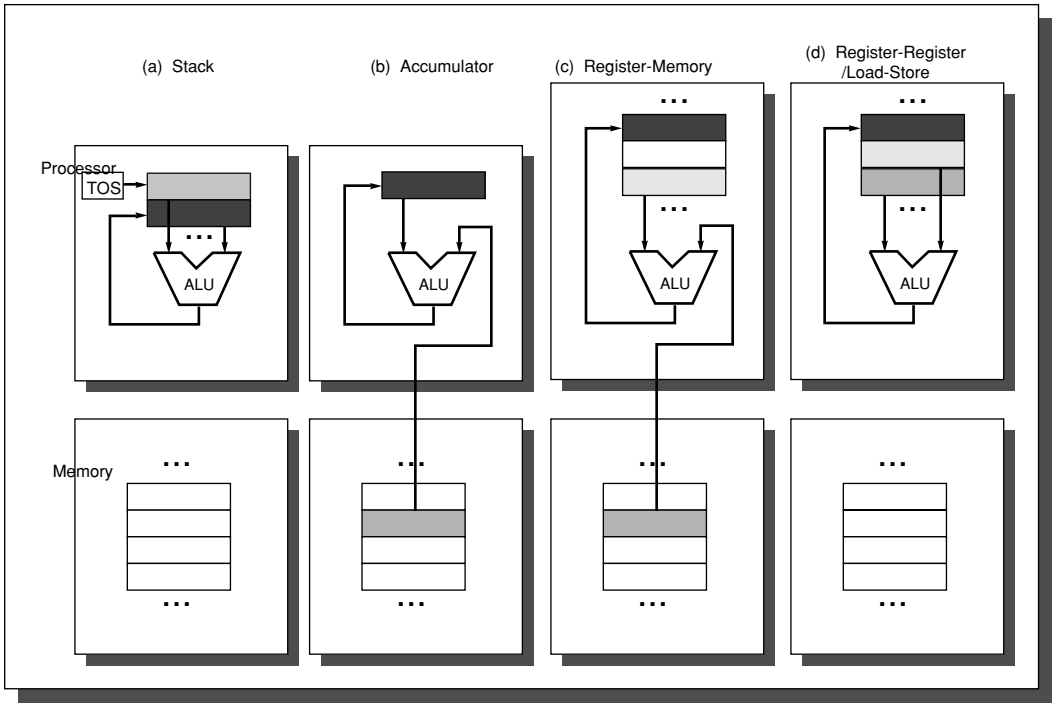


FIGURE 2.1 Operand locations for four instruction set architecture classes. The arrows indicate whether the operand is an input or the result of the ALU operation, or both an input and result. Lighter shades indicate inputs and the dark shade indicates the result. In (a), a Top Of Stack register (TOS), points to the top input operand, which is combined with the operand below. The first operand is removed from the stack, the result takes the place of the second operand, and TOS is updated to point to the result. All operands are implicit. In (b), the Accumulator is both an implicit input operand and a result. In (c) one input operand is a register, one is in memory, and the result goes to a register. All operands are registers in (d), and, like the stack architecture, can be transferred to memory only via separate instructions: push or pop for (a) and load or store for (d).

Stack	Accumulator	Register (register-memory)	Register (load-store)
`Push A`	`Load A`	`Load R1,A`	`Load R1,A`
`Push B`	`Add B`	`Add R3,R1,B`	`Load R2,B`
`Add`	`Store C`	`Store R3,C`	`Add R3,R1,R2`
`Pop C`			`Store R3,C`

FIGURE 2.2 The code sequence for C = A + B for four classes of instruction sets. Note that the Add instruction has implicit operands for stack and accumulator architectures, and explicit operands for register architectures. It is assumed that A, B, and C all belong in memory and that the values of A and B cannot be destroyed. Figure 2.1 shows the Add operation for each class of architecture.

puters shipping today, keeps all operands in memory and is called a *memory-memory* architecture. Some instruction set architectures have more registers than a single accumulator, but place restrictions on uses of these special registers. Such an architecture is sometimes called an *extended accumulator* or *special-purpose register* computer.

Although most early computers used stack or accumulator-style architectures, virtually every new architecture designed after 1980 uses a load-store register architecture. The major reasons for the emergence of general-purpose register (GPR) computers are twofold. First, registers—like other forms of storage internal to the processor—are faster than memory. Second, registers are more efficient for a compiler to use than other forms of internal storage. For example, on a register computer the expression `(A*B) - (B*C) - (A*D)` may be evaluated by doing the multiplications in any order, which may be more efficient because of the location of the operands or because of pipelining concerns (see Chapter 3). Nevertheless, on a stack computer the hardware must evaluate the expression in only one order, since operands are hidden on the stack, and it may have to load an operand multiple times.

More importantly, registers can be used to hold variables. When variables are allocated to registers, the memory traffic reduces, the program speeds up (since registers are faster than memory), and the code density improves (since a register can be named with fewer bits than can a memory location).

As explained in section 2.11, compiler writers would prefer that all registers be equivalent and unreserved. Older computers compromise this desire by dedicating registers to special uses, effectively decreasing the number of general-purpose registers. If the number of truly general-purpose registers is too small, trying to allocate variables to registers will not be profitable. Instead, the compiler will reserve all the uncommitted registers for use in expression evaluation. The dominance of hand-optimized code in the DSP community has lead to DSPs with many special-purpose registers and few general-purpose registers.

How many registers are sufficient? The answer, of course, depends on the effectiveness of the compiler. Most compilers reserve some registers for expression evaluation, use some for parameter passing, and allow the remainder to be allocated to hold variables. Just as people tend to be bigger than their parents, new instruction set architectures tend to have more registers than their ancestors.

Two major instruction set characteristics divide GPR architectures. Both characteristics concern the nature of operands for a typical arithmetic or logical instruction (ALU instruction). The first concerns whether an ALU instruction has two or three operands. In the three-operand format, the instruction contains one result operand and two source operands. In the two-operand format, one of the operands is both a source and a result for the operation. The second distinction among GPR architectures concerns how many of the operands may be memory addresses in ALU instructions. The number of memory operands supported by a typical ALU instruction may vary from none to three. Figure 2.3 shows combinations of these two attributes with examples of computers. Although there are seven possi-

ble combinations, three serve to classify nearly all existing computers. As we mentioned earlier, these three are register-register (also called load-store), register-memory, and memory-memory.

Number of memory addresses	Maximum number of operands allowed	Type of architecture	Examples
0	3	Register-register	Alpha, ARM, MIPS, PowerPC, SPARC, SuperH, Trimedia TM5200
1	2	Register-memory	IBM 360/370, Intel 80x86, Motorola 68000, TI TMS320C54x
2	2	Memory-memory	VAX (also has three-operand formats)
3	3	Memory-memory	VAX (also has two-operand formats)

FIGURE 2.3 Typical combinations of memory operands and total operands per typical ALU instruction with examples of computers. Computers with no memory reference per ALU instruction are called load-store or register-register computers. Instructions with multiple memory operands per typical ALU instruction are called register-memory or memory-memory, according to whether they have one or more than one memory operand.

Type	Advantages	Disadvantages
Register-register (0,3)	Simple, fixed-length instruction encoding. Simple code-generation model. Instructions take similar numbers of clocks to execute (see App. A).	Higher instruction count than architectures with memory references in instructions. More instructions and lower instruction density leads to larger programs.
Register-memory (1,2)	Data can be accessed without a separate load instruction first. Instruction format tends to be easy to encode and yields good density.	Operands are not equivalent since a source operand in a binary operation is destroyed. Encoding a register number and a memory address in each instruction may restrict the number of registers. Clocks per instruction vary by operand location.
Memory-memory (2,2) or (3,3)	Most compact. Doesn't waste registers for temporaries.	Large variation in instruction size, especially for three-operand instructions. In addition, large variation in work per instruction. Memory accesses create memory bottleneck. (Not used today.)

FIGURE 2.4 Advantages and disadvantages of the three most common types of general-purpose register computers. The notation (*m, n*) means *m* memory operands and *n* total operands. In general, computers with fewer alternatives simplify the compiler's task since there are fewer decisions for the compiler to make (see section 2.11). Computers with a wide variety of flexible instruction formats reduce the number of bits required to encode the program. The number of registers also affects the instruction size since you need $\log_2$ (number of registers) for each register specifier in an instruction. Thus, doubling the number of registers takes 3 extra bits for a register-register architecture, or about 10% of a 32-bit instruction.

Figure 2.4 shows the advantages and disadvantages of each of these alternatives. Of course, these advantages and disadvantages are not absolutes: They are qualitative and their actual impact depends on the compiler and implementation strategy. A GPR computer with memory-memory operations could easily be ig-

nored by the compiler and used as a register-register computer. One of the most pervasive architectural impacts is on instruction encoding and the number of instructions needed to perform a task. We will see the impact of these architectural alternatives on implementation approaches in Chapters 3 and 4.

Summary: Classifying Instruction Set Architectures

Here and at the end of sections 2.3 to 2.11 we summarize those characteristics we would expect to find in a new instruction set architecture, building the foundation for the MIPS architecture introduced in section 2.12. From this section we should clearly expect the use of general-purpose registers. Figure 2.4, combined with Appendix A on pipelining, lead to the expectation of a register-register (also called load-store) version of a general-purpose register architecture.

With the class of architecture covered, the next topic is addressing operands.

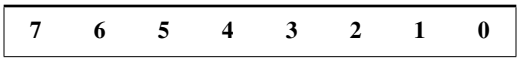
2.3 Memory Addressing

Independent of whether the architecture is register-register or allows any operand to be a memory reference, it must define how memory addresses are interpreted and how they are specified. The measurements presented here are largely, but not completely, computer independent. In some cases the measurements are significantly affected by the compiler technology. These measurements have been made using an optimizing compiler, since compiler technology plays a critical role.

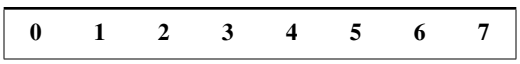
Interpreting Memory Addresses

How is a memory address interpreted? That is, what object is accessed as a function of the address and the length? All the instruction sets discussed in this book—except some DSPs—are byte addressed and provide access for bytes (8 bits), half words (16 bits), and words (32 bits). Most of the computers also provide access for double words (64 bits).

There are two different conventions for ordering the bytes within a larger object. *Little Endian* byte order puts the byte whose address is "x...x000" at the least-significant position in the double word (the little end). The bytes are numbered:



Big Endian byte order puts the byte whose address is "x...x000" at the most-significant position in the double word (the big end). The bytes are numbered:



When operating within one computer, the byte order is often unnoticeable—only programs that access the same locations as both, say, words and bytes can notice the difference. Byte order is a problem when exchanging data among computers with different orderings, however. Little Endian ordering also fails to match normal ordering of words when strings are compared. Strings appear "SDRAWKCAB" (backwards) in the registers.

A second memory issue is that in many computers, accesses to objects larger than a byte must be *aligned*. An access to an object of size s bytes at byte address A is aligned if $A \bmod s = 0$. Figure 2.5 shows the addresses at which an access is aligned or misaligned.

Width of object:	Value of 3 low order bits of byte address: 0	1	2	3	4	5	6	7
1 Byte (Byte)	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned
2 Bytes (Half word)	Aligned		Aligned		Aligned		Aligned	
2 Bytes (Half word)		Misaligned		Misaligned		Misaligned		Misalig.
4 Bytes (Word)	Aligned				Aligned			
4 Bytes (Word)		Misaligned				Misaligned		
4 Bytes (Word)			Misaligned				Misaligned	
4 Bytes (Word)				Misaligned				Misalig.
8 bytes (Double word)	Aligned							
8 bytes (Double word)		Misaligned						
8 bytes (Double word)			Misaligned					
8 bytes (Double word)				Misaligned				
8 bytes (Double word)					Misaligned			
8 bytes (Double word)						Misaligned		
8 bytes (Double word)							Misaligned	
8 bytes (Double word)								Misalig.

FIGURE 2.5 Aligned and misaligned addresses of byte, half word, word, and double word objects for byte addressed computers. For each misaligned example some objects require two memory accesses to complete. Every aligned object can always complete in one memory access, as long as the memory is as wide as the object. The figure shows the memory organized as 8 bytes wide. The byte offsets that label the columns specify the low-order three bits of the address.

Why would someone design a computer with alignment restrictions? Misalignment causes hardware complications, since the memory is typically aligned on a multiple of a word or double-word boundary. A misaligned memory access may, therefore, take multiple aligned memory references. Thus, even in computers that allow misaligned access, programs with aligned accesses run faster.

Even if data are aligned, supporting byte, half-word, and word accesses requires an alignment network to align bytes, half words, and words in 64-bit registers. For example, in Figure 2.5 above, suppose we read a byte from an address with its three low order bits having the value 4. We will need shift right 3 bytes to align the byte to the proper place in a 64-bit register. Depending on the instruction, the computer may also need to sign-extend the quantity. Stores are easy: only the addressed bytes in memory may be altered. On some computers a byte, half word, and word operation does not affect the upper portion of a register. Although all the computers discussed in this book permit byte, half-word, and word accesses to memory, only the IBM 360/370, Intel 80x86, and VAX supports ALU operations on register operands narrower than the full width.

Now that we have discussed alternative interpretations of memory addresses, we can discuss the ways addresses are specified by instructions, called *addressing modes*.

Addressing Modes

Given an address, we now know what bytes to access in memory. In this subsection we will look at addressing modes—how architectures specify the address of an object they will access. Addressing mode specify constants and registers in addition to locations in memory. When a memory location is used, the actual memory address specified by the addressing mode is called the *effective address*.

Figure 2.6 above shows all the data-addressing modes that have been used in recent computers. Immediates or literals are usually considered memory-addressing modes (even though the value they access is in the instruction stream), although registers are often separated. We have kept addressing modes that depend on the program counter, called *PC-relative addressing*, separate. PC-relative addressing is used primarily for specifying code addresses in control transfer instructions, discussed in section 2.9.

Figure 2.6 shows the most common names for the addressing modes, though the names differ among architectures. In this figure and throughout the book, we will use an extension of the C programming language as a hardware description notation. In this figure, only one non-C feature is used: The left arrow (←) is used for assignment. We also use the array `Mem` as the name for main memory and the array `Regs` for registers. Thus, `Mem[Regs[R1]]` refers to the contents of the memory location whose address is given by the contents of register 1 (`R1`). Later, we will introduce extensions for accessing and transferring data smaller than a word.

Addressing modes have the ability to significantly reduce instruction counts; they also add to the complexity of building a computer and may increase the average CPI (clock cycles per instruction) of computers that implement those modes. Thus, the usage of various addressing modes is quite important in helping the architect choose what to include.

Figure 2.7 above shows the results of measuring addressing mode usage patterns in three programs on the VAX architecture. We use the old VAX architec-

Addressing mode	Example instruction	Meaning	When used
Register	`Add R4,R3`	`Regs[R4]←Regs[R4] + Regs[R3]`	When a value is in a register.
Immediate	`Add R4,#3`	`Regs[R4]←Regs[R4]+3`	For constants.
Displacement	`Add R4,100(R1)`	`Regs[R4]←Regs[R4] + Mem[100+Regs[R1]]`	Accessing local variables (+ simulates register indirect, direct addressing modes)
Register indirect	`Add R4,(R1)`	`Regs[R4]←Regs[R4] + Mem[Regs[R1]]`	Accessing using a pointer or a computed address.
Indexed	`Add R3,(R1 + R2)`	`Regs[R3]←Regs[R3] +Mem[Regs[R1]+Regs[R2]]`	Sometimes useful in array addressing: `R1` = base of array; `R2` = index amount.
Direct or absolute	`Add R1,(1001)`	`Regs[R1]←Regs[R1] + Mem[1001]`	Sometimes useful for accessing static data; address constant may need to be large.
Memory indirect	`Add R1,@(R3)`	`Regs[R1]←Regs[R1] + Mem[Mem[Regs[R3]]]`	If `R3` is the address of a pointer p, then mode yields $*p$.
Autoincrement	`Add R1,(R2)+`	`Regs[R1]←Regs[R1] + Mem[Regs[R2]]` `Regs[R2]←Regs[R2]+`d	Useful for stepping through arrays within a loop. `R2` points to start of array; each reference increments `R2` by size of an element, d.
Autodecrement	`Add R1,-(R2)`	`Regs[R2]←Regs[R2]-`d `Regs[R1]←Regs[R1] + Mem[Regs[R2]]`	Same use as autoincrement. Autodecrement/increment can also act as push/pop to implement a stack.
Scaled	`Add R1,100(R2)[R3]`	`Regs[R1]← Regs[R1]+ Mem[100+Regs[R2] + Regs[R3]*`d`]`	Used to index arrays. May be applied to any indexed addressing mode in some computers.

FIGURE 2.6 Selection of addressing modes with examples, meaning, and usage. In autoincrement/decrement and scaled addressing modes, the variable d designates the size of the data item being accessed (i.e., whether the instruction is accessing 1, 2, 4, or 8 bytes). These addressing modes are only useful when the elements being accessed are adjacent in memory. RISC computers use Displacement addressing to simulate Register Indirect with 0 for the address and simulate Direct addressing using 0 in the base register. In our measurements, we use the first name shown for each mode. The extensions to C used as hardware descriptions are defined on the next page, also on page 144, and on the back inside cover.

ture for a few measurements in this chapter because it has the richest set of addressing modes and fewest restrictions on memory addressing. For example, Figure 2.6 shows all the modes the VAX supports. Most measurements in this chapter, however, will use the more recent register-register architectures to show how programs use instruction sets of current computers.

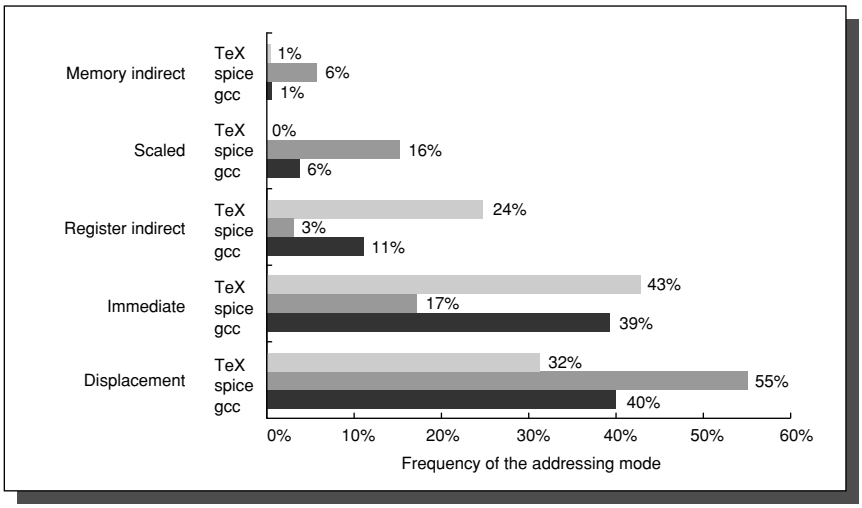


FIGURE 2.7 Summary of use of memory addressing modes (including immediates). These major addressing modes account for all but a few percent (0% to 3%) of the memory accesses. Register modes, which are not counted, account for one-half of the operand references, while memory addressing modes (including immediate) account for the other half. Of course, the compiler affects what addressing modes are used; see section 2.11. The memory indirect mode on the VAX can use displacement, autoincrement, or autodecrement to form the initial memory address; in these programs, almost all the memory indirect references use displacement mode as the base. Displacement mode includes all displacement lengths (8, 16, and 32 bit). The PC-relative addressing modes, used almost exclusively for branches, are not included. Only the addressing modes with an average frequency of over 1% are shown. The data are from a VAX using three SPEC89 programs.

As Figure 2.7 shows, immediate and displacement addressing dominate addressing mode usage. Let's look at some properties of these two heavily used modes.

Displacement Addressing Mode

The major question that arises for a displacement-style addressing mode is that of the range of displacements used. Based on the use of various displacement sizes, a decision of what sizes to support can be made. Choosing the displacement field sizes is important because they directly affect the instruction length. Figure 2.8 shows the measurements taken on the data access on a load-store architecture using our benchmark programs. We look at branch offsets in section 2.9—data accessing patterns and branches are different; little is gained by combining them, although in practice the immediate sizes are made the same for simplicity.

Immediate or Literal Addressing Mode

Immediates can be used in arithmetic operations, in comparisons (primarily for branches), and in moves where a constant is wanted in a register. The last case oc-

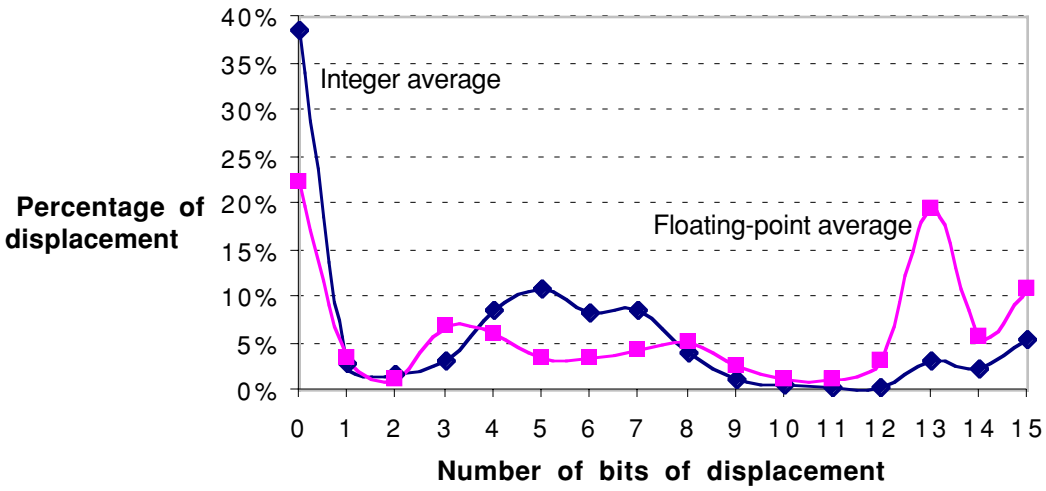


FIGURE 2.8 Displacement values are widely distributed. There are both a large number of small values and a fair number of large values. The wide distribution of displacement values is due to multiple storage areas for variables and different displacements to access them (see section 2.11) as well as the overall addressing scheme the compiler uses. The x axis is $\log_2$ of the displacement; that is, the size of a field needed to represent the magnitude of the displacement. Zero on the x axis shows the percentage of displacements of value 0. The graph does not include the sign bit, which is heavily affected by the storage layout. Most displacements are positive, but a majority of the largest displacements (14+ bits) is negative. Since this data was collected on a computer with 16-bit displacements, it cannot tell us about longer displacements. These data were taken on the Alpha architecture with full optimization (see section 2.11) for SPEC CPU2000, showing the average of integer programs (CINT2000) and the average of floating-point programs (CFP2000).

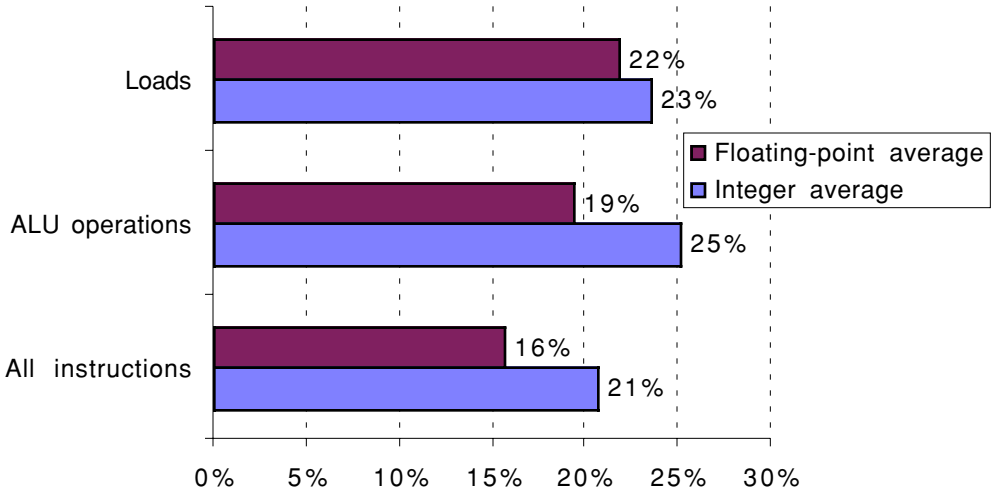


FIGURE 2.9 About one-quarter of data transfers and ALU operations have an immediate operand. The bottom bars show that integer programs use immediates in about one-fifth of the instructions, while floating-point programs use immediates in about one-sixth of the instructions. For loads, the load immediate instruction loads 16 bits into either half of a 32-bit register. Load immediates are not loads in a strict sense because they do not access memory. Occasionally a pair of load immediates is used to load a 32-bit constant, but this is rare. (For ALU operations, shifts by a constant amount are included as operations with immediate operands.) These measurements as in Figure 2.8.

curs for constants written in the code–which tend to be small–and for address constants, which tend to be large. For the use of immediates it is important to know whether they need to be supported for all operations or for only a subset. The chart in Figure 2.9 shows the frequency of immediates for the general classes of integer operations in an instruction set.

Another important instruction set measurement is the range of values for immediates. Like displacement values, the size of immediate values affects instruction length. As Figure 2.10 shows, small immediate values are most heavily used. Large immediates are sometimes used, however, most likely in addressing calculations.

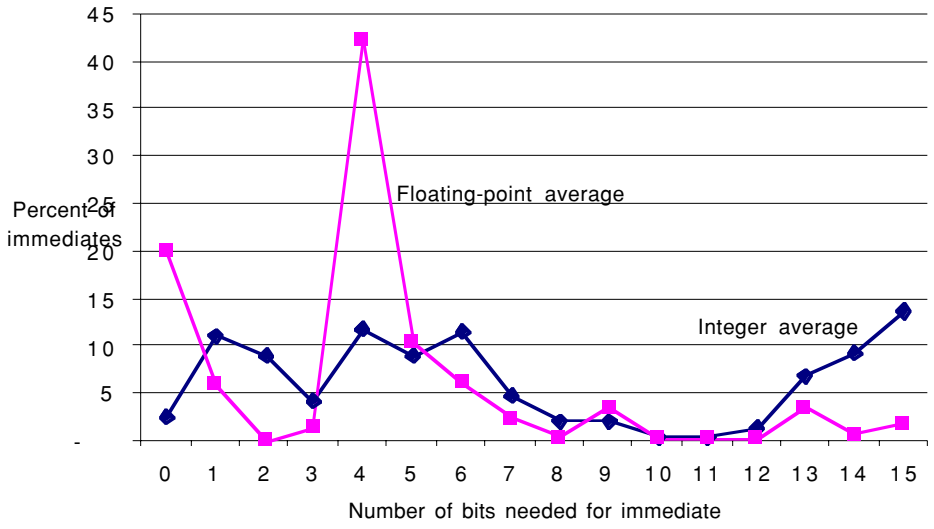


FIGURE 2.10 The distribution of immediate values. The *x* axis shows the number of bits needed to represent the magnitude of an immediate value—0 means the immediate field value was 0. The majority of the immediate values are positive. About 20% were negative for CINT2000 and about 30% were negative for CFP2000. These measurements were taken on a Alpha, where the maximum immediate is 16 bits, for the same programs as in Figure 2.8. A similar measurement on the VAX, which supported 32-bit immediates, showed that about 20% to 25% of immediates were longer than 16 bits.

2.4 Addressing Modes for Signal Processing

To give a flavor of the different perspective between different architecture cultures, here are two addressing modes that distinguish DSPs.

Since DSPs deal with infinite, continuous streams of data, they routinely rely on circular buffers. Hence, as data is added to the buffer, a pointer is checked to see if it is pointing at the end of the buffer. If not, it increments the pointer to the next address; if it is, the pointer is set instead to the start of the buffer. Similar issues arise when emptying a buffer.

Every recent DSP has a *modulo* or *circular* addressing mode to handle this case automatically, our first novel DSP addressing mode. It keeps a start register and an end register with every address register, allowing the autoincrement and autodecrement addressing modes to reset when the reach the end of the buffer. One variation makes assumptions about the buffer size starting at an address that ends in "xxx00.00" and so uses just a single buffer length register per address register,

Even though DSPs are tightly targeted to a small number of algorithms, its surprising this next addressing mode is included for just one application: Fast Fourier Transform (FFT). FFTs start or end their processing with data shuffled in a particular order. For eight data items in a radix-2 FFT, the transformation is listed below, with addresses in parentheses shown in binary:

0 (000_2)	=>	0 (000_2)
1 (001_2)	=>	4 (100_2)
2 (010_2)	=>	2 (010_2)
3 (011_2)	=>	6 (110_2)
4 (100_2)	=>	1 (001_2)
5 (101_2)	=>	5 (101_2)
6 (110_2)	=>	3 (011_2)
7 (111_2)	=>	7 (111_2)

Without special support such address transformation would take an extra memory access to get the new address, or involve a fair amount of logical instructions to transform the address.

The DSP solution is based on the observation that the resulting binary address is simply the reverse of the initial address! For example, address 100_2 (4) becomes 001_2(1). Hence, many DSPs have this second novel addressing mode— *bit reverse* addressing—whereby the hardware reverses the lower bits of the address, with the number of bits reversed depending on the step of the FFT algorithm.

As DSP programmers migrate towards larger programs and hence become more attracted to compilers, they have been trying to use the compiler technology developed for the desktop and embedded computers. Such compilers have no hope of taking high-level language code and producing these two addressing modes, so they are limited to assembly language programmer. As stated before, the DSP community routinely uses library routines, and hence programmers may benefit even if *they* write at a higher level.

Figure 2.11 shows the static frequency of data addressing modes in a DSP for a set of 54 library routines. This architecture has 17 addressing modes, yet the 6 modes also found in Figure 2.6 on page 108 for desktop and server computers account for 95% of the DSP addressing. Despite measuring hand-coded routines to derive Figure 2.11, the use of novel addressing mode is sparse.

These results are just for one library for just one DSP, other libraries might use more addressing modes, and static and dynamic frequencies may differ. Yet Figure 2.11 still makes the point that there is often a mismatch between what programmers and compilers actually use versus what architects expect, and this is just as true for DSPs as it is for more traditional processors.

Addressing Mode	Assembly Symbol	Percent
Immediate	#num	30.02%
Displacement	ARx(num)	10.82%
Register indirect	*ARx	17.42%
Direct	num	11.99%
Autoincrement, pre increment (increment register *before* use contents as address)	*+ARx	0
Autoincrement, post increment (increment register *after* use contents as address)	*ARx+	18.84%
Autoincrement, pre increment with 16b immediate	*+ARx(num)	0.77%
Autoincrement, pre increment, with circular addressing	*ARx+%	0.08%
Autoincrement, post increment with 16b immediate, with circular addressing	*ARx+(num)%	0
Autoincrement, post increment by contents of AR0	*ARx+0	1.54%
Autoincrement, post increment by contents of AR0, with circular addressing	*ARx+0%	2.15%
Autoincrement, post increment by contents of AR0, with bit reverse addressing	*ARx+0B	0
Autodecrement, post decrement (decrement register *after* use contents as address	*ARx-	6.08%
Autodecrement, post decrement, with circular addressing	*ARx-%	0.04%
Autodecrement, post decrement by contents of AR0	*ARx-0	0.16%
Autodecrement, post decrement by contents of AR0, with circular addressing	*ARx-0%	0.08%
Autodecrement, post decrement by contents of AR0, with bit reverse addressing	*ARx-0B	0
Total		100.00%

FIGURE 2.11 Frequency of addressing modes for TI TMS320C54x DSP. The C54x has 17 data addressing modes, not counting register access, but the four found in MIPS account for 70% of the modes. Autoincrement and autodecrement, found in some RISC architectures, account for another 25% of the usage. This data was collected form a measurement of static instructions for the C-callable library of 54 DSP routines coded in assembly language. See http://www.ti.com/sc/docs/products/dsp/c5000/c54x/54dsplib.htm

Summary: Memory Addressing

First, because of their popularity, we would expect a new architecture to support at least the following addressing modes: displacement, immediate, and register indirect. Figure 2.7 on page 109 shows they represent 75% to 99% of the addressing modes used in our SPEC measurements. Second, we would expect the size of the address for displacement mode to be at least 12 to 16 bits, since the caption in Figure 2.8 on page 110 suggests these sizes would capture 75% to

99% of the displacements. Third, we would expect the size of the immediate field to be at least 8 to 16 bits. As the caption in Figure 2.10 suggests, these sizes would capture 50% to 80% of the immediates.

Desktop and server processors rely on compilers and so addressing modes must match the ability of the compilers to use them, while historically DSPs rely on hand-coded libraries to exercise novel addressing modes. Even so, there are times when programmers find they do not need the clever tricks that architects thought would be useful—or tricks that other programmers promised that they would use. As DSPs head towards relying even more on compiled code, we expect increasing emphasis on simpler addressing modes.

Having covered instruction set classes and decided on register-register architectures plus the recommendations on data addressing modes above, we next cover the sizes and meanings of data.

2.5 Type and Size of Operands

How is the type of an operand designated? Normally, encoding in the opcode designates the type of an operand—this is the method used most often. Alternatively, the data can be annotated with tags that are interpreted by the hardware. These tags specify the type of the operand, and the operation is chosen accordingly. Computers with tagged data, however, can only be found in computer museums.

Let's start with desktop and server architectures. Usually the type of an operand—integer, single-precision floating point, character, and so on—effectively gives its size. Common operand types include character (8 bits), half word (16 bits), word (32 bits), single-precision floating point (also 1 word), and double-precision floating point (2 words). Integers are almost universally represented as two's complement binary numbers. Characters are usually in ASCII, but the 16-bit Unicode (used in Java) is gaining popularity with the internationalization of computers. Until the early 1980s, most computer manufacturers chose their own floating-point representation. Almost all computers since that time follow the same standard for floating point, the IEEE standard 754. The IEEE floating-point standard is discussed in detail in Appendix G <Float>.

Some architectures provide operations on character strings, although such operations are usually quite limited and treat each byte in the string as a single character. Typical operations supported on character strings are comparisons and moves.

For business applications, some architectures support a decimal format, usually called *packed decimal* or *binary-coded decimal*—4 bits are used to encode the values 0–9, and 2 decimal digits are packed into each byte. Numeric character strings are sometimes called *unpacked decimal,* and operations—called *packing* and *unpacking*—are usually provided for converting back and forth between them.

One reason to use decimal operands is to get results that exactly match decimal numbers, as some decimal fractions do not have an exact representation in binary. For example, 0.10_{10} is a simple fraction in decimal but in binary it requires an infinite set of repeating digits: $0.0001100110\overline{0011}..._2$. Thus, calculations that are exact in decimal can be close but inexact in binary, which can be a problem for financial transactions. (See Appendix G <Float> to learn more about precise arithmetic.)

Our SPEC benchmarks use byte or character, half word (short integer), word (integer), double word (long integer) and floating-point data types. Figure 2.12 shows the dynamic distribution of the sizes of objects referenced from memory for these programs. The frequency of access to different data types helps in deciding what types are most important to support efficiently. Should the computer have a 64-bit access path, or would taking two cycles to access a double word be satisfactory? As we saw earlier, byte accesses require an alignment network: How important is it to support bytes as primitives? Figure 2.12 uses memory references to examine the types of data being accessed.

In some architectures, objects in registers may be accessed as bytes or half words. However, such access is very infrequent—on the VAX, it accounts for no more than 12% of register references, or roughly 6% of all operand accesses in these programs.

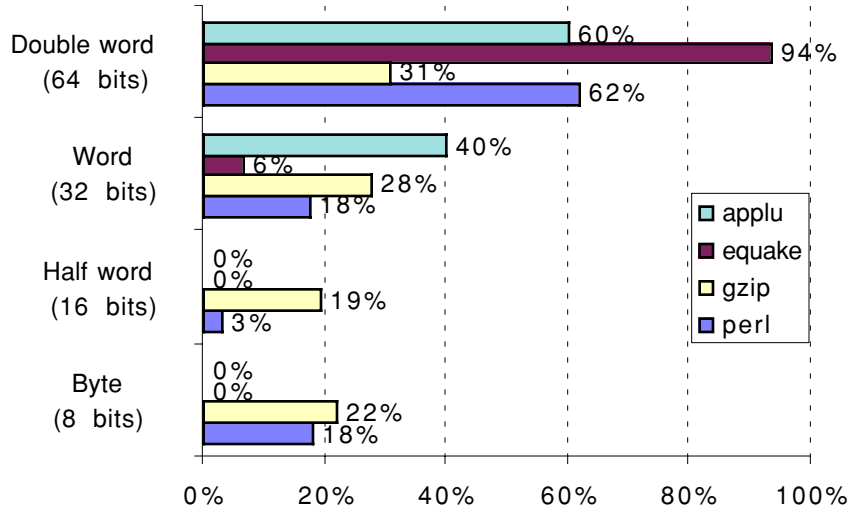


FIGURE 2.12 Distribution of data accesses by size for the benchmark programs. The double word data type is used for double-precision floating-point in floating-point programs and for addresses, since the computer uses 64-bit addresses. On a 32-bit address computer the 64-bit addresses would be replaced by 32-bit addresses, and so almost all double-word accesses in integer programs would become single word accesses.

2.6 Operands for Media and Signal Processing

Graphics applications deal with 2D and 3D images. A common 3D data type is called a *vertex*, a data structure with three components: x coordinate, y coordinate, a z coordinate, and a fourth coordinate (w) to help with color or hidden surfaces. Three vertices specify a graphics primitive such as a triangle. Vertex values are usually 32-bit floating-point values.

Assuming a triangle is visible, when it is rendered it is filled with pixels. *Pixels* are usually 32 bits, usually consisting of four 8-bit channels: R (red), G (green), B (blue) and A (which denotes the transparency of the surface or transparency of the pixel when the pixel is rendered).

DSPs add *fixed point* to the data types discussed so far. If you think of integers as having a binary point to the right of the least significant bit, fixed point has a binary point just to the right of the sign bit. Hence, fixed-point data are fractions between -1 and +1.

EXAMPLE Here are three simple16-bit patterns:

0100 0000 0000 0000

0000 1000 0000 0000

0100 1000 0000 1000

What values do they represent if they are two's complement integers? Fixed-point numbers?

ANSWER Number representation tells us that the i-th digit to the left of the binary point represents 2^{i-1} and the i-th digit to the right of the binary point represents 2^{-i}. First assume these three patterns are integers. Then the binary point is to the far right, so they represent 2^{14}, 2^{11}, and (2^{14}+ 2^{11}+ 2^{3}), or 16384, 2048, and 18440.

Fixed point places the binary point just to the right of the sign bit, so as fixed point these patterns represent 2^{-1}, 2^{-4}, and (2^{-1}+ 2^{-4} + 2^{-12}). The fractions are 1/2, 1/16, and (2048 + 256 + 1)/4096 or 2305/4096,which represents about 0.50000, 0.06250, and 0.56274. Alternatively, for an n-bit two's-complement, fixed-point number we could just use the divide the integer presentation by the 2^{n-1} to derive the same results:
16384/32768=1/2, 2048/32768=1/16, and 18440/32768=2305/4096.

Fixed point can be thought of as just low cost floating point. It doesn't include an exponent in every word and have hardware that automatically aligns and normalizes operands. Instead, fixed point relies on the DSP programmer to keep the

exponent in a separate variable and ensure that each result is shifted left or right to keep the answer aligned to that variable. Since this exponent variable is often shared by a set of fixed-point variables, this style of arithmetic is also called *blocked floating point*, since a block of variables have a common exponent

To support such manual calculations, DSPs usually have some registers that are wider to guard against round-off error, just as floating-point units internally have extra guard bits. Figure 2.13 surveys four generations of DSPs, listing data sizes and width of the accumulating registers. Note that DSP architects are not bound by the powers of 2 for word sizes. Figure 2.14 shows the size of data operands for the TI TMS320C540x DSP.

Generation	Year	Example DSP	Data Width	Accumulator Width
1	1982	TI TMS32010	16 bits	32 bits
2	1987	Motorola DSP56001	24 bits	56 bits
3	1995	Motorola DSP56301	24 bits	56 bits
4	1998	TI TMS320C6201	16 bits	40 bits

FIGURE 2.13 Four generations of DSPs, their data width, and the width of the registers that reduces round-off error. Section 2.8 explains that multiply-accumulate operations use wide registers to avoid loosing precision when accumulating double-length products [Bier 1997].

Data Size	Memory Operand in Operation	Memory Operand in Data Transfer
16 bits	89.3%	89.0%
32 bits	10.7%	11.0%

FIGURE 2.14 Size of data operands for TMS320C540x DSP. About 90% of operands are 16 bits. This DSP has two 40-bit accumulators. There are no floating-point operations, as is typical of many DSPs, so these data are all fixed-point integers. For details on these measurements, see the caption of Figure 2.11 on page 113.

Summary: Type and Size of Operands

From this section we would expect a new 32-bit architecture to support 8-, 16-, and 32-bit integers and 32-bit and 64-bit IEEE 754 floating-point data. A new 64-bit address architecture would need to support 64-bit integers as well. The level of support for decimal data is less clear, and it is a function of the intended use of the computer as well as the effectiveness of the decimal support. DSPs need wider accumulating registers than the size in memory to aid accuracy in fixed-point arithmetic.

We have reviewed instruction set classes and chosen the register-register class, reviewed memory addressing and selected displacement, immediate, and register indirect addressing modes, and selected the operand sizes and types above. Now we are ready to look at instructions that do the heavy lifting in the architecture.

2.7 Operations in the Instruction Set

The operators supported by most instruction set architectures can be categorized as in Figure 2.15. One rule of thumb across all architectures is that the most widely executed instructions are the simple operations of an instruction set. For example Figure 2.16 shows 10 simple instructions that account for 96% of instructions executed for a collection of integer programs running on the popular Intel 80x86. Hence, the implementor of these instructions should be sure to make these fast, as they are the common case.

Operator type	Examples
Arithmetic and logical	Integer arithmetic and logical operations: add, subtract, and, or, multiple, divide
Data transfer	Loads-stores (move instructions on computers with memory addressing)
Control	Branch, jump, procedure call and return, traps
System	Operating system call, virtual memory management instructions
Floating point	Floating-point operations: add, multiply, divide, compare
Decimal	Decimal add, decimal multiply, decimal-to-character conversions
String	String move, string compare, string search
Graphics	Pixel and vertex operations, compression/decompression operations

FIGURE 2.15 Categories of instruction operators and examples of each. All computers generally provide a full set of operations for the first three categories. The support for system functions in the instruction set varies widely among architectures, but all computers must have some instruction support for basic system functions. The amount of support in the instruction set for the last four categories may vary from none to an extensive set of special instructions. Floating-point instructions will be provided in any computer that is intended for use in an application that makes much use of floating point. These instructions are sometimes part of an optional instruction set. Decimal and string instructions are sometimes primitives, as in the VAX or the IBM 360, or may be synthesized by the compiler from simpler instructions. Graphics instructions typically operate on many smaller data items in parallel; for example, performing eight 8-bit additions on two 64-bit operands.

As mentioned before, the instructions in Figure 2.16 are found in every computer for every application—desktop, server, embedded—with the variations of operations in Figure 2.15 largely depending on which data types that the instruction set includes.

2.8 Operations for Media and Signal Processing

Because media processing is judged by human perception, the data for multimedia operations is often much narrower than the 64-bit data word of modern desktop and server processors. For example, floating-point operations for graphics are normally in single precision, not double precision, and often at precession less than required by IEEE 754. Rather than waste the 64-bit ALUs when operating on 32-bit, 16-bit, or even 8-bit integers, multimedia instructions can operate on

Rank	80x86 instruction	Integer average (% total executed)
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
Total		96%

FIGURE 2.16 The top 10 instructions for the 80x86. Simple instructions dominate this list, and are responsible for 96% of the instructions executed. These percentages are the average of the five SPECint92 programs.

several narrower data items at the same time. Thus, a *partitioned add* operation on 16-bit data with a 64-bit ALU would perform four 16-bit adds in a single clock cycle. The extra hardware cost is simply to prevent carries between the four 16-bit partitions of the ALU. For example, such instructions might be used for graphical operations on pixels.

These operations are commonly called *Single-Instruction Multiple Data* (*SIMD*) or *vector* instructions. Chapters 6 and Appendix F <vector> describe the full machines that pioneered these architectures.

Most graphics multimedia applications use 32-bit floating-point operations. Some computers double peak performance of single-precision, floating-point operations; they allow a single instruction to launch two 32-bit operations on operands found side-by-side in a double precision register. Just as in the prior case, the two partitions must be insulated to prevent operations on one half to affect the other. Such floating-point operations are called *paired-single operations*. For example, such an operation might be used to graphical transformations of vertices. This doubling in performance is typically accomplished by doubling the number of floating-point units, making it more expensive than just suppressing carries in integer adders.

Figure 2.17 summaries the SIMD multimedia instructions found in several recent computers..

DSP operations

DSPs also provide operations found in the first three rows of Figure 2.15, but they change the semantics a bit. First, because they are often used in real time ap-

Instruction category	Alpha MAX	HP PA-RISC MAX2	Intel Pentium MMX	Power PC AltiVec	SPARC VIS
Add/subtract		4H	8B,4H,2W	16B, 8H, 4W	4H,2W
Saturating add/sub		4H	8B,4H	16B, 8H, 4W	
Multiply			4H	16B, 8H	
Compare	8B (>=)		8B,4H,2W (=,>)	16B, 8H, 4W (=,>,>=,<,<=)	4H,2W (=,not=,>,<=)
Shift right/left		4H	4H,2W	16B, 8H, 4W	
Shift right arithmetic		4H		16B, 8H, 4W	
Multiply and add				8H	
Shift and add (saturating)		4H			
And/or/xor	8B,4H,2W	8B,4H,2W	8B,4H,2W	16B, 8H, 4W	8B,4H,2W
Absolute difference	8B			16B, 8H, 4W	8B
Maximum/minimum	8B, 4W			16B, 8H, 4W	
Pack (2n bits --> n bits)	2W->2B, 4H->4B	2*4H->8B	4H->4B, 2W->2H	4W->4B, 8H->8B	2W->2H, 2W->2B, 4H->4B
Unpack/merge	2B->2W, 4B->4H		2B->2W, 4B->4H	4B->4W, 8B->8H	4B->4H, 2*4B->8B
Permute/shuffle		4H		16B, 8H, 4W	

FIGURE 2.17 Summary of multimedia support for desktop RISCs. Note the diversity of support, with little in common across the five architectures. All are fixed width operations, performing multiple narrow operations on either a 64-bit or 128-bit ALU. B stands for byte (8 bits), H for halfword (16 bits), and W for word (32 bits). Thus, 8B means an operation on 8 bytes in a single instruction. Note that AltiVec assume a128-bit ALU, and the rest assume 64 bits. Pack and unpack use the notation 2*2W to mean 2 operands each with 2 words. This table is a simplification of the full multimedia architectures, leaving out many details. For example, HP MAX2 includes an instruction to calculate averages, and SPARC VIS includes instructions to set registers to constants. Also, this table does not include the memory alignment operation of AltiVec, MAX and VIS

plications, there is not an option of causing an exception on arithmetic overflow (otherwise it could miss an event); thus, the result will be used no matter what the inputs. To support such an unyielding environment, DSP architectures use *saturating arithmetic*: if the result is too large to be represented, it is set to the largest representable number, depending on the sign of the result. In contrast, two's complement arithmetic can add a small positive number to a large positive number and end up with a negative result. DSP algorithms rely on saturating arithmetic, and would be incorrect if run on a computer without it.

A second issue for DSPs is that there are several modes to round the wider accumulators into the narrower data words, just as the IEEE 754 has several rounding modes to chose from.

Finally, the targeted kernels for DSPs accumulate a series of products, and hence have a *multiply-accumulate* or *MAC instruction*. MACs are key to dot product operations for vector and matrix multiplies. In fact, MACs/second is the primary peak-performance metric that DSP architects brag about. The wide accumulators are used primarily to accumulate products, with rounding used when transferring results to memory.

Instruction	Percent
store mem16	32.2%
load mem16	9.4%
add mem16	6.8%
call	5.0%
push mem16	5.0%
subtract mem16	4.9%
multiple-accumulate (MAC) mem16	4.6%
move mem-mem 16	4.0%
change status	3.7%
pop mem16	2.8%
conditional branch	2.6%
load mem32	2.5%
return	2.5%
store mem32	2.0%
branch	2.0%
repeat	2.0%
multiply	1.8%
NOP	1.5%
add mem32	1.3%
subtract mem32	0.9%
Total	97.2%

FIGURE 2.18 Mix of instructions for TMS320C540x DSP. As in Figure 2.16, simple instructions dominate this list of most frequent instructions. Mem16 stands for a 16-bit memory operand and mem32 stands for a 32-bit memory operand. The large number of change status instructions is to set mode bits to affect instructions, essentially saving opcode space in these 16-bit instructions by keeping some of it in a status register. For example, status bits determine whether 32-bit operations operate in SIMD mode to produce16-bit results in parallel or act as a single 32-bit result. For details on these measurements, see the caption of Figure 2.11 on page 113.

Figure 2.18 shows the static mix of instructions for the TI TMS320C540x DSP for a set of library routines. This 16-bit architecture uses two 40-bit accumulators, plus a stack for passing parameters to library routines and for saving return addresses. Note that DSPs have many more multiplies and MACs than in desktop

programs. Although not shown in the figure, 15% to 20% of the multiplies and MACs round the final sum. The C54 also has 8 address registers that can be accessed via load and store instructions, as these registers are memory mapped: that is, each register also has a memory address. The larger number of stores is due in part to writing portions of the 40-bit accumulators to 16-bit words, and also to transfer between registers as their index registers also have memory addressees. There are no floating-point operations, as is typical of many DSPs, so these operations are all on fixed-point integers.

Summary: Operations in the Instruction Set

From this section we see the importance and popularity of simple instructions: load, store, add, subtract, move register-register, and, shift. DSPs add multiplies and multiply-accumulates to this simple set of primitives.

Reviewing where we are in the architecture space, we have looked at instruction classes and selected register-register. We selected displacement, immediate, and register indirect addressing and selected 8-,16-, 32-, and 64-bit integers and 32- and 64-bit floating point. For operations we emphasize the simple list mentioned above. We are now ready to show how computers make decisions.

2.9 Instructions for Control Flow

Because the measurements of branch and jump behavior are fairly independent of other measurements and applications, we now examine the use of control-flow instructions, which have little in common with operations of the prior sections.

There is no consistent terminology for instructions that change the flow of control. In the 1950s they were typically called *transfers*. Beginning in 1960 the name *branch* began to be used. Later, computers introduced additional names. Throughout this book we will use *jump* when the change in control is unconditional and *branch* when the change is conditional.

We can distinguish four different types of control-flow change:

1. Conditional branches
2. Jumps
3. Procedure calls
4. Procedure returns

We want to know the relative frequency of these events, as each event is different, may use different instructions, and may have different behavior. Figure 2.19 shows the frequencies of these control-flow instructions for a load-store computer running our benchmarks.

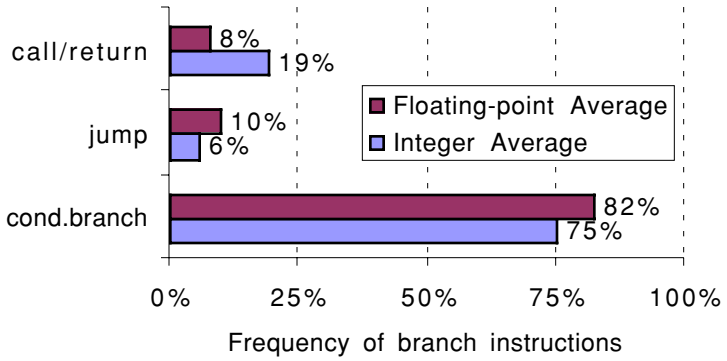


FIGURE 2.19 Breakdown of control flow instructions into three classes: calls or returns, jumps, and conditional branches. Conditional branches clearly dominate. Each type is counted in one of three bars. The programs and computer used to collect these statistics are the same as those in Figure 2.8.

Addressing Modes for Control Flow Instructions

The destination address of a control flow instruction must always be specified. This destination is specified explicitly in the instruction in the vast majority of cases—procedure return being the major exception—since for return the target is not known at compile time. The most common way to specify the destination is to supply a displacement that is added to the *program counter*, or PC. Control flow instructions of this sort are called *PC-relative*. PC-relative branches or jumps are advantageous because the target is often near the current instruction, and specifying the position relative to the current PC requires fewer bits. Using PC-relative addressing also permits the code to run independently of where it is loaded. This property, called *position independence*, can eliminate some work when the program is linked and is also useful in programs linked dynamically during execution.

To implement returns and indirect jumps when the target is not known at compile time, a method other than PC-relative addressing is required. Here, there must be a way to specify the target dynamically, so that it can change at runtime. This dynamic address may be as simple as naming a register that contains the target address; alternatively, the jump may permit any addressing mode to be used to supply the target address.

These register indirect jumps are also useful for four other important features:

1. *case* or *switch* statements found in most programming languages (which select among one of several alternatives);
2. *virtual functions* or *methods* in object-oriented languages like C++ or Java (which allow different routines to be called depending on the type of the argument);
3. *high order functions* or *function pointers* in languages like C or C++ (which allows functions to be passed as arguments giving some of the flavor of object oriented programming), and

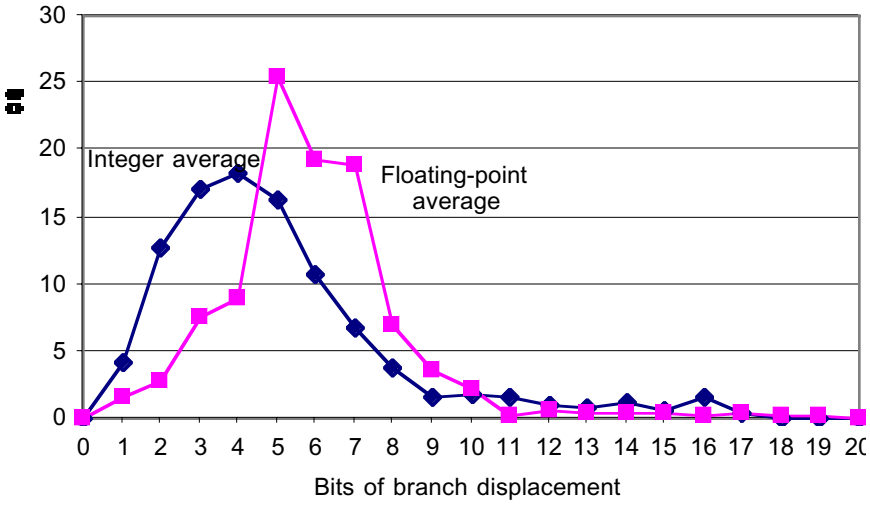


FIGURE 2.20 Branch distances in terms of number of instructions between the target and the branch instruction. The most frequent branches in the integer programs are to targets that can be encoded in four to eight bits. This result tells us that short displacement fields often suffice for branches and that the designer can gain some encoding density by having a shorter instruction with a smaller branch displacement. These measurements were taken on a load-store computer (Alpha architecture) with all instructions aligned on word boundaries. An architecture that requires fewer instructions for the same program, such as a VAX, would have shorter branch distances. However, the number of bits needed for the displacement may increase if the computer has variable length instructions to be aligned on any byte boundary. Exercise 2.1 shows the accumulative distribution of this branch displacement data (see Figure 2.42 on page 173). The programs and computer used to collect these statistics are the same as those in Figure 2.8.

4. *dynamically shared libraries* (which allow a library to be loaded and linked at runtime only when it is actually invoked by the program rather than loaded and linked statically before the program is run).

In all four cases the target address is not known at compile time, and hence is usually loaded from memory into a register before the register indirect jump.

As branches generally use PC-relative addressing to specify their targets, an important question concerns how far branch targets are from branches. Knowing the distribution of these displacements will help in choosing what branch offsets to support and thus will affect the instruction length and encoding. Figure 2.20 shows the distribution of displacements for PC-relative branches in instructions. About 75% of the branches are in the forward direction.

Conditional Branch Options

Since most changes in control flow are branches, deciding how to specify the branch condition is important. Figure 2.21 shows the three primary techniques in use today and their advantages and disadvantages.

Name	Examples	How condition is tested	Advantages	Disadvantages
Condition code (CC)	80x86, ARM, PowerPC, SPARC, SuperH	Special bits are set by ALU operations, possibly under program control.	Sometimes condition is set for free.	CC is extra state. Condition codes constrain the ordering of instructions since they pass information from one instruction to a branch.
Condition register	Alpha, MIPS	Tests arbitrary register with the result of a comparison.	Simple.	Uses up a register.
Compare and branch	PA-RISC, VAX	Compare is part of the branch. Often compare is limited to subset.	One instruction rather than two for a branch.	May be too much work per instruction for pipelined execution.

FIGURE 2.21 The major methods for evaluating branch conditions, their advantages, and their disadvantages. Although condition codes can be set by ALU operations that are needed for other purposes, measurements on programs show that this rarely happens. The major implementation problems with condition codes arise when the condition code is set by a large or haphazardly chosen subset of the instructions, rather than being controlled by a bit in the instruction. Computers with compare and branch often limit the set of compares and use a condition register for more complex compares. Often, different techniques are used for branches based on floating-point comparison versus those based on integer comparison. This dichotomy is reasonable since the number of branches that depend on floating-point comparisons is much smaller than the number depending on integer comparisons.

One of the most noticeable properties of branches is that a large number of the comparisons are simple tests, and a large number are comparisons with zero. Thus, some architectures choose to treat these comparisons as special cases, especially if a *compare and branch* instruction is being used. Figure 2.22 shows the frequency of different comparisons used for conditional branching.

DSPs add another looping structure, usually called a *repeat* instruction. It allows a single instruction or a block of instructions to be repeated up to, say, 256 times. For example, the TMS320C54 dedicates three special registers to hold the block starting address, ending address, and repeat counter. The memory instructions in a repeat loop will typically have autoincrement or autodecrement addressing to access a vector. The goal of such instructions is to avoid loop overhead, which can be significant in the small loops of DSP kernels.

Procedure Invocation Options

Procedure calls and returns include control transfer and possibly some state saving; at a minimum the return address must be saved somewhere, sometimes in a special link register or just a GPR. Some older architectures provide a mecha-

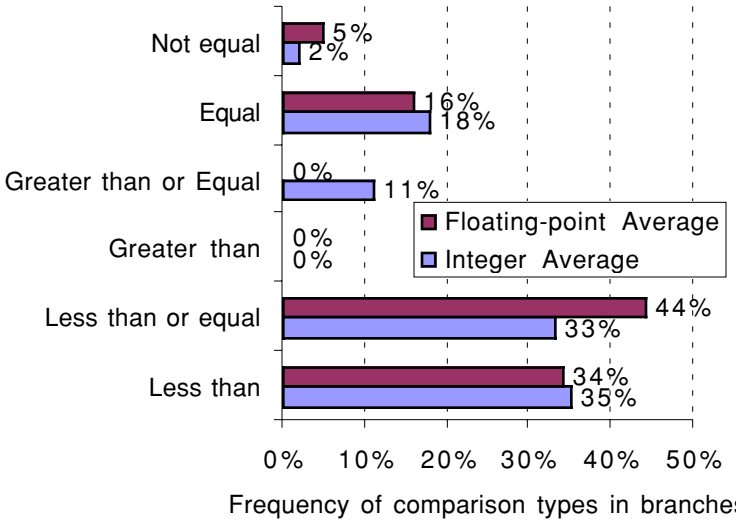


FIGURE 2.22 Frequency of different types of compares in conditional branches. Less than (or equal) branches dominate this combination of compiler and architecture. These measurements include both the integer and floating-point compares in branches. The programs and computer used to collect these statistics are the same as those in Figure 2.8

nism to save many registers, while newer architectures require the compiler to generate stores and loads for each register saved and restored.

There are two basic conventions in use to save registers: either at the call site or inside the procedure being called. *Caller saving* means that the calling procedure must save the registers that it wants preserved for access after the call, and thus the called procedure need not worry about registers. *Callee saving* is the opposite: the called procedure must save the registers it wants to use, leaving the caller is unrestrained.

There are times when caller save must be used because of access patterns to globally visible variables in two different procedures. For example, suppose we have a procedure P1 that calls procedure P2, and both procedures manipulate the global variable x. If P1 had allocated x to a register, it must be sure to save x to a location known by P2 before the call to P2. A compiler's ability to discover when a called procedure may access register-allocated quantities is complicated by the possibility of separate compilation. Suppose P2 may not touch x but can call another procedure, P3, that may access x, yet P2 and P3 are compiled separately. Because of these complications, most compilers will conservatively caller save *any* variable that may be accessed during a call.

In the cases where either convention could be used, some programs will be more optimal with callee save and some will be more optimal with caller save. As a result, the most real systems today use a combination of the two mechanisms.

This convention is specified in an application binary interface (ABI) that sets down the basic rules as to which registers should be caller saved and which should be callee saved. Later in this chapter we will examine the mismatch between sophisticated instructions for automatically saving registers and the needs of the compiler.

Summary: Instructions for Control Flow

Control flow instructions are some of the most frequently executed instructions. Although there are many options for conditional branches, we would expect branch addressing in a new architecture to be able to jump to hundreds of instructions either above or below the branch. This requirement suggests a PC-relative branch displacement of at least 8 bits. We would also expect to see register-indirect and PC-relative addressing for jump instructions to support returns as well as many other features of current systems.

We have now completed our instruction architecture tour at the level seen by assembly language programmer or compiler writer. We are leaning towards a register-register architecture with displacement, immediate, and register indirect addressing modes. These data are 8-,16-, 32-, and 64-bit integers and 32- and 64-bit floating-point data. The instructions include simple operations, PC-relative conditional branches, jump and link instructions for procedure call, and register indirect jumps for procedure return (plus a few other uses.)

Now we need to select how to represent this architecture in a form that makes it easy for the hardware to execute.

2.10 Encoding an Instruction Set

Clearly, the choices mentioned above will affect how the instructions are encoded into a binary representation for execution by the processor. This representation affects not only the size of the compiled program; it affects the implementation of the processor, which must decode this representation to quickly find the operation and its operands. The operation is typically specified in one field, called the *opcode*. As we shall see, the important decision is how to encode the addressing modes with the operations.

This decision depends on the range of addressing modes and the degree of independence between opcodes and modes. Some older computers have one to five operands with 10 addressing modes for each operand (see Figure 2.6 on page 108). For such a large number of combinations, typically a separate *address specifier* is needed for each operand: the address specifier tells what addressing mode is used to access the operand. At the other extreme are load-store computers with only one memory operand and only one or two addressing modes; obviously, in this case, the addressing mode can be encoded as part of the opcode.

When encoding the instructions, the number of registers and the number of addressing modes both have a significant impact on the size of instructions, as the the register field and addressing mode field may appear many times in a single instruction. In fact, for most instructions many more bits are consumed in encoding addressing modes and register fields than in specifying the opcode. The architect must balance several competing forces when encoding the instruction set:

1. The desire to have as many registers and addressing modes as possible.
2. The impact of the size of the register and addressing mode fields on the average instruction size and hence on the average program size.
3. A desire to have instructions encoded into lengths that will be easy to handle in a pipelined implementation. (The importance of having easily decoded instructions is discussed in Chapters 3 and 4.) As a minimum, the architect wants instructions to be in multiples of bytes, rather than an arbitrary bit length. Many desktop and server architects have chosen to use a fixed-length instruction to gain implementation benefits while sacrificing average code size.

Figure 2.23 shows three popular choices for encoding the instruction set. The first we call *variable*, since it allows virtually all addressing modes to be with all operations. This style is best when there are many addressing modes and operations. The second choice we call *fixed*, since it combines the operation and the addressing mode into the opcode. Often fixed encoding will have only a single size for all instructions; it works best when there are few addressing modes and operations. The trade-off between variable encoding and fixed encoding is size of programs versus ease of decoding in the processor. Variable tries to use as few bits as possible to represent the program, but individual instructions can vary widely in both size and the amount of work to be performed.

Let's look at an 80x86 instruction to see an example of the variable encoding:

```
add EAX,1000(EBX)
```

The name `add` means a 32-bit integer add instruction with two operands, and this opcode takes 1 byte. An 80x86 address specifier is 1 or 2 bytes, specifying the source/destination register (`EAX`) and the addressing mode (displacement in this case) and base register (`EBX`) for the second operand. This combination takes one byte to specify the operands. When in 32-bit mode (see Appendix C <80x86>), the size of the address field is either 1 byte or 4 bytes. Since 1000 is bigger than 2^8, the total length of the instruction is

$$1 + 1 + 4 = 6 \text{ bytes}$$

The length of 80x86 instructions varies between 1 and 17 bytes. 80x86 programs are generally smaller than the RISC architectures, which use fixed formats (Appendix B <RISC>)

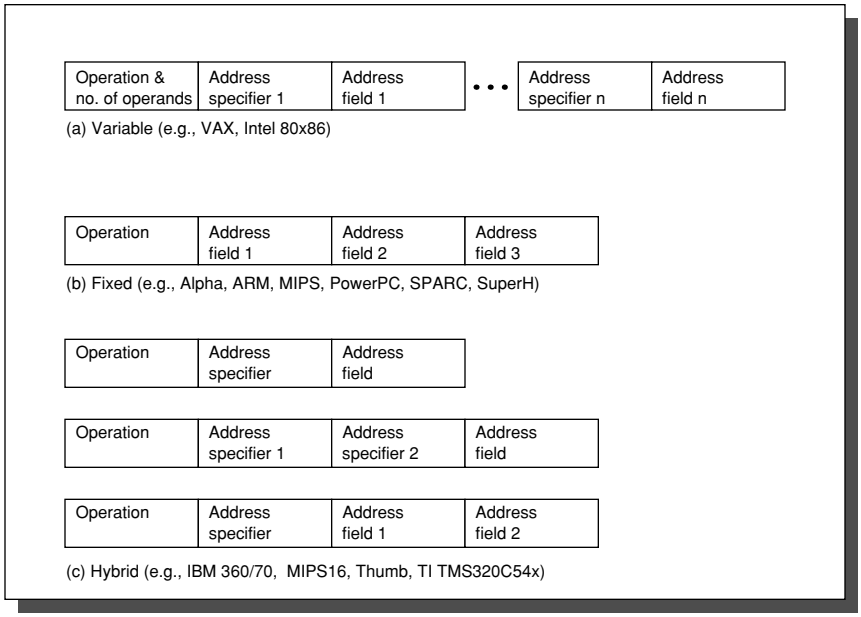


FIGURE 2.23 Three basic variations in instruction encoding: variable length, fixed length, and hybrid. The variable format can support any number of operands, with each address specifier determining the addressing mode and the length of the specifier for that operand. It generally enables the smallest code representation, since unused fields need not be included. The fixed format always has the same number of operands, with the addressing modes (if options exist) specified as part of the opcode (see also Figure C.3 on page C-4). It generally results in the largest code size. Although the fields tend not to vary in their location, they will be used for different purposes by different instructions. The hybrid approach has multiple formats specified by the opcode, adding one or two fields to specify the addressing mode and one or two fields to specify the operand address (see also Figure D.7 on page D-12).

Given these two poles of instruction set design of variable and fixed, the third alternative immediately springs to mind: Reduce the variability in size and work of the variable architecture but provide multiple instruction lengths to reduce code size. This *hybrid* approach is the third encoding alternative, and we'll see examples shortly.

Reduced Code Size in RISCs

As RISC computers started being used in embedded applications, the 32-bit fixed format became a liability since cost and hence smaller code are important. In response, several manufacturers offered a new hybrid version of their RISC instruction sets, with both 16-bit and 32-bit instructions. The narrow instructions

support fewer operations, smaller address and immediate fields, fewer registers, and two-address format rather than the classic three-address format of RISC computers. Appendix B <RISC> gives two examples, the ARM Thumb and MIPS MIPS16, which both claim a code size reduction of up to 40%.

In contrast to these instruction set extensions, IBM simply compresses its standard instruction set, and then adds hardware to decompress instructions as they are fetched from memory on an instruction cache miss. Thus, the instruction cache contains full 32-bit instructions, but compressed code is kept in main memory, ROMs, and the disk. The advantage of MIPS16 and Thumb is that instruction caches acts as it they are about 25% larger, while IBM's *CodePack* means that compilers need not be changed to handle different instruction sets and instruction decoding can remain simple.

CodePack starts with run-length encoding compression on any PowerPC program, and then loads the resulting compression tables in a 2KB table on chip. Hence, every program has its own unique encoding. To handle branches, which are no longer to an aligned word boundary, the PowerPC creates a hash-table in memory that maps between compressed and uncompressed addresses. Like a TLB (Chapter 5), it caches the most recently used address maps to reduce the number of memory accesses. IBM claims an overall performance cost of 10%, resulting in a code size reduction of 35% to 40%.

Hitachi simply invented a RISC instruction set with a fixed,16-bit format, called SuperH, for embedded applications (see Appendix B <RISC>). It has 16 rather than 32 registers to make it fit the narrower format and fewer instructions, but otherwise looks like a classic RISC architecture.

Summary: Encoding the Instruction Set

Decisions made in the components of instruction set design discussed in prior sections determine whether the architect has the choice between variable and fixed instruction encodings. Given the choice, the architect more interested in code size than performance will pick variable encoding, and the one more interested in performance than code size will pick fixed encoding. The appendices give 11 examples of the results of architect's choices. In Chapters 3 and 4, the impact of variability on performance of the processor will be discussed further.

We have almost finished laying the groundwork for the MIPS instruction set architecture that will be introduced in section 2.12. Before we do that, however, it will be helpful to take a brief look at compiler technology and its effect on program properties.

2.11 Crosscutting Issues: The Role of Compilers

Today almost all programming is done in high-level languages for desktop and server applications. This development means that since most instructions execut-

ed are the output of a compiler, an instruction set architecture is essentially a compiler target. In earlier times for these applications, and currently for DSPs, architectural decisions were often made to ease assembly language programming or for a specific kernel. Because the compiler will be significantly affect the performance of a computer, understanding compiler technology today is critical to designing and efficiently implementing an instruction set.

Once it was popular to try to isolate the compiler technology and its effect on hardware performance from the architecture and its performance, just as it was popular to try to separate architecture from its implementation. This separation is essentially impossible with today's desktop compilers and computers. Architectural choices affect the quality of the code that can be generated for a computer and the complexity of building a good compiler for it, for better or for worse. For example, section 2.14 shows the substantial performance impact on a DSP of compiling vs. hand optimizing the code.

In this section, we discuss the critical goals in the instruction set primarily from the compiler viewpoint. It starts with a review of the anatomy of current compilers. Next we discuss how compiler technology affects the decisions of the architect, and how the architect can make it hard or easy for the compiler to produce good code. We conclude with a review of compilers and multimedia operations, which unfortunately is a bad example of cooperation between compiler writers and architects.

The Structure of Recent Compilers

To begin, let's look at what optimizing compilers are like today. Figure 2.24 shows the structure of recent compilers

A compiler writer's first goal is correctness—all valid programs must be compiled correctly. The second goal is usually speed of the compiled code. Typically, a whole set of other goals follows these two, including fast compilation, debugging support, and interoperability among languages. Normally, the passes in the compiler transform higher-level, more abstract representations into progressively lower-level representations. Eventually it reaches the instruction set. This structure helps manage the complexity of the transformations and makes writing a bug-free compiler easier.

The complexity of writing a correct compiler is a major limitation on the amount of optimization that can be done. Although the multiple-pass structure helps reduce compiler complexity, it also means that the compiler must order and perform some transformations before others. In the diagram of the optimizing compiler in Figure 2.24, we can see that certain high-level optimizations are performed long before it is known what the resulting code will look like. Once such a transformation is made, the compiler can't afford to go back and revisit all steps, possibly undoing transformations. Such iteration would be prohibitive, both in compilation time and in complexity. Thus, compilers make assumptions about the ability of later steps to deal with certain problems. For example, com-

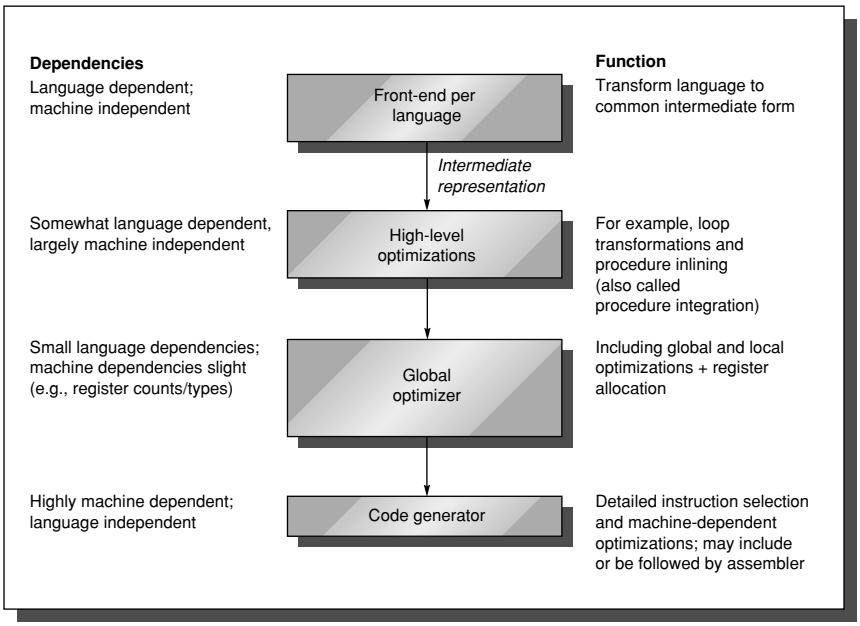


FIGURE 2.24 Compilers typically consist of two to four passes, with more highly optimizing compilers having more passes. This structure maximizes the probability that a program compiled at various levels of optimization will produce the same output when given the same input. The optimizing passes are designed to be optional and may be skipped when faster compilation is the goal and lower quality code is acceptable. A *pass* is simply one phase in which the compiler reads and transforms the entire program. (The term *phase* is often used interchangeably with *pass*.) Because the optimizing passes are separated, multiple languages can use the same optimizing and code-generation passes. Only a new front end is required for a new language.

pilers usually have to choose which procedure calls to expand in-line before they know the exact size of the procedure being called. Compiler writers call this problem the *phase-ordering problem*.

How does this ordering of transformations interact with the instruction set architecture? A good example occurs with the optimization called *global common subexpression elimination*. This optimization finds two instances of an expression that compute the same value and saves the value of the first computation in a temporary. It then uses the temporary value, eliminating the second computation of the common expression.

For this optimization to be significant, the temporary must be allocated to a register. Otherwise, the cost of storing the temporary in memory and later reloading it may negate the savings gained by not recomputing the expression. There are, in fact, cases where this optimization actually slows down code when the temporary is not register allocated. Phase ordering complicates this problem, be-

cause register allocation is typically done near the end of the global optimization pass, just before code generation. Thus, an optimizer that performs this optimization *must* assume that the register allocator will allocate the temporary to a register.

Optimizations performed by modern compilers can be classified by the style of the transformation, as follows:

1. *High-level optimizations* are often done on the source with output fed to later optimization passes.
2. *Local optimizations* optimize code only within a straight-line code fragment (called a *basic block* by compiler people).
3. *Global optimizations* extend the local optimizations across branches and introduce a set of transformations aimed at optimizing loops.
4. *Register allocation*.
5. *processor-dependent optimizations* attempt to take advantage of specific architectural knowledge.

Register Allocation

Because of the central role that register allocation plays, both in speeding up the code and in making other optimizations useful, it is one of the most important—if not the most important—optimizations. Register allocation algorithms today are based on a technique called *graph coloring*. The basic idea behind graph coloring is to construct a graph representing the possible candidates for allocation to a register and then to use the graph to allocate registers. Roughly speaking, the problem is how to use a limited set of colors so that no two adjacent nodes in a dependency graph have the same color. The emphasis in the approach is to achieve 100% register allocation of active variables. The problem of coloring a graph in general can take exponential time as a function of the size of the graph (NP-complete). There are heuristic algorithms, however, that work well in practice yielding close allocations that run in near linear time.

Graph coloring works best when there are at least 16 (and preferably more) general-purpose registers available for global allocation for integer variables and additional registers for floating point. Unfortunately, graph coloring does not work very well when the number of registers is small because the heuristic algorithms for coloring the graph are likely to fail.

Impact of Optimizations on Performance

It is sometimes difficult to separate some of the simpler optimizations—local and processor-dependent optimizations—from transformations done in the code generator. Examples of typical optimizations are given in Figure 2.25. The last column of Figure 2.25 indicates the frequency with which the listed optimizing transforms were applied to the source program.

Optimization name	Explanation	Percentage of the total number of optimizing transforms
High-level	**At or near the source level; processor-independent**	
Procedure integration	Replace procedure call by procedure body	N.M.
Local	**Within straight-line code**	
Common subexpression elimination	Replace two instances of the same computation by single copy	18%
Constant propagation	Replace all instances of a variable that is assigned a constant with the constant	22%
Stack height reduction	Rearrange expression tree to minimize resources needed for expression evaluation	N.M.
Global	**Across a branch**	
Global common subexpression elimination	Same as local, but this version crosses branches	13%
Copy propagation	Replace all instances of a variable A that has been assigned X (i.e., $A = X$) with X	11%
Code motion	Remove code from a loop that computes same value each iteration of the loop	16%
Induction variable elimination	Simplify/eliminate array-addressing calculations within loops	2%
Processor-dependent	**Depends on processor knowledge**	
Strength reduction	Many examples, such as replace multiply by a constant with adds and shifts	N.M.
Pipeline scheduling	Reorder instructions to improve pipeline performance	N.M.
Branch offset optimization	Choose the shortest branch displacement that reaches target	N.M.

FIGURE 2.25 Major types of optimizations and examples in each class. These data tell us about the relative frequency of occurrence of various optimizations. The third column lists the static frequency with which some of the common optimizations are applied in a set of 12 small FORTRAN and Pascal programs. There are nine local and global optimizations done by the compiler included in the measurement. Six of these optimizations are covered in the figure, and the remaining three account for 18% of the total static occurrences. The abbreviation *N.M.* means that the number of occurrences of that optimization was not measured. Processor-dependent optimizations are usually done in a code generator, and none of those was measured in this experiment. The percentage is the portion of the static optimizations that are of the specified type. Data from Chow [1983] (collected using the Stanford UCODE compiler).

Figure 2.26 shows the effect of various optimizations on instructions executed for two programs. In this case, optimized programs executed roughly 25% to 90% fewer instructions than unoptimized programs. The figure illustrates the importance of looking at optimized code before suggesting new instruction set features, for a compiler might completely remove the instructions the architect was trying to improve.

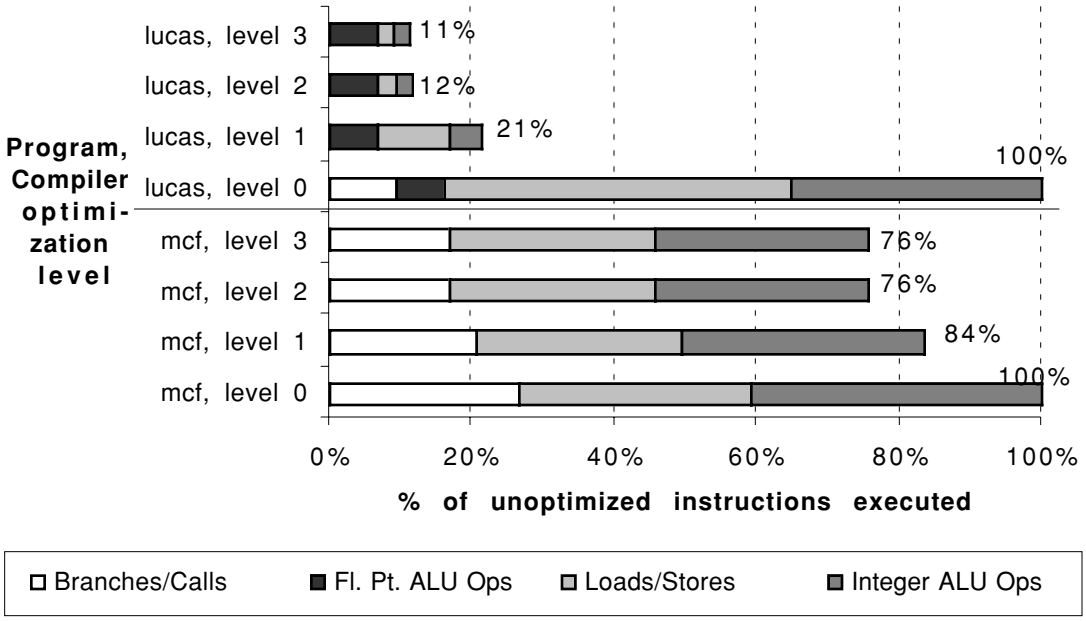


FIGURE 2.26 Change in instruction count for the programs `lucas` and `mcf` from the SPEC2000 as compiler optimization levels vary. Level 0 is the same as unoptimized code. Level 1 includes local optimizations, code scheduling, and local register allocation. Level 2 includes global optimizations, loop transformations (*software pipelining*), and global register allocation. Level 3 adds procedure integration. These experiments were performed on the Alpha compilers.

The Impact of Compiler Technology on the Architect's Decisions

The interaction of compilers and high-level languages significantly affects how programs use an instruction set architecture. There are two important questions: How are variables allocated and addressed? How many registers are needed to allocate variables appropriately? To address these questions, we must look at the three separate areas in which current high-level languages allocate their data:

- The *stack* is used to allocate local variables. The stack is grown and shrunk on procedure call or return, respectively. Objects on the stack are addressed relative to the stack pointer and are primarily scalars (single variables) rather than arrays. The stack is used for activation records, *not* as a stack for evaluating expressions. Hence, values are almost never pushed or popped on the stack.
- The *global data area* is used to allocate statically declared objects, such as global variables and constants. A large percentage of these objects are arrays or other aggregate data structures.

- The *heap* is used to allocate dynamic objects that do not adhere to a stack discipline. Objects in the heap are accessed with pointers and are typically not scalars.

Register allocation is much more effective for stack-allocated objects than for global variables, and register allocation is essentially impossible for heap-allocated objects because they are accessed with pointers. Global variables and some stack variables are impossible to allocate because they are *aliased*, which means that there are multiple ways to refer to the address of a variable, making it illegal to put it into a register. (Most heap variables are effectively aliased for today's compiler technology.)

For example, consider the following code sequence, where `&` returns the address of a variable and `*` dereferences a pointer:

```
p = &a          -- gets address of a in p
a = ...         -- assigns to a directly
*p = ...        -- uses p to assign to a
...a...         -- accesses a
```

The variable `a` could not be register allocated across the assignment to `*p` without generating incorrect code. Aliasing causes a substantial problem because it is often difficult or impossible to decide what objects a pointer may refer to. A compiler must be conservative; some compilers will not allocate *any* local variables of a procedure in a register when there is a pointer that may refer to *one* of the local variables.

How the Architect Can Help the Compiler Writer

Today, the complexity of a compiler does not come from translating simple statements like A = B + C. Most programs are *locally simple,* and simple translations work fine. Rather, complexity arises because programs are large and globally complex in their interactions, and because the structure of compilers means decisions are made one step at a time about which code sequence is best.

Compiler writers often are working under their own corollary of a basic principle in architecture: *Make the frequent cases fast and the rare case correct*. That is, if we know which cases are frequent and which are rare, and if generating code for both is straightforward, then the quality of the code for the rare case may not be very important—but it must be correct!

Some instruction set properties help the compiler writer. These properties should not be thought of as hard and fast rules, but rather as guidelines that will make it easier to write a compiler that will generate efficient and correct code.

1. *Regularity*—Whenever it makes sense, the three primary components of an instruction set—the operations, the data types, and the addressing modes—should be *orthogonal*. Two aspects of an architecture are said to be orthogonal if they are independent. For example, the operations and addressing modes are orthogonal if for every operation to which one addressing mode can be applied, all addressing modes are applicable. This regularity helps simplify code generation and is particularly important when the decision about what code to generate is split into two passes in the compiler. A good counterexample of this property is restricting what registers can be used for a certain class of instructions. Compilers for special-purpose register architectures typically get stuck in this dilemma. This restriction can result in the compiler finding itself with lots of available registers, but none of the right kind!

2. *Provide primitives, not solutions*—Special features that "match" a language construct or a kernel function are often unusable. Attempts to support high-level languages may work only with one language, or do more or less than is required for a correct and efficient implementation of the language. An example of how such attempts have failed is given in section 2.14.

3. *Simplify trade-offs among alternatives*—One of the toughest jobs a compiler writer has is figuring out what instruction sequence will be best for every segment of code that arises. In earlier days, instruction counts or total code size might have been good metrics, but—as we saw in the last chapter—this is no longer true. With caches and pipelining, the trade-offs have become very complex. Anything the designer can do to help the compiler writer understand the costs of alternative code sequences would help improve the code. One of the most difficult instances of complex trade-offs occurs in a register-memory architecture in deciding how many times a variable should be referenced before it is cheaper to load it into a register. This threshold is hard to compute and, in fact, may vary among models of the same architecture.

4. *Provide instructions that bind the quantities known at compile time as constants*—A compiler writer hates the thought of the processor interpreting at runtime a value that was known at compile time. Good counterexamples of this principle include instructions that interpret values that were fixed at compile time. For instance, the VAX procedure call instruction (`calls`) dynamically interprets a mask saying what registers to save on a call, but the mask is fixed at compile time (see section 2.14).

Compiler Support (or lack thereof) for Multimedia Instructions

Alas, the designers of the SIMD instructions that operate on several narrow data times in a single clock cycle consciously ignored the prior subsection. These instructions tend to be solutions, not primitives, they are short of registers, and the data types do not match existing programming languages. Architects hoped to find an inexpensive solution that would help some users, but in reality, only a few low-level graphics library routines use them.

The SIMD instructions are really an abbreviated version of an elegant architecture style that has its own compiler technology. As explained in Appendix F, *vector architectures* operate on vectors of data. Invented originally for scientific codes, multimedia kernels are often vectorizable as well. Hence, we can think of Intel's MMX or PowerPC's AltiVec as simply short vector computers: MMX with vectors of eight 8-bit elements, four 16-bit elements, or two 32-bit elements, and AltiVec with vectors twice that length. They are implemented as simply adjacent, narrow elements in wide registers

These abbreviated architectures build the vector register size into the architecture: the sum of the sizes of the elements is limited to 64 bits for MMX and 128 bits for AltiVec. When Intel decided to expand to 128 bit vectors, it added a whole new set of instructions, called SSE.

The missing elegance from these architectures involves the specification of the vector length and the memory addressing modes. By making the vector width variable, these vectors seemlessly switch between different data widths simply by increasing the number of elements per vector. For example, vectors could have, say, 32 64-bit elements, 64 32-bit elements, 128 16-bit elements, and 256 8-bit elements. Another advantage is that the number of elements per vector register can vary between generations while remaining binary compatible. One generation might have 32 64-bit elements per vector register, and the next have 64 64-bit elements. (The number of elements per register is located in a status register.) The number of elements executed per clock cycle is also implementation dependent, and all run the same binary code. Thus, one generation might operate 64-bits per clock cycle, and another at 256-bits per clock cycle.

A major advantage of vector computers is hiding latency of memory access by loading many elements at once and then overlapping execution with data transfer. The goal of vector addressing modes is to collect data scattered about memory, place them in a compact form so that they can be operated on efficiently, and then place the results back where they belong.

Over the years traditional vector computers added *strided addressing* and *gather/scatter addressing* to increase the number of programs that can be vectorized. Strided addressing skips a fixed number of words between each access, so sequential addressing is often called *unit stride addressing*. Gather and scatter find their addresses in another vector register: think of it as register indirect addressing for vector computers. From a vector perspective, in contrast these short-vector SIMD computers support only unit strided accesses: memory accesses load or store all elements at once from a single wide memory location. Since the data for multimedia applications are often streams that start and end in memory, strided and gather/scatter addressing modes such are essential to successful vectoization.

EXAMPLE As an example, compare a vector computer to MMX for color representation conversion of pixels from RBG (red blue green) to YUV (luminosity chrominance), with each pixel represented by three bytes. The conversion is just 3 lines of C code placed in a loop:

```
Y = ( 9798*R + 19235*G + 3736*B)/ 32768;
U = (-4784*R -  9437*G + 4221*B)/ 32768 + 128;
V = (20218*R - 16941*G - 3277*B) / 32768 + 128;
```

A 64-bit wide vector computer can calculate eight pixels simultaneously. One vector computer for media with strided addresses takes:

- 3 vector loads (to get RGB),
- 3 vector multiplies (to convert R),
- 6 vector multiply adds (to convert G and B),
- 3 vector shifts (to divide by 32768),
- 2 vector adds (to add 128), and
- 3 vector stores (to store YUV).

The total is 20 instructions to perform the 20 operations in the C code above to convert 8 pixels [Kozyrakis 2000]. (Since a vector might have 32 64-bit elements, this code actually converts up to 32 x 8 or 256 pixels.)

In contrast, Intel's web site shows a library routine to perform the same calculation on eight pixels takes 116 MMX instructions plus 6 80x86 instructions [Intel 2001]. This sixfold increase in instructions is due to the large number of instructions to load and unpack RBG pixels and to pack and store YUV pixels, since there are no strided memory accesses.

Having short, architecture limited vectors with few registers and simple memory addressing modes makes it more difficult to use vectorizing compiler technology. Another challenge is that no programming language (yet) has support for operations on these narrow data. Hence, these SIMD instructions are commonly found only in hand coded libraries.

Summary: The Role of Compilers

This section leads to several recommendations. First, we expect a new instruction set architecture to have at least 16 general-purpose registers—not counting separate registers for floating-point numbers—to simplify allocation of registers using graph coloring. The advice on orthogonality suggests that all supported addressing modes apply to all instructions that transfer data. Finally, the last three pieces

of advice—provide primitives instead of solutions, simplify trade-offs between alternatives, don't bind constants at runtime—all suggest that it is better to err on the side of simplicity. In other words, understand that less is more in the design of an instruction set. Alas, SIMD extensions are more an example of good marketing than outstanding achievement of hardware/software co-design.

2.12 Putting It All Together: The MIPS Architecture

In this section we describe a simple 64-bit load-store architecture called MIPS. The instruction set architecture of MIPS and RISC relatives was based on observations similar to those covered in the last sections. (In section 2.16 we discuss how and why these architectures became popular.) Reviewing our expectations from each section: for desktop applications:

- *Section 2.2*—Use general-purpose registers with a load-store architecture.
- *Section 2.3*—Support these addressing modes: displacement (with an address offset size of 12 to 16 bits), immediate (size 8 to 16 bits), and register indirect.
- *Section 2.5*—Support these data sizes and types: 8-, 16-, 32-bit, and 64-bit integers and 64-bit IEEE 754 floating-point numbers.
- *Section 2.7*—Support these simple instructions, since they will dominate the number of instructions executed: load, store, add, subtract, move register-register, and, shift.
- *Section 2.9*—Compare equal, compare not equal, compare less, branch (with a PC-relative address at least 8 bits long), jump, call, and return.
- *Section 2.10*—Use fixed instruction encoding if interested in performance and use variable instruction encoding if interested in code size.
- *Section 2.11*—Provide at least 16 general-purpose registers, and be sure all addressing modes apply to all data transfer instructions, and aim for a minimalist instruction set. This section didn't cover floating-point programs, but they often use separate floating-point registers. The justification is to increase the total number of registers without raising problems in the instruction format or in the speed of the general-purpose register file. This compromise, however, is not orthogonal.

We introduce MIPS by showing how it follows these recommendations. Like most recent computers, MIPS emphasizes

- A simple load-store instruction set
- Design for pipelining efficiency (discussed in Appendix A), including a fixed instruction set encoding
- Efficiency as a compiler target

MIPS provides a good architectural model for study, not only because of the popularity of this type of processor (see Chapter 1), but also because it is an easy architecture to understand. We will use this architecture again in Chapters 3 and 4, and it forms the basis for a number of exercises and programming projects.

In the 15 years since the first MIPS processor, there have been many versions of MIPS (see Appendix B <RISC>). We will use a subset of what is now called MIPS64, which will often abbreviate to just MIPS, but the full instruction set is found in Appendix B.

Registers for MIPS

MIPS64 has 32 64-bit general-purpose registers (GPRs), named R0, R1, …, R31. GPRs are also sometimes known as integer registers. Additionally, there is a set of 32 floating-point registers (FPRs), named F0, F1, ..., F31, which can hold 32 single-precision (32-bit) values or 32 double-precision (64-bit) values. (When holding one single-precision number, the other half of the FPR is unused.) Both single- and double-precision floating-point operations (32-bit and 64-bit) are provided. MIPS also includes instructions that operate on two single precision operands in a single 64-bit floating-point register.

The value of R0 is always 0. We shall see later how we can use this register to synthesize a variety of useful operations from a simple instruction set.

A few special registers can be transferred to and from the general-purpose registers. An example is the floating-point status register, used to hold information about the results of floating-point operations. There are also instructions for moving between a FPR and a GPR.

Data types for MIPS

The data types are 8-bit bytes, 16-bit half words, 32-bit words, and 64-bit double words for integer data and 32-bit single precision and 64-bit double precision for floating point. Half words were added because they are found in languages like C and popular in some programs, such as the operating systems, concerned about size of data structures. They will also become more popular if Unicode becomes widely used. Single-precision floating-point operands were added for similar reasons. (Remember the early warning that you should measure many more programs before designing an instruction set.)

The MIPS64 operations work on 64-bit integers and 32- or 64-bit floating point. Bytes, half words, and words are loaded into the general-purpose registers with either zeros or the sign bit replicated to fill the 32 bits of the GPRs. Once loaded, they are operated on with the 64-bit integer operations.

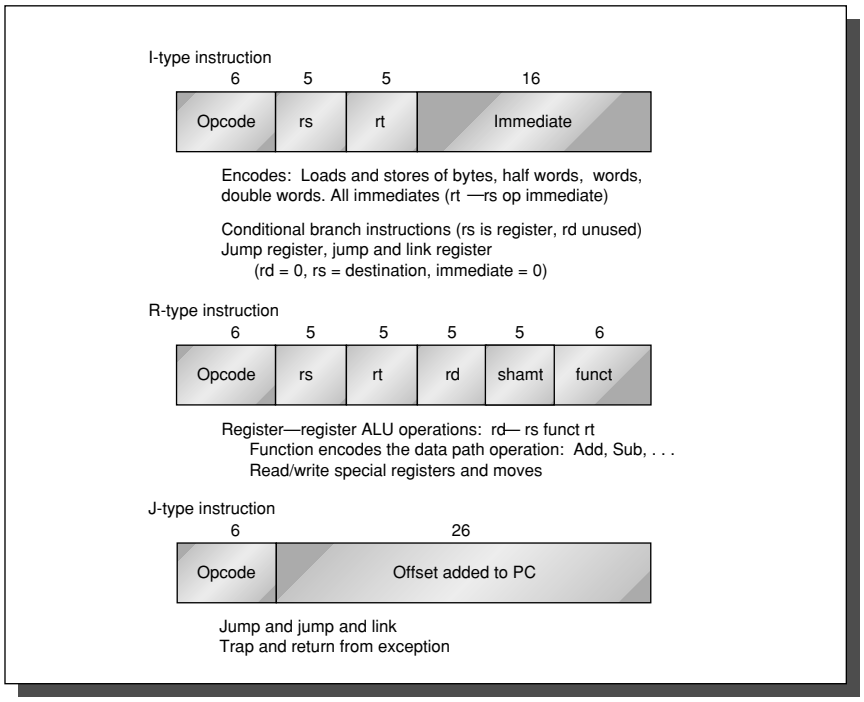


FIGURE 2.27 Instruction layout for MIPS. All instructions are encoded in one of three types, with common fields in the same location in each format.

Addressing modes for MIPS data transfers

The only data addressing modes are immediate and displacement, both with 16-bit fields. Register indirect is accomplished simply by placing 0 in the 16-bit displacement field, and absolute addressing with a 16-bit field is accomplished by using register 0 as the base register. Embracing zero gives us four effective modes, although only two are supported in the architecture.

MIPS memory is byte addressable in Big Endian mode with a 64-bit address. As it is a load-store architecture, all references between memory and either GPRs or FPRs are through loads or stores. Supporting the data types mentioned above, memory accesses involving GPRs can be to a byte, half word, word, or double word. The FPRs may be loaded and stored with single-precision or double-precision numbers. All memory accesses must be aligned.

MIPS Instruction Format

Since MIPS has just two addressing modes, these can be encoded into the opcode. Following the advice on making the processor easy to pipeline and decode,

all instructions are 32 bits with a 6-bit primary opcode. Figure 2.27 shows the instruction layout. These formats are simple while providing 16-bit fields for displacement addressing, immediate constants, or PC-relative branch addresses.

Appendix B shows a variant of MIPS—called MIPS16—which has 16-bit and 32-bit instructions to improve code density for embedded applications. We will stick to the traditional 32-bit format in this book.

MIPS Operations

MIPS supports the list of simple operations recommended above plus a few others. There are four broad classes of instructions: loads and stores, ALU operations, branches and jumps, and floating-point operations.

Example instruction	Instruction name	Meaning
LD R1,30(R2)	Load double word	Regs[R1]$\leftarrow_{64}$ Mem[30+Regs[R2]]
LD R1,1000(R0)	Load double word	Regs[R1]$\leftarrow_{64}$ Mem[1000+0]
LW R1,60(R2)	Load word	Regs[R1]$\leftarrow_{64}$ (Mem[60+Regs[R2]]$_0$)^{32} ## Mem[60+Regs[R2]]
LB R1,40(R3)	Load byte	Regs[R1]$\leftarrow_{64}$ (Mem[40+Regs[R3]] $_0$)^{56} ## Mem[40+Regs[R3]]
LBU R1,40(R3)	Load byte unsigned	Regs[R1]$\leftarrow_{64}$ 0^{56} ## Mem[40+Regs[R3]]
LH R1,40(R3)	Load half word	Regs[R1]$\leftarrow_{64}$ (Mem[40+Regs[R3]]$_0$)^{48} ## Mem[40+Regs[R3]]##Mem[41+Regs[R3]]
L.S F0,50(R3)	Load FP single	Regs[F0]$\leftarrow_{64}$ Mem[50+Regs[R3]] ## 0^{32}
L.D F0,50(R2)	Load FP double	Regs[F0]$\leftarrow_{64}$ Mem[50+Regs[R2]]
SD R3,500(R4)	Store double word	Mem[500+Regs[R4]]$\leftarrow_{64}$ Regs[R3]
SW R3,500(R4)	Store word	Mem[500+Regs[R4]]$\leftarrow_{32}$ Regs[R3]
S.S F0,40(R3)	Store FP single	Mem[40+Regs[R3]]$\leftarrow_{32}$ Regs[F0]$_{0..31}$
S.D F0,40(R3)	Store FP double	Mem[40+Regs[R3]]$\leftarrow_{64}$ Regs[F0]
SH R3,502(R2)	Store half	Mem[502+Regs[R2]]$\leftarrow_{16}$ Regs[R3]$_{48..63}$
SB R2,41(R3)	Store byte	Mem[41+Regs[R3]]$\leftarrow_{8}$ Regs[R2]$_{56..63}$

FIGURE 2.28 The load and store instructions in MIPS. All use a single addressing mode and require that the memory value be aligned. Of course, both loads and stores are available for all the data types shown.

Any of the general-purpose or floating-point registers may be loaded or stored, except that loading R0 has no effect. Figure 2.28 gives examples of the load and store instructions. Single-precision floating-point numbers occupy half a floating-point register. Conversions between single and double precision must be done explicitly. The floating-point format is IEEE 754 (see Appendix G). A list of the all the MIPS instructions in our subset appears in Figure 2.31 (page 146).

To understand these figures we need to introduce a few additional extensions to our C description language presented initially on page 107:

- A subscript is appended to the symbol ← whenever the length of the datum being transferred might not be clear. Thus, $\leftarrow_n$ means transfer an n-bit quantity. We use $x, y \leftarrow z$ to indicate that z should be transferred to x and y.
- A subscript is used to indicate selection of a bit from a field. Bits are labeled from the most-significant bit starting at 0. The subscript may be a single digit (e.g., `Regs[R4]`$_0$ yields the sign bit of R4) or a subrange (e.g., `Regs[R3]`$_{56..63}$ yields the least-significant byte of R3).
- The variable `Mem`, used as an array that stands for main memory, is indexed by a byte address and may transfer any number of bytes.
- A superscript is used to replicate a field (e.g., 0^{48} yields a field of zeros of length 48 bits).
- The symbol ## is used to concatenate two fields and may appear on either side of a data transfer.

A summary of the entire description language appears on the back inside cover. As an example, assuming that R8 and R10 are 64-bit registers:

$$\texttt{Regs[R10]}_{32..63} \leftarrow_{32} (\texttt{Mem[Regs[R8]]}_0)^{24} \ \#\# \ \texttt{Mem[Regs[R8]]}$$

means that the byte at the memory location addressed by the contents of register R8 is sign-extended to form a 32-bit quantity that is stored into the lower half of register R10. (The upper half of R10 is unchanged.)

All ALU instructions are register-register instructions. Figure 2.29 gives some examples of the arithmetic/logical instructions. The operations include simple arithmetic and logical operations: add, subtract, AND, OR, XOR, and shifts. Immediate forms of all these instructions are provided using a 16-bit sign-extended immediate. The operation `LUI` (load upper immediate) loads bits 32 to 47 of a register, while setting the rest of the register to 0. `LUI` allows a 32-bit constant to be built in two instructions, or a data transfer using any constant 32-bit address in one extra instruction.

As mentioned above, R0 is used to synthesize popular operations. Loading a constant is simply an add immediate where one source operand is R0, and a register-register move is simply an add where one of the sources is R0. (We sometimes use the mnemonic `LI`, standing for load immediate, to represent the former and the mnemonic `MOV` for the latter.)

MIPS Control Flow Instructions

MIPS provides compare instructions, which compare two registers to see if the first is less than the second. If the condition is true, these instructions place a

Example instruction	Instruction name	Meaning
DADDU R1,R2,R3	Add unsigned	Regs[R1]←Regs[R2]+Regs[R3]
DADDIU R1,R2,#3	Add immediate unsigned	Regs[R1]←Regs[R2]+3
LUI R1,#42	Load upper immediate	Regs[R1]←0^{32}##42##0^{16}
SLL R1,R2,#5	Shift left logical	Regs[R1]←Regs[R2]<<5
SLT R1,R2,R3	Set less than	if (Regs[R2]<Regs[R3]) Regs[R1]←1 else Regs[R1]←0

FIGURE 2.29 Examples of arithmetic/logical instructions on MIPS, both with and without immediates.

1 in the destination register (to represent true); otherwise they place the value 0. Because these operations "set" a register, they are called set-equal, set-not-equal, set-less-than, and so on. There are also immediate forms of these compares.

Example instruction	Instruction name	Meaning
J name	Jump	$PC_{36..63}$←name
JAL name	Jump and link	Regs[R31]←PC+4; $PC_{36..63}$←name; ((PC+4)−2^{27}) ≤ name < ((PC+4)+2^{27})
JALR R2	Jump and link register	Regs[R31]←PC+4; PC←Regs[R2]
JR R3	Jump register	PC←Regs[R3]
BEQZ R4,name	Branch equal zero	if (Regs[R4]==0) PC←name; ((PC+4)−2^{17}) ≤ name < ((PC+4)+2^{17})
BNE R3,R4,name	Branch not equal zero	if (Regs[R3]!= Regs[R4]) PC←name; ((PC+4)−2^{17}) ≤ name < ((PC+4)+2^{17})
MOVZ R1,R2,R3	Conditional move if zero	if (Regs[R3]==0) Regs[R1]←Regs[R2]

FIGURE 2.30 Typical control-flow instructions in MIPS. All control instructions, except jumps to an address in a register, are PC-relative. Note that the branch distances are longer than the address field would suggestion; since MIPS instructions are all 32-bits long, the byte branch address is multiplied by 4 to get a longer distance.

Control is handled through a set of jumps and a set of branches. Figure 2.30 gives some typical branch and jump instructions. The four jump instructions are differentiated by the two ways to specify the destination address and by whether or not a link is made. Two jumps use a 26-bit offset shifted two bits and then replaces the lower 28 bits of the program counter (of the instruction sequentially following the jump) to determine the destination address. The other two jump instructions specify a register that contains the destination address. There are two flavors of jumps: plain jump, and jump and link (used for procedure calls). The latter places the return address—the address of the next sequential instruction—in R31.

Instruction type/opcode	Instruction meaning
Data transfers	**Move data between registers and memory, or between the integer and FP or special registers; only memory address mode is 16-bit displacement + contents of a GPR**
LB,LBU,SB	Load byte, load byte unsigned, store byte (to/from integer registers)
LH,LHU,SH	Load half word, load half word unsigned, store half word (to/from integer registers)
LW,LWU,SW	Load word, Load word unsigned, store word (to/from integer registers)
LD,SD	Load double word, store double word (to/from integer registers)
L.S,L.D,S.S,S.D	Load SP float, load DP float, store SP float, store DP float
MFC0,MTC0	Move from/to GPR to/from a special register
MOV.S,MOV.D	Copy one SP or DP FP register to another FP register
MFC1,MTC1	Move 32 bits from/to FP registers to/from integer registers
Arithmetic/logical	**Operations on integer or logical data in GPRs; signed arithmetic trap on overflow**
DADD,DADDI,DADDU, DADDIU	Add, add immediate (all immediates are 16 bits); signed and unsigned
DSUB,DSUBU	Subtract, subtract immediate; signed and unsigned
DMUL,DMULU,DDIV,DDIVU	Multiply and divide, signed and unsigned; all operations take and yield 64-bit values
AND,ANDI	And, and immediate
OR,ORI,XOR,XORI	Or, or immediate, exclusive or, exclusive or immediate
LUI	Load upper immediate—loads bits 32 to 47 of register with immediate; then sign extends
DSLL, SDRL, DSRA, DSLLV, DSRLV, DSRAV	Shifts: both immediate (DS__) and variable form (DS__V); shifts are shift left logical, right logical, right arithmetic
SLT,SLTI,SLTU,SLTIU	Set less than, set less than immediate; signed and unsigned
Control	**Conditional branches and jumps; PC-relative or through register**
BEQZ,BNEZ	Branch GPR equal/not equal to zero; 16-bit offset from PC+4
BC1T,BC1F	Test comparison bit in the FP status register and branch; 16-bit offset from PC+4
J, JR	Jumps: 26-bit offset from PC+4 (J) or target in register (JR)
JAL, JALR	Jump and link: save PC+4 in R31, target is PC-relative (JAL) or a register (JALR)
TRAP	Transfer to operating system at a vectored address
ERET	Return to user code from an exception; restore user mode
Floating point	**FP operations on DP and SP formats**
ADD.D,ADD.S,ADD.PS	Add DP, SP numbers, an d pairs of SP numbers
SUB.D,SUB.S,ADD.PS	Subtract DP, SP numbers, an d pairs of SP numbers
MUL.D,MUL.S,MUL.PS	Multiply DP, SP floating point, an d pairs of SP numbers
DIV.D,DIV.S,DIV.PS	Divide DP, SP floating point, an d pairs of SP numbers
CVT._._	Convert instructions: CVT.x.y converts from type x to type y, where x and y are L (64-bit integer), W (32-bit integer), D (DP), or S (SP). Both operands are FPRs.
C.__.D,C.__.S	DP and SP compares: "__" = LT,GT,LE,GE,EQ,NE; sets bit in FP status register

FIGURE 2.31 Subset of the instructions in MIPS64. Figure 2.27 lists the formats of these instructions. SP = single precision; DP = double precision. This list can also be found on the page preceding the back inside cover.

All branches are conditional. The branch condition is specified by the instruction, which may test the register source for zero or nonzero; the register may contain a data value or the result of a compare. There are also conditional branch instructions to test for whether a register is negative and for equality between two registers. The branch target address is specified with a 16-bit signed offset that is added to the program counter, which is pointing to the next sequential instruction. There is also a branch to test the floating-point status register for floating-point conditional branches, described below.

Chapters 3 and 4 show that conditional branches are a major challenge to pipelined execution; hence many architectures have added instructions to convert a simple branch into a condition arithmetic instruction. MIPS included conditional move on zero or not zero. The value of the destination register either is left unchanged or is replaced by a copy of one of the source registers depending on whether or not the value of the other source register is zero.

MIPS Floating-Point Operations

Floating-point instructions manipulate the floating-point registers and indicate whether the operation to be performed is single or double precision. The operations `MOV.S` and `MOV.D` copy a single-precision (`MOV.S`) or double-precision (`MOV.D`) floating-point register to another register of the same type. The operations `MFC1` and `MTC1` move data between a single floating-point register and an integer register; moving a double-precision value to two integer registers requires two instructions. Conversions from integer to floating point are also provided, and vice versa.

The floating-point operations are add, subtract, multiply, and divide; a suffix `D` is used for double precision and a suffix `S` is used for single precision (e.g., `ADD.D`, `ADD.S`, `SUB.D`, `SUB.S`, `MUL.D`, `MUL.S`, `DIV.D`, `DIV.S`). Floating-point compares set a bit in the special floating-point status register that can be tested with a pair of branches: `BC1T` and `BC1F`, branch floating-point true and branch floating-point false.

To get greater performance for graphics routines, MIPS64 has instructions that perform two 32-bit floating-point operations on each half of the 64-bit floating-point register. These *paired single* operations include `ADD.PS`, `SUB.PS`, `MUL.PS`, and `DIV.PS`. (They are loaded and store using double precision loads and stores.)

Giving a nod towards the importance of DSP applications, MIPS64 also includes both integer and floating-point multiply-add instructions: `MADD`, `MADD.S`, `MADD.D`, and `MADD.PS`. Unlike DSPs, the registers are all the same width in these combined operations.

Figure 2.31 on page 146 contains a list of a subset of MIPS64 operations and their meaning.

Instruction	gap	gcc	gzip	mcf	perl	Integer average
load	44.7%	35.5%	31.8%	33.2%	41.6%	37%
store	10.3%	13.2%	5.1%	4.3%	16.2%	10%
add	7.7%	11.2%	16.8%	7.2%	5.5%	10%
sub	1.7%	2.2%	5.1%	3.7%	2.5%	3%
mul	1.4%	0.1%				0%
compare	2.8%	6.1%	6.6%	6.3%	3.8%	5%
cond branch	9.3%	12.1%	11.0%	17.5%	10.9%	12%
cond move	0.4%	0.6%	1.1%	0.1%	1.9%	1%
jump	0.8%	0.7%	0.8%	0.7%	1.7%	1%
call	1.6%	0.6%	0.4%	3.2%	1.1%	1%
return	1.6%	0.6%	0.4%	3.2%	1.1%	1%
shift	3.8%	1.1%	2.1%	1.1%	0.5%	2%
and	4.3%	4.6%	9.4%	0.2%	1.2%	4%
or	7.9%	8.5%	4.8%	17.6%	8.7%	9%
xor	1.8%	2.1%	4.4%	1.5%	2.8%	3%
other logical	0.1%	0.4%	0.1%	0.1%	0.3%	0%
load FP						0%
store FP						0%
add FP						0%
sub FP						0%
mul FP						0%
div FP						0%
mov reg-reg FP						0%
compare FP						0%
cond mov FP						0%
other FP						0%

FIGURE 2.32 MIPS dynamic instruction mix for five SPECint2000 programs. Note that integer register-register move instructions are included in the or instruction. Blank entries have the value 0.0%.

MIPS Instruction Set Usage

To give an idea which instructions are popular, Figure 2.32 shows the frequency of instructions and instruction classes for five SPECint92 programs and Figure 2.33 shows the same data for five SPECfp92 programs. To give a more intuitive

Instruction	applu	art	equake	lucas	swim	FP average
load	32.2%	28.0%	29.0%	15.4%	27.5%	26%
store	2.9%		0.8%	3.4%	1.3%	2%
add	25.7%	20.2%	11.7%	8.2%	15.3%	16%
sub	2.5%		0.1%	2.1%	3.8%	2%
mul	2.3%			1.2%		1%
compare		7.4%	2.1%			2%
cond branch	2.5%	11.5%	2.9%	0.6%	1.3%	4%
cond mov		0.3%	0.1%			0%
jump			0.1%			0%
call			0.7%			0%
return			0.7%			0%
shift	0.7%		0.2%	1.9%		1%
and			0.2%	1.8%		0%
or	0.8%	1.1%	2.3%	1.0%	7.2%	2%
xor		3.2%	0.1%			1%
other logical			0.1%			0%
load FP	11.4%	12.0%	19.7%	16.2%	16.8%	15%
store FP	4.2%	4.5%	2.7%	18.2%	5.0%	7%
add FP	2.3%	4.5%	9.8%	8.2%	9.0%	7%
sub FP	2.9%		1.3%	7.6%	4.7%	3%
mul FP	8.6%	4.1%	12.9%	9.4%	6.9%	8%
div FP	0.3%	0.6%	0.5%		0.3%	0%
mov reg-reg FP	0.7%	0.9%	1.2%	1.8%	0.9%	1%
compare FP		0.9%	0.6%	0.8%		0%
cond mov FP		0.6%		0.8%		0%
other FP				1.6%		0%

FIGURE 2.33 MIPS dynamic instruction mix for five programs from SPECfp2000. Note that integer register-register move instructions are included in the or instruction. Blank entries have the value 0.0%.

feeling, Figure 2.34 shows the data graphically for all instructions that are responsible on average for more than 1% of the instructions executed.

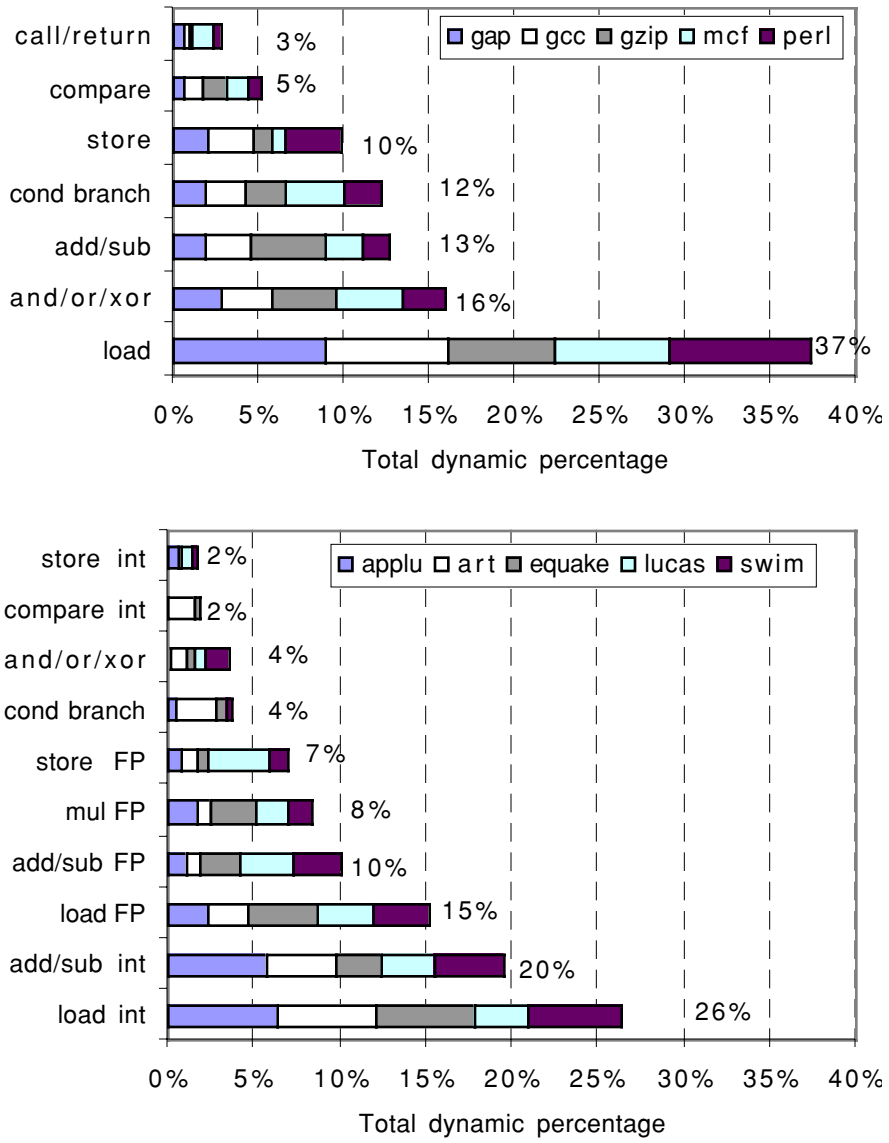


FIGURE 2.34 Graphical display of instructions executed of the five programs from SPECint2000 in Figure 2.32 (top) and the five programs from SPECfp2000 in Figure 2.33 (bottom). Just as in Figures 2.16 and 2.18, the most popular instructions are simple. These instruction classes collectively are responsible on average for 96% of instructions executed for SPECint2000 and 97% of instructions executed for SPECfp2000.

2.13 Another View: The Trimedia TM32 CPU

Media processor is a name given to a class of embedded processors that are dedicated to multimedia processing, typically being cost sensitive like embedded processors but following the compiler orientation from desktop and server computing. Like DSPs, they operate on narrower data types than the desktop, and must often deal with infinite, continuous streams of data. Figure 2.35 gives a list of media application areas and benchmark algorithms for media processors.

Application area	Benchmarks
Data Communication	Verterbi decoding
Audio coding	AC3 Decode
Video coding	MPEG2 encode, DVD decode
Video processing	Layered natural motion, Dynamic noise, Reduction, Peaking
Graphics	3D renderer library

FIGURE 2.35 Media processor application areas and example benchmarks. From Riemens [1999]. This lists shares only Viterbi decoding with the EEMBC benchmarks (see Figure 1.12 in Chapter 1), with the rest being generally larger programs than EEMBC.

The Trimedia TM32 CPU is a representative of this class. As multimedia applications have considerable parallelism in the processing of these data streams, the instruction set architectures often look different from the desktop. Its is intended for products like set top boxes and advanced televisions.

First, there are many more registers: 128 32-bit registers, which contain either integer or floating point data. Second, and not surprisingly, it offers the partitioned ALU or SIMD instructions to allow computations on multiple instances of narrower data, as described in Figure 2.17 on page 120. Third, showing its heritage, for integers it offers both two's complement arithmetic favored by desktop processors and saturating arithmetic favored by DSPs. Figure 2.36 lists the operations found in the Trimedia TM32 CPU.

However, the most unusual feature from the perspective of the desktop is that the architecture allows the programmer to specify five independent operations to be issued at the same time. If there are not five independent instructions available for the compiler to schedule together–that is, the rest are dependent–then NOPs are placed in the leftover slots. This instruction coding technique is called, naturally enough, *Very Long Instruction Word* (*VLIW*), and it predates the Trimedia processors. VLIW is the subject of Chapter 4, so just give a preview of VLIW here. An example helps explain how the Trimedia TM32 CPU works, and one can be found in Chapter 4 on page 279 <<Xref to example in section 4.8>>. This section also compares the performance of the Trimedia TM32 CPU using the EEMBC benchmarks.

Operation Category	Examples	Number of Operations	Comment
Load/store ops	ld8, ld16, ld32,limm. st8, st16, st32	33	signed, unsigned, register indirect, indexed, scaled addressing
Byte shuffles	shift right 1-, 2-, 3-bytes, select byte, merge, pack	11	SIMD type convert
Bit shifts	asl, asr, lsl, lsr, rol,	10	shifts, SIMD
Multiplies and multimedia	mul, sum of products, sum-of-SIMD-elements, multimedia, e.g. sum of products (FIR)	23	round, saturate, 2's comp, SIMD
Integer arithmetic	add, sub, min, max, abs, average, bitand, bitor, bitxor, bitinv, bitandinv eql, neq, gtr, geq, les, leq, sign extend, zero extend, sum of absolute differences	62	saturate, 2's comp, unsigned, immediate, SIMD
Floating point	add, sub, neg, mul, div, sqrt eql, neq, gtr, geq, les, leq, IEEE flags	42	scalar
Special ops	alloc, prefetch, copy back, read tag read, cache status, read counter	20	cache, special regs
Branch	jmpt, jmpf	6	(un)interruptible
Total		207	

FIGURE 2.36 List of operations and number of variations in Trimedia TM32 CPU. The data transfer opcodes include addressing modes in the count of operations, so the number is high compared to other architectures. SIMD means partitioned ALU operations of multiple narrow data items being operated on simultaneously in a 32-bit ALU, these include special operations for multimedia. The branches are delayed 3 slots.

Given the Trimedia TM32 CPU has longer instruction words and they often contain NOPs, Trimedia compacts its instructions in memory, decoding them to the full size when loaded into the cache.

Figure 2.37 shows the TM32 CPU instruction mix for the EEMBC benchmarks. Using the unmodified source code, the instruction mix is similar to others, although there are more byte data transfers. If the C code is hand-tuned, it can extensively use SIMD instructions. Note the large number of pack and merge instructions to align the data for the SIMD instructions. The cost in code size of these VLIW instructions is still a factor of two to three larger than MIPS *after* compaction.

2.14 Fallacies and Pitfalls

Architects have repeatedly tripped on common, but erroneous, beliefs. In this section we look at a few of them.

Operation	Out-of-the-box	Modified C Source Code
add word	26.5%	20.5%
load byte	10.4%	1.0%
subtract word	10.1%	1.1%
shift left arithmetic	7.8%	0.2%
store byte	7.4%	1.5%
multiply word	5.5%	0.4%
shift right arithmetic	3.6%	0.7%
and word	3.6%	6.8%
load word	3.5%	7.2%
load immediate	3.1%	1.6%
set greater than, equal	2.9%	1.3%
store word	2.0%	5.3%
jump	1.8%	0.8%
conditional branch	1.3%	1.0%
pack/merge bytes	2.6%	16.8%
SIMD sum of half word products	0.0%	10.1%
SIMD sum of byte products	0.0%	7.7%
pack/merge half words	0.0%	6.5%
SIMD subtract half word	0.0%	2.9%
SIMD maximum byte	0.0%	1.9%
Total	92.2%	95.5%
TM32 CPU Code Size (bytes)	243,968	387,328
MIPS Code Size (bytes)	120,729	

FIGURE 2.37 TM32 CPU instruction mix running EEMBC consumer benchmark. The instruction mix for "out-of-the-box" C code is similar to general-purpose computers, with a higher emphasis of byte data transfers. The hand-optimized C code uses the SIMD instructions and the pack and merge instructions to align the data. The middle column shows the relative instruction mix for unmodified kernels, while the right column allows modification at the C level. These columns list of all operation that were responsible for at least 1% of the total in one of the mixes. MIPS code size is for the Apogee compiler for the NECVR5432.

Pitfall: Designing a "high-level" instruction set feature specifically oriented to supporting a high-level language structure.

Attempts to incorporate high-level language features in the instruction set have led architects to provide powerful instructions with a wide range of flexibility. However, often these instructions do more work than is required in the frequent case, or they don't exactly match the requirements of some languages. Many such efforts have been aimed at eliminating what in the 1970s was called the *semantic gap*. Although the idea is to supplement the instruction set with additions that bring the hardware up to the level of the language, the additions can generate what Wulf [1981] has called a *semantic clash*:

... by giving too much semantic content to the instruction, the computer designer made it possible to use the instruction only in limited contexts. [p. 43]

More often the instructions are simply overkill—they are too general for the most frequent case, resulting in unneeded work and a slower instruction. Again, the VAX `CALLS` is a good example. `CALLS` uses a callee-save strategy (the registers to be saved are specified by the callee) *but* the saving is done by the call instruction in the caller. The `CALLS` instruction begins with the arguments pushed on the stack, and then takes the following steps:

1. Align the stack if needed.
2. Push the argument count on the stack.
3. Save the registers indicated by the procedure call mask on the stack (as mentioned in section 2.11). The mask is kept in the called procedure's code—this permits callee to specify the registers to be saved by the caller even with separate compilation.
4. Push the return address on the stack, and then push the top and base of stack pointers (for the activation record).
5. Clear the condition codes, which sets the trap enables to a known state.
6. Push a word for status information and a zero word on the stack.
7. Update the two stack pointers.
8. Branch to the first instruction of the procedure.

The vast majority of calls in real programs do not require this amount of overhead. Most procedures know their argument counts, and a much faster linkage convention can be established using registers to pass arguments rather than the stack in memory. Furthermore, the `CALLS` instruction forces two registers to be used for linkage, while many languages require only one linkage register. Many attempts to support procedure call and activation stack management have failed to be useful, either because they do not match the language needs or because they are too general and hence too expensive to use.

The VAX designers provided a simpler instruction, JSB, that is much faster since it only pushes the return PC on the stack and jumps to the procedure. However, most VAX compilers use the more costly CALLS instructions. The call instructions were included in the architecture to standardize the procedure linkage convention. Other computers have standardized their calling convention by agreement among compiler writers and without requiring the overhead of a complex, very general-procedure call instruction.

Fallacy: There is such a thing as a typical program.

Many people would like to believe that there is a single "typical" program that could be used to design an optimal instruction set. For example, see the synthetic benchmarks discussed in Chapter 1. The data in this chapter clearly show that programs can vary significantly in how they use an instruction set. For example, Figure 2.38 shows the mix of data transfer sizes for four of the SPEC2000 programs: It would be hard to say what is typical from these four programs. The variations are even larger on an instruction set that supports a class of applications, such as decimal instructions, that are unused by other applications.

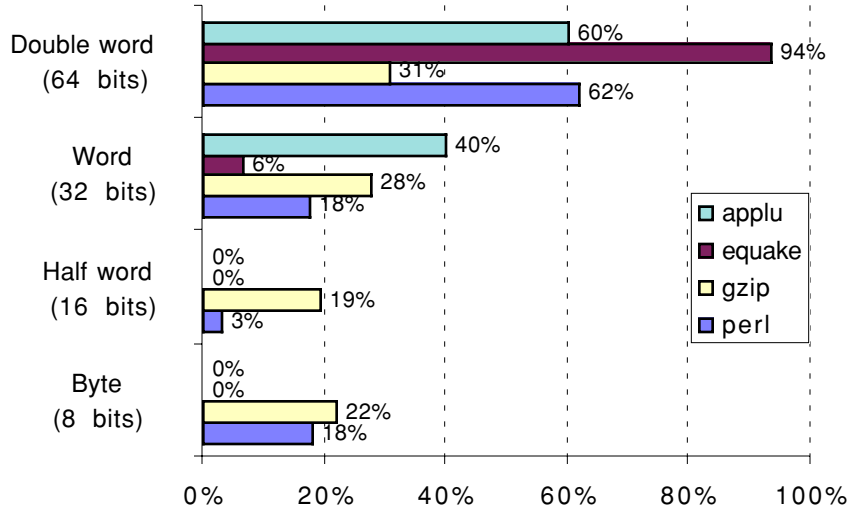


FIGURE 2.38 Data reference size of four programs from SPEC2000. Although you can calculate an average size, it would be hard to claim the average is typical of programs. **<<Artist: make data label font smaller>>**

Pitfall: Innovating at the instruction set architecture to reduce code size without accounting for the compiler.

Figure 2.39 shows the relative code sizes for four compilers for the MIPS instruction set. Whereas architects struggle to reduce code size by 30% to 40%, different compiler strategies can change code size by much larger factors. Similar to performance optimization techniques, the architect should start with the tightest code the compilers can produce before proposing hardware innovations to save space.

Compiler	Apogee Software: Version 4.1	Green Hills: Multi2000 Version 2.0	Algorithmics SDE4.0B	IDT/c 7.2.1
Architecture	MIPS IV	MIPS IV	MIPS 32	MIPS 32
Processor	NEC VR5432	NEC VR5000	IDT 32334	IDT 79RC32364
Auto Correlation kernel	1.0	2.1	1.1	2.7
Convolutional Encoder kernel	1.0	1.9	1.2	2.4
Fixed-Point Bit Allocation kernel	1.0	2.0	1.2	2.3
Fixed-Point Complex FFT kernel	1.0	1.1	2.7	1.8
Viterbi GSM Decoder kernel	1.0	1.7	0.8	1.1
Geometric Mean of 5 kernels	1.0	1.7	1.4	2.0

FIGURE 2.39 Code size relative to Apogee Software Version 4.1 C compiler for Telecom application of EEMBC benchmarks. The instruction set architectures are virtually identical, yet the code sizes vary by factors of two. These results were reported February to June 2000.

Pitfall: Expecting to get good performance from a compiler for DSPs.

Figure 2.40 shows the performance improvement to be gained by using assembly language, versus compiling from C for two Texas Instruments DSPs. Assembly language programming gains factors of 3 to 10 in performance and factors of 1 to 8 in code size. This gain is large enough to lure DSP programmers away from high-level languages, despite their well-documented advantages in programmer productivity and software maintenance.

Fallacy: An architecture with flaws cannot be successful.

The 80x86 provides a dramatic example: The instruction set architecture is one only its creators could love (see Appendix C). Succeeding generations of Intel engineers have tried to correct unpopular architectural decisions made in designing the 80x86. For example, the 80x86 supports segmentation, whereas all others picked paging; it uses extended accumulators for integer data, but other processors use general-purpose registers; and it uses a stack for floating-point data, when everyone else abandoned execution stacks long before.

TMS320C54 D ("C54") for DSPstone kernels	ratio to assembly in execution time (> 1 means slower)	ratio to assembly code space (> 1 means bigger)	TMS 320C6203("C62") for EEMBC Telecom kernels	ratio to assembly in execution time (> 1 means slower)	ratio to assembly code space (> 1 means bigger)
Convolution	11.8	16.5	Convolutional Encoder	44.0	0.5
FIR	11.5	8.7	Fixed-Point Complex FFT	13.5	1.0
Matrix 1x3	7.7	8.1	Viterbi GSM Decoder	13.0	0.7
FIR2dim	5.3	6.5	Fixed-point Bit Allocation	7.0	1.4
Dot product	5.2	14.1	Auto Collrelation	1.8	0.7
LMS	5.1	0.7			
N real update	4.7	14.1			
IIR n biquad	2.4	8.6			
N complex update	2.4	9.8			
Matrix	1.2	5.1			
Complex update	1.2	8.7			
IIR one biquad	1.0	6.4			
Real update	0.8	15.6			
C54 Geometric Mean	3.2	7.8	C62 Geometric Mean	10.0	0.8

FIGURE 2.40 Ratio of execution time and code size for compiled code vs. hand written code for TMS320C54 DSPs on left (using the 14 DSPstone kernels) and Texas Instruments TMS 320C6203 on right (using the 6 EEMBC Telecom kernels). The geometric mean of performance improvements is 3.2:1 for C54 running DSPstone and 10.0:1 for the C62 running EEMBC. The compiler does a better job on code space for the C62, which is a VLIW processor, but the geometric mean of code size for the C54 is almost a factor of 8 larger when compiled. Modifying the C code gives much better results. The EEMBC results were reported May 2000. For DSPstone, see Ropers [1999]

Despite these major difficulties, the 80x86 architecture has been enormously successful. The reasons are threefold: first, its selection as the microprocessor in the initial IBM PC makes 80x86 binary compatibility extremely valuable. Second, Moore's Law provided sufficient resources for 80x86 microprocessors to translate to an internal RISC instruction set and then execute RISC-like instructions (see section 3.8 in the next chapter). This mix enables binary compatibility with the valuable PC software base and performance on par with RISC processors. Third, the very high volumes of PC microprocessors means Intel can easily pay for the increased design cost of hardware translation. In addition, the high volumes allow the manufacturer to go up the learning curve, which lowers the cost of the product.

The larger die size and increased power for translation may be a liability for embedded applications, but it makes tremendous economic sense for the desktop. And its cost-performance in the desktop also makes it attractive for servers, with its main weakness for servers being 32-bit addresses: companies already offer high-end servers with more than one terabyte (2^{40} bytes) of memory.

Fallacy: You can design a flawless architecture.

All architecture design involves trade-offs made in the context of a set of hardware and software technologies. Over time those technologies are likely to change, and decisions that may have been correct at the time they were made look like mistakes. For example, in 1975 the VAX designers overemphasized the importance of code-size efficiency, underestimating how important ease of decoding and pipelining would be five years later. An example in the RISC camp is delayed branch (see Appendix B <RISC>). It was a simple to control pipeline hazards with five-stage pipelines, but a challenge for processors with longer pipelines that issue multiple instructions per clock cycle. In addition, almost all architectures eventually succumb to the lack of sufficient address space.

In general, avoiding such flaws in the long run would probably mean compromising the efficiency of the architecture in the short run, which is dangerous, since a new instruction set architecture must struggle to survive its first few years.

2.15 Concluding Remarks

The earliest architectures were limited in their instruction sets by the hardware technology of that time. As soon as the hardware technology permitted, computer architects began looking for ways to support high-level languages. This search led to three distinct periods of thought about how to support programs efficiently. In the 1960s, stack architectures became popular. They were viewed as being a good match for high-level languages—and they probably were, given the compiler technology of the day. In the 1970s, the main concern of architects was how to reduce software costs. This concern was met primarily by replacing software with hardware, or by providing high-level architectures that could simplify the task of software designers. The result was both the high-level-language computer architecture movement and powerful architectures like the VAX, which has a large number of addressing modes, multiple data types, and a highly orthogonal architecture. In the 1980s, more sophisticated compiler technology and a renewed emphasis on processor performance saw a return to simpler architectures, based mainly on the load-store style of computer.

The following instruction set architecture changes occurred in the 1990s:

- *Address size doubles*: The 32-bit address instruction sets for most desktop and server processors were extended to 64-bit addresses, expanding the width of the registers (among other things) to 64 bits. Appendix B <RISC> gives three examples of architectures that have gone from 32 bits to 64 bits.
- *Optimization of conditional branches via conditional execution*: In the next two chapters we see that conditional branches can limit the performance of aggressive computer designs. Hence, there was interest in replacing conditional branches with conditional completion of operations, such as conditional move (see Chapter 4), which was added to most instruction sets.

- *Optimization of cache performance via prefetch*: Chapter 5 explains the increasing role of memory hierarchy in performance of computers, with a cache miss on some computers taking as many instruction times as page faults took on earlier computers. Hence, prefetch instructions were added to try to hide the cost of cache misses by prefetching (see Chapter 5).
- *Support for multimedia*: Most desktop and embedded instruction sets were extended with support for multimedia and DSP applications, as discussed in this chapter.
- *Faster floating-point Operations*: Appendix G <Float> describes operations added to enhance floating-point performance, such as operations that perform a multiply and an add and paired single execution. (We include them in MIPS.)

Looking to the next decade, we see the following trends in instruction set design:

- *Long Instruction Words*: The desire to achieve more instruction level parallelism by making changing the architecture to support wider instructions (see Chapter 4).
- *Increased Conditional Execution*: More support for conditional execution of operations to support greater speculation.
- *Blending of general purpose and DSP architectures*: Parallel efforts between desktop and embedded processors to add DSP support vs. extending DSP processors to make them better targets for compilers, suggesting a culture clash in the marketplace between general purpose and DSPs.
- *80x86 emulation*: Given the popularity of software for the 80x86 architecture, many companies are looking to see if changes to the instruction sets can significantly improve performance, cost, or power when emulating the 80x86 architecture.

Between 1970 and 1985 many thought the primary job of the computer architect was the design of instruction sets. As a result, textbooks of that era emphasize instruction set design, much as computer architecture textbooks of the 1950s and 1960s emphasized computer arithmetic. The educated architect was expected to have strong opinions about the strengths and especially the weaknesses of the popular computers. The importance of binary compatibility in quashing innovations in instruction set design was unappreciated by many researchers and textbook writers, giving the impression that many architects would get a chance to design an instruction set.

The definition of computer architecture today has been expanded to include design and evaluation of the full computer system—not just the definition of the instruction set and not just the processor—and hence there are plenty of topics for the architect to study. (You may have guessed this the first time you lifted this book.) Hence, the bulk of this book is on design of computers versus instruction sets.

The many appendices may satisfy readers interested in instruction set architecture: Appendix B compares seven popular load-store computers with MIPS. Appendix C describes the most widely used instruction set, the Intel 80x86, and compares instruction counts for it with that of MIPS for several programs. For those interested in the historical computers, Appendix D summarizes the VAX architecture and Appendix E summarizes the IBM 360/370.

2.16 Historical Perspective and References

One's eyebrows should rise whenever a future architecture is developed with a stack- or register-oriented instruction set. [p. 20]

Meyers [1978]

The earliest computers, including the UNIVAC I, the EDSAC, and the IAS computers, were accumulator-based computers. The simplicity of this type of computer made it the natural choice when hardware resources were very constrained. The first general-purpose register computer was the Pegasus, built by Ferranti, Ltd. in 1956. The Pegasus had eight general-purpose registers, with R0 always being zero. Block transfers loaded the eight registers from the drum memory.

Stack Architectures

In 1963, Burroughs delivered the B5000. The B5000 was perhaps the first computer to seriously consider software and hardware-software trade-offs. Barton and the designers at Burroughs made the B5000 a stack architecture (as described in Barton [1961]). Designed to support high-level languages such as ALGOL, this stack architecture used an operating system (MCP) written in a high-level language. The B5000 was also the first computer from a U.S. manufacturer to support virtual memory. The B6500, introduced in 1968 (and discussed in Hauck and Dent [1968]), added hardware-managed activation records. In both the B5000 and B6500, the top two elements of the stack were kept in the processor and the rest of the stack was kept in memory. The stack architecture yielded good code density, but only provided two high-speed storage locations. The authors of both the original IBM 360 paper [Amdahl, Blaauw, and Brooks 1964] and the original PDP-11 paper [Bell et al. 1970] argue against the stack organization. They cite three major points in their arguments against stacks:

1. Performance is derived from fast registers, not the way they are used.
2. The stack organization is too limiting and requires many swap and copy operations.
3. The stack has a bottom, and when placed in slower memory there is a performance loss.

Stack-based hardware fell out of favor in the late 1970s and, except for the Intel 80x86 floating-point architecture, essentially disappeared. For example, except for the 80x86, none of the computers listed in the SPEC report uses a stack.

In the 1990s, however, stack architectures received a shot in the arm with the success of Java Virtual Machine (JVM). The JVM is a software interpreter for an intermediate language produced by Java compilers, called *Java bytecodes* ([Lindholm 1999]). The purpose of the interpreter is to provide software compatibility across many platforms, with the hope of "write once, run everywhere." Although the slowdown is about a factor of ten due to interpretation, there are times when compatibility is more important than performance, such as when downloading a Java "applet" into an Internet browser.

Although a few have proposed hardware to directly execute the JVM instructions (see [McGhan 1998]), thus far none of these proposals have been significant commercially. The hope instead is that *Just In Time (JIT) Java compilers*—which compile during run time to the native instruction set of the computer running the Java program—will overcome the performance penalty of interpretation. The popularity of Java has also lead to compilers that compile directly into the native hardware instruction sets, bypassing the illusion of the Java bytecodes.

Computer Architecture Defined

IBM coined the term *computer architecture* in the early 1960s. Amdahl, Blaauw, and Brooks [1964] used the term to refer to the programmer-visible portion of the IBM 360 instruction set. They believed that a *family* of computers of the same architecture should be able to run the same software. Although this idea may seem obvious to us today, it was quite novel at that time. IBM, although it was the leading company in the industry, had *five* different architectures before the 360. Thus, the notion of a company standardizing on a single architecture was a radical one. The 360 designers hoped that defining a common architecture would bring six different divisions of IBM together. Their definition of architecture was

... the structure of a computer that a machine language programmer must understand to write a correct (timing independent) program for that machine.

The term "machine language programmer" meant that compatibility would hold, even in machine language, while "timing independent" allowed different implementations. This architecture blazed the path for binary compatibility, which others have followed.

The IBM 360 was the first computer to sell in large quantities with both byte addressing using 8-bit bytes and general-purpose registers. The 360 also had register-memory and limited memory-memory instructions. Appendix E <IBM> summarizes this instruction set.

In 1964, Control Data delivered the first supercomputer, the CDC 6600. As Thornton [1964] discusses, he, Cray, and the other 6600 designers were among the first to explore pipelining in depth. The 6600 was the first general-purpose,

load-store computer. In the 1960s, the designers of the 6600 realized the need to simplify architecture for the sake of efficient pipelining. Microprocessor and minicomputer designers largely neglected this interaction between architectural simplicity and implementation during the 1970s, but it returned in the 1980s.

High Level Language Computer Architecture

In the late 1960s and early 1970s, people realized that software costs were growing faster than hardware costs. McKeeman [1967] argued that compilers and operating systems were getting too big and too complex and taking too long to develop. Because of inferior compilers and the memory limitations of computers, most systems programs at the time were still written in assembly language. Many researchers proposed alleviating the software crisis by creating more powerful, software-oriented architectures. Tanenbaum [1978] studied the properties of high-level languages. Like other researchers, he found that most programs are simple. He then argued that architectures should be designed with this in mind and that they should optimize for program size and ease of compilation. Tanenbaum proposed a stack computer with frequency-encoded instruction formats to accomplish these goals. However, as we have observed, program size does not translate directly to cost/performance, and stack computers faded out shortly after this work.

Strecker's article [1978] discusses how he and the other architects at DEC responded to this by designing the VAX architecture. The VAX was designed to simplify compilation of high-level languages. Compiler writers had complained about the lack of complete orthogonality in the PDP-11. The VAX architecture was designed to be highly orthogonal and to allow the mapping of a high-level-language statement into a single VAX instruction. Additionally, the VAX designers tried to optimize code size because compiled programs were often too large for available memories. Appendix D <Vax> summarizes this instruction set.

The VAX-11/780 was the first computer announced in the VAX series. It is one of the most successful—and most heavily studied—computers ever built. The cornerstone of DEC's strategy was a single architecture, VAX, running a single operating system, VMS. This strategy worked well for over 10 years. The large number of papers reporting instruction mixes, implementation measurements, and analysis of the VAX make it an ideal case study [Wiecek 1982; Clark and Levy 1982]. Bhandarkar and Clark [1991] give a quantitative analysis of the disadvantages of the VAX versus a RISC computer, essentially a technical explanation for the demise of the VAX.

While the VAX was being designed, a more radical approach, called *high-level-language computer architecture* (HLLCA), was being advocated in the research community. This movement aimed to eliminate the gap between high-level languages and computer hardware—what Gagliardi [1973] called the "semantic gap"—by bringing the hardware "up to" the level of the programming language. Meyers [1982] provides a good summary of the arguments and a history of high-level-language computer architecture projects.

HLLCA never had a significant commercial impact. The increase in memory size on computers eliminated the code-size problems arising from high-level languages and enabled operating systems to be written in high-level languages. The combination of simpler architectures together with software offered greater performance and more flexibility at lower cost and lower complexity.

Reduced Instruction Set Computers

In the early 1980s, the direction of computer architecture began to swing away from providing high-level hardware support for languages. Ditzel and Patterson [1980] analyzed the difficulties encountered by the high-level-language architectures and argued that the answer lay in simpler architectures. In another paper [Patterson and Ditzel 1980], these authors first discussed the idea of reduced instruction set computers (RISC) and presented the argument for simpler architectures. Clark and Strecker [1980], who were VAX architects, rebutted their proposal.

The simple load-store computers such as MIPS are commonly called RISC architectures. The roots of RISC architectures go back to computers like the 6600, where Thornton, Cray, and others recognized the importance of instruction set simplicity in building a fast computer. Cray continued his tradition of keeping computers simple in the CRAY-1. Commercial RISCs are built primarily on the work of three research projects: the Berkeley RISC processor, the IBM 801, and the Stanford MIPS processor. These architectures have attracted enormous industrial interest because of claims of a performance advantage of anywhere from two to five times over other computers using the same technology.

Begun in 1975, the IBM project was the first to start but was the last to become public. The IBM computer was designed as 24-bit ECL minicomputer, while the university projects were both MOS-based, 32-bit microprocessors. John Cocke is considered the father of the 801 design. He received both the Eckert-Mauchly and Turing awards in recognition of his contribution. Radin [1982] describes the highlights of the 801 architecture. The 801 was an experimental project that was never designed to be a product. In fact, to keep down cost and complexity, the computer was built with only 24-bit registers.

In 1980, Patterson and his colleagues at Berkeley began the project that was to give this architectural approach its name (see Patterson and Ditzel [1980]). They built two computers called RISC-I and RISC-II. Because the IBM project was not widely known or discussed, the role played by the Berkeley group in promoting the RISC approach was critical to the acceptance of the technology. They also built one of the first instruction caches to support hybrid format RISCs (see Patterson [1983]). It supported 16-bit and 32-bit instructions in memory but 32 bits in the cache (see Patterson [1983]). The Berkeley group went on to build RISC computers targeted toward Smalltalk, described by Ungar et al. [1984], and LISP, described by Taylor et al. [1986].

In 1981, Hennessy and his colleagues at Stanford published a description of the Stanford MIPS computer. Efficient pipelining and compiler-assisted scheduling of the pipeline were both important aspects of the original MIPS design. MIPS stood for Microprocessor without Interlocked Pipeline Stages, reflecting the lack of hardware to stall the pipeline, as the compiler would handle dependencies.

These early RISC computers—the 801, RISC-II, and MIPS—had much in common. Both university projects were interested in designing a simple computer that could be built in VLSI within the university environment. All three computers used a simple load-store architecture, fixed-format 32-bit instructions, and emphasized efficient pipelining. Patterson [1985] describes the three computers and the basic design principles that have come to characterize what a RISC computer is. Hennessy [1984] provides another view of the same ideas, as well as other issues in VLSI processor design.

In 1985, Hennessy published an explanation of the RISC performance advantage and traced its roots to a substantially lower CPI—under 2 for a RISC processor and over 10 for a VAX-11/780 (though not with identical workloads). A paper by Emer and Clark [1984] characterizing VAX-11/780 performance was instrumental in helping the RISC researchers understand the source of the performance advantage seen by their computers.

Since the university projects finished up, in the 1983–84 time frame, the technology has been widely embraced by industry. Many manufacturers of the early computers (those made before 1986) claimed that their products were RISC computers. These claims, however, were often born more of marketing ambition than of engineering reality.

In 1986, the computer industry began to announce processors based on the technology explored by the three RISC research projects. Moussouris et al. [1986] describe the MIPS R2000 integer processor, while Kane's book [1986] is a complete description of the architecture. Hewlett-Packard converted their existing minicomputer line to RISC architectures; Lee [1989] describes the HP Precision Architecture. IBM never directly turned the 801 into a product. Instead, the ideas were adopted for a new, low-end architecture that was incorporated in the IBM RT-PC and described in a collection of papers [Waters 1986]. In 1990, IBM announced a new RISC architecture (the RS 6000), which is the first superscalar RISC processor (see Chapter 4). In 1987, Sun Microsystems began delivering computers based on the SPARC architecture, a derivative of the Berkeley RISC-II processor; SPARC is described in Garner et al. [1988]. The PowerPC joined the forces of Apple, IBM, and Motorola. Appendix B <RISC> summarizes several RISC architectures.

To help resolve the RISC vs. traditional design debate, designers of VAX processors later performed a quantitative comparison of VAX and a RISC processor for implementations with comparable organizations. Their choices were the VAX 8700 and the MIPS M2000. The differing goals for VAX and MIPS have led to very different architectures. The VAX goals, simple compilers and code density,

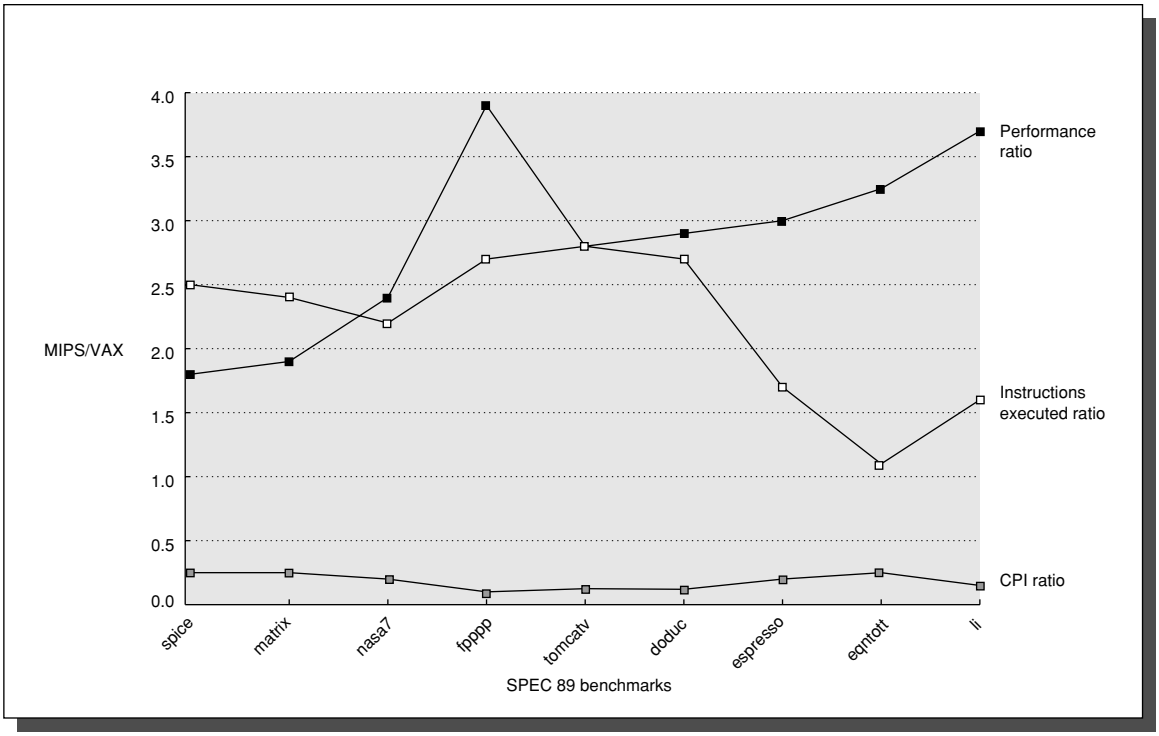


FIGURE 2.41 Ratio of MIPS M2000 to VAX 8700 in instructions executed and performance in clock cycles using SPEC89 programs. On average, MIPS executes a little over twice as many instructions as the VAX, but the CPI for the VAX is almost six times the MIPS CPI, yielding almost a threefold performance advantage. (Based on data from Bhandarkar and Clark [1991].)

led to powerful addressing modes, powerful instructions, efficient instruction encoding, and few registers. The MIPS goals were high performance via pipelining, ease of hardware implementation, and compatibility with highly optimizing compilers. These goals led to simple instructions, simple addressing modes, fixed-length instruction formats, and a large number of registers.

Figure 2.41 shows the ratio of the number of instructions executed, the ratio of CPIs, and the ratio of performance measured in clock cycles. Since the organizations were similar, clock cycle times were assumed to be the same. MIPS executes about twice as many instructions as the VAX, while the CPI for the VAX is about six times larger than that for the MIPS. Hence, the MIPS M2000 has almost three times the performance of the VAX 8700. Furthermore, much less hardware is needed to build the MIPS processor than the VAX processor. This cost/performance gap is the reason the company that used to make the VAX has dropped it and is now making the Alpha, which is quite similar to MIPS. Bell and Strecker summarize the debate inside the company.

Looking back, only one CISC instruction set survived the RISC/CISC debate, and that one that had binary compatibility with PC-software. The volume of chips is so high in the PC industry that there is sufficient revenue stream to pay the ex-

tra design costs—and sufficient resources due to Moore's Law—to build microprocessors which translate from CISC to RISC internally. Whatever loss in efficiency, due to longer pipeline stages and bigger die size to accommodate translation on the chip, was hedged by having a semiconductor fabrication line dedicated to producing just these microprocessors. The high volumes justify the economics of a fab line tailored to these chips.

Thus, in the desktop/server market, RISC computers use compilers to translate into RISC instructions and the remaining CISC computer uses hardware to translate into RISC instructions. One recent novel variation for the laptop market is the Transmeta Crusoe (see section 4.8 of Chapter 4), which interprets 80x86 instructions and compiles on the fly into internal instructions.

The embedded market, which competes in cost and power, cannot afford the luxury of hardware translation and thus uses compilers and RISC architectures. More than twice as many 32-bit embedded microprocessors were shipped in 2000 than PC microprocessors, with RISC processors responsible for over 90% of that embedded market.

A Brief History of Digital Signal Processors

(Jeff Bier prepared this DSP history.)

In the late 1990s, digital signal processing (DSP) applications, such as digital cellular telephones, emerged as one of the largest consumers of embedded computing power. Today, microprocessors specialized for DSP applications —sometimes called digital signal processors, DSPs, or DSP processors—are used in most of these applications. In 2000 this was a $6 billion market. Compared to other embedded computing applications, DSP applications are differentiated by:

- Computationally demanding, iterative numeric algorithms often composed of vector dot products; hence the importance of multiply and multiply-accumulate instructions.
- Sensitivity to small numeric errors; for example, numeric errors may manifest themselves as audible noise in an audio device.
- Stringent real-time requirements.
- "Streaming" data; typically, input data is provided from an analog-to-digital converter as a infinite stream. Results are emitted in a similar fashion.
- High data bandwidth.
- Predictable, simple (though often eccentric) memory access patterns.
- Predictable program flow (typically characterized by nested loops).

In the 1970s there was strong interest in using DSP techniques in telecommunications equipment, such as modems and central office switches. The microprocessors of the day did not provide adequate performance, though. Fixed-function

hardware proved effective in some applications, but lacked the flexibility and reusability of a programmable processor. Thus, engineers were motivated to adapt microprocessor technology to the needs of DSP applications.

The first commercial DSPs emerged in the early 1980s, about 10 years after Intel's introduction of the 4004. A number of companies, including Intel, developed early DSPs, but most of these early devices were not commercially successful. NEC's μPD7710, introduced in 1980, became the first merchant-market DSP to ship in volume quantities, but was hampered by weak development tools. AT&T's DSP1, also introduced in 1980, was limited to use within AT&T, but it spawned several generations of successful devices which AT&T soon began offering to other system manufacturers. In 1982, Texas Instruments introduced its first DSP, the TMS32010. Backed by strong tools and applications engineering support, the TI processor was a solid success.

Like the first microprocessors, these early DSPs had simple architectures. In contrast with their general-purpose cousins, though, DSPs adopted a range of specialized features to boost performance and efficiency in signal processing tasks. For example, a single-cycle multiplier aided arithmetic performance. Specialized datapaths streamlined multiply-accumulate operations and provided features to minimize numeric errors, such as saturation arithmetic. Separate program and data memories provided the memory bandwidth required to keep the relatively powerful datapaths fed. Dedicated, specialized addressing hardware sped simple addressing operations, such autoincrement addressing. Complex, specialized instruction sets allowed these processors to combine many operations in a single instruction, but only certain limited combinations of operations were supported.

From the mid 1980s to the mid 1990s, many new commercial DSP architectures were introduced. For the most part, these architectures followed a gradual, evolutionary path, adopting incremental improvements rather than fundamental innovations when compared with the earliest DSPs like the TMS32010. DSP application programs expanded from a few hundred lines of source code to tens of thousands of lines. Hence, the quality of development tools and the availability of off-the-shelf application software components became, for many users, more important than performance in selecting a processor. Today, chips based on these "conventional DSP" architectures still dominate DSP applications, and are used in products such as cellular telephones, disk drives (for servo control), and consumer audio devices.

Early DSP architectures had proven effective, but the highly specialized and constrained instruction sets that gave them their performance and efficiency also created processors that were difficult targets for compiler writers. The performance and efficiency demands of most DSP applications could not be met by the resulting weak compilers, so much software—all software for some processor—was written in assembly language. As applications became larger and more complex, assembly language programming became less practical. Users also suffered from the incompatibility of many new DSP architectures with their predecessors, which forced them to periodically rewrite large amounts of existing application software.

In roughly 1995, architects of digital signal processors began to experiment with very different types of architectures, often adapting techniques from earlier high-performance general-purpose or scientific-application processor designs. These designers sought to further increase performance and efficiency, but to do so with architectures that would be better compiler targets, and that would offer a better basis for future compatible architectures. For example, in 1997, Texas Instruments announced the TMS320C62xx family, an eight-issue VLIW design boasting increased parallelism, a higher clock speed, and a radically simple, RISC-like instruction set. Other DSP architects adopted SIMD approaches, superscalar designs, chip multiprocessing, or a combination of these of techniques. Therefore, DSP architectures today are more diverse than ever, and the rate of architectural innovation is increasing.

DSP architects were experimenting with new approaches, often adapting techniques from general-purpose processors. In parallel, designers of general-purpose processors (both those targeting embedded applications and those intended for computers) noticed that DSP tasks were becoming increasingly common in all kinds of microprocessor applications. In many cases, these designers added features to their architectures to boost performance and efficiency in DSP tasks. These features ranged from modest instruction set additions to extensive architectural retrofits. In some cases, designers created all-new architectures intended to encompass capabilities typically found in a DSP and those typically found in a general-purpose processor. Today, virtually every commercial 32-bit microprocessor architecture—from ARM to 80x86—has been subject to some kind of DSP-oriented enhancement.

Throughout the 1990s, an increasing number of system designers turned to system-on-chip devices. These are complex integrated circuits typically containing a processor core and a mix of memory, application-specific hardware (such as algorithm accelerators), peripherals, and I/O interfaces tuned for a specific application. An example is second-generation cellular phones. In some cases, chip manufacturers provide a complete complement of application software along with these highly integrated chips. These processor-based chips are often the solution of choice for relatively mature, high-volume applications. Though these chips are not sold as "processors," the processors inside them define their capabilities to a significant degree.

More information on the history of DSPs can be found Boddie [2000], Stauss [1998], and Texas Instruments [2000].

Multimedia Support in Desktop Instruction Sets

Since every desktop microprocessor by definition has its own graphical displays, as transistor budgets increased it was inevitable that support would be added for graphics operations. The earliest color for PCs used 8 bits per pixel in the "256 color" format of VGA, which some PCs still support for compatibility. The next step was 16 bits per pixel by encoding R in 5 bits, G in 6 bits, and B in 5 bits.

This format is called *high color* on PCs. On PCs the 32-bit format discussed above, with R, G, B, and A, is called *true color*.

The addition of speakers and microphones for teleconferencing and video games suggested support of sound as well. Audio samples of 16 bit are sufficient for most end users, but professional audio work uses 24 bits.

The architects of the Intel i860, which was justified as a graphical accelerator within the company, recognized that many graphics and audio applications would perform the same operation on vectors of these data. Although a vector unit was beyond the transistor budget of the i860 in 1989, by partitioning the carry chains within a 64-bit ALU, it could perform simultaneous operations on short vectors. It operated on eight 8-bit operands, four 16-bit operands, or two 32-bit operands. The cost of such partitioned ALUs was small. Applications that lend themselves to such support include MPEG (video), games like DOOM (3D graphics), Adobe Photoshop (digital photography), and teleconferencing (audio and image processing). Operations on four 8-bit operands were for operating on pixels.

Like a virus, over time such multimedia support has spread to nearly every desktop microprocessor. HP was the first successful desktop RISC to include such support. The pair single floating-point operations, which came later, are useful for operations on vertices.

These extensions have been called partitioned ALU, subword parallelism, vector, or SIMD (single instruction, multiple data). Since Intel marketing uses SIMD to describe the MMX extension of the 80x86, SIMD has become the popular name.

Summary

Prior to the RISC architecture movement, the major trend had been highly microcoded architectures aimed at reducing the semantic gap and code size. DEC, with the VAX, and Intel, with the iAPX 432, were among the leaders in this approach.

Although those two computers have faded into history, one contemporary survives: the 80x86. This architecture did not have a philosophy about high level language, it had a deadline. Since the iAPX 432 was late and Intel desperately needed a 16-bit microprocessor, the 8086 was designed in a few months. It was forced to be assembly language compatible with the 8-bit 8080, and assembly language was expected to be widely used with this architecture. Its saving grace has been its ability to evolve.

The 80x86 dominates the desktop with an 85% share, which is due in part to the importance of binary compatibility as a result of IBM's selection of the 8086 in the early 1980s. Rather than change the instruction set architecture, recent 80x86 implementations translate into RISC-like instructions internally and then execute them (see section 3.8 in the next chapter). RISC processors dominate the embedded market with a similar market share, because binary compatibility is unimportant plus die size and power goals make hardware translation a luxury.

VLIW is currently being tested across the board, from DSPs to servers. Will code size be a problem in the embedded market, where the instruction memory in a chip could be bigger than the processor? Will VLIW DSPs achieve respectable cost-performance if compilers to produce the code? Will the high power and large die of server VLIWs be successful, at a time when concern for power efficiency of servers is increasing? Once again an attractive feature of this field is that time will shortly tell how VLIW fares, and we should know answers to these questions by the fourth edition of this book.

References

AMDAHL, G. M., G. A. BLAAUW, AND F. P. BROOKS, JR. [1964]. "Architecture of the IBM System 360," *IBM J. Research and Development* 8:2 (April), 87–101.

BARTON, R. S. [1961]. "A new approach to the functional design of a computer," *Proc. Western Joint Computer Conf.,* 393–396.

Bier, J. [1997] "The Evolution of DSP Processors", presentation at U.C.Berkeley, November 14.

BELL, G., R. CADY, H. MCFARLAND, B. DELAGI, J. O'LAUGHLIN, R. NOONAN, AND W. WULF [1970]. "A new architecture for mini-computers: The DEC PDP-11," *Proc. AFIPS SJCC*, 657–675.

Bell, G. and W. D. Strecker [1998]. "Computer Structures: What Have We Learned from the PDP-11?" *25 Years of the International Symposia on Computer Architecture (Selected Papers*). ACM, 138-151.

BHANDARKAR, D., AND D. W. CLARK [1991]. "Performance from architecture: Comparing a RISC and a CISC with similar hardware organizations," *Proc. Fourth Conf. on Architectural Support for Programming Languages and Operating System*s, IEEE/ACM (April), Palo Alto, Calif., 310–19.

BODDIE, J.R. [2000] "HISTORY OF DSPS," HTTP://WWW.LUCENT.COM/MICRO/DSP/DSPHIST.HTML.

CHOW, F. C. [1983]. *A Portable Machine-Independent Global Optimizer—Design and Measurements*, Ph.D. Thesis, Stanford Univ. (December).

CLARK, D. AND H. LEVY [1982]. "Measurement and analysis of instruction set use in the VAX-11/780," *Proc. Ninth Symposium on Computer Architecture* (April), Austin, Tex., 9–17.

CLARK, D. AND W. D. STRECKER [1980]. "Comments on 'the case for the reduced instruction set computer'," *Computer Architecture News* 8:6 (October), 34–38.

CRAWFORD, J. AND P. GELSINGER [1988]. *Programming the 80386,* Sybex Books, Alameda, Calif.

DITZEL, D. R. AND D. A. PATTERSON [1980]. "Retrospective on high-level language computer architecture," in *Proc. Seventh Annual Symposium on Computer Architecture,* La Baule, France (June), 97–104.

EMER, J. S. AND D. W. CLARK [1984]. "A characterization of processor performance in the VAX-11/780," *Proc. 11th Symposium on Computer Architecture* (June), Ann Arbor, Mich., 301–310.

GAGLIARDI, U. O. [1973]. "Report of workshop 4–Software-related advances in computer hardware," *Proc. Symposium on the High Cost of Software*, Menlo Park, Calif., 99–120.

GAME, M. and A. BOOKER [1999]. "CodePack code compression for PowerPC processors," MicroNews, First Quarter 1999, Vol. 5, No. 1., http://www.chips.ibm.com/micronews/vol5_no1/codepack.html

GARNER, R., A. AGARWAL, F. BRIGGS, E. BROWN, D. HOUGH, B. JOY, S. KLEIMAN, S. MUNCHNIK, M. NAMJOO, D. PATTERSON, J. PENDLETON, AND R. TUCK [1988]. "Scalable processor architecture (SPARC)," *COMPCON, IEEE* (March), San Francisco, 278–283.

HAUCK, E. A., AND B. A. DENT [1968]. "Burroughs' B6500/B7500 stack mechanism," *Proc. AFIPS*

SJCC, 245–251.

HENNESSY, J. [1984]. "VLSI processor architecture," *IEEE Trans. on Computers* C-33:11 (December), 1221–1246.

HENNESSY, J. [1985]. "VLSI RISC processors," *VLSI Systems Design* VI:10 (October), 22–32.

HENNESSY, J., N. JOUPPI, F. BASKETT, AND J. GILL [1981]. "MIPS: A VLSI processor architecture," *Proc. CMU Conf. on VLSI Systems and Computations* (October), Computer Science Press, Rockville, MY.

Intel [2001] *Using MMX™ Instructions to Convert RGB To YUV Color Conversion*, http://cedar.intel.com/cgi-bin/ids.dll/content/content.jsp?cntKey=Legacy::irtm_AP548_9996&cntType=IDS_EDITORIAL

KANE, G. [1986]. *MIPS R2000 RISC Architecture*, Prentice Hall, Englewood Cliffs, N.J.

Kozyrakis, C. [2000] "Vector IRAM: A Media-oriented vector processor with embedded DRAM," presentation at Hot Chips 12 Conference, Palo Alto, CA, 13-15, 2000

LEE, R. [1989]. "Precision architecture," *Computer* 22:1 (January), 78–91.

LEVY, H. AND R. ECKHOUSE [1989]. *Computer Programming and Architecture: The VAX,* Digital Press, Boston.

Lindholm, T. and F. Yellin [1999]. The Java Virtual Machine Specification, second edition, Addison-Wesley. Also available online at http://java.sun.com/docs/books/vmspec/.

LUNDE, A. [1977]. "Empirical evaluation of some features of instruction set processor architecture," *Comm. ACM* 20:3 (March), 143–152.

McGhan, H.; O'Connor, M. [1998] "PicoJava: a direct execution engine for Java bytecode." *Computer*, vol.31, (no.10), Oct. 1998. p.22-30.

MCKEEMAN, W. M. [1967]. "Language directed computer design," *Proc. 1967 Fall Joint Computer Conf.,* Washington, D.C., 413–417.

MEYERS, G. J. [1978]. "The evaluation of expressions in a storage-to-storage architecture," *Computer Architecture News* 7:3 (October), 20–23.

MEYERS, G. J. [1982]. *Advances in Computer Architecture*, 2nd ed., Wiley, New York.

MOUSSOURIS, J., L. CRUDELE, D. FREITAS, C. HANSEN, E. HUDSON, S. PRZYBYLSKI, T. RIORDAN, AND C. ROWEN [1986]. "A CMOS RISC processor with integrated system functions," *Proc. COMPCON, IEEE* (March), San Francisco, 191.

PATTERSON, D. [1985]. "Reduced instruction set computers," *Comm. ACM* 28:1 (January), 8–21.

PATTERSON, D. A. AND D. R. DITZEL [1980]. "The case for the reduced instruction set computer," *Computer Architecture News* 8:6 (October), 25–33.

Patterson, D.A.; Garrison, P.; Hill, M.; Lioupis, D.; Nyberg, C.; Sippel, T.; Van Dyke, K. "Architecture of a VLSI instruction cache for a RISC," 10th Annual International Conference on Computer Architecture Conference Proceedings, Stockholm, Sweden, 13-16 June 1983, 108-16.

RADIN, G. [1982]. "The 801 minicomputer," *Proc. Symposium Architectural Support for Programming Languages and Operating Systems* (March), Palo Alto, Calif., 39–47.

Riemens, A. Vissers, K.A.; Schutten, R.J.; Sijstermans, F.W.; Hekstra, G.J.; La Hei, G.D. [1999] "Trimedia CPU64 application domain and benchmark suite." Proceedings 1999 IEEE International Conference on Computer Design: VLSI in Computers and Processors, ICCD'99, Austin, TX, USA, 10-13 Oct. 1999, 580-585.

Ropers, A. H.W. Lollman, and J. Wellhausen [1999] "DSPstone: Texas Instruments TMS320C54x", Technical Report Nr.IB 315 1999/9-ISS-Version 0.9, Aachen University of Technology, http://www.ert.rwth-aachen.de/Projekte/Tools/coal/dspstone_c54x/index.html

STRAUSS, W. "DSP Strategies 2002," Forward Concepts, 1998. http://www.usadata.com/market_research/spr_05/spr_r127-005.htm

STRECKER, W. D. [1978]. "VAX-11/780: A virtual address extension of the PDP-11 family," *Proc. AFIPS National Computer Conf.* 47, 967–980.

TANENBAUM, A. S. [1978]. "Implications of structured programming for machine architecture," *Comm. ACM* 21:3 (March), 237–246.

TAYLOR, G., P. HILFINGER, J. LARUS, D. PATTERSON, AND B. ZORN [1986]. "Evaluation of the SPUR LISP architecture," *Proc. 13th Symposium on Computer Architecture (*June), Tokyo.

TEXAS INSTRUMENTS [2000]. "History of Innovation: 1980s," http://www.ti.com/corp/docs/company/history/1980s.shtml.

THORNTON, J. E. [1964]. "Parallel operation in Control Data 6600," *Proc. AFIPS Fall Joint Computer Conf.* 26, part 2, 33–40.

UNGAR, D., R. BLAU, P. FOLEY, D. SAMPLES, AND D. PATTERSON [1984]. "Architecture of SOAR: Smalltalk on a RISC," *Proc. 11th Symposium on Computer Architecture* (June), Ann Arbor, Mich., 188–197.

van Eijndhoven, J.T.J.; Sijstermans, F.W.; Vissers, K.A.; Pol, E.J.D.; Tromp, M.I.A.; Struik, P.; Bloks, R.H.J.; van der Wolf, P.; Pimentel, A.D.; Vranken, H.P.E.[1999] "Trimedia CPU64 architecture," Proc. 1999 IEEE International Conference on Computer Design: VLSI in Computers and Processors, ICCD'99, Austin, TX, USA, 10-13 Oct. 1999, 586-592.

WAKERLY, J. [1989]. *Microcomputer Architecture and Programming,* J. Wiley, New York.

WATERS, F., ED. [1986]. *IBM RT Personal Computer Technology,* IBM, Austin, Tex., SA 23-1057.

WIECEK, C. [1982]. "A case study of the VAX 11 instruction set usage for compiler execution," *Proc. Symposium on Architectural Support for Programming Languages and Operating Systems* (March), IEEE/ACM, Palo Alto, Calif., 177–184.

WULF, W. [1981]. "Compilers and computer architecture," *Computer* 14:7 (July), 41–47.

EXERCISES

- Where do instruction sets come from? Since the earliest computes date from just after WWII, it should be possible to derive the ancestry of the instructions in modern computers. This project will take a good deal of delving into libraries and perhaps contacting pioneers, but see if you can derive the ancestry of the instructions in, say, MIPS.
- It would be nice to try to do some comparisons with media processors and DSPs. How about this. "Very long instruction word (VLIW) computers are discussed in Chapter 4, but increasingly DSPs and media processors are adopting this style of instruction set architecture. One example is the TI TMS320C6203. See if you can compare code size of VLIW to more traditional computers. One attempt would be to code a common kernel across several computers. Another would be to get access to compilers for each computer and compare code sizes. Based on your data, is VLIW an appropriate architecture for embedded applications? Why or why not?
- Explicit reference to example Trimedia code
- 2.1 Seems like a reasonable exercise, but make it second or third instead of leadoff?

2.1 [20/15/10] <2.3,2.12> We are designing instruction set formats for a load-store architecture and are trying to decide whether it is worthwhile to have multiple offset lengths for branches and memory references. We have decided that both branch and memory references can have only 0-, 8-, and 16-bit offsets. The length of an instruction would be equal to 16 bits + offset length in bits. ALU instructions will be 16 bits. Figure 2.42 contains the data in cumulative form. Assume an additional bit is needed for the sign on the offset.

For instruction set frequencies, use the data for MIPS from the average of the five benchmarks for the load-store computer in Figure 2.32. Assume that the miscellaneous instructions are all ALU instructions that use only registers.

Offset bits	Cumulative data references	Cumulative branches
0	30%	0%
1	34%	3%
2	35%	11%
3	40%	23%
4	47%	37%
5	54%	57%
6	60%	72%
7	67%	85%
8	72%	91%
9	73%	93%
10	74%	95%
11	75%	96%
12	77%	97%
13	88%	98%
14	92%	99%
15	100%	100%

FIGURE 2.42 The second and third columns contain the cumulative percentage of the data references and branches, respectively, that can be accommodated with the corresponding number of bits of magnitude in the displacement. These are the average distances of all programs in Figure 2.8.

a. [20] <2.3,2.12> Suppose offsets were permitted to be 0, 8, or 16 bits in length, including the sign bit. What is the average length of an executed instruction?

b. [15] <2.3,2.12> Suppose we wanted a fixed-length instruction and we chose a 24-bit instruction length (for everything, including ALU instructions). For every offset of longer than 8 bits, an additional instruction is required. Determine the number of instruction bytes fetched in this computer with fixed instruction size versus those fetched with a byte-variable-sized instruction as defined in part (a).

c. [10] <2.3,2.12> Now suppose we use a fixed offset length of 16 bits so that no addi-

tional instruction is ever required. How many instruction bytes would be required? Compare this result to your answer to part (b), which used 8-bit fixed offsets that used additional instruction words when larger offsets were required.

- OK exercise

2.2 [15/10] <2.2> Several researchers have suggested that adding a register-memory addressing mode to a load-store computer might be useful. The idea is to replace sequences of

```
LOAD    R1,0(Rb)
ADD     R2,R2,R1
```

by

```
ADD     R2,0(Rb)
```

Assume the new instruction will cause the clock cycle to increase by 10%. Use the instruction frequencies for the gcc benchmark on the load-store computer from Figure 2.32. The new instruction affects only the clock cycle and not the CPI.

a. [15] <2.2> What percentage of the loads must be eliminated for the computer with the new instruction to have at least the same performance?

b. [10] <2.2> Show a situation in a multiple instruction sequence where a load of R1 followed immediately by a use of R1 (with some type of opcode) could not be replaced by a single instruction of the form proposed, assuming that the same opcode exists.

- Classic exercise, although it has been a confusing to some in the past.

2.3 [20] <2.2> Your task is to compare the memory efficiency of four different styles of instruction set architectures. The architecture styles are

1. *Accumulator*—All operations occur between a single register and a memory location.
2. *Memory-memory*—All three operands of each instruction are in memory.
3. *Stack*—All operations occur on top of the stack. Only push and pop access memory; all other instructions remove their operands from stack and replace them with the result. The implementation uses a stack for the top two entries; accesses that use other stack positions are memory references.
4. *Load-store*—All operations occur in registers, and register-to-register instructions have three operands per instruction. There are 16 general-purpose registers, and register specifiers are 4 bits long.

To measure memory efficiency, make the following assumptions about all four instruction sets:

- The opcode is always 1 byte (8 bits).
- All memory addresses are 2 bytes (16 bits).
- All data operands are 4 bytes (32 bits).
- All instructions are an integral number of bytes in length.

There are no other optimizations to reduce memory traffic, and the variables `A`, `B`, `C`, and `D`

are initially in memory.

Invent your own assembly language mnemonics and write the best equivalent assembly language code for the high-level-language fragment given. Write the four code sequences for

```
A = B + C;
B = A + C;
D = A - B;
```

Calculate the instruction bytes fetched and the memory-data bytes transferred. Which architecture is most efficient as measured by code size? Which architecture is most efficient as measured by total memory bandwidth required (code + data)?

2.4 [Discussion] <2.2–2.14> What are the *economic* arguments (i.e., more computers sold) for and against changing instruction set architecture in desktop and server markets? What about embedded markets?

2.5 [25] <2.1–2.5> Find an instruction set manual for some older computer (libraries and private bookshelves are good places to look). Summarize the instruction set with the discriminating characteristics used in Figure 2.3. Write the code sequence for this computer for the statements in Exercise 2.3. The size of the data need not be 32 bits as in Exercise 2.3 if the word size is smaller in the older computer.

2.6 [20] <2.12> Consider the following fragment of C code:

```
for (i=0; i<=100; i++)
        {A[i] = B[i] + C;}
```

Assume that `A` and `B` are arrays of 32-bit integers, and `C` and `i` are 32-bit integers. Assume that all data values and their addresses are kept in memory (at addresses 0, 5000, 1500, and 2000 for `A`, `B`, `C`, and `i`, respectively) except when they are operated on. Assume that values in registers are lost between iterations of the loop.

Write the code for MIPS; how many instructions are required dynamically? How many memory-data references will be executed? What is the code size in bytes?

n Unlikely there is enough detail for people to write programs just from the Appendix.

2.7 [20] <App. D> Repeat Exercise 2.6, but this time write the code for the 80x86.

2.8 [20] <2.12> For this question use the code sequence of Exercise 2.6, but put the scalar data—the value of `i`, the value of `C`, and the addresses of the array variables (but not the actual array)—in registers and keep them there whenever possible.

Write the code for MIPS; how many instructions are required dynamically? How many memory-data references will be executed? What is the code size in bytes?

2.9 [20] <App. D> Make the same assumptions and answer the same questions as the prior exercise, but this time write the code for the 80x86.

2.10 [15] <2.12> When designing memory systems it becomes useful to know the frequency of memory reads versus writes and also accesses for instructions versus data. Using the

average instruction-mix information for MIPS in Figure 2.32, find

- the percentage of all memory accesses for data
- the percentage of data accesses that are reads
- the percentage of all memory accesses that are reads

Ignore the size of a datum when counting accesses.

2.11 [18] <2.12> Compute the effective CPI for MIPS using Figure 2.32. Suppose we have made the following measurements of average CPI for instructions:

Instruction	Clock cycles
All ALU instructions	1.0
Loads-stores	1.4
Conditional branches	
Taken	2.0
Not taken	1.5
Jumps	1.2

Assume that 60% of the conditional branches are taken and that all instructions in the miscellaneous category of Figure 2.32 are ALU instructions. Average the instruction frequencies of gcc and espresso to obtain the instruction mix.

2.12 [20/10] <2.3,2.12> Consider adding a new index addressing mode to MIPS. The addressing mode adds two registers and an 11-bit signed offset to get the effective address.

Our compiler will be changed so that code sequences of the form

```
ADD R1, R1, R2
LW  Rd, 100(R1)(or store)
```

will be replaced with a load (or store) using the new addressing mode. Use the overall average instruction frequencies from Figure 2.32 in evaluating this addition.

a. [20] <2.3,2.12> Assume that the addressing mode can be used for 10% of the displacement loads and stores (accounting for both the frequency of this type of address calculation and the shorter offset). What is the ratio of instruction count on the enhanced MIPS compared to the original MIPS?

b. [10] <2.3,2.12> If the new addressing mode lengthens the clock cycle by 5%, which computer will be faster and by how much?

2.13 [25/15] <2.11> Find a C compiler and compile the code shown in Exercise 2.6 for one of the computers covered in this book. Compile the code both optimized and unoptimized.

a. [25] <2.11> Find the instruction count, dynamic instruction bytes fetched, and data accesses done for both the optimized and unoptimized versions.

b. [15] <2.11> Try to improve the code by hand and compute the same measures as in

part (a) for your hand-optimized version.

2.14 [30] <2.12> Small synthetic benchmarks can be very misleading when used for measuring instruction mixes. This is particularly true when these benchmarks are optimized. In this exercise and Exercises 2.15–2.17, we want to explore these differences. These programming exercises can be done with any load-store processor.

Compile Whetstone with optimization. Compute the instruction mix for the top 20 most frequently executed instructions. How do the optimized and unoptimized mixes compare? How does the optimized mix compare to the mix for swim256 on the same or a similar processor?

2.15 [30] <2.12> Follow the same guidelines as the prior exercise, but this time use Dhrystone and compare it with gcc.

2.16 [30] <2.12> Many computer manufacturers now include tools or simulators that allow you to measure the instruction set usage of a user program. Among the methods in use are processor simulation, hardware-supported trapping, and a compiler technique that instruments the object-code module by inserting counters. Find a processor available to you that includes such a tool. Use it to measure the instruction set mix for one of TeX, gcc, or spice. Compare the results to those shown in this chapter.

2.17 [30] <2.3,2.12> MIPS has only three operand formats for its register-register operations. Many operations might use the same destination register as one of the sources. We could introduce a new instruction format into MIPS called R_2 that has only two operands and is a total of 24 bits in length. By using this instruction type whenever an operation had only two different register operands, we could reduce the instruction bandwidth required for a program. Modify the MIPS simulator to count the frequency of register-register operations with only two different register operands. Using the benchmarks that come with the simulator, determine how much more instruction bandwidth MIPS requires than MIPS with the R_2 format.

2.18 [25] <App. C> How much do the instruction set variations among the RISC processors discussed in Appendix C affect performance? Choose at least three small programs (e.g., a sort), and code these programs in MIPS and two other assembly languages. What is the resulting difference in instruction count?

3 Instruction-Level Parallelism and its Dynamic Exploitation

"Who's first?"

"America."

"Who's second?"

"Sir, there is no second."

Dialog between two observers of the sailing race later named "The America's Cup" and run every few years.

This quote was the inspiration for John Cocke's naming of the IBM research processor as "America." This processor was the precursor to the RS/6000 series and the first superscalar microprocessor.

3.1 Instruction-Level Parallelism: Concepts and Challenges 221
3.2 Overcoming Data Hazards with Dynamic Scheduling 231
3.3 Dynamic Scheduling: Examples and the Algorithm 239
3.4 Reducing Branch Costs with Dynamic Hardware Prediction 247
3.5 High Performance Instruction Delivery 261
3.6 Taking Advantage of More ILP with Multiple Issue 268
3.7 Hardware-Based Speculation 278
3.8 Studies of the Limitations of ILP 294
3.9 Limitations on ILP for Realizable Processors 309
3.10 Putting It All Together: The P6 Microarchitecture 316
3.11 Another View: Thread Level Parallelism 329
3.12 Crosscutting Issues: Using an ILP Datapath to Exploit TLP 330
3.13 Fallacies and Pitfalls 330
3.14 Concluding Remarks 333
3.15 Historical Perspective and References 337

Exercises 345

3.1 Instruction-Level Parallelism: Concepts and Challenges

All processors since about 1985, including those in the embedded space, use pipelining to overlap the execution of instructions and improve performance. This potential overlap among instructions is called *instruction-level parallelism* (ILP) since the instructions can be evaluated in parallel. In this chapter and the next, we look at a wide range of techniques for extending the pipelining ideas by increasing the amount of parallelism exploited among instructions. This chapter is at a considerably more advanced level than the material in Appendix A. If you are not familiar with the ideas in Appendix A, you should review that Appendix before venturing into this chapter.

We start this chapter by looking at the limitation imposed by data and control hazards and then turn to the topic of increasing the ability of the processor to exploit parallelism. Section 3.1 introduces a large number of concepts, which we build on throughout these two chapters. While some of the more basic material in

this chapter could be understood without all of the ideas in Section 3.1, this basic material is important to later sections of this chapter as well as to chapter 4.

There are two largely separable approaches to exploiting ILP. This chapter covers techniques that are largely dynamic and depend on the hardware to locate the parallelism. The next chapter focuses on techniques that are static and rely much more on software. In practice, this partitioning between dynamic and static and between hardware-intensive and software-intensive is not clean, and techniques from one camp are often used by the other. Nonetheless, for exposition purposes, we have separated the two approaches and tried to indicate where an approach is transferable.

The dynamic, hardware intensive approaches dominate the desktop and server markets and are used in a wide range of processors, including: the Pentium III and 4, the Althon, the MIPS R10000/12000, the Sun ultraSPARC III, the PowerPC 603, G3, and G4, and the Alpha 21264. The static, compiler-intensive approaches, which we focus on in the next chapter, have seen broader adoption in the embedded market than the desktop or server markets, although the new IA-64 architecture and Intel's Itanium, use this more static approach.

In this section, we discuss features of both programs and processors that limit the amount of parallelism that can be exploited among instructions, as well as the critical mapping between program structure and hardware structure, which is key to understanding whether a program property will actually limit performance and under what circumstances.

Recall that the value of the CPI (Cycles per Instruction) for a pipelined processor is the sum of the base CPI and all contributions from stalls:

$$\text{Pipeline CPI} = \text{Ideal pipeline CPI} + \text{Structural stalls} + \text{Data hazard stalls} + \text{Control stalls}$$

The *ideal pipeline CPI* is a measure of the maximum performance attainable by the implementation. By reducing each of the terms of the right-hand side, we minimize the overall pipeline CPI and thus increase the IPC (Instructions per Clock). In this chapter we will see that the techniques we introduce to increase the ideal IPC, can increase the importance of dealing with structural, data hazard, and control stalls. The equation above allows us to characterize the various techniques we examine in this chapter by what component of the overall CPI a technique reduces. Figure 3.1 shows the techniques we examine in this chapter and in the next, as well as the topics covered in the introductory material in Appendix A.

Before we examine these techniques in detail, we need to define the concepts on which these techniques are built. These concepts, in the end, determine the limits on how much parallelism can be exploited.

Instruction-Level Parallelism

All the techniques in this chapter and the next exploit parallelism among instructions. As we stated above, this type of parallelism is called instruction-level parallelism or ILP. The amount of parallelism available within a *basic block*–a straight-

Technique	Reduces	Section
Forwarding and bypassing	Potential data hazard stalls	A.2
Delayed branches and simple branch scheduling	Control hazard stalls	A.2
Basic dynamic scheduling (scoreboarding)	Data hazard stalls from true dependences	A.8
Dynamic scheduling with renaming	Data hazard stalls and stalls from antidependences and output dependences	3.2
Dynamic branch prediction	Control stalls	3.4
Issuing multiple instructions per cycle	Ideal CPI	3.6
Speculation	Data hazard and control hazard stalls	3.5
Dynamic memory disambiguation	Data hazard stalls with memory	3.2, 3.7
Loop unrolling	Control hazard stalls	4.1
Basic compiler pipeline scheduling	Data hazard stalls	A.2, 4.1
Compiler dependence analysis	Ideal CPI, data hazard stalls	4.4
Software pipelining, trace scheduling	Ideal CPI, data hazard stalls	4,3
Compiler speculation	Ideal CPI, data, control stalls	4.4

FIGURE 3.1 The major techniques examined in Appendix A, chapter 3, or chapter 4 are shown together with the component of the CPI equation that the technique affects.

line code sequence with no branches in except to the entry and no branches out except at the exit–is quite small. For typical MIPS programs the average dynamic branch frequency often between 15% and 25%, meaning that between four and seven instructions execute between a pair of branches. Since these instructions are likely to depend upon one another, the amount of overlap we can exploit within a basic block is likely to be much less than the average basic block size. To obtain substantial performance enhancements, we must exploit ILP across multiple basic blocks.

The simplest and most common way to increase the amount of parallelism available among instructions is to exploit parallelism among iterations of a loop. This type of parallelism is often called *loop-level parallelism*. Here is a simple example of a loop, which adds two 1000-element arrays, that is completely parallel:

```
for (i=1; i<=1000; i=i+1)
        x[i] = x[i] + y[i];
```

Every iteration of the loop can overlap with any other iteration, although within each loop iteration there is little or no opportunity for overlap.

There are a number of techniques we will examine for converting such loop-level parallelism into instruction-level parallelism. Basically, such techniques work by unrolling the loop either statically by the compiler (an approach we explore in the next chapter) or dynamically by the hardware (the subject of this chapter).

An important alternative method for exploiting loop-level parallelism is the use of vector instructions (see Appendix B). Essentially, a vector instruction operates on a sequence of data items. For example, the above code sequence could execute in four instructions on some vector processors: two instructions to load the vectors x and y from memory, one instruction to add the two vectors, and an instruction to store back the result vector. Of course, these instructions would be pipelined and have relatively long latencies, but these latencies may be overlapped. Vector instructions and the operation of vector processors are described in detail in the online Appendix B. Although the development of the vector ideas preceded many of the techniques we examine in these two chapters for exploiting ILP, processors that exploit ILP have almost completely replaced vector-based processors. Vector instruction sets, however, may see a renaissance, at least for use in graphics, digital signal processing, and multimedia applications.

Data Dependence and Hazards

Determining how one instruction depends on another is critical to determining how much parallelism exists in a program and how that parallelism can be exploited. In particular, to exploit instruction-level parallelism we must determine which instructions can be executed in parallel. If two instructions are *parallel*, they can execute simultaneously in a pipeline without causing any stalls, assuming the pipeline has sufficient resources (and hence no structural hazards exist). If two instructions are dependent they are not parallel and must be executed in order, though they may often be partially overlapped. The key in both cases is to determine whether an instruction is dependent on another instruction.

Data Dependences

There are three different types of dependences: data dependences (also called true data dependences), name dependences, and control dependences. An instruction *j* is *data dependent* on instruction *i* if either of the following holds:

- Instruction *i* produces a result that may be used by instruction *j*, or
- Instruction *j* is data dependent on instruction *k*, and instruction *k* is data dependent on instruction *i*.

The second condition simply states that one instruction is dependent on another if there exists a chain of dependences of the first type between the two instructions. This dependence chain can be as long as the entire program.

For example, consider the following code sequence that increments a vector of values in memory (starting at 0(R1) and with the last element at 8(R2)) by a scalar in register F2:

```
Loop:  L.D    F0,0(R1);F0=array element
       ADD.D  F4,F0,F2;add scalar in F2
       S.D    F4,0(R1);store result
       DADDUI R1,R1,#-8;decrement pointer 8 bytes (/e
       BNE    R1,R2,LOOP; branch R1!=zero
```

The data dependences in this code sequence involve both floating point data:

```
Loop:  L.D    F0,0(R1);F0=array element
       ADD.D  F4,F0,F2;add scalar in F2
       S.D    F4,0(R1);store result
```

and integer data:

```
       DADDIU R1,R1,-8;decrement pointer
                     ;8 bytes (per DW)
       BNE    R1,R2,Loop; branch R1!=zero
```

Both of the above dependent sequences, as shown by the arrows, with each instruction depending on the previous one. The arrows here and in following examples show the order that must be preserved for correct execution. The arrow points from an instruction that must precede the instruction that the arrowhead points to.

If two instructions are data dependent they cannot execute simultaneously or be completely overlapped. The dependence implies that there would be a chain of one or more data hazards between the two instructions. Executing the instructions simultaneously will cause a processor with pipeline interlocks to detect a hazard and stall, thereby reducing or eliminating the overlap. In a processor without interlocks that relies on compiler scheduling, the compiler cannot schedule dependent instructions in such a way that they completely overlap, since the program will not execute correctly. The presence of a data dependence in an instruction sequence reflects a data dependence in the source code from which the instruction sequence was generated. The effect of the original data dependence must be preserved.

Dependences are a property of *programs*. Whether a given dependence results in an actual hazard being detected and whether that hazard actually causes a stall are properties of the *pipeline organization*. This difference is critical to understanding how instruction-level parallelism can be exploited.

In our example, there is a data dependence between the DADDIU and the BNE; this dependence causes a stall because we moved the branch test for the MIPS pipeline to the ID stage. Had the branch test stayed in EX, this dependence would not cause a stall. Of course, the branch delay would then still be 2 cycles, rather than 1.

The presence of the dependence indicates the potential for a hazard, but the actual hazard and the *length of any stall* is a property of the pipeline. The importance of the data dependences is that a dependence (1) indicates the possibility of a hazard, (2) determines the order in which results must be calculated, and (3) sets an upper bound on how much parallelism can possibly be exploited. Such limits are explored in section 3.8.

Since a data dependence can limit the amount of instruction-level parallelism we can exploit, a major focus of this chapter and the next is overcoming these limitations. A dependence can be overcome in two different ways: maintaining the dependence but avoiding a hazard, and eliminating a dependence by transforming the code. Scheduling the code is the primary method used to avoid a hazard without altering a dependence. In this chapter, we consider hardware schemes for scheduling code dynamically as it is executed. As we will see, some types of dependences can be eliminated, primarily by software, and in some cases by hardware techniques.

A data value may flow between instructions either through registers or through memory locations. When the data flow occurs in a register, detecting the dependence is reasonably straightforward since the register names are fixed in the instructions, although it gets more complicated when branches intervene and correctness concerns cause a compiler or hardware to be conservative.

Dependences that flow through memory locations are more difficult to detect since two addresses may refer to the same location, but look different: For example, `100(R4)` and `20(R6)` may be identical. In addition, the effective address of a load or store may change from one execution of the instruction to another (so that `20(R4)` and `20(R4)` will be different), further complicating the detection of a dependence. In this chapter, we examine hardware for detecting data dependences that involve memory locations, but we shall see that these techniques also have limitations. The compiler techniques for detecting such dependences are critical in uncovering loop-level parallelism, as we shall see in the next chapter.

Name Dependences

The second type of dependence is a *name dependence.* A name dependence occurs when two instructions use the same register or memory location, called a *name*, but there is no flow of data between the instructions associated with that name. There are two types of name dependences between an instruction *i* that *precedes* instruction *j* in program order:

1. An *antidependence* between instruction *i* and instruction *j* occurs when instruction *j* writes a register or memory location that instruction *i* reads. The original ordering must be preserved to ensure that *i* reads the correct value.
2. An *output dependence* occurs when instruction *i* and instruction *j* write the same register or memory location. The ordering between the instructions must be preserved to ensure that the value finally written corresponds to instruction *j*.

Both antidependences and output dependences are name dependences, as opposed to true data dependences, since there is no value being transmitted between the instructions. Since a name dependence is not a true dependence, instructions involved in a name dependence can execute simultaneously or be reordered, if the name (register number or memory location) used in the instructions is changed so the instructions do not conflict. This renaming can be more easily done for register operands, where it is called *register renaming*. Register renaming can be done either statically by a compiler or dynamically by the hardware. Before describing dependences arising from branches, let's examine the relationship between dependences and pipeline data hazards.

Data Hazards

A hazard is created whenever there is a dependence between instructions, and they are close enough that the overlap caused by pipelining, or other reordering of instructions, would change the order of access to the operand involved in the dependence. Because of the dependence, we must preserve what is called *program order*, that is the order that the instructions would execute in, if executed sequentially one at a time as determined by the original source program. The goal of both our software and hardware techniques is to exploit parallelism by preserving program order *only where it affects the outcome of the program*. Detecting and avoiding hazards ensures that necessary program order is preserved.

Data hazards may be classified as one of three types, depending on the order of read and write accesses in the instructions. By convention, the hazards are named by the ordering in the program that must be preserved by the pipeline. Consider two instructions *i* and *j*, with *i* occurring before *j* in program order. The possible data hazards are

- RAW *(read after write) — j* tries to read a source before *i* writes it, so *j* incorrectly gets the *old* value. This hazard is the most common type and corresponds to a true data dependence. Program order must be preserved to ensure that *j* receives the value from *i*. In the simple common five-stage static pipeline (see Appendix A) a load instruction followed by an integer ALU instruction that directly uses the load result will lead to a RAW hazard.

- WAW *(write after write) — j* tries to write an operand before it is written by *i*. The writes end up being performed in the wrong order, leaving the value written by *i* rather than the value written by *j* in the destination. This hazard corresponds to an output dependence. WAW hazards are present only in pipelines that write in more than one pipe stage or allow an instruction to proceed even when a previous instruction is stalled. The classic five-stage integer pipeline used in Appendix A writes a register only in the WB stage and avoids this class of hazards, but this chapter explores pipelines that allow instructions to be reordered, creating the possibility of WAW hazards. WAW hazards can also between a short integer pipeline and a longer floating-point pipeline (see the pipelines in Sections A.5 and A.6 of Appendix A). For example, a floating point multiply instruction that writes F4, shortly followed by a load of F4 could yield a WAW hazard, since the load could complete before the multiply completed.

- WAR *(write after read) — j* tries to write a destination before it is read by *i*, so *i* incorrectly gets the *new* value. This hazard arises from an antidependence. WAR hazards cannot occur in most static issue pipelines even deeper pipelines or floating point pipelines because all reads are early (in ID) and all writes are late (in WB). (See Appendix A to convince yourself.) A WAR hazard occurs either when there are some instructions that write results early in the instruction pipeline, *and* other instructions that read a source late in the pipeline or when instructions are reordered, as we will see in this chapter.

Note that the RAR *(read after read)* case is not a hazard.

Control Dependences

The last type of dependence is a *control dependence*. A control dependence determines the ordering of an instruction, *i*, with respect to a branch instruction so that the instruction *i* is executed in correct program order and only when it should be. Every instruction, except for those in the first basic block of the program, is control dependent on some set of branches, and, in general, these control dependences must be preserved to preserve program order. One of the simplest examples of a control dependence is the dependence of the statements in the "then" part of an if statement on the branch. For example, in the code segment:

```
if p1 {
        S1;
};
if p2 {
        S2;
}
```

`S1` is control dependent on `p1`, and `S2` is control dependent on `p2` but not on `p1`.

In general, there are two constraints imposed by control dependences:

1. An instruction that is control dependent on a branch cannot be moved *before* the branch so that its execution *is no longer controlled* by the branch. For example, we cannot take an instruction from the then-portion of an if-statement and move it before the if-statement.
2. An instruction that is not control dependent on a branch cannot be moved *after* the branch so that its execution *is controlled* by the branch. For example, we cannot take a statement before the if-statement and move it into the then-portion.

Control dependence is preserved by two properties in a simple pipeline, such as that in Chapter 1. First, instructions execute in program order. This ordering ensures that an instruction that occurs before a branch is executed before the branch. Second, the detection of control or branch hazards ensures that an in-

struction that is control dependent on a branch is not executed until the branch direction is known.

Although preserving control dependence is a useful and simple way to help preserve program order, the control dependence in itself is not the fundamental performance limit. We may be willing to execute instructions that should not have been executed, thereby violating the control dependences, *if* we can do so without affecting the correctness of the program. Control dependence is not the critical property that must be preserved. Instead, the two properties critical to program correctness–and normally preserved by maintaining both data and control dependence–are the *exception behavior* and the *data flow.*

Preserving the *exception behavior* means that any changes in the ordering of instruction execution must not change how exceptions are raised in the program. Often this is relaxed to mean that the reordering of instruction execution must not cause any new exceptions in the program. A simple example shows how maintaining the control and data dependences can prevent such situations. Consider this code sequence:

```
        DADDU   R2,R3,R4
        BEQZ    R2,L1
        LW      R1,0(R2)
L1:
```

In this case, it is easy to see that if we do not maintain the data dependence involving R2, we can change the result of the program. Less obvious is the fact that if we ignore the control dependence and move the load instruction before the branch, the load instruction may cause a memory protection exception. Notice that *no data dependence* prevents us from interchanging the `BEQZ` and the `LW`; it is only the control dependence. To allow us to reorder these instructions (and still preserve the data dependence), we would like to just ignore the exception when the branch is taken. In section 3.5, we will look at a hardware technique, speculation, which allows us to overcome this exception problem. The next chapter looks at other techniques for the same problem.

The second property preserved by maintenance of data dependences and control dependences is the *data flow.* The data flow is the actual flow of data values among instructions that produce results and those that consume them. Branches make the data flow dynamic, since they allow the source of data for a given instruction to come from many points. Put another way, it is not sufficient to just maintain data dependences because an instruction may be data dependent on more than one predecessor. Program order is what determines which predecessor will actually deliver a data value to an instruction. Program order is ensured by maintaining the control dependences.

For example, consider the following code fragment:

```
        DADDU   R1,R2,R3
        BEQZ    R4,L
        DSUBU   R1,R5,R6
L:      ...
```

```
                OR      R7,R1,R8
```

In this example, the value of R1 used by the OR instruction depends on whether the branch is taken or not. Data dependence alone is not sufficient to preserve correctness. The OR instruction is data dependent on both the DAAU and DSUBU instructions, but preserving this order alone is insufficient for correct execution. Instead, when the instructions execute, the data flow must be preserved: If the branch is not taken then the value of R1 computed by the DSUBU should be used by the OR, and if the branch is taken the value of R1 computed by the DADDU should be used by the OR. By preserving the control dependence of the OR on the branch, we prevent an illegal change to the data flow. For similar reasons, the DSUBU instruction cannot be moved above the branch. Speculation, which helps with the exception problem, will also allow us to lessen the impact of the control dependence while still maintaining the data flow, as we will see in section 3.5.

Sometimes we can determine that violating the control dependence cannot affect either the exception behavior or the data flow. Consider the following code sequence:

```
                DADDU   R1,R2,R3
                BEQZ    R12,skipnext
                DSUBU   R4,R5,R6
                DADDU   R5,R4,R9
        skipnext:       OR      R7,R8,R9
```

Suppose we knew that the register destination of the DSUBU instruction (R4) was unused after the instruction labeled skipnext. (The property of whether a value will be used by an upcoming instruction is called *liveness*.) If R4 were unused, then changing the value of R4 just before the branch would not affect the data flow since R4 would be *dead* (rather than live) in the code region after skipnext. Thus, if R4 were dead and the existing DSUBU instruction could not generate an exception (other than those from which the processor resumes the same process), we could move the DSUBU instruction before the branch, since the data flow cannot be affected by this change. If the branch is taken, the DSUBU instruction will execute and will be useless, but it will not affect the program results. This type of code scheduling is sometimes called *speculation*, since the compiler is betting on the branch outcome; in this case, the bet is that the branch is usually not taken. More ambitious compiler speculation mechanisms are discussed in Chapter 4.

Control dependence is preserved by implementing control hazard detection that causes control stalls. Control stalls can be eliminated or reduced by a variety of hardware and software techniques. Delayed branches, which we saw in Chapter 1, can reduce the stalls arising from control hazards; scheduling a delayed branch requires that the compiler preserve the data flow.

The key focus of the rest of this chapter is on techniques that exploit instruction-level parallelism using hardware. The data dependences in a compiled program act as a limit on how much ILP can be exploited. The challenge is to approach that limit by trying to minimize the actual hazards and associated stalls that arise. The techniques we examine become ever more sophisticated in an attempt to ex-

ploit all the available parallelism while maintaining the necessary true data dependences in the code.

3.2 Overcoming Data Hazards with Dynamic Scheduling

A simple satirically scheduled pipeline fetches an instruction and issues it, unless there was a data dependence between an instruction already in the pipeline and the fetched instruction that cannot be hidden with bypassing or forwarding. (Forwarding logic reduces the effective pipeline latency so that the certain dependences do not result in hazards). If there is a data dependence that cannot be hidden, then the hazard detection hardware stalls the pipeline (starting with the instruction that uses the result). No new instructions are fetched or issued until the dependence is cleared.

In this section, we explore an important technique, called *dynamic scheduling*, in which the hardware rearranges the instruction execution to reduce the stalls while maintaining data flow and exception behavior. Dynamic scheduling offers several advantages: It enables handling some cases when dependences are unknown at compile time (e.g., because they may involve a memory reference), and it simplifies the compiler. Perhaps most importantly, it also allows code that was compiled with one pipeline in mind to run efficiently on a different pipeline. In section 3.5, we will explore hardware speculation, a technique with significant performance advantages, which builds on dynamic scheduling. As we will see, the advantages of dynamic scheduling are gained at a cost of a significant increase in hardware complexity.

Although a dynamically scheduled processor cannot change the data flow, it tries to avoid stalling when dependences, which could generate hazards, are present. In contrast, static pipeline scheduling by the compiler (covered in the next chapter) tries to minimize stalls by separating dependent instructions so that they will not lead to hazards. Of course, compiler pipeline scheduling can also be used on code destined to run on a processor with a dynamically scheduled pipeline.

Dynamic Scheduling: The Idea

A major limitation of the simple pipelining techniques we discuss in Appendix A is that they all use in-order instruction issue and execution: Instructions are issued in program order and if an instruction is stalled in the pipeline, no later instructions can proceed. Thus, if there is a dependence between two closely spaced instructions in the pipeline, this will lead to a hazard and a stall will result. If there are multiple functional units, these units could lie idle. If instruction *j* depends on a long-running instruction *i*, currently in execution in the pipeline, then all instruc-

Edited by Foxit Reader
Copyright(C) by Foxit Software Company,2005-2008
For Evaluation Only.

tions after *j* must be stalled until *i* is finished and *j* can execute. For example, consider this code:

```
DIV.D   F0,F2,F4
ADD.D   F10,F0,F8
SUB.D   F12,F8,F14
```

The `SUB.D` instruction cannot execute because the dependence of `ADD.D` on `DIV.D` causes the pipeline to stall; yet `SUB.D` is not data dependent on anything in the pipeline. This hazard creates a performance limitation that can be eliminated by not requiring instructions to execute in program order.

In the classic five-stage pipeline developed in the first chapter, both structural and data hazards could be checked during instruction decode (ID): When an instruction could execute without hazards, it was issued from ID knowing that all data hazards had been resolved. To allow us to begin executing the `SUB.D` in the above example, we must separate the issue process into two parts: checking for any structural hazards and waiting for the absence of a data hazard. We can still check for structural hazards when we issue the instruction; thus, we still use in-order instruction issue (i.e., instructions issue in program order), but we want an instruction to begin execution as soon as its data operand is available. Thus, this pipeline does *out-of-order execution*, which implies *out-of-order completion*.

Out-of-order execution introduces the possibility of WAR and WAW hazards, which do not exist in the five-stage integer pipeline and its logical extension to an in-order floating-point pipeline. Consider the following MIPS floating-point code sequence:

```
DIV.D   F0,F2,F4
ADD.D   F6,F0,F8
SUB.D   F8,F10,F14
MULT.D  F6,F10,F8
```

There is an antidependence between the `ADD.D` and the `SUB.D`, and if the pipeline executes the `SUB.D` before the `ADD.D` (which is waiting for the `DIV.D`), it will violate the antidependence, yielding a WAR hazard. Likewise, to avoid violating output dependences, such as the write of `F6` by `MULT.D`, WAW hazards must be handled. As we will see, both these hazards are avoided by the use of register renaming.

Out-of-order completion also creates major complications in handling exceptions. Dynamic scheduling with out-of-order completion must preserve exception behavior in the sense that *exactly* those exceptions that would arise if the program were executed in strict program order *actually* do arise. Dynamically scheduled processors preserve exception behavior by ensuring that no instruction can generate an exception until the processor knows that the instruction raising the exception will be executed; we will see shortly how this property can be guaranteed. Although exception behavior must be preserved, dynamically scheduled processors may generate *imprecise* exceptions. An exception is *imprecise* if the processor state when an exception is raised does not look exactly as if the instructions

were executed sequentially in strict program order. Imprecise exceptions can occur because of two possibilities:

1. the pipeline may have already *completed* instructions that are *later* in program order than the instruction causing the exception, and
2. the pipeline may have *not yet completed* some instructions that are *earlier* in program order than the instruction causing the exception.

Imprecise exceptions make it difficult to restart execution after an exception. Rather than address these problems in this section, we will discuss a solution that provides precise exceptions in the context of a processor with speculation in section 3.5. For floating-point exceptions, other solutions have been used, as discussed in Appendix A.

To allow out-of-order execution, we essentially split the ID pipe stage of our simple five-stage pipeline into two stages:

1. *Issue*—Decode instructions, check for structural hazards.
2. *Read operands*—Wait until no data hazards, then read operands.

An instruction fetch stage precedes the issue stage and may fetch either into an instruction register or into a queue of pending instructions; instructions are then issued from the register or queue. The EX stage follows the read operands stage, just as in the five-stage pipeline. Execution may take multiple cycles, depending on the operation.

We will distinguish when an instruction *begins execution* and when it *completes execution*; between the two times, the instruction is *in execution.* Our pipeline allows multiple instructions to be in execution at the same time, and without this capability, a major advantage of dynamic scheduling is lost. Having multiple instructions in execution at once requires multiple functional units, pipelined functional units, or both. Since these two capabilities—pipelined functional units and multiple functional units—are essentially equivalent for the purposes of pipeline control, we will assume the processor has multiple functional units.

In a dynamically scheduled pipeline, all instructions pass through the issue stage in order (in-order issue); however, they can be stalled or bypass each other in the second stage (read operands) and thus enter execution out of order. *Scoreboarding* is a technique for allowing instructions to execute out-of-order when there are sufficient resources and no data dependences; it is named after the CDC 6600 scoreboard, which developed this capability. We focus on a more sophisticated technique, called *Tomasulo's algorithm*, that has several major enhancements over scoreboarding. The reader wishing a gentler introduction to these

concepts may want to consult the online version of Appendix G that thoroughly discusses scoreboarding and includes several examples.

Dynamic Scheduling Using Tomasulo's Approach

A key approach to allow execution to proceed in the presence of dependences was used by the IBM 360/91 floating-point unit. Invented by Robert Tomasulo, this scheme tracks when operands for instructions are available, to minimize RAW hazards, and introduces register renaming, to minimize WAW and RAW hazards. There are many variations on this scheme in modern processors, though the key concept of tracking instruction dependencies to allow execution as soon as operands are available and renaming registers to avoid WAR and WAW hazards are common characteristics.

The IBM 360/91 was completed just before caches appeared in commercial processors. IBM's goal was to achieve high floating-point performance from an instruction set and from compilers designed for the entire 360-computer family, rather than from specialized compilers for the high-end processors. The 360 architecture had only four double-precision floating-point registers, which limits the effectiveness of compiler scheduling; this fact was another motivation for the Tomasulo approach. In addition, the IBM 360/91 had long memory accesses and long floating-point delays, which Tomasulo's algorithm was designed to overcome. At the end of the section, we will see that Tomasulo's algorithm can also support the overlapped execution of multiple iterations of a loop.

We explain the algorithm, which focuses on the floating-point unit and load/store unit, in the context of the MIPS instruction set. The primary difference between MIPS and the 360 is the presence of register-memory instructions in the latter processor. Because Tomasulo's algorithm uses a load functional unit, no significant changes are needed to add register-memory addressing modes. The IBM 360/91 also had pipelined functional units, rather than multiple functional units, but we describe the algorithm as if there were multiple functional units. It is a simple conceptual extension to also pipeline those functional units.

As we will see RAW hazards are avoided by executing an instruction only when its operands are available. WAR and WAW hazards, which arise from name dependences, are eliminated by register renaming. *Register renaming* eliminates these hazards by renaming all destination registers, including those with a pending read or write for an earlier instruction, so that the out-of-order write does not affect any instructions that depend on an earlier value of an operand.

To better understand how register renaming eliminates WAR and WAW hazards consider the following example code sequence that includes both a potential WAR and WAW hazard:

```
DIV.D   F0,F2,F4
ADD.D   F6,F0,F8
S.D     F6,0(R1)
SUB.D   F8,F10,F14
MULT.D F6,F10,F8
```

There is an antidependence between the ADD.D and the SUB.D and an output dependence between the ADD.D and the MULT.D leading to three possible hazards: a WAR hazard on the use of F8 by ADD.D and on the use of F8 by the MULT.D, and a WAW hazard since the ADD.D may finish later than the MULT.D. There are also three true data dependences between the DIV.D and the ADD.D, between the SUB.D and the MULT.D, and between the ADD.D and the S.D.

These name dependences can both be eliminated by register renaming. For simplicity, assume the existence of two temporary registers, S and T. Using S and T, the sequence can be rewritten without any dependences as:

```
DIV.D   F0,F2,F4
ADD.D   S,F0,F8
S.D     S,0(R1)
SUB.D   T,F10,F14
MULT.D  F6,F10,T
```

In addition, any subsequent uses of F8 must be replaced by the register T. In this code segment, the renaming process can be done statically by the compiler. Finding any uses of F8 that are later in the code requires either sophisticated compiler analysis or hardware support, since there may be intervening branches between the above code segment and a later use of F8. As we will see Tomasulo's algorithm can handle renaming across branches.

In Tomasulo's scheme, register renaming is provided by the *reservation stations*, which buffer the operands of instructions waiting to issue, and by the issue logic. The basic idea is that a reservation station fetches and buffers an operand as soon as it is available, eliminating the need to get the operand from a register. In addition, pending instructions designate the reservation station that will provide their input. Finally, when successive writes to a register overlap in execution, only the last one is actually used to update the register. As instructions are issued, the register specifiers for pending operands are renamed to the names of the reservation station, which provides register renaming. Since there can be more reservation stations than real registers, the technique can even eliminate hazards arising from name dependences that could not be eliminated by a compiler. As we explore the components of Tomasulo's scheme, we will return to the topic of register renaming and see exactly how the renaming occurs and how it eliminates WAR and WAW hazards.

The use of reservation stations, rather than a centralized register file, leads to two other important properties. First, hazard detection and execution control are distributed: The information held in the reservation stations at each functional unit determine when an instruction can begin execution at that unit. Second, results are passed directly to functional units from the reservation stations where they are buffered, rather than going through the registers. This bypassing is done with a common result bus that allows all units waiting for an operand to be loaded simultaneously (on the 360/91 this is called the *common data bus*, or CDB). In pipelines with multiple execution units and issuing multiple instructions per clock, more than one result bus will be needed.

Figure 3.2 shows the basic structure of a Tomasulo-based MIPS processor, including both the floating-point unit and the load/store unit; none of the execution control tables are shown. Each reservation station holds an instruction that has been issued and is awaiting execution at a functional unit, and either the operand values for that instruction, if they have already been computed, or else the names of the functional units that will be provide the operand values.

The load buffers and store buffers hold data or addresses coming from and going to memory and behave almost exactly like reservation stations, so we distinguish them only when necessary. The floating-point registers are connected by a pair of buses to the functional units and by a single bus to the store buffers. All results from the functional units and from memory are sent on the common data bus, which goes everywhere except to the load buffer. All reservation stations have tag fields, employed by the pipeline control.

Before we describe the details of the reservation stations and the algorithm, let's look at the steps an instruction goes through, just as we did for the five-stage pipeline of Chapter 1. Since the structure is dramatically different, there are only three steps (though each one can now take an arbitrary number of clock cycles):

1. *Issue*—Get the next instruction from the head of the instruction queue, which is maintained in FIFO order to ensure the maintenance of correct data flow. If there is a matching reservation station that is empty, issue the instruction to the station with the operand values, if they are currently in the registers. If there is not an empty reservation station, then there is a structural hazard and the instruction stalls until a station or buffer is freed. If the operands are not in the registers, enter the functional units that will produce the operands into the Qi and Qj fields. This step renames registers, eliminating WAR and WAW hazards.

2. *Execute*—If one or more of the operands is not yet available, monitor the common data bus (CDB) while waiting for it to be computed. When an operand becomes available, it is placed into the corresponding reservation station. When all the operands are available, the operation can be executed at the corresponding functional unit. By delaying instruction execution until the operands are available RAW, hazards are avoided. Notice that several instructions could become ready in the same clock cycle for the same functional unit. Although independent functional units could begin execution in the same clock cycle for different instructions, if more than one instruction is ready for a single functional unit, the unit will have to choose among them. For the floating point reservation stations, this choice may be made arbitrarily; loads and stores, however, present an additional complication.

   Loads and stores require a two-step execution process. The first step computes the effective address when the base register is available, and the effective address is then placed in the load or store buffer. Loads in the load buffer execute as soon as the memory unit is available. Stores in the store buffer wait from the value to be stored before being sent to the memory unit. Loads

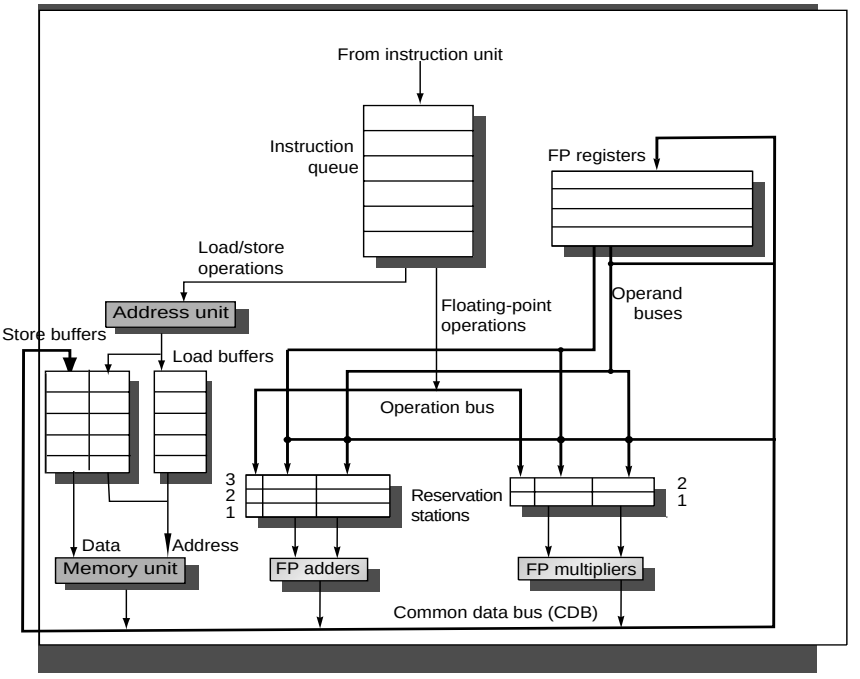


FIGURE 3.2 The basic structure of a MIPS floating point unit using Tomasulo's algorithm. Instructions are sent from the instruction unit into the instruction queue from which they are issued in FIFO order. The reservation stations include the operation and the actual operands, as well as information used for detecting and resolving hazards. Load buffers have three functions: hold the components of the effective address until it is computed, track outstanding loads that are waiting on the memory, and hold the results of completed loads that are waiting for the CDB. Similarly, store buffers have three functions: hold the components of the effective address until it is computed, hold the destination memory addresses of outstanding stores that are waiting for the data value to store, and hold the address and value to store until the memory unit is available. All results from either the FP units or the load unit are put on the CDB, which goes to the FP register file as well as to the reservation stations and store buffers. The FP adders implement addition and subtraction, and the FP multipliers do multiplication and division.

and stores are maintained in program order through the effective address calculation, which will help to prevent hazards through memory, as we will see shortly.

To preserve exception behavior, no instruction is allowed to initiate execution until all branches that precede the instruction in program order have completed. This restriction guarantees that an instruction that causes an exception during execution really would have been executed. In a processor using branch prediction (as all dynamically schedule processors do), this means that the processor must know that the branch prediction was correct before allowing an instruction after the branch to begin execution. It is possible by recording

the occurrence of the exception, but not actually raising it, to allow execution of the instruction to start and not stall the instruction until it enters write result. As we will see, speculation provides a more flexible and more complete method to handle exceptions, so we will delay making this enhancement and show how speculation handles this problem later.

3. *Write result*—When the result is available, write it on the CDB and from there into the registers and into any reservation stations (including store buffers) waiting for this result. Stores also write data to memory during this step: When both the address and data value are available, they are sent to the memory unit and the store completes.

The data structures used to detect and eliminate hazards are attached to the reservation stations, to the register file, and to the load and store buffers with slightly different information attached to different objects. These tags are essentially names for an extended set of virtual registers used in renaming. In our example, the tag field is a four-bit quantity that denotes one of the five reservation stations or one of the six load buffers. As we will see, this produces the equivalent of eleven registers that can be designated as result registers (as opposed to the four double-precision registers that the 360 architecture contains). In a processor with more real registers, we would want renaming to provide an even larger set of virtual registers. The tag field describes which reservation station contains the instruction that will produce a result needed as a source operand.

Once an instruction has issued and is waiting for a source operand, it refers to the operand by the reservation station number where the instruction that will write the register has been assigned. Unused values, such as zero, indicate that the operand is already available in the registers. Because there are more reservation stations than actual register numbers, WAW and WAR hazards are eliminated by renaming results using reservation station numbers. Although in Tomasulo's scheme the reservation stations are used as the extended virtual registers, other approaches could use a register set with additional registers or a structure like the reorder buffer, which we will see in section 3.5.

In describing the operation of this scheme, we use a terminology taken from the CDC scoreboard scheme, showing the terminology used by the IBM 360/91 for historical reference. It is important to remember that the tags in the Tomasulo scheme refer to the buffer or unit that will produce a result; the register names are discarded when an instruction issues to a reservation station.

Each reservation station has six fields:

Op—The operation to perform on source operands S1 and S2.

Qj, Qk—The reservation stations that will produce the corresponding source operand; a value of zero indicates that the source operand is already available in Vj or Vk, or is unnecessary. (The IBM 360/91 calls these SINKunit and SOURCEunit.)

Vj, Vk—The value of the source operands. Note that only one of the V field or

the Q field is valid for each operand. For loads, the Vk field is used to the offset from the instruction.(These fields are called SINK and SOURCE on the IBM 360/91.)

A–used to hold information for the memory address calculation for a load or store. Initially, the immediate field of the instruction is stored here; after the address calculation, the effective address is stored here.

Busy—Indicates that this reservation station and its accompanying functional unit are occupied.

The register file has a field, Qi:

Qi—The number of the reservation station that contains the operation whose result should be stored into this register. If the value of Qi is blank (or 0), no currently active instruction is computing a result destined for this register, meaning that the value is simply the register contents.

The load and store buffers each have a field, A, which holds the result of the effective address once the first step of execution has been completed.

In the next section, we will first consider some examples that show how these mechanisms work and then examine the detailed algorithm.

3.3 Dynamic Scheduling: Examples and the Algorithm

Before we examine Tomasulo's algorithm in detail, let's consider as few examples, which will help illustrate how the algorithm works.

EXAMPLE Show what the information tables look like for the following code sequence when only the first load has completed and written its result:

```
1.      L.D     F6,34(R2)
2.      L.D     F2,45(R3)
3.      MUL.D   F0,F2,F4
4.      SUB.D   F8,F2,F6
5.      DIV.D   F10,F0,F6
6.      ADD.D   F6,F8,F2
```

ANSWER The result is shown in the three tables in Figure 3.3. The numbers appended to the names add, mult, and load stand for the tag for that reservation station—Add1 is the tag for the result from the first add unit. In addition we have included an instruction status table. This table is included only to help you understand the algorithm; it is *not* actually a part of the hardware. Instead, the reservation station keeps the state of each operation that has issued .

Instruction		Instruction status		
		Issue	Execute	Write result
L.D	F6,34(R2)	√	√	√
L.D	F2,45(R3)	√	√	
MUL.D	F0,F2,F4	√		
SUB.D	F8,F2,F6	√		
DIV.D	F10,F0,F6	√		
ADD.D	F6,F8,F2	√		

	Reservation stations						
Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	no						
Load2	yes	Load					45+Regs[R3]
Add1	yes	SUB		Mem[34+Regs[R2]]	Load2		
Add2	yes	ADD			Add1	Load2	
Add3	no						
Mult1	yes	MUL		Regs[F4]	Load2		
Mult2	yes	DIV		Mem[34+Regs[R2]]	Mult1		

	Register status								
Field	F0	F2	F4	F6	F8	F10	F12	...	F30
Qi	Mult1	Load2		Add2	Add1	Mult2			

FIGURE 3.3 Reservation stations and register tags shown when all of the instructions have issued, but only the first load instruction has completed and written its result to the CDB. The second load has completed effective address calculation, but is waiting on the memory unit. We use the array Regs[ ] to refer to the register file and the array Mem[ ] to refer to the memory. Remember that an operand is specified by either a Q field or a V field at any time. Notice that the ADD.D instruction, which has a WAR hazard at the WB stage, has issued and could complete before the DIV.D initiates.

Tomasulo's scheme offers two major advantages over earlier and simpler schemes: (1) the distribution of the hazard detection logic and (2) the elimination of stalls for WAW and WAR hazards.

The first advantage arises from the distributed reservation stations and the use of the CDB. If multiple instructions are waiting on a single result, and each instruction already has its other operand, then the instructions can be released simultaneously by the broadcast on the CDB. If a centralized register file were used, the units would have to read their results from the registers when register buses are available.

The second advantage, the elimination of WAW and WAR hazards, is accomplished by renaming registers using the reservation stations, and by the process of storing operands into the reservation station as soon as they are available. For example, in our code sequence in Figure 3.3 we have issued both the DIV.D and the ADD.D, even though there is a WAR hazard involving F6. The hazard is eliminated in one of two ways. First, if the instruction providing the value for the DIV.D has completed, then Vk will store the result, allowing DIV.D to execute independent of the ADD.D (this is the case shown).

On the other hand, if the L.D had not completed, then Qk would point to the Load1 reservation station, and the DIV.D instruction would be independent of the ADD.D. Thus, in either case, the ADD.D can issue and begin executing. Any uses of the result of the DIV.D would point to the reservation station, allowing the ADD.D to complete and store its value into the registers without affecting the DIV.D. We'll see an example of the elimination of a WAW hazard shortly. But let's first look at how our earlier example continues execution. In this example, and the ones that follow in this chapter, assume the following latencies: Load is 1 cycle, Add is 2 clock cycles, multiply is 10 clock cycles, and divide is 40 clock cycles.

EXAMPLE Using the same code segment as the previous example (page 239), show what the status tables look like when the MUL.D is ready to write its result.

ANSWER The result is shown in the three tables in Figure 3.4. Notice that ADD.D has completed since the operands of DIV.D were copied, thereby overcoming the WAR hazard. Notice that even if the load of F6 was delayed, the add into F6 could be executed without triggering a WAW hazard.

Instruction		Instruction status		
		Issue	**Execute**	**Write result**
L.D	F6,34(R2)	√	√	√
L.D	F2,45(R3)	√	√	√
MUL.D	F0,F2,F4	√	√	
SUB.D	F8,F2,F6	√	√	√
DIV.D	F10,F0,F6	√		
ADD.D	F6,F8,F2	√	√	√

	Reservation stations						
Name	**Busy**	**Op**	**Vj**	**Vk**	**Qj**	**Qk**	**A**
Load1	no						
Load2	no						
Add1	no						
Add2	no						
Add3	no						
Mult1	yes	MUL	Mem[45+Regs[R3]]	Regs[F4]			
Mult2	yes	DIV		Mem[34+Regs[R2]]	Mult1		

	Register status								
Field	**F0**	**F2**	**F4**	**F6**	**F8**	**F10**	**F12**	**...**	**F30**
Qi	Mult1					Mult2			

FIGURE 3.4 Multiply and divide are the only instructions not finished.

n

Tomasulo's Algorithm: the details

Figure 3.5 gives the checks and steps that each instruction must go through. As mentioned earlier, loads and stores go through a functional unit for effective address computation before proceeding to independent load or store buffers. Loads take a second execution step to access memory and then go to Write Result to send the value from memory to the register file and/or any waiting reservation stations. Stores complete their execution in the Write Result stage, which writes the result to memory. Notice that all writes occur in Write Result, whether the destination is a register or memory. This restriction simplifies Tomasulo's algorithm and is critical to its extension with speculation in section 3.5.

Instruction state	Wait until	Action or bookkeeping
Issue FP Operation	Station r empty	`if (Register Stat[rs].Qi ≠0) {RS[r].Qj← RegisterStat[rs].Qi} else {RS[r].Vj← Regs[rs]; RS[r].Qj← 0}; if (RegisterStat[rt].Qi≠0) {RS[r].Qk← RegisterStat[rt]Q.i} else {RS[r].Vk← Regs[rt]; RS[r].Qk← 0}; RS[r].Busy← yes; RegisterStat[rd].Qi=r;`
Load or Store	Buffer r empty	`if (Register Stat[rs].Qi ≠0) {RS[r].Qj← RegisterStat[rs].Qi} else {RS[r].Vj← Regs[rs]; RS[r].Qj← 0}; RS[r].A← imm; RS[r].Busy← yes;`
Load only		`RegisterStat[rt].Qi=r;`
Store only		`if (Register Stat[rt].Qi ≠0) {RS[r].Qk← RegisterStat[rs].Qi} else {RS[r].Vk← Regs[rt]; RS[r].Qk← 0};`
Execute FP Operation	(RS[r].Qj=0) and (RS[r].Qk=0)	Compute result: operands are in Vj and Vk
Load/Store step 1	RS[r].Qj=0 & r is head of load/store queue	`RS[r].A←RS[r].Vj + RS[r].A;`
Load step 2	RS[r].A<>0	Read from Mem[RS[r].A]
Write result FP Operation or Load	Execution complete at r & CDB available	`∀x(if (RegisterStat[x].Qi=r) {Regs[x]← result; RegisterStat[x].Qi← 0}); ∀x(if (RS[x].Qj=r) {RS[x].Vj← result;RS[x].Qj ← 0}); ∀x(if (RS[x].Qk=r) {RS[x].Vk← result;RS[x].Qk ← 0}); RS[r].Busy← no;`
Store	Execution complete at r & RS[r].Qk=0	`Mem[RS[r].A]←RS[r].Vk; RS[r].Busy← no;`

FIGURE 3.5 Steps in the algorithm and what is required for each step. For the issuing instruction, `rd` is the destination, `rs` and `rt` are the source register numbers, `imm` is the sign-extended immediate field, and `r` is the reservation station or buffer that the instruction is assigned to. `RS` is the reservation-station data structure. The value returned by a FP unit or by the load unit is called `result`. `RegisterStat` is the register status data structure (not the register file, which is `Regs[]`). When an instruction is issued, the destination register has its Qi field set to the number of the buffer or reservation station to which the instruction is issued. If the operands are available in the registers, they are stored in the V fields. Otherwise, the Q fields are set to indicate the reservation station that will produce the values needed as source operands. The instruction waits at the reservation station until both its operands are available, indicated by zero in the Q fields. The Q fields are set to zero either when this instruction is issued, or when an instruction on which this instruction depends completes and does its write back. When an instruction has finished execution and the CDB is available, it can do its write back. All the buffers, registers, and reservation stations whose value of Qj or Qk is the same as the completing reservation station update their values from the CDB and mark the Q fields to indicate that values have been received. Thus, the CDB can broadcast its result to many destinations in a single clock cycle, and if the waiting instructions have their operands, they can all begin execution on the next clock cycle. Loads go through two steps in Execute, and stores perform slightly differently during Write Result, where they may have to wait for the value to store. Remember that to preserve exception behavior, instructions should not be allowed to execute if a branch that is earlier in program order has not yet completed. Because any concept of program order is not maintained after the Issue stage, this restriction is usually implemented by preventing any instruction from leaving the Issue step, if there is a pending branch already in the pipeline. In Section 3.7, we will see how speculation support removes this restriction.

Tomasulo's Algorithm: A Loop-Based Example

To understand the full power of eliminating WAW and WAR hazards through dynamic renaming of registers, we must look at a loop. Consider the earlier following simple sequence for multiplying the elements of an array by a scalar in F2:

```
Loop:  L.D    F0,0(R1)
       MUL.D  F4,F0,F2
       S.D    F4,0(R1)
       DADDUI R1,R1,-8
       BNE    R1,R2,Loop; branches if R1≠0
```

If we predict that branches are taken, using reservation stations will allow multiple executions of this loop to proceed at once. This advantage is gained without changing the code—in effect, the loop is unrolled dynamically by the hardware, using the reservation stations obtained by renaming to act as additional registers.

Let's assume we have issued all the instructions in two successive iterations of the loop, but none of the floating-point loads-stores or operations has completed. The reservation stations, register-status tables, and load and store buffers at this point are shown in Figure 3.6. (The integer ALU operation is ignored, and it is assumed the branch was predicted as taken.) Once the system reaches this state, two copies of the loop could be sustained with a CPI close to 1.0 provided the multiplies could complete in four clock cycles. As we will see later in this chapter, when extended with multiple instruction issue, Tomasulo's approach can sustain more than one instruction per clock.

		Instruction status			
Instruction		**From iteration**	**Issue**	**Execute**	**Write result**
L.D	F0,0(R1)	1	√	√	
MUL.D	F4,F0,F2	1	√		
S.D	F4,0(R1)	1	√		
L.D	F0,0(R1)	2	√	√	
MUL.D	F4,F0,F2	2	√		
S.D	F4,0(R1)	2	√		

	Reservation stations						
Name	**Busy**	**Op**	**Vj**	**Vk**	**Qj**	**Qk**	**A**
Load1	yes	Load					Regs[R1]+0
Load2	yes	Load					Regs[R1]-8
Add1	no						
Add2	no						
Add3	no						
Mult1	yes	MUL		Regs[F2]	Load1		
Mult2	yes	MUL		Regs[F2]	Load2		
Store1	yes	Store	Regs[R1]			Mult1	
Store2	yes	Store	Regs[R1]-8			Mult2	

	Register status								
Field	**F0**	**F2**	**F4**	**F6**	**F8**	**F10**	**F12**	**...**	**F30**
Qi	Load2		Mult2						

FIGURE 3.6 Two active iterations of the loop with no instruction yet completed. Entries in the multiplier reservation stations indicate that the outstanding loads are the sources. The store reservation stations indicate that the multiply destination is the source of the value to store.

A load and store can safely be done in a different order, provided the they access different addresses. If a load and a store access the same address, then either:

- the load is before the store in program order and interchanging them results in a WAR hazard, or
- the store is before the load in program order and interchanging them results in a RAW hazard.

Similarly, interchanging two stores to the same address results in WAW hazard.

Hence, to determine if a load can be executed at a given time, the processor can check whether any uncompleted store that precedes the load in program order shares the same data memory address as the load. Similarly, a store must wait until there are no unexecuted loads or stores that are earlier in program order and share the same data memory address.

To detect such hazards, the processor must have computed the data memory address associated with any earlier memory operation. A simple, but not necessarily optimal, way to guarantee that the processor has all such addresses is to perform the effective address calculations in program order. (We really only need keep the relative order between stores and other memory references; that is, loads can be reordered freely.).

Let's consider the situation of a load first. If we perform effective address calculation in program order, then when a load has completed effective address calculation, we can check whether there is an address conflict by examining the A field of all active store buffers. If the load address matches the address of any active entries in the store buffer, the load instruction is not sent to the load buffer until the conflicting store completes. (Some implementations bypass the value directly to the load from a pending store, reducing the delay for this RAW hazard.)

Stores operate similarly, except that the processor must check for conflicts in both the load buffers and the store buffers, since conflicting stores cannot be reordered with respect to either a load or a store. This dynamic disambiguation of addresses is an alternative to the techniques, discussed in the next chapter, that a compiler would use when interchanging a load and store.

A dynamically scheduled pipeline can yield very high performance, provided branches are predicted accurately--an issue we address in the next section. The major drawback of this approach is the complexity of the Tomasulo scheme, which requires a large amount of hardware. In particular, each reservation station must contain an associative buffer, which must run at high speed, as well as complex control logic. Lastly, the performance can be limited by the single completion bus (CDB). Although additional CDBs can be added, each CDB must interact with each the reservation station, and the associative tag-matching hardware would need to be duplicated at each station for each CDB.

In Tomasulo's scheme two different techniques are combined: the renaming of the architectural registers to a larger set of registers and the buffering of source operands from the register file. Source operand buffering resolves WAR hazards that arise when the operand is available in the registers. As we will see later, it is also possible to eliminate WAR hazards by the renaming of a register together with the buffering of a result until no outstanding references to the earlier version of the register remain. This approach will be used when we discuss hardware speculation.

Tomasulo's scheme is particularly appealing if the designer is forced to pipeline an architecture for which it is difficult to schedule code, that has a shortage

of registers, or for which the designer wishes to obtain high performance without pipeline specific compilation. On the other hand, the advantages of the Tomasulo approach versus compiler scheduling for a efficient single-issue pipeline are probably fewer than the costs of implementation. But, as processors become more aggressive in their issue capability and designers are concerned with the performance of difficult-to-schedule code (such as most nonnumeric code), techniques such as register renaming and dynamic scheduling have become more important. Furthermore, the role of dynamic scheduling as a basis for hardware speculation has made this approach very popular in the past five years.

The key components for enhancing ILP in Tomasulo's algorithm are dynamic scheduling, register renaming, and dynamic memory disambiguation. It is difficult to assess the value of these features independently. When we examine the studies of ILP in section 3.8, we will look at how these features affect the amount of parallelism discovered until ideal circumstances.

Corresponding to the dynamic hardware techniques for scheduling around data dependences are dynamic techniques for handling branches efficiently. These techniques are used for two purposes: to predict whether a branch will be taken and to find the target more quickly. *Hardware branch prediction*, the name for these techniques, is the next topic we discuss.

3.4 Reducing Branch Costs with Dynamic Hardware Prediction

The previous section describes techniques for overcoming data hazards. The frequency of branches and jumps demands that we also attack the potential stalls arising from control dependences. Indeed, as the amount of ILP we attempt to exploit grows, control dependences rapidly become the limiting factor. Although schemes in this section are helpful in processors that try to maintain one instruction issue per clock, for two reasons they are *crucial* to any processor that tries to issue more than one instruction per clock. First, branches will arrive up to *n* times faster in an *n*-issue processor and providing an instruction stream to the processor will probably require that we predict the outcome of branches. Second, Amdahl's Law reminds us that relative impact of the control stalls will be larger with the lower potential CPI in such machines.

In the first chapter, we examined a variety of basic schemes (e.g., predict not taken and delayed branch) for dealing with branches. Those schemes were all static: the action taken does not depend on the dynamic behavior of the branch. This section focuses on using hardware to dynamically predict the outcome of a branch—the prediction will depend on the behavior of the branch at runtime and will change if the branch changes its behavior during execution.

We start with a simple branch prediction scheme and then examine approaches that increase the accuracy of our branch prediction mechanisms. After that, we look at more elaborate schemes that try to find the instruction following a branch even earlier. The goal of all these mechanisms is to allow the processor to resolve the outcome of a branch early, thus preventing control dependences from causing stalls. The effectiveness of a branch prediction scheme depends not only on the accuracy, but also on the cost of a branch when the prediction is correct and when the prediction is incorrect. These branch penalties depend on the structure of the

pipeline, the type of predictor, and the strategies used for recovering from misprediction.

Basic Branch Prediction and Branch-Prediction Buffers

The simplest dynamic branch-prediction scheme is a *branch-prediction buffer* or *branch history table*. A branch-prediction buffer is a small memory indexed by the lower portion of the address of the branch instruction. The memory contains a bit that says whether the branch was recently taken or not. This scheme is the simplest sort of buffer; it has no tags and is useful only to reduce the branch delay when it is longer than the time to compute the possible target PCs. We don't know, in fact, if the prediction is correct—it may have been put there by another branch that has the same low-order address bits. But this doesn't matter. The prediction is a hint that is assumed to be correct, and fetching begins in the predicted direction. If the hint turns out to be wrong, the prediction bit is inverted and stored back. Of course, this buffer is effectively a cache where every access is a hit, and, as we will see, the performance of the buffer depends on both how often the prediction is for the branch of interest and how accurate the prediction is when it matches. Before we analyze the performance, it is useful to make a small, but important, improvement in the accuracy of the branch prediction scheme.

This simple one-bit prediction scheme has a performance shortcoming: Even if a branch is almost always taken, we will likely predict incorrectly twice, rather than once, when it is not taken. The following example shows this.

EXAMPLE Consider a loop branch whose behavior is taken nine times in a row, then not taken once. What is the prediction accuracy for this branch, assuming the prediction bit for this branch remains in the prediction buffer?

ANSWER The steady-state prediction behavior will mispredict on the first and last loop iterations. Mispredicting the last iteration is inevitable since the prediction bit will say taken (the branch has been taken nine times in a row at that point). The misprediction on the first iteration happens because the bit is flipped on prior execution of the last iteration of the loop, since the branch was not taken on that iteration. Thus, the prediction accuracy for this branch that is taken 90% of the time is only 80% (two incorrect predictions and eight correct ones). In general, for branches used to form loops—a branch is taken many times in a row and then not taken once—a one-bit predictor will mispredict at twice the rate that the branch is not taken. It seems that we should expect that the accuracy of the predictor would at least match the taken branch frequency for these highly regular branches. n

To remedy this, two-bit prediction schemes are often used. In a two-bit scheme, a prediction must miss twice before it is changed. Figure 3.7 shows the finite-state processor for a two-bit prediction scheme.

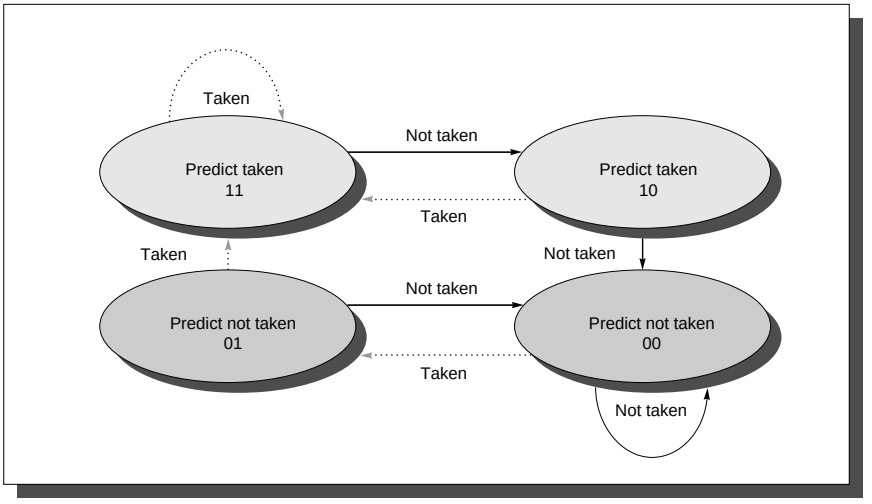


FIGURE 3.7 The states in a two-bit prediction scheme. By using two bits rather than one, a branch that strongly favors taken or not taken—as many branches do—will be mispredicted less often than with a one-bit predictor. The two bits are used to encode the four states in the system. In a counter implementation, the counters are incremented when a branch is taken and decremented when it is not taken; the counters saturate at 00 or 11. One complication of the two-bit scheme is that it updates the prediction bits more often than a one-bit predictor, which only updates the prediction bit on a mispredict. Since we typically read the prediction bits on every cycle, a two-bit predictor will typically need both a read and a write access port.

The two-bit scheme is actually a specialization of a more general scheme that has an *n*-bit saturating counter for each entry in the prediction buffer. With an *n*-bit counter, the counter can take on values between 0 and $2^n - 1$: when the counter is greater than or equal to one half of its maximum value (2^{n-1}), the branch is predicted as taken; otherwise, it is predicted untaken. As in the two-bit scheme, the counter is incremented on a taken branch and decremented on an untaken branch. Studies of *n*-bit predictors have shown that the two-bit predictors do almost as well, and thus most systems rely on two-bit branch predictors rather than the more general *n*-bit predictors.

A branch-prediction buffer can be implemented as a small, special "cache" accessed with the instruction address during the IF pipe stage, or as a pair of bits attached to each block in the instruction cache and fetched with the instruction. If the instruction is decoded as a branch and if the branch is predicted as taken, fetching begins from the target as soon as the PC is known. Otherwise, sequential fetching and executing continue. If the prediction turns out to be wrong, the prediction bits are changed as shown in Figure 3.7.

Although this scheme is useful for most pipelines, the five-stage, classic pipeline finds out both whether the branch is taken and what the target of the branch is at roughly the same time, *assuming* no hazard in accessing the register specified in the conditional branch. (Remember that this is true for the five-stage pipeline because the branch does a compare of a register against zero during the ID stage, which is when the effective address is also computed.) Thus, this scheme does not help for the five-stage pipeline; we will explore a scheme that can work for such pipelines, and for machines issuing multiple instructions per clock, a little later. First, let's see how well branch prediction works in general.

What kind of accuracy can be expected from a branch-prediction buffer using two bits per entry on real applications? For the SPEC89 benchmarks a branch-prediction buffer with 4096 entries results in a prediction accuracy ranging from over 99% to 82%, or a *misprediction rate* of 1% to 18%, as shown in Figure 3.8.

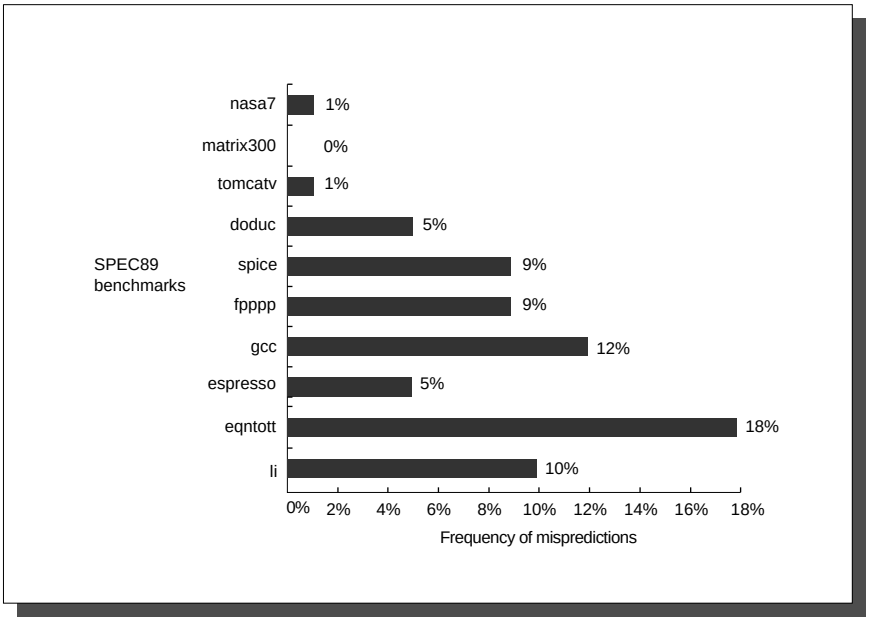


FIGURE 3.8 Prediction accuracy of a 4096-entry two-bit prediction buffer for the SPEC89 benchmarks. The misprediction rate for the integer benchmarks (gcc, espresso, eqntott, and li) is substantially higher (average of 11%) than that for the FP programs (average of 4%). Even omitting the FP kernels (nasa7, matrix300, and tomcatv) still yields a higher accuracy for the FP benchmarks than for the integer benchmarks. These data, as well as the rest of the data in this section, are taken from a branch prediction study done using the IBM Power architecture and optimized code for that system. See Pan et al. [1992].

To show the differences more clearly, we plot misprediction frequency rather

than prediction frequency. A 4K-entry buffer, like that used for these results, is considered large; smaller buffers would have worse results.

Knowing just the prediction accuracy, as shown in Figure 3.8, is not enough to determine the performance impact of branches, even given the branch costs and penalties for misprediction. We also need to take into account the branch frequency, since the importance of accurate prediction is larger in programs with higher branch frequency. For example, the integer programs—li, eqntott, espresso, and gcc—have higher branch frequencies than those of the more easily predicted FP programs.

As we try to exploit more ILP, the accuracy of our branch prediction becomes critical. As we can see in Figure 3.8, the accuracy of the predictors for integer programs, which typically also have higher branch frequencies, is lower than for the loop-intensive scientific programs. We can attack this problem in two ways: by increasing the size of the buffer and by increasing the accuracy of the scheme we use for each prediction. A buffer with 4K entries is already large and, as Figure 3.9 shows, performs quite comparably to an infinite buffer. The data in Figure 3.9 make it clear that the hit rate of the buffer is not the limiting factor. As we mentioned above, simply increasing the number of bits per predictor without changing the predictor structure also has little impact. Instead, we need to look at how we might increase the accuracy of each predictor.

Correlating Branch Predictors

These two-bit predictor schemes use only the recent behavior of a single branch to predict the future behavior of that branch. It may be possible to improve the prediction accuracy if we also look at the recent behavior of *other* branches rather than just the branch we are trying to predict. Consider a small code fragment from the SPEC92 benchmark eqntott (the worst case for the two-bit predictor):

```
if (aa==2)
       aa=0;
if (bb==2)
       bb=0;
if (aa!=bb) {
```

Here is the MIPS code that we would typically generate for this code fragment assuming that `aa` and `bb` are assigned to registers R1 and R2:

```
       DSUBUI  R3,R1,#2
       BNEZ    R3,L1          ;branch b1 (aa!=2)
       DADD    R1,R0,R0       ;aa=0
L1:    DSUBUI  R3,R2,#2
       BNEZ    R3,L2          ;branch b2(bb!=2)
       DADD    R2,R0,R0       ; bb=0
```

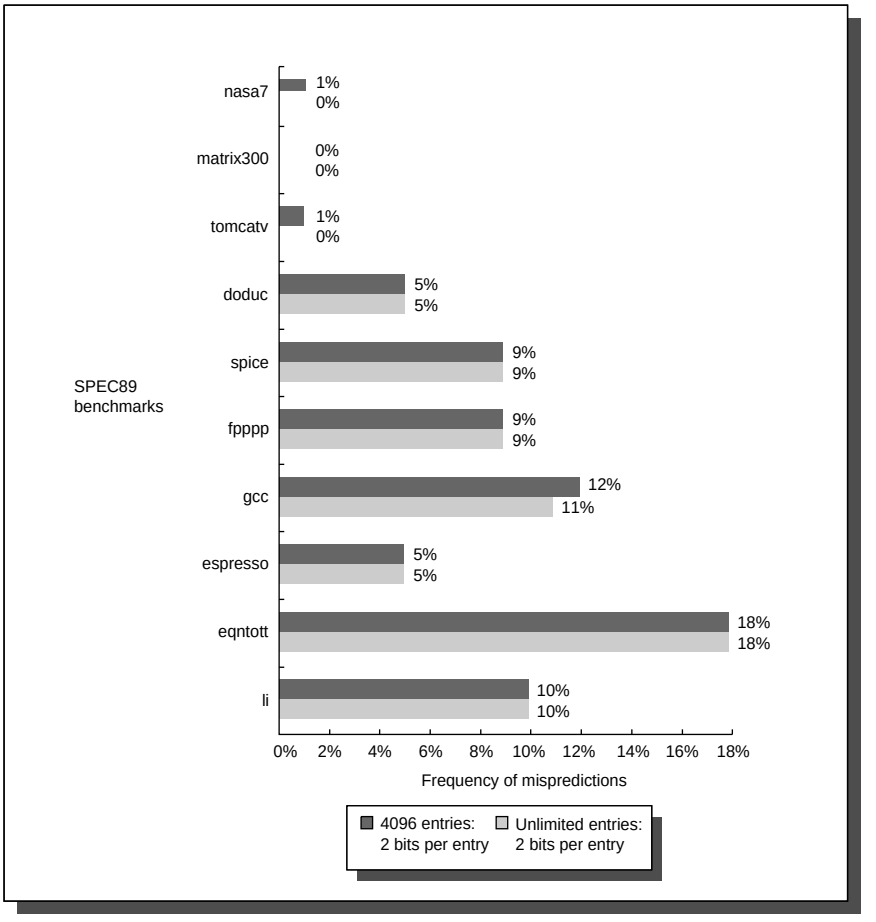


FIGURE 3.9 Prediction accuracy of a 4096-entry two-bit prediction buffer versus an infinite buffer for the SPEC89 benchmarks.

```
L2:     DSUBU   R3,R1,R2        ;R3=aa-bb
        BEQZ    R3,L3           ;branch b3 (aa==bb)
```

Let's label these branches b1, b2, and b3. The key observation is that the behavior of branch b3 is correlated with the behavior of branches b1 and b2. Clearly, if branches b1 and b2 are both not taken (i.e., the if conditions both evaluate to true and `aa` and `bb` are both assigned 0), then b3 will be taken, since `aa` and `bb` are clearly equal. A predictor that uses only the behavior of a single branch to predict the outcome of that branch can never capture this behavior.

Branch predictors that use the behavior of other branches to make a prediction are called *correlating predictors* or *two-level predictors*. To see how such predic-

tors work, let's choose a simple hypothetical case. Consider the following simplified code fragment (chosen for illustrative purposes):

```
if (d==0)
        d=1;
if (d==1)
```

Here is the typical code sequence generated for this fragment, assuming that `d` is assigned to R1:

```
        BNEZ    R1,L1;branch b1(d!=0)
        DADDIU R1,R0,#1;d==0, so d=1
L1:     DADDIU R3,R1,#-1
        BNEZ    R3,L2;branch b2(d!=1)
...
L2:
```

The branches corresponding to the two if statements are labeled b1 and b2. The possible sequences for an execution of this fragment, assuming `d` has values 0, 1, and 2, are shown in Figure 3.10. To illustrate how a correlating predictor works, assume the sequence above is executed repeatedly and ignore other branches in the program (including any branch needed to cause the above sequence to repeat).

Initial value of d	d==0?	b1	Value of d before b2	d==1?	b2
0	yes	not taken	1	yes	not taken
1	no	taken	1	yes	not taken
2	no	taken	2	no	taken

FIGURE 3.10 Possible execution sequences for a code fragment.

From Figure 3.10, we see that if b1 is not taken, then b2 will be not taken. A correlating predictor can take advantage of this, but our standard predictor cannot. Rather than consider all possible branch paths, consider a sequence where `d` alternates between 2 and 0. A one-bit predictor initialized to not taken has the behavior shown in Figure 3.11. As the figure shows, *all* the branches are mispredicted!

d=?	b1 prediction	b1 action	New b1 prediction	b2 prediction	b2 action	New b2 prediction
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT

FIGURE 3.11 Behavior of a one-bit predictor initialized to not taken. T stands for taken, NT for not taken.

Alternatively, consider a predictor that uses one bit of correlation. The easiest way to think of this is that every branch has two separate prediction bits: one prediction assuming the last branch executed was not taken and another prediction that is used if the last branch executed was taken. Note that, in general, the last branch executed is *not* the same instruction as the branch being predicted, though this can occur in simple loops consisting of a single basic block (since there are no other branches in the loops).

We write the pair of prediction bits together, with the first bit being the prediction if the last branch in the program is not taken and the second bit being the prediction if the last branch in the program is taken. The four possible combinations and the meanings are listed in Figure 4.18.

Prediction bits	Prediction if last branch not taken	Prediction if last branch taken
NT/NT	not taken	not taken
NT/T	not taken	taken
T/NT	taken	not taken
T/T	taken	taken

FIGURE 3.12 Combinations and meaning of the taken/not taken prediction bits. T stands for taken, NT for not taken.

The action of the one-bit predictor with one bit of correlation, when initialized to NT/NT is shown in Figure 3.13.

d=?	b1 prediction	b1 action	New b1 prediction	b2 prediction	b2 action	New b2 prediction
2	**NT**/NT	T	T/NT	NT/**NT**	T	NT/T
0	T/**NT**	NT	T/NT	**NT**/T	NT	NT/T
2	**T**/NT	T	T/NT	NT/**T**	T	NT/T
0	T/**NT**	NT	T/NT	**NT**/T	NT	NT/T

FIGURE 3.13 The action of the one-bit predictor with one bit of correlation, initialized to not taken/not taken. T stands for taken, NT for not taken. The prediction used is shown in bold.

In this case, the only misprediction is on the first iteration, when d = 2. The correct prediction of b1 is because of the choice of values for d, since b1 is not obviously correlated with the previous prediction of b2. The correct prediction of b2, however, shows the advantage of correlating predictors. Even if we had chosen different values for d, the predictor for b2 would correctly predict the case when b1 is not taken on every execution of b2 after one initial incorrect prediction.

The predictor in Figures 3.12 and 3.13 is called a (1,1) predictor since it uses the behavior of the last branch to choose from among a pair of one-bit branch

predictors. In the general case an (*m*,*n*) predictor uses the behavior of the last *m* branches to choose from 2^m branch predictors, each of which is a *n*-bit predictor for a single branch. The attraction of this type of correlating branch predictor is that it can yield higher prediction rates than the two-bit scheme and requires only a trivial amount of additional hardware. The simplicity of the hardware comes from a simple observation: The global history of the most recent *m* branches can be recorded in an *m*-bit shift register, where each bit records whether the branch was taken or not taken. The branch-prediction buffer can then be indexed using a concatenation of the low-order bits from the branch address with the *m*-bit global history. For example, Figure 3.14 shows a (2,2) predictor and how the prediction is accessed.

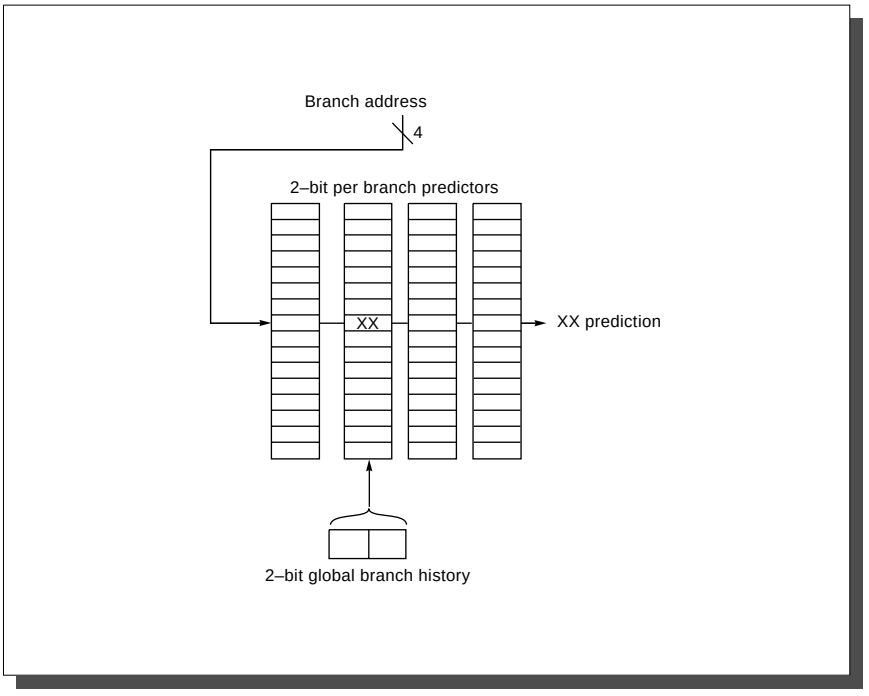


FIGURE 3.14 A (2,2) branch-prediction buffer uses a two-bit global history to choose from among four predictors for each branch address. Each predictor is in turn a two-bit predictor for that particular branch. The branch-prediction buffer shown here has a total of 64 entries; the branch address is used to choose four of these entries and the global history is used to choose one of the four. The two-bit global history can be implemented as a shifter register that simply shifts in the behavior of a branch as soon as it is known.

There is one subtle effect in this implementation. Because the prediction buffer is not a cache, the counters indexed by a single value of the global predictor may in fact correspond to different branches at some point in time. This insight is no different from our earlier observation that the prediction may not correspond to the current branch. In Figure 3.14 we draw the buffer as a two-dimensional object to ease understanding. In reality, the buffer can simply be implemented as a linear memory array that is two bits wide; the indexing is done by concatenating the global history bits and the number of required bits from the branch address. For the example in Figure 3.14, a (2,2) buffer with 64 total entries, the four low-order address bits of the branch (word address) and the two global bits form a six-bit index that can be used to index the 64 counters.

How much better do the correlating branch predictors work when compared with the standard two-bit scheme? To compare them fairly, we must compare predictors that use the same number of state bits. The number of bits in an (m,n) predictor is

$$2^m \times n \times \text{Number of prediction entries selected by the branch address}$$

A two-bit predictor with no global history is simply a (0,2) predictor.

EXAMPLE How many bits are in the (0,2) branch predictor we examined earlier? How many bits are in the branch predictor shown in Figure 3.14?

ANSWER The earlier predictor had 4K entries selected by the branch address. Thus the total number of bits is

$$2^0 \times 2 \times 4\text{K} = 8\text{K}.$$

The predictor in Figure 3.14 has

$$2^2 \times 2 \times 16 = 128 \text{ bits}.$$

n

To compare the performance of a correlating predictor with that of our simple two-bit predictor examined in Figure 3.8, we need to determine how many entries we should assume for the correlating predictor.

EXAMPLE How many branch-selected entries are in a (2,2) predictor that has a total of 8K bits in the prediction buffer?

ANSWER We know that

$$2^2 \times 2 \times \text{Number of prediction entries selected by the branch} = 8\text{K}.$$

Hence

$$\text{Number of prediction entries selected by the branch} = 1\text{K}.$$

n

Figure 3.15 compares the performance of the earlier two-bit simple predictor

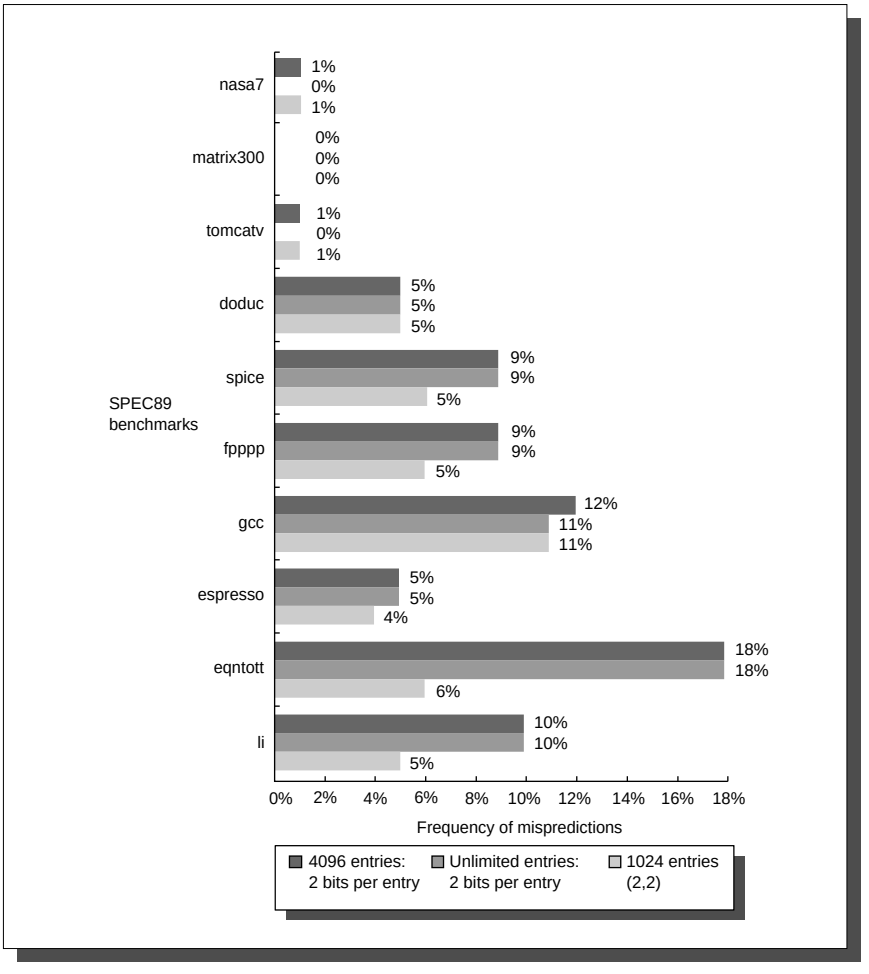


FIGURE 3.15 Comparison of two-bit predictors. A noncorrelating predictor for 4096 bits is first, followed by a noncorrelating two-bit predictor with unlimited entries and a two-bit predictor with two bits of global history and a total of 1024 entries.

with 4K entries and a (2,2) predictor with 1K entries. As you can see, this predictor not only outperforms a simple two-bit predictor with the same total number of state bits, it often outperforms a two-bit predictor with an unlimited number of entries.

There are a wide spectrum of correlating predictors, with the (0,2) and (2,2) predictors being among the most interesting. The Exercises ask you to explore the performance of a third extreme: a predictor that does not rely on the branch address. For example, a (12,2) predictor that has a total of 8K bits does not use the branch address in indexing the predictor, but instead relies solely on the global branch history. Surprisingly, this degenerate case can outperform a noncorrelating two-bit predictor if enough global history is used and the table is large enough!

Tournament Predictors: Adaptively Combining Local and Global Predictors

The primary motivation for correlating branch predictors came from the observation that the standard 2-bit predictor using only local information failed on some important branches and that by adding global information, the performance could be improved. Tournament predictors take this insight to the next level, by using multiple predictors, usually one based on global information and one based on local information, and combining them with a selector. Tournament predictors can achieve both better accuracy at medium sizes (8Kb-32Kb) and also make use of very large numbers of prediction bits effectively.

Tournament predictors are the most popular form of *multilevel branch predictors*. A multilevel branch predictor use several levels of branch prediction tables together with an algorithm for choosing among the multiple predictors; we will see several variations on multilevel predictors in this section. Existing tournament predictors use a 2-bit saturating counter per branch to choose among two different predictors. The four states of the counter dictate whether to use predictor 1 or predictor 2. The state transition diagram is shown in Figure 3.16.

The advantage of a tournament predictor is its ability to select the right predictor for the right branch. Figure 3.17 shows how the tournament predictor selects between a local and global predictor depending on the benchmark, as well as on the branch. The ability to choose between a prediction based on strictly local information and one incorporating global information on a per branch basis is particularly critical in the integer benchmarks.

Figure 3.18 looks at the performance of three different predictors (a local 2-bit predictor, a correlating predictor, and a tournament predictor) for different numbers of bits using SPEC89 as the benchmark. As we saw earlier, the prediction capability of the local predictor does not improve beyond a certain size. The correlating predictor shows a significant improvement, and the tournament predictor generates slightly better performance.

An Example: the Alpha 21264 Branch Predictor

The 21264 uses the most sophisticated branch predictor in any processor as of 2001. The 21264 has a tournament predictor using 4K 2-bit counters indexed by the local branch address to choose from among a global predictor and a local pre-

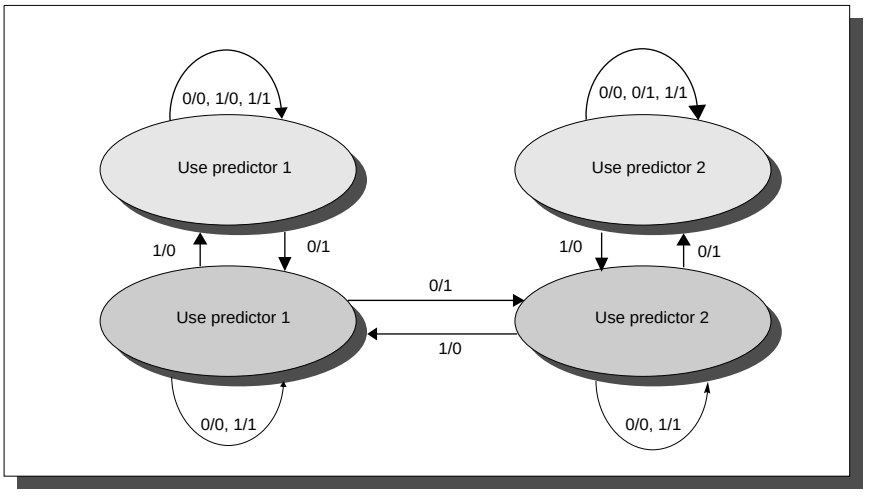


FIGURE 3.16 The state transition diagram for a tournament predictor has four states corresponding to which predictor to use. The counter is incremented whenever the "predicted" predictor is correct and the other predictor is incorrect, and it is decremented in the reverse situation.

dictor. The global predictor also has 4K entries and is indexed by the history of the last 12 branches; each entry in the global predictor is a standard 2-bit predictor.

The local predictor consists of a two-level predictor. The top level is a local history table consisting of 1024 10-bit entries; each 10-bit entry corresponds to the most recent ten branch outcomes for the entry. That is, if the branch was taken 10 or more times in a row, the entry in the local history table will be all 1s. If the branch is alternately taken and untaken the history entry consist of alternating 0s nd 1s. This 10-bit history allows patterns of up to ten branches to be discovered and predicted. The selected entry from the local history table is used to index a table of 1K entries consisting a three-bit saturating counters, which provide the local prediction. This combination, which uses a total of 29 Kbits, leads to high accuracy in branch prediction. For the SPECfp95 benchmarks there is less than

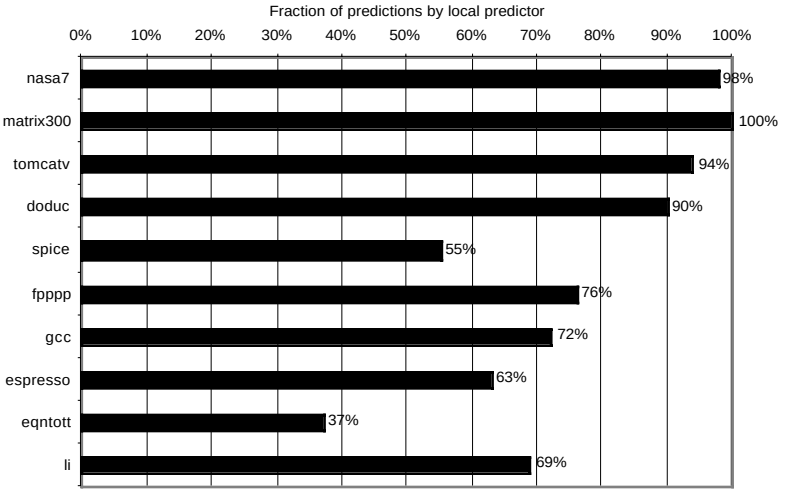


FIGURE 3.17 The fraction of predictions coming from the local predictor for a tournament predictor using the SPEC89 benchmarks. The tournament predictor selects between a local 2-bit predictor and a 2-bit local/global predictor, called gshare. Gshare is indexed by an exclusive or of the branch address bits and the global history; it performs similarly to the correlating predictor discussed earlier. In this case each predictor has 1,024 entries, each 2-bits, for a total of 6Kbits.

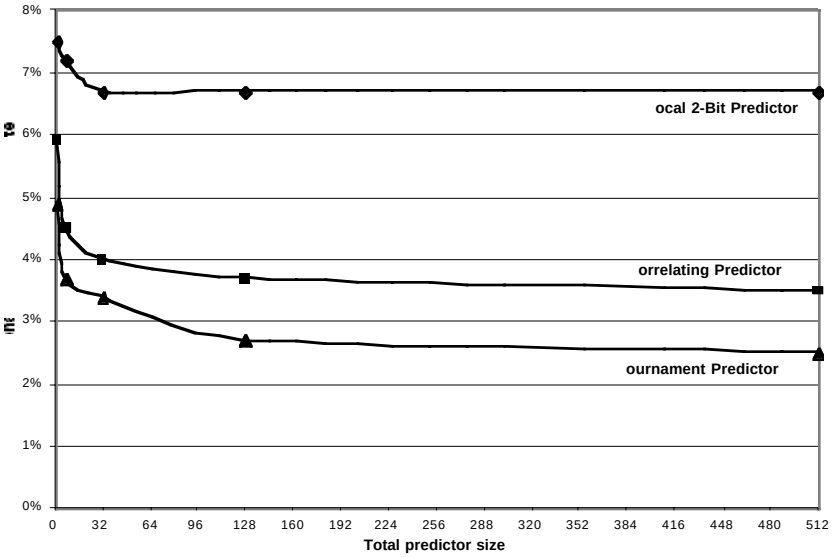


FIGURE 3.18 The misprediction rate for three different predictors on SPEC89 as the total number of bits is increased. The predictors are: a local 2-bit predictor, a correlating predictor, which is optimally structured at each point in the graph, and a tournament predictor using the same structure as in Figure 3.17.

one misprediction per 1000 completed instructions, and for SPECint95, there are about 11.5 mispredictions per 1000 completed instructions.

3.5 High Performance Instruction Delivery

In a high performance pipeline, especially one with multiple issue, predicting branches well is not enough: we actually have to be able to deliver a high bandwidth instruction stream. In recent multiple issue processors, this has meant delivering 4-8 instructions every clock cycle. To accomplish this, we consider three concepts in this section: a branch target buffer, an integrated instruction fetch unit, and dealing with indirect branches, by predicting return addresses.

Branch Target Buffers

To reduce the branch penalty for our five-stage pipeline, we need to know from what address to fetch by the end of IF. This requirement means we must know whether the as-yet-undecoded instruction is a branch and, if so, what the next PC should be. If the instruction is a branch and we know what the next PC should be, we can have a branch penalty of zero. A branch-prediction cache that stores the predicted address for the next instruction after a branch is called a *branch-target buffer* or *branch-target cache*.

For the classic, five-stage pipeline, a branch-*prediction* buffer is accessed during the ID cycle, so that at the end of ID we know the branch-target address (since it is computed during ID), the fall-through address (computed during IF), and the prediction. Thus, by the end of ID we know enough to fetch the next predicted instruction. For a branch-*target* buffer, we access the buffer during the IF stage using the instruction address of the fetched instruction, a possible branch, to index the buffer. If we get a hit, then we know the predicted instruction address at the end of the IF cycle, which is one cycle earlier than for a branch-prediction buffer.

Because we are predicting the next instruction address and will send it out *before* decoding the instruction, we *must* know whether the fetched instruction is predicted as a taken branch. Figure 3.19 shows what the branch-target buffer looks like. If the PC of the fetched instruction matches a PC in the buffer, then the corresponding predicted PC is used as the next PC. In Chapter 5 we will discuss caches in much more detail; we will see that the hardware for this branch-target buffer is essentially identical to the hardware for a cache.

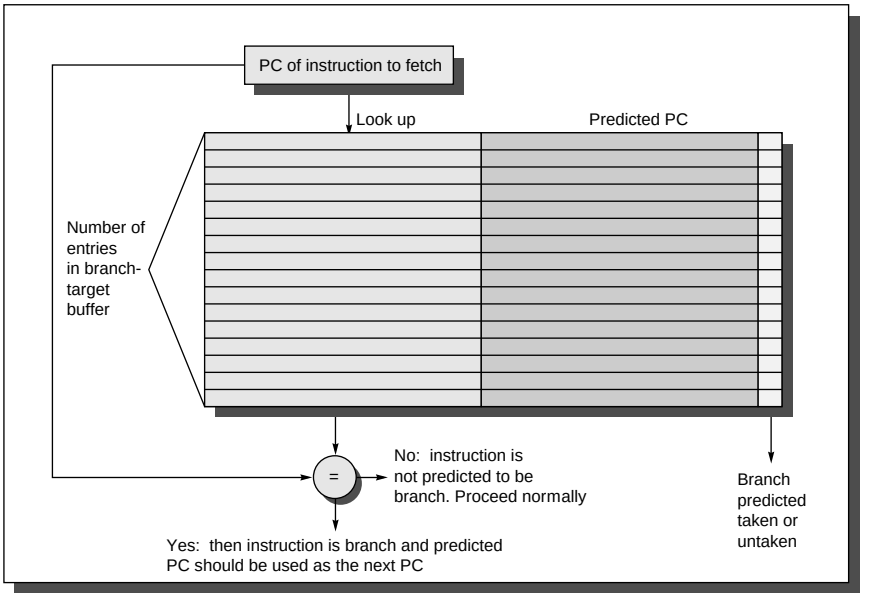


FIGURE 3.19 A branch-target buffer. The PC of the instruction being fetched is matched against a set of instruction addresses stored in the first column; these represent the addresses of known branches. If the PC matches one of these entries, then the instruction being fetched is a taken branch, and the second field, predicted PC, contains the prediction for the next PC after the branch. Fetching begins immediately at that address. The third field, which is optional, may be used for extra prediction state bits.

If a matching entry is found in the branch-target buffer, fetching begins immediately at the predicted PC. Note that (unlike a branch-prediction buffer) the entry must be for this instruction, because the predicted PC will be sent out before it is known whether this instruction is even a branch. If we did not check whether the entry matched this PC, then the wrong PC would be sent out for instructions that were not branches, resulting in a slower processor. We only need to store the predicted-taken branches in the branch-target buffer, since an untaken branch follows the same strategy (fetch the next sequential instruction) as a nonbranch. Complications arise when we are using a two-bit predictor, since this requires that we store information for both taken and untaken branches. One way to resolve this is to use both a target buffer and a prediction buffer, which is the solution used by several PowerPC processors. We assume that the buffer only holds PC-relative conditional branches, since this makes the target address a constant; it is not hard to extend the mechanism to work with indirect branches.

Figure 3.20 shows the steps followed when using a branch-target buffer and where these steps occur in the pipeline. From this we can see that there will be no branch delay if a branch-prediction entry is found in the buffer and is correct. Otherwise, there will be a penalty of at least two clock cycles. In practice, this penalty could be larger, since the branch-target buffer must be updated. We could assume that the instruction following a branch or at the branch target is not a branch, and do the update during that instruction time; however, this does complicate the control. Instead, we will take a two-clock-cycle penalty when the branch is not correctly predicted or when we get a miss in the buffer. Dealing with the mispredictions and misses is a significant challenge, since we typically will have to halt instruction fetch while we rewrite the buffer entry. Thus, we would like to make this process fast to minimize the penalty.

To evaluate how well a branch-target buffer works, we first must determine the penalties in all possible cases. Figure 3.21 contains this information.

EXAMPLE Determine the total branch penalty for a branch-target buffer assuming the penalty cycles for individual mispredictions from Figure 3.21. Make the following assumptions about the prediction accuracy and hit rate:

- prediction accuracy is 90% (for instructions in the buffer)
- hit rate in the buffer is 90% (for branches predicted taken)

Assume that 60% of the branches are taken.

ANSWER We compute the penalty by looking at the probability of two events: the branch is predicted taken but ends up being not taken, and the branch is taken but is not found in the buffer. Both carry a penalty of two cycles.

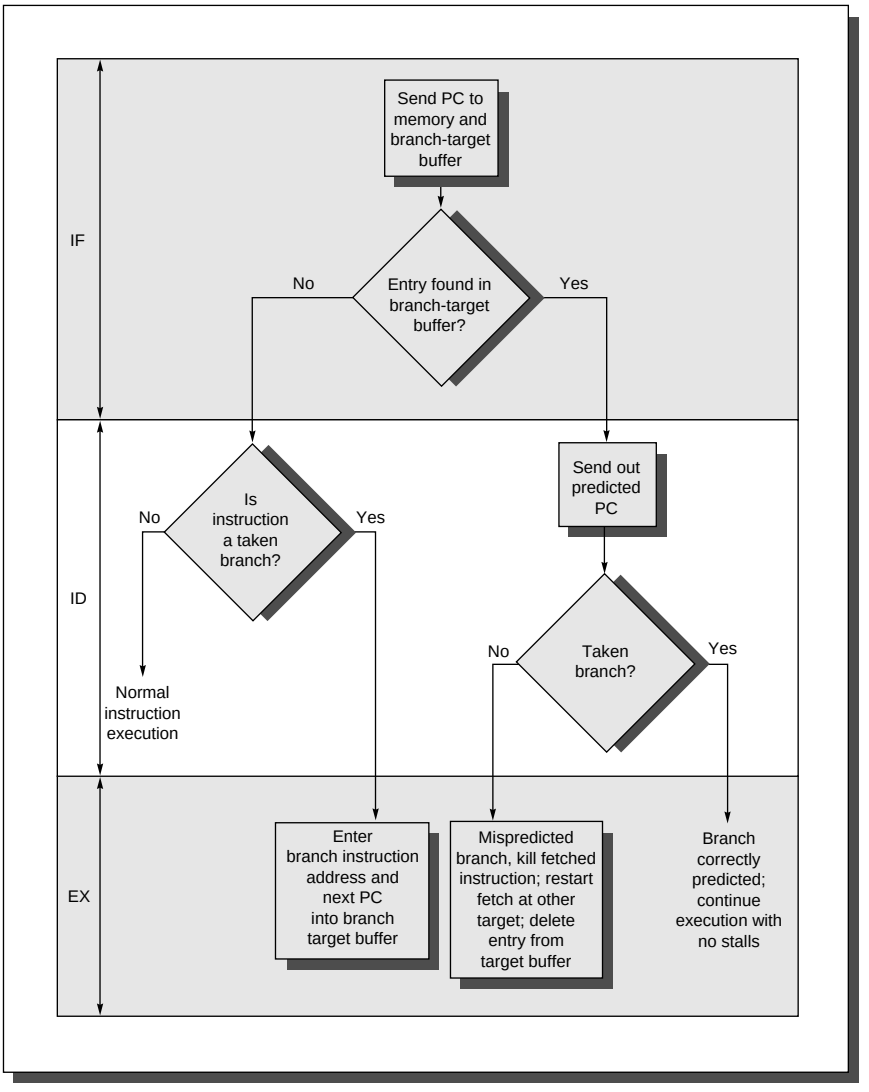


FIGURE 3.20 The steps involved in handling an instruction with a branch-target buffer. If the PC of an instruction is found in the buffer, then the instruction must be a branch that is predicted taken; thus, fetching immediately begins from the predicted PC in ID. If the entry is not found and it subsequently turns out to be a taken branch, it is entered in the buffer along with the target, which is known at the end of ID. If the entry is found, but the instruction turns out not to be a taken branch, it is removed from the buffer. If the instruction is a branch, is found, and is correctly predicted, then execution proceeds with no delays. If the prediction is incorrect, we suffer a one-clock-cycle delay fetching the wrong instruction and restart the fetch one clock cycle later, leading to a total mispredict penalty of two clock cycles. If the branch is not found in the buffer and the instruction turns out to be a branch, we will have proceeded as if the instruction were not a branch and can turn this into an assume-not-taken strategy. The penalty will differ depending on whether the branch is actually taken or not.

Instruction in buffer	Prediction	Actual branch	Penalty cycles
yes	taken	taken	0
yes	taken	not taken	2
no		taken	2
no		not taken	0

FIGURE 3.21 Penalties for all possible combinations of whether the branch is in the buffer and what it actually does, assuming we store only taken branches in the buffer. There is no branch penalty if everything is correctly predicted and the branch is found in the target buffer. If the branch is not correctly predicted, the penalty is equal to one clock cycle to update the buffer with the correct information (during which an instruction cannot be fetched) and one clock cycle, if needed, to restart fetching the next correct instruction for the branch. If the branch is not found and taken, a two-cycle penalty is encountered, during which time the buffer is updated.

$$\begin{aligned}
\text{Probability (branch in buffer, but actually not taken)} &= \text{Percent buffer hit rate} \times \text{Percent incorrect predictions} \\
&= 90\% \times 10\% = 0.09 \\
\text{Probability (branch not in buffer, but actually taken)} &= 10\% \\
\text{Branch penalty} &= (0.09 + 0.10) \times 2 \\
\text{Branch penalty} &= 0.38
\end{aligned}$$

This penalty compares with a branch penalty for delayed branches, which we evaluated in Chapter 1, of about 0.5 clock cycles per branch. Remember, though, that the improvement from dynamic branch prediction will grow as the branch delay grows; in addition, better predictors will yield a larger performance advantage. n

One variation on the branch-target buffer is to store one or more *target instructions* instead of, or in addition to, the predicted *target address*. This variation has two potential advantages. First, it allows the branch-target buffer access to take longer than the time between successive instruction fetches, possibly allowing a larger branch-target buffer. Second, buffering the actual target instructions allows us to perform an optimization called *branch folding*. Branch folding can be used to obtain zero-cycle unconditional branches, and sometimes zero-cycle conditional branches. Consider a branch-target buffer that buffers instructions from the predicted path and is being accessed with the address of an unconditional branch. The only function of the unconditional branch is to change the PC. Thus, when the branch-target buffer signals a hit and indicates that the branch is unconditional, the pipeline can simply substitute the instruction from the branch-target buffer in place of the instruction that is returned from the cache (which is the unconditional branch). If the processor is issuing multiple instructions per cycle, then the buffer will need to supply multiple instructions to obtain the maxi-

mum benefit. In some cases, it may be possible to eliminate the cost of a conditional branch when the condition codes are preset.

Integrated Instruction Fetch Units

To meet the demands of multiple issue processor many recent designers have chosen to implement an integrated instruction fetch unit, as a separate autonomous unit that feeds instructions to the rest of the pipeline. Essentially, this amounts to recognizing that characterizing instruction fetch as a simple single pipestage given the complexities of multiple issue is no longer valid.

Instead, recent designs have used an integrated instruction fetch unit that integrates several functions:

1. Integrated branch prediction: the branch predictor becomes part of the instruction fetch unit and is constantly predicting branches, so to drive the fetch pipeline.
2. Instruction prefetch: to deliver multiple instructions per clock, the instruction fetch unit will likely need to fetch ahead. The unit autonomously manages the prefetching of instructions (see Chapter 5 for discussion of techniques for doing this), integrating it with branch prediction.
3. Instruction memory access and buffering: when fetching multiple instructions per cycle a variety of complexities are encountered, including the difficulty that fetching multiple instructions may require accessing multiple cache lines. The instruction fetch unit encapsulates this complexity, using prefetch to try to hide the cost of crossing cache blocks. The instruction fetch unit also provides buffering, essentially acting as an on-demand unit to provide instructions to the issue stage as needed and in the quantity needed.

As designers try to increase the number of instructions executed per clock, instruction fetch will become an ever more significant bottleneck and clever new ideas will be needed to deliver instructions at the necessary rate. One of the emerging ideas, called *trace caches*, is discussed in Chapter 5.

Return Address Predictors

Another method that designers have studied and included in many recent processors is a technique for predicting indirect jumps, that is, jumps whose destination address varies at runtime. Although high-level language programs will generate such jumps for indirect procedure calls, select or case statements, and FORTRAN-computed gotos, the vast majority of the indirect jumps come from procedure returns. For example, for the SPEC89 benchmarks procedure returns account for 85% of the indirect jumps on average. For languages like C++ and Java, procedure returns are even more frequent. Thus, focusing on procedure returns seems appropriate.

Though procedure returns can be predicted with a branch-target buffer, the accuracy of such a prediction technique can be low if the procedure is called from

multiple sites and the calls from one site are not clustered in time. To overcome this problem, the concept of a small buffer of return addresses operating as a stack has been proposed. This structure caches the most recent return addresses: pushing a return address on the stack at a call and popping one off at a return. If the cache is sufficiently large (i.e., as large as the maximum call depth), it will predict the returns perfectly. Figure 3.22 shows the performance of such a return buffer with 1–16 elements for a number of the SPEC benchmarks. We will use this type of return predictor when we examine the studies of ILP in section 3.8.

Branch prediction schemes are limited both by prediction accuracy and by the penalty for misprediction. As we have seen, typical prediction schemes achieve prediction accuracy in the range of 80–95% depending on the type of program and the size of the buffer. In addition to trying to increase the accuracy of the predictor, we can try to reduce the penalty for misprediction. The penalty can be reduced by fetching from both the predicted and unpredicted direction. Fetching both paths requires that the memory system be dual-ported, have an interleaved cache, or fetch from one path and then the other. Although this adds cost to the system, it may be the only way to reduce branch penalties below a certain point. Caching addresses or instructions from multiple paths in the target buffer is another alternative that some processors have used.

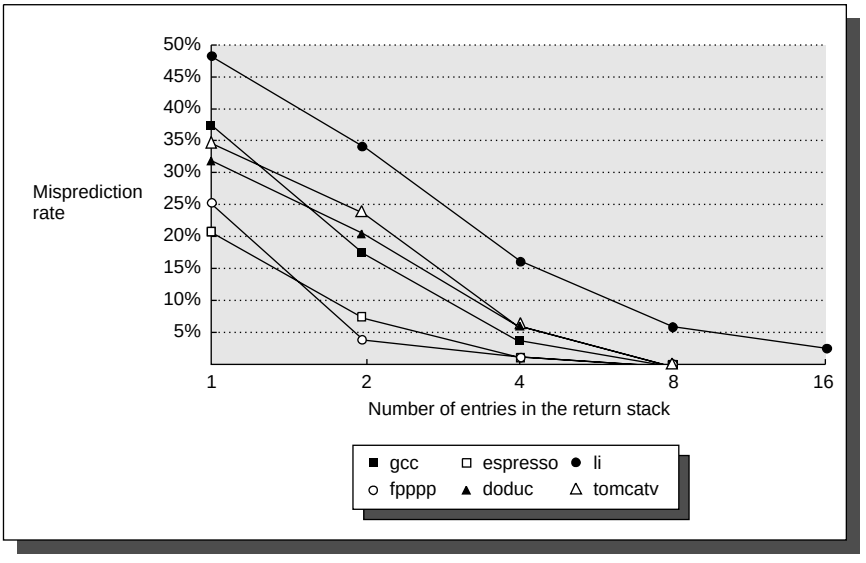


FIGURE 3.22 Prediction accuracy for a return address buffer operated as a stack. The accuracy is the fraction of return addresses predicted correctly. Since call depths are typically not large, with some exceptions, a modest buffer works well. On average returns account for 81% of the indirect jumps in these six benchmarks.

We have seen a variety of software-based static schemes and hardware-based dynamic schemes for trying to boost the performance of our pipelined processor. These schemes attack both the data dependences (discussed in the previous subsections) and the control dependences (discussed in this subsection). Our focus to date has been on sustaining the throughput of the pipeline at one instruction per clock. In the next section we will look at techniques that attempt to exploit more parallelism by issuing multiple instructions in a clock cycle.

3.6 Taking Advantage of More ILP with Multiple Issue

The techniques of the previous two sections can be used to eliminate data and control stalls and achieve an ideal CPI of 1. To improve performance further we would like to decrease the CPI to less than one. But the CPI cannot be reduced below one if we issue only one instruction every clock cycle.

The goal of the *multiple-issue processors,* discussed in this section, is to allow multiple instructions to issue in a clock cycle. Multiple-issue processors come in two basic flavors: *superscalar* processors and *VLIW* (very long instruction word) processors. Superscalar processors issue varying numbers of instructions per clock and are either statically scheduled (using compiler techniques covered in the next chapter) or dynamically scheduled using techniques based on Tomasulo's algorithm. Statically scheduled processor use in-order execution, while dynamically scheduled processors use out-of-order execution.

VLIW processors, in contrast, issue a fixed number of instructions formatted either as one large instruction or as a fixed instruction packet with the parallelism among instructions explicitly indicated by the instruction (hence, they are also known as EPIC--Explicitly Parallel Instruction Computers). VLIW and EPIC processors are inherently statically scheduled by the compiler. The next chapter covers both VLIWs and the necessary compiler technology in detail, so between this chapter and the next, we will have cover most of the techniques for exploiting instruction level parallelism through multiple issue that are in use in existing processors. Figure 3.23 summarizes the basic approaches to multiple issue, their distinguishing characteristics, and shows processors that use each approach.

Although early superscalar processors used static instruction scheduling, and embedded processors still do, most leading-edge desktop and servers now use superscalars with some degree of dynamic scheduling. In this section, we introduce the superscalar concept with a simple, statically scheduled processor, which will require the techniques from the next chapter to achieve good efficiency. We then explore in detail a dynamically scheduled superscalar that builds on the Tomasulo scheme.

Statically-Scheduled Superscalar Processors

In a typical superscalar processor, the hardware might issue from zero (since it may be stalled) to eight instructions in a clock cycle. In a statically-scheduled superscalar, instructions issue in order and all pipeline hazards are checked for at issue time. The pipeline control logic must check for hazards among the

Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	dynamic	hardware	static	in-order execution	Sun UltraSPARC II/ III
Superscalar (dynamic)	dynamic	hardware	dynamic	some out-of-order execution	HP PA 8500, IBM RS64 III
Superscalar (speculative)	dynamic	hardware	dynamic with speculation	out-of-order execution with speculation	Pentium III/4, MIPS R10K, Alpha 21264
VLIW/LIW	static	software	static	no hazards between issue packets	Trimedia, i860
EPIC	mostly static	mostly software	mostly static	explicit dependences marked by compiler	Itanium

FIGURE 3.23 There are five primary approaches in use for multiple-issue processors, and this table shows the primary characteristics that distinguish them. This chapter has focused on the hardware-intensive techniques, which are all some form of superscalar. The next chapter focuses on compiler-based approaches, which are either VLIW or EPIC. Figure 3.61 on page 341 near the end of this chapter provides more details on a variety of recent superscalar processors.

instructions being issued in a given clock cycle, as well as among the issuing instructions and all those still in execution. If some instruction in the instruction stream is dependent (i.e., will cause a data hazard) or doesn't meet the issue criteria (i.e., will cause a structural hazard), only the instructions preceding that one in instruction sequence will be issued. In contrast, in VLIWs, the compiler has complete responsibility for creating a package of instructions that can be simultaneously issued, and the hardware does not dynamically make any decisions about multiple issue. (As we will see, for example, the Intel IA-64 architecture relies on the programmer to describe the presence of register dependences within an issue packet.) Thus, we say that a superscalar processor has *dynamic* issue capability, and a VLIW processor has *static* issue capability.

Before we look at an example, let's explore the process of instruction issue in slightly more detail. Suppose we had a four-issue, static superscalar processor. During instruction fetch the pipeline would receive from one to four instructions from the instruction fetch unit, which may not always be able to deliver four instructions. We call this group of instructions received from the fetch unit that could *potentially* issue in one clock cycle the *issue packet*. Conceptually, the instruction fetch unit examines each instruction in the issue packet in program order. If an instruction would cause a structural hazard or a data hazard either due to an earlier instruction already in execution or due to an instruction earlier in the issue packet, then the instruction is not issued. This issue limitation results in zero to four instructions from the issue packet actually being issued in a given clock cycle. Although the instruction decode and issue process *logically* proceeds in sequential order through the instructions, in practice, the issue unit examines all the instructions in the issue packet at once, checks for hazards among the in-

structions in the packet and those in the pipeline, and decides which instructions can issue.

These issue checks are sufficiently complex that performing them in one cycle could mean that the issue logic determined the minimum clock cycle length. As a result, in many statically scheduled and *all* dynamically scheduled superscalars, the issue stage is split and pipelined, so that it can issue instructions every clock cycle.

This division is not, however, totally straightforward because the processor must also detect any hazards between the two packets of instructions while they are still in the issue pipeline. One approach is to use the first stage of the issue pipeline to decide how many instructions from the packet can issue simultaneously, ignoring instructions already issued, and use the second stage to examine hazards among the selected instructions and those that have already been issued. By splitting the issue pipestage and pipelining it, the performance cost of superscalar instruction issue tends to be higher branch penalties, further increasing the importance of branch prediction.

As we increase the processor's issue rate, further pipelining of the issue stage could become necessary. Although breaking the issue stage into two stages is reasonably straightforward, it is less obvious how to pipeline it further. Thus, instruction issue is likely to be one limitation on the clock rate of superscalar processors.

A Statically Scheduled Superscalar MIPS Processor

What would the MIPS processor look like as a superscalar? For simplicity, let's assume two instructions can be issued per clock cycle and that one of the instructions can be a load, store, branch, or integer ALU operation, and the other can be any floating-point operation. Note that we consider loads and stores, including those to floating-point registers, as integer operations. As we will see, issue of an integer operation in parallel with a floating-point operation is much simpler and less demanding than arbitrary dual issue. This configuration is, in fact, very close to the organization used in the HP 7100 processor. Although high-end desktop processors now do four or more issues per clock, dual issue superscalar pipelines are becoming common at the high-end of the embedded processor market.

Issuing two instructions per cycle will require fetching and decoding 64 bits of instructions. Early superscalars often limited the placement of the instruction types; for example, the integer instruction must be first, but modern superscalars have dropped this restriction. Assuming the instruction placement is not limited, there are three steps involved in fetch and issue: fetch two instructions from the cache, determine whether zero, one, or two instructions can issue, and issue them to the correct functional unit.

Fetching two instructions is more complex than fetching one, since the instruction pair could appear anywhere in the cache block. Many processors will only fetch one instruction if the first instruction of the pair is the last word of a cache block. High-end superscalars generally rely on an independent instruction

prefetch unit, as mentioned in the previous section and described further in Chapter 5.

For this simple superscalar doing the hazard checking is relatively straightforward, since the restriction of one integer and one FP instruction eliminates most hazard possibilities within the issue packet, making it sufficient in many cases to look only at the opcodes of the instructions. The only difficulties that arise are when the integer instruction is a floating-point load, store, or move. This possibility creates contention for the floating-point register ports and may also create a new RAW hazard when the second instruction of the pair depends on the first (e.g., the first is an FP load and the second an FP operation, or the first is an FP operation and the second an FP store). This use of an issue restriction, which represents a structural hazard, to reduce the complexity of both hazard detection and pipeline structure is common in multiple issue processors. (There is also the possibility of new WAR and WAW hazards across issue packet boundaries.)

Finally, the instructions chosen for execution are dispatched to their appropriate functional units. Figure 3.24 shows how the instructions look as they go into the pipeline in pairs; for simplicity the integer instruction is always shown first, though it may be the second instruction in the issue packet.

Instruction type	**Pipe stages**							
Integer instruction	IF	ID	EX	MEM	WB			
FP instruction	IF	ID	EX	EX	EX	MEM	WB	
Integer instruction		IF	ID	EX	MEM	WB		
FP instruction		IF	ID	EX	EX	EX	MEM	WB
Integer instruction			IF	ID	EX	MEM	WB	
FP instruction			IF	ID	EX	EX	EX	MEM
Integer instruction				IF	ID	EX	MEM	WB
FP instruction				IF	ID	EX	EX	EX

FIGURE 3.24 Superscalar pipeline in operation. The integer and floating-point instructions are issued at the same time, and each executes at its own pace through the pipeline. This figure assumes that all the FP instructions are adds that take three execution cycles. This scheme will only improve the performance of programs with a large fraction of floating-point operations.

With this pipeline, we have substantially boosted the rate at which we can issue floating-point instructions. To make this worthwhile, however, we need either pipelined floating-point units or multiple independent units. Otherwise, the floating-point datapath will quickly become the bottleneck, and the advantages gained by dual issue will be small.

By issuing an integer and a floating-point operation in parallel, the need for additional hardware, beyond the enhanced hazard detection logic, is minimized—integer and floating-point operations use different register sets and dif-

ferent functional units on load-store architectures. Allowing FP loads and stores to issue with FP operations, a highly desirable capability for performance reasons, creates the need for an additional read/write port on the FP register file. In addition, because there are twice as many instructions in the pipeline, a larger set of bypass paths will be needed.

A final complication is maintaining a precise exception model. To see how imprecise exceptions can happen, consider the following:

- A floating point instruction can finish execution after an integer instruction that is later in program order (e.g., when an FP instruction is the first instruction in an issue packet and both instructions are issued).
- The floating point instruction exception could be detected after the integer instruction completed.

Left untouched, this situation would result in an imprecise exception because the integer instruction, which in program order follows the FP instruction that raised the exception, will have been completed. This situation represents a slight complication over those that can arise in a single issue pipeline when the floating point pipeline is deeper than the integer pipeline, but is no different than what we saw could arise with a dynamically scheduled pipeline. Several solutions are possible: early detection of FP exceptions (see the pipelining appendix), the use of software mechanisms to restore a precise exception state before resuming execution, and delaying instruction completion until we know an exception is impossible (the speculation approach we cover in the next section uses this approach).

Maintaining the peak throughput for this dual issue pipeline is much harder than it is for a single-issue pipeline. In our classic, five-stage pipeline, loads had a latency of one clock cycle, which prevented one instruction from using the result without stalling. In the superscalar pipeline, the result of a load instruction cannot be used on the *same* clock cycle or on the *next* clock cycle, and hence, the next three instructions cannot use the load result without stalling. The branch delay for a taken branch becomes either two or three instructions, depending on whether the branch is the first or second instruction of a pair.

To effectively exploit the parallelism available in a superscalar processor, more ambitious compiler or hardware scheduling techniques will be needed. In fact, without such techniques, a superscalar processor is likely to provide little or no additional performance.

In the next chapter, we will show how relatively simple compiler techniques suffice for a two-issue pipeline such as this one. Alternatively, we can employ an extension of Tomasulo's algorithm to schedule the pipeline, as the next section shows.

Multiple Instruction Issue with Dynamic Scheduling

Dynamic scheduling is one method for improving performance in a multiple instruction issue processor. When applied to a superscalar processor, dynamic scheduling has the traditional benefit of boosting performance in the face of data hazards, but it also allows the processor to potentially overcome the issue restrictions. Put another way, although the hardware may not be able to initiate execution of more than one integer and one FP operation in a clock cycle, dynamic scheduling can eliminate this restriction at instruction issue, at least until the hardware runs out of reservation stations.

Let's assume we want to extend Tomasulo's algorithm to support our two-issue superscalar pipeline. We do not want to issue instructions to the reservation stations out of order, since this could lead to a violation of the program semantics. To gain the full advantage of dynamic scheduling we should remove the constraint of issuing one integer and one FP instruction in a clock, but this will significantly complicate instruction issue.

Alternatively, we could use a simpler scheme: separate the data structures for the integer and floating-point registers, then we can simultaneously issue a floating-point instruction and an integer instruction to their respective reservation stations, as long as the two issued instructions do not access the same register set. Unfortunately, this approach bars issuing two instructions with a dependence in the same clock cycle, such as a floating-point load (an integer instruction) and a floating-point add. Rather than try to fix this problem, let's explore the general scheme for allowing the issue stage to handle two arbitrary instructions per clock.

Two different approaches have been used to issue multiple instructions per clock in a dynamically scheduled processor, and both rely on the observation that they key is assigning a reservation station and updating the pipeline control tables. One approach is to run this step in half a clock cycle, so that two instructions can be processed in one clock cycle. A second alternative is to build the logic necessary to handle two instructions at once, including any possible dependences between the instructions. Modern superscalar processors that issue four or more instructions per clock often include both approaches: they both pipeline and widen the issue logic.

There is one final issue to discuss before we look at an example: how should dynamic branch prediction be integrated into a dynamically scheduled pipeline. The IBM 360/91 used a simple static prediction scheme, but only allowed instructions to be fetched and issued (but not actually executed) until the branch had completed. In this section, we follow the same approach. In the next section, we will examine speculation, which takes this a step further and actually executes instructions based on branch predictions.

Assume that we have the most general implementation of a two issue dynamically scheduled processor, meaning that it can issue any pair of instructions if there are reservation stations of the right type available. Because the interaction of the integer and floating point instructions is crucial, we also extend Tomasulo's

scheme to deal with both the integer and floating point functional units and registers. Let's see how a simple loop executes on this processor.

EXAMPLE Consider the execution of the following simple loop, which adds a scalar in F2 to each element of a vector in memory. Use a MIPS pipeline extended with Tomasulo's algorithm and with multiple issue:

```
Loop:  L.D     F0,0(R1)       ; F0=array element
       ADD.D   F4,F0,F2       ; add scalar in F2
       S.D     F4,0(R1)       ; store result
       DADDIU  R1,R1,#-8      ; decrement pointer
                              ; 8 bytes (per DW)
       BNE     R1,R2,LOOP     ; branch R1!=zero
```

Assume that both a floating-point and an integer operation can be issued on every clock cycle, even if they are dependent. Assume one integer functional unit used for both ALU operations and effective address calculations and a separate pipelined FP functional unit for each operation type. Assume that issue and write results take one cycle each and that there is dynamic branch-prediction hardware and a separate functional unit to evaluate branch conditions. As in most dynamically scheduled processors, the presence of the write results stage means that the effective instruction latencies will be one cycle longer than in a simple in-order pipeline. Thus, the number of cycles of latency between a source instruction and an instruction consuming the result is one cycle for integer ALU operations, two cycles for loads, and three cycles for FP add. Create a table showing when each instruction issues, begins execution, and writes its result to the CDB for the first three iterations of the loop. Assume two CDBs and assume that branches single issue (no delayed branches) but that branch prediction is perfect. Also show the resource usage for the integer unit, the floating point unit, the data cache, and the two CDBs.

ANSWER The loop will be dynamically unwound and, whenever possible, instructions will be issued in pairs. The execution timing is shown in Figure 3.25 and Figure 3.26 shows the resource utilization. The loop will continue to fetch and issue a new loop iteration every three clock cycles and sustaining one iteration every three cycles would lead to an IPC of 5/3 = 1.67. The instruction execution rate, however, is lower: by looking at the execute stage we can see that the sustained instruction completion rate is 15/16 = 0.94. Assuming the branches are perfectly predicted, the issue unit will eventually fill all the reservation stations and will stall.

n

Iter. #	Instructions	Issues at	Executes	Memory access at	Write CDB at	Comment
1	L.D F0,0(R1)	1	2	3	4	First issue
1	ADD.D F4,F0,F2	1	5		8	Wait for L.D
1	S.D F4,0(R1)	2	3	9		Wait for ADD.D
1	DADDIU R1,R1,#-8	2	4		5	Wait for ALU
1	BNE R1,R2,Loop	3	6			Wait for DADDIU
2	L.D F0,0(R1)	4	7	8	9	Wait for BNE complete
2	ADD.D F4,F0,F2	4	10		13	Wait for L.D
2	S.D F4,0(R1)	5	8	14		Wait for ADD.D
2	DADDIU R1,R1,#-8	5	9		10	Wait for ALU
2	BNE R1,R2,Loop	6	11			Wait for DADDIU
3	L.D F0,0(R1)	7	12	13	14	Wait for BNE complete
3	ADD.D F4,F0,F2	7	15		18	Wait for L.D
3	S.D F4,0(R1)	8	13	19		Wait for ADD.D
3	DAADIU R1,R1,#-8	8	14		15	Wait for ALU
3	BNE R1,R2,Loop	9	16			Wait for DADDIU

FIGURE 3.25 The clock cycle of issue, execution, and writing result for a dual-issue version of our Tomasulo pipeline. The write-result stage does not apply to either stores or branches, since they do not write any registers. We assume a result is written to the CDB at the end of the clock cycle it is available in. This figure also assumes a wider CDB. For L.D and S.D, the execution is effective address calculation. For branches, the execute cycle shows when the branch condition can be evaluated and the prediction checked; we assume that this can happen as early as the cycle after issue, if the operands are available. Any instructions following a branch cannot start execution until after the branch condition has been evaluated. We assume one memory unit, one integer pipeline, and one FP adder.If two instructions could use the same functional unit at the same point, priority is given to the "older" instruction. Note that the load of the next iteration performs its memory access before the store of the current iteration.

The throughput improvement versus a single issue pipeline is small because there is only one floating-point operation per iteration and, thus, the integer pipeline becomes a bottleneck. The performance could be enhanced by compiler techniques we will discuss in the next chapter. Alternatively, if the processor could execute more integer operations per cycle, larger improvements would be possible. A revised example demonstrates this potential improvement and the flexibility of dynamic scheduling to adapt to different hardware capabilities.

EXAMPLE Consider the execution of the same loop on two-issue processor, but, in addition, assume that there are separate integer functional units for effective address calculation and for ALU operations. Create a table as in Figure 3.25 for the first three iterations of the same loop and another table

Clock #	Integer ALU	FP ALU	Data Cache	CDB
2	1/ L.D			
3	1 / S.D		1/ L.D	
4	1 / DADDIU			1/ L.D
5		1 / ADD.D		1 / DADDIU
6				
7	2 / L.D			
8	2 / S.D		2 / L.D	1 / ADD.D
9	2 / DADDIU		1 / S.D	2 / L.D
10		2 / ADD.D		2 / DADDIU
11				
12	3 / L.D			
13	3 / S.D		3 / L.D	2 / ADD.D
14	3 / DADDIU		2 / S.D	3 / L.D
15		3 / ADD.D		3 / DADDIU
16				
17				
18				3 / ADD.D
19			3 / S.D	
20				

FIGURE 3.26 Resource usage table for the example shown in Figure 3.25. The entry in each box shows the opcode and iteration number of the instruction that uses the functional unit heading the column at the clock cycle corresponding to the row. Only a single CDB is actually required and that is what we show.

to show the resource usage.

ANSWER Figure 3.27 shows the improvement in performance: the loop executes in five clock cycles less (11 versus 16 execution cycles). The cost of this improvement is both a separate address adder and the logic to issue to it; note that, in contrast to the earlier example, a second CDB is needed. As Figure 3.28 shows this example has a higher instruction execution rate but lower efficiency as measured by the utilization of the functional units.

n

Three factors limit the performance (as shown in Figure 3.27) of the two-issue dynamically scheduled pipeline:

Iter. #	Instructions	Issues at	Executes	Memory access at	Write CDB at	Comment
1	L.D F0,0(R1)	1	2	3	4	First issue
1	ADD.D F4,F0,F2	1	5		8	Wait for L.D
1	S.D F4,0(R1)	2	3	9		Wait for ADD.D
1	DADDIU R1,R1,#-8	2	3		4	Executes earlier
1	BNE R1,R2,Loop	3	5			Wait for DADDIU
2	L.D F0,0(R1)	4	6	7	8	Wait for BNE complete
2	ADD.D F4,F0,F2	4	9		12	Wait for L.D
2	S.D F4,0(R1)	5	7	13		Wait for ADD.D
2	DADDIU R1,R1,#-8	5	6		7	Executes earlier
2	BNE R1,Loop	6	8			Wait for DADDIU
3	L.D F0,0(R1)	7	9	10	11	Wait for BNE complete
3	ADD.D F4,F0,F2	7	12		15	Wait for L.D
3	S.D F4,0(R1)	8	10	16		Wait for ADD.D
3	DADDIU R1,R1,#-8	8	9		10	Executes earlier
3	BNE R1,Loop	9	11			Wait for DADDIU

FIGURE 3.27 The clock cycle of issue, execution, and writing result for a dual-issue version of our Tomasulo pipeline with separate functional units for integer ALU operations and effective address calculation, which also uses a wider CDB. The extra integer ALU allows the DADDIU to execute earlier, in turn allowing the BNE to execute earlier, and, thereby, starting the next iteration earlier.

Clock #	Integer ALU	Address Adder	FP ALU	Data Cache	CDB #1	CDB #2
2		1/ L.D				
3	1 / DADDIU	1 / S.D		1/ L.D		
4					1/ L.D	1 / DADDIU
5			1 / ADD.D			
6	2 / DADDIU	2 / L.D				
7		2 / S.D		2 / L.D	2 / DADDIU	
8					1 / ADD.D	2 / L.D
9	3 / DADDIU	3 / L.D	2 / ADD.D	1 / S.D		
10		3 / S.D		3 / L.D	3 / DADDIU	
11					3 / L.D	
12			3 / ADD.D		2 / ADD.D	
13				2 / S.D		
14						
15						3 / ADD.D
16				3 / S.D		

FIGURE 3.28 Resource usage table for the example shown in Figure 3.27, using the same format as Figure 3.26.

1. There is an imbalance between the functional unit structure of the pipeline and the example loop. This imbalance means that it is impossible to fully use the FP units. To remedy this, we would need fewer dependent integer operations per loop. The next point is a different way of looking at this limitation.
2. The amount of overhead per loop iteration is very high: two of out of five instructions (the `DADDIU` and the `BNE`) are overhead. In the next chapter we look at how this overhead can be reduced.
3. The control hazard, which prevents us from starting the next `L.D` before we know whether the branch was correctly predicted, causes a one-cycle penalty on every loop iteration. The next section introduces a technique that addresses this limitation.

3.7 Hardware-Based Speculation

As we try to exploit more instruction level parallelism, maintaining control dependences becomes an increasing burden. Branch prediction reduces the direct stalls attributable to branches, but for a processor executing multiple instructions per clock, just predicting branches accurately may not be sufficient to generate the desired amount of instruction level parallelism. A wide issue processor may need to execute a branch every clock cycle to maintain maximum performance. Hence, exploiting more parallelism requires that we overcome the limitation of control dependence. The performance of the pipeline in Figure 3.25 makes this clear: there is one stall cycle each loop iteration due to a branch hazard. In programs with more branches and more data dependent branches, this penalty could be larger.

Overcoming control dependence is done by speculating on the outcome of branches and *executing* the program as if our guesses were correct. This mechanism represents a subtle, but important, extension over branch prediction with dynamic scheduling. In particular, with speculation, we fetch, issue, and *execute* instructions, as if our branch predictions were always correct; dynamic scheduling only fetches and issues such instructions. Of course, we need mechanisms to handle the situation where the speculation is incorrect. The next chapter discusses a variety of mechanisms for supporting speculation by the compiler. In this section, we explore *hardware speculation*, which extends the ideas of dynamic scheduling.

Hardware-based speculation combines three key ideas: dynamic branch prediction to choose which instructions to execute, speculation to allow the execution of instructions before the control dependences are resolved (with the ability to undo the effects of an incorrectly speculated sequence), and dynamic scheduling to deal with the scheduling of different combinations of basic blocks. (In comparison, dynamic scheduling without speculation only partially overlaps basic blocks, because it requires that a branch be resolved before actually executing any instructions in the successor basic block.) Hardware-based speculation fol-

lows the predicted flow of data values to choose when to execute instructions. This method of executing programs is essentially a *data-flow execution*: operations execute as soon as their operands are available.

The approach we examine here, and the one implemented in a number of processors (PowerPC 603/604/G3/G4, MIPS R10000/R12000, Intel Pentium II/III/4, Alpha 21264, and AMD K5/K6/Athlon), is to implement speculative execution based on Tomasulo's algorithm. Just as with Tomasulo's algorithm, we explain hardware speculation in the context of the floating-point unit, but the ideas are easily applicable to the integer unit.

The hardware that implements Tomasulo's algorithm can be extended to support speculation. To do so, we must separate the bypassing of results among instructions, which is needed to execute an instruction speculatively, from the actual completion of an instruction. By making this separation, we can allow an instruction to execute and to bypass its results to other instructions, without allowing the instruction to perform any updates that cannot be undone, until we know that the instruction is no longer speculative. Using the bypassed value is like performing a speculative register read, since we do not know whether the instruction providing the source register value is providing the correct result until the instruction is no longer speculative. When an instruction is no longer speculative, we allow it to update the register file or memory; we call this additional step in the instruction execution sequence *instruction commit*.

The key idea behind implementing speculation is to allow instructions to execute out of order but to force them to commit *in order* and to prevent any irrevocable action (such as updating state or taking an exception) until an instruction commits. In the simple single-issue five-stage pipeline we could ensure that instructions committed in order, and only after any exceptions for that instruction had been detected, simply by moving writes to the end of the pipeline. When we add speculation, we need to separate the process of completing execution from instruction commit, since instructions may finish execution considerably before they are ready to commit. Adding this commit phase to the instruction execution sequence requires some changes to the sequence as well as an additional set of hardware buffers that hold the results of instructions that have finished execution but have not committed. This hardware buffer, which we call the *reorder buffer*, is also used to pass results among instructions that may be speculated.

The reorder buffer (ROB, for short) provides additional registers in the same way as the reservation stations in Tomasulo's algorithm extend the register set. The ROB holds the result of an instruction between the time the operation associated with the instruction completes and the time the instruction commits. Hence, the ROB is a source of operands for instructions, just as the reservation stations provide operands in Tomasulo's algorithm. The key difference is that in Tomasulo's algorithm, once an instruction writes its result, any subsequently issued instructions will find the result in the register file. With speculation, the register file is not updated until the instruction commits (and we know definitively that the instruction should execute); thus, the ROB supplies operands in the

interval between completion of instruction execution and instruction commit. The ROB is similar the store buffer in Tomasulo's algorithm, and we integrate the function of the store buffer into the ROB for simplicity.

Each entry in the ROB contains three fields: the instruction type, the destination field, and the value field. The instruction-type field indicates whether the instruction is a branch (and has no destination result), a store (which has a memory address destination), or a register operation (ALU operation or load, which have register destinations). The destination field supplies the register number (for loads and ALU operations) or the memory address (for stores), where the instruction result should be written. The value field is used to hold the value of the instruction result until the instruction commits. We will see an example of ROB entries shortly.

Figure 3.29 shows the hardware structure of the processor including the ROB. The ROB completely replaces the store buffers. Stores still execute in two steps, but the second step is performed by instruction commit. Although the renaming function of the reservation stations is replaced by the ROB, we still need a place to buffer operations (and operands) between the time they issue and the time they begin execution. This function is still provided by the reservation stations. Since every instruction has a position in the ROB until it commits, we tag a result using the ROB entry number rather than using the reservation station number. This tagging requires that the ROB assigned for an instruction must be tracked in the reservation station. Later in this section, we will explore an alternative implementation that uses extra registers for renaming and the ROB only to track when instructions can commit.

Here are the four steps involved in instruction execution:

1. *Issue*—Get an instruction from the instruction queue. Issue the instruction if there is an empty reservation station and an empty slot in the ROB, send the operands to the reservation station if they available in either the registers or the ROB. Update the control entries to indicate the buffers are in use. The number of the ROB allocated for the result is also sent to the reservation station, so that the number can be used to tag the result when it is placed on the CDB. If either all reservations are full or the ROB is full, then instruction issue is stalled until both have available entries. This stage is sometimes called *dispatch* in a dynamically scheduled processor.

2. *Execute*—If one or more of the operands is not yet available, monitor the CDB (common data bus) while waiting for the register to be computed. This step checks for RAW hazards. When both operands are available at a reservation station, execute the operation. (Some dynamically scheduled processors call this step *issue*, but we use the name execute, which was used in the first dynamically scheduled processor, the CDC 6600.) Instructions may take multiple clock cycles in this stage, and loads still require two steps in this stage. Stores need only have the base register available at this step, since execution

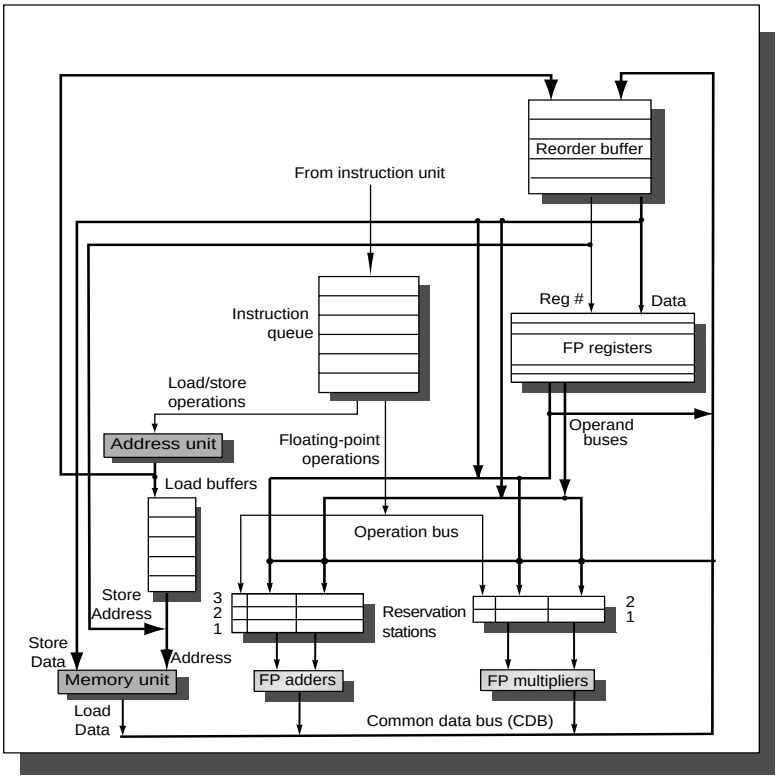


FIGURE 3.29 The basic structure of a MIPS FP unit using Tomasulo's algorithm and extended to handle speculation. Comparing this to Figure 3.2 on page 237, which implemented Tomasulo's algorithm, the major change is the addition of the ROB and the elimination of the store buffer, whose function is integrated into the ROB. This mechanism can be extended to multiple issue by making the CDB (common data bus) wider to allow for multiple completions per clock.

for a store at this point is only effective address calculation.

3. *Write result*—When the result is available, write it on the CDB (with the ROB tag sent when the instruction issued) and from the CDB into the ROB, as well as to any reservation stations waiting for this result. Mark the reservation station as available. Special actions are required for store instructions. If the value to be stored is available, it is written into the Value field of the ROB entry for the store. If the value to be stored is not available yet, the CDB must be monitored until that value is broadcast, at which time the Value field of the ROB entry of the store is updated. For simplicity in our description, we assume that this occurs during the Write Results stage of a store; we discuss relaxing this requirement later.

4. *Commit*—There are three different sequences of actions at commit depending on whether the committing instruction is: a branch with an incorrect prediction, a store, or any other instruction (normal commit). The normal commit case occurs when an instruction reaches the head of the ROB and its result is present in the buffer; at this point, the processor updates the register with the result and removes the instruction from the ROB. Committing a store is similar except that memory is updated rather than a result register. When a branch with incorrect prediction reaches the head of the ROB, it indicates that the speculation was wrong. The ROB is flushed and execution is restarted at the correct successor of the branch. If the branch was correctly predicted, the branch is finished. Some machines call this commit phase *completion* or *graduation.*

Once an instruction commits, its entry in the ROB is reclaimed and the register or memory destination is updated, eliminating the need for the ROB entry. If the ROB fills, we simply stop issuing instructions until an entry is made free. Now, let's examine how this scheme would work with the same example we used for Tomasulo's algorithm.

EXAMPLE Assume the same latencies for the floating-point functional units as in earlier examples: add is 2 clock cycles, multiply is 10 clock cycles, and divide is 40 clock cycles. Using the code segment below, the same one we used to generate Figure 3.4 on page 242, show what the status tables look like when the `MUL.D` is ready to go to commit.

```
L.D     F6,34(R2)
L.D     F2,45(R3)
MUL.D   F0,F2,F4
SUB.D   F8,F6,F2
DIV.D   F10,F0,F6
ADD.D   F6,F8,F2
```

ANSWER The result is shown in the three tables in Figure 3.30. Notice that although the `SUB.D` instruction has completed execution, it does not commit until the `MUL.D` commits. The reservation stations and register status field contain the same basic information that they did for Tomasulo's algorithm (see page 238 for a description of those fields). The differences are that reservation station numbers are replaced with ROB entry numbers in the Qj and Qk fields, as well as in the register status fields, and we have added the Dest field to the reservation stations. The Dest field designates the ROB number that is the destination for the result produced by this reservation station entry. n

The above Example illustrates the key important difference between a processor with speculation and a processor with dynamic scheduling. Compare the content of Figure 3.30 with that of Figure 3.4 (page 242), which shows the same code sequence in operation on a processor with Tomasulo's algorithm. The key difference is that in the example above, no instruction after the earliest uncompleted instruction (`MUL.D` above) is allowed to complete. In contrast, in Figure 3.4 the `SUB.D` and `ADD.D` instructions have also completed.

	Reservation stations							
Name	**Busy**	**Op**	**Vj**	**Vk**	**Qj**	**Qk**	**Dest**	**A**
Load1	no							
Load2	no							
Add1	no							
Add2	no							
Add3	no							
Mult1	no	MUL.D	Mem[45+Regs[R3]]	Regs[F4]			#3	
Mult2	yes	DIV.D		Mem[34+Regs[R2]]	#3		#5	

	Reorder buffer					
Entry	**Busy**	**Instruction**		**State**	**Destination**	**Value**
1	no	L.D	F6,34(R2)	Commit	F6	Mem[34+Regs[R2]]
2	no	L.D	F2,45(R3)	Commit	F2	Mem[45+Regs[R3]]
3	yes	MUL.D	F0,F2,F4	Write result	F0	#2 x Regs[F4]
4	yes	SUB.D	F8,F6,F2	Write result	F8	#1 – #2
5	yes	DIV.D	F10,F0,F6	Execute	F10	
6	yes	ADD.D	F6,F8,F2	Write result	F6	#4 + #2

	FP register status									
Field	**F0**	**F1**	**F2**	**F3**	**F4**	**F5**	**F6**	**F7**	**F8**	**F10**
Reorder #	3						6		4	5
Busy	yes	no	no	no	no	no	yes	...	yes	yes

FIGURE 3.30 At the time the MUL.D is ready to commit, only the two L.D instructions have committed, though several others have completed execution. The MUL.D is at the head of the ROB, and the two L.D instructions are there only to ease understanding. The SUB.D and ADD.D instructions will not commit until the MUL.D instruction commits, though the results of the instructions are available and can be used as sources for other instructions.The DIV.D is in execution, but has not completed solely due to its longer latency than MUL.D. The value column indicates the value being held, the format #X is used to refer to a value field of ROB entry X. Reorder buffers 1 and 2 are actually completed, but are shown for informational purposes. we do not show the entries for the load/store queue, but these entries are kept in order.

One implication of this difference is that the processor with the ROB can dynamically execute code while maintaining a precise interrupt model. For example, if the MUL.D instruction caused an interrupt, we could simply wait until it reached the head of the ROB and take the interrupt, flushing any other pending instructions. Because instruction commit happens in order, this yields a precise

exception. By contrast, in the example using Tomasulo's algorithm, the SUB.D and ADD.D instructions could both complete before the MUL.D raised the exception. The result is that the registers F8 and F6 (destinations of the SUB.D and ADD.D instructions) could be overwritten, and the interrupt would be imprecise. Some users and architects have decided that imprecise floating-point exceptions are acceptable in high-performance processors, since the program will likely terminate; see Appendix A for further discussion of this topic. Other types of exceptions, such as page faults, are much more difficult to accommodate if they are imprecise, since the program must transparently resume execution after handling such an exception. The use of a ROB with in-order instruction commit provides precise exceptions, in addition to supporting speculative execution, as the next Example shows.

EXAMPLE Consider the code example used earlier for Tomasulo's algorithm and shown in Figure 3.6 on page 245 in execution:

```
Loop:  L.D    F0,0(R1)
       MUL.D  F4,F0,F2
       S.D    F4,0(R1)
       DADDIU R1,R1,#-8
       BNE    R1,R2,Loop     ; branches if R1≠0
```

Assume that we have issued all the instructions in the loop twice. Let's also assume that the L.D and MUL.D from the first iteration have committed and all other instructions have completed execution.Normally, the store would wait in the ROB for both the effective address operand (R1 in this example) and the value (F4 in this example). Since we are only considering the floating-point pipeline, assume the effective address for the store is computed by the time the instruction is issued.

ANSWER The result is shown in the two tables in Figure 3.31.

n

Because neither the register values nor any memory values are actually written until an instruction commits, the processor can easily undo its speculative actions when a branch is found to be mispredicted. Suppose that in the above example (see Figure 3.31), the branch BNE is not taken the first time. The instructions prior to the branch will simply commit when each reaches the head of the ROB; when the branch reaches the head of that buffer, the buffer is simply cleared and the processor begins fetching instructions from the other path.

	Reorder buffer				
Entry	**Busy**	**Instruction**	**State**	**Destination**	**Value**
1	no	`L.D F0,0(R1)`	Commit	F0	Mem[0+Regs[R1]]
2	no	`MUL.D F4,F0,F2`	Commit	F4	#1 x Regs[F2]
3	yes	`S.D F4,0(R1)`	Write result	0+Regs[R1]	#2
4	yes	`DADDIU R1,R1,#-8`	Write result	R1	Regs[R1]–8
5	yes	`BNE R1,R2,Loop`	Write result		
6	yes	`L.D F0,0(R1)`	Write result	F0	Mem[#4]
7	yes	`MUL.D F4,F0,F2`	Write result	F4	#6 x Regs[F2]
8	yes	`S.D F4,0(R1)`	Write result	0+#4	#7
9	yes	`DADDIU R1,R1,#-8`	Write result	R1	#4 – 8
10	yes	`BNE R1,R2,Loop`	Write result		

	FP register status								
Field	**F0**	**F1**	**F2**	**F3**	**F4**	**F5**	**F6**	**F7**	**F8**
Reorder #	6				7				
Busy	yes	no	no	no	yes	no	no	...	no

FIGURE 3.31 Only the L.D and MUL.D instructions have committed, though all the others have completed execution. Hence, no reservation stations are busy and none are shown. The remaining instructions will be committed as fast as possible. The first two reorder buffers are empty, but are shown for completeness.

In practice, machines that speculate try to recover as early as possible after a branch is mispredicted. This recovery can be done by clearing the ROB for all entries that appear after the mispredicted branch, allowing those that are before the branch in the ROB to continue, and restarting the fetch at the correct branch successor. In speculative processors, however, performance is more sensitive to the branch prediction mechanisms, since the impact of a misprediction will be higher. Thus, all the aspects of handling branches—prediction accuracy, misprediction detection, and misprediction recovery—increase in importance.

Exceptions are handled by not recognizing the exception until it is ready to commit. If a speculated instruction raises an exception, the exception is recorded in the ROB. If a branch misprediction arises and the instruction should not have been executed, the exception is flushed along with the instruction when the ROB is cleared. If the instruction reaches the head of the ROB, then we know it is no longer speculative and the exception should really be taken. We can also try to handle exceptions as soon as they arise and all earlier branches are resolved, but this is more challenging in the case of exceptions than for branch mispredict and, because it occurs less frequently, not as critical.

Figure 3.32 shows the steps of execution for an instruction, as well as the conditions that must be satisfied to proceed to the step and the actions taken. We show the case where mispredicted branches are not resolved until commit. Although speculation seems like a simple addition to dynamic scheduling, a comparison of Figure 3.32 with the comparable figure for Tomasulo's algorithm (see Figure 3.5 on page 243) shows that speculation adds significant complications to the control. In addition, remember that branch mispredictions are somewhat more complex as well.

There is an important difference in how stores are handled in a speculative processor, versus in Tomasulo's algorithm. In Tomasulo's algorithm, a store can update memory when it reaches Write Results (which ensures that the effective address has been calculated) and the data value to store is available. In a speculative processor, a store updates memory only when it reaches the head of the ROB.This difference ensures that memory is not updated until an instruction is no longer speculative.

Figure 3.32 has one significant simplification for stores, which is unneeded in practice. Figure 3.32 requires stores to wait in the write result stage for the register source operand whose value is to be stored; the value is then moved from the Vk field of the store's reservation station to the Value field of the store's ROB entry. In reality, however, the value to be stored need not arrive until *just before* the store commits and can be placed directly into the store's ROB entry by the sourcing instruction. This is accomplished by having the hardware track when the source value to be stored is available in the store's ROB entry and searching the ROB on every instruction completion to look for dependent stores. This addition is not complicated but adding it has two effects: we would need to add a field to the ROB and Figure 3.32, which is already in a small font, would no longer fit on one page! Although Figure 3.32 makes this simplification, in our examples, we will allow the store to pass through the write-results stage and simply wait for the value to be ready when it commits.

Like Tomasulo's algorithm, we must avoid hazards through memory. WAW and WAR hazards through memory are eliminated with speculation, because the actual updating of memory occurs in order, when a store is at the head of the ROB, and hence, no earlier loads or stores can still be pending. RAW hazards through memory are maintained by two restrictions:

1. not allowing a load to initiate the second step of its execution if any active ROB entry occupied by a store has an Destination field that matches the value of the A field of the load, and
2. maintaining the program order for the computation of an effective address of a load with respect to all earlier stores.

Together, these two restrictions ensure that any load that accesses a memory location written to by an earlier store, cannot perform the memory access until the

Status	Wait until	Action or bookkeeping
Issue All instructions	Reservation station (r) and ROB (b) both available	`if (RegisterStat[rs].Busy)/*in-flight instr. writes rs*/ {h← RegisterStat[rs].Reorder; if (ROB[h].Ready)/* Instr completed already */ {RS[r].Vj← ROB[h].Value; RS[r].Qj ← 0;} else {RS[r].Qj← h;} /* wait for instruction */ } else {RS[r].Vj← Regs[rs]; RS[r].Qj← 0;}; RS[r].Busy← yes; RS[r].Dest← b; ROB[h].Instruction ← opcode; ROB[b].Ready← no;`
FP Operations and Stores		`if (RegisterStat[rt].Busy) /*in-flight instr writes rt*/ {h← RegisterStat[rt].Reorder; if (ROB[b].Ready) /* Instr completed already */ {RS[r].Vk← ROB[h].Value; RS[r].Qk ← 0;} else {RS[r].Qk← h;} /* Wait for instruction */ } else {RS[r].Vk← Regs[rt]; RS[r].Qk← 0;};`
FP Operations		`RegisterStat[rd].Qi=b; RegisterStat[rd].Busy← yes; ROB[b].Dest← rd;`
Loads		`RS[r].A← imm; RegisterStat[rt].Qi=b; RegisterStat[rt].Busy← yes; ROB[b].Dest← rt;`
Stores		`RS[r].A← imm;`
Execute FP Op	(RS[r].Qj=0) and (RS[r].Qk=0)	Compute results—operands are in Vj and Vk
Load step1	(RS[r].Qj=0) & there are no stores earlier in the queue	`RS[r].A←RS[r].Vj + RS[r].A;`
Load step 2	Load step 1 done & all stores earlier in ROB have different address	Read from Mem[RS[r].A]
Store	(RS[r].Qj=0) & store at queue head	`ROB[h].Address←RS[r].Vj + RS[r].A;`
Write result All but store	Execution done at *r* & CDB available.	`b←RS[r].Reorder; RS[r].Busy← no; ∀x(if (RS[x].Qj=b) {RS[x].Vj← result; RS[x].Qj ← 0}); ∀x(if (RS[x].Qk=b) {RS[x].Vk← result; RS[x].Qk ← 0}); ROB[b].Value←result; ROB[b].Ready←yes;`
Store	Execution done at *r* & (RS[r].Qk=0)	`ROB[h].Value←RS[r].Vk;`

FIGURE 3.32 Steps in the algorithm and what is required for each step. For the issuing instruction, `rd` is the destination, `rs` and `rt` are the sources, and `r` is the reservation station allocated and `b` is the assigned ROB entry. `RS` is the reservation-station data structure. The value returned by a reservation station is called the `result`. `RegisterStat` is the register data structure, `Regs` represents the actual registers, and `ROB` is the reorder buffer data structure.

Commit	Instruction is at the head of the ROB (entry h) and ROB[h].ready = yes	`r = ROB[h].Dest; /* register dest, if exists */`
		`if (ROB[h].Instruction==Branch)`
		`{if (branch is mispredicted)`
		`{clear ROB[h], RegisterStat; fetch branch dest;};}`
		`else if (ROB[h].Instruction==Store)`
		`{Mem[ROB[h].Address]← ROB[h].Value;}`
		`else /* put the result in the register destination */`
		`{Regs[r]← ROB[h].Value;};`
		`ROB[h].Busy← no; /* free up ROB entry */`
		`/* free up dest register if no one else writing it */`
		`if (RegisterStat[r].Qi==h) {RegisterStat[r].Busy← no;};`

FIGURE 3.32 Steps in the algorithm and what is required for each step. For the issuing instruction, `rd` is the destination, `rs` and `rt` are the sources, and `r` is the reservation station allocated and `b` is the assigned ROB entry. `RS` is the reservation-station data structure. The value returned by a reservation station is called the `result`. `RegisterStat` is the register data structure, `Regs` represents the actual registers, and `ROB` is the reorder buffer data structure.

store has written the data. Some speculative machines will actually bypass the value from the store to the load directly, when such a RAW hazard occurs.

Although this explanation of speculative execution has focused on floating point, the techniques easily extend to the integer registers and functional units, as we will see in the Putting It All Together section. Indeed, speculation may be more useful in integer programs, since such programs tend to have code where the branch behavior is less predictable. Additionally, these techniques can be extended to work in a multiple-issue processor by allowing multiple instructions to issue and commit every clock. In fact, speculation is probably most interesting in such processors, since less ambitious techniques can probably exploit sufficient ILP within basic blocks when assisted by a compiler.

Multiple Issue with Speculation

A speculative processor can be extended to multiple issue using the same techniques we employed when extending a Tomasulo-based processor in section 3.6. The same techniques for implementing the instruction issue unit can be used: We process multiple instructions per clock assigning reservation stations and reorder buffers to the instructions.

The two challenges of multiple issue with Tomasulo's algorithm--instruction issue and monitoring the CDBs for instruction completion--become the major challenges for multiple issue with speculation. In addition, to maintain throughput of greater than one instruction per cycle, a speculative processor must be able to handle multiple instruction commits per clock cycle. To show how speculation can improve performance in a multiple issue processor consider the following example using speculation.

EXAMPLE Consider the execution of the following loop, which searches an array, on a two issue processor one with dynamic scheduling and one with specu-

lation:

```
Loop:  LW     R2,0(R1);R2=array element
       DADDIU R2,R2,#1; increment R2
       SW     0(R1),R2;store result
       DADDIU R1,R1,#4;increment pointer
       BNE    R2,R3,LOOP; branch if last element!=0
```

Assume that there are separate integer functional units for effective address calculation, for ALU operations, and for branch condition evaluation. Create a table as in Figure 3.27 for the first three iterations of this loop for both machines. Assume that up to two instructions of any type can commit per clock.

ANSWER Figure 3.33 and 3.34 show the performance for a two issue dynamically scheduled processor, without and with speculation. In this case, where a branch is a key potential performance limitation, speculation helps significantly. The third branch in the speculative processor executes in clock cycle 11, while it executes in clock cycle 19 on nonspeculative pipeline. Because the completion rate on the nonspeculative pipeline is falling behind the issue rate rapidly, the nonspeculative pipeline will stall when a few more iterations are issued. The performance of the nonspeculative processor could be improved by allowing load instructions to complete effective address calculation before a branch is decided, but unless speculative memory accesses are allowed, this improvement will gain only one clock per iteration.

The above example clearly shows how speculation can be advantageous when there are data dependent branches, which otherwise would limit performance. This advantage depends, however, on accurate branch prediction. Incorrect speculation will not improve performance, but will, in fact, typically harm performance.

Design Considerations for Speculative Machines

In this section we briefly examine a number of important considerations that arise in speculative machines.

Register renaming versus Reorder Buffers

One alternative to the use of a ROB is the explicit use of a larger physical set of registers combined with register renaming. This approach builds on the concept of renaming used in Tomasulo's algorithm, but extends it. In Tomasulo's algorithm, the values of the *architecturally visible registers* (R0,..., R31 and

Iter. #	Instructions	Issues at clock cycle #	Executes at clock cycle #	Memory access at clock cycle #	Write CDB at clock cycle #	Comment
1	LW R2,0(R1)	1	2	3	4	First issue
1	DADDIU R2,R2,#1	1	5		6	Wait for LW
1	SW 0(R1),R2	2	3	7		Wait for DADDIU
1	DADDIU R1,R1,#4	2	3		4	Execute directly
1	BNE R2,R3,LOOP	3	7			Wait for DADDIU
2	LW R2,0(R1)	4	8	9	10	Wait for BNE
2	DADDIU R2,R2,#1	4	11		12	Wait for LW
2	SW 0(R1),R2	5	9	13		Wait for DADDIU
2	DADDIU R1,R1,#4	5	8		9	Wait for BNE
2	BNE R2,R3,LOOP	6	13			Wait for DADDIU
3	LW R2,0(R1)	7	14	15	16	Wait for BNE
3	DADDIU R2,R2,#1	7	17		18	Wait for LW
3	SW 0(R1),R2	8	19	20		Wait for DADDIU
3	DADDIU R1,R1,#4	8	14		15	Wait for BNE
3	BNZ R2,R3,LOOP	9	19			Wait for DADDIU

FIGURE 3.33 The time of issue, execution, and writing result for a dual-issue version of our pipeline *without* speculation. Note that the L.D following the BNE cannot start execution earlier, because it must wait until the branch outcome is determined. This type of program with data dependent branches that cannot be resolved earlier, shows the strength of speculation. Separate functional units for address calculation, ALU operations, and branch condition evaluation allow multiple instructions to execute in the same cycle.

F0,...,F31) are contained, at any point in execution, in some combination of the register set and the reservation stations. With the addition of speculation, register values may also temporarily reside in the ROB. In either case, if the processor does not issue new instructions for a period of time, all existing instructions will commit, and the register values will appear in the register file, which directly corresponds to the architecturally visible registers.

In the register renaming approach, an extended set of physical registers is used to hold both the architecturally visible registers as well as temporary values. Thus, the extended registers replace the function both of the ROB and the reservation stations. During instruction issue, a renaming process maps the names of architectural registers to physical register numbers in the extended register set, allocating a new unused register for the destination. WAW and WAR hazards are avoided by renaming of the destination register, and speculation recovery is handled because a physical register holding an instruction destination does not become the architectural register until the instruction commits. The renaming map is a simple data structure that supplies the physical register number of the register that currently corresponds to the specified architectural register. This structure is similar is structure and function to the register status table in Tomasulo's algorithm,

Iter. #	Instructions	Issues at clock #	Executes at clock #	Read access at clock #	Write CDB at clock #	Commits at clock #	Comment
1	LW R2,0(R1)	1	2	3	4	5	First issue
1	DADDIU R2,R2,#1	1	5		6	7	Wait for LW
1	SW 0(R1),R2	2	3			7	Wait for DADDIU
1	DADDIU R1,R1,#4	2	3		4	8	Commit in order
1	BNE R2,R3,LOOP	3	7			8	Wait for ADDDI
2	LW R2,0(R1)	4	5	6	7	9	No execute delay
2	DADDIU R2,R2,#1	4	8		9	10	Wait for LW
2	SW 0(R1),R2	5	6			10	Wait for DADDIU
2	DADDIU R1,R1,#4	5	6		7	11	Commit in order
2	BNE R2,R3,LOOP	6	10			11	Wait for DADDIU
3	LW R2,0(R1)	7	8	9	10	12	Earliest possible
3	DADDIU R2,R2,#1	7	11		12	13	Wait for LW
3	SW 0(R1),R2	8	9			13	Wait for DADDIU
3	DADDIU R1,R1,#4	8	9		10	14	Executes earlier
3	BNE R2,R3,LOOP	9	11			14	Wait for DADDIU

FIGURE 3.34 The time of issue, execution, and writing result for a dual-issue version of our pipeline *with* speculation. Note that the L.D following the BNE can start execution early, because it is speculative.

One question you may be asking is: How do we ever know which registers are the architectural registers if they are constantly changing? Most of the time when the program is executing it does not matter. There are clearly cases, however, where another process, such as the operating system, must be able to know exactly where the contents of a certain architectural register resides. To understand how this capability is provided, assume the processor does not issue instructions for some period of time. Then eventually, all instructions in the pipeline will commit, and the mapping between the architecturally visible registers and physical registers will become stable. At that point, a subset of the physical registers contains the architecturally visible registers, and the value of any physical register not associated with an architectural register is unneeded. It is then easy to move the architectural registers to a fixed subset of physical registers so that the values can be communicated to another process.

An advantage of the renaming approach versus the ROB approach is that instruction commit is simplified, since it requires only two simple actions: record that the mapping between an architectural register number and physical register number is no longer speculative, and free up any physical registers being used to hold the "older" value of the architectural register. In a design with reservation stations, a station is freed up when the instruction using it completes execution, and a ROB is freed up when the corresponding instruction commits.

With register renaming, deallocating registers is more complex, since before we free up a physical register, we must know that it no it longer corresponds to an architectural register, and that no further uses of the physical register are outstanding. A physical register corresponds to an architectural register until the architectural register is rewritten, causing the renaming table to point elsewhere. That is, if no renaming entry points to a particular physical register, then it no longer corresponds to an architectural register. There may, however, still be uses of the physical register outstanding. The processor can determine whether this is the case by examining the source register specifiers of all instructions in the functional unit queues. If a given physical register does not appear as a source and it is not designated as an architectural register, it may be reclaimed and reallocated.

The process of reclamation can be simplified by counting the register source uses as instructions issue and decrementing the count as the instructions fetch their operands. When the count reaches zero, there are no further outstanding uses.

In addition to simplifying instruction commit, a renaming approach means that instruction issue need not examine both the ROB and the register file for an operand, since all results are in the register file. One possibly disconcerting aspect of the renaming approach is that the "real" architectural registers are never fixed but constantly change according to the contents of a renaming map. Although this complicates the design and debugging, it is not inherently problematic, and is an accepted fact in many newer implementations and sometimes even made architecturally visible, as we will see in the IA-64 architecture in the next chapter.

The PowerPC 603/604 series, the MIPS R1000/12000, the Alpha 21264, and the Pentium II, III and 4 all use register renaming, adding from 20 to 80 extra registers. Since all results are allocated a new virtual register until they commit, these extra registers replace a primary function of the ROB and largely determine how many instructions may be in execution (between issue and commit) at one time.

How much to speculate

One of the significant advantages of speculation is its ability to uncover events that would otherwise stall the pipeline early, such as cache misses. This potential advantage, however, comes with a significant potential disadvantage: the processor may speculate that some costly exceptional event occurs and begin processing the event, when in fact, the speculation was incorrect.

To maintain some of the advantage, while minimizing the disadvantages, most pipelines with speculation will allow only low-cost exceptional events (such as a first-level cache miss) to be handled in speculative mode. If an expensive exceptional event occurs, such as a second-level cache miss or a TLB miss, the processor will wait until the instruction causing the event is no longer speculative before handling the event. Although this may slightly degrade the performance of some programs, it avoids significant performance losses in others, especially those that suffer from a high frequency of such events coupled with less than excellent branch prediction.

Speculating through multiple branches

In the examples we have considered so far, it has been possible to resolve a branch before having to speculate on another. Three different situations can benefit from speculating on multiple branches simultaneously: a very high branch frequency, significant clustering of branches, and long delays in functional units. In the first two cases, achieving high performance may mean that multiple branches are speculated, and it may even mean handling more than one branch per clock. Database programs, and other less structured integer computations, often exhibit these properties, making speculation on multiple branches important. Likewise, long delays in functional units can raise the importance of speculating on multiple branches as a way to avoid stalls from the longer pipeline delays.

Speculating on multiple branches slightly complicates the process of speculation recovery, but is straightforward otherwise. A more complex technique is predicting and speculating on more than one branch per cycle. Although no existing processor has done this for general instruction execution as of 2000, we can expect that it may be needed in the future.

Of course, all the techniques described in the next chapter and in this one cannot take advantage of more parallelism than is provided by the application. The question of how much parallelism is available, and under what circumstances, has been hotly debated and is the topic of the next section.

3.8 Studies of the Limitations of ILP

Exploiting ILP to increase performance began with the first pipelined processors in the 1960s. In the 1980s and 1990s, these techniques were key to achieving rapid performance improvements. The question of how much ILP exists is critical to our long-term ability to enhance performance at a rate that exceeds the increase in speed of the base integrated-circuit technology. On a shorter scale, the critical question of what is needed to exploit more ILP is crucial to both computer designers and compiler writers. The data in this section also provide us with a way to examine the value of ideas that we have introduced in this chapter, including memory disambiguation, register renaming, and speculation.

In this section we review one of the studies done of these questions. The historical section (3.15) describes several studies, including the source for the data in this section (which is Wall's 1993 study). All these studies of available parallelism operate by making a set of assumptions and seeing how much parallelism is available under those assumptions. The data we examine here are from a study that makes the fewest assumptions; in fact, the ultimate hardware model is probably unrealizable. Nonetheless, all such studies assume a certain level of compiler technology and some of these assumptions could affect the results, despite the

use of incredibly ambitious hardware. In addition, new ideas may invalidate the very basic assumptions of this and other studies; for example, value prediction, a technique we discuss at the end of this section, may allow us to overcome the limit of data dependences.

In the future, advances in compiler technology together with significantly new and different hardware techniques may be able to overcome some limitations assumed in these studies; however, it is unlikely that such advances *when coupled with realistic hardware* will overcome all these limits in the near future. Instead, developing new hardware and software techniques to overcome the limits seen in these studies will continue to be one of the most important challenges in computer design.

The Hardware Model

To see what the limits of ILP might be, we first need to define an ideal processor. An ideal processor is one where all artificial constraints on ILP are removed. The only limits on ILP in such a processor are those imposed by the actual data flows either through registers or memory.

The assumptions made for an ideal or perfect processor are as follows:

1. *Register renaming*—There are an infinite number of virtual registers available and hence all WAW and WAR hazards are avoided and an unbounded number of instructions can begin execution simultaneously.
2. *Branch prediction*—Branch prediction is perfect. All conditional branches are predicted exactly.
3. *Jump prediction*—All jumps (including jump register used for return and computed jumps) are perfectly predicted. When combined with perfect branch prediction, this is equivalent to having a processor with perfect speculation and an unbounded buffer of instructions available for execution.
4. *Memory-address alias analysis*—All memory addresses are known exactly and a load can be moved before a store provided that the addresses are not identical.

Assumptions 2 and 3 eliminate *all* control dependences. Likewise, assumptions 1 and 4 eliminate *all but the true* data dependences. Together, these four assumptions mean that *any* instruction in the of the program's execution can be scheduled on the cycle immediately following the execution of the predecessor on which it depends. It is even possible, under these assumptions, for the *last* dynamically executed instruction in the program to be scheduled on the very first cycle! Thus, this set of assumptions subsumes both control and address speculation and implements them as if they were perfect.

Initially, we examine a processor that can issue an unlimited number of instructions at once looking arbitrarily far ahead in the computation. For all the processor models we examine, there are no restrictions on what types of instruc-

tions can execute in a cycle. For the unlimited-issue case, this means there may be an unlimited number of loads or stores issuing in one clock cycle. In addition, all functional unit latencies are assumed to be one cycle, so that any sequence of dependent instructions can issue on successive cycles. Latencies longer than one cycle would decrease the number of issues per cycle, although not the number of instructions under execution at any point. (The instructions in execution at any point are often referred to as *in-flight*.)

Finally, we assume perfect caches, which is equivalent to saying that all loads and stores always complete in one cycle. This assumption allows our study to focus on fundamental limits to ILP. The resulting data, however, will be very optimistic, because realistic caches would significantly reduce the amount of ILP that could be successfully exploited, even if the rest of the processor were perfect!

Of course, this processor is on the edge of unrealizable. For example, the Alpha 21264 is one of the most advanced superscalar processors announced to date. The 21264 issues up to four instructions per clock and initiates execution on up to six (with significant restrictions on the instruction type, e.g., at most two load/stores), supports a large set of renaming registers (41 integer and 41 floating point, allowing up to 80 instructions in-flight), and uses a large tournament-style branch predictor. After looking at the parallelism available for the perfect processor, we will examine the impact of restricting various features.

To measure the available parallelism, a set of programs were compiled and optimized with the standard MIPS optimizing compilers. The programs were instrumented and executed to produce a trace of the instruction and data references. Every instruction in the trace is then scheduled as early as possible, limited only by the data dependences. Since a trace is used, perfect branch prediction and perfect alias analysis are easy to do. With these mechanisms, instructions may be scheduled much earlier than they would otherwise, moving across large numbers of instructions on which they are not data dependent, including branches, since branches are perfectly predicted.

Figure 3.35 shows the average amount of parallelism available for six of the SPEC92 benchmarks. Throughout this section the parallelism is measured by the average instruction issue rate (remember that all instructions have a one-cycle latency), which is the ideal IPC. Three of these benchmarks (fpppp, doduc, and tomcatv) are floating-point intensive, and the other three are integer programs. Two of the floating-point benchmarks (fpppp and tomcatv) have extensive parallelism, which could be exploited by a vector computer or by a multiprocessor (the structure in fpppp is quite messy, however, since some hand transformations have been done on the code). The doduc program has extensive parallelism, but the parallelism does not occur in simple parallel loops as it does in fpppp and tomcatv. The program li is a LISP interpreter that has many short dependences.

In the next few sections, we restrict various aspects of this processor to show what the effects of various assumptions are before looking at some ambitious but realizable processors.

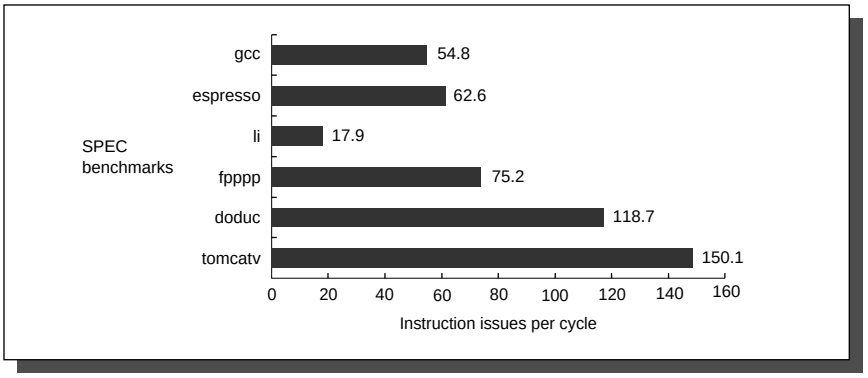


FIGURE 3.35 ILP available in a perfect processor for six of the SPEC92 benchmarks. The first three programs are integer programs, and the last three are floating-point programs. The floating-point programs are loop-intensive and have large amounts of loop-level parallelism. ***Artist, Please round the value labels to integers on this graph.***

Limitations on the Window Size and Maximum Issue Count

To build a processor that even comes close to perfect branch prediction and perfect alias analysis requires extensive dynamic analysis, since static compile-time schemes cannot be perfect. Of course, most realistic dynamic schemes will not be perfect, but the use of dynamic schemes will provide the ability to uncover parallelism that cannot be analyzed by static compile-time analysis. Thus, a dynamic processor might be able to more closely match the amount of parallelism uncovered by our ideal processor.

How close could a real dynamically scheduled, speculative processor come to the ideal processor? To gain insight into this question, consider what the perfect processor must do:

1. Look arbitrarily far ahead to find a set of instructions to issue, predicting all branches perfectly.
2. Rename all register uses to avoid WAR and WAW hazards.
3. Determine whether there are any data dependencies among the instructions in the issue packet; if so, rename accordingly.
4. Determine if any memory dependences exist among the issuing instructions and handle them appropriately.
5. Provide enough replicated functional units to allow all the ready instructions to issue.

Obviously, this analysis is quite complicated. For example, to determine whether n issuing instructions have any register dependences among them, assuming all instructions are register-register and the total number of registers is unbounded, requires

$$2n - 2 + 2n - 4 + \ldots + 2 = 2\sum_{i=1}^{n-1} i = 2\frac{(n-1)n}{2} = n^2 - n$$

comparisons. Thus, to detect dependences among the next 2000 instructions—the default size we assume in several figures—requires almost *four million* comparisons! Even issuing only 50 instructions requires 2450 comparisons. This cost obviously limits the number of instructions that can be considered for issue at once.

In existing and near-term processors, the costs are not quite so high, since we need only detect dependence pairs and the limited number of registers allows different solutions. Furthermore, in a real processor, issue occurs in-order and dependent instructions are handled by a renaming process that accommodates dependent renaming in one clock. Once instructions are issued, the detection of dependences is handled in a distributed fashion by the reservation stations or scoreboard.

The set of instructions that are examined for simultaneous execution is called the *window*. Each instruction in the window must be kept in the processor and the number of comparisons required every clock is equal to the maximum completion rate times the window size times the number of operands per instruction (today typically 6 x 80 x 2= 960), since every pending instruction must look at every completing instruction for either of its operands. Thus, the total window size is limited by the required storage, the comparisons, and a limited issue rate, which makes larger window less helpful. To date, the window size has been in the range of 32 to 126, which can require over 2,000 comparisons. The HP PA 8600 reportedly has over 7,000 comparators!

The window size directly limits the number of instructions that begin execution in a given cycle. In practice, real processors will have a more limited number of functional units (e.g., no processor has handled more than two memory references per clock or more than two FP operations), as well as limited numbers of buses and register access ports, which serve as limits on the number of instructions initiated in the same clock. Thus, the maximum number of instructions that may issue, begin execution, or commit in the same clock cycle is usually much smaller than the window size.

Obviously, the number of possible implementation constraints in a multiple issue processor is large, including: issues per clock, functional units and unit latency, register file ports, functional unit queues (which may be fewer than units), issue limits for branches, and limitations on instruction commit. Each of these acts as constraint on the ILP. Rather than try to understand each of these effects, however, we will focus on limiting the size of the window, with the understanding that all other restrictions would further reduce the amount of parallelism that can be exploited.

Figures 3.36 and 3.37 show the effects of restricting the size of the window from which an instruction can execute; the *only* difference in the two graphs is the format—the data are identical. As we can see in Figure 3.36, the amount of

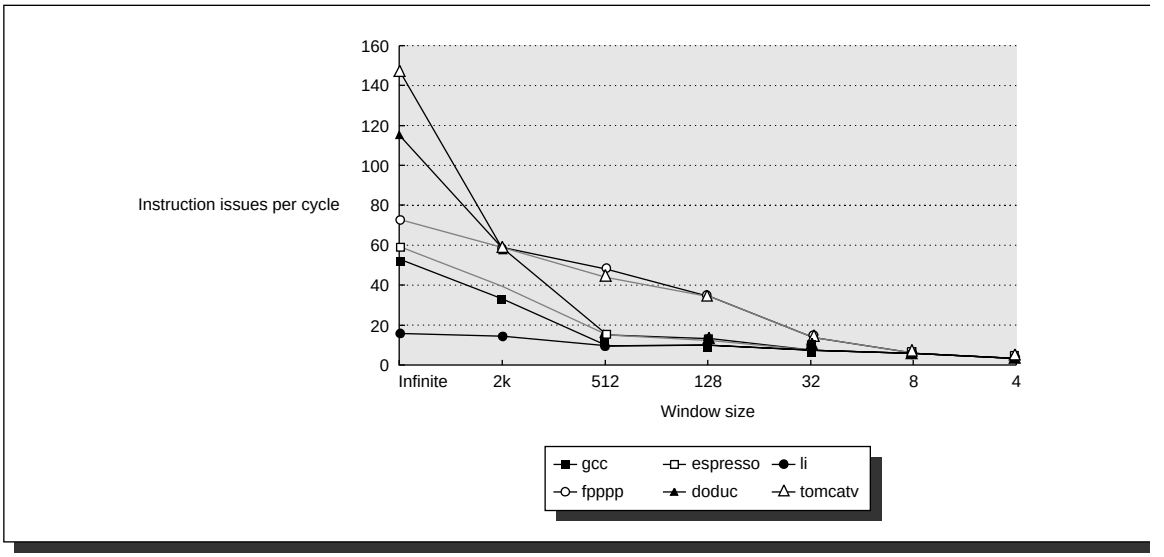


FIGURE 3.36 The effects of reducing the size of the window. The window is the group of instructions from which an instruction can execute. The start of the window is the earliest uncompleted instruction (remember that instructions complete in one cycle), and the last instruction in the window is determined by the window size. The instructions in the window are obtained by perfectly predicting branches and selecting instructions until the window is full. ***Artist: The espresso datapoint with coordinates (2K,40) is missing.***

parallelism uncovered falls sharply with decreasing window size. In 2000, the most advanced processors have window sizes in the rang of 64-128, but these window sizes are not strictly comparable to those shown in Figure 3.36 for two reasons. First, the functional units are pipelined, reducing the effective window size compared to the case where all units have single-cycle latency. Second, in real processors the window must also hold any memory references waiting on a cache miss, which are not considered in this model, since it assumes a perfect, single-cycle cache access.

As we can see in Figure 3.37, the integer programs do not contain nearly as much parallelism as the floating-point programs. This result is to be expected. Looking at how the parallelism drops off in Figure 3.37 makes it clear that the parallelism in the floating-point cases is coming from loop-level parallelism. The fact that the amount of parallelism at low window sizes is not that different among the floating-point and integer programs implies a structure where there are dependences within loop bodies, but few dependences between loop iterations in programs such as tomcatv. At small window sizes, the processors simply cannot see the instructions in the next loop iteration that could be issued in parallel with instructions from the current iteration. This case is an example of where better compiler technology (see the next chapter) could uncover higher amounts of ILP,

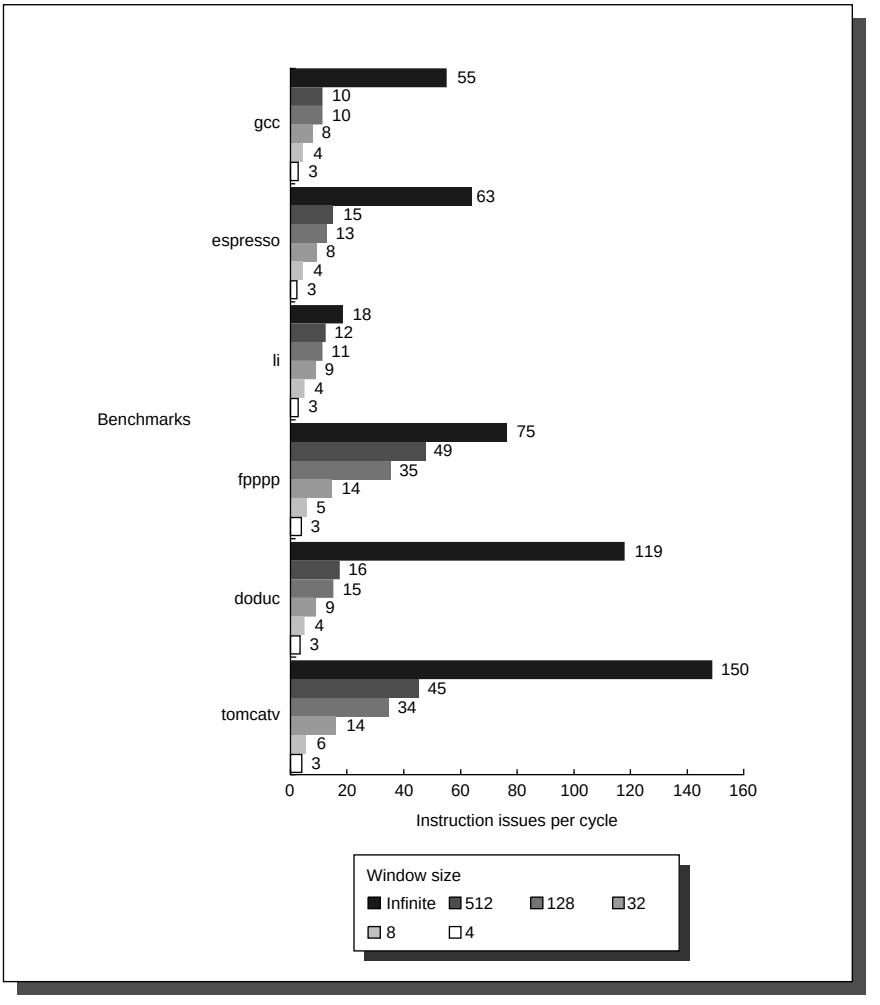


FIGURE 3.37 The effect of window size shown by each application by plotting the average number of instruction issues per clock cycle. The most interesting observation is that at modest window sizes, the amount of parallelism found in the integer and floating-point programs is similar. ***Artist: please add a data series to this graph. Legend label is 2K, this set of points goes between the Infinite and 512 series. The values of the points to add from top to bottom are: 36, 41, 15, 61, 59, 60. (So that, e.g., there will be a bar labeled 41, with the appropriate height as the second bar in the set corresponding to espresso.***

since it could find the loop-level parallelism and schedule the code to take advantage of it, even with small window sizes.

We know that large window sizes are impractical and inefficient, and the data in Figures 3.36 and 3.37 tell us that issue rates will be considerably reduced with realistic windows, thus we will assume a base window size of 2K entries and a maximum issue capability of 64 instructions per clock for the rest of this analysis. As we will see in the next few sections, when the rest of the processor is not perfect, a 2K window and a 64-issue limitation do not constrain the amount of ILP the processor can exploit.

The Effects of Realistic Branch and Jump Prediction

Our ideal processor assumes that branches can be perfectly predicted: The outcome of any branch in the program is known before the first instruction is executed! Of course, no real processor can ever achieve this. Figures 3.38 and 3.39 show the effects of more realistic prediction schemes in two different formats. Our data is for several different branch-prediction schemes varying from perfect to no predictor. We assume a separate predictor is used for jumps. Jump predictors are important primarily with the most accurate branch predictors, since the branch frequency is higher and the accuracy of the branch predictors dominates.

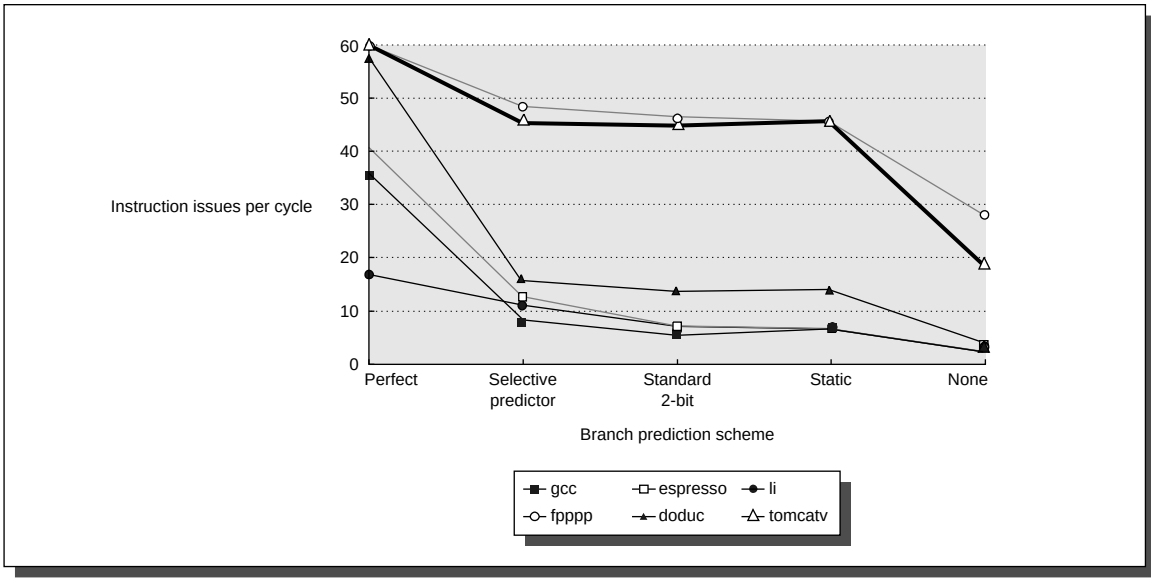


FIGURE 3.38 The effect of branch-prediction schemes. This graph shows the impact of going from a perfect model of branch prediction (all branches predicted correctly arbitrarily far ahead) to various dynamic predictors (selective and two-bit), to compile time, profile-based prediction, and finally to using no predictor. The predictors are described precisely in the text. ***artist: change label "selective predictor" to "Tournament predictor"***

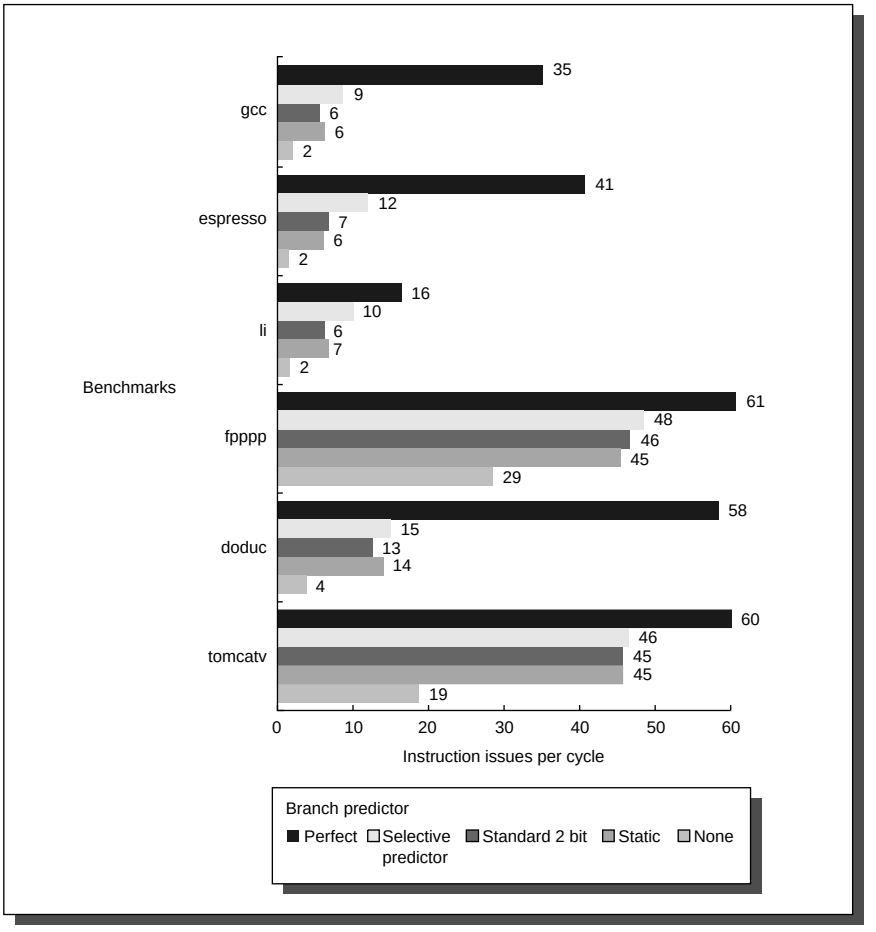


FIGURE 3.39 The effect of branch-prediction schemes sorted by application. This graph highlights the differences among the programs with extensive loop-level parallelism (tomcatv and fpppp) and those without (the integer programs and doduc). **Artist:** ***change label "selective predictor" to "Tournament predictor" also can you make the caption vertical and place it in the upper right corner.***

The five levels of branch prediction shown in these figures are

1. *Perfect*—All branches and jumps are perfectly predicted at the start of execution.
2. *Tournament-based branch predictor*—The prediction scheme uses a correlating two-bit predictor and a noncorrelating two-bit predictor together with a selector, which chooses the best predictor for each branch. The prediction buffer

contains 2^{13} (8K) entries, each consisting of three two-bit fields, two of which are predictors and the third is a selector. The correlating predictor is indexed using the exclusive-or of the branch address and the global branch history. The noncorrelating predictor is the standard two-bit predictor indexed by the branch address. The selector table is also indexed by the branch address and specifies whether the correlating or noncorrelating predictor should be used. The selector is incremented or decremented just as we would for a standard two-bit predictor. This predictor, which uses a total of 48K bits, outperforms both the correlating and noncorrelating predictors, achieving an average accuracy of 97% for these six SPEC benchmarks; this predictor is comparable in strategy and somewhat larger than the best predictors in use in 2000. Jump prediction is done with a pair of 2K-entry predictors, one organized as a circular buffer for predicting returns and one organized as a standard predictor and used for computed jumps (as in case statement or computed gotos). These jump predictors are nearly perfect.

3. *Standard two-bit predictor with 512 two-bit entries*—In addition, we assume a 16-entry buffer to predict returns.

4. *Static*—A static predictor uses the profile history of the program and predicts that the branch is always taken or always not taken based on the profile.

5. *None*—No branch prediction is used, though jumps are still predicted. Parallelism is largely limited to within a basic block.

Since we do *not* charge additional cycles for a mispredicted branch, the only effect of varying the branch prediction is to vary the amount of parallelism that can be exploited across basic blocks by speculation. Figure 3.40 shows the accuracy of the three realistic predictors for the conditional branches for the subset of SPEC92 benchmarks we include here. By comparison, Figure 3.61 on page 341 shows the size and type of branch predictor in recent high performance processors.

Figure 3.39 shows that the branch behavior of two of the floating-point programs is much simpler than the other programs, primarily because these two programs have many fewer branches and the few branches that exist are more predictable. This property allows significant amounts of parallelism to be exploited with realistic prediction schemes. In contrast, for all the integer programs and for doduc, the FP benchmark with the least loop-level parallelism, even the difference between perfect branch prediction and the ambitious selective predictor is dramatic. Like the window size data, these figures tell us that to achieve significant amounts of parallelism in integer programs, the processor must select and execute instructions that are widely separated. When branch prediction is not highly accurate, the mispredicted branches become a barrier to finding the parallelism.

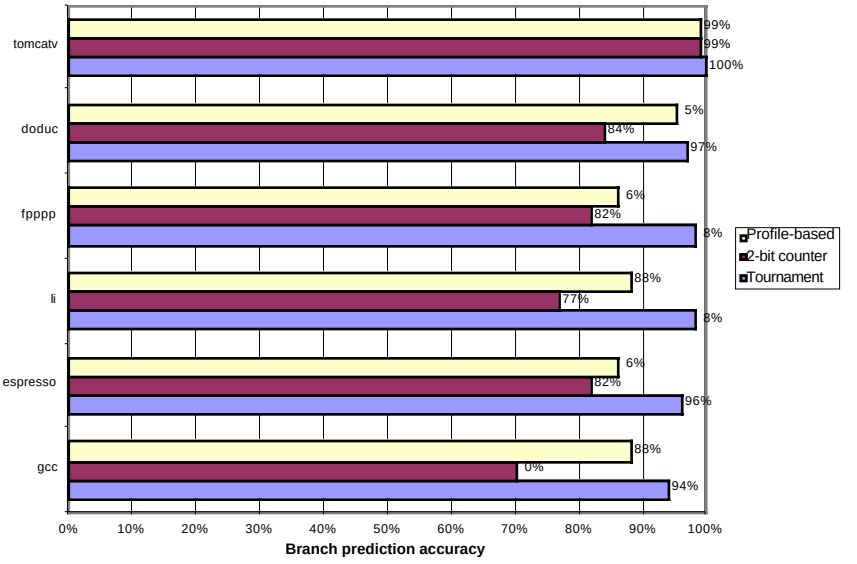


FIGURE 3.40 Branch prediction accuracy for the conditional branches in the SPEC92 subset.

As we have seen, branch prediction is critical, especially with a window size of 2K instructions and an issue limit of 64. For the rest of the studies, in addition to the window and issue limit, we assume as a base a more ambitious tournament predictor that uses two levels of prediction and a total of 8K entries. This predictor, which requires more than 150K bits of storage (roughly four times the largest predictor to date), slightly outperforms the selective predictor described above (by about 0.5–1%). We also assume a pair of 2K jump and return predictors, as described above.

The Effects of Finite Registers

Our ideal processor eliminates all name dependences among register references using an infinite set of physical registers. To date, the Alpha 21264 has provided the largest number of extended registers: 41 integer and 41 FP registers, in addition to 32 integer and 32 floating point architectural registers. Figures 3.41 and 3.42 show the effect of reducing the number of registers available for renaming, again using the same data in two different forms. Both the FP and GP registers are increased by the number of registers shown on the axis or in the legend.

At first, the results in these figures might seem somewhat surprising: you might expect that name dependences should only slightly reduce the parallelism avail-

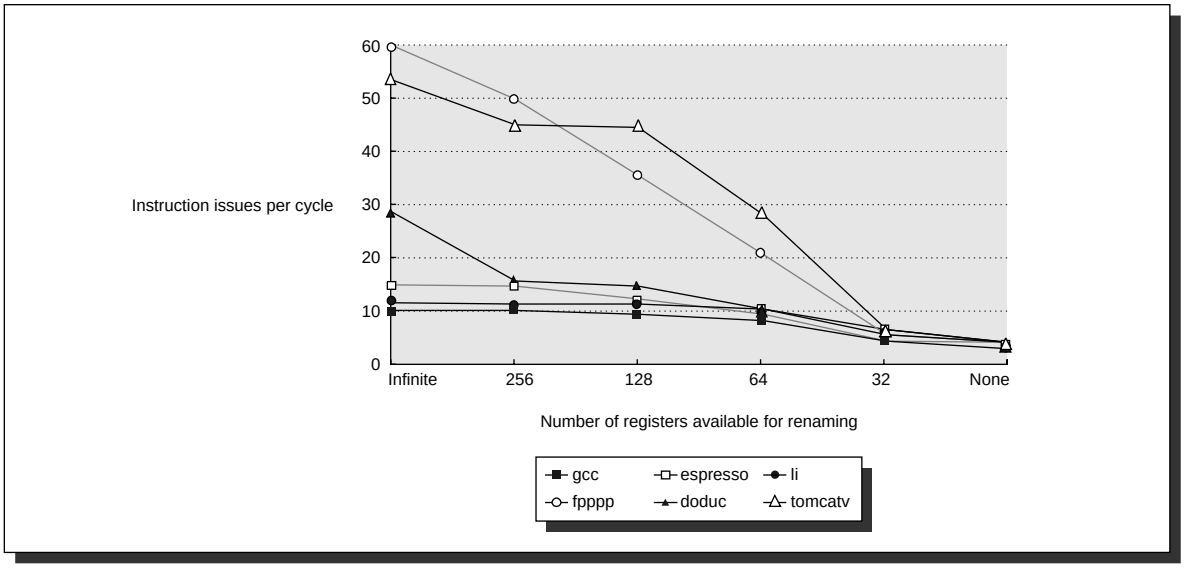


FIGURE 3.41 The effect of finite numbers of registers available for renaming. *Both* the number of FP registers and the number of GP registers are increased by the number shown on the x axis. The effect is most dramatic on the FP programs, although having only 32 extra GP and 32 extra FP registers has a significant impact on all the programs. As stated earlier, we assume a window size of 2K entries and a maximum issue width of 64 instructions. None implies no extra registers available.

able. Remember though, exploiting large amounts of parallelism requires evaluating many independent threads of execution. Thus, many registers are needed to hold live variables from these threads. Figure 3.41 shows that the impact of having only a finite number of registers is significant if extensive parallelism exists. Although these graphs show a large impact on the floating-point programs, the impact on the integer programs is small primarily because the limitations in window size and branch prediction have limited the ILP substantially, making renaming less valuable. In addition, notice that the reduction in available parallelism is significant even if 64 additional integer and 64 additional FP registers are available for renaming, which is more than the number of extra registers available on any existing processor as of 2000.

Although register renaming is obviously critical to performance, an infinite number of registers is obviously not practical. Thus, for the next section, we assume that there are 256 integer and 256 FP registers available for renaming—far more than any anticipated processor has.

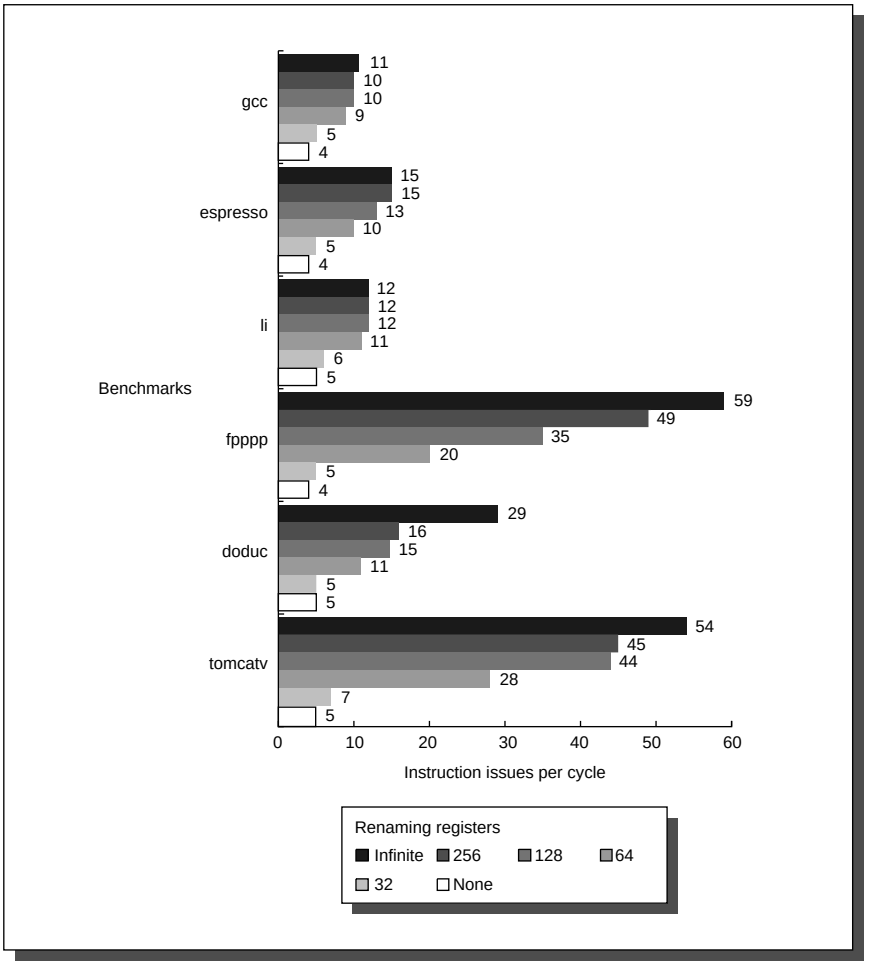


FIGURE 3.42 The reduction in available parallelism is significant when fewer than an unbounded number of renaming registers are available. For the integer programs, the impact of having more than 64 registers is not seen here. To use more than 64 registers requires uncovering lots of parallelism, which for the integer programs requires essentially perfect branch prediction. **Artist:** ***can you make the caption vertical and place it in the upper right corner.***

The Effects of Imperfect Alias Analysis

Our optimal model assumes that it can perfectly analyze all memory dependences, as well as eliminate all register name dependences. Of course, perfect alias analysis is not possible in practice: The analysis cannot be perfect at com-

pile time, and it requires a potentially unbounded number of comparisons at runtime (since the number of simultaneous memory references is unconstrained). Figures 3.43 and 3.44 show the impact of three other models of memory alias

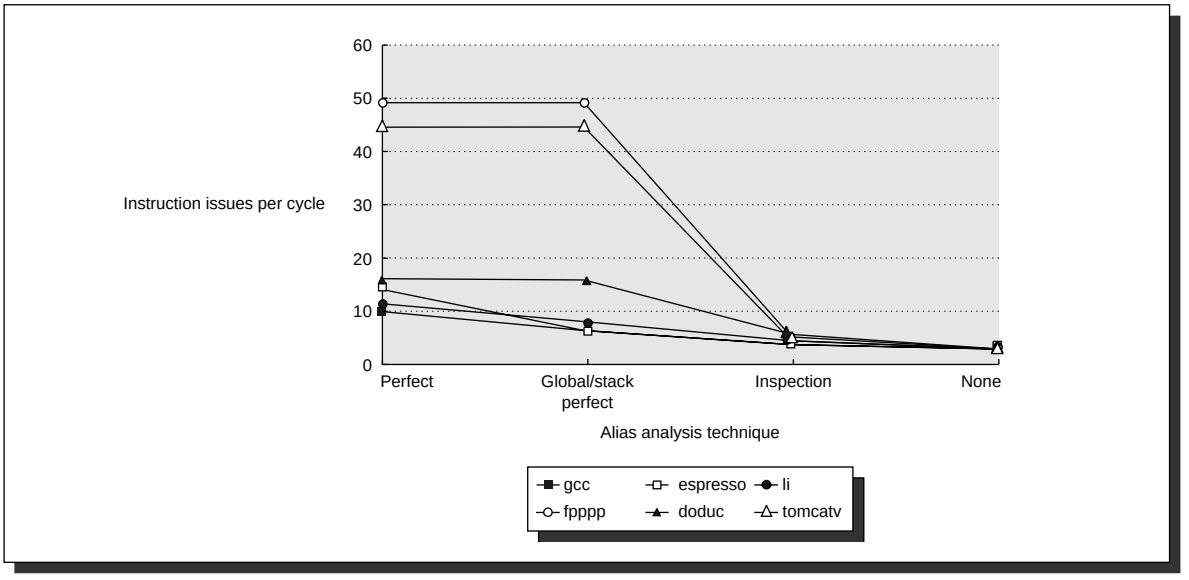


FIGURE 3.43 The effect of various alias analysis techniques on the amount of ILP. Anything less than perfect analysis has a dramatic impact on the amount of parallelism found in the integer programs, and global/stack analysis is perfect (and unrealizable) for the FORTRAN programs. As we said earlier, we assume a maximum issue width of 64 instructions and a window of 2K instructions.**Artist:** ***can you make the caption vertical and place it on the right side***

analysis, in addition to perfect analysis. The three models are:

1. *Global/stack perfect*—This model does perfect predictions for global and stack references and assumes all heap references conflict. This model represents an idealized version of the best compiler-based analysis schemes currently in production. Recent and ongoing research on alias analysis for pointers should improve the handling of pointers to the heap in the future.
2. *Inspection*—This model examines the accesses to see if they can be determined not to interfere at compile time. For example, if an access uses R10 as a base register with an offset of 20, then another access that uses R10 as a base register with an offset of 100 cannot interfere. In addition, addresses based on registers that point to different allocation areas (such as the global area and the stack area) are assumed never to alias. This analysis is similar to that performed by many existing commercial compilers, though newer compilers can

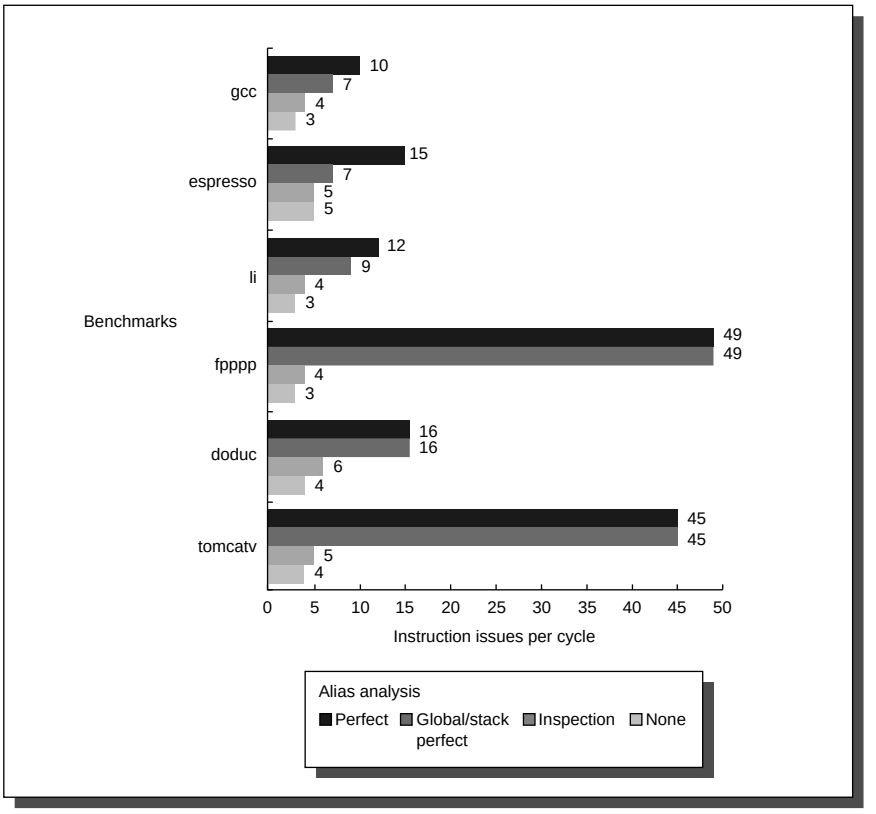


FIGURE 3.44 The effect of varying levels of alias analysis on individual programs.

Artist: ***can you make the caption vertical and place it in the upper right corner also delete the phrase "Alias analysis" in the legend.***

do better, at least for loop-oriented programs.

3. *None*—All memory references are assumed to conflict.

As one might expect, for the FORTRAN programs (where no heap references exist), there is no difference between perfect and global/stack perfect analysis. The global/stack perfect analysis is optimistic, since no compiler could ever find all array dependences exactly. The fact that perfect analysis of global and stack references is still a factor of two better than inspection indicates that either sophisticated compiler analysis or dynamic analysis on the fly will be required to obtain much parallelism. In practice, dynamically scheduled processors rely on dynamic memory disambiguation and are limited by three factors:

1. To implement perfect dynamic disambiguation for a given load, we must know the memory addresses of all earlier stores that not yet committed, since a load may have a dependence through memory on a store. One technique for reducing this limitation on in-order address calculation is memory address speculation. With memory address speculation, the processor either assumes that no such memory dependences exist or uses a hardware prediction mechanism to predict if a dependence exists, stalling the load if a dependence is predicted. Of course, the processor can be wrong about the absence of the dependence, so we need a mechanism to discover if a dependence truly exists and to recover if so. To discover if a dependence exists, the processor examines the destination address of each completing store that is earlier in program order than the given load. If a dependence that should have been enforced occurs, the processor uses the speculative restart mechanism to redo the load and the following instructions. (We will see how this type of address speculation can be supported with instruction set extensions in the next chapter.)
2. Only a small number of memory references can be disambiguated per clock cycle.
3. The number of the load/store buffers determines how much earlier or later in the instruction stream a load or store may be moved.

Both the number of simultaneous disambiguations and the number of the load/store buffers will affect the clock cycle time.

3.9 Limitations on ILP for Realizable Processors

In this section we look at the performance of processors ambitious levels of hardware support equal to or better than what is likely in the next five years. In particular we assume the following fixed attributes:

1. Up to 64 instruction issues per clock with *no* issue restrictions. As we discuss later, the practical implications of very wide issue widths on clock rate, logic complexity, and power may be the most important limitation on exploiting ILP.
2. A tournament predictor with 1K entries and a 16-entry return predictor. This predictor is fairly comparable to the best predictors in 2000; the predictor is not a primary bottleneck.
3. Perfect disambiguation of memory references done dynamically—this is ambitious but perhaps attainable for small window sizes (and hence small issue rates and load/store buffers) or through a memory dependence predictor.
4. Register renaming with 64 additional integer and 64 additional FP registers,

exceeding largest number available on any processor in 2001 (41 and 41 in the Alpha 21264), but probably easily reachable within two or three years.

Figures 3.45 and 3.46 show the result for this configuration as we vary the window size. This configuration is more complex and expensive than any existing implementations, especially in terms of the number of instruction issues, which is more than ten times larger than the largest number of issues available on any processor in 2001. Nonetheless, it gives a useful bound on what future implementations might yield. The data in these figures is likely to be very optimistic for another reason. There are no issue restrictions among the 64 instructions: they may all be memory references. No one would even contemplate this capability in a processor in the near future. Unfortunately, it is quite difficult to bound the performance of a processor with reasonable issue restrictions; not only is the space of possibilities quite large, but the existence of issue restrictions requires that the parallelism be evaluated with an accurate instruction scheduler, making the cost of studying processors with large numbers of issues very expensive.

In addition, remember that in interpreting these results, cache misses and non-unit latencies have not been taken into account, and both these effects will have significant impact (see the Exercises).

Figure 3.45 shows the parallelism versus window size. The most startling observation is that with the realistic processor constraints listed above, the effect of the window size for the integer programs is not so severe as for FP programs. This result points to the key difference between these two types of programs. The availability of loop-level parallelism in two of the FP programs means that the amount of ILP that can be exploited is higher, but that for integer programs other factors—such as branch prediction, register renaming, and less parallelism to start with—are all important limitations. This observation is critical, because of the increased emphasis on integer performance in the last few years. As we will see in the next section, for a realistic processor in 2000, the actual performance levels are much lower than those shown in Figure 3.45.

Given the difficulty of increasing the instruction rates with realistic hardware designs, designers face a challenge in deciding how best to use the limited resources available on a integrated circuit. One of the most interesting trade-offs is between simpler processors with larger caches and higher clock rates versus more emphasis on instruction-level parallelism with a slower clock and smaller caches. The following Example illustrates the challenges.

EXAMPLE Consider the following three hypothetical, but not atypical, processors, which we run with the SPEC gcc benchmark:

1. A simple MIPS two-issue static pipe running at a clock rate of 1 GHz

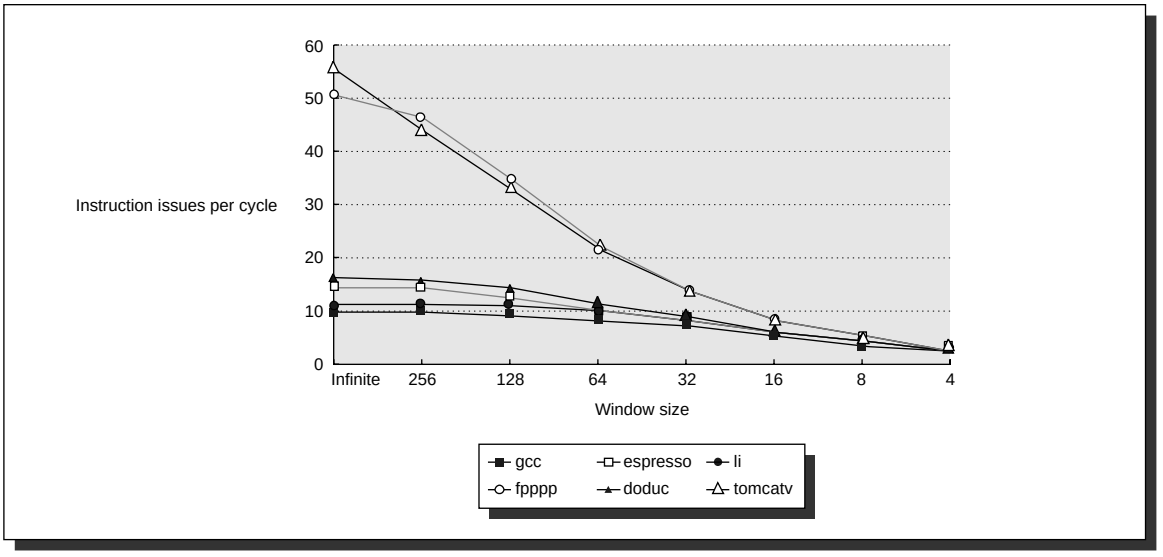


FIGURE 3.45 The amount of parallelism available for a wide variety of window sizes and a fixed implementation with up to 64 issues per clock. Although there are fewer rename registers than the window size, the fact that all operations have zero latency and that the number of rename registers equals the issue width, allows the processor to exploit parallelism within the entire window. In a real implementation, the window size and the number of renaming registers must be balanced to prevent one of these factors from overly constraining the issue rate.

and achieving a pipeline CPI of 1.0. This processor has a cache system that yields 0.01 misses per instruction.

2. A deeply pipelined version of MIPS with slightly smaller caches and a 1.2 GHz clock rate. The pipeline CPI of the processor is 1.2, and the smaller caches yield 0.015 misses per instruction on average.

3. A speculative superscalar with a 64-entry window. It achieves one-half of the ideal issue rate measured for this window size. (Use the data in Figure 3.45 on page 311.) This processor has the smallest caches, which leads to 0.02 misses per instruction, but it hides 10% of the miss penalty on every miss by dynamic scheduling. This processor has a 800-MHz clock.

Assume that the main memory time (which sets the miss penalty) is 100 ns. Determine the relative performance of these three processors.

ANSWER First, we use the miss penalty and miss rate information to compute the contribution to CPI from cache misses for each configuration. We do this with the following formula:

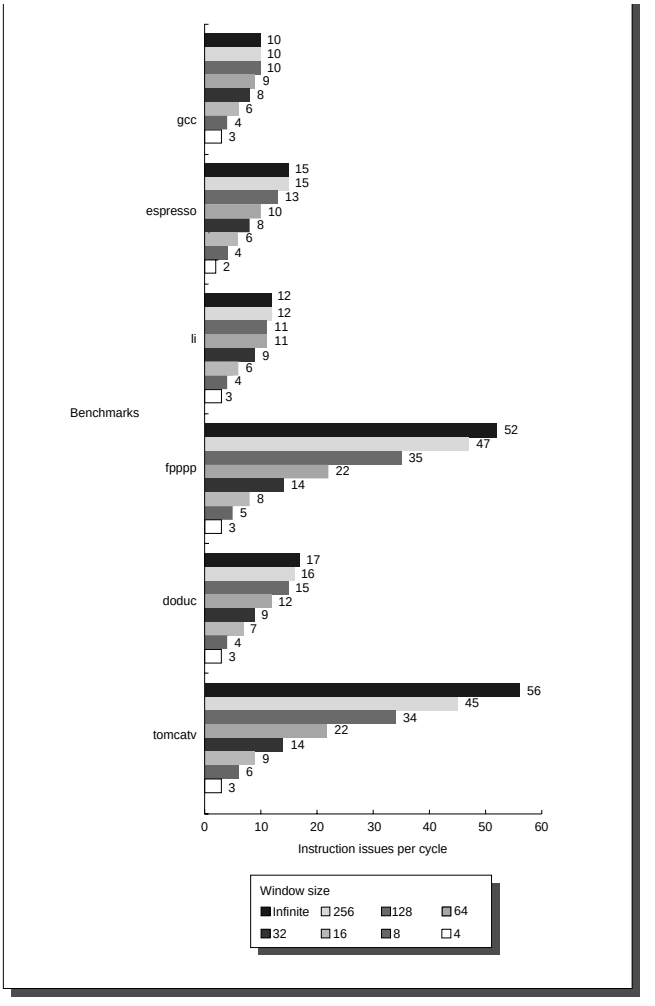


FIGURE 3.46 The amount of parallelism available versus the window size for a variety of integer and floating-point programs with up to 64 arbitrary instruction issues per clock.
Artist: could you make legend vertical and move to upper right corner.

$$\text{Cache CPI} = \text{Misses per instruction} \times \text{Miss penalty}$$

We need to compute the miss penalties for each system:

$$\text{Miss penalty} = \frac{\text{Memory access time}}{\text{Clock cycle}}$$

The clock cycle times for the processors are 1 ns, 0.83 ns, and 1.25 ns, respectively. Hence, the miss penalties are

$$\text{Miss penalty}_1 = \frac{100 \text{ ns}}{1 \text{ ns}} = 100 \text{ cycles}$$

$$\text{Miss penalty}_2 = \frac{100 \text{ ns}}{0.83 \text{ ns}} = 120 \text{ cycles}$$

$$\text{Miss penalty}_3 = \frac{0.9 \times 100 \text{ ns}}{1.25 \text{ ns}} = 72 \text{ cycles}$$

Applying this for each cache:

$\text{Cache CPI}_1 = 0.01 \times 100 = 1.0$
$\text{Cache CPI}_2 = 0.015 \times 120 = 1.8$
$\text{Cache CPI}_3 = 0.02 \times 72 = 1.44$

We know the pipeline CPI contribution for everything but processor 3; its pipeline CPI is given by

$$\text{Pipeline CPI}_3 = \frac{1}{\text{Issue rate}} = \frac{1}{9 \times 0.5} = \frac{1}{4.5} = 0.22$$

Now we can find the CPI for each processor by adding the pipeline and cache CPI contributions.

$\text{CPI}_1 = 1.0 + 1.0 = 2.0$
$\text{CPI}_2 = 1.2 + 1.8 = 3.0$
$\text{CPI}_3 = 0.22 + 1.44 = 1.66$

Since this is the same architecture we can compare instruction execution rates to determine relative performance:

$$\text{Instruction execution rate} = \frac{\text{CR}}{\text{CPI}}$$

$$\text{Instruction execution rate}_1 = \frac{1000 \text{ MHz}}{2} = 500 \text{ MIPS}$$

$$\text{Instruction execution rate}_2 = \frac{1200 \text{ MHz}}{3.0} = 400 \text{ MIPS}$$

$$\text{Instruction execution rate}_3 = \frac{800 \text{ MHz}}{1.66} = 482 \text{ MIPS}$$

In this example, the moderate issue processor looks best. Of course, the designer building either system 2 or system 3 will probably be alarmed by

the large fraction of the system performance lost to cache misses. In the Chapter 5 we'll see the most common solution to this problem: adding another level of caches. n

Beyond the limits of this study

Like any limit study, the study we have examined in this section has its own limitations. We divide these into two classes: limitations that arise even for the perfect speculative processor and limitations that arise for one or more realistic models. Of course, all the limitations in the first class apply to the second. The most important limitations that apply even to the perfect model are:

1. *WAR and WAW hazards through memory:* the study eliminated WAW and WAR hazards through register renaming, but not in memory usage. Although, at first glance it might appear that such circumstances are rare (especially WAW hazards), they arise due to the allocation of stack frames. A called procedure reuses the memory locations of a previous procedure on the stack and this can lead to WAW and WAR hazards that are unnecessarily limiting. Austin and Sohi's 1992 paper examines this issue.

2. *Unnecessary dependences:* with infinite numbers of registers, all but true register data dependences are removed. There are, however, dependences arising from either recurrences or code generation conventions that introduce unnecessary true data dependences. One example of these is the dependence on the control variable in a simple do-loop: since the control variable is incremented on every loop iteration, the loop contains at least one dependence. As we show in the next chapter, loop unrolling and aggressive algebraic optimization can remove such dependent computation. Wall's study includes a limited amount of such optimizations, but applying them more aggressively could lead to increased amounts of ILP. In addition, certain code generation conventions introduce unneeded dependences, in particular the use of return address registers and a register for the stack pointer (which is incremented and decremented in the call/return sequence). Wall removes the effect of the return address register, but the use of a stack pointer in the linkage convention can cause "unnecessary" dependences. Postiff, Greene, Tyson, and Mudge explored the advantages of removing this constraint in a 1999 paper.

3. *Overcoming the data flow limit*: a recent proposed idea to boost ILP, which goes beyond the capability of the study above, is *value prediction*. Value prediction consists of predicting data values and speculating on the prediction. There are two obvious uses of this scheme: predicting data values and speculating on the result and predicting address values for memory alias elimination. The latter affects parallelism only under less than perfect circumstances,

as we discuss shortly.

Value prediction has possibly the most potential for increasing ILP. *Data value prediction and speculation* predicts data values and uses them in destination instructions speculatively. Such speculation allows multiple dependent instructions to be executed in the same clock cycle, thus increasing the potential ILP. To be effective, however, data values must be predicted very accurately, since they will be used by consuming instructions, just as if they were correctly computed. Thus, inaccurate prediction will lead to incorrect speculation and recovery, just as when branches are mispredicted.

One insight that gives some hope is that certain instructions produce the same values with high frequency, so it may be possible to selectively predict values for certain instructions with high accuracy. Obviously, perfect data value prediction would lead to infinite parallelism, since every value of every instruction could be predicted a priori.

Thus, studying the effect of value prediction in true limit studies is difficult and has not yet been done. Several studies have examined the role of value prediction in exploiting ILP in more realistic processors (e.g., Lipasti, Wilkerson, and Shen in 1996). The extent to which general value prediction will be used in real processors remains unclear at the present.

For a less than perfect processor, there are several ideas, which have been proposed, that could expose more ILP. We mention the two most important here:

1. *Address value prediction and speculation* predicts memory address values and speculates by reordering loads and stores. This technique eliminates the need to compute effective addresses in-order to determine whether memory references can be reordered, and could provide better aliasing analysis than any practical scheme. Because we need not actually predict data values, but only if effective addresses are identical, this type of prediction can be accomplished by simpler techniques. Recent processors include limited versions of this technique and it can be expected that future implementations of address value prediction may yield an approximation to perfect alias analysis, allowing processors to eliminate this limit to exploiting ILP.

2. Speculating on multiple paths: this idea was discussed by Lam and Wilson in 1992 and explored in the study covered in this section. By speculating on multiple paths, the cost of incorrect recovery is reduced and more parallelism can be uncovered. It only makes sense to evaluate this scheme for a limited number of branches, because the hardware resources required grow exponentially. Wall's 1993 study provides data for speculating in both directions on up to eight branches. Whether such schemes ever become practical, or whether it will always be better to devote the equivalent silicon area to better branch predictors remains to be seen. In Chapter 8, we discuss thread-level parallelism and the use of speculative threads.

It is critical to understand that none of the limits in this section are fundamental in the sense that overcoming them requires a change in the laws of physics!

Instead, they are practical limitations that imply the existence of some formidable barriers to exploiting additional ILP. These limitations–whether they be window size, alias detection, or branch prediction–represent challenges for designers and researchers to overcome! As we discuss in the concluding remarks, there are a variety of other practical issues that may actually be the more serious limits to exploiting ILP in future processors.

3.10 Putting It All Together: The P6 Microarchitecture

The Intel P6 microarchitecture forms the basis for the Pentium Pro, Pentium II, and the Pentium III. In addition to some specialized instruction set extensions (MMX and SSE), these three processors differ in clock rate, cache architecture, and memory interface and is summarized in Figure 3.47.

Processor	First ship date	Clock rate range	L1 cache	L2 cache
Pentium Pro	1995	100–200 MHz	8KB instr. + 8KB data	256 KB–1,024 KB
Pentium II	1998	233–450 MHz	16KB instr. + 16KB data	256 KB–512 KB
Pentium II Xenon	1999	400-450 MHz	16KB instr. + 16KB data	512 KB–2 MB
Celeron	1999	500-900 MHz	16KB instr. + 16KB data	128 KB
Pentium III	1999	450–1,100 MHz	16KB instr. + 16KB data	256KB–512 KB
Pentium III Xenon	2000	700-900 MHz	16KB instr. + 16KB data	1 MB–2 MB

FIGURE 3.47 The Intel processors based on the P6 microarchitecture and their important differences. In the Pentium Pro, the processor and specialized cache SRAMs were integrated into a multichip module. In the Pentium II standard SRAMs are used. In the Pentium III, there is either on on-chip 256 KB L2 cache or an off-chip 512 KB cache. The Xenon version are intended for server applications; they use an off-chip L2 and support multiprocessing. The Pentium II added the MMX instruction extension. while the Pentium III added the SSE extensions.

The P6 microarchitecture is a dynamically scheduled processor that translates each IA-32 instruction to a series of micro-operations (uops) that are executed by the pipeline; the uops are similar to typical RISC instructions. Up to three IA-32 instructions are fetched, decoded, and translated into uops every clock cycle. If an IA-32 instruction requires more than four uops, it is implemented by a microcoded sequence that generates the necessary uops in multiple clock cycles. The maximum number of uops that may be generated per clock cycle is six, with four allocated to the first IA-32 instruction, and one uop slot to each of the remaining two IA-32 instructions.

The uops are executed by an out-of-order speculative pipeline using register renaming and a ROB. This pipeline is very similar to that in section 3.7, except that the functional unit capability and the sizes of buffers are different. Up to three uops per clock can be renamed and dispatched to the reservation stations; instruction commit can also complete up to three uops per clock. The pipeline is structured in 14 stages composed of the following:

- 8 stages are used for in-order instruction fetch, decode, and dispatch. The next instruction is selected during fetch using a 512-entry, two-level branch predictor. The decode and issue stages including register renaming (using 40 virtual registers) and dispatch to one of 20 reservation stations and to one of 40 entries in the ROB.
- 3 stages are used for out-of-order execution in one of five separate functional units (integer unit, FP unit, branch unit, memory address unit, and memory access unit). The execution pipeline is from 1 cycle (for simple integer ALU operations) to 32 cycles for FP divide. The issue rate and latency of some typical operations appears in Figure 3.48.
- 3 stages are used for instruction commit.

Instruction name	Pipeline stages	Repeat rate
Integer ALU	1	1
Integer load	3	1
Integer multiply	4	1
FP add	3	1
FP multiply	5	2
FP divide (64-bit)	32	32

FIGURE 3.48 The latency and repeat rate for common uops in the P6 microarchitecture. A repeat rate of 1 means that the unit is fully pipelined, and a repeat rate of 2 means that operations can start every other cycle.

Figure 3.49 shows a high-level picture of the pipeline, the throughput of each stage, and the capacity of buffers between stages. A stage will not achieve its throughput if either the input buffer cannot supply enough operands or the output buffer lacks capacity. In addition, internal restrictions or dynamic events (such as a cache miss) can cause a stall within all the units. For example, an instruction cache miss will prevent the instruction fetch stage from generating 16 bytes of instructions; similarly, three instructions can be decoded only under certain restrictions in how they map to uops.

Performance of the Pentium Pro Implementation

This section looks at some performance measurements for the Pentium Pro implementation. The Pentium Pro has the smallest set of primary caches among the P6 based microprocessors; it has, however, a high bandwidth interface to the secondary caches. Thus, while we would expect more performance to be lost to cache misses than on the Pentium II, the relatively faster and higher bandwidth secondary caches should reduce this effect somewhat. The measurements in this section use a 200 MHz Pentium Pro with a 256KB secondary cache and a 66 MHz main memory bus. The data for this section comes from a study by Bhandarkar and Ding [1997] that uses SPEC CPU95 as the benchmark set.

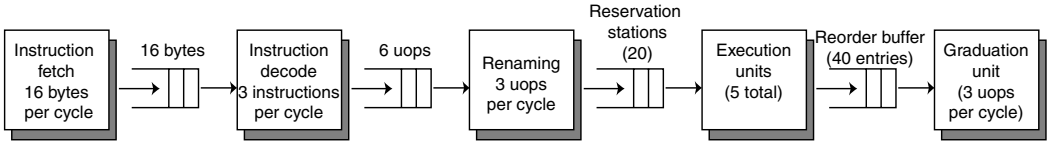


FIGURE 3.49 The P6 processor pipeline showing the throughput of each stage and the total buffering provided between stages. The buffering provided is either as bytes (before instruction decoding), as uops (after decoding and translation), as reservation station entries (after issue), or as reorder buffer entries (after execution). There are five execution units, each of which can potentially initiate a new uop every cycle (though some are not fully pipelined as shown in Figure 3.48). Recall that during renaming an instruction reserves a reorder buffer entry, so that stalls can occur during renaming/issue when the reorder buffer is full. Notice that the instruction fetch unit can fill the entire prefetch buffer in one cycle; if the buffer is partially full, fewer bytes will be fetched.

Understanding the performance of a dynamically-scheduled processor is complex. To see why, consider first that the actual CPI will be significantly greater than the ideal CPI, which in the case of the P6 architecture is 0.33. If the effective CPI is, for example, 0.66, then the processor can fall behind, achieving an CPI of 1, during some part of the execution and subsequently catch up by issuing and graduating two instructions per clock. Furthermore, consider how stalls actually occur in dynamically-scheduled, speculative processors. Since cache misses are overlapped, branches outcomes are speculated, and data dependences are dynamically scheduled around, what does a stall actually mean? In the limit, stalls occur when the processor fails to commit its full complement of instructions in a clock cycle.

Of course, the lack of instructions to complete means that somewhere earlier in the pipeline, some instructions failed to make progress (or in the limit, failed to even issue). This blockage can occur for a combination of several reasons in the Pentium Pro:

1. Less than a three IA-32 instructions could be fetched, due to instruction cache misses.
2. Less than three instructions could issue, because one of the three IA-32 instructions generated more than the allocated number of uops (4 for the first instruction and 1 for each of other two).
3. Not all the microoperations generated in a clock cycle could issue because of a shortage of reservation stations or reorder buffers.
4. A data dependence led to a stall because every reservations station or the reorder buffer was filled with instructions that are dependent.
5. A data cache misses led to a stall because every reservation station or the reorder buffer was filled with instructions waiting for a cache miss.

6. Branch mispredicts cause stalls directly, since the pipeline will need to be flushed and refilled. A mispredict can also cause a stall that arises from interference between speculated instructions that will be canceled and instructions that will be completed.

Because of the ability to overlap potential stall cycles from multiple sources, it is difficult to assign the cost of a stall cycle to any single cause. Instead, we will look at the contributions to stalls and conclude by showing that the actual CPI is less than what would be observed if no overlap of stalls were possible.

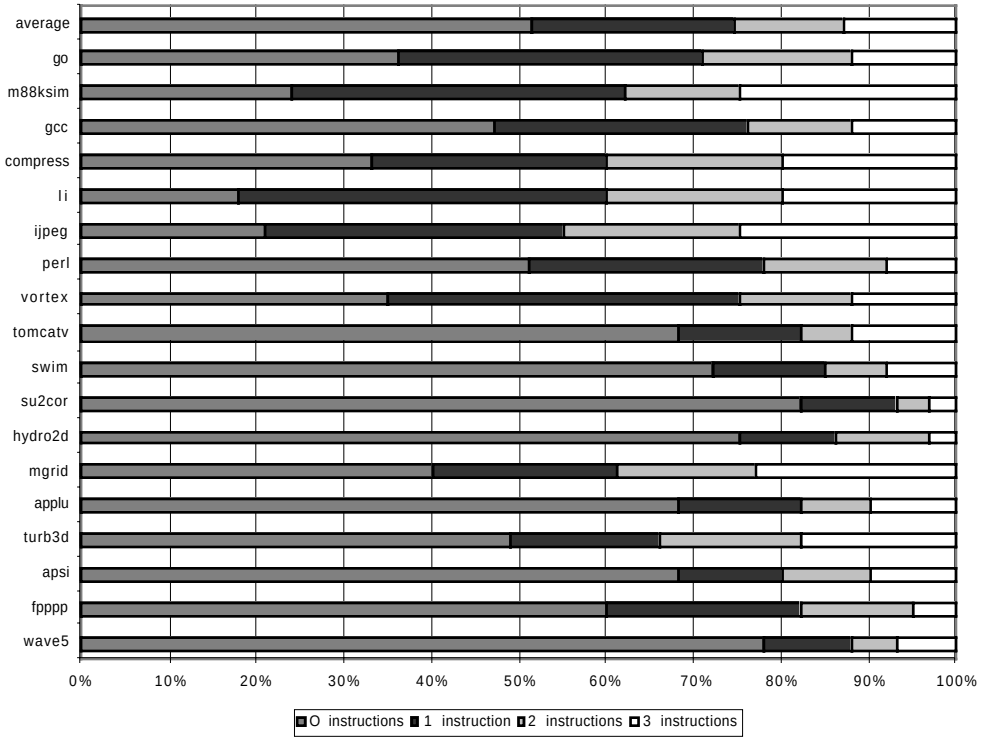


FIGURE 3.50 The number of instructions decoded each clock varies widely and depends upon a variety of facts including the instruction cache miss rate, the instruction decode rate, and the downstream execution rate. On average for these benchmarks, 0.87 instructions are decoded per cycle.

Stalls in the Decode Cycle

To start, let's look at the rate at which instructions are fetched and issued. Although the processor attempts to fetch three instructions every cycle, it cannot maintain this rate if the instruction cache generates a miss, if one of the instruc-

tions requires more than the number of microoperations available to it or if the six-entry uop issue buffer is full. Figure 3.50 shows the fraction of time in which 0, 1, 2, or 3 IA-32 instructions are decoded,

Figure 3.51 breaks out the stalls at decode time according to whether they are due to instruction cache stalls, which lead to less than three instructions available to decode, or resource capacity limitations, which means that a lack of reservation station or reorder buffers prevents a uop from issuing. Failure to issue a uop, eventually leads to a full uop buffer (recall that it has six entries), which then blocks instruction decode.

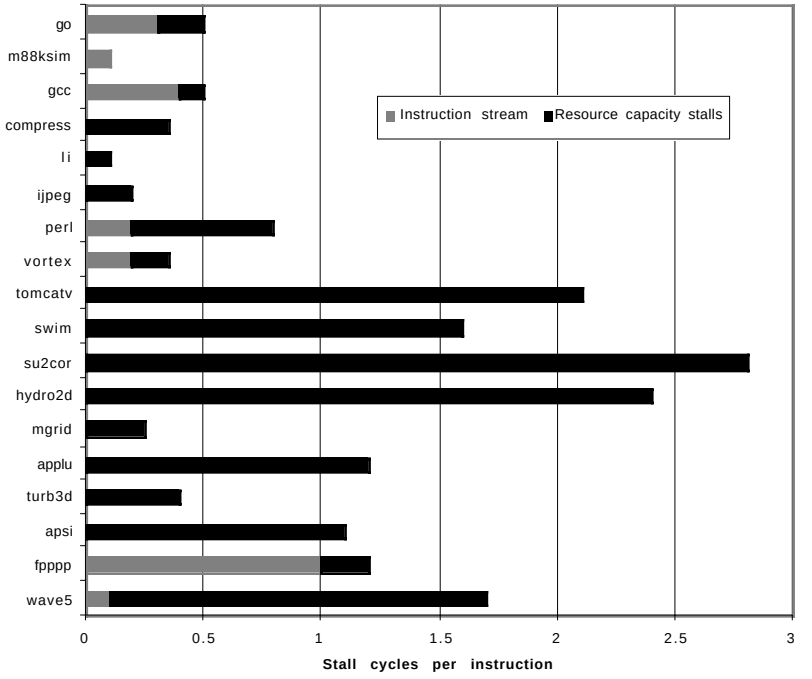


FIGURE 3.51 Stall cycles per instruction at decode time and the breakdown due to instruction stream stalls, which occur because of instruction cache misses, or resource capacity stalls, which occur because of a lack of reservation stations or reorder buffer entries. SPEC CPU95 is used as the benchmark suite, for this, and the rest of the measurements in this section.

The instruction cache miss rate for the SPEC95 FP benchmarks is small, and, for most of the FP benchmarks, resource capacity is the primary cause of decode stalls. The resource limitation arises because of lack of progress further down the pipeline, due either to large numbers of dependent operations or to long latency operations; the latter is a limitation for floating point programs, in particular. For example, the programs su2cor and hydro2d, which both have large numbers of resource stalls, also have long running, dependent floating-point calculations,

Another possible reason for the reduction in decode throughput could be that the expansion of IA-32 instructions into uops causes the uop buffer to fill. This would be the case if the number of uops per IA-32 instruction were large. Figure 3.52 shows, however, that most IA-32 instructions map to a single uop, and that on average there are 1.37 microoperations per IA-32 instruction (which means that the CPI for the processor is 1.37 times higher than the CPI of the microoperations). Surprisingly, the integer programs take slightly more microoperations per IA-32 instruction on average than the floating-point programs!

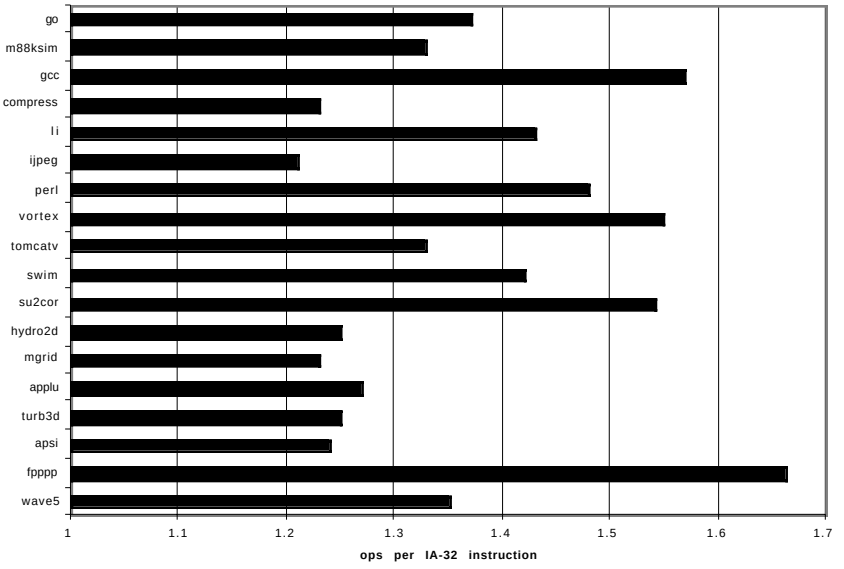


FIGURE 3.52 The number of microoperations per IA-32 instruction. Other than fpppp the integer programs typically require more uops. Most instructions will take only one uop, and, thus, the uop buffer fills primarily because of delays in the execution unit.

Data Cache Behavior

Figure 3.53 shows the number of first level (L1) and second level (L2) cache misses per thousand instructions. The L2 misses, although smaller in number, cost more than five times as much as L1 misses, and thus, dominate in some applications. Instruction cache misses are a minor effect in most of the programs. Although the speculative, out-of-order pipeline may be effective at hiding stalls

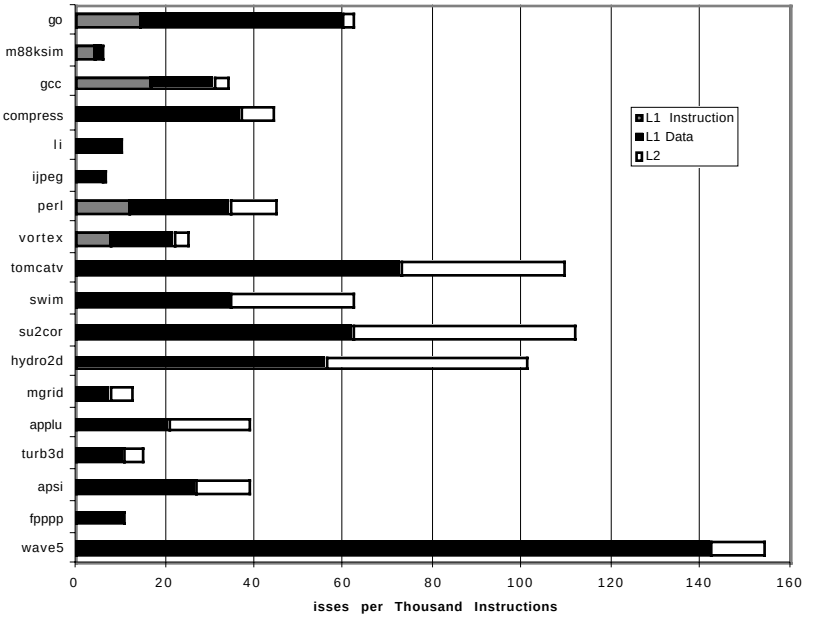


FIGURE 3.53 The number of misses per thousand instructions for the primary (L1) and secondary (L2) caches. Recall that the primary consists of a pair of 8KB caches and the secondary is 256KB. Because the cost of a secondary cache is about five times higher, the potential stalls from L2 cache misses are more serious than a simple frequency comparison would show.

due to L1 data misses, it cannot hide the long latency L2 cache misses, and L2 miss rates and effective CPI track similarly,

Branch Performance and Speculation Costs

Branch target addresses are predicted with a 512-entry BTB, based on the two-level adaptive scheme of Yeh and Patt, which is similar to the predictor described on page 258. If the BTB does not hit, a static prediction is used: backward branches are predicted taken (and have a one cycle penalty if correctly predicted) and forward branches are predicted not taken (and have no penalty if correctly predicted). Branch mispredicts have both a direct performance penalty, which is between 10-15 cycles, and an indirect penalty due to the overhead of incorrectly

speculated instructions, which is essentially impossible to measure. (Sometimes misspeculated instructions can result in a performance advantage, but this is likely to be rare.) Figure 3.54 shows the fraction of branches mispredicted either because of BTB misses or because of incorrect predictions. On average about 20% of the branches either miss or are mispredicted and use the simple static predictor rule.

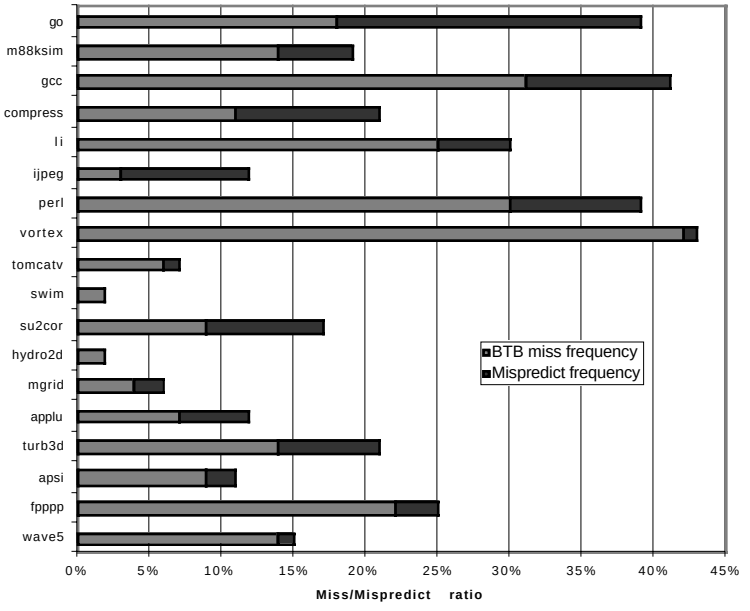


FIGURE 3.54 The BTB miss frequency dominates the mispredict frequency, arguing for a larger predictor, even at the cost of a slightly higher mispredict rate.

To understand the secondary effects arising from speculation that will be canceled, Figure 3.53 plots the average number of speculated uops that do not commit. On average about 1.2 times as many uops issue as commit. By factoring in the branch frequency and the mispredict rates, we find that, on average, each mispredicted branch issues 20 uops that will later be canceled. Unfortunately, accessing the exact costs of incorrectly speculated operations is virtually impossible, since they may cost nothing (if they do not block the progress of other instructions) or may be very costly.

Putting the Pieces Together: Overall Performance of the P6 Pipeline

Overall performance depends on the rate at which instructions actually complete and commit. Figure 3.56 shows the fraction of the time that 0, 1, 2, or 3 uops

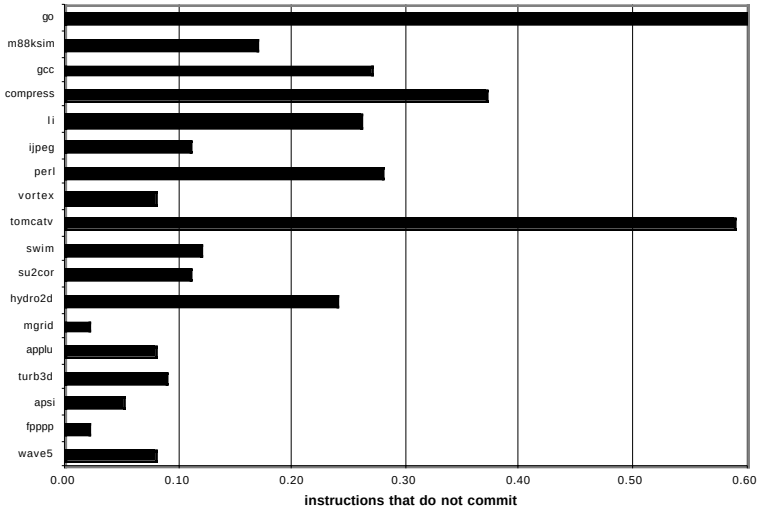


FIGURE 3.55 The "speculation factor" can be thought of as the fraction of issued instructions that do not commit. For the benchmarks with high speculation factors (> 30%), there are almost certainly some negative performance effects.

commit. On average, one uop commits per cycle, but, as Figure 3.56 shows, 23% of the time 3 uops commit in a cycle. This distribution demonstrates the ability of a dynamically-scheduled pipeline to fall behind (on 55% of the cycles, no uops commit) and later catch up (31% of the cycles have 2 or 3 uops committing).

Figure 3.57 sums up all the possible issue and stall cycles per IA-32 instruction and compares it to the actual measured CPI on the processor. The uop cycles in Figure 3.57 are the number of cycles per instruction assuming that the processor sustains three uops per cycle and accounting for the number of uops required per IA-32 instruction for that benchmark. The sum of the issue cycles plus stalls exceeds the actual measured CPI by an average of 1.37, varying from 1.0 to 1.75. This difference arises from the ability of the dynamically-scheduled pipeline to overlap and hide different classes of stalls arising in different types of programs. The average CPI is 1.15 for the SPECint programs and 2.0 for the SPECFP programs. The P6 microarchitecture is clearly designed to focus on integer programs.

The Pentium III versus the Pentium 4

The microarchitecture of the Pentium 4, which is called NetBurst, is similar to that of the Pentium III (called the P6 microarchitecture): both fetch up to three IA-32 instructions per cycle, decode them into micro-ops, and send the uops to an

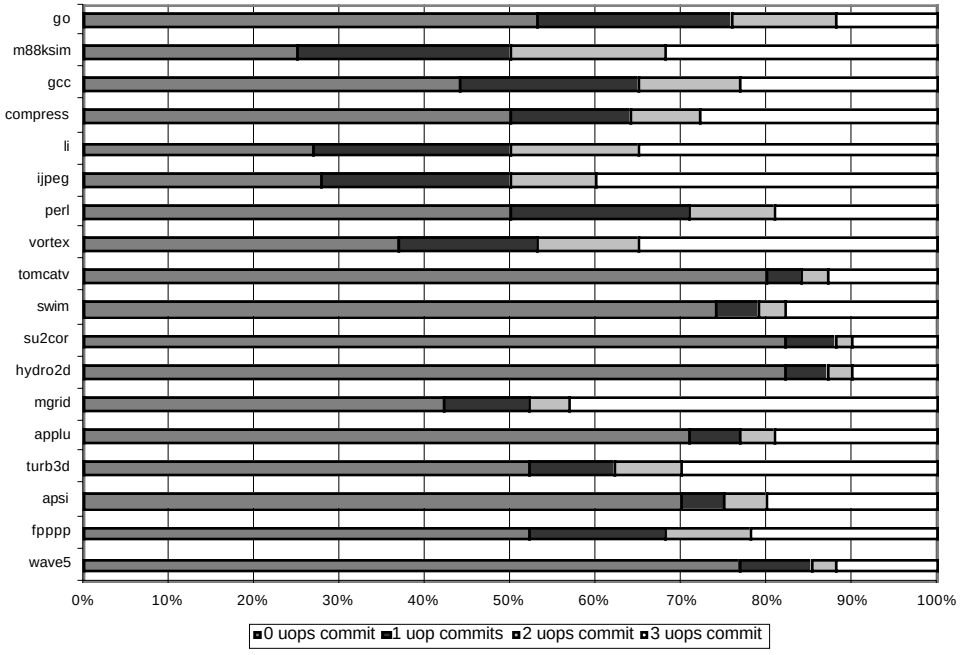


FIGURE 3.56 The breakdown in how often 0, 1, 2, or 3 uops commit in a cycle. The average number of uop completions per cycle is distributed as: 0 completions 55% of the cycles, 1 completion 13% of the cycles, 2 completions 8% of the cycles, and 3 completions 23% of the cycles,

out-of-order execution engine that can graduate up to three uops per cycle. There are, however, many differences that are designed to allow the NetBurst microarchitecture to operate at a significantly higher clock rate than the P6 microarchitecture and to help maintain or close the peak to sustained execution throughput. Among the most important of these are:

- A much deeper pipeline: P6 requires about 10 clock cycles from the time a simple add instruction is fetched until the availability of its results. In comparison, NetBurst takes about 20 cycles, including 2 cycles reserved simply to drive results across the chip!
- NetBurst uses register renaming (like the MIPS R10K and the Alpha 21264) rather than the reorder buffer, which is used in P6. Use of register renaming allows many more outstanding results (potentially up to 128) in NetBurst versus the 40 that are permitted in P6..
- There are seven integer execution units in NetBurst versus five in P6. The additions are an additional integer ALU and an additionaladdress computation unit.

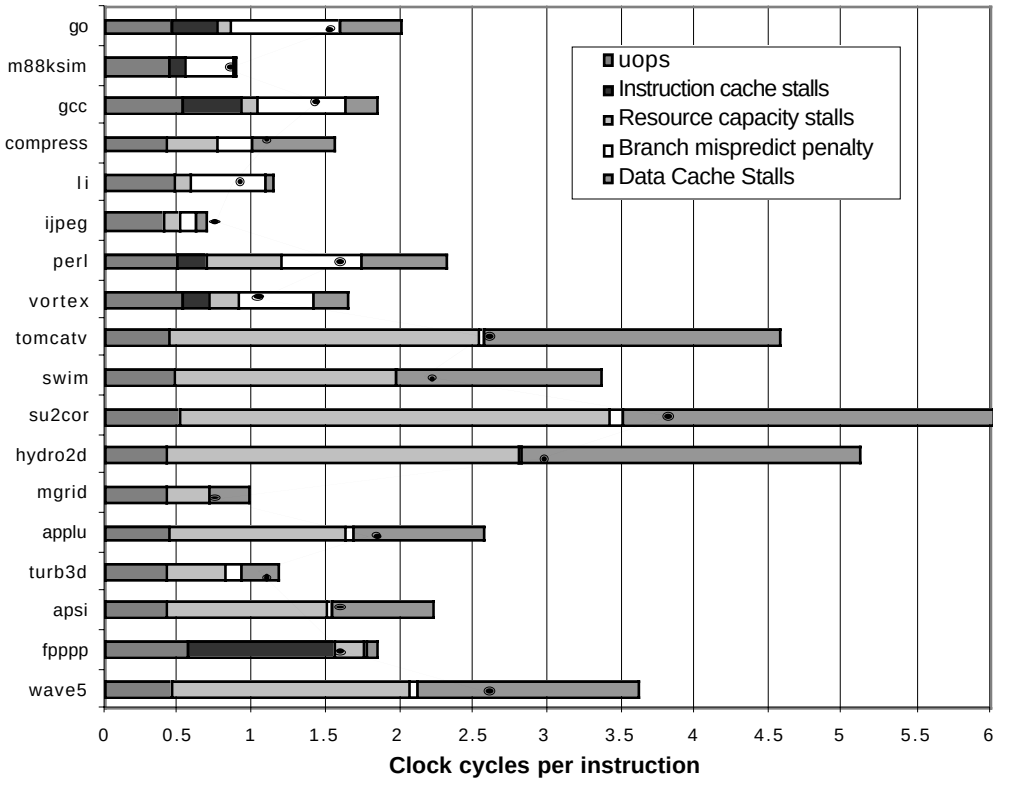


FIGURE 3.57 The actual CPI (shown as a line) is lower than the sum of the number of uop cycles plus all stalls. The uop cycles assume that three uops are completed every cycle and include the number of uops per instruction for the specific benchmark. All other stalls are the actual number of stall cycles. (TLB stalls that contribute less than 0.1 stalls/cycle are omitted). The overall CPI is lower than the sum of the uop cycles plus stalls through the use of dynamic scheduling.

- An aggressive ALU (operating at twice the clock rate) and an aggressive data cache lead to lower latencies for the basic ALU operations (effectively one-half a clock cycle in NetBurst versus one in P6) and for data loads (effectively two cycles in NetBurst versus three in P6). These high-speed functional units are critical to lowering the potential increase in stalls from the very deep pipeline.
- NetBurst uses a sophisticated trace cache (see Chapter 5) to improve instruction fetch performance, while P6 uses a conventional prefetch buffer and instruction cache.
- Netburst has a branch target buffer that is eight times larger and has an improved prediction algorithm.

- NetBurst has a level 1 data cache that is 8KB compared to P6's 16KB L1 data cache. NetBurst's larger level two cache (256KB) with higher bandwidth should offset this disadvantage.
- NetBurst implements the new SSE2 floating point instructions that allow two floating operations per instruction; these operations are structured as a 128-bit SIMD or short-vector structure. As we saw in Chapter 1 this gives Pentium 4 a considerable advantage over Pentium III on floating point code.

A Brief Performance Comparison of the Pentium III and Pentium 4

As we saw in Figure 1.28 on page 60 the Pentium 4 at 1.7 Ghz outperforms the Pentium III at 1 GHz by a factor of 1.26 for SPEC CINT2000 and 1.8 for SPEC CFP2000. Figure 3.58 shows the performance of the Pentium III and Pentium 4 on four of the SPEC benchmarks that are in both SPEC95 and SPEC2000. The floating point benchmarks clearly take advantage of the new instruction set extensions and yield an advantage of 1.6–1.7 above clock rate scaling.

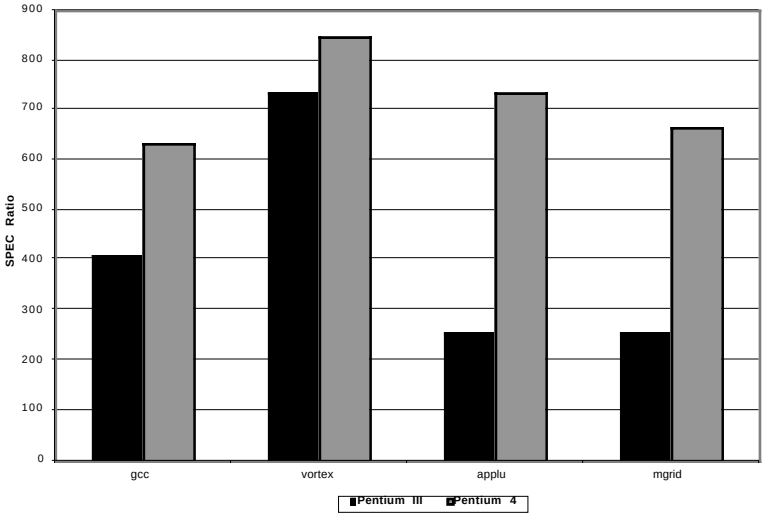


FIGURE 3.58 The performance of the Pentium 4 for four SPEC2000 benchmarks (two integer: gcc and vortex, and two floating point: apllu and mgrid) exceeds the Pentium III by a factor of between 1.2 and 2.9. This exceeds the purely clock speed advantage for the floating point benchmarks and is less than the clock speed advantage for the integer programs.

For the two integer benchmarks, the situation is somewhat different. In both cases the Pentium 4 delivers less than linear scaling with the increase in clock rate. If we assume the instruction counts are identical for integer codes on the two

processors, then the CPI for the two integer benchmarks is higher on the Pentium 4 (by a factor of 1.1 for gcc and a factor of 1.5 for vortex). Looking at the data for the Pentium Pro, we can see that the these benchmarks have relatively low level-2 miss rates and that they hide much of their level-1 miss penalty through dynamic scheduling and speculation. Thus, it is likely that the deeper pipeline and larger pipeline stall penalties on the Pentium 4 lead to a higher CPI for these two programs and reduce some of the gain from the high clock rate.

One interesting question is: why did the designers at Intel decide on the approach they took for the Pentium 4? On the surface, the alternative of doubling the issue rate of the Pentium III, as opposed to doubling the pipeline depth and the clock rate, looks at least as attractive. Of course, there are numerous changes between the two architectures, making an exact analysis of the tradeoffs difficult. Furthermore, because of the changes in the floating point instruction set, a comparison of the two pipeline organizations needs to focus on integer performance.

There are two sources of performance loss that arise if we compare the deeper pipeline of the Pentium 4 with that of the Pentium III. The first is the increase in clock overhead that occurs due to increased clock skew and jitter. This overhead is given by the difference between the ideal clock speed and the achieved clock speed. In comparable technologies, the Pentium 4 clock rate is between 1.7 and 1.8 times higher than the Pentium III clock rate. This range represents between 85% and 90% of the ideal clock rate, which is 2 times higher.

The second source of performance loss is the increase in CPI that arises from the deeper pipeline. We can estimate this by taking the ratio in clock rate versus the ratio in achieved overall performance. Using SPECInt as the performance measure and comparing a 1 GHz Pentium III to a 1.7 GHz Pentium 4, the performance ratio is 1.26. This tells us that the CPI for SPECInt on the Pentium 4 must be 1.7/1.26 = 1.34 times higher., or alternatively that the Pentium 4 is about 1.26/1.7 = 74% of the efficiency of the Pentium III. Of course, some of this loss is in the memory system, rather than in the pipeline.

The key question is whether doubling the issue width would result in a greater than 1.26 times overall performance gain. This is a vey difficult question to answer, since we must account for the improvement in pipeline CPI, the relative increase in cost of memory stalls, and the potential clock rate impact of a processor with twice the issue width. It is unlikely, looking at the data in Section 3.9, that doubling the issue rate will achieve better than a factor of 1.5 improvement in ideal instruction throughput. When combined with the potential impact on clock rate and the memory system costs, it appears that the choice of the Intel Pentium 4 designers to favor a deeper pipeline rather than wider issue, is at least a reasonable design choice.

3.11 Another View: Thread Level Parallelism

Throughout this chapter, our discussion has focused on exploiting parallelism in programs by finding and using the parallelism among instructions within the program. Although this approach has the great advantage that it is reasonably transparent to the programmer, as we have seen ILP can be quite limited or hard to exploit in some applications. Furthermore, there may be significant parallelism occurring naturally at a higher level in the application that cannot be exploited with the approaches discussed in this chapter. For example, an online transaction processing system has natural parallelism among the multiple queries and updates that are presented by requests. These queries and updates can be processed mostly in parallel, since they are largely independent of one another. Similarly, embedded applications often have natural high-level parallelism. For example, a processor in a network router can exploit parallelism among independent packets.

This higher level parallelism is called *thread level parallelism* because it is logically structured as separate threads of execution. A *thread* is a separate process with its own instructions and data. A thread may represent a process that is part of a parallel program consisting of multiple processes, or it may represent an independent program on its own. Each thread has all the state (instructions, data, PC, register state, and so on) necessary to allow it to execute. Unlike instruction level parallelism, which exploits implicit parallel operations within a loop or straight-line code segment, thread level parallelism is explicitly represented by the use of multiple threads of execution that are inherently parallel.

Thread level parallelism is important alternative to instruction level parallelism primarily because it could be more cost-effective to exploit than instruction level parallelism. There are many important applications where thread level parallelism occurs naturally, as it does in many server applications, In other cases, the software is being written from scratch and expressing the inherent parallelism is easy, as is true in some embedded applications. Chapter 6 explores multiprocessors and the support they provide for thread level parallelism.

The investment required to program applications to expose thread-level parallelism, makes it costly to switch the large established base of software to multiprocessors. This is especially true for desktop applications, where the natural paralelism that is present in many server environments, is harder to find. Thus, despite the potentially greater efficiency of exploiting thread-level parallelism, it is likely that ILP-based approaches will continue to be the primary focus for desktop-oriented processors.

3.12 Crosscutting Issues: Using an ILP Datapath to Exploit TLP

Thread-level and instruction-level parallelism exploit two different kinds of parallel structure in a program. One natural question to ask is whether it is possible for a processor oriented at instruction level parallelism to exploit thread level parallelism.

The motivation for this question comes from the observation that a datapath designed to exploit higher amounts of ILP, will find that functional units are often idle because of either stalls or dependences in the code. Could the parallelism among threads to be used a source of independent instructions that might be used to keep the processor busy during stalls? Could this thread-level parallelism be used to employ the functional units that would otherwise lie idle when insufficient ILP exists?

Multithreading, and a variant called *simultaneous multithreading*, take advantage of these insights by using thread level parallelism either as the primary form of parallelism exploitation–for example, on top of a simple pipelined processor–or as a method that works in conjunction with ILP mechanisms. In both cases, multiple threads are being executed with in single processor by duplicating the thread-specific state (program counter, registers, and so on.) and sharing the other processor resources by multiplexing them among the threads. Since multithreading is a method for exploiting thread level parallelism, we discuss it in more depth in Chapter 6.

3.13 Fallacies and Pitfalls

Our first fallacy is a two part one that indicates that simple rules do not hold and that the choice of benchmarks plays a major role.

Fallacies:

Processors with lower CPIs will always be faster.
Processors with faster clock rates will always be faster.

Although a lower CPI is certainly better, sophisticated pipelines typically have slower clock rates than processors with simple pipelines. In applications with limited ILP or where the parallelism cannot be exploited by the hardware resources, the faster clock rate often wins. But, when significant ILP exists, a processor that exploits lots of ILP may be better.

The IBM Power III processor is designed for high-performance FP and capable of sustaining four instructions per clock, including two FP and two load-store

instructions. It offers a 400 MHz clock rate in 2000, capable and achieves a SPEC CINT2000 peak rating of 249 and a SPEC CFP2000 peak rating of 344. The Pentium III has a comparably aggressive integer pipeline but has less aggressive FP units. An 800 MHz Pentium III in 2000 achieves a SPEC CINT 2000 peak rating of 344 and a SPEC CFP2000 peak rating of 237.

Thus, the faster clock rate of the Pentium III (800 MHz vs. 400 MHz) leads to an integer rating that is 1.38 times higher than the Power III, but the more aggressive FP pipeline of the Power III (and a better instruction set for floating point) leads to a lower CPI. If we assume comparable instruction counts, the Power III CPI must be almost 3x better than that of the Pentium III for the SPECFP 2000 benchmarks, leading to an overall performance advantage of 1.45.

Of course, this fallacy is nothing more than a restatement of a pitfall from Chapter 2 (see page XXX) about comparing processors using only one part of the performance equation.

Pitfall: Emphasizing an improvement in CPI by increasing issue rate while sacrificing clock rate can lead to lower performance.

The TI SuperSPARC design is a flexible multiple-issue processor capable of issuing up to three instructions per cycle. It had a 1994 clock rate of 60 MHz. The HP PA 7100 processor is a simple dual-issue processor (integer and FP combination) with a 99-MHz clock rate in 1994. The HP processor is faster on all the SPEC92 benchmarks except two of the integer benchmarks and one FP benchmark, as shown in Figure 3.59. On average, the two processors are close on integer, but the HP processor is about 1.5 times faster on the FP benchmarks. Of course, differences in compiler technology, detailed tradeoffs in the processor (including the cache size and memory organization), and the implementation technology, could all contribute to the performance differences.

The potential of multiple-issue techniques has caused *many* designers to focus on improving CPI while possibly not focusing adequately on the trade-off in cycle time incurred when implementing these sophisticated techniques. This inclination arises at least partially because it is easier with good simulation tools to evaluate the impact of enhancements that affect CPI than it is to evaluate the cycle time impact.

There are two factors that lead to this outcome. First, it is difficult to know the clock rate impact of an approach until the design is well underway, and then it may be too late to make large changes in the organization. Second, the design simulation tools available for determining and improving CPI are generally better than those available for determining and improving cycle time.

In understanding the complex interaction between cycle time and various organizational approaches, the experience of the designers seems to be one of the most valuable factors. With ever more complex designs, however, even the best designers find it hard to understand the complex tradeoffs between clock rate and other organizational decisions. At the end of Section 3.10, we will see the oppo-

site problem: how emphasizing a high clock rate, obtained through a deeper pipeline, can lead to degraded CPI and a lower performance gain than might be expected based sole on the higher clock rate.

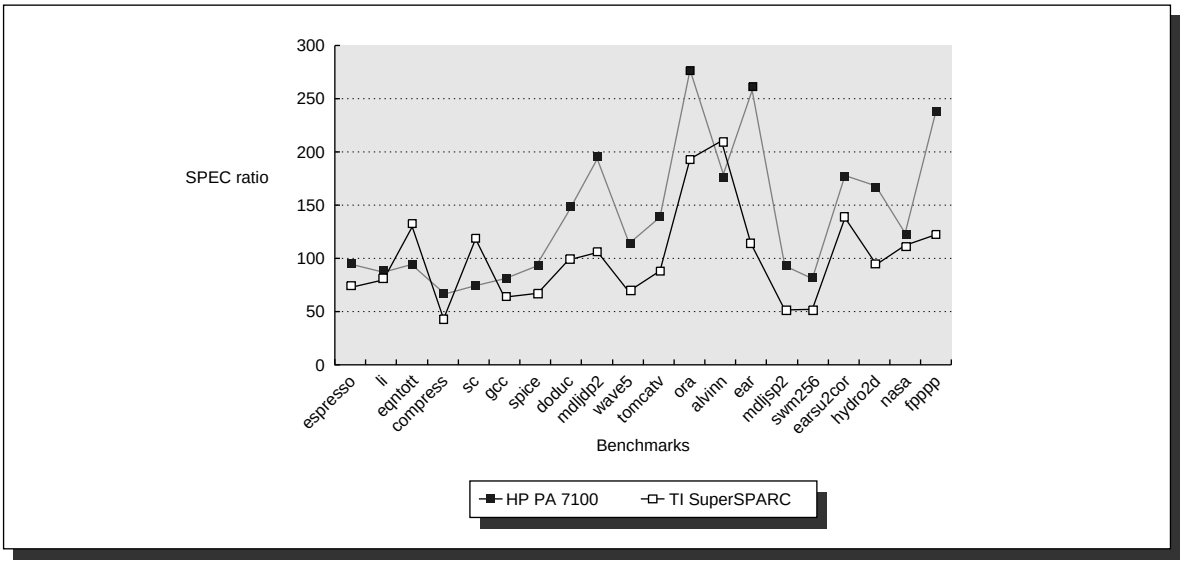


FIGURE 3.59 The performance of a 99-MHz HP PA 7100 processor versus a 60-MHz SuperSPARC. The comparison is based on 1994 measurements.

Pitfall: Improving only one aspect of a multiple-issue processor and expecting overall performance improvement.

This pitfall is simply a restatement of Amdahl's Law. A designer might simply look at a design, see a poor branch prediction mechanism and improve it, expecting to see significant performance improvements. The difficulty is that many factors limit the performance of multiple-issue machines, and improving one aspect of a processor often exposes some other aspect that previously did not limit performance.

We can see examples of this in the data on ILP. For example, looking just at the effect of branch prediction in Figure 3.39 on page 302, we can see that going from a standard two-bit predictor to a tournament predictor significantly improves the parallelism in espresso (from an issue rate of 7 to an issue rate of 12). If the processor provides only 32 registers for renaming, however, the amount of parallelism is limited to 5 issues per clock cycle, even with a branch prediction scheme better than either alternative.

Pitfalls: Sometimes bigger and dumber is better.

Advanced pipelines have focused on novel and increasingly sophisticated schemes for improving CPI. The 21264 uses a sophisticated tournament predictor with a total of 29 Kbits (see page 258), while the earlier 21164 uses a simple 2-bit predictor with 2K entries (or a total of 4 Kbits). For the SPEC95 benchmarks, the more sophisticated branch predictor of the 21264 outperforms the simpler 2-bit scheme on all but one benchmark. On average, for SPECInt95, the 21264 has 11.5 mispredictions per 1000 instructions committed while the 21164 has about 16.5 mispredictions.

Somewhat surprisingly, the simpler 2-bit scheme works better for the transaction processing workload than the sophisticated 21264 scheme (17 mispredictions vs. 19 per 1000 completed instructions)! How can a predictor with less than 1/7 the number of bits and a much simpler scheme actually work better? The answer lies in the structure of the workload. The transaction processing workload has a very large code size (more than an order of magnitude larger than any SPEC95 benchmark) with a large branch frequency. The ability of the 21164 predictor to hold twice as many branch predictions based on purely local behavior (2K vs. the 1K local predictor in the 21264) seems to provide a slight advantage.

This pitfall also reminds us that different applications can produce different behaviors. As processors become more sophisticated including specific microarchitectural features aimed at some particular program behavior, it is likely that different applications will see more divergent behavior.

3.14 Concluding Remarks

The tremendous interest in multiple-issue organizations came about because of an interest in improving performance without affecting the standard uniprocessor programming model. Although taking advantage of ILP is conceptually simple, the design problems are amazingly complex in practice. It is extremely difficult to achieve the performance you might expect from a simple first-level analysis.

The trade-offs between increasing clock speed and decreasing CPI through multiple issue are extremely hard to quantify. In the 1995 edition of this book, we stated: “Although you might expect that it is possible to build an advanced multiple-issue processor with a high clock rate, a factor of 1.5 to 2 in clock rate has consistently separated the highest clock rate processors and the most sophisticated multiple-issue processors. It is simply too early to tell whether this difference is due to fundamental implementation trade-offs, or to the difficulty of dealing with the complexities in multiple-issue processors, or simply a lack of experience in implementing such processors.”

Given the availability of the Alpha 21264 at 800 MHz, the Pentium III at 1.1 GHz, the AMD Athlon at 1.3 GHz, and the Pentium 4 at 2 GHz, it is clear that the limitation was primarily our understanding of how to build such processors. It is

also likely that this the first generation of CAD tools used for more than two million logic transistors was a limitation.

One insight that was clear in 1995 and remains clear in 2000 is that the peak to sustained performance ratios for multiple-issue processors are often quite large and typically grow as the issue rate grows. Thus, increasing the clock rate by *X* is almost always a better choice than increasing the issue width by *X*, though often the clock rate increase may rely largely on deeper pipelining, substantially narrowing the advantage. This insight probably played a role in motivating Intel to pursue a deeper pipeline for the Pentium 4, rather than trying to increase the issue width. Recall, however, the fundamental observation we made in Chapter 1 about the improvement in semiconductor technologies: the number of transistors available grows faster than the speed of the transistors. Thus, a strategy that focuses only on deeper pipelining may not be the best use of the technology in the long run..

Rather than embracing dramatic new approaches in microarchitecture, the last five years have focused on raising the clock rates of multiple issue machines and narrowing the gap between peak and sustained performance. The dynamically-scheduled, multiple-issue processors announced in the last two years (the Alpha 21264, the Pentium III and 4, and the AMD Athlon) have same basic structure and similar sustained issue rates (three to four instructions per clock) as the first dynamically-scheduled, multiple-issue processors announced in 1995! But, the clock rates are 4 to 8 times higher, the caches are 2 to 4 times bigger, there are 2 to 4 times as many renaming registers, and twice as many load/store units! The result is performance that is 6 to 10 times higher.

All the leading edge desktop and server processors are large, complex chips with more than 15 million transistors per processor. Notwithstanding, a simple two-way superscalar that issues FP instructions in parallel with integer instructions, or dual issues integer instructions (but not memory references) can probably be built with little impact on clock rate and with a tiny die size (in comparison to today's processes). Such a processor should perform well with a higher sustained to peak ratio than the high-end wide-issue processors and can be amazingly cost-effective. As a result, the high-end of the embedded space has recently moved to multiple-issue processors!

Whether approaches based primarily on faster clock rates, simpler hardware, and more static scheduling or approaches using more sophisticated hardware to achieve lower CPI will win out is difficult to say and may depend on the benchmarks.

Practical Limitations on Exploiting More ILP

Independent of the method used to exploit ILP, there are potential limitations that arise from employing more transistors. When the number of transistors employed is increased, the clock period is often determined by wire delays encountered both in distributing the clock and in the communication path of critical signals,

such as those that signal exceptions. These delays make it more difficult to employ increased numbers of transistors to exploit more ILP, while also increasing the clock rate. These problems are sometimes overcome by adding additional stages, which are reserved just for communicating signals across longer wires. The Pentium 4 doe this. These increased clock stages, however, can lead to more stalls and a higher CPI, since they increase pipeline latency. We saw exactly this phoenom when comparing the Pentium 4 to the Pentium III.

Although the limitations explored in Section 3.8 act as significant barriers to exploiting more ILP, it may be that more basic challenges would prevent the efficient exploitation of additional ILP, even if it could be uncovered. For example, doubling the issue rates above the current rates of four instructions per clock will probably require a processor to sustain three or four memory accesses per cycle and probably resolve two or three branches per cycle. In addition, supplying eight instructions per cycle will probably require fetching sixteen, speculating through multiple branches, and accessing roughly twenty registers per cycle. None of this is impossible, but whether it can be done while simultaneously maintaining clock rates exceeding 2 GHz is an open question and will surely be a significant challenge for any design team!

Equal in importance to the CPI versus clock rate trade-off, are realistic limitations on power. Recall that dynamic power is proportional to the product of the number of switching transistors and the switching rate. A microprocessor trying to achieve both a low CPI and a high CR fights both of these factors. Achieving an improved CPI means more instructions in flight and more transistors switching every clock cycle.

Two factors make it likely that the switching transistor count grows faster than performance. The first is the gap between peak issue rates and sustained performance, which continues to grow. Since the number of transistors switching is likely to be proportional to the peak issue rate and the performance is proportional to the sustained rate, the growing performance gap translates to increasing transistors switches per unit of performance. Second, issuing multiple instructions incurs some overhead in logic that grows as the issue rate grows. This logic is responsible for instruction issue analysis, including dependence checking, register renaming, and similar functions. The combined result is that, without voltage reductions to decrease power, lower CPIs are likely to lead to lower ratios of performance per watt.

A similar conundrum applies to attempts to increase clock rate. Of course, increasing the clock rate will increase transistor switching frequency and directly increase power consumption. As we saw in the Pentium 4 discussion, a deeper pipeline structure can be used to achieve a clock rate increase that exceeds what could be obtained just from improvements in transistor speed. Deeper pipelines, however, incur additional power penalties, resulting from several sources. The most important of these is the simple observation that a deeper pipeline means more operations are in flight every clock cycle, which means more transistors are switching, which means more power!

What is key to understand is the extent to which this potential growth in power caused by an increase in both the switching frequency and number of transistors switching is offset by a reduction in the operating voltage. Although these relationship is complex to understand, we can look at the results empirically and draw some conclusions.

The Pentium III and Pentium 4 provide an opportunity to examine this issue. As discussed on page 324, the Pentium 4 has a much deeper pipeline and can exploit more ILP than the Pentium III, although its basic peak issue rate is the same. The operating voltage of the Pentium 4 at 1.7 GHz is slightly higher than a 1 GHz Pentium III: 1.75V versus 1.70V. The power difference, however, is much larger: the 1.7 GHz Pentium 4 consumes 64 W typical, while the 1 GHz Pentium III consumes only 30 W by comparison. Figure 3.60 shows the effective performance of a 1.7 GHz Pentium 4 per watt relative to the performance per watt of a 1 GHz Pentium III using the same benchmarks presented in Figure 1.28 on page 60. Clearly, while the Pentium 4 is faster, its higher clock rate, deeper pipeline and higher sustained execution rate, make it significantly less power efficient. Whether the decreased power efficiency between the Pentium III and Pentium 4 are deep issues and unlikely to be overcome, or to whether they are artifacts of the two implementations is a key question that will probably be settled in future implementations. What is clear is that neither deeper pipelines nor wider issue rates can circumvent the need to consume more power to improve performance.

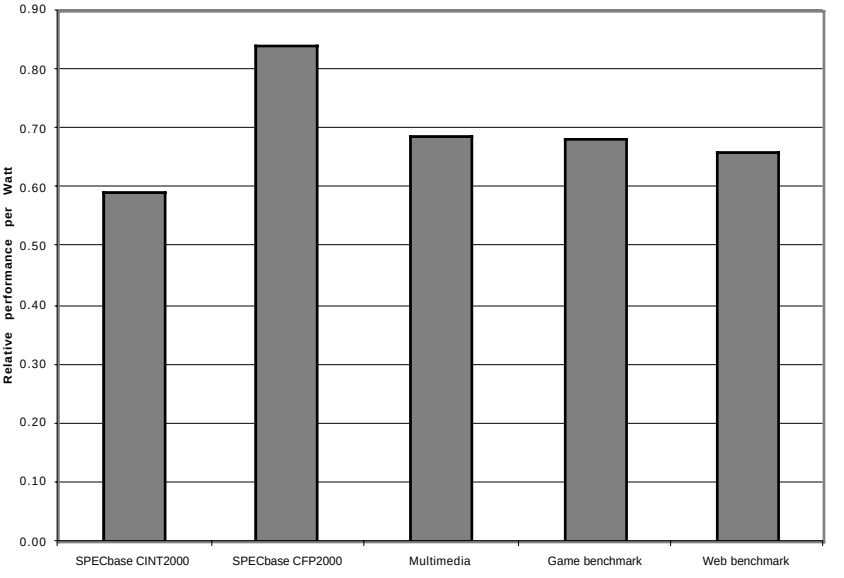


FIGURE 3.60 The relative performance per Watt of the Pentium 4 is 15% to 40% less than the Pentium III on these five sets of benchmarks.

More generally, the question of how best to exploit parallelism remains open. Clearly ILP will continue to play a big role because of its smaller impact on programmers and applications when compared to an explicitly parallel model using multiple threads and parallel processors. What sort of parallelism computer architects will employ as they try to achieve higher performance levels, and what type of parallelism programmers will accept are hard to predict. Likewise, it is unclear whether vectors will play a larger role in processors designed for multimedia and DSP applications, or whether such processors will rely on limited SIMD and ILP approaches. We will return to these questions in the next chapter as well as in Chapter 6.

3.15 Historical Perspective and References

This section describes some of the major advances in dynamically scheduled pipelines and ends with some of the recent literature on multiple-issue processors. Ideas such as data flow computation derived from observations that programs were limited by data dependence. The history of basic pipelining and the CDC 6600, the first dynamically scheduled processor, are contained in Appendix A.

The IBM 360 Model 91: A Landmark Computer

The IBM 360/91 introduced many new concepts, including tagging of data, register renaming, dynamic detection of memory hazards, and generalized forwarding. Tomasulo's algorithm is described in his 1967 paper. Anderson, Sparacio, and Tomasulo [1967] describe other aspects of the processor, including the use of branch prediction. Many of the ideas in the 360/91 faded from use for nearly 25 years before being broadly resurrected in the 1990s. Unfortunately, the 360/91 was not successful and only a handful were sold. The complexity of the design made it late to the market and allowed the Model 85, which was the first IBM processor with a cache, to outperform the 91.

Branch Prediction Schemes

The two-bit dynamic hardware branch prediction scheme was described by J. E. Smith [1981]. Ditzel and McLellan [1987] describe a novel branch-target buffer for CRISP, which implements branch folding. McFarling and Hennessy [1986] did a quantitative comparison of a variety of compile-time and runtime branch prediction schemes. The correlating predictor we examine was described by Pan, So, and Rameh [1992]. Yeh and Patt [1992, 1993] generalized the correlation idea and described multilevel predictors that use branch histories for each branch, similar to the local history predictor used in the 21264. McFarling's tournament prediction scheme, which he refers to a combined predictor, is described in his

1993 technical report. There are a variety of more recent papers on branch prediction based on variations in the multilevel and correlating predictor ideas. Kaeli and Emma [1991] describe return address prediction.

The Development of Multiple-Issue Processors

The concept of multiple-issue designs has been around for a while, though much of the work in the 1970s focused on statically scheduled approaches, which we discuss in the next chapter. IBM did pioneering work on multiple issue. In the 1960s, a project called ACS was underway in California. It included multiple-issue concepts, a proposal for dynamic scheduling (although with a simpler mechanism than Tomasulo's scheme, which used back-up registers), and fetching down both branch paths. The project originally started as a new architecture to follow Stretch and surpass the CDC 6600/6800. ACS started in New York but was moved to California, later changed to be S/360 compatible, and eventually canceled. John Cocke was one of the intellectual forces behind the team that included a number of IBM veterans and younger contributors many of whom went on to other important roles in IBM and elsewhere: Jack Bertram, Ed Sussenguth, Gene Amdahl, Herb Schorr, Fran Allen, Lynn Conway, and Phil Dauber among others. While the compiler team published many of their ideas and had great influence outside IBM, the architecture ideas were not widely disseminated at that time. The most complete accessible documentation of this important project is at: http://www.cs.clemson.edu/~mark/acs.html, which includes interviews with the ACS veterans and pointers to other sources. Sussenguth [1999] is a good overview of ACS.

More than 10 years after ACS was cancelled, John Cocke made a new proposal for a superscalar processor that dynamically made issue decisions; he described the key ideas in several talks in the mid 1980s and coined the name *superscalar*. He called the design America; it is described by Agerwala and Cocke [1987]. The IBM Power-1 architecture (the RS/6000 line) is based on these ideas (see Bakoglu et al. [1989]).

J. E. Smith [1984] and his colleagues at Wisconsin proposed the decoupled approach that included multiple issue with limited dynamic pipeline scheduling. A key feature of this processor is the use of queues to maintain order among a class of instructions (such as memory references) while allowing it to slip behind or ahead of another class of instructions. The Astronautics ZS-1 described by Smith et al. [1987] embodies this approach with queues to connect the load-store unit and the operation units. The Power-2 design uses queues in a similar fashion. J. E. Smith [1989] also describes the advantages of dynamic scheduling and compares that approach to static scheduling.

The concept of speculation has its roots in the original 360/91, which performed a very limited form of speculation. The approach used in recent processors combines the dynamic scheduling techniques of the 360/91 with a buffer to allow in-order commit. J. E. Smith and Pleszkun [1988] explored the use of buff-

ering to maintain precise interrupts and described the concept of a reorder buffer. Sohi [1990] describes adding renaming and dynamic scheduling, making it possible to use the mechanism for speculation. Patt and his colleagues were early proponents of aggressive reordering and speculation. They focused on checkpoint and restart mechanisms and pioneered an approach called HPSm, which is also an extension of Tomasulo's algorithm [Hwu and Patt 1986].

The use of speculation as a technique in multiple-issue processors was evaluated by Smith, Johnson, and Horowitz [1989] using the reorder buffer technique; their goal was to study available ILP in nonscientific code using speculation and multiple issue. In a subsequent book, M. Johnson [1990] describes the design of a speculative superscalar processor. Johnson later led the AMD K-5 design, one of the first speculative superscalars.

Studies of ILP and Ideas to Increase ILP

A series of early papers, including Tjaden and Flynn [1970] and Riseman and Foster [1972], concluded that only small amounts of parallelism could be available at the instruction level without investing an enormous amount of hardware. These papers dampened the appeal of multiple instruction issue for more than ten years. Nicolau and Fisher [1984] published a paper based on their work with trace scheduling and asserted the presence of large amounts of potential ILP in scientific programs.

Since then there have been many studies of the available ILP. Such studies have been criticized since they presume some level of both hardware support and compiler technology. Nonetheless, the studies are useful to set expectations as well as to understand the sources of the limitations. Wall has participated in several such studies, including Jouppi and Wall [1989], Wall [1991], and Wall [1993]. Although the early studies were criticized as being conservative (e.g., they didn't include speculation), the last study is by far the most ambitious study of ILP to date and the basis for the data in section 3.10. Sohi and Vajapeyam [1989] give measurements of available parallelism for wide-instruction-word processors. Smith, Johnson, and Horowitz [1989] also used a speculative superscalar processor to study ILP limits. At the time of their study, they anticipated that the processor they specified was an upper bound on reasonable designs. Recent and upcoming processors, however, are likely to be at least as ambitious as their processor.

Lam and Wilson [1992] looked at the limitations imposed by speculation and shown that additional gains are possible by allowing processors to speculate in multiple directions, which requires more than one PC. (Such schemes cannot exceed what perfect speculation accomplishes, but they help close the gap between realistic prediction schemes and perfect prediction.) Wall's 1993 study includes a limited evaluation of this approach (up to 8 branches are explored).

Going Beyond the Data Flow Limit

One other approach that has been explored in the literature is the use of value prediction. Value prediction can allow speculation based on data values. There have been a number of studies of the use of value prediction. Lipasti and Shen published two papers in 1996 evaluating the concept of value prediction and its potential impact on ILP exploitation. Sodani and Sohi [1997] approaches the same problem from the viewpoint of reusing the values produced by instructions. Moshovos, Breach, Vijaykumar and Sohi [1997] show that by deciding when to speculate on values, by tracking whether such speculation has been accurate in the past, is important to achieving performance gains with value speculation. Moshovos and Sohi [1997] and Chrysos and Emer [1998] focus on predicting memory dependences and using this information to eliminate the dependence through memory. Gonzalez and Gozalez [1998], Babbay and Mendelson [1998], and Calder, Reinman and Tullsen [1999] are more recent studies of the use of value prediction. This area is currently highly active with new results being published in every conference.

Recent Advanced Microprocessors

The years 1994–95 saw the announcement of wide superscalar processors (3 or more issues per clock) by every major processor vendor: Intel Pentium Pro and Pentium II (these processors share the same core pipeline architecture, described by Cowell and Steck [1995]), AMD K5, K6, and Althon, Sun UltraSPARC (see Lauterbach and Horel [1999]), Alpha 21164 (see Edmonston et. al [1995]) and 21264 (see Kessler [2000]), MIPS R10000 and R12000 (see Yeager [1996]), PowerPC 603, 604, 620 (see Diep, Nelson, and Shen [1995]), and HP 8000 (Kumar [1997]). The latter part of the decade (1996-2000), saw second generations of much of these processors (Pentium III, AMD Athlon, Alpha 21264, among others). The second generation, although similar in issue rate, could sustain a lower CPI, provided much higher clock rates, all included dynamic scheduling, and almost universally supported speculation. In practice, many factors, including the implementation technology, the memory hierarchy, the skill of the designers, and the type of applications benchmarked, all play a role in determining which approach is best. Figure 3.61 shows the most interesting processors of the past five years, their characteristics.

Processor	System ship	Max. current CR (MHz)	Power (W)	Transistors (M)	Window size	Rename registers (int/FP)	Issue rate: Maximum/ Memory / Integer / FP / Branch	Branch Predict Buffer	Pipe-stages (int/ load)
MIPS R14000	2000	400	25	7	48	32/32	4/1/2/2/1	2K x 2	6
Ultra SPARC III	2001	900	65	29	N.A.	None	4/1/4/3/1	16K x 2	14/15
Pentium III	2000	1000	30	24	40	Total: 40	3/2/2/1/1	512 entries	12/14
Pentium 4	2001	1700	64	42	126	Total:128	3/2/3/2/1	4K x 2	22/24
HP PA 8600	2001	552	60	130	56	Total: 56	4/2/2/2/1	2K x 2	7/9
Alpha 21264B	2001	833	75	15	80	41/41	4/2/4/2/1	multilevel (see p. 258)	7/9
Power PC 7400 (G4)	2000	450	5	7	5	6/6	3/1/2/1/1	512 x 2	4/5
AMD Athlon	2001	1330	76	37	72	36/36	3/2/3/3/1	4K x 9	9/11
IBM Power 3-II	2000	450	36	23	32	16/24	4/2/2/2/2	2K x 2	7/8

FIGURE 3.61 Recent high-performance processors and their characteristics. The window size column shows the size of the buffer available for instructions, and, hence, the maximum number of instructions in flight. Both the Pentium III and the Althon schedule microoperations and the window is the maximum number of microoperations in execution. The IBM, HP, and UltraSPARC processors support dynamic issue, but not speculation. To read more about these processors the following references are useful: *IBM Journal of Research and Development* (contains issues on Power and PowerPC designs), the *Digital Technical Journal* (contains issues on various Alpha processors), and *Proceedings of the Hot Chips Symposium* (annual meeting at Stanford, which reviews the newest microprocessors), the International Solid State Circuits Conference, and the annual Microprocessor Forum meetings, and the annual International Symposium on Computer Architecture. Much of this data in this table came from Microprocessor Report online April 30, 2001.

References

AGERWALA, T. AND J. COCKE [1987]. "High performance reduced instruction set processors," IBM Tech. Rep. (March).

ANDERSON, D. W., F. J. SPARACIO, AND R. M. TOMASULO [1967]. "The IBM 360 Model 91: Processor philosophy and instruction handling," *IBM J. Research and Development* 11:1 (January), 8–24.

AUSTIN, T. M. AND G. SOHI [1992]. "Dynamic dependency analysis of ordinary programs," *Proc. 19th Symposium on Computer Architecture* (May), Gold Coast, Australia, 342-351.

BABBAY F. AND A. MENDELSON [1998]. "Using Value Prediction to Increase the Power of Speculative Execution Hardware." ACM Transactions on Computer Systems, vol. 16, No. 3 (August), pages

234-270.

BAKOGLU, H. B., G. F. GROHOSKI, L. E. THATCHER, J. A. KAELI, C. R. MOORE, D. P. TATTLE, W. E. MALE, W. R. HARDELL, D. A. HICKS, M. NGUYEN PHU, R. K. MONTOYE, W. T. GLOVER, AND S. DHAWAN [1989]. "IBM second-generation RISC processor organization," *Proc. Int'l Conf. on Computer Design*, IEEE (October), Rye, N.Y., 138–142.

BHANDARKAR, D. AND D. W. CLARK [1991]. "Performance from architecture: Comparing a RISC and a CISC with similar hardware organizations," *Proc. Fourth Conf. on Architectural Support for Programming Languages and Operating Systems*, IEEE/ACM (April), Palo Alto, Calif., 310–319.

BHANDARKAR, D. AND J. DING [1997]. "Performance Characterization of the Pentium Pro Processor," *Proc. Third International Sym. on High Performance Computer Architecture*, IEEE, (February), San Antonio, 288-297.

BLOCH, E. [1959]. "The engineering design of the Stretch computer," *Proc. Fall Joint Computer Conf.*, 48–59.

BUCHOLTZ, W. [1962]. *Planning a Computer System: Project Stretch*, McGraw-Hill, New York.

CALDER, B., REINMAN, G. AND D. TULLSEN[1999] "Selective Value Prediction". Proc. 26th International Symposium on Computer Architecture (ISCA), Atlanta, June

CHEN, T. C. [1980]. "Overlap and parallel processing," in *Introduction to Computer Architecture*, H. Stone, ed., Science Research Associates, Chicago, 427–486.

CHRYSOS, G.Z. AND J.S. EMER [1998]. "Memory Dependence Prediction using Store Sets". Proc. 25th Int. Symposium on Computer Architecture (ISCA), June, Barcelona, 142-153.

CLARK, D. W. [1987]. "Pipelining and performance in the VAX 8800 processor," *Proc. Second Conf. on Architectural Support for Programming Languages and Operating Systems*, IEEE/ACM (March), Palo Alto, Calif., 173–177.

COLWELL R. P. AND R. STECK [1995]. "A 0.6um BiCMOS process with Dynamic Execution." Proceedings of Int. Sym. on Solid State Circuits.

CVETANOVIC, Z. AND R.E. KESSLER [2000]. "Performance Analysis of the Alpha 21264-based Compaq ES40 System," *Proc. 27th Symposium on Computer Architecture* (June), Vancouver, Canada., 192-202.

DAVIDSON, E. S. [1971]. "The design and control of pipelined function generators," *Proc. Conf. on Systems, Networks, and Computers*, IEEE (January), Oaxtepec, Mexico, 19–21.

DAVIDSON, E. S., A. T. THOMAS, L. E. SHAR, AND J. H. PATEL [1975]. "Effective control for pipelined processors," *COMPCON, IEEE* (March), San Francisco, 181–184.

DIEP, T. A., C. NELSON, AND J. P. SHEN [1995]. "Performance evaluation of the PowerPC 620 microarchitecture," *Proc. 22th Symposium on Computer Architecture* (June), Santa Margherita, Italy.

DITZEL, D. R. AND H. R. MCLELLAN [1987]. "Branch folding in the CRISP microprocessor: Reducing the branch delay to zero," *Proc. 14th Symposium on Computer Architecture* (June), Pittsburgh, 2–7.

EMER, J. S. AND D. W. CLARK [1984]. "A characterization of processor performance in the VAX-11/780," *Proc. 11th Symposium on Computer Architecture* (June), Ann Arbor, Mich., 301–310.

EDMONDSON, J.H., RUBINFIELD, P.I., PRESTON, R., AND V. RAJAGOPALAN [1995]. "Superscalar Instruction Execution in the 21164 Alpha Microprocessor", IEEE Micro, Vol. 15, 2. 33–43.

FOSTER, C. C. AND E. M. RISEMAN [1972]. "Percolation of code to enhance parallel dispatching and execution," *IEEE Trans. on Computers* C-21:12 (December), 1411–1415.

J. GONZÁLEZ AND A. GONZÁLEZ [1998], "Limits of Instruction Level Parallelism with Data Speculation", in Proc. of the VECPAR Conf., pp. 585-598.

HEINRICH, J. [1993]. *MIPS R4000 User's Manual*, Prentice Hall, Englewood Cliffs, N.J.

HWU, W.-M. AND Y. PATT [1986]. “HPSm, a high performance restricted data flow architecture having minimum functionality,” *Proc. 13th Symposium on Computer Architecture* (June), Tokyo, 297–307.

IBM [1990]. “The IBM RISC System/6000 processor” (collection of papers), *IBM J. of Research and Development* 34:1 (January).

JORDAN, .H.F. [1983] “Performance measurements on HEP: A pipelined MIMD computer,” Proc. 10th *Symposium on Computer Architecture* (June), pp. 207--212.

JOHNSON, M. [1990]. *Superscalar Microprocessor Design*, Prentice Hall, Englewood Cliffs, N.J.

JOUPPI, N. P. AND D. W. WALL [1989]. “Available instruction-level parallelism for superscalar and superpipelined processors,” *Proc. Third Conf. on Architectural Support for Programming Languages and Operating Systems*, IEEE/ACM (April), Boston, 272–282.

KAELI, D.R. AND P.G. EMMA [1991]. “Branch History Table Prediction of Moving Target Branches Due to Subroutine Returns, Proc. 18th Int. Sym. on Computer Architecture (ISCA), Toronto, May, 34-42.

KELLER R. M. [1975]. “Look-ahead processors,” *ACM Computing Surveys* 7:4 (December), 177–195.

KESSLER. R. [1999]. “The Alpha 21264 microprocessor,” IEEE Micro, 19(2) (March/April):pp 24--36.

KILLIAN, E. [1991]. “MIPS R4000 technical overview–64 bits/100 MHz or bust,” *Hot Chips III Symposium Record* (August), Stanford University, 1.6–1.19.

KOGGE, P. M. [1981]. *The Architecture of Pipelined Computers*, McGraw-Hill, New York.

KUMAR , A. [1997]. “The HP PA-8000 RISC CPU, “ IEEE Micro, Vol. 17, No. 2 (March/April).

KUNKEL, S. R. AND J. E. SMITH [1986]. “Optimal pipelining in supercomputers,” *Proc. 13th Symposium on Computer Architecture* (June), Tokyo, 404–414.

LAM, M. S. AND R. P. WILSON [1992]. “Limits of control flow on parallelism,” *Proc. 19th Symposium on Computer Architecture* (May), Gold Coast, Australia, 46–57.

LAUTERBACH G. AND HOREL, T. [1999]. “UltraSPARC-III: Designing Third Generation 64-Bit Performance, “ IEEE Micro, Vol. 19, No. 3 (May/June).

LIPASTI, M.H., WILKERSON, C.B., AND J.P. SHEN [1996]. "Value Locality and Load Value Prediction". *Proc. Seventh Symposium on Architectural Support for Programming Languages and Operating Systems* (October), pp. 138-147.

LIPASTI, M.H. AND J. P. SHEN [1996]. Exceeding the Dataflow Limit via Value Prediction. Proc. of the 29 th Annual ACM/IEEE International Symposium on Microarchitecture (December), .

MCFARLING, S. [1993] “Combining branch predictors,” WRL Technical Note TN-36 (June), Digital Western Research Laboratory, Palo Alto, Calif.

MOSHOVOS, A.AND G.S. SOHI [1997] “Streamlining Inter-operation Memory Communication via Data Dependence Prediction”. Proc. 30th Annual Int. Sym on Microarchitecture (MICRO-30), Dec, 235-245.

MOSHOVOS, A. BREACH, S, VIJAYKUMAR, T.N. AND G.S. SOHI [1997] “Dynamic Speculation and Synchronization of Data Dependences”. Proc. 24th Int. Sym. on Computer Architecture (ISCA), June,Boulder.

NICOLAU, A. AND J. A. FISHER [1984]. “Measuring the parallelism available for very long instruction word architectures,” *IEEE Trans. on Computers* C-33:11 (November), 968–976.

PAN, S.-T., K. SO, AND J. T. RAMEH [1992]. “Improving the accuracy of dynamic branch prediction using branch correlation,” *Proc. Fifth Conf. on Architectural Support for Programming Languages and Operating Systems*, IEEE/ACM (October), Boston, 76-84.

Postiff, M.A.; Greene, D.A.; Tyson, G.S.; Mudge, T.N. “The limits of instruction level parallelism in SPEC95 Applications” . Computer Architecture News, vol.27, (no.1), ACM, March 1999. p.31-40.

RAMAMOORTHY, C. V. AND H. F. LI [1977]. “Pipeline architecture,” *ACM Computing Surveys* 9:1 (March), 61–102.

RISEMAN, E. M. AND C. C. FOSTER [1972]. “Percolation of code to enhance parallel dispatching and execution,” *IEEE Trans. on Computers* C-21:12 (December), 1411–1415.

RYMARCZYK, J. [1982]. “Coding guidelines for pipelined processors,” *Proc. Symposium on Architectural Support for Programming Languages and Operating Systems*, IEEE/ACM (March), Palo Alto, Calif., 12–19.

SITES, R. [1979]. *Instruction Ordering for the CRAY-1 Computer*, Tech. Rep. 78-CS-023 (July), Dept. of Computer Science, Univ. of Calif., San Diego.

SMITH, A. AND J. LEE [1984]. “Branch prediction strategies and branch-target buffer design,” *Computer* 17:1 (January), 6–22.

SMITH, J. E. AND A. R. PLESZKUN [1988]. “Implementing precise interrupts in pipelined processors,” *IEEE Trans. on Computers* 37:5 (May), 562–573.

SMITH, J. E. [1981]. “A study of branch prediction strategies,” *Proc. Eighth Symposium on Computer Architecture* (May), Minneapolis, 135–148.

SMITH, J. E. [1984]. “Decoupled access/execute computer architectures,” *ACM Trans. on Computer Systems* 2:4 (November), 289–308.

SMITH, J. E. [1989]. “Dynamic instruction scheduling and the Astronautics ZS-1,” *Computer* 22:7 (July), 21–35.

SMITH, J. E. AND A. R. PLESZKUN [1988]. “Implementing precise interrupts in pipelined processors,” *IEEE Trans. on Computers* 37:5 (May), 562–573. This paper is based on an earlier paper that appeared in *Proc. 12th Symposium on Computer Architecture*, June 1988.

SMITH, J. E., G. E. DERMER, B. D. VANDERWARN, S. D. KLINGER, C. M. ROZEWSKI, D. L. FOWLER, K. R. SCIDMORE, AND J. P. LAUDON [1987]. “The ZS-1 central processor,” *Proc. Second Conf. on Architectural Support for Programming Languages and Operating Systems*, IEEE/ACM (March), Palo Alto, Calif., 199–204.

SMITH, M. D., M. HOROWITZ, AND M. S. LAM [1992]. “Efficient superscalar performance through boosting,” *Proc. Fifth Conf. on Architectural Support for Programming Languages and Operating Systems* (October), Boston, IEEE/ACM, 248–259.

SMITH, M. D., M. JOHNSON, AND M. A. HOROWITZ [1989]. “Limits on multiple instruction issue,” *Proc. Third Conf. on Architectural Support for Programming Languages and Operating Systems*, IEEE/ACM (April), Boston, 290–302.

SODANI, A. AND G. SOHI [1997]. "Dynamic Instruction Reuse", Proc. of the 24th Int. Symp. on Computer Architecture (June)..

SOHI, G. S. [1990]. “Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers,” *IEEE Trans. on Computers* 39:3 (March), 349-359.

SOHI, G. S. AND S. VAJAPEYAM [1989]. “Tradeoffs in instruction format design for horizontal architectures,” *Proc. Third Conf. on Architectural Support for Programming Languages and Operating Systems*, IEEE/ACM (April), Boston, 15–25.

SUSSENGUTH, E[1999]. "IBM's ACS-1 Machine," IEEE Computer, 22: 11(November).

THORLIN, J. F. [1967]. “Code generation for PIE (parallel instruction execution) computers,” *Proc. Spring Joint Computer Conf.* 27.

THORNTON, J. E. [1964]. “Parallel operation in the Control Data 6600,” *Proc. AFIPS Fall Joint Computer Conf., Part II*, 26, 33–40.

THORNTON, J. E. [1970]. *Design of a Computer, the Control Data 6600*, Scott, Foresman, Glenview,

Ill.

TJADEN, G. S. AND M. J. FLYNN [1970]. “Detection and parallel execution of independent instructions,” *IEEE Trans. on Computers* C-19:10 (October), 889–895.

TOMASULO, R. M. [1967]. “An efficient algorithm for exploiting multiple arithmetic units,” *IBM J. Research and Development* 11:1 (January), 25–33.

WALL, D. W. [1991]. “Limits of instruction-level parallelism,” *Proc. Fourth Conf. on Architectural Support for Programming Languages and Operating Systems* (April), Santa Clara, Calif., IEEE/ACM, 248–259.

WALL, D. W. [1993]. *Limits of Instruction-Level Parallelism*, Research Rep. 93/6, Western Research Laboratory, Digital Equipment Corp. (November).

WEISS, S. AND J. E. SMITH [1984]. “Instruction issue logic for pipelined supercomputers,” *Proc. 11th Symposium on Computer Architecture* (June), Ann Arbor, Mich., 110–118.

WEISS, S. AND J. E. SMITH [1987]. “A study of scalar compilation techniques for pipelined supercomputers,” *Proc. Second Conf. on Architectural Support for Programming Languages and Operating Systems* (March), IEEE/ACM, Palo Alto, Calif., 105–109.

WEISS, S. AND J. E. SMITH [1994]. *Power and PowerPC*, Morgan Kaufmann, San Francisco.

YEAGER, K. ET AL. [1996] "The MIPS R10000 Superscalar Microprocessor". IEEE Micro, vol 16, No 2, (April), pp 28-40.

YEH, T. AND Y. N. PATT [1992]. “Alternative implementations of two-level adaptive branch prediction,” *Proc. 19th International Symposium on Computer Architecture* (May), Gold Coast, Australia, 124–134.

YEH, T. AND Y. N. PATT [1993]. “A comparison of dynamic branch predictors that use two levels of branch history,” *Proc. 20th Symposium on Computer Architecture* (May), San Diego, 257–266.

EXERCISES

3.1 Exercise from Dave (not fully thought out, but a good direction): Given a table like that in Figures 3.25 on page 275 or 3.26 on page 276 and some of the following deduce the rest of the following:

a. the original code

b. the number of functional units

c. the number of instructions issued per clock

d. the functional units

3.2 [10] <3.1> For the following code fragment, list the control dependences. For each control dependence, tell whether the statement can be scheduled before the if statement based on the data references. Assume that all data references are shown, that all values are defined before use, and that only `b` and `c` are used again after this segment. You may ignore any possible exceptions.

```
if (a>c) {
        d = d + 5;
        a = b + d + e;}
else {
        e = e + 2;
        f = f + 2;
```

```
        c = c + f;
}
b = a + f;
```

A good exercise but requires describing how scoreboards work. There are a number of problems based on scoreboards, which may be salvagable by one of the following: introducing scoreboards (maybe not worth it), removing part of the reanming capability (WAW ror WAR) and asking about the result, recasting the problem to ask how Tomasulo avoids the problem.

3.3 [20] <3.2> It is critical that the scoreboard be able to distinguish RAW and WAR hazards, since a WAR hazard requires stalling the instruction doing the writing until the instruction reading an operand initiates execution, but a RAW hazard requires delaying the reading instruction until the writing instruction finishes—just the opposite. For example, consider the sequence:

```
MUL.D   F0,F6,F4
SUB.D   F8,F0,F2
ADD.D   F2,F10,F2
```

The `SUB.D` depends on the `MUL.D` (a RAW hazard) and thus the `MUL.D` must be allowed to complete before the `SUB.D`; if the `MUL.D` were stalled for the `SUB.D` due to the inability to distinguish between RAW and WAR hazards, the processor will deadlock. This sequence contains a WAR hazard between the `ADD.D` and the `SUB.D`, and the `ADD.D` cannot be allowed to complete until the `SUB.D` begins execution. The difficulty lies in distinguishing the RAW hazard between `MUL.D` and `SUB.D`, and the WAR hazard between the `SUB.D` and `ADD.D`.

Describe how the scoreboard for a processor with two multiply units and two add units avoids this problem and show the scoreboard values for the above sequence assuming the `ADD.D` is the only instruction that has completed execution (though it has not written its result). (*Hint*: Think about how WAW hazards are prevented and what this implies about active instruction sequences.)

A good exercise I would transform it by saving that sometimes the CDB bandwidth acts as a limit, using the 2-issue tomasulo pipeline, show a sequence where 2 CDBs is not enough and can eventually cause a stall

3.4 [12] <3.2> A shortcoming of the scoreboard approach occurs when multiple functional units that share input buses are waiting for a single result. The units cannot start simultaneously, but must serialize. This property is not true in Tomasulo's algorithm. Give a code sequence that uses no more than 10 instructions and shows this problem. Assume the hardware configuration from Figure 4.3, for the scoreboard, and Figure 3.2, for Tomasulo's scheme. Use the FP latencies from Figure 4.2 (page 224). Indicate where the Tomasulo approach can continue, but the scoreboard approach must stall.

A good exercise but requires reworking (e.g., show how even with 1 issue/clock, a single cdb can be problem) to save it?

3.5 [15] <3.2> Tomasulo's algorithm also has a disadvantage versus the scoreboard: only one result can complete per clock, due to the CDB. Use the hardware configuration from Figures 4.3 and 3.2 and the FP latencies from Figure 4.2 (page 224). Find a code sequence

of no more than 10 instructions where the scoreboard does not stall, but Tomasulo's algorithm must due to CDB contention. Indicate where this occurs in your sequence.

Maybe also try a version of this with multiple issue?

3.6 [45] <3.2> One benefit of a dynamically scheduled processor is its ability to tolerate changes in latency or issue capability without requiring recompilation. This capability was a primary motivation behind the 360/91 implementation. The purpose of this programming assignment is to evaluate this effect. Implement a version of Tomasulo's algorithm for MIPS to issue one instruction per clock; your implementation should also be capable of in-order issue. Assume fully pipelined functional units and the latencies shown in Figure 3.62.

Unit	Latency
Integer	7
Branch	9
Load-store	11
FP add	13
FP mul	15
FP divide	17

FIGURE 3.62 Latencies for functional units.

A one-cycle latency means that the unit and the result are available for the next instruction. Assume the processor takes a one-cycle stall for branches, in addition to any data-dependent stalls shown in the above table. Choose 5–10 small FP benchmarks (with loops) to run; compare the performance with and without dynamic scheduling. Try scheduling the loops by hand and see how close you can get with the statically scheduled processor to the dynamically scheduled results.

Change the processor to the configuration shown in Figure 3.63.

Unit	Latency
Integer	19
Branch	21
Load-store	23
FP add	25
FP mul	27
FP divide	29

FIGURE 3.63 Latencies for functional units, configuration 2.

Rerun the loops and compare the performance of the dynamically scheduled processor and

the statically scheduled processor.

3.7 [15] <3.4> Suppose we have a deeply pipelined processor, for which we implement a branch-target buffer for the conditional branches only. Assume that the misprediction penalty is always 4 cycles and the buffer miss penalty is always 3 cycles. Assume 90% hit rate and 90% accuracy, and 15% branch frequency. How much faster is the processor with the branch-target buffer versus a processor that has a fixed 2-cycle branch penalty? Assume a base CPI without branch stalls of 1.

3.8 [10] <3.4> Determine the improvement from branch folding for unconditional branches. Assume a 90% hit rate, a base CPI without unconditional branch stalls of 1, and an unconditional branch frequency of 5%. How much improvement is gained by this enhancement versus a processor whose effective CPI is 1.1?

3.9 [30] <3.6> Implement a simulator to evaluate the performance of a branch-prediction buffer that does not store branches that are predicted as untaken. Consider the following prediction schemes: a one-bit predictor storing only predicted taken branches, a two-bit predictor storing all the branches, a scheme with a target buffer that stores only predicted taken branches and a two-bit prediction buffer. Explore different sizes for the buffers keeping the total number of bits (assuming 32-bit addresses) the same for all schemes. Determine what the branch penalties are, using Figure 3.21 as a guideline. How do the different schemes compare both in prediction accuracy and in branch cost?

3.10 [30] <3.6> Implement a simulator to evaluate various branch prediction schemes. You can use the instruction portion of a set of cache traces to simulate the branch-prediction buffer. Pick a set of table sizes (e.g., 1K bits, 2K bits, 8K bits, and 16K bits). Determine the performance of both (0,2) and (2,2) predictors for the various table sizes. Also compare the performance of the degenerate predictor that uses no branch address information for these table sizes. Determine how large the table must be for the degenerate predictor to perform as well as a (0,2) predictor with 256 entries.

this is an interesting exercise to do in several forms: tomsulo, multiple issue with tomasulo and even speculation. Needs some reqorking. may want to ask them to create tables like those in the text (Figures 3.25 on page 275 and 3.26 on page 276)

3.11 [20/22/22/22/22/25/25/25/20/22/22] <3.1,3.2,3.6> In this Exercise, we will look at how a common vector loop runs on a variety of pipelined versions of MIPS. The loop is the so-called SAXPY loop (discussed extensively in Appendix B) and the central operation in Gaussian elimination. The loop implements the vector operation $Y = a \times X + Y$ for a vector of length 100. Here is the MIPS code for the loop:

```
foo:    L.D     F2,0(R1)        ;load X(i)
        MUL.D   F4,F2,F0        ;multiply a*X(i)
        L.D     F6,0(R2)        ;load Y(i)
        ADD.D   F6,F4,F6        ;add a*X(i) + Y(i)
        S.D     F6,0(R2)        ;store Y(i)
        DADDUI  R1,R1,#8        ;increment X index
        DADDUI  R2,R2,#8        ;increment Y index
        DSGTUI  R3,R1,done      ;test if done
        BEQZ    R3,foo          ; loop if not done
```

For (a)–(e), assume that the integer operations issue and complete in one clock cycle (in-

cluding loads) and that their results are fully bypassed. Ignore the branch delay. You will use the FP latencies shown in Figure 4.2 (page 224). Assume that the FP unit is fully pipelined.

a. [20] <3.1> For this problem use the standard single-issue MIPS pipeline with the pipeline latencies from Figure 4.2. Show the number of stall cycles for each instruction and what clock cycle each instruction begins execution (i.e., enters its first EX cycle) on the first iteration of the loop. How many clock cycles does each loop iteration take?

b. [22] <3.2> Use the MIPS code for SAXPY above and a fully pipelined FPU with the latencies of Figure 4.2. Assume Tomasulo's algorithm for the hardware with one integer unit taking one execution cycle (a latency of 0 cycles to use) for all integer operations. Show the state of the reservation stations and register-status tables (as in Figure 3.3) when the `SGTI` writes its result on the CDB. Do not include the branch.

c. [22] <3.2> Using the MIPS code for SAXPY above, assume a scoreboard with the FP functional units described in Figure 4.3, plus one integer functional unit (also used for load-store). Assume the latencies shown in Figure 3.64. Show the state of the scoreboard (as in Figure 4.4) when the branch issues for the second time. Assume the branch was correctly predicted taken and took one cycle. How many clock cycles does each loop iteration take? You may ignore any register port/bus conflicts.

Instruction producing result	Instruction using result	Latency in clock cycles
FP multiply	FP ALU op	6
FP add	FP ALU op	4
FP multiply	FP store	5
FP add	FP store	3
Integer operation (including load)	Any	0

FIGURE 3.64 Pipeline latencies where latency is number of cycles between producing and consuming instruction.

d. [25] <3.2> Use the MIPS code for SAXPY above. Assume Tomasulo's algorithm for the hardware using one fully pipelined FP unit and one integer unit. Assume the latencies shown in Figure 3.64.

Show the state of the reservation stations and register status tables (as in Figure 3.3) when the branch is executed for the second time. Assume the branch was correctly predicted as taken. How many clock cycles does each loop iteration take?

e. [25] <3.1,3.6> Assume a superscalar architecture with Tomasulo's algorithm for scheduling that can issue any two independent operations in a clock cycle (including two integer operations). Unwind the MIPS code for SAXPY to make four copies of the body and schedule it assuming the FP latencies of Figure 4.2. Assume one fully pipelined copy of each functional unit (e.g., FP adder, FP multiplier) and two integer

functional units with latency to use of 0. How many clock cycles will each iteration on the original code take? When unwinding, you should optimize the code as in section 3.1. What is the speedup versus the original code?

f. [25] <3.6> In a superpipelined processor, rather than have multiple functional units, we would fully pipeline all the units. Suppose we designed a superpipelined MIPS that had twice the clock rate of our standard MIPS pipeline and could issue any two unrelated instructions in the same time that the normal MIPS pipeline issued one operation. If the second instruction is dependent on the first, only the first will issue. Unroll the MIPS SAXPY code to make four copies of the loop body and schedule it for this superpipelined processor, assuming the FP latencies of Figure 3.64. Also assume the load to use latency is 1 cycle, but other integer unit latencies are 0 cycles. How many clock cycles does each loop iteration take? Remember that these clock cycles are half as long as those on a standard MIPS pipeline or a superscalar MIPS.

g. [22] <3.2,3.5> Using the MIPS code for SAXPY above, assume a speculative processor with the functional unit organization used in section 3.5 and separate functional units for comparison, for branches, for effective address calculation, and for ALU operations. Assume the latencies shown in Figure 3.64. Show the state of the processor (as in Figure 3.30) when the branch issues for the second time. Assume the branch was correctly predicted taken and took one cycle. How many clock cycles does each loop iteration take?

h. [22] <3.2,3.5> Using the MIPS code for SAXPY above, assume a speculative processor like Figure 3.29 that can issue one load-store, one integer operation, and one FP operation each cycle. Assume the latencies in clock cycles of Figure 3.64. Show the state of the processor (as in Figure 3.30) when the branch issues for the second time. Assume the branch was correctly predicted taken and took one cycle. How many clock cycles does each loop iteration take?

3.12 [15/15] <3.5> Consider our speculative processor from section 3.5. Since the reorder buffer contains a value field, you might think that the value field of the reservation stations could be eliminated.

a. [15] <3.5> Show an example where this is the case and an example where the value field of the reservation stations is still needed. Use the speculative processor shown in Figure 3.29. Show MIPS code for both examples. How many value fields are needed in each reservation station?

b. [15] <3.5> Find a modification to the rules for instruction commit that allows elimination of the value fields in the reservation station. What are the negative side effects of such a change?

3.13 [20] <3.5> Our implementation of speculation uses a reorder buffer and introduces the concept of instruction commit, delaying commit and the irrevocable updating of the registers until we know an instruction will complete. There are two other possible implementation techniques, both originally developed as a method for preserving precise interrupts when issuing out of order. One idea introduces a future file that keeps future values of a register; this idea is similar to the reorder buffer. An alternative is to keep a history buffer that records values of registers that have been speculatively overwritten.

Design a speculative processor like the one in section 3.5 but using a history buffer. Show the state of the processor, including the contents of the history buffer, for the example in Figure 3.31. Show the changes needed to Figure 3.32 for a history buffer implementation. Describe exactly how and when entries in the history buffer are read and written, including what happens on an incorrect speculation.

3.14 [30/30] <3.10> This exercise involves a programming assignment to evaluate what types of parallelism might be expected in more modest, and more realistic, processors than those studied in section 3.8. These studies can be done using traces available with this text or obtained from other tracing programs. For simplicity, assume perfect caches. For a more ambitious project, assume a real cache. To simplify the task, make the following assumptions:

- Assume perfect branch and jump prediction: hence you can use the trace as the input to the window, without having to consider branch effects—the trace is perfect.
- Assume there are 64 spare integer and 64 spare floating-point registers; this is easily implemented by stalling the issue of the processor whenever there are more live registers required.
- Assume a window size of 64 instructions (the same for alias detection). Use greedy scheduling of instructions in the window. That is, at any clock cycle, pick for execution the first *n* instructions in the window that meet the issue constraints.

a. [30] <3.10> Determine the effect of limited instruction issue by performing the following experiments:

   - Vary the issue count from 4–16 instructions per clock,
   - Assuming eight issues per clock: determine what the effect of restricting the processor to two memory references per clock is.

b. [30] <3.10> Determine the impact of latency in instructions. Assume the following latency models for a processor that issues up to 16 instructions per clock:

   - Model 1: All latencies are one clock.
   - Model 2: Load latency and branch latency are one clock; all FP latencies are two clocks.
   - Model 3: Load and branch latency is two clocks; all FP latencies are five clocks.

Remember that with limited issue and a greedy scheduler, the impact of latency effects will be greater.

3.15 [Discussion] <3.4,3.5> Dynamic instruction scheduling requires a considerable investment in hardware. In return, this capability allows the hardware to run programs that could not be run at full speed with only compile-time, static scheduling. What trade-offs should be taken into account in trying to decide between a dynamically and a statically scheduled implementation? What situations in either hardware technology or program characteristics are likely to favor one approach or the other? Most speculative schemes rely on dynamic scheduling; how does speculation affect the arguments in favor of dynamic scheduling?

3.16 [Discussion] <3.4> There is a subtle problem that must be considered when imple-

menting Tomasulo's algorithm. It might be called the "two ships passing in the night problem." What happens if an instruction is being passed to a reservation station during the same clock period as one of its operands is going onto the common data bus? Before an instruction is in a reservation station, the operands are fetched from the register file; but once it is in the station, the operands are always obtained from the CDB. Since the instruction and its operand tag are in transit to the reservation station, the tag cannot be matched against the tag on the CDB. So there is a possibility that the instruction will then sit in the reservation station forever waiting for its operand, which it just missed. How might this problem be solved? You might consider subdividing one of the steps in the algorithm into multiple parts. (This intriguing problem is courtesy of J. E. Smith.)

3.17 [Discussion] <3.6-3.5> Discuss the advantages and disadvantages of a superscalar implementation, a superpipelined implementation, and a VLIW approach in the context of MIPS. What levels of ILP favor each approach? What other concerns would you consider in choosing which type of processor to build? How does speculation affect the results?

Need some more exercises on speculation, newer branch predictors, and probably also multiple issue with Tomasulo and with speculation--maybe an integer loop?

Add something on multiple processors/chip

4 Exploiting Instruction Level Parallelism with Software Approaches

Processors are being produced with the potential for very many parallel operations on the instruction level....Far greater extremes in instruction-level parallelism are on the horizon.

J. Fisher [1981], in the paper that inaugurated the term "instruction-level parallelism"

One of the surprises about IA-64 is that we hear no claims of high frequency, despite claims that an EPIC processor is less complex than a superscalar processor. It's hard to know why this is so, but one can speculate that the overall complexity involved in focusing on CPI, as IA-64 does, makes it hard to get high megahertz.

M. Hopkins [2000], in a commentary on the IA-64 architecture, a joint development of HP and Intel designed to achieve dramatic increases in the exploitation of ILP while retaining a simple architecture, which would allow higher performance.

4.1	Basic Compiler Techniques for Exposing ILP	221
4.2	Static Branch Prediction	231
4.3	Static Multiple Issue: the VLIW Approach	234
4.4	Advanced Compiler Support for Exposing and Exploiting ILP	238
4.5	Hardware Support for Exposing More Parallelism at Compile-Time	260
4.6	Crosscutting Issues	270
4.7	Putting It All Together: The Intel IA-64 Architecture and Itanium Processor	271
4.8	Another View: ILP in the Embedded and Mobile Markets	283
4.9	Fallacies and Pitfalls	292
4.10	Concluding Remarks	293
4.11	Historical Perspective and References	295
	Exercises	299

4.1 Basic Compiler Techniques for Exposing ILP

This chapter starts by examining the use of compiler technology to improve the performance of pipelines and simple multiple-issue processors. These techniques are key even for processors that make dynamic issue decisions but use static scheduling and are crucial for processors that use static issue or static scheduling. After applying these concepts to reducing stalls from data hazards in single issue pipelines, we examine the use of compiler-based techniques for branch prediction. Armed with this more powerful compiler technology, we examine the design and performance of multiple-issue processors using static issuing or scheduling. Sections 4.4 and 4.5 examine more advanced software and hardware techniques, designed to enable a processor to exploit more instruction-level parallelism. Putting It All Together examines the IA-64 architecture and its first implementation, Itanium. Two different static, VLIW-style processors are covered in Another View.

Basic Pipeline Scheduling and Loop Unrolling

To keep a pipeline full, parallelism among instructions must be exploited by finding sequences of unrelated instructions that can be overlapped in the pipeline. To avoid a pipeline stall, a dependent instruction must be separated from the source instruction by a distance in clock cycles equal to the pipeline latency of that source instruction. A compiler's ability to perform this scheduling depends both on the amount of ILP available in the program and on the latencies of the functional units in the pipeline. Throughout this chapter we will assume the FP unit latencies shown in Figure 4.1, unless different latencies are explicitly stated. We assume the standard 5-stage integer pipeline, so that branches have a delay of one clock cycle. We assume that the functional units are fully pipelined or replicated (as many times as the pipeline depth), so that an operation of any type can be issued on every clock cycle and there are no structural hazards.

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

FIGURE 4.1 Latencies of FP operations used in this chapter. The first column shows the originating instruction type. The second column is the type of the consuming instruction. The last column is the number of intervening clock cycles needed to avoid a stall. These numbers are similar to the average latencies we would see on an FP unit. The latency of a floating-point load to a store is zero, since the result of the load can be bypassed without stalling the store. We will continue to assume an integer load latency of 1 and an integer ALU operation latency of 0.

In this subsection, we look at how the compiler can increase the amount of available ILP by unrolling loops. This example serves both to illustrate an important technique as well as to motivate the more powerful program transformations described later in this chapter. We will rely on an example similar to the one we used in the last chapter, adding a scalar to a vector:

```
for (i=1000; i>0; i=i-1)
        x[i] = x[i] + s;
```

We can see that this loop is parallel by noticing that the body of each iteration is independent. We will formalize this notion later in this chapter and describe how we can test whether loop iterations are independent at compile-time. First, let's look at the performance of this loop, showing how we can use the parallelism to improve its performance for a MIPS pipeline with the latencies shown above.

The first step is to translate the above segment to MIPS assembly language. In the following code segment, `R1` is initially the address of the element in the array

with the highest address, and F2 contains the scalar value, s. Register R2 is precomputed, so that 8(R2) is the last element to operate on.

The straightforward MIPS code, not scheduled for the pipeline, looks like this:

```
Loop:  L.D     F0,0(R1)     ;F0=array element
       ADD.D   F4,F0,F2     ;add scalar in F2
       S.D     F4,0(R1)     ;store result
       DADDUI  R1,R1,#-8    ;decrement pointer
                            ;8 bytes (per DW)
       BNE     R1,R2,Loop   ;branch R1!=zero
```

Let's start by seeing how well this loop will run when it is scheduled on a simple pipeline for MIPS with the latencies from Figure 4.1.

EXAMPLE Show how the loop would look on MIPS, both scheduled and unscheduled, including any stalls or idle clock cycles. Schedule for both delays from floating-point operations and from the delayed branch.

ANSWER Without any scheduling the loop will execute as follows:

```
                                        Clock cycle issued
Loop:  L.D      F0,0(R1)                        1
       stall                                    2
       ADD.D    F4,F0,F2                        3
       stall                                    4
       stall                                    5
       S.D      F4,0(R1)                        6
       DADDUI   R1,R1,#-8                       7
       stall                                    8
       BNE      R1,R2,Loop                      9
       stall                                   10
```

This code requires 10 clock cycles per iteration. We can schedule the loop to obtain only one stall:

```
Loop:  L.D      F0,0(R1)
       DADDUI   R1,R1,#-8
       ADD.D    F4,F0,F2
       stall
       BNE      R1,R2,Loop  ;delayed branch
       S.D      F4,8(R1)    ;altered & interchanged
                             with DADDUI
```

Execution time has been reduced from 10 clock cycles to 6. The stall after ADD.D is for the use by the S.D. n

Notice that to schedule the delayed branch, the compiler had to determine that it could swap the DADDUI and S.D by changing the address to which the S.D stored: the address was 0(R1) and is now 8(R1). This change is not trivial, since most compilers would see that the S.D instruction depends on the DADDUI and would refuse to interchange them. A smarter compiler, capable of limited symbolic optimization, could figure out the relationship and perform the interchange. The chain of dependent instructions from the L.D to the ADD.D and then to the S.D determines the clock cycle count for this loop. This chain must take at least 6 cycles because of dependencies and pipeline latencies.

In the above example, we complete one loop iteration and store back one array element every 6 clock cycles, but the actual work of operating on the array element takes just 3 (the load, add, and store) of those 6 clock cycles. The remaining 3 clock cycles consist of loop overhead—the DADDUI and BNE—and a stall. To eliminate these 3 clock cycles we need to get more operations within the loop relative to the number of overhead instructions.

A simple scheme for increasing the number of instructions relative to the branch and overhead instructions is *loop unrolling*. Unrolling simply replicates the loop body multiple times, adjusting the loop termination code.

Loop unrolling can also be used to improve scheduling. Because it eliminates the branch, it allows instructions from different iterations to be scheduled together. In this case, we can eliminate the data use stall by creating additional independent instructions within the loop body. If we simply replicated the instructions when we unrolled the loop, the resulting use of the same registers could prevent us from effectively scheduling the loop. Thus, we will want to use different registers for each iteration, increasing the required register count.

EXAMPLE Show our loop unrolled so that there are four copies of the loop body, assuming R1 is initially a multiple of 32, which means that the number of loop iterations is a multiple of 4. Eliminate any obviously redundant computations and do not reuse any of the registers.

ANSWER Here is the result after merging the DADDUI instructions and dropping the unnecessary BNE operations that are duplicated during unrolling. Note that R2 must now be set so that 32(R2) is the starting address of the last four elements.

```
Loop:  L.D    F0,0(R1)
       ADD.D  F4,F0,F2
       S.D    F4,0(R1)       ;drop DADDUI & BNE
       L.D    F6,-8(R1)
       ADD.D  F8,F6,F2
       S.D    F8,-8(R1)      ;drop DADDUI & BNE
       L.D    F10,-16(R1)
```

```
ADD.D   F12,F10,F2
S.D     F12,-16(R1)    ;drop DADDUI & BNE
L.D     F14,-24(R1)
ADD.D   F16,F14,F2
S.D     F16,-24(R1)
DADDUI  R1,R1,#-32
BNE     R1,R2,Loop
```

We have eliminated three branches and three decrements of `R1`. The addresses on the loads and stores have been compensated to allow the `DADDUI` instructions on `R1` to be merged. This optimization may seem trivial, but it is not; it requires symbolic substitution and simplification. We will see more general forms of these optimizations that eliminate dependent computations in Section 4.4.

Without scheduling, every operation in the unrolled loop is followed by a dependent operation and thus will cause a stall. This loop will run in 28 clock cycles—each `L.D` has 1 stall, each `ADD.D` 2, the `DADDUI` 1, the branch 1, plus 14 instruction issue cycles—or 7 clock cycles for each of the four elements. Although this unrolled version is currently slower than the *scheduled* version of the original loop, this will change when we schedule the unrolled loop. Loop unrolling is normally done early in the compilation process, so that redundant computations can be exposed and eliminated by the optimizer. n

In real programs we do not usually know the upper bound on the loop. Suppose it is n, and we would like to unroll the loop to make k copies of the body. Instead of a single unrolled loop, we generate a pair of consecutive loops. The first executes (n mod k) times and has a body that is the original loop. The second is the unrolled body surrounded by an outer loop that iterates (n/k) times. For large values of n, most of the execution time will be spent in the unrolled loop body.

In the above Example, unrolling improves the performance of this loop by eliminating overhead instructions, although it increases code size substantially. How will the unrolled loop perform when it is scheduled for the pipeline described earlier?

EXAMPLE Show the unrolled loop in the previous example after it has been scheduled for the pipeline with the latencies shown in Figure 4.1 on page 222.

ANSWER

```
Loop:  L.D     F0,0(R1)
       L.D     F6,-8(R1)
       L.D     F10,-16(R1)
       L.D     F14,-24(R1)
       ADD.D   F4,F0,F2
```

```
ADD.D    F8,F6,F2
ADD.D    F12,F10,F2
ADD.D    F16,F14,F2
S.D      F4,0(R1)
S.D      F8,-8(R1)
DADDUI   R1,R1,#-32
S.D      F12,16(R1)
BNE      R1,R2,Loop
S.D      F16,8(R1)     ;8-32 = -24
```

The execution time of the unrolled loop has dropped to a total of 14 clock cycles, or 3.5 clock cycles per element, compared with 7 cycles per element before scheduling and 6 cycles when scheduled but not unrolled.

n

The gain from scheduling on the unrolled loop is even larger than on the original loop. This increase arises because unrolling the loop exposes more computation that can be scheduled to minimize the stalls; the code above has no stalls. Scheduling the loop in this fashion necessitates realizing that the loads and stores are independent and can be interchanged.

Summary of the Loop Unrolling and Scheduling Example

Throughout this chapter we will look at a variety of hardware and software techniques that allow us to take advantage of instruction-level parallelism to fully utilize the potential of the functional units in a processor. The key to most of these techniques is to know when and how the ordering among instructions may be changed. In our example we made many such changes, which to us, as human beings, were obviously allowable. In practice, this process must be performed in a methodical fashion either by a compiler or by hardware. To obtain the final unrolled code we had to make the following decisions and transformations:

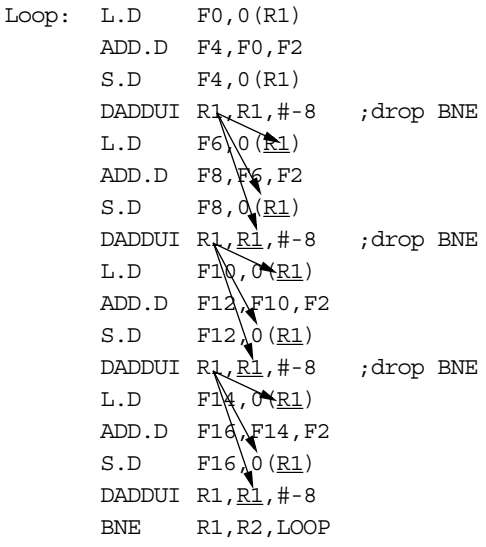
1. Determine that it was legal to move the `S.D` after the `DADDUI` and `BNE`, and find the amount to adjust the `S.D` offset.
2. Determine that unrolling the loop would be useful by finding that the loop iterations were independent, except for the loop maintenance code.
3. Use different registers to avoid unnecessary constraints that would be forced by using the same registers for different computations.
4. Eliminate the extra test and branch instructions and adjust the loop termination and iteration code.
5. Determine that the loads and stores in the unrolled loop can be interchanged by observing that the loads and stores from different iterations are independent. This transformation requires analyzing the memory addresses and finding that they do not refer to the same address.

6. Schedule the code, preserving any dependences needed to yield the same result as the original code.

The key requirement underlying all of these transformations is an understanding of how an instruction depends on another and how the instructions can be changed or reordered given the dependences. Before examining how these techniques work for higher issue rate pipelines, let us examine how the loop unrolling and scheduling techniques affect data dependences.

EXAMPLE Show how the process of optimizing the loop overhead by unrolling the loop actually eliminates data dependences. In this example and those used in the remainder of this chapter, we use nondelayed branches for simplicity; it is easy to extend the examples to use delayed branches.

ANSWER Here is the unrolled but unoptimized code with the extra DADDUI instructions, but without the branches. (Eliminating the branches is another type of transformation, since it involves control rather than data.) The arrows show the data dependences that are within the unrolled body and involve the DADDUI instructions. The underlined registers are the dependent uses.

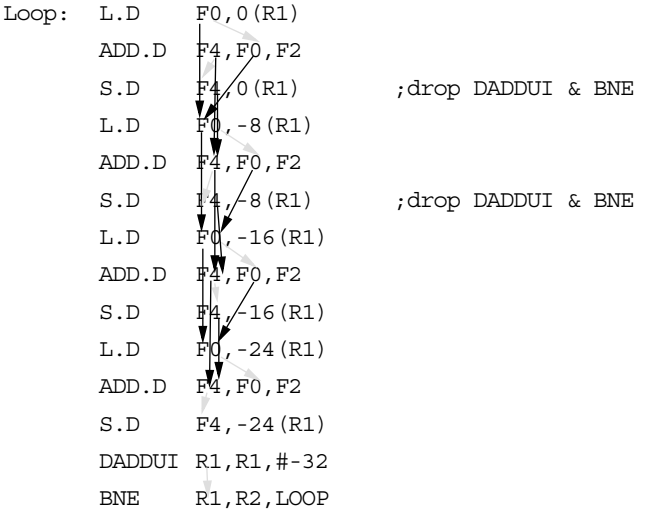


As the arrows show, the DADDUI instructions form a dependent chain that involves the DADDUI, L.D, and S.D instructions. This chain forces the body to execute in order, as well as making the DADDUI instructions necessary, which increases the instruction count. The compiler removes this dependence by symbolically computing the intermediate values of R1 and fold-

ing the computation into the offset of the L.D and S.D instructions and by changing the final DADDUI into a decrement by 32. This transformation makes the three DADDUI unnecessary, and the compiler can remove them. There are other types of dependences in this code, as the next few example show. n

EXAMPLE Unroll our example loop, eliminating the excess loop overhead, but using the same registers in each loop copy. Indicate both the data and name dependences within the body. Show how renaming eliminates name dependences that reduce parallelism.

ANSWER Here's the loop unrolled but with the same registers in use for each copy. The data dependences are shown with gray arrows and the name dependences with black arrows. As in earlier examples, the direction of the arrow indicates the ordering that must be preserved for correct execution of the code:



The name dependences force the instructions in the loop to be almost completely ordered, allowing only the order of the L.D following each S.D to be interchanged. When the registers used for each copy of the loop

body are renamed only the true dependences within each body remain:

```
Loop:  L.D    F0,0(R1)
       ADD.D  F4,F0,F2
       S.D    F4,0(R1)        ;drop DADDUI & BNE
       L.D    F6,-8(R1)
       ADD.D  F8,F6,F2
       S.D    F8,-8(R1)       ;drop DADDUI & BNE
       L.D    F10,-16(R1)
       ADD.D  F12,F10,F2
       S.D    F12,-16(R1)
       L.D    F14,-24(R1)
       ADD.D  F16,F14,F2
       S.D    F16,-24(R1)
       DADDUI R1,R1,#-32
       BNE    R1,R2,LOOP
```

With the renaming, the copies of each loop body become independent and can be overlapped or executed in parallel. This renaming process can be performed either by the compiler or in hardware, as we saw in the last chapter. n

There are three different types of limits to the gains that can be achieved by loop unrolling: a decrease in the amount of overhead amortized with each unroll, code size limitations, and compiler limitations. Let's consider the question of loop overhead first. When we unrolled the loop four times, it generated sufficient parallelism among the instructions that the loop could be scheduled with no stall cycles. In fact, in fourteen clock cycles, only two cycles were loop overhead: the DSUBI, which maintains the index value, and the BNE, which terminates the loop. If the loop is unrolled eight times, the overhead is reduced from 1/2 cycle per original iteration to 1/4. One of the exercises asks you to compute the theoretically optimal number of times to unroll this loop for a random number of iterations.

A second limit to unrolling is the growth in code size that results. For larger loops, the code size growth may be a concern either in the embedded space where memory may be at a premium or if the larger code size causes a decrease in the instruction cache miss rate. We return to the issue of code size when we consider more aggressive techniques for uncovering instruction level parallelism in Section 4.4.

Another factor often more important than code size is the potential shortfall in registers that is created by aggressive unrolling and scheduling. This secondary affect that results from instruction scheduling in large code segments is called *register pressure*. It arises because scheduling code to increase ILP causes the number of live values to increase. After aggressive instruction scheduling, it not be possi-

ble to allocate all the live values to registers. The transformed code, while theoretically faster, may lose some or all of its advantage, because it generates a shortage of registers. Without unrolling, aggressive scheduling is sufficiently limited by branches so that register pressure is rarely a problem. The combination of unrolling and aggressive scheduling can, however, cause this problem. The problem becomes especially challenging in multiple issue machines that require the exposure of more independent instruction sequences whose execution can be overlapped. In general, the use of sophisticated high-level transformations, whose potential improvements are hard to measure before detailed code generation, has led to significant increases in the complexity of modern compilers.

Loop unrolling is a simple but useful method for increasing the size of straightline code fragments that can be scheduled effectively. This transformation is useful in a variety of processors, from simple pipelines like those in MIPS to the statically scheduled superscalars we described in the last chapter, as we will see now.

Using Loop Unrolling and Pipeline Scheduling with Static Multiple Issue

We begin by looking at a simple two-issue, statically-scheduled superscalar MIPS pipeline from the last chapter, using the pipeline latencies from Figure 4.1 on page 222 and the same example code segment we used for the single issue examples above. This processor can issue two instructions per clock cycle, where one of the instructions can be a load, store, branch, or integer ALU operation, and the other can be any floating-point operation.

Recall that this pipeline did not generate a significant performance enhancement for the example above, because of the limited ILP in a given loop iteration. Let's see how loop unrolling and pipeline scheduling can help.

EXAMPLE Unroll and schedule the loop used in the earlier examples and shown on page 223.

ANSWER To schedule this loop without any delays, we will need to unroll the loop to make five copies of the body. After unrolling, the loop will contain five each of `L.D`, `ADD.D`, and `S.D`; one `DADDUI`; and one `BNE`. The unrolled and scheduled code is shown in Figure 4.2.

This unrolled superscalar loop now runs in 12 clock cycles per iteration, or 2.4 clock cycles per element, versus 3.5 for the scheduled and unrolled loop on the ordinary MIPS pipeline. In this Example, the performance of the superscalar MIPS is limited by the balance between integer and floating-point computation. Every floating-point instruction is issued together with an integer instruction, but there are not enough floating-point instructions to keep the floating-point pipeline full. When scheduled, the original loop ran in 6 clock cycles per iteration. We have improved on that by a factor of 2.5, more than half of which came from loop unrolling. Loop unrolling took us from 6 to 3.5 (a factor of 1.7), while superscalar execution gave us

	Integer instruction		FP instruction	Clock cycle
Loop:	L.D	F0,0(R1)		1
	L.D	F6,-8(R1)		2
	L.D	F10,-16(R1)	ADD.D F4,F0,F2	3
	L.D	F14,-24(R1)	ADD.D F8,F6,F2	4
	L.D	F18,-32(R1)	ADD.D F12,F10,F2	5
	S.D	F4,0(R1)	ADD.D F16,F14,F2	6
	S.D	F8,-8(R1)	ADD.D F20,F18,F2	7
	S.D	F12,-16(R1)		8
	DADDUI	R1,R1,#-40		9
	S.D	F16,16(R1)		10
	BNE	R1,R2,Loop		11
	S.D	F20,8(R1)		12

FIGURE 4.2 The unrolled and scheduled code as it would look on a superscalar MIPS.

a factor of 1.5 improvement. n

.

4.2 Static Branch Prediction

In Chapter 3, we examined the use of dynamic branch predictors. Static branch predictors are sometimes used in processors where the expectation is that branch behavior is highly predictable at compile-time; static prediction can also be used to assist dynamic predictors.

In Chapter 1, we discussed an architectural feature that supports static branch predication, namely delayed branches. Delayed branches expose a pipeline hazard so that the compiler can reduce the penalty associated with the hazard. As we saw, the effectiveness of this technique partly depends on whether we correctly guess which way a branch will go. Being able to accurately predict a branch at compile time is also helpful for scheduling data hazards. Loop unrolling is on simple example of this; another example, arises from conditional selection branches. Consider the following code segment:

```
LD      R1,0(R2)
DSUBU   R1,R1,R3
BEQZ    R1,L
OR      R4,R5,R6
DADDU   R10,R4,R3
```

```
L:      DADDU   R7,R8,R9
```

The dependence of the DSUBU and BEQZ on the LD instruction means that a stall will be needed after the LD. Suppose we knew that this branch was almost always taken and that the value of R7 was not needed on the fall-through path. Then we could increase the speed of the program by moving the instruction DADD R7,R8,R9 to the position after the LD. Correspondingly, if we knew the branch was rarely taken and that the value of R4 was not needed on the taken path, then we could contemplate moving the OR instruction after the LD. In addition, we can also use the information to better schedule any branch delay, since choosing how to schedule the delay depends on knowing the branch behavior. We will return to this topic in section 4.4, when we discuss global code scheduling.

To perform these optimizations, we need to predict the branch statically when we compile the program. There are several different methods to statically predict branch behavior. The simplest scheme is to predict a branch as taken. This scheme has an average misprediction rate that is equal to the untaken branch frequency, which for the SPEC programs is 34%. Unfortunately, the misprediction rate ranges from not very accurate (59%) to highly accurate (9%).

A better alternative is to predict on the basis of branch direction, choosing backward-going branches to be taken and forward-going branches to be not taken. For some programs and compilation systems, the frequency of forward taken branches may be significantly less than 50%, and this scheme will do better than just predicting all branches as taken. In the SPEC programs, however, more than half of the forward-going branches are taken. Hence, predicting all branches as taken is the better approach. Even for other benchmarks or compilers, direction-based prediction is unlikely to generate an overall misprediction rate of less than 30% to 40%. An enhancement of this technique was explored by Ball and Larus; their approach uses program context information and generates more accurate predictions than a simple scheme based solely on branch direction.

A still more accurate technique is to predict branches on the basis of profile information collected from earlier runs. The key observation that makes this worthwhile is that the behavior of branches is often bimodally distributed; that is, an individual branch is often highly biased toward taken or untaken. Figure 4.3 shows the success of branch prediction using this strategy. The same input data were used for runs and for collecting the profile; other studies have shown that

changing the input so that the profile is for a different run leads to only a small change in the accuracy of profile-based prediction.

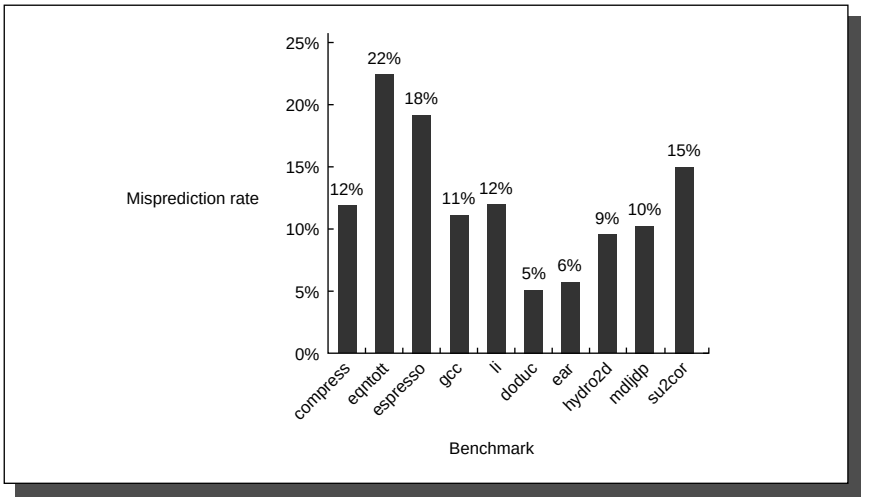


FIGURE 4.3 Misprediction rate on SPEC92 for a profile-based predictor varies widely but is generally better for the FP programs, which have an average misprediction rate of 9% with a standard deviation of 4%, than for the integer programs, which have an average misprediction rate of 15% with a standard deviation of 5%. The actual performance depends on both the prediction accuracy and the branch frequency, which varies from 3% to 24%; we will examine the combined effect in Figure 4.4.

Although we can derive the prediction accuracy of a predict-taken strategy and measure the accuracy of the profile scheme, as in Figure 4.3, the wide range of frequency of conditional branches in these programs, from 3% to 24%, means that the overall frequency of a mispredicted branch varies widely. Figure 4.4 shows the number of instructions executed between mispredicted branches for both a profile-based and a predict-taken strategy. The number varies widely, both because of the variation in accuracy and the variation in branch frequency. On average, the predict-taken strategy has 20 instructions per mispredicted branch and the profile-based strategy has 110. These averages, however, are very different for integer and FP programs, as the data in Figure 4.4 show.

Static branch behavior is useful for scheduling instructions when the branch delays are exposed by the architecture (either delayed or canceling branches), for assisting dynamic predictors (as we will see in the IA-64 architecture in section 4.7), and for determining which code paths are more frequent, which is a key step in code scheduling (see section 4.4, page 251).

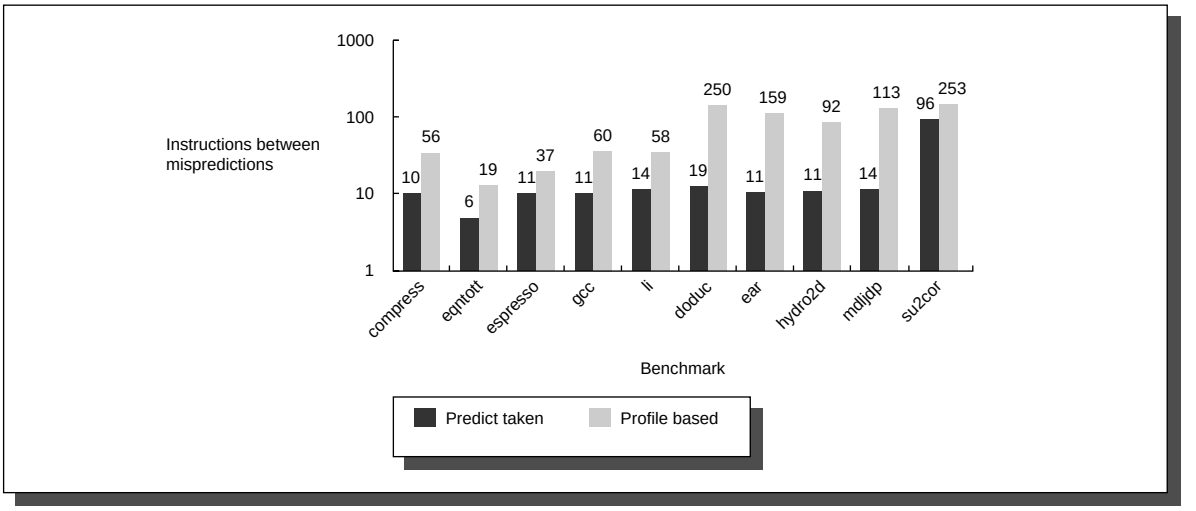


FIGURE 4.4 Accuracy of a predict-taken strategy and a profile-based predictor for SPEC92 benchmarks as measured by the number of instructions executed between mispredicted branches and shown on a log scale. The average number of instructions between mispredictions is 20 for the predict-taken strategy and 110 for the profile-based prediction; however, the standard deviations are large: 27 instructions for the predict-taken strategy and 85 instructions for the profile-based scheme. This wide variation arises because programs such as su2cor have both low conditional branch frequency (3%) and predictable branches (85% accuracy for profiling), although eqntott has eight times the branch frequency with branches that are nearly 1.5 times *less* predictable. The difference between the FP and integer benchmarks as groups is large. For the predict-taken strategy, the average distance between mispredictions for the integer benchmarks is 10 instructions, and it is 30 instructions for the FP programs. With the profile scheme, the distance between mispredictions for the integer benchmarks is 46 instructions, and it is 173 instructions for the FP benchmarks.

.

4.3 Static Multiple Issue: the VLIW Approach

Superscalar processors decide on the fly how many instructions to issue. A statically scheduled superscalar must check for any dependences between instructions in the issue packet as well as between any issue candidate and any instruction already in the pipeline. As we have seen in Section 4.1, a statically-scheduled superscalar requires significant compiler assistance to achieve good performance. In contrast, a dynamically-scheduled superscalar requires less compiler assistance, but has significant hardware costs.

An alternative to the superscalar approach is to rely on compiler technology not only to minimize the potential data hazard stalls, but to actually format the instructions in a potential issue packet so that the hardware need not check explicitly for dependences. The compiler may be required to ensure that dependences within the issue packet cannot be present or, at a minimum, indicate when a dependence may be present. Such an approach offers the potential advantage of simpler hardware while still exhibiting good performance through extensive compiler optimization.

The first multiple-issue processors that required the instruction stream to be explicitly organized to avoid dependences used wide instructions with multiple operations per instruction. For this reason, this architectural approach was named VLIW, standing for Very Long Instruction Word, and denoting that the instructions, since they contained several instructions, were very wide (64 to 128 bits, or more). The basic architectural concepts and compiler technology are the same whether multiple operations are organized into a single instruction, or whether a set of instructions in an issue packet is preconfigured by a compiler to exclude dependent operations (since the issue packet can be thought of as a very large instruction). Early VLIWs were quite rigid in their instruction formats and effectively required recompilation of programs for different versions of the hardware.

To reduce this inflexibility and enhance performance of the approach, several innovations have been incorporated into more recent architectures of this type, while still requiring the compiler to do most of the work of finding and scheduling instructions for parallel execution. This second generation of VLIW architectures is the approach being pursued for desktop and server markets.

In the remainder of this section, we look at the basic concepts in a VLIW architecture. Section 4.4 introduces additional compiler techniques that are required to achieve good performance for compiler-intensive approaches, and Section 4.5 describes hardware innovations that improve flexibility and performance of explicitly parallel approaches. Finally, Section 4.7 describes how the Intel IA-64 supports explicit parallelism.

The Basic VLIW Approach

VLIWs use multiple, independent functional units. Rather than attempting to issue multiple, independent instructions to the units, a VLIW packages the multiple operations into one very long instruction, or requires that the instructions in the issue packet satisfy the same constraints. Since there is not fundamental difference in the two approaches, we will just assume that multiple operations are placed in one instruction, as in the original VLIW approach. Since the burden for choosing the instructions to be issued simultaneously falls on the compiler, the hardware in a superscalar to make these issue decisions is unneeded.

Since this advantage of a VLIW increases as the maximum issue rate grows, we focus on a wider-issue processor. Indeed, for simple two issue processors, the overhead of a superscalar is probably minimal. Many designers would probably argue that a four issue processor has manageable overhead, but as we saw in the last chapter, this overhead grows with issue width.

Because VLIW approaches make sense for wider processors, we choose to focus our example on such an architecture. For example, a VLIW processor might have instructions that contain five operations, including: one integer operation (which could also be a branch), two floating-point operations, and two memory references. The instruction would have a set of fields for each functional unit—perhaps 16 to 24 bits per unit, yielding an instruction length of between 112 and 168 bits.

To keep the functional units busy, there must be enough parallelism in a code sequence to fill the available operation slots. This parallelism is uncovered by unrolling loops and scheduling the code within the single larger loop body. If the unrolling generates straighline code, then *local scheduling* techniques, which operate on a single basic block can be used. If finding and exploiting the parallelism requires scheduling code across branches, a substantially more complex *global scheduling* algorithm must be used. Global scheduling algorithms are not only more complex in structure, but they must deal with significantly more complicated tradeoffs in optimization, since moving code across branches is expensive. In the next section, we will discuss *trace scheduling*, one of these global scheduling techniques developed specifically for VLIWs. In Section 4.5, we will examine hardware support that allows some conditional branches to be eliminated, extending the usefulness of local scheduling and enhancing the performance of global scheduling.

For now, let's assume we have a technique to generate long, straight-line code sequences, so that we can use local scheduling to build up VLIW instructions and instead focus on how well these processors operate.

EXAMPLE Suppose we have a VLIW that could issue two memory references, two FP operations, and one integer operation or branch in every clock cycle. Show an unrolled version of the loop x[i] = x[i] +s (see page 223 for the MIPS ode) for such a processor. Unroll as many times as necessary to eliminate any stalls. Ignore the branch-delay slot.

ANSWER The code is shown in Figure 4.5. The loop has been unrolled to make seven copies of the body, which eliminates all stalls (i.e., completely empty issue cycles), and runs in 9 cycles. This code yields a running rate of seven results in 9 cycles, or 1.29 cycles per result, nearly twice as fast as the two-issue superscalar of Section 4.1 that used unrolled and scheduled code. n

For the original VLIW model, there are both technical and logistical problems. The technical problems are the increase in code size and the limitations of lock-step operation. Two different elements combine to increase code size substantially for a VLIW. First, generating enough operations in a straight-line code fragment requires ambitiously unrolling loops (as earlier examples) thereby increasing code size. Second, whenever instructions are not full, the unused functional units translate to wasted bits in the instruction encoding. In Figure 4.5, we saw that only about 60% of the functional units were used, so almost half of each instruction was empty. In most VLIWs, an instruction may need to be left completely empty if no operations can be scheduled.

To combat this code size increase, clever encodings are sometimes used. For example, there may be only one large immediate field for use by any functional unit. Another technique is to compress the instructions in main memory and ex-

Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation/branch
L.D F0,0(R1)	L.D F6,-8(R1)			
L.D F10,-16(R1)	L.D F14,-24(R1)			
L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2	
L.D F26,-48(R1)		ADD.D F12,F10,F2	ADD.D F16,F14,F2	
		ADD.D F20,F18,F2	ADD.D F24,F22,F2	
S.D F4,0(R1)	S.D -8(R1),F8	ADD.D F28,F26,F2		
S.D F12,-16(R1)	S.D -24(R1),F16			
S.D F20,-32(R1)	S.D -40(R1),F24			DADDUI R1,R1,#-56
S.D F28,8(R1)				BNE R1,R2,Loop

FIGURE 4.5 VLIW instructions that occupy the inner loop and replace the unrolled sequence. This code takes nine cycles assuming no branch delay; normally the branch delay would also need to be scheduled. The issue rate is 23 operations in nine clock cycles, or 2.5 operations per cycle. The efficiency, the percentage of available slots that contained an operation, is about 60%. To achieve this issue rate requires a larger number of registers than MIPS would normally use in this loop. The VLIW code sequence above requires at least eight FP registers, while the same code sequence for the base MIPS processor can use as few as two FP registers or as many as five when unrolled and scheduled. In the superscalar example in Figure 4.2, six registers were needed.

pand them when they are read into the cache or are decoded. We will see techniques to reduce code size increases in both Sections 4.7 and 4.8.

Early VLIWs operated in lock-step; there was no hazard detection hardware at all. This structure dictated that a stall in any functional unit pipeline must cause the entire processor to stall, since all the functional units must be kept synchronized. Although a compiler may be able to schedule the deterministic functional units to prevent stalls, predicting which data accesses will encounter a cache stall and scheduling them is very difficult. Hence, caches needed to be blocking and to cause *all* the functional units to stall. As the issue rate and number of memory references becomes large, this synchronization restriction becomes unacceptable. In more recent processors, the functional units operate more independently, and the compiler is used to avoid hazards at issue time, while hardware checks allow for unsynchronized execution once instructions are issued.

Binary code compatibility has also been a major logistical problem for VLIWs. In a strict VLIW approach, the code sequence makes use of both the instruction set definition and the detailed pipeline structure, including both functional units and their latencies. Thus, different numbers of functional units and unit latencies require different versions of the code. This requirement makes migrating between successive implementations, or between implementations with different issue widths, more difficult than it is for a superscalar design. Of course, obtaining improved performance from a new superscalar design may require recompilation. Nonetheless, the ability to run old binary files is a practical advantage for the superscalar approach.

One possible solution to this migration problem, and the problem of binary code compatibility in general, is object-code translation or emulation. This technology is developing quickly and could play a significant role in future migration

schemes. Another approach is to temper the strictness of the approach so that binary compatibility is still feasible. This later approach is used in the IA-64 architecture, as we will see in Section 4.7.

The major challenge for all multiple-issue processors is to try to exploit large amounts of ILP. When the parallelism comes from unrolling simple loops in FP programs, the original loop probably could have been run efficiently on a vector processor (described in Appendix B). It is not clear that a multiple-issue processor is preferred over a vector processor for such applications; the costs are similar, and the vector processor is typically the same speed or faster. The potential advantages of a multiple-issue processor versus a vector processor are twofold. First, a multiple-issue processor has the potential to extract some amount of parallelism from less regularly structured code, and, second, it has the ability to use a more conventional, and typically less expensive, cache-based memory system. For these reasons multiple-issue approaches have become the primary method for taking advantage of instruction-level parallelism, and vectors have become primarily an extension to these processors.

4.4 Advanced Compiler Support for Exposing and Exploiting ILP

In this section we discuss compiler technology for increasing the amount of parallelism that we can exploit in a program. We begin by defining when a loop is parallel and how a dependence can prevent a loop from being parallel. We also discuss techniques for eliminating some types of dependences. As we will see in later sections, hardware support for these compiler techniques can greatly increase their effectiveness. This section serves as an introduction to these techniques. We do not attempt to explain the details of ILP-oriented compiler techniques, since this would take hundreds of pages, rather than the 20 we have allotted. Instead, we view this material as providing general background that will enable the reader to have a basic understanding of the compiler techniques used to exploit ILP in modern computers.

Detecting and Enhancing Loop-Level Parallelism

Loop-level parallelism is normally analyzed at the source level or close to it, while most analysis of ILP is done once instructions have been generated by the compiler. Loop-level analysis involves determining what dependences exist among the operands in a loop across the iterations of that loop. For now, we will consider only data dependences, which arise when an operand is written at some point and read at a later point. Name dependences also exist and may be removed by renaming techniques like those we used earlier.

The analysis of loop-level parallelism focuses on determining whether data accesses in later iterations are dependent on data values produced in earlier iterations, such a dependence is called a *loop-carried dependence*. Most of the exam-

ples we considered in Section 4.1 have no loop-carried dependences and, thus, are loop-level parallel. To see that a loop is parallel, let us first look at the source representation:

```
for (i=1000; i>0; i=i-1)
       x[i] = x[i] + s;
```

In this loop, there is a dependence between the two uses of `x[i]`, but this dependence is within a single iteration and is not loop-carried. There is a dependence between successive uses of `i` in different iterations, which is loop-carried, but this dependence involves an induction variable and can be easily recognized and eliminated. We saw examples of how to eliminate dependences involving induction variables during loop unrolling in Section 4.1, and we will look at additional examples later in this section.

Because finding loop-level parallelism involves recognizing structures such as loops, array references, and induction variable computations, the compiler can do this analysis more easily at or near the source level, as opposed to the machine-code level. Let's look at a more complex example.

EXAMPLE Consider a loop like this one:

```
for (i=1; i<=100; i=i+1) {
       A[i+1] = A[i] + C[i];   /* S1 */
       B[i+1] = B[i] + A[i+1]; /* S2 */
}
```

Assume that `A`, `B`, and `C` are distinct, nonoverlapping arrays. (In practice, the arrays may sometimes be the same or may overlap. Because the arrays may be passed as parameters to a procedure, which includes this loop, determining whether arrays overlap or are identical often requires sophisticated, interprocedural analysis of the program.) What are the data dependences among the statements S1 and S2 in the loop?

ANSWER There are two different dependences:

1. S1 uses a value computed by S1 in an earlier iteration, since iteration `i` computes `A[i+1]`, which is read in iteration `i+1`. The same is true of S2 for `B[i]` and `B[i+1]`.
2. S2 uses the value, `A[i+1]`, computed by S1 in the same iteration.

These two dependences are different and have different effects. To see how they differ, let's assume that only one of these dependences exists at a time. Because the dependence of statement S1 on an earlier iteration of S1, this dependence is loop-carried. This dependence forces successive iterations of this loop to execute in series.

The second dependence above (S2 depending on S1) is within an it-

eration and is not loop-carried. Thus, if this were the only dependence, multiple iterations of the loop could execute in parallel, as long as each pair of statements in an iteration were kept in order. We saw this type of dependence in an example in Section 4.1, where unrolling was able to expose the parallelism.

n

It is also possible to have a loop-carried dependence that does not prevent parallelism, as the next example shows.

EXAMPLE Consider a loop like this one:

```
for (i=1; i<=100; i=i+1) {
       A[i] = A[i] + B[i];   /* S1 */
       B[i+1] = C[i] + D[i]; /* S2 */
}
```

What are the dependences between S1 and S2? Is this loop parallel? If not, show how to make it parallel.

ANSWER Statement S1 uses the value assigned in the previous iteration by statement S2, so there is a loop-carried dependence between S2 and S1. Despite this loop-carried dependence, this loop can be made parallel. Unlike the earlier loop, this dependence is not circular: Neither statement depends on itself, and although S1 depends on S2, S2 does not depend on S1. A loop is parallel if it can be written without a cycle in the dependences, since the absence of a cycle means that the dependences give a partial ordering on the statements.

Although there are no circular dependences in the above loop, it must be transformed to conform to the partial ordering and expose the parallelism. Two observations are critical to this transformation:

1. There is no dependence from S1 to S2. If there were, then there would be a cycle in the dependences and the loop would not be parallel. Since this other dependence is absent, interchanging the two statements will not affect the execution of S2.

2. On the first iteration of the loop, statement S1 depends on the value of `B[1]` computed prior to initiating the loop.

These two observations allow us to replace the loop above with the following code sequence:

```
A[1] = A[1] + B[1];
for (i=1; i<=99; i=i+1) {
       B[i+1] = C[i] + D[i];
       A[i+1] = A[i+1] + B[i+1];
```

```
}
B[101] = C[100] + D[100];
```

The dependence between the two statements is no longer loop-carried, so that iterations of the loop may be overlapped, provided the statements in each iteration are kept in order.

n

Our analysis needs to begin by finding all loop-carried dependences. This dependence information is *inexact*, in the sense that it tells us that such a dependence *may* exist. Consider the following example:

```
for (i=1;i<=100;i=i+1) {
       A[i] = B[i] + C[i]
       D[i] = A[i] * E[i]
}
```

The second reference to `A` in this example need not be translated to a load instruction, since we know that the value is computed and stored by the previous statement; hence, the second reference to `A` can simply be a reference to the register into which `A` was computed. Performing this optimization requires knowing that the two references are *always* to the same memory address and that there is no intervening access to the same location. Normally, data dependence analysis only tells that one reference *may* depend on another; a more complex analysis is required to determine that two references *must be* to the exact same address. In the example above, a simple version of this analysis suffices, since the two references are in the same basic block.

Often loop-carried dependences are in the form of a *recurrence*:

```
for (i=2;i<=100;i=i+1) {
       Y[i] = Y[i-1] + Y[i];
}
```

A recurrence is when a variable is defined based on the value of that variable in an earlier iteration, often the one immediately preceding, as in the above fragment. Detecting a recurrence can be important for two reasons: Some architectures (especially vector computers) have special support for executing recurrences, and some recurrences can be the source of a reasonable amount of parallelism. To see how the latter can be true, consider this loop:

```
for (i=6;i<=100;i=i+1) {
       Y[i] = Y[i-5] + Y[i];
}
```

On the iteration i, the loop references element $i - 5$. The loop is said to have a *dependence distance* of 5. Many loops with carried dependences have a dependence distance of 1. The larger the distance, the more potential parallelism can be obtained by unrolling the loop. For example, if we unroll the first loop, with a dependence distance of 1, successive statements are dependent on one another; there is still some parallelism among the individual instructions, but not much. If we unroll the loop that has a dependence distance of 5, there is a sequence of five statements that have no dependences, and thus much more ILP. Although many loops with loop-carried dependences have a dependence distance of 1, cases with larger distances do arise, and the longer distance may well provide enough parallelism to keep a processor busy.

Finding Dependences

Finding the dependences in a program is an important part of three tasks: (1) good scheduling of code, (2) determining which loops might contain parallelism, and (3) eliminating name dependences. The complexity of dependence analysis arises because of the presence of arrays and pointers in languages like C or C++ or pass-by-reference parameter passing in Fortran. Since scalar variable references explicitly refer to a name, they can usually be analyzed quite easily, with aliasing because of pointers and reference parameters causing some complications and uncertainty in the analysis.

How does the compiler detect dependences in general? Nearly all dependence analysis algorithms work on the assumption that array indices are *affine.* In simplest terms, a one-dimensional array index is affine if it can be written in the form $a \times i + b$, where a and b are constants, and i is the loop index variable. The index of a multidimensional array is affine if the index in each dimension is affine. Sparse array accesses, which typically have the form `x[y[i]]`, are one of the major examples of nonaffine accesses.

Determining whether there is a dependence between two references to the same array in a loop is thus equivalent to determining whether two affine functions can have the same value for different indices between the bounds of the loop. For example, suppose we have stored to an array element with index value $a \times i + b$ and loaded from the same array with index value $c \times i + d$, where i is the for-loop index variable that runs from m to n. A dependence exists if two conditions hold:

1. There are two iteration indices, j and k, both within the limits of the for loop. That is $m \le j \le n$, $m \le k \le n$.
2. The loop stores into an array element indexed by $a \times j + b$ and later fetches from that *same* array element when it is indexed by $c \times k + d$. That is, $a \times j + b = c \times k + d$.

In general, we cannot determine whether a dependence exists at compile time. For example, the values of *a, b, c*, and *d* may not be known (they could be values in other arrays), making it impossible to tell if a dependence exists. In other cases, the dependence testing may be very expensive but decidable at compile time. For example, the accesses may depend on the iteration indices of multiple nested loops. Many programs, however, contain primarily simple indices where *a, b, c*, and *d* are all constants. For these cases, it is possible to devise reasonable compile-time tests for dependence.

As an example, a simple and sufficient test for the absence of a dependence is the *greatest common divisor*, or GCD, test. It is based on the observation that if a loop-carried dependence exists, then GCD (*c,a*) must *divide* (*d – b*). (Recall that an integer, *x, divides* another integer, *y*, if there is no remainder when we do the division *y/x* and get an integer quotient.)

EXAMPLE Use the GCD test to determine whether dependences exist in the following loop:

```
for (i=1; i<=100; i=i+1) {
        X[2*i+3] = X[2*i] * 5.0;
}
```

ANSWER Given the values $a = 2$, $b = 3$, $c = 2$, and $d = 0$, then GCD(a,c) = 2, and $d – b = –3$. Since 2 does not divide –3, no dependence is possible. n

The GCD test is sufficient to guarantee that no dependence exists (you can show this in the Exercises); however, there are cases where the GCD test succeeds but no dependence exists. This can arise, for example, because the GCD test does not take the loop bounds into account.

In general, determining whether a dependence actually exists is NP-complete. In practice, however, many common cases can be analyzed precisely at low cost. Recently, approaches using a hierarchy of exact tests increasing in generality and cost have been shown to be both accurate and efficient. (A test is *exact* if it precisely determines whether a dependence exists. Although the general case is NP-complete, there exist exact tests for restricted situations that are much cheaper.)

In addition to detecting the presence of a dependence, a compiler wants to classify the type of dependence. This classification allows a compiler to recognize name dependences and eliminate them at compile time by renaming and copying.

EXAMPLE The following loop has multiple types of dependences. Find all the true dependences, output dependences, and antidependences, and eliminate

the output dependences and antidependences by renaming.

```
for (i=1; i<=100; i=i+1) {
    Y[i] = X[i] / c; /*S1*/
    X[i] = X[i] + c; /*S2*/
    Z[i] = Y[i] + c; /*S3*/
    Y[i] = c - Y[i]; /*S4*/
}
```

ANSWER The following dependences exist among the four statements:

1. There are true dependences from S1 to S3 and from S1 to S4 because of `Y[i]`. These are not loop carried, so they do not prevent the loop from being considered parallel. These dependences will force S3 and S4 to wait for S1 to complete.
2. There is an antidependence from S1 to S2, based on `X[i]`.
3. There is an antidependence from S3 to S4 for `Y[i]`.
4. There is an output dependence from S1 to S4, based on `Y[i]`.

The following version of the loop eliminates these false (or pseudo) dependences.

```
for (i=1; i<=100; i=i+1 {
    /* Y renamed to T to remove output dependence*/
    T[i] = X[i] / c;
    /* X renamed to X1 to remove antidependence*/
    X1[i] = X[i] + c;
    /* Y renamed to T to remove antidependence */
    Z[i] = T[i] + c;
    Y[i] = c - T[i];
}
```

After the loop the variable `X` has been renamed `X1`. In code that follows the loop, the compiler can simply replace the name `X` by `X1`. In this case, renaming does not require an actual copy operation but can be done by substituting names or by register allocation. In other cases, however, renaming will require copying. n

Dependence analysis is a critical technology for exploiting parallelism. At the instruction level it provides information needed to interchange memory references when scheduling, as well as to determine the benefits of unrolling a loop. For detecting loop-level parallelism, dependence analysis is the basic tool. Effectively compiling programs to either vector computers or multiprocessors depends criti-

cally on this analysis. The major drawback of dependence analysis is that it applies only under a limited set of circumstances, namely among references within a single loop nest and using affine index functions. Thus, there are a wide variety of situations in which array-oriented dependence analysis *cannot* tell us what we might want to know, including

- when objects are referenced via pointers rather than array indices (but see discussion below);
- when array indexing is indirect through another array, which happens with many representations of sparse arrays;
- when a dependence may exist for some value of the inputs, but does not exist in actuality when the code is run since the inputs never take on those values;
- when an optimization depends on knowing more than just the possibility of a dependence, but needs to know on *which* write of a variable does a read of that variable depend.

To deal with the issue of analyzing programs with pointers, another type of analysis, often called *points-to* analysis, is required (see Wilson and Lam [1995]). The key question that we want answered from dependence analysis of pointers is whether two pointers can designate the same address. In the case of complex dynamic data structures, this problem is extremely difficult. For example, we may want to know whether two pointers can reference the *same* node in a list at a given point in a program, which in general is undecidable and in practice is extremely difficult to answer. We may, however, be able to answer a simpler question: can two pointers designate nodes in the *same* list, even if they may be separate nodes. This more restricted analysis can still be quite useful in scheduling memory accesses performed through pointers.

The basic approach used in points-to analysis relies on information from three major sources:

1. Type information, which restricts what a pointer can point to.
2. Information derived when an object is allocated or when the address of an object is taken, which can be used to restrict what a pointer can point to. For example, if p always points to an object allocated in a given source line and q never points to that object, then p and q can never point to the same object.
3. Information derived from pointer assignments. For example, if p may be assigned the value of q, then p may point to anything q points to.

There are several cases where analyzing pointers has been successfully applied and is extremely useful:

- When pointers are used to pass the address of an object as a parameter, it is pos-

sible to use points-to analysis to determine the possible set of objects referenced by a pointer. One important use is to determine if two pointer parameters may designate the same object.

- When a pointer can point to one of several types, it is sometimes possible to determine that the type of the data object a pointer designates at different parts of the program.
- It is often possible to separate out pointers that may only point to a local object versus a global one.

There are two different types of limitations that affect our ability to do accurate dependence analysis for large programs. The first type of limitation arises from restrictions in the analysis algorithms. Often, we are limited by the lack of applicability of the analysis rather than a shortcoming in dependence analysis per se. For example, dependence analysis for pointers is essentially impossible for programs that use pointers in arbitrary fashion–for example, by doing arithmetic on pointers.

The second limitation is the need to analyze behavior across procedure boundaries to get accurate information. For example, if a procedure accepts two parameters that are pointers, determining whether the values could be the same requires analyzing across procedure boundaries. This type of analysis, called *interprocedural analysis*, is much more difficult and complex than analysis within a single procedure. Unlike the case of analyzing array indices within a single loop nest, points-to analysis usually requires an interprocedural analysis. The reason for this is simple. Suppose we are analyzing a program segment with two pointers; if the analysis does not know anything about the two pointers at the start of the program segment, it must be conservative and assume the worst case. The worst case is that the two pointers *may* designate the same object, but they are not *guaranteed* to designate the same object. This worst case is likely to propagate through the analysis producing useless information. In practice, getting fully accurate interprocedural information is usually too expensive for real programs. Instead, compilers usually use approximations in interprocedural analysis. The result is that the information may be too inaccurate to be useful.

Modern programming languages that use strong typing, such as Java, make the analysis of dependences easier. At the same time the extensive use of procedures to structure programs, as well as abstract data types, makes the analysis more difficult. Nonetheless, we expect that continued advances in analysis algorithms combined with the increasing importance of pointer dependency analysis will mean that there is continued progress on this important problem.

Eliminating Dependent Computations

Compilers can reduce the impact of dependent computations so as to achieve more ILP. The key technique is to eliminate or reduce a dependent computation

by back substitution, which increases the amount of parallelism and sometimes increases the amount of computation required. These techniques can be applied both within a basic block and within loops, and we describe them differently.

Within a basic block, algebraic simplifications of expressions and an optimization called *copy propagation*, which eliminates operations that copy values, can be used to simplify sequences like the following:

```
DADDUI R1,R2,#4
DADDUI R1,R1,#4
```

to:

```
DADDUI R1,R2,#8
```

assuming this is the only use of `R1`. In fact, the techniques we used to reduce multiple increments of array indices during loop unrolling and to move the increments across memory addresses in Section 4.1 are examples of this type of optimization.

In these examples, computations are actually eliminated, but it also possible that we may want to increase the parallelism of the code, possibly even increasing the number of operations. Such optimizations are called *tree height reduction*, since they reduce the height of the tree structure representing a computation, making it wider but shorter. Consider the following code sequence:

```
ADD    R1,R2,R3
ADD    R4,R1,R6
ADD    R8,R4,R7
```

Notice that this sequence requires at least three execution cycles, since all the instructions depend on the immediate predecessor. By taking advantage of associativity, we can transform the code and rewrite it as:

```
ADD    R1,R2,R3
ADD    R4,R6,R7
ADD    R8,R1,R4
```

This sequence can be computed in two execution cycles. When loop unrolling is used, opportunities for these types of optimizations occur frequently.

Although arithmetic with unlimited range and precision is associative, computer arithmetic is not associative, either for integer arithmetic, because of limited range, or floating point arithmetic, because of both range and precision. Thus, using these restructuring techniques can sometimes lead to erroneous behavior, although such occurrences are rare. For this reason, most compilers require that optimizations that rely on associativity be explicitly enabled.

When loops are unrolled this sort of algebraic optimization is important to reduce the impact of dependences arising from recurrences. *Recurrences* are expressions whose value on one iteration is given by a function that depends on the previous iterations. When a loop with a recurrence is unrolled, we may be able to algebraically optimize the unrolled loop, so that the recurrence need only be eval-

uated once per unrolled iteration. One common type of recurrence arises from an explicit program statements, such as:

```
sum = sum + x;
```

Assume we unroll a loop with this recurrence five times, if we let the value of x on these five iterations be given by `x1`, `x2`, `x3`, `x4`, and `x5`, then we can write the value of `sum` at the end of each unroll as:

```
sum = sum + x1 + x2 + x3 + x4 + x5;
```

If unoptimized this expression requires five dependent operations, but it can be rewritten as:

```
sum = ((sum + x1) + (x2 + x3)) + (x4 + x5);
```

which can be evaluated in only three dependent operations.

Recurrences also arise from implicit calculations, such as those associated with array indexing. Each array index translates to an address that is computed based on the loop index variable. Again, with unrolling and algebraic optimization, the dependent computations can be minimized.

Software Pipelining: Symbolic Loop Unrolling

We have already seen that one compiler technique, loop unrolling, is useful to uncover parallelism among instructions by creating longer sequences of straight-line code. There are two other important techniques that have been developed for this purpose: *software pipelining* and *trace scheduling.*

Software pipelining is a technique for reorganizing loops such that each iteration in the software-pipelined code is made from instructions chosen from different iterations of the original loop. This approach is most easily understood by looking at the scheduled code for the superscalar version of MIPS, which appeared in Figure 4.2 on page 231. The scheduler in this example essentially interleaves instructions from different loop iterations, so as to separate the dependent instructions that occur within a single loop iteration. By choosing instructions from different iterations, dependent computations are separated from one another by an entire loop body, increasing the possibility that the unrolled loop can be scheduled without stalls.

A software-pipelined loop interleaves instructions from different iterations without unrolling the loop, as illustrated in Figure 4.6. This technique is the software counterpart to what Tomasulo's algorithm does in hardware. The software-pipelined loop for the earlier example would contain one load, one add, and one store, each from a different iteration. There is also some start-up code that is needed before the loop begins as well as code to finish up after the loop is completed. We will ignore these in this discussion, for simplicity; the topic is addressed in the Exercises.

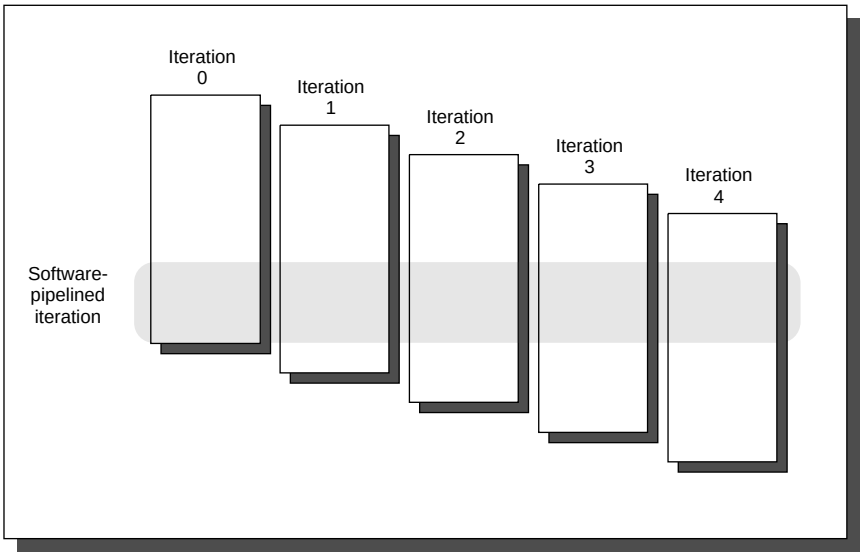


FIGURE 4.6 A software-pipelined loop chooses instructions from different loop iterations, thus separating the dependent instructions within one iteration of the original loop. The start-up and finish-up code will correspond to the portions above and below the software-pipelined iteration.

EXAMPLE Show a software-pipelined version of this loop, which increments all the elements of an array whose starting address is in R1 by the contents of F2:

```
Loop:  L.D    F0,0(R1)
       ADD.D  F4,F0,F2
       S.D    F4,0(R1)
       DADDUI R1,R1,#-8
       BNE    R1,R2,Loop
```

You may omit the start-up and clean-up code.

ANSWER Software pipelining symbolically unrolls the loop and then selects instructions from each iteration. Since the unrolling is symbolic, the loop overhead instructions (the `DADDUI` and `BNE`) need not be replicated. Here's the body of the unrolled loop without overhead instructions, highlighting the instructions taken from each iteration:

```
Iteration i:     L.D     F0,0(R1)
                 ADD.D   F4,F0,F2
                 S.D     F4,0(R1)
Iteration i+1:   L.D     F0,0(R1)
                 ADD.D   F4,F0,F2
                 S.D     0(R1),F4
Iteration i+2:   L.D     F0,0(R1)
                 ADD.D   F4,F0,F2
                 S.D     F4,0(R1)
```

The selected instructions from different iterations are then put together in the loop with the loop control instructions:

```
Loop:  S.D     F4,16(R1)      ;stores into M[i]
       ADD.D   F4,F0,F2       ;adds to M[i-1]
       L.D     F0,0(R1)       ;loads M[i-2]
       DADDUI  R1,R1,#-8
       BNE     R1,R2,Loop
```

This loop can be run at a rate of 5 cycles per result, ignoring the start-up and clean-up portions, and assuming that DADDUI is scheduled after the ADD.D and the L.D instruction, with an adjusted offset, is placed in the branch delay slot. Because the load and store are separated by offsets of 16 (two iterations), the loop should run for two fewer iterations. (We address this and the start-up and clean-up portions in Exercise 4.18.) Notice that the reuse of registers (e.g., F4, F0, and R1) requires the hardware to avoid the WAR hazards that would occur in the loop. This hazard should not be a problem in this case, since no data-dependent stalls should occur.

By looking at the unrolled version we can see what the start-up code and finish code will need to be. For start-up, we will need to execute any instructions that correspond to iteration 1 and 2 that will not be executed. These instructions are the L.D for iterations 1 and 2 and the ADD.D for iteration 1. For the finish code, we need to execute any instructions that will not be executed in the final two iterations. These include the ADD.D for the last iteration and the S.D for the last two iterations. n

Register management in software-pipelined loops can be tricky. The example above is not too hard since the registers that are written on one loop iteration are read on the next. In other cases, we may need to increase the number of iterations between when we issue an instruction and when the result is used. This increase is required when there are a small number of instructions in the loop body and the latencies are large. In such cases, a combination of software pipelining and loop unrolling is needed. An example of this is shown in the Exercises.

Software pipelining can be thought of as *symbolic* loop unrolling. Indeed, some of the algorithms for software pipelining use loop-unrolling algorithms to figure out how to software pipeline the loop. The major advantage of software pipelining over straight loop unrolling is that software pipelining consumes less code space. Software pipelining and loop unrolling, in addition to yielding a better scheduled inner loop, each reduce a different type of overhead. Loop unrolling reduces the overhead of the loop—the branch and counter-update code. Software pipelining reduces the time when the loop is not running at peak speed to once per loop at the beginning and end. If we unroll a loop that does 100 iterations a constant number of times, say 4, we pay the overhead 100/4 = 25 times—every time the inner unrolled loop is initiated. Figure 4.7 shows this behavior graphically. Because these techniques attack two different types of overhead, the best performance can come from doing both.In practice, compilation using software pipelining is quite difficult for several reasons: many loops require significant transformation before they can be software pipelined, the tradeoffs in terms of overhead versus efficiency of the software-Pipelined loop are complex, and the issue of register management creates additional complexities. To help deal with the last two of these issues, the IA-64 added extensive hardware support for software pipelining. Although this hardware can make it more efficient to apply software pipelining, it does not eliminate the need for complex compiler support, or for the need to make difficult decisions about the best way to compile a loop.

Global Code Scheduling

In section 4.1 we examined the use of loop unrolling and code scheduling to improve ILP. The techniques in section 4.1 work well when the loop body is straightline code, since the resulting unrolled loop looks like a single basic block. Similarly, software pipelining works well when the body is single basic block, since it is easier to find the repeatable schedule. When the body of an unrolled loop contains internal control flow, however, scheduling the code is much more complex. In general, effective scheduling of a loop body with internal control flow will require moving instructions across branches, which is global code scheduling. In this section, we first examine the challenge and limitations of global code scheduling. In section 4.5 we examine hardware support for eliminating control flow within an inner loop; then, we examine two compiler techniques that can be used when eliminating the control flow is not a viable approach.

Global code scheduling aims to compact a code fragment with internal control structure into the shortest possible sequence that preserves the data and control dependences. The data dependences force a partial order on operations, while the control dependences dictate instructions across which code cannot be easily moved. Data dependences are overcome by unrolling and, in the case of memory operations, using dependence analysis to determine if two references refer to the same address. Finding the shortest possible sequence for a piece of code means

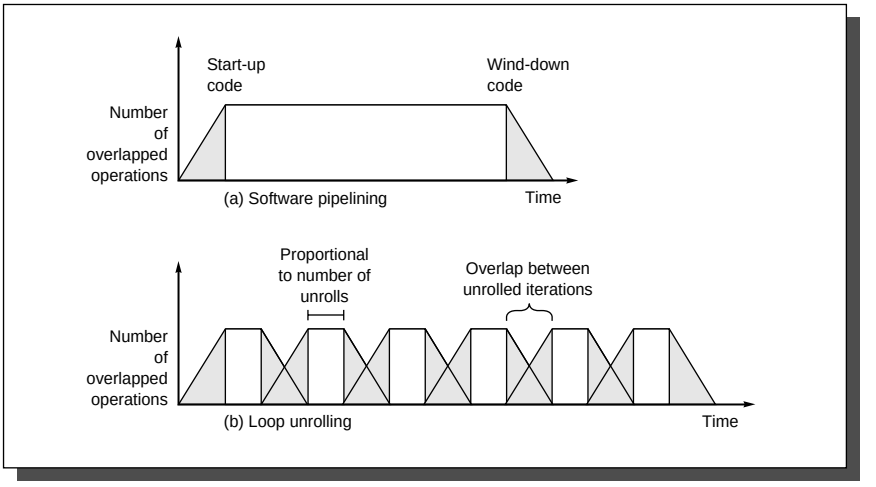


FIGURE 4.7 The execution pattern for (a) a software-pipelined loop and (b) an unrolled loop. The shaded areas are the times when the loop is not running with maximum overlap or parallelism among instructions. This occurs once at the beginning and once at the end for the software-pipelined loop. For the unrolled loop it occurs *m/n* times if the loop has a total of *m* iterations and is unrolled *n* times. Each block represents an unroll of *n* iterations. Increasing the number of unrollings will reduce the start-up and clean-up overhead. The overhead of one iteration overlaps with the overhead of the next, thereby reducing the impact. The total area under the polygonal region in each case will be the same, since the total number of operations is just the execution rate multiplied by the time.

finding the shortest sequence for the *critical path*, which is the longest sequence of dependent instructions.

Control dependences arising from loop branches are reduced by unrolling. Global code scheduling can reduce the effect of control dependences arising from conditional nonloop branches by moving code. Since moving code across branches will often affect the frequency of execution of such code, effectively using global code motion requires estimates of the relative frequency of different paths. Although global code motion cannot guarantee faster code, if the frequency information is accurate, the compiler can determine whether such code movement is likely to lead to faster code.

Global code motion is important since many inner loops contain conditional statements. Figure 4.8 shows a typical code fragment, which may be thought of as an iteration of an unrolled loop and highlights the more common control flow.

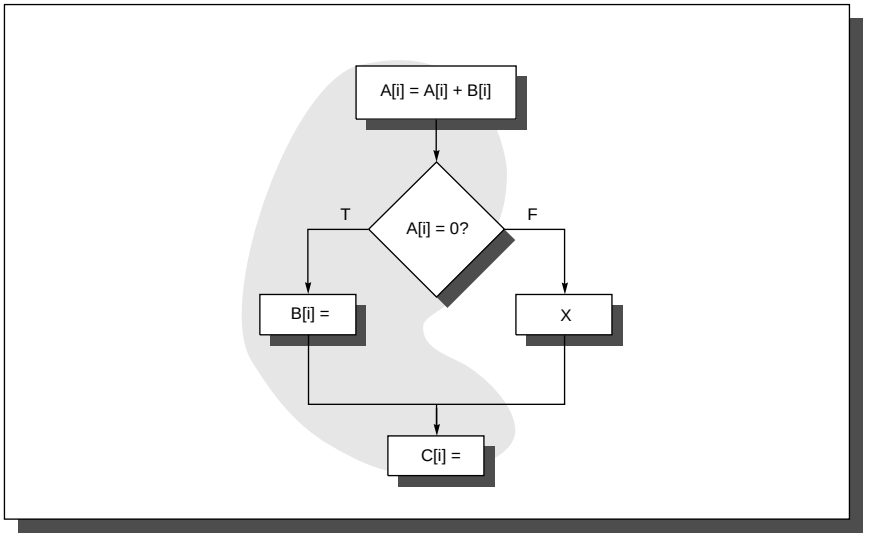


FIGURE 4.8 A code fragment and the common path shaded with gray. Moving the assignments to B or C requires a more complex analysis than for straightline code. In this section we focus on scheduling this code segment efficiently without hardware assistance. Predication or conditional instructions, which we discuss in the next section, provide another way to schedule this code.

Effectively scheduling this code could require that we move the assignments to B and C to earlier in the execution sequence, before the test of A. Such global code motion must satisfy a set of constraints to be legal. In addition, the movement of the code associated with B, unlike that associated with C, is speculative: it will speed the computation up only when the path containing the code would be taken.

To perform the movement of B, we must ensure that neither the data flow nor the exception behavior is changed. Compilers avoid changing the exception behavior by not moving certain classes of instructions, such as memory references, that can cause exceptions. In section 4.5, we will see how hardware support allow for more opportunities for speculative code motion as well as remove control dependences. Although such enhanced support for speculation can make it possible to explore more opportunities, the difficulty of choosing how to best compile the code remains complex.

How can the compiler ensure that the assignments to B and C can be moved without affecting the data flow? To see what's involved, let's look at a typical code generation sequence for the flowchart in Figure 4.8. Assuming that the addresses for A, B, C are in R1, R2, and R3, respectively, here is such a sequence:

```
        LD      R4,0(R1)        ; load A
        LD      R5,0(R2)        ; load B
        DADDU   R4,R4,R5        ; Add to A
        SD      0(R1),R4        ; Store A
        ...
        BNEZ    R4,elsepart     ; Test A
        ...                     ; then part
        SD      0(R2),...       ; Stores to B
        J       join            ; jump over else
elsepart:...                    ; else part
        X                       ; code for X
        ...
join:   ...                     ; after if
        SD      0(R3),...       ; store C[i]
```

Let's first consider the problem of moving the assignment to B to before the BNEZ instruction. Call the last instruction to assign to B before the if statement, *i*. If B is referenced before it is assigned either in code segment X or after the if-statement, call the referencing instruction *j*. If there is such an instruction *j*, then moving the assignment to B will change the data flow of the program. In particular, moving the assignment to B will cause *j* to become data-dependent on the moved version of the assignment to B rather than on *i* on which *j* originally depended. One could imagine more clever schemes to allow B to be moved even when the value is used: for example, in the first case, we could make a shadow copy of B before the if statement and use that shadow copy in X. Such schemes are usually avoided, both because they are complex to implement and because they will slow down the program if the trace selected is not optimal and the operations end up requiring additional instructions to execute.

Moving the assignment to C up to before the first branch requires two steps. First, the assignment is moved over the join point of the else part into the portion corresponding to the then part. This movement makes the instructions for C control-dependent on the branch and means that they will not execute if the else path, which is the infrequent path, is chosen. Hence, instructions that were data-dependent on the assignment to C, and which execute after this code fragment, will be affected. To ensure the correct value is computed for such instructions, a copy is made of the instructions that compute and assign to C on the else path. Second, we can move C from the then part of the branch across the branch condition, if it does not affect any data flow into the branch condition. If C is moved to before the if-test, the copy of C in the else branch can usually be eliminated, since it will be redundant.

We can see from this example that global code scheduling is subject to many constraints. This observation is what led designers to provide hardware support to make such code motion easier, and section 4.5 explores such support in detail.

Global code scheduling also requires complex tradeoffs to make code motion decisions. For example, assuming that the assignment to B can be moved before the conditional branch (possibly with some compensation code on the alternative branch) will this movement make the code run faster? The answer is: possibly! Similarly, moving the copies of C into the if and else branches, makes the code initially bigger! Only if the compiler can successfully move the computation across the if-test will there be a likely benefit.

Consider the factors that the compiler would have to consider in moving the computation and assignment of B:

- What are the relative execution frequencies of the then-case and the else-case in the branch? If the then-case is much more frequent, the code motion may be beneficial. If not, it is less likely, although not impossible to consider moving the code.
- What is the cost of executing the computation and assignment to B above the branch? It may be that there are a number of empty instruction issue slots in the code above the branch and that the instructions for B can be placed into these slots that would otherwise go empty. This opportunity makes the computation of B "free" at least to first order.
- How will the movement of B change the execution time for the then-case? If B is at the start of the critical path for the then-case, moving it may be highly beneficial.
- Is B the best code fragment that can be moved above the branch? How does it compare with moving C or other statements within the then-case?
- What is the cost of the compensation code that may be necessary for the else-case? How effectively can this code be scheduled and what is its impact on execution time?

As we can see from this *partial* list, global code scheduling is an extremely complex problem. The tradeoffs depend on many factors and individual decisions to globally schedule instructions are highly interdependent. Even choosing which instructions to start considering as candidates for global code motion is complex!

To try to simplify this process, several different methods for global code scheduling have been developed. The two methods we briefly explore here rely on a simple principle: focus the attention of the compiler on a straightline code segment representing what is estimated to be the most frequently executed code path. Unrolling is used to generate the straightline code, but, of course, the complexity arises in how conditional branches are handled. In both cases, they are effectively straightened by choosing and scheduling the most frequent path.

Trace Scheduling: Focusing on the Critical Path

Trace scheduling is useful for processors with a large number of issues per clock, where conditional or predicated execution (see Section 4.5) is inappropriate or unsupported, and where simple loop unrolling may not be sufficient by itself to uncover enough ILP to keep the processor busy. Trace scheduling is a way to organize the global code motion process, so as to simplify the code scheduling by incurring the costs of possible code motion on the less frequent paths. Because it can generate *significant* overheads on the designated infrequent path, it is best used where profile information indicates significant differences in frequency between different paths and where the profile information is highly indicative of program behavior independent of the input. Of course, this limits its effective applicability to certain classes of programs.

There are two steps to trace scheduling. The first step, called *trace selection*, tries to find a likely sequence of basic blocks whose operations will be put together into a smaller number of instructions; this sequence is called a *trace*. Loop unrolling is used to generate long traces, since loop branches are taken with high probability. Additionally, by using static branch prediction, other conditional branches are also chosen as taken or not taken, so that the resultant trace is a straight-line sequence resulting from concatenating many basic blocks. If, for example, the program fragment shown in Figure 4.8 on page 253 corresponds to an inner loop with the highlighted path being much more frequent, and the loop were unwound four times, the primary trace would consist of four copies of the shaded portion of the program, as shown in Figure 4.9. .

Once a trace is selected, the second process, called *trace compaction*, tries to squeeze the trace into a small number of wide instructions. Trace compaction is code scheduling; hence, it attempts to move operations as early as it can in a sequence (trace), packing the operations into as few wide instructions (or issue packets) as possible.

The advantage of the trace scheduling approach is that it simplifies the decisions concerning global code motion. In particular, branches are viewed as jumps into or out of the selected trace, which is assumed to the most probable path. When code is moved across such trace entry and exit points, additional bookkeeping code will often be needed on the entry or exit point. The key assumption is that the trace is so much more probable than the alternatives that the cost of the bookkeeping code need not be a deciding factor: if an instruction can be moved and make the main trace execute faster, it is moved.

Although trace scheduling has been successfully applied to scientific code with its intensive loops and accurate profile data, it remains unclear whether this approach is suitable for programs that are less simply characterized and less loop-intensive. In such programs, the significant overheads of compensation code may make trace scheduling an unattractive approach, or, at best, its effective use will be extremely complex for the compiler.

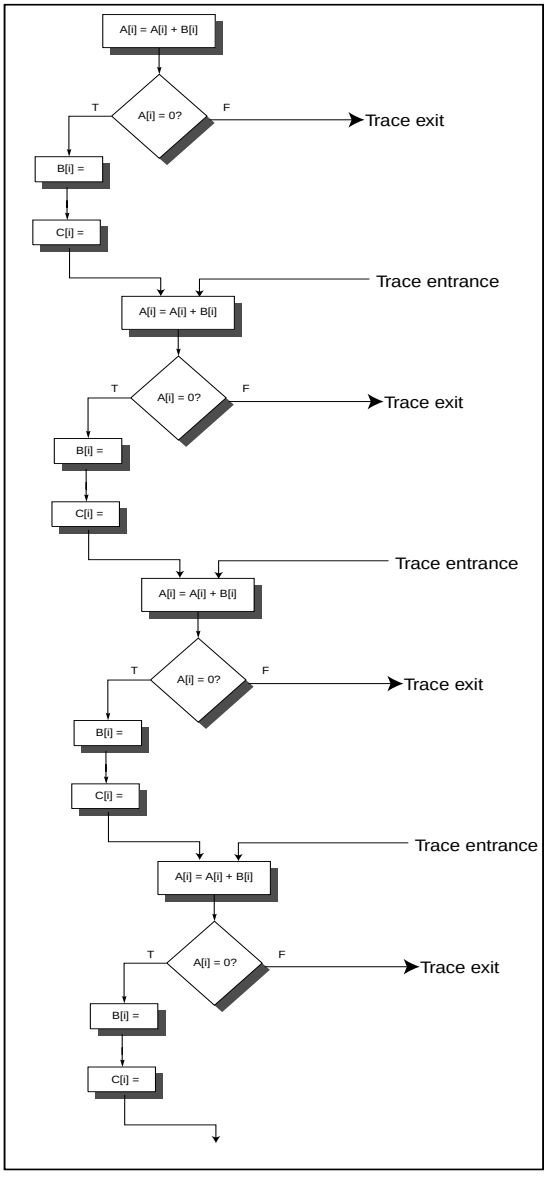


FIGURE 4.9 This trace is obtained by assuming that the program fragment in Figure 4.8 is the inner loop and unwinding it four times treating the shaded portion in Figure 4.8 as the likely path. The trace exits correspond to jumps off the frequent path, and the trace entrances correspond to returns to the trace.

Superblocks

One of the major drawbacks of trace scheduling is that the entries and exits into the middle of the trace cause significant complications requiring the compiler to generate and track the compensation code and often making it difficult to assess the cost of such code. *Superblocks* are formed by a process similar to that used for traces, but, are a form of extended basic blocks, which are restricted to have a a single entry point but allow multiple exits.

Because superblocks have only a single entry point, compacting a superblock is easier than compacting a trace since only code motion across an exit need be considered. In our earlier example, we would form superblock that did not contain any entrances and hence, moving C would be easier. Furthermore, in loops that have a single loop exit based on a count (for example, a for-loop with no loop exit other than the loop termination condition), the resulting superblocks have only one exit as well as one entrance. Such blocks can then be scheduled more easily.

How can a superblock with only one entrance be constructed? The answer is to use *tail duplication* to create a separate block that corresponds to the portion of the trace after the entry. In our example above, each unrolling of the loop would create an exit from the superblock to a residual loop that handles the remaining iterations. Figure 4.10 shows the superblock structure if the code fragment from Figure 4.8 is treated as the body of an inner loop and unrolled four times. The residual loop handles any iterations that occur if the superblock is exited, which, in turn, occurs when the unpredicted path is selected. If the expected frequency of the residual loop were still high, a superblock could be created for that loop as well.

The superblock approach reduces the complexity of bookkeeping and scheduling versus the more general trace generation approach, but may enlarge code size more than a trace-based approach. Like trace scheduling, superblock scheduling my be most appropriate when other techniques (if-conversion, e.g.) fail. Even in such cases, assessing the cost of code duplication may limit the usefulness of the approach and will certainly complicate the compilation process.

Loop unrolling, software pipelining, trace scheduling, and superblock scheduling all aim at trying to increase the amount of ILP that can be exploited by a processor issuing more than one instruction on every clock cycle. The effectiveness of each of these techniques and their suitability for various architectural approaches are among the hottest topics being actively pursued by researchers and designers of high-speed processors.

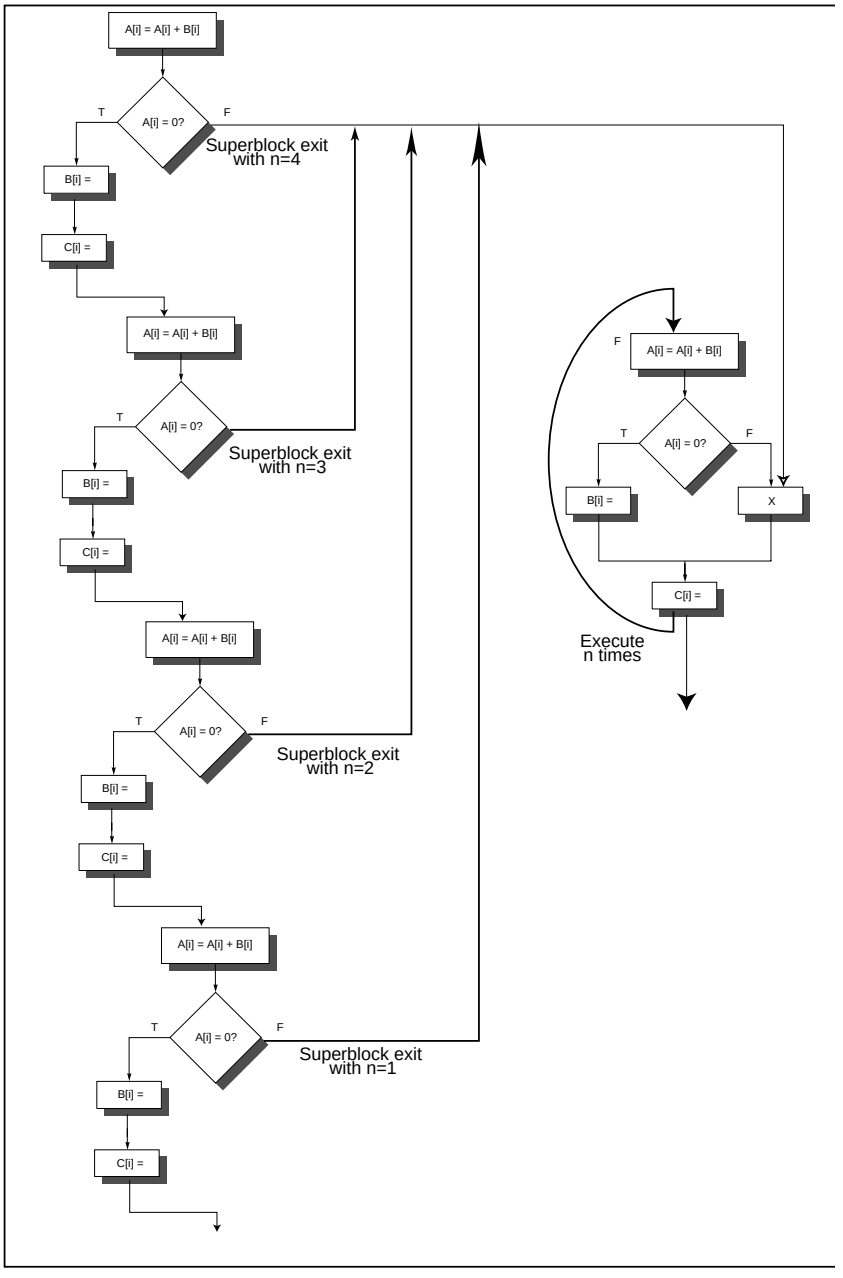


FIGURE 4.10 This superblock results from unrolling the code in Figure 4.8 four times and creating a superblock.

4.5 Hardware Support for Exposing More Parallelism at Compile-Time

Techniques such as loop unrolling, software pipelining, and trace scheduling can be used to increase the amount of parallelism available when the behavior of branches is fairly predictable at compile time. When the behavior of branches is not well known, compiler techniques alone may not be able to uncover much ILP. In such cases, the control dependences may severely limit the amount of parallelism that can be exploited. Similarly, potential dependences between memory reference instructions could prevent code movement that would increase available ILP. This section introduces several techniques that can help overcome such limitations.

The first is an extension of the instruction set to include *conditional* or *predicated instructions*. Such instructions can be used to eliminate branches converting a control dependence into a data dependence and potentially improving performance. Such approaches are useful with either the hardware-intensive schemes of the last chapter or the software-intensive approaches discussed in this chapter, since in both cases, predication can be used to eliminate branches.

Hardware speculation with in-order commit preserved exception behavior by detecting and raising exceptions only at commit time when the instruction was no longer speculative. To enhance the ability of the *compiler* to speculatively move code over branches, while still preserving the exception behavior, we consider several different methods, which either include explicit checks for exceptions or techniques to ensure that only those exceptions that should arise are generated.

Finally, the hardware speculation schemes of the last chapter provided support for reordering loads and stores, by checking for potential address conflicts at runtime. To allow the compiler to reorder loads and stores when it suspects they do not conflict, but cannot be absolutely certain, a mechanism for checking for such conflicts can be added to the hardware. This mechanism permits additional opportunities for memory reference speculation.

Conditional or Predicated Instructions

The concept behind conditional instructions is quite simple: An instruction refers to a condition, which is evaluated as part of the instruction execution. If the condition is true, the instruction is executed normally; if the condition is false, the execution continues as if the instruction was a no-op. Many newer architectures include some form of conditional instructions. The most common example of such an instruction is conditional move, which moves a value from one register to another if the condition is true. Such an instruction can be used to completely eliminate a branch in simple code sequences.

EXAMPLE Consider the following code:

```
if (A==0) {S=T;}
```

Assuming that registers R1, R2, and R3 hold the values of A, S, and T, respectively, show the code for this statement with the branch and with the conditional move.

ANSWER The straightforward code using a branch for this statement is (remember that we are assuming normal rather than delayed branches)

```
        BNEZ   R1,L
        ADDU   R2,R3,R0
L:
```

Using a conditional move that performs the move only if the third operand is equal to zero, we can implement this statement in one instruction:

```
        CMOVZ  R2,R3,R1
```

The conditional instruction allows us to convert the control dependence present in the branch-based code sequence to a data dependence. (This transformation is also used for vector computers, where it is called *if-conversion*.) For a pipelined processor, this moves the place where the dependence must be resolved from near the front of the pipeline, where it is resolved for branches, to the end of the pipeline where the register write occurs. n

One obvious use for conditional move is to implement the absolute value function: `A = abs (B)`, which is implemented as `if (B<0) {A=-B;) else {A=B;}`. This if statement can be implemented as a pair of conditional moves, or as one unconditional move (`A=B`) and one conditional move (`A=-B`).

In the example above or in the compilation of absolute value, conditional moves are used to change a control dependence into a data dependence. This enables us to eliminate the branch and possibly improve the pipeline behavior. As issue rates increase, designers are faced with one of two choices: execute multiple branches per clock cycle or find a method to eliminate branches to avoid this requirement. Handling multiple branches per clock is complex, since one branch must be control dependent on the other. The difficulty of accurately predicting two branch outcomes, updating the prediction tables, and executing the correct sequence, has so far caused most designers to avoid processors that execute multiple branches per clock. Conditional moves and predicated instructions provide a way of reducing the branch pressure. In addition, a conditional move can often eliminate a branch that is hard to predict, increasing the potential gain.

Conditional moves are the simplest form of conditional or predicated instructions, and although useful for short sequences, have limitations. In particular, using conditional move to eliminate branches that guard the execution of large blocks of code can be inefficient, since many conditional moves may need to be introduced.

To remedy the inefficiency of using conditional moves, some architectures support full predication, whereby the execution of all instructions is controlled by a predicate. When the predicate is false, the instruction becomes a no-op. Full predication allows us to simply convert large blocks of code that are branch dependent. For example, an if-then-else statement within a loop can be entirely converted to predicated execution, so that the code in the then-case executes only if the value of the condition is true, and the code in the else-case executes only if the value of the condition is false. Predication is particularly valuable with global code scheduling, since it can eliminate nonloop branches, which significantly complicate instruction scheduling.

Predicated instructions can also be used to speculatively move an instruction that is time-critical, but may cause an exception if moved before a guarding branch. Although it is possible to do this with conditional move, it is more costly, as we explore in the exercises.

EXAMPLE Here is a code sequence for a two-issue superscalar that can issue a combination of one memory reference and one ALU operation, or a branch by itself, every cycle:

First instruction slot	Second instruction slot
`LW R1,40(R2)`	`ADD R3,R4,R5`
	`ADD R6,R3,R7`
`BEQZ R10,L`	
`LW R8,0(R10)`	
`LW R9,0(R8)`	

This sequence wastes a memory operation slot in the second cycle and will incur a data dependence stall if the branch is not taken, since the second `LW` after the branch depends on the prior load. Show how the code can be improved using a predicated form of `LW`.

ANSWER Call the predicated version load word `LWC` and assume the load occurs unless the third operand is 0. The `LW` immediately following the branch can be converted to a `LWC` and moved up to the second issue slot:

First instruction slot	Second instruction slot
`LW R1,40(R2)`	`ADD R3,R4,R5`
`LWC R8,20(R10),R10`	`ADD R6,R3,R7`
`BEQZ R10,L`	
`LW R9,0(R8)`	

This improves the execution time by several cycles since it eliminates one instruction issue slot and reduces the pipeline stall for the last instruction in the sequence. Of course, if the compiler mispredicted the branch, the predicated instruction will have no effect and will not improve the running time. This is why the transformation is speculative.

If the sequence following the branch were short, the entire block of code might be converted to predicated execution, and the branch eliminated. ■

When we convert an entire code segment to predicated execution or speculatively move an instruction and make it predicted, we remove a control dependence. Correct code generation and the conditional execution of predicated instructions ensure that we maintain the data flow enforced by the branch. To ensure that the exception behavior is also maintained, a predicated instruction must not generate an exception if the predicate is false. The property of not causing exceptions is quite critical, as the Example above shows: If register `R10` contains zero, the instruction `LW R8,0(R10)` executed unconditionally is likely to cause a protection exception, and this exception should not occur. Of course, if the condition is satisfied (i.e. `R10` is not zero), the `LW` may still cause a legal and resumable exception (e.g., a page fault), and the hardware must take the exception when it knows that the controlling condition is true.

The major complication in implementing predicated instructions is deciding when to annul an instruction. Predicated instructions may either be annulled during instruction issue or later in the pipeline before they commit any results or raise an exception. Each choice has a disadvantage. If predicated instructions are annulled early in the pipeline, the value of the controlling condition must be known early to prevent a stall for a data hazard. Since data dependent branch conditions, which tend to be less predictable, are candidates for conversion to predicated execution, this choice can lead to more pipeline stalls. Because of this potential for data hazard stalls, no design with predicated execution (or conditional move) annuls instructions early. Instead, all existing processors annul instructions later in the pipeline, which means that annulled instructions will consume functional unit resources and potentially have a negative impact on performance. A variety of other pipeline implementation techniques, such as forwarding, interact with predicated instructions further complicating the implementation.

Predicated or conditional instructions are extremely useful for implementing short alternative control flows, for eliminating some unpredictable branches, and for reducing the overhead of global code scheduling. Nonetheless, the usefulness of conditional instructions is limited by several factors:

- Predicated instructions that are annulled (i.e., whose conditions are false) still take some processor resources. An annulled predicated instruction requires fetch resources at a minimum, and in most processors functional unit execution

time. Therefore, moving an instruction across a branch and making it conditional will slow the program down whenever the moved instruction would not have been normally executed. Likewise, predicating a control dependent portion of code and eliminating a branch may slow down the processor if that code would not have been executed. An important exception to these situations occurs when the cycles used by the moved instruction when it is not performed would have been idle anyway (as in the superscalar example above). Moving an instruction across a branch or converting a code segment to predicated execution is essentially speculating on the outcome of the branch. Conditional instructions make this easier but do not eliminate the execution time taken by an incorrect guess. In simple cases, where we trade a conditional move for a branch and a move, using conditional moves or predication is almost always better. When longer code sequences are made conditional, the benefits are more limited.

- Predicated instructions are most useful when the predicate can be evaluated early. If the condition evaluation and predicated instructions cannot be separated (because of data dependences in determining the condition), then a conditional instruction may result in a stall for a data hazard. With branch prediction and speculation, such stalls can be avoided, at least when the branches are predicted accurately.
- The use of conditional instructions can be limited when the control flow involves more than a simple alternative sequence. For example, moving an instruction across multiple branches requires making it conditional on both branches, which requires two conditions to be specified or requires additional instructions to compute the controlling predicate. If such capabilities are not present, the overhead of if-conversion will be larger, reducing its advantage.
- Conditional instructions may have some speed penalty compared with unconditional instructions. This may show up as a higher cycle count for such instructions or a slower clock rate overall. If conditional instructions are more expensive, they will need to be used judiciously.

For these reasons, many architectures have included a few simple conditional instructions (with conditional move being the most frequent), but only a few architectures include conditional versions for the majority of the instructions. The MIPS, Alpha, Power-PC, SPARC and Intel x86 (as defined in the Pentium processor) all support conditional move. The IA-64 architecture supports full predication for all instructions, as we will see section 4.7.

Compiler Speculation with Hardware Support

As we saw earlier in this chapter, many programs have branches that can be accurately predicted at compile time either from the program structure or by using a profile. In such cases, the compiler may want to speculate either to improve the

scheduling or to increase the issue rate. Predicated instructions provide one method to speculate, but they are really more useful when control dependences can be completely eliminated by if-conversion. In many cases, we would like to move speculated instructions not only before branch, but before the condition evaluation, and predication cannot achieve this.

As pointed out earlier, to speculate ambitiously requires three capabilities:

1. the ability of the compiler to find instructions that, with the possible use of register renaming, can be speculatively moved and not affect the program data flow,
2. the ability to ignore exceptions in speculated instructions, until we know that such exceptions should really occur, and
3. the ability to speculatively interchange loads and stores, or stores and stores, which may have address conflicts.

The first of these is a compiler capability, while the last two require hardware support, which we explore next.

Hardware Support for Preserving Exception Behavior

To speculate ambitiously, we must be able to move any type of instruction and still preserve its exception behavior. The key to being able to do this is to observe that the results of a speculated sequence that is mispredicted will not be used in the final computation, and such a speculated instruction should not cause an exception.

There are four methods that have been investigated for supporting more ambitious speculation without introducing erroneous exception behavior:

1. The hardware and operating system cooperatively ignore exceptions for speculative instructions. As we will see below, this approach preserves exception behavior for correct programs, but not for incorrect ones. This approach may be viewed as unacceptable for some programs, but it has been used, under program control, as a “fast mode” in several processors.
2. Speculative instructions that never raise exceptions are used, and checks are introduced to determine when an exception should occur.
3. A set of status bits, called *poison bits*, are attached to the result registers written by speculated instructions when the instructions cause exceptions. The poison bits cause a fault when a normal instruction attempts to use the register.
4. A mechanism is provided to indicate that an instruction is speculative and the hardware buffers the instruction result until it is certain that the instruction is no longer speculative.

To explain these schemes, we need to distinguish between exceptions that indicate a program error and would normally cause termination, such as a memory protection violation, and those that are handled and normally resumed, such as a page fault. Exceptions that can be resumed can be accepted and processed for speculative instructions just as if they were normal instructions. If the speculative instruction should not have been executed, handling the unneeded exception may have some negative performance effects, but it cannot cause incorrect execution. The cost of these exceptions may be high, however, and some processors use hardware support to avoid taking such exceptions, just as processors with hardware speculation may take some exceptions in speculative mode, while avoiding others until an instruction is known not to be speculative.

Exceptions that indicate a program error should not occur in correct programs, and the result of a program that gets such an exception is not well defined, except perhaps when the program is running in a debugging mode. If such exceptions arise in speculated instructions, we cannot take the exception until we know that the instruction is no longer speculative.

In the simplest method for preserving exceptions, the hardware and the operating system simply handle all resumable exceptions when the exception occurs and simply return an undefined value for any exception that would cause termination. If the instruction generating the terminating exception was not speculative, then the program is in error. Note that instead of terminating the program, the program is allowed to continue, though it will almost certainly generate incorrect results. If the instruction generating the terminating exception is speculative, then the program may be correct and the speculative result will simply be unused; thus, returning an undefined value for the instruction cannot be harmful. This scheme can never cause a correct program to fail, no matter how much speculation is done. An incorrect program, which formerly might have received a terminating exception, will get an incorrect result. This is acceptable for some programs, assuming the compiler can also generate a normal version of the program, which does not speculate and can receive a terminating exception.

EXAMPLE Consider the following code fragment from an if-then-else statement of the form

```
if (A==0) A = B; else A = A+4;
```

where `A` is at `0(R3)` and `B` is at `0(R2)`:

```
        LD      R1,0(R3)    ;load A
        BNEZ    R1,L1       ;test A
        LD      R1,0(R2)    ;then clause
        J       L2          ;skip else
L1:     DADDI   R1,R1,#4    ;else clause
L2:     SD      0(R3),R1    ;store A
```

Assume the then clause is *almost always* executed. Compile the code using compiler-based speculation. Assume `R14` is unused and available.

ANSWER Here is the new code:

```
        LD      R1,0(R3)    ;load A
        LD      R14,0(R2)   ;speculative load B
        BEQZ    R1,L3       ;other branch of the if
        DADDI   R14,R1,#4   ;the else clause
L3:     SD      0(R3),R14   ;nonspeculative store
```

The then clause is completely speculated. We introduce a temporary register to avoid destroying R1 when B is loaded; if the load is speculative R14 will be useless. After the entire code segment is executed, A will be in R14. The else clause could have also been compiled speculatively with a conditional move, but if the branch is highly predictable and low cost, this might slow the code down, since two extra instructions would always be executed as opposed to one branch. n

In such a scheme, it is not necessary to know that an instruction is speculative. Indeed, it is helpful only when a program is in error and receives a terminating exception on a normal instruction; in such cases, if the instruction were not marked as speculative, the program could be terminated.

In this method for handling speculation, as in the next one, renaming will often be needed to prevent speculative instructions from destroying live values. Renaming is usually restricted to register values. Because of this restriction, the targets of stores cannot be destroyed and stores cannot be speculative. The small number of registers and the cost of spilling will act as one constraint on the amount of speculation. Of course, the major constraint remains the cost of executing speculative instructions when the compiler's branch prediction is incorrect.

A second approach to preserving exception behavior when speculating introduces speculative versions of instructions that do not generate terminating exceptions and instructions to check for such exceptions. This combination preserves the exception behavior exactly.

EXAMPLE Show how the previous example can be coded using a speculative load (sLD) and a speculation check instruction (SPECCK) to completely preserve exception behavior. Assume R14 is unused and available.

ANSWER Here is the code that achieves this:

```
        LD      R1,0(R3)    ;load A
        sLD     R14,0(R2)   ;speculative, no termination
        BNEZ    R1,L1       ;test A
        SPECCK  0(R2)       ;perform speculation check
```

```
        J       L2          ;skip else
L1:     DADDI   R14,R1,#4   ;else clause
L2:     SD      0(R3),R14   ;store A
```

Notice that the speculation check requires that we maintain a basic block for the then-case. If we had speculated only a portion of the then-case, then a basic block representing the then-case would exist in any event. More importantly, notice that checking for a possible exception requires extra code. n

A third approach for preserving exception behavior tracks exceptions as they occur but postpones any terminating exception until a value is actually used, preserving the occurrence of the exception, although not in a completely precise fashion. The scheme is simple: A poison bit is added to every register and another bit is added to every instruction to indicate whether the instruction is speculative. The poison bit of the destination register is set whenever a speculative instruction results in a terminating exception; all other exceptions are handled immediately. If a speculative instruction uses a register with a poison bit turned on, the destination register of the instruction simply has its poison bit turned on. If a normal instruction attempts to use a register source with its poison bit turned on, the instruction causes a fault. In this way, any program that would have generated an exception still generates one, albeit at the first instance where a result is used by an instruction that is not speculative. Since poison bits exist only on register values and not memory values, stores are never speculative and thus trap if either operand is "poison."

EXAMPLE Consider the code fragment from page 267 and show how it would be compiled with speculative instructions and poison bits. Show where an exception for the speculative memory reference would be recognized. Assume R14, is unused and available.

ANSWER Here is the code (an "s" proceeding the opcode indicates a speculative instruction):

```
        LD      R1,0(R3)      ;load A
        sLD     R14,0(R2)     ;speculative load B
        BEQZ    R1,L3         ;
        DADDI   R14,R1,#4     ;
L3:     SD      0(R3),R14     ;exception for speculative LW
```

If the speculative `sLD` generates a terminating exception, the poison bit of `R14` will be turned on. When the nonspeculative `SW` instruction occurs, it will raise an exception if the poison bit for `R14` is on. n

One complication that must be overcome is how the OS saves the user registers on a context switch if the poison bit is set. A special instruction is needed to save and reset the state of the poison bits to avoid this problem.

The fourth and final approach listed above relies on a hardware mechanism that operates like a reorder buffer. In such an approach, instructions are marked by the compiler as speculative and include an indicator of how many branches the instruction was speculatively moved across and what branch action (taken/not taken) the compiler assumed. This last piece of information basically tells the hardware the location of the code block where the speculated instruction originally was. In practice, most of the benefit of speculation is gained by allowing movement across a single branch, and, thus, only one bit saying whether the speculated instruction came from the taken or not taken path is required. Alternatively, the original location of the speculative instruction is marked by a *sentinel*, which tells the hardware that the earlier speculative instruction is no longer speculative and values may be committed.

All instructions are placed in a reorder buffer when issued and are forced to commit in order, as in a hardware speculation approach. (Notice, though that no actual speculative branch prediction or dynamic scheduling occurs.) The reorder buffer tracks when instructions are ready to commit and delays the "write back" portion of any speculative instruction. Speculative instructions are not allowed to commit until the branches they have been speculatively moved over are also ready to commit, or, alternatively, until the corresponding sentinel is reached. At that point, we know whether the speculated instruction should have been executed or not. If it should have been executed and it generated a terminating exception, then we know that the program should be terminated. If the instruction should not have been executed, then the exception can be ignored. Notice that the compiler, rather than the hardware, has the job of register renaming to ensure correct usage of the speculated result, as well as correct program execution.

Hardware Support for Memory Reference Speculation

Moving loads across stores is usually done when the compiler is certain the addresses do not conflict. As we saw with the examples in section 4.1, such transformations are critical to reducing the critical path length of a code segment. To allow the compiler to undertake such code motion, when it cannot be absolutely certain that such a movement is correct, a special instruction to check for address conflicts can be included in the architecture. The special instruction is left at the original location of the load instruction (and acts like a guardian) and the load is moved up across one or more stores.

When a speculated load is executed, the hardware saves the address of the accessed memory location. If a subsequent store changes the location before the check instruction, then the speculation has failed. If the location has not been touched then the speculation is successful. Speculation failure can be handled in two ways. If only the load instruction was speculated, then it suffices to redo the load at the point of the check instruction (which could supply the target register in addition to the memory address). If additional instructions that depended on the load were also speculated, then a fix-up sequence that re-executes all the

speculated instructions starting with the load is needed. In this case, the check instruction specifies the address where the fix-up code is located.

In this section we have seen a variety of hardware assist mechanisms. Such mechanisms are key to achieving good support with the compiler intensive approaches of this chapter. In addition, several of them can be easily integrated in the hardware-intensive approaches of the prior chapter and provide additional benefits.

4.6 Crosscutting Issues

Hardware versus Software Speculation Mechanisms

The hardware-intensive approaches to speculation in the previous chapter and the software approaches of this chapter provide alternative approaches to exploiting ILP. Some of the tradeoffs, and the limitations, for these approaches are listed below:

- To speculate extensively, we must be able to disambiguate memory references. This capability is difficult to do at compile time for integer programs that contain pointers. In a hardware-based scheme, dynamic runtime disambiguation of memory addresses is done using the techniques we saw earlier for Tomasulo's algorithm. This disambiguation allows us to move loads past stores at runtime. Support for speculative memory references can help overcome the conservatism of the compiler, but unless such approaches are used carefully, the overhead of the recovery mechanisms may swamp the advantages.
- Hardware-based speculation works better when control flow is unpredictable, and when hardware-based branch prediction is superior to software-based branch prediction done at compile time. These properties hold for many integer programs. For example, a good static predictor has a misprediction rate of about 16% for four major integer SPEC92 programs, and a hardware predictor has a misprediction rate of under 10%. Because speculated instructions may slow down the computation when the prediction is incorrect, this difference is significant. One result of this difference is that even statically scheduled processors normally include dynamic branch predictors.
- Hardware-based speculation maintains a completely precise exception model even for speculated instructions. Recent software-based approaches have added special support to allow this as well.
- Hardware-based speculation does not require compensation or bookkeeping code, which is needed by ambitious software speculation mechanisms.
- Compiler-based approaches may benefit from the ability to see further in the code sequence, resulting in better code scheduling than a purely hardware-driv-

en approach.

- Hardware-based speculation with dynamic scheduling does not require different code sequences to achieve good performance for different implementations of an architecture. Although this advantage is the hardest to quantify, it may be the most important in the long run. Interestingly, this was one of the motivations for the IBM 360/91. On the other hand, more recent explicitly parallel architectures, such as IA-64, have added flexibility that reduces the hardware dependence inherent in a code sequence.

Against these advantages stands a major disadvantage: supporting speculation in hardware is complex and requires additional hardware resources. This hardware cost must be evaluated against both the complexity of a compiler for a software-based approach and the amount and usefulness of the simplifications in a processor that relies on such a compiler. We return to this topic in the concluding remarks.

Some designers have tried to combine the dynamic and compiler-based approaches to achieve the best of each. Such a combination can generate interesting and obscure interactions. For example, if conditional moves are combined with register renaming, a subtle side-effect appears. A conditional move that is annulled must still copy a value to the destination register, since it was renamed earlier in the instruction pipeline. These subtle interactions complicate the design and verification process and can also reduce performance. For example, in the Alpha 21264 this problem is overcome by mapping conditional to two instructions in the pipeline.

4.7 Putting It All Together: The Intel IA-64 Architecture and Itanium Processor

This section is an overview of the Intel IA-64 architecture and the initial implementation, the Itanium processor

The Intel IA-64 Instruction Set Architecture

The IA-64 is a RISC-style, register-register instruction set, but with many novel features designed to support compiler-based exploitation of ILP. Our focus here is on the unique aspects of the IA-64 ISA. Most of these aspects have been discussed already in this chapter, including predication, compiler-based parallelism detection, and support for memory reference speculation.

The IA-64 Register Model

The components of the IA-64 register state are:

- 128 64-bit general-purpose registers, which as we will see shortly are actually 65 bits wide;
- 128 82-bit floating point registers, which provides two extra exponent bits over the standard 80-bit IEEE format,
- 64 1-bit predicate registers,
- 8 64-bit branch registers, which are used for indirect branches, and
- a variety of registers used for system control, memory mapping, performance counters, and communication with the OS.

The integer registers are configured to help accelerate procedure calls using a register stack mechanism similar to that developed in the Berkeley RISC-I processor and used in the SPARC architecture. Registers 0-31 are always accessible and are addressed as 0-31. Registers 32-128 are used as a register stack and each procedure is allocated a set of registers (from 0 to 96) for its use.The new register stack frame is created for a called procedure by renaming the registers in hardware; a special register called the current frame pointer (CFM) points to the set of registers to be used by a given procedure. The frame consists of two parts: the local area and the output area. The local area is used for local storage, while the output area is used to pass values to any called procedure. The `alloc` instruction specifies the size of these areas. Only the integer registers have register stack support.

On a procedure call, the CFM pointer is updated so that R32 of the called procedure points to the first register of the output area of the called procedure. This update enables the parameters of the caller to be passed into the addressable registers of the callee. The callee executes an `alloc` instruction to allocate both the number of required local registers, which include the output registers of the caller, and the number of output registers needed for parameter passing to a called procedure. Special load and store instructions are available for saving and restoring the register stack, and special hardware (called the *register stack engine*) handles overflow of the register stack.

In addition to the integer registers, there are three other sets of registers: the floating point registers, the predicate registers, and the branch registers. The floating point registers are used for floating point data, and the branch registers are used to hold branch destination addresses for indirect branches. The predication registers hold predicates, which control the execution of predicated instructions; we describe the predication mechanism later in this section.

Both the integer and floating point registers support register rotation for registers 32-128. Register rotation is designed to ease the task of allocating of registers in software pipelined loops, a problem that we discussed in Section 4.4. In addition, when combined with the use of predication, it is possible to avoid the need for unrolling and for separate prologue and epilogue code for a software pipelined loop This capability reduces the code expansion incurred to use soft-

ware pipelining and makes the technique usable for loops with smaller numbers of iterations, where the overheads would traditionally negate many of the advantages.

Instruction Format and Support for Explicit Parallelism

The IA-64 architecture is designed to achieve the major benefits of a VLIW-approach–implicit parallelism among operations in an instruction and fixed formatting of the operation fields–while maintaining greater flexibility than a VLIW normally allows. This combination is achieved by relying on the compiler to detect ILP and schedule instructions into parallel instruction slots, but adding flexibility in the formatting of instructions and allowing the compiler to indicate when an instruction cannot be executed in parallel with its successors.

The IA-64 architecture uses two different concepts to achieve the benefits of implicit parallelism and ease of instruction decode. Implicit parallelism is achieved by placing instructions into *instruction groups*, while the fixed formatting of multiple instructions is achieved through the introduction of a concept called a *bundle*, which contains three instructions. Let's start by defining an instruction group.

An *instruction group* is a sequence of consecutive instructions with no register data dependences among them (there are a few minor exceptions). All the instructions in a group could be executed in parallel, if sufficient hardware resources existed and if any dependences through memory were preserved. An instruction group can be arbitrarily long, but the compiler must *explicitly* indicate the boundary between one instruction group and another. This boundary is indicated by placing a *stop* between two instructions that belong to different groups. To understand how stops are indicated, we must first explain how instructions are placed into bundles.

IA-64 instructions are encoded in *bundles*, which are 128 bits wide. Each bundle consists of a five-bit template field and three instructions, each 41 bits in length. To simply the decoding and instruction issue process, the template field of a bundle specifies what types of execution unit each instruction in the bundle requires. Figure 4.11 shows the five different execution unit types and describes what instruction classes they may hold, together with some examples.

The five-bit template field within each bundle describes *both* the presence of any stops associated with the bundle and the execution unit type required by each instruction within the bundle. Figure 4.12 shows the possible formats that the template field encodes and the position of any stops it specifies. The bundle formats can specify only a subset of all possible combinations of instruction types and stops. To see how the bundle works, let's consider an example.

EXAMPLE Unroll the array increment example, x[i] = x[i] +s (introduced on page 223), seven times (see page 236 for the unrolled code) and place the instructions into bundles, first ignoring pipeline latencies (to minimize the number

Execution Unit Slot	Instruction type	Instruction Description	Example Instructions
I-unit	A	Integer ALU	add, subtract, and, or, compare
	I	Non-ALU Integer	integer and multimedia shifts, bit tests, moves
M-unit	A	Integer ALU	add, subtract, and, or, compare
	M	Memory access	Loads and stores for integer/FP registers
F-unit	F	Floating point	Floating point instructions.
B-unit	B	Branches	Conditional branches, calls, loop branches
L+X	L+X	Extended	Extended immediates, stops and no-ops.

FIGURE 4.11 The five execution unit slots in the IA-64 architecture and what instructions types they may hold are shown. A-type instructions, which correspond to integer ALU instructions, may be placed in either a I-unit or M-unit slot. L+X slots are special, as they occupy two instruction slots; L+X instructions are used to encode 64-bit immediates, and a few special instructions. L+X instructions are executed either by the I-unit or the B-unit.

of bundles) and then scheduling the code to minimize stalls. In scheduling the code assume 1 bundle executes per clock and that any stalls cause the entire bundle to be stalled. Use the pipeline latencies from Figure 4.1 on page 222. Use MIPS instruction mnemonics for simplicity.

ANSWER The two different versions are shown in Figure 4.13. Although the latencies are different from those in Itanium, the most common bundle, MMF, must be issued by itself in Itanium, just as our example assumes. n

Instruction Set Basics

Before turning to the special support for speculation, we briefly discuss the major instruction encodings and survey the instructions in each of the primary five instruction classes (A, I, M, F, and B). Each IA-64 instruction is 41 bits in length. The high-order four bits, together with the bundle bits that specify the execution unit slot, are used as the major opcode. (That is, the four-bit opcode field is reused across the execution field slots, and it is appropriate to think of the opcode as being 4 bits + the M, F, I, B, L+X designation.) The low order six-bit of every instruction are used for specifying the predicate register that guards the instruction (see the next section).

Figure 4.14 summarizes most of the major instruction formats, other than the multimedia instructions, and gives examples of the instructions encoded for each format.

Predication and Speculation Support

The IA-64 architecture provides comprehensive support for predication: nearly every instruction in the IA-64 architecture can be predicated. An instruction is

Template	Slot 0	Slot 1	Slot 2
0	M	I	I
1	M	I	I
2	M	I	I
3	M	I	I
4	M	L	X
5	M	L	X
8	M	M	I
9	M	M	I
10	M	M	I
11	M	M	I
12	M	F	I
13	M	F	I
14	M	M	F
15	M	M	F
16	M	I	B
17	M	I	B
18	M	B	B
19	M	B	B
22	B	B	B
23	B	B	B
24	M	M	B
25	M	M	B
28	M	F	B
29	M	F	B

FIGURE 4.12 The 24 possible template values (8 possible values are reserved) and the instructions slots and stops for shown for each format. Stops are indicated by heavy lines and may appear *within and/or* at the end of the bundle. For example, template 9 specifies that the instructions slots are M, M, and I (in that order) and that the only stop is between this bundle and the next. Template 11 has the same type of instructions slots but also includes a stop after the first slot. The L+X format is used when slot 1 is L and slot 2 is X.

predicated by a specifying a predicate register, whose identity is placed in the lower six bits of each instruction field. Because nearly all instructions can be predicated, both if-conversion and code motion have lower overhead than they would with only limited support for conditional instructions. One consequence of full predication is that a conditional branch is simply a branch with a guarding predicate!

Bundle template	Slot 0	Slot 1	Slot 2	Execute cycle (1 bundle / cycle)
9: M M I	L.D F0,0(R1)	L.D F6,-8(R1)		1
14: M M F	L.D F10,-16(R1)	L.D F14,-24(R1)	ADD.D F4,F0,F2	3
15: M M F	L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F8,F6,F2	4
15: M M F	L.D F26,-48(R1)	S.D F4,0(R1)	ADD.D F12,F10,F2	6
15: M M F	S.D F8,-8(R1)	S.D F12,-16(R1)	ADD.D F16,F14,F2	9
15: M M F	S.D F16,-24(R1)		ADD.D F20,F18,F2	12
15: M M F	S.D F20,-32(R1)		ADD.D F24,F22,F2	15
15: M M F	S.D F24,-40(R1)		ADD.D F28,F26,F2	18
12: M M F	S.D F28,-48(R1)	DADDUI R1,R1,#-56	BNE R1,R2,Loop	21

a. The code scheduled to minimize the number of bundles.

Bundle template	Slot 0	Slot 1	Slot 2	Execute cycle (1 bundle / cycle)
8: M M I	L.D F0,0(R1)	L.D F6,-8(R1)		1
9: M M I	L.D F10,-16(R1)	L.D F14,-24(R1)		2
14: M M F	L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	3
14: M M F	L.D F26,-48(R1)		ADD.D F8,F6,F2	4
15: M M F			ADD.D F12,F10,F2	5
14: M M F		S.D F4,0(R1)	ADD.D F16,F14,F2	6
14: M M F		S.D F8,-8(R1)	ADD.D F20,F18,F2	7
15: M M F		S.D F12,-16(R1)	ADD.D F24,F22,F2	8
14: M M F		S.D F16,-24(R1)	ADD.D F28,F26,F2	9
9: M M I	S.D F20,-32(R1)	S.D F24,-40(R1)		10
8: M M I	S.D F28,-48(R1)	DADDUI R1,R1,#-56	BNE R1,R2,Loop	11

b. The code scheduled to minimize the number of cycles assuming one bundle executed per cycle.

FIGURE 4.13 The IA-64 instructions, including bundle bits and stops, for the unrolled version of x[i] = x[i] +s, when unrolled seven times and scheduled (a) to minimize the number of instruction bundles and (b) to minimize the number of cycles (assuming that a hazard stalls an entire bundle). Blank entries indicate unused slots, which are encoded as no-ops. The absence of stops indicates that some bundles could be executed in parallel. Minimizing the number of bundles yields 9 bundles versus the 11 needed to minimize the number of cycles. The scheduled version executes in just over half the number of cycles. Version (a) fills 85% of the instructions slots, while (b) fills 70%. The number of empty slots in the scheduled code and the use of bundles may lead to code sizes that are much larger than other RISC architectures.

Instr type	# formats	Representative instructions	Extra opcode bits	GPRs/ FPRs	Imm. bits	Other/Comment
A	8	add, subtract, and, or	9	3	0	
		shift left and add	7	3	0	2-bit shift count
		ALU immediates	9	2	8	
		Add immediate	3	2	14	
		Add immediate	0	2	22	
		Compare	4	2	0	2 predicate register destinations
		Compare immediate	3	1	8	2 predicate register destinations
I	29	Shift R/L variable	9	3	0	Many multimedia instructions use this format
		Test bit	6	3	6-bit field specifier	2 predicate register destinations
		Move to BR	6	1	9-bit branch predict	branch register specifier
M	46	Integer/FP load and store, Line prefetch	10	2	0	Speculative /non-speculative
		Integer/FP load/store, and line prefetch & post-increment by immediate	9	2	8	Speculative /non-speculative
		Integer/FP load prefetch & register postincrement	10	3		Speculative /non-speculative
		Integer/FP speculation check	3	1	21 in two fields	
B	9	PC-relative branch, counted branch	7	0	21	
		PC relative call	4	0	21	1 branch register
F	15	FP arithmetic	2	4		
		FP compare	2	2		2 6-bit predicate regs
L+X	4	Move immediate long	2	1	64	

FIGURE 4.14 A summary of some of the instruction formats of the IA-64 ISA is shown. The major opcode bits and the guarding predication register specifier add 10 bits to every instruction. The number of formats indicated for each instruction class in the second column (a total of 111) is a strict interpretation: where a different use of a field, even of the same size, is considered a different format. The number of formats that actually have *different field sizes* is one-third to one-half as large, Some instructions have unused bits that are reserved, we have not included those in this table. Immediate bits include the sign bit. The branch instructions include prediction bits, which are used when the predictor does not have a valid prediction. None of the many formats for the multimedia instructions are shown in this table.

Predicate registers are set using compare or test instructions. A compare instruction specifies one of ten different comparison tests and two predicate registers as destinations. The two predicate registers are written either with the result of the comparison (0 or 1) and the complement, or with some logical function that combines the two tests (such as `and`) and the complement. This capability allows multiple comparisons to be done in one instruction.

Speculation support in the IA-64 architecture consists of separate support for control speculation, which deals with deferring exception for speculated instructions, and memory reference speculation, which supports speculation of load instructions.

Deferred exception handling for speculative instructions is supported by providing the equivalent of poison bits. For the GPRs, these bits are called NaTs for Not a Thing, and this extra bit makes the GPRs effectively 65 bits wide. For the FP registers this capability is obtained using a special value, NaTVal, for Not A Thing Value; this value is encoded using a significand of 0 and an exponent outside of the IEEE range. Only speculative load instructions generate such values, but all instructions that do not affect memory will cause a NaT or NatVal to be propagated to the result register. (There are both speculative and nonspeculative loads; the latter can only raise immediate exceptions and cannot defer them.) Floating point exceptions are not handled through this mechanism, but use floating point status registers to record exceptions.

A deferred exception can be resolved in two different ways. First, if a nonspeculative instruction, such as a store, receives a NaT or NaTVal as a source operand, it generates an immediate and unrecoverable exception. Alternatively, a `chk.s` instruction can be used to detect the presence of NaT or NatVal and branch to a routine designed by the compiler to recover from the speculative operation. Such a recovery approach makes more sense for memory reference speculation.

The inability to store the contents of instructions with NaT or NatVal set would make it impossible for the OS to save the state of the processor. Thus, IA-64 includes special instructions to save and restore registers that do not cause an exception for a NaT or NaTVal and also save and restore the NaT bits.

Memory reference support in the IA-64 uses the a concept called advanced loads. An *advanced load* is a load that has been speculatively moved above store instructions on which it is potentially dependent. To speculatively perform a load, the `ld.a` (for advanced load) instruction is used. Executing this instruction creates an entry in a special table, called the *ALAT*. The ALAT stores both the register destination of the load and the address of the accessed memory location. When a store is executed, an associative look-up against the active ALAT entries is performed. If there is an ALAT entry with the same memory address as the store, the ALAT entry is marked as invalid.

Before any nonspeculative instruction (i.e., a store) uses the value generated by an advanced load or a value derived from the result of an advanced load, an explicit check is required. The check specifies the destination register of the advanced load. If the ALAT for that register is still valid, the speculation was legal

and the only effect of the check is to clear the ALAT entry. If the check fails, the action taken depends on which of two different types of checks was employed. The first type of check is an instruction `ld.c`, which simply causes the data to be reloaded from memory at that point. An `ld.c` instruction is used when *only* the load is advanced. The alternative form of a check, `chk.a`, specifies the address of a fix-up routine that is used to re-execute the load *and any other* speculated code that depended on the value of the load.

The Itanium Processor

The Itanium processor is the first implementation of the IA-64 architecture. It became available in the middle of 2001 with an 800 MHz clock. The processor core is capable of up to six issues per clock, with up to three branches and two memory references. The memory hierarchy consists of a three-level cache. The first level uses split instruction and data caches; floating point data is not placed in the first level cache. The second and third levels are unified caches, with the third level being an off-chip 4MB cache placed in the same container as the Itanium die.

Functional Units and Instruction Issue

There are nine functional units in the Itanium processor: 2 I-units, 2 M-units, 3 B-units, and 2 F-units. All the functional units are pipelined. Figure 4.15 gives the pipeline latencies for some typical instructions. In addition, when a result is bypassed from one unit to another, there is usually at least one additional cycle of delay.

Instruction	Latency
Integer load	1
Floating point load	9
Correctly predicted taken branch	0-3
Mispredicted branch	9
Integer ALU operations	0
FP arithmetic	4

FIGURE 4.15 The latency of some typical instructions on Itanium. The latency is defined as the smallest number of intervening instructions between two dependent instructions. Integer load latency assumes a hit in the first-level cache. FP loads always bypass the primary cache, so the latency is until a hit in the second-level cache. There are some minor restrictions for the some of the functional units, but these primarily involve the execution of infrequent instructions.

Itanium has an instruction issue window that contains up to two bundles at any given time. With this window size, Itanium can issue up to six instructions in a clock. In the worst case, if a bundle is split when it is issued, the hardware could see as few as four instructions: one from the first bundle to be executed and three from the second bundle. Instructions are allocated to functional units based on the bundle bits, ignoring the presence of no-ops or predicated instructions with untrue predicates. In addition, when issue to a functional unit is blocked, because the next instruction to be issued needs an already committed unit, the resulting bundle is split. A split bundle still occupies one of the two bundle slots, even if it has only one instruction remaining. In addition, there are several Itanium-dependent restrictions that cause a bundle to be split and issue a stop. For example, an MMF bundle, which contains two memory type instructions and a floating point type instruction, always generates a split before this bundle and after this bundle. This issue limitation means that a sequence of MMF bundles (like those in our earlier example shown in 4.13 on page 276) can execute at most three instructions per clock, even if no data dependences are present and no cache misses occur.

The Itanium processor uses a 10-stage pipeline divided into four major parts:

- Front-end (stages IPG, Fetch, and Rotate): prefetches up to 32 bytes per clock (2 bundles) into a prefetch buffer, which can hold up to eight bundles (24 instructions). Branch prediction is done using a multilevel adaptive predictor like that in the P6 microarchitecture we saw in Chapter 3.
- Instruction delivery (stages EXP and REN): distributes up to six instructions to the nine functional units. Implements registers renaming for both rotation and register stacking.
- Operand delivery (WLD and REG): accesses the register file, performs register bypassing, accesses and updates a register scoreboard, and checks predicate dependences. The scoreboard is used to detect when individual instructions can proceed, so that a stall of one instruction in a bundle need not cause the entire bundle to stall. (As we saw in Figure 4.13 on page 276, stalling the entire bundle leads to poor performance unless the instructions are carefully scheduled.)
- Execution (EXE, DET, and WRB): executes instructions through ALUs and load/store units, detects exceptions and posts NaTs, retires instructions and performs write-back.

Remarkably, the Itanium has many of the features more commonly associated with the dynamically-scheduled pipelines described in the last chapter: strong emphasis on branch prediction, register renaming, scoreboarding, a deep pipeline with many stages before execution (to handle instruction alignment, renaming, etc.), and several stages following execution to handle exception detection. It is somewhat surprising that an approach whose goal is to rely on compiler technol-

ogy and simpler hardware seems to be at least as complex as the dynamically scheduled processors we saw in the last chapter!

Itanium Performance

Figure 4.16 shows the performance of an 800 MHz Itanium versus a 1 GHz Alpha 21264 and a 2 GHz Pentium 4 for SPECint. The Itanium is only about 60% of the performance of the Pentium 4 and 68% of the performance of the Alpha 21264. What is perhaps even more surprising is that even if we normalize for clock rate, the Itanium is still only about 85% of the performance of the Alpha 21264, which is an older design in an older technology with about 20% less power consumption, despite its higher clock rate!

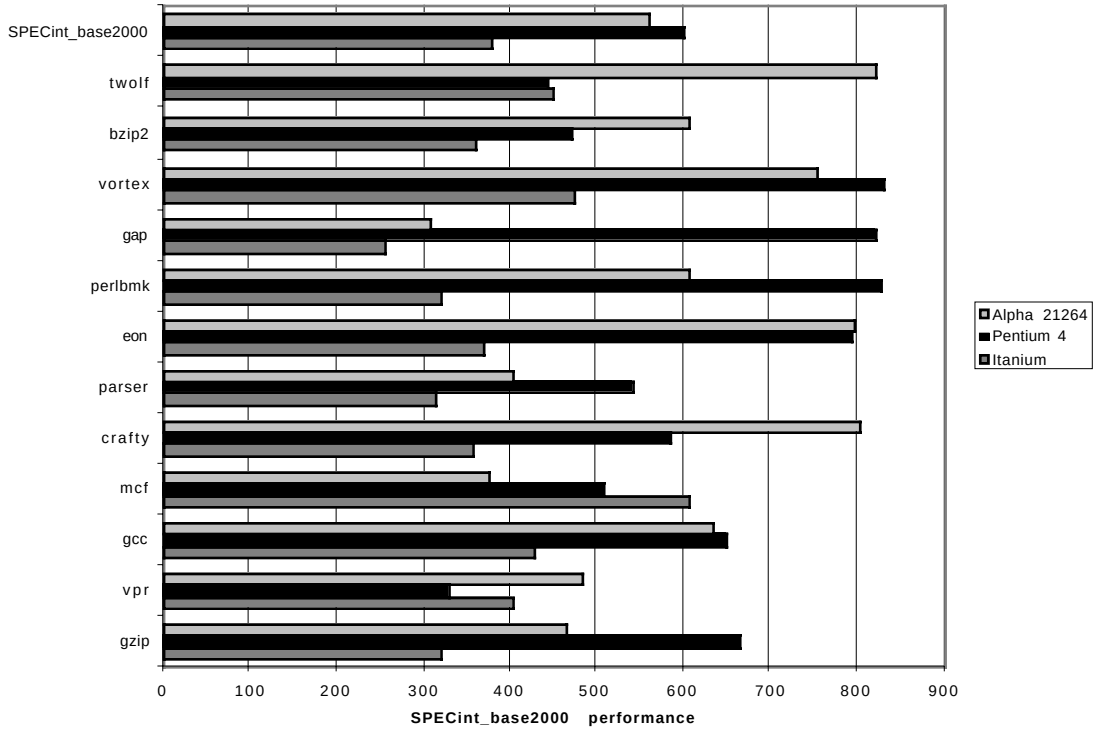


FIGURE 4.16 The SPECint benchmark set shows that the Itanium is considerably slower than either the Alp-ha 21264 or the Pentium 4. The Itanium system is a Hewlett Packard server rx4610 with an 800MHz Itanium and a 4 MB off-chip, level 3 cache. The Alpha system is a 1 GHz Compaq Alphaserver GS320 with only an on-chip L2 cache. The Pentium 4 system is a Compaq Precision 330 workstation with a 2 GHz part with a 256KB on-chip L2 cache. The overall SPECint_base 2000 number is computed as the geometric mean of the individual ratios.

The SPECfp benchmarks reveal a different story, as we can see in Figure 4.17. Overall, the Itanium processor is 1.08 times faster than the Pentium 4 and about 1.20 times faster than the Alpha 21264, at a clock rate that is only 40% to 80% as high. For floating point applications, the Itanium looks like a very competitive processor. As we saw in Chapter 3, floating point benchmarks exploit higher degrees of ILP well and also can make effective use of an aggressive memory system, including a large L3 cache. Both of these factors probably play a role.

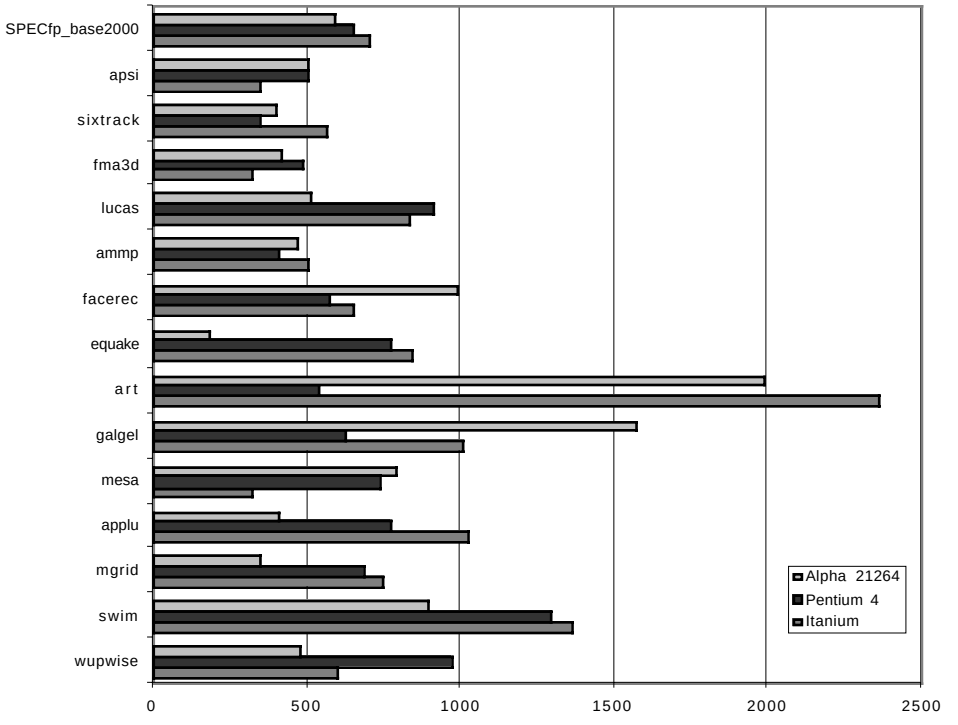


FIGURE 4.17 The SPECfp benchmark set shows that the Itanium is somewhat faster than either the Alpha 21264 or the Pentium 4. The Itanium system is an Hewlett Packard server rx4610 with one 800 MHz Itanium processor enabled and a 4MB off-chip, level 3 cache. The Alpha system is a Compaq Alphaserver GS320 with an 1 GHz Alpha 212164. The Pentium 4 system is a Compaq Precision 330 workstation with a 2 GHz part and like the Alpha system only the on-chip level L2 caches. The overall SPECfp_base 2000 number is computed as the geometric mean of the individual ratios.

There are two unusual aspects of the SPECfp performance measurements. First, the Itanium gains its performance advantage over the Pentium 4 primarily on the strength of its performance on one benchmark: “art”, where it is over four times faster than the Pentium 4. If the benchmark “art” was excluded, the Pentium 4 would outperform the Itanium for SPECfp. The other unusual aspect of

this performance data is that the Alpha processor shows a large gap of almost 30% between tuned and base performance for SPECfp. This compares to a gap between base and peak for the Itanium system of 0% and for the Pentium 4 system of 3%. Looking at the benchmark specific flags for the Alpha system, which primarily describe loop unrolling optimizations, it appears that this difference is due to compiler immaturity for the Alpha system. If the base performance could be brought to 95% of the peak performance, the Alpha system would have the highest SPECfp rating among these three processors.

As we mentioned in the last chapter, power may be most difficult hurdle in future processors and in achieving their performance goals.The limitations on power seem to be serious independent of how ILP is exploited, whether through pipelining and faster clock rates or through wider issue. The SPECFP data confirms this view. Although the Itanium achieves better floating point performance than either the Alpha 21264 or the Pentium 4, its floating point performance per watt is no better that of the Alpha 21264 and only 56% of that of the Pentium 4!

.

4.8 Another View: ILP in the Embedded and Mobile Markets

The Trimedia and Crusoe chips represent interesting approaches to applying the VLIW concepts in the embedded space.The Trimedia CPU is perhaps the closest current processor to a “classic” VLIW processor; it also supports a mechanism for compressing instructions while they are in main memory and the instruction cache and decompressing them during instruction fetch. This approach addresses the code size disadvantages of a VLIW processor, which would be especially troublesome in the embedded space. In contrast, the Crusoe processor uses software translation from the x86 architecture to a VLIW processor, achieving lower power consumption than typical x86 processors, which is key for Crusoe’s target market--mobile applications.

The Trimedia TM32 Architecture

The Trimedia TM32 CPU is a classic VLIW architecture: every instruction contains five operations and the processor is completely statically scheduled. In particular, the compiler is responsible for explicitly including no-ops both within an instruction-- when an operation field cannot be used--and between dependent instructions. The processor does not detect hazards, which if present will lead to incorrect execution. To reduce the cost of explicit no-ops in code size, the Trimedia processor compresses the code stream until the instructions are fetched from the instruction cache when they are expanded.

A Trimedia instruction consists of five operation slots, each able to specify an operation to one functional unit or an immediate field. Each individual operation in an instruction is predicated with a single register value, which if 0 causes *that* operation in the instruction to be canceled. The compiler must ensure that when multiple branches are included in an instruction, *at most* predicate is true. Loads can be freely speculated in the Trimedia architecture, since they do not generate exceptions. (There is no support for paged virtual memory.)

The mapping between instruction slots and units is limited, both for instruction encoding reasons and to simply instruction dispatch. As Figure 4.18 shows, there are 23 functional units of 11 different types. An instruction can specify any combination that will fit within the restrictions on the five fields.

Functional Unit	Unit Latency	Operation Slots					Typical operations performed by functional unit
		1	2	3	4	5	
ALU	0	yes	yes	yes	yes	yes	Integer add/subtract/compare, logicals
DMem	2				yes	yes	Loads and stores
DMemSpec	2					yes	Cache invalidate, prefetch, allocate
Shifter	0	yes	yes				Shifts and rotates
DSPALU	1	yes		yes			Simple DSP arithmetic operations
DSPMul	2		yes	yes			DSP operations with multiplication
Branch	3		yes	yes	yes		Branches and jumps
FALU	2	yes			yes		FP add, subtract
IFMul	2		yes	yes			Integer and FP multiply
FComp	0			yes			FP compare
FTough	16		yes				FP divide, square root

FIGURE 4.18 There are 23 functional units of 11 different types in the Trimedia CPU. This table shows the type of operations executed by each functional unit and the instruction slots available for specifying a particular functional unit. The number of instructions slots available for specifying a unit is equal to the number of copies of that unit. Hence, there are five ALU units and two FALU units.

To see how this VLIW processor operates, let's look at an example.

EXAMPLE First compile the loop for the following C code into MIPS instructions, and then show what it might look like if the Trimedia processor's operations fields were the same as MIPS instructions. (In fact, the Trimedia operation types are very close to MIPS instructions in capability.) Assume the func-

```
Loop:LD   R11,R0(R4)     # R11 = a[i]
   LD     R12,R0(R5))    # R12 = b[i]
   DADDU  R17,R11,R12    # R17 = a[i]+b[i]
   SD     0(R6),R17,     # c[i] = a[i]+b[i]
   DADDIU R4,R4,8        # R4 = next a[] address
   DADDIU R5,R5,8        # R5 = next b[] address
   DADDIU R6,R6,8        # R6 = next c[] address
   BNE    R4,R7,Loop     # if not last go to Loop
```

a. The MIPS code before unrolling.

```
Loop:LD   R11,0(R4))     load a[i]
   LD     R12,R0(R5))    # load b[i]
   DADDU  R17,R11,R12    # load b[i]
   SD     0(R6),R17,     # c[i] = a[i]+b[i]
   LD     R14,8(R4)      # load a[i]
   LD     R15,8(R5)      # load b[i]
   DADDU  R18,R14,R15    # a[i]+b[i]
   SD     8(R6),R18      # c[i] = a[i]+b[i]
   LD     R19,16(R4)     # load a[i]
   LD     R20,16(R5)     # load b[i]
   DADDU  R21,R19,R20    # a[i]+b[i]
   SD     16(R6),R21     # c[i] = a[i]+b[i]
   LD     R22,24(R4)     # load a[i]
   LD     R23,24(R5)     # load b[i]
   DADDU  R24,R22,R23    # a[i]+b[i]
   SD     24(R6),R24     # c[i] = a[i]+b[i]
   DADDIU R4,R4,32       # R4 = next a[] address
   DADDIU R5,R5,32       # R5 = next b[] address
   DADDIU R6,R6,32       # R6 = next c[] address
   BNE    R4,R7,Loop     # if not last go to Loop
```

b. The MIPS code after unrolling four times and optimizing the code. For simplicity, we have assumed that n is a multiple of four.

FIGURE 4.19 The MIPS code for the integer vector sum shown in part a before unrolling and in part b after unrolling four times. These code sequences assume that the starting addresses of `a`, `b`, and `c` are in registers `R4`, `R5`, and `R6`, and that `R7` contains the address of `a[n]`.

tional unit capacities and latencies shown in Figure 4.18.

```
void sum (int a[], int b[], int c[], int n)
{       int i;
        for (i=0; i<n; i++)
                c[i] = a[i]+b[i];
}
```

Unroll the loop so there are up to four copies of the body, if needed.

ANSWER Figure 4.19 shows the MIPS code before and after unrolling. Figure 4.20 shows the code for the Trimedia processor is shown in. (We assume that R30 contains the address of the first instruction in the sequence.) A standard MIPS processor needs 20 32-bit instructions for the unrolled loop and the Trimedia processor takes 8 instructions, meaning that 1/2 of the VLIW operation slots are full. The importance of compressing the code stream in the Trimedia CPU is clear from this example. As Figure 2.37 showed, even after compression, Trimedia code is two to three times larger than MIPS code. n

Slot 1	Slot 2	Slot 3	Slot 4	Slot 5
			LD R11,0(R4)	LD R12,R0(R5)
DADDUI R25,R6,32			LD R14,8(R4)	LD R15,8(R5)
SETEQ R25,R25,R7			LD R19,16(R4)	LD R20,16(R5)
DADDU R17,R11,R12	DADDIU R4,R4,32		LD R22,24(R4)	LD R23,24(R5)
DADDU R18,R14,R15	JMPF R25,R30		SD 0(R6),R17,	
DADDU R21,R19,R20	DADDIU R5,R5,32		SD 8(R6),R18	
DADDU R24,R22,R23			SD 16(R6),R21	
DADDIU R6,R6,32			SD 24(R6),R24	

FIGURE 4.20 The Trimedia code for a simple loop summing two vectors to generate a third makes good use of multiple memory ports but still contains a large fraction of idle slots. Loops with more computation within the body would make better use of the available operation slots. The DADDUI and SETEQ operations in the second and third instruction (first slot) serve to compute the loop termination test. The DADDUI duplicates a later add, so that the computation can be done early enough to schedule the branch and fill the 3 branch delay slots. The loop branch uses the JMPF instruction that tests a register (R25) and branches to an address specified by another register (R30);

Figure 4.21 shows the performance and code size of the TM1300, a 166 MHz implementation of the TM-32 architecture, and the NEC VR5000, a 250 MHz

version of the MIPS-32 architecture using the EEMBC consumer benchmarks or the measurements. The performance, which is plotted with columns on the left axis, and code size, which is plotted with lines on the right axis, are both shown relative to the NEC VR4122, a low-end embedded processor implementing the MIPS instruction set. Two different performance measurements are shown for the TM1300. The "out-of-box" measurement allows no changes to the source; the optimized version allows changes, including hand-coding. In the case of the TM1300 only source code medications and pragmas, which supply directions to the compiler, are used. The optimized TM1300 result is almost five times faster overall than the out-of-the-box result when the performance is summarized by the geometric mean. The out-of-the-box result for the TM1300 is 1.6 times faster than the VR5000.

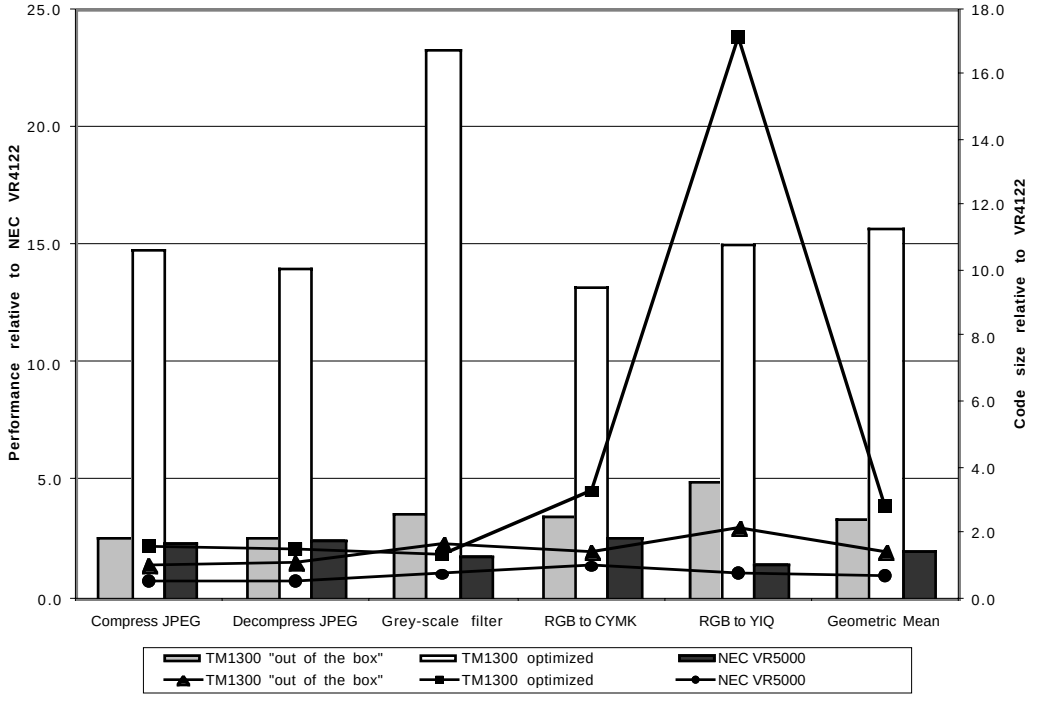


FIGURE 4.21 The performance and the code size for the EEMBC consumer benchmarks run on the Trimedia TM1300 and the NEC VR5000 and shown relative to the performance and code size for the low-end NEC VR4122. The columns and the left axis show the performance of the processors normalized to the out-of-the-box performance of the NEC VR4122. The TM1300 has a clock speed of 166 MHz and results are shown for a version with no source code changes (the "out-of-the-box" version) and with a set of changes at the source level (the "optimized" version) consisting of code changes and pragmas, which is then compiled. The lines and the right axis show the code size relative to the out-of-the-box NEC VR4122 using the Green Hills Compiler. thee measurements all come from the EEMBC website: http://www.eembc.org/benchmark/benchmain.asp.

One cost that is paid for this performance gain is a significant increase in code size. The code size of the out-of-the-box version of the benchmarks on the TM1300 is twice as large overall as the code size on the VR5000. For the TM1300 optimized version the code size is four times larger than the VR5000 version. Imagine how much larger the code size might be, if the code compression techniques were not used in the TM-32 architecture.

This tradeoff between code size and performance illustrates a fundamental difference in the design objectives of the MIPS and TM-32 architectures. The MIPS architecture is a general-purpose architecture with some extensions for the embedded market. The TM-32 architecture is an architecture designed for specific classes of embedded applications. The much larger code size of the TM-32 would simply make it unsuitable for many market segments and its specialized instruction set might not have significant performance benefits. On the other hand, its high performance for certain important functions, especially those in media processing, may allow it to be used in place of special-purpose chips designed for a single function, such as JPEG compression or image conversion. By replacing several special-purpose.dedicated chips with a single programmable processor, system cost might be reduced.

The Transmeta Crusoe Processor

The Crusoe processor is a VLIW processor designed for the low-power marketplace, especially mobile PCs and mobile Internet appliances, but, what makes it most unusual is that it achieves instruction set compatibility with the x86 instruction set through a software system that translates from the x86 instruction set to the VLIW instruction set implemented by Crusoe.

The Crusoe processor: Instruction Set Basics

The Crusoe processor is a reasonably straightforward VLIW with in-order execution. Instructions come in two sizes: 64 bits (containing two operations) and 128 bits (containing four operations).

There are five different types of operation slots:

1. ALU operations: typical RISC ALU operations with three integer register operands, each specifying one of 64 integer registers.
2. Compute: this slot may specify any integer ALU operation (there are two integer ALUs in the datapath), a floating point operation (using the 32 floating point registers), or a multimedia operation.
3. Memory: a load or store operation.
4. Branch: a branch instruction.
5. Immediate: a 32-bit immediate used by another operation in this instruction.

There are two different 128-bit instruction formats, characterized by what operation slots they have::

Memory	Compute	ALU	Immediate

Memory	Compute	ALU	Branch

The Crusoe processor uses a simple in-order, six-stage pipeline for integer instructions–two fetch stages, decode, register read, executive, and register write-back–and a ten-stage pipeline for floating point, which has four extra execute stages.

The Crusoe processor: software translation and hardware support

The software responsible for implementing the x86 instruction set uses a variety of techniques to establish a balance between execution speed and translation time. Initially, and for lowest overhead execution, the x86 code can be interpreted on an instruction by instruction basis. If a code segment is executed several times, it can be translated into an equivalent Crusoe code sequence, and the translation can be cached. The unit of translation is at least a basic block, since we know that if any instruction is executed in the block, they will all be executed. Translating an entire block both improves the translated code quality and reduces the translation overhead, since the translator need only be called once per basic block. Even a quick translation of a basic block can produce acceptable results, since simple code scheduling can be done during the translation.

One of the major challenges of a software-based implementation of an instruction set is maintaining the exception behavior of the original ISA while achieving good performance. In particular, achieving good performance often means reordering operations that correspond to instructions in the original program, which means that operations will be executed in a different order than in a strict sequential interpretation. This reordering is crucial to obtaining good performance when the target is a VLIW. Hence, just as other VLIW processors have found it useful to have support for speculative reordering, such support is important in Crusoe.

The Crusoe support for speculative reordering consists of four major parts: a shadowed register file, a program-controlled store buffer, memory alias detection hardware with speculative loads, and a conditional move instruction (called select) that is used to do if-conversion on x86 code sequences.

The shadowed register file and the program-controlled store buffer allow operations to be executed in a different order while ensuring that permanent state is not committed until no exceptions are possible. 48 of the integer registers and 16 of the floating point registers are shadowed. The shadow registers are used to hold the precise state and are updated only when a translated sequence that may

correspond to several x86 instructions has been executed without an exception. To indicate that the shadow registers should be updated a commit is executed, which has no overhead since every instruction has a bit used to indicate that a commit should be executed at the end of the instruction. If an exception occurs, the primary register set (called the working registers) can be restored from the shadow registers using a rollback operation. This mechanism allows out-of-order completion of register writes without sacrificing the precise exception model of the x86.

One of the most novel features of the Crusoe processor is the program-controlled write buffer. Stores generally cause irrevocable state update. Thus, in the dynamically-scheduled pipelines of Chapter 3, stores are committed in-order. Similarly, in the IA-64 architecture, stores are not speculated, since the state update cannot be undone. The Crusoe architecture provides a novel solution to this scheme: it includes the ability to control when the write buffer is allowed to update the memory. A gate instruction causes all stores to be held in the buffer, until a commit is executed. A rollback will cause the buffer to be flushed. This feature allows for speculative store execution without violating the exception model.

By using special speculative loads and stores (similar to the `ld.s` and `chk.s` mechanisms in IA-64) together with the rollback capability, the software translator can speculatively reorder loads and stores. The `ldp` instruction indicates a speculative load, whose effective address is stored in a special cache. A special store, `stam`, indicates a store that a load was moved across; if the `ldp` and `stam` touch the same address, then the speculative load was incorrect. A rollback is initiated and the code sequence is re-executed starting at the x86 instruction that followed the last gate.

This combination can also be used to do speculative data reuse in the case where a store intervenes between two loads that the compiler believes are to the same address. By making the first load a `ldp` and the store a `stam`, the translator can then reuse the value of the first load, knowing that if the store was to the same address as the load, it would cause a trap. The resulting trap can then re-execute the sequence using a more conservative interpretation.

The Crusoe processor: performance measures

Since the aim of the Crusoe processor is to achieve competitive performance at low power, benchmarks that measure both performance and power are critical. Because Crusoe depends on realistic behavior to tune the code translation process, it will not perform in a predictive manner when benchmarked using simple, but unrealistic scripts. Unfortunately, existing standard benchmarks use simple scripts that do not necessarily reflect actual user behavior (for benchmarks such as Microsoft Office) in terms of both repetition and timing. To remedy this factor, Transmeta has proposed a new set of benchmark scripts. Unfortunately, these scripts have not been released and endorsed by either a group of vendors or an independent entity.

Instead of including such results, Figure 4.22 summarizes the results of benchmarks whose behavior is well known (both are multimedia benchmarks). Since the execution time is constrained by real-time constraints the execution times are identical, and we compare only the power required.

Workload description	Power consumption for the workload (Watts)		Relative consumption TM 3200 / Mobile Pentium III
	Mobile Pentium III @ 500 MHz, 1.6V	TM 3200 @400MHz 1.5V	
MP3 playback	0.672	0.214	0.32
DVD playback	1.13	0.479	0.42

FIGURE 4.22 The energy performance of the processor and memory interface modules using two multimedia benchmarks is shown for the Mobile Pentium III and the Transmeta 3200. Both these chips are available in more recent versions that have additional power management features.

Although processor power differences can certainly affect battery life, with new processor designed to reduce energy consumption, the processor is often a minor contributor to overall energy usage. Figure shows power measurements for a typical laptop based on a Mobile Pentium III. As you can see small differences in processor power consumption are unlikely to make a large difference in overall power usage.

Major system	Component	Power (W)	Percent of total power
Processor	Low power Pentium III	0.8	8%
	Processor interface/ memory controller	0.65	6%
	Memory	0.1	1%
	Graphics	0.5	5%
I/O	Hard drive	0.65	6%
	DVD drive	2.51	24%
	Audio	0.5	5%
	Control and other	1.3	12%
	TFT display	2.8	27%
Power	Power supply	0.72	7%
Total		10.43	100%

FIGURE 4.23 Power distribution inside a laptop doing DVD payback shows that the processor subsystem consumes only 20% of the power! The I/O subsystem consumes an astonishing 74% of the power, with the display and DVD drive alone responsible for more than 50% of the total system power. The lesson for laptop users is clear: do not use your disk drive and keep your display off! This data was measured by Intel and is available on their web site.

4.9 Fallacies and Pitfalls

Fallacy: There is a simple approach to multiple issue processors that yields high performance without a significant investment in silicon area or design complexity.

This is a fallacy in the sense that many designers have believed it and committed significant effort to trying to find this "silver bullet" approach. Although it is possible to build relatively simple multiple-issue processors, as issue rates increase the gap between peak and sustained performance grows quickly. This gap has forced designers to explore sophisticated techniques for maintaining performance (dynamic scheduling, hardware and software support for speculation, more flexible issue, sophisticated instruction prefetch, and branch prediction). As Figure 4.24--which includes data on Itanium, Pentium III and 4, and Alpha 21264--shows, the result is uniformly high transistor counts, as well as high power consumption. See if you match the characteristics to the processor without reading the answer in the caption!

Issue rate: Total / Memory / Integer / FP / Branch	Max. clock rate available (in mid 2001)	Transistors with / without caches	On-chip caches: 1st level second level	Power (Watts)	SPECbase CPU2000 INT / FP
4/2/4/2/1	1 GHz	15 M / 6 M	64KB + 64KB	107	561 / 643
3/2/2/1/1	2 GHz	42 M / 23M	12K entries + 8KB 256 KB	67	636 / 648
3/2/2/1/1	1 GHz	28 M / 9.5 M	16KB + 16KB 256 KB	36	454 / 329
6/2/2/2/3	0.8 GHz	25 M / 17 M	16K + 16K 96 KB	130	379 / 714

FIGURE 4.24 The key characteristics of four recent multiple issue microprocessors show significant dramatic variety. These vary from a dynamically-scheduled speculative processor to a statically schedule multiple issue processor to a VLIW. They range in die size from just over 100 mm2 to almost 300 mm2 and in power from 26 W to just under 100W, although the integrated circuit processes differ significantly. The SPEC numbers are the highest official numbers reported as of August 2001, and the clock rate of that system is shown. Can you guess what these four processors are?

Answer: Alpha 21264,Intel Pentium 4, Intel Pentium III, Intel Itanium.

In addition to the hardware complexity, it has become clear that compiling for processors with significant amounts of ILP has become extremely complex. Not only must the compiler support a wide set of sophisticated transformation, but tuning the compiler to achieve good performance across a wide set of benchmarks appears to be very difficult.

Obtaining good performance is also affected by design decisions at the system level, and such choices can be complex. For example, for the first machine in Figure 4.24 the highest SPECInt number comes from a 1 GHz part, but the highest SPECFP number comes from a system with a 833 MHz part!

.

4.10 Concluding Remarks

The EPIC approach is based on the application of massive resources. These resources include more load-store, computational, and branch units, as well as larger, lower-latency caches than would be required for a superscalar processor. Thus, IA-64 gambles that, in the future, power will not be the critical limitation, and that massive resources, along with the machinery to exploit them, will not penalize performance with their adverse effect on clock speed, path length, or CPI factors.

M. Hopkins [2000], in a commentary on the EPIC approach and the IA-64 architecture

.

The relative merits of software-intensive and hardware-intensive approaches to exploiting ILP continue to be debated. Over time, it appears that advantageous elements from the "enemy camp" are slowly being incorporated into each approach. Examples include:

"Software" techniques in hardware-centric approaches	"Hardware" techniques in software-intensive approaches
Support for conditional instructions.	Scoreboard scheduling of instructions.
Prefetch instructions and other cache "hints".	Dynamic branch prediction.
Branch prediction hints.	Rollback or trap-and-fixup support for speculation.
Special support for speculative (non-excepting) loads.	Hardware for checking speculated load correctness.

Initially, the software-intensive and hardware-intensive approaches were quite different, and the ability to manage the complexity of the hardware-intensive approaches was in doubt. The development of several high performance dynamic speculation processors, which have high clock rates, has eased this concern. The complexity of the IA-64 architecture and the Itanium design has indicated to many designers that it is unlikely that a software-intensive approach will produce processors that are much faster, much smaller (in transistor count or die size), much simpler, or much more power efficient. Similarly, the development of compilers for these processors has proved challenging. Although it is likely that both future compilers for IA-64 and future implementations will be more effective, the IA-64 architecture does *not* appear to represent a significant breakthrough in

scaling ILP or in avoiding the problems of complexity and power consumption in high performance processors.

As both approaches have proven to have advantages, each has tended to incorporate techniques from the other. It remains unclear whether the two approaches will continue to move toward the middle, or whether a new architectural approach that truly combines the best of each will be developed.

The alternative to trying to continue to push uniprocessors to exploit ILP is to look toward multiprocessors, the topic of Chapter 6. Looking toward multiprocessors to take advantage of parallelism overcomes a fundamental problem in ILP processors: building a cost-effective memory system. A multiprocessor memory system is inherently multiported and, as we will see, can even be distributed in a larger processor.

Using multiprocessors to exploit parallelism encounters two difficulties. First, it is likely that the software model will need to change. Second, multiprocessors may have difficulty in exploiting fine-grained, low-level parallelism. Although it appears clear that using a large number of processors requires new programming approaches, using a smaller number of processors efficiently could be based on compiler or language approaches, or might even be used for multiple independent processes. Exploiting the type of fine-grained parallelism that a compiler can easily uncover can be quite difficult in a multiprocessor, since the processors are relatively far apart. Simultaneous multithreading (see Chapter 6) may be the intermediate step between ILP and true multiprocessing.

In 2000, IBM announced the first commercial single-chip general-purpose multiprocessor, the IBM Power4, which contains two Power3 processors and an integrated second level cache, for a total transistor count of 174 million! Because the Power4 chip also contains a memory interface, a third-level cache interface, and a direct multiprocessor interconnect, IBM used the Power4 to build an 8-processor module using four Power4 chips. The module has a total size of about 64 in^2 and is capable of a peak performance of 32 billion floating point operations per second! The challenge for multiprocessors appears to be the same as for ILP-intensive uniprocessors: translate this enormous peak performance into delivered performance on real applications. In the case of the IBM design, the intended market is large-scale servers, where the available application parallelism may make a multiprocessor attractive.

The embedded world actually delivered multiple processors on a die first! The TI TMS320C80 provides four DSPs and a RISC processor, which acts as controller, on a single die. Likewise, several embedded versions of MIPS processors use multiple processors per die. The obvious parallelism in embedded applications and the lack of stringent software compatibility requirements may allow the embedded world to embrace on-chip multiprocessing faster than the desktop environment. We will return to this discussion in Chapter 6.

4.11 Historical Perspective and References

This section describes the historical development of multiple issue approaches, which began with static multiple issue and proceeds to the most recent work leading to IA-64. Similarly, we look at. the long history of compiler technology in this area.

The Development of Multiple-Issue Processors

Most of the early multiple-issue processors followed a LIW or VLIW design approach. Charlesworth [1981] reports on the Floating Point Systems AP-120B, one of the first wide-instruction processors containing multiple operations per instruction. Floating Point Systems applied the concept of software pipelining in both a compiler and by hand-writing assembly language libraries to use the processor efficiently. Since the processor was an attached processor, many of the difficulties of implementing multiple issue in general-purpose processors, for example, virtual memory and exception handling, could be ignored.

The Stanford MIPS processor had the ability to place two operations in a single instruction, though this capability was dropped in commercial variants of the architecture, primarily for performance reasons. Along with his colleagues at Yale, Fisher [1983] proposed creating a processor with a very wide instruction (512 bits), and named this type of processor a VLIW. Code was generated for the processor using trace scheduling, which Fisher [1981] had developed originally for generating horizontal microcode. The implementation of trace scheduling for the Yale processor is described by Fisher et al. [1984] and by Ellis [1986]. The Multiflow processor (see Colwell et al. [1987]) was based on the concepts developed at Yale, although many important refinements were made to increase the practicality of the approach. Among these was a controllable store buffer that provided support for a form of speculation. Although more than 100 Multiflow processors were sold, a variety of problems, including the difficulties of introducing a new instruction set from a small company and the competition provided from commercial RISC microprocessors that changed the economics in the minicomputer market, led to failure of Multiflow as a company.

Around the same time as Multiflow, Cydrome was founded to build a VLIW-style processor (see Rau et al. [1989]), which was also unsuccessful commercially. Dehnert, Hsu, and Bratt [1989] explain the architecture and performance of the Cydrome Cydra 5, a processor with a wide-instruction word that provides dynamic register renaming and additional support for software pipelining. The Cydra 5 is a unique blend of hardware and software, including conditional instructions and register rotation, aimed at extracting ILP. Cydrome relied on more hardware than the Multiflow processor and achieved competitive performance primarily on vector-style codes. In the end, Cydrome suffered from problems

similar to those of Multiflow and was not a commercial success. Both Multiflow and Cydrome, though unsuccessful as commercial entities, produced a number of people with extensive experience in exploiting ILP as well as advanced compiler technology; many of those people have gone on to incorporate their experience and the pieces of the technology in newer processors. Fisher and Rau [1993] edited a comprehensive collection of papers covering the hardware and software of these two important processors.

Rau had also developed a scheduling technique called *polycyclic scheduling*, which is a basis for most software pipelining schemes (see Rau, Glaeser, and Picard [1982]). Rau's work built on earlier work by Davidson and his colleagues on the design of optimal hardware schedulers for pipelined processors. Other historical LIW processors have included the Apollo DN 10000 and the Intel i860, both of which could dual issue FP and integer operations.

One of the interesting approaches used in early VLIW processors, such as the AP-120B and i860, was the idea of a pipeline organization that requires operations to be "pushed through" a functional unit and the results to be caught at the end of the pipeline. In such processors, operations advance only when another operation pushes them from behind (in sequence). Furthermore, an instruction specifies the destination for an instruction issued earlier that will be pushed out of the pipeline when this new operation is pushed in. Such an approach has the advantage that it does not specify a result destination when an operation first issues but only when the result register is actually written. This separation eliminates the need to detect WAW and WAR hazards in the hardware. The disadvantage is that it increases code size since no-ops may be needed to push results out when there is a dependence on an operation that is still in the pipeline and no other operations of that type are immediately needed. Instead of the "push-and-catch" approach used in these two processors, almost all designers have chosen to use *self-draining pipelines* that specify the destination in the issuing instruction and in which an issued instruction will complete without further action. The advantages in code density and simplifications in code generation seem to outweigh the advantages of the more unusual structure.

Compiler Technology and Hardware-Support for Scheduling

Loop-level parallelism and dependence analysis was developed primarily by D. Kuck and his colleagues at the University of Illinois in the 1970s. They also coined the commonly used terminology of *antidependence* and *output dependence* and developed several standard dependence tests, including the GCD and Banerjee tests. The latter test was named after Uptal Banerjee and comes in a variety of flavors. Recent work on dependence analysis has focused on using a variety of exact tests ending with an algorithm called Fourier-Motzkin, which is a linear programming algorithm. D. Maydan and W. Pugh both showed that the sequences of exact tests were a practical solution.

In the area of uncovering and scheduling ILP, much of the early work was connected to the development of VLIW processors, described earlier. Lam [1988] developed algorithms for software pipelining and evaluated their use on Warp, a wide-instruction-word processor designed for special-purpose applications. Weiss and J. E. Smith [1987] compare software pipelining versus loop unrolling as techniques for scheduling code on a pipelined processor. Rau [1994] developed modulo scheduling to deal with the issues of software pipelining loops and simultaneously handling register allocation.

Support for speculative code scheduling was explored in a variety of contexts, including several processors that provided a mode in which exceptions were ignored, allowing more aggressive scheduling of loads (e.g., the MIPS TFP processor, see [Hsu 1994].). Several groups explored ideas for more aggressive hardware support for speculative code scheduling. For example, Smith, Horowitz, and Lam [1992] created a concept called boosting that contains a hardware facility for supporting speculation but provides a checking and recovery mechanism, similar to those in IA-64 and Crusoe. The sentinel scheduling idea, which is also similar to the speculate-and-check approach used in both Crusoe and the IA-64 architectures, was developed jointly by researchers at U. of Illinois and HP Laboratories (see [Mahlke et al. 1992]).

In the early 1990s, Wen-Mei Hwu and his colleagues at the University of Illinois developed a compiler framework, called IMPACT (see [Chang, et. al. 1991]), for exploring the interaction between multiple-issue architectures and compiler technology. This project led to several important ideas, including: superblock scheduling (see [Hwu et. al. 1993]), extensive use of profiling for guiding a variety of optimizations (e.g., procedure inlining), and the use of a special buffer (similar to the ALAT or program-controlled store buffer) for compile-aided memory conflict detection (see [Gallagher, et. al. 1994]). They also explored the performance trade-offs between partial and full support for predication in [Mahlke, et. al. 1995].

The early RISC processors all had delayed branches, a scheme inspired from microprogramming, and several studies on compile-time branch prediction were inspired by delayed branch mechanisms. McFarling and Hennessy [1986] did a quantitative comparison of a variety of compile-time and runtime branch prediction schemes. Fisher and Freudenberger [1992] evaluated a range of compile-time branch prediction schemes using the metric of distance between mispredictions.

EPIC and the IA-64 Development

The roots of the EPIC approach lie in earlier attempts to build LIW and VLIW machines–especially those at Cydrome and Multiflow–and in a long history of compiler work that continued after these companies failed at HP, the University of Illinois, and elsewhere. Insights gained from that work led designers at HP to propose a VLIW-style, 64-bit architecture to follow on to the HP-PA RISC archi-

tecture. Intel was looking for a new architecture to replace the x86 (now called IA-32) architecture and to provide 64-bit capability. In 1995, they formed a partnership to design a new architecture, IA-64, and build processors based on it. Itanium is the first such processor. A description of the IA-64 architecture is available online at: http://devresource.hp.com/devresource/Docs/Refs/IA64ISA/. A description of the highlights of the Itanium processor is available at: http://www.intel.com/design/itanium/microarch_ovw/index.htm.

References

BALL, T. AND J.R. LARUS [1993]. "Branch prediction for free," *Proc. SIGPLAN 1993 Conf. on Programming Language Design and Implementation*, June, 300-313.

CHANG, P. P., MAHLKE, S. A., CHEN, W. Y., WARTER, N. J., AND W. W. HWU [1991], "IMPACT: An architectural framework for multiple-instruction-issue processors," Proceedings of the 18th International Symposium on Computer Architecture (May), pp. 266--275.

CHARLESWORTH, A. E. [1981]. "An approach to scientific array processing: The architecture design of the AP-120B/FPS-164 family," *Computer* 14:9 (September), 18–27.

COLWELL, R. P., R. P. NIX, J. J. O'DONNELL, D. B. PAPWORTH, AND P. K. RODMAN [1987]. "A VLIW architecture for a trace scheduling compiler," *Proc. Second Conf. on Architectural Support for Programming Languages and Operating Systems*, IEEE/ACM (March), Palo Alto, Calif., 180–192.

DEHNERT, J. C., P. Y.-T. HSU, AND J. P. BRATT [1989]. "Overlapped loop support on the Cydra 5," *Proc. Third Conf. on Architectural Support for Programming Languages and Operating Systems* (April), IEEE/ACM, Boston, 26–39.

ELLIS, J. R. [1986]. *Bulldog: A Compiler for VLIW Architectures*, MIT Press, Cambridge, Mass.

FISHER, J. A. [1981]. "Trace scheduling: A technique for global microcode compaction," *IEEE Trans. on Computers* 30:7 (July), 478–490.

FISHER, J. A. [1983]. "Very long instruction word architectures and ELI-512," *Proc. Tenth Symposium on Computer Architecture* (June), Stockholm, 140–150.

FISHER, J. A., J. R. ELLIS, J. C. RUTTENBERG, AND A. NICOLAU [1984]. "Parallel processing: A smart compiler and a dumb processor," *Proc. SIGPLAN Conf. on Compiler Construction* (June), Palo Alto, Calif., 11–16.

FISHER, J. A. AND S. M. FREUDENBERGER [1992]. "Predicting conditional branches from previous runs of a program," *Proc. Fifth Conf. on Architectural Support for Programming Languages and Operating Systems*, IEEE/ACM (October), Boston, 85-95.

FISHER, J. A. AND B. R. RAU [1993]. *Journal of Supercomputing (*January*)*, Kluwer.

FOSTER, C. C. AND E. M. RISEMAN [1972]. "Percolation of code to enhance parallel dispatching and execution," *IEEE Trans. on Computers* C-21:12 (December), 1411–1415.

GALLAGHER, D.M., CHEN, W.Y., MAHLKE, S.A., GYLLENHAAL, J.C., AND W.W. HWU [1994]. "Dynamic memory disambiguation using the memory conflict buffer." Proc. Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (October), Santa Clara.

HOPKINS, M. [2000]. "A Critical Look at IA-64: Massive Resources, Massive ILP, But Can It Deliver?" Microprocessor Report, Feb.

P.HSU [1994]. "Designing the TFP Microprocessor", IEEE Micro, Vol.18 Nr.2 (April), pp 2333.

HWU, W. W., MAHLKE, S. A., CHEN, W. Y., CHANG, P. P., WARTER, N. J., BRINGMANN, R. A.,

OUELLETTE, R. O., HANK, R. E., KIYOHARA, T., HAAB, G. E., HOLM, J. G., AND D. M. LAVERY [1993]. "The superblock: An effective technique for VLIW and superscalar compilation." Journal of Supercomputing, 7(1,2) (March),:229--248.

LAM, M. [1988]. "Software pipelining: An effective scheduling technique for VLIW processors," SIGPLAN Conf. on Programming Language Design and Implementation, ACM (June), Atlanta, Ga., 318–328.

MAHLKE, S. A., W. Y. CHEN, W.-M. HWU, B. R. RAU, AND M. S. SCHLANSKER [1992]. "Sentinel scheduling for VLIW and superscalar processors," *Proc. Fifth Conf. on Architectural Support for Programming Languages and Operating Systems* (October), Boston, IEEE/ACM, 238–247.

MAHLKE, S.A., HANK, R.E. MCCORMICK, J.E. AUGUST, D.I. AND HWU.W.W. [1995]. "A Comparison of Full and Partial Predicated Execution Support for ILP Processors." Proceedings of the 22nd Annual International Symposium on Computer Architecture (June), pages 138--149, Santa Margherita Ligure, Italy.,

MCFARLING, S. AND J. HENNESSY [1986]. "Reducing the cost of branches," *Proc. 13th Symposium on Computer Architecture* (June), Tokyo, 396–403.

NICOLAU, A. AND J. A. FISHER [1984]. "Measuring the parallelism available for very long instruction word architectures," *IEEE Trans. on Computers* C-33:11 (November), 968–976.

B. R. RAU [1994]. "Iterative Modulo Scheduling: An Algorithm For Software Pipelining Loops." Proc. 27th Annual International Symposium on Microarchitecture (November), pages 63--74,San Jose, CA.

RAU, B. R., C. D. GLAESER, AND R. L. PICARD [1982]. "Efficient code generation for horizontal architectures: Compiler techniques and architectural support," *Proc. Ninth Symposium on Computer Architecture* (April), 131–139.

RAU, B. R., D. W. L. YEN, W. YEN, AND R. A. TOWLE [1989]. "The Cydra 5 departmental supercomputer: Design philosophies, decisions, and trade-offs," *IEEE Computers* 22:1 (January), 12–34.

RISEMAN, E. M. AND C. C. FOSTER [1972]. "Percolation of code to enhance parallel dispatching and execution," *IEEE Trans. on Computers* C-21:12 (December), 1411–1415.

THORLIN, J. F. [1967]. "Code generation for PIE (parallel instruction execution) computers," *Proc. Spring Joint Computer Conf.* 27.

WILSON, R.P. AND MONICA S. LAM [1995]. "Efficient Context-Sensitive Pointer Analysis for C Programs," Proc. ACM SIGPLAN'95 Conference on Programming Language Design and Implementation, La Jolla, CA, June , pp. 1-12.

EXERCISES

Most of these exercises are still good. What we need are exercises that explore the concepts of predication and speculative scheduling.

New addition: consider the simple loop in Section 4.1. Assume the number of iterations is unknown, but large. Find the theoretically optimal number of unrollings using the timing of the loop in Section 4.1 Hint: recall that you will need two loops: one unrolled and one not!

4.1 [15] <4.1> List all the dependences (output, anti, and true) in the following code fragment. Indicate whether the true dependences are loop-carried or not. Show why the loop is not parallel.

```
for (i=2;i<100;i=i+1) {
```

```
        a[i] = b[i] + a[i];   /* S1 */
        c[i-1] = a[i] + d[i]; /* S2 */
        a[i-1] = 2 * b[i];    /* S3 */
        b[i+1] = 2 * b[i];    /* S4 */
}
```

4.2 [15] <4.1> Here is an unusual loop. First, list the dependences and then rewrite the loop so that it is parallel.

```
for (i=1;i<100;i=i+1) {
        a[i] = b[i] + c[i];    /* S1 */
        b[i] = a[i] + d[i];    /* S2 */
        a[i+1] = a[i] + e[i]; /* S3 */
}
```

4.3 [15] <4.1> Assuming the pipeline latencies from Figure 4.1, unroll the following loop as many times as necessary to schedule it without any delays, collapsing the loop overhead instructions. Assume a one-cycle delayed branch. Show the schedule. The loop computes Y[i] = a × X[i] + Y[i], the key step in a Gaussian elimination.

```
loop:   L.D     F0,0(R1)
        MUL.D   F0,F0,F2
        L.D     F4,0(R2)
        ADD.D   F0,F0,F4
        S.D     F0,0(R2)
        DADDUI  R1,R1,#-8
        DADDUI  R1,R1,#-8
        BNE     R1,R3,loop
```

4.4 [15] <4.1> Assume the pipeline latencies from Figure 4.1 and a one-cycle delayed branch. Unroll the following loop a sufficient number of times to schedule it without any delays. Show the schedule after eliminating any redundant overhead instructions. The loop is a dot product (assuming F2 is initially 0) and contains a recurrence. Despite the fact that the loop is not parallel, it can be scheduled with no delays.

```
loop:   L.D     F0,0(R1)
        L.D     F4,0(R2)
        MUL.D   F0,F0,F4
        ADD.D   F2,F0,F2
        DADDUI  R1,R1,#-8
        DADDUI  R1,R1,#-8
        BNE     R1,R3,loop
```

4.5 [20/22/22/22/22/25/25/25/20/22/22] <4.1,4.2,4.3> In this Exercise, we will look at how a common vector loop runs on a variety of pipelined versions of MIPS. The loop is the so-called SAXPY loop (discussed extensively in Appendix B) and the central operation in Gaussian elimination. The loop implements the vector operation Y = a × X + Y for a vector of length 100. Here is the MIPS code for the loop:

```
foo:    L.D     F2,0(R1)        ;load X(i)
        MUL.D   F4,F2,F0        ;multiply a*X(i)
        L.D     F6,0(R2)        ;load Y(i)
        ADD.D   F6,F4,F6        ;add a*X(i) + Y(i)
        S.D     F6,0(R2)        ;store Y(i)
        ADDI    R1,R1,#8        ;increment X index
        ADDI    R2,R2,#8        ;increment Y index
        SGTI    R3,R1,done      ;test if done
        BEQZ    R3,foo          ; loop if not done
```

For (a)–(e), assume that the integer operations issue and complete in one clock cycle (including loads) and that their results are fully bypassed. Ignore the branch delay. You will use the FP latencies shown in Figure 4.1 on page 222. Assume that the FP unit is fully pipelined.

a. [20] <4.1> For this problem use the standard single-issue MIPS pipeline with the pipeline latencies from Figure 4.1. Show the number of stall cycles for each instruction and what clock cycle each instruction begins execution (i.e., enters its first EX cycle) on the first iteration of the loop. How many clock cycles does each loop iteration take?

b. [22] <4.1> Unroll the MIPS code for SAXPY to make four copies of the body and schedule it for the standard MIPS integer pipeline and a fully pipelined FPU with the FP latencies of Figure 4.1 on page 222. When unwinding, you should optimize the code as we did in section 4.1. Significant reordering of the code will be needed to maximize performance.

c. [25] <4.1,4.3> Assume a superscalar architecture that can issue any two independent operations in a clock cycle (including two integer operations). Unwind the MIPS code for SAXPY to make four copies of the body and schedule it assuming the FP latencies of Figure 4.1 on page 222. Assume one fully pipelined copy of each functional unit (e.g., FP adder, FP multiplier) and two integer functional units with latency to use of 0. How many clock cycles will each iteration on the original code take? When unwinding, you should optimize the code as in section 4.1. What is the speedup versus the original code?

d. [25] <4.3> To further boost clock rates, a number of processors have added additional pipelining to units traditionally assigned only one stage in the pipeline, resulting in pipelines with depths of 10-14 cycles for the integer pipeline. Suppose we designed a deeply pipelined MIPS processor that had twice the clock rate of our standard MIPS pipeline and could issue any two unrelated instructions in the same time that the normal MIPS pipeline issued one operation. If the second instruction is dependent on the first, only the first will issue. Unroll the MIPS SAXPY code to make four copies of the loop body and schedule it for this deeply pipelined processor, assuming the FP latencies of Figure 4.1 on page 222. Also assume the load to use latency is 1 cycle, but other integer unit latencies are 0 cycles. How many clock cycles does each loop iteration take? Remember that these clock cycles are half as long as those on a standard MIPS pipeline or a superscalar MIPS.

e. [20] <4.3> Start with the SAXPY code and the processor used in Figure 4.5. Unroll the SAXPY loop to make four copies of the body, performing simple optimizations (as in section 4.1). Assume all integer unit latencies are 0 cycles and the FP latencies are given in Figure 4.1. Fill in a table like Figure 4.28 for the unrolled loop. How many clock cycles does each loop iteration take?

4.6 [15] <4.4> Here is a simple code fragment:

```
for (i=2;i<=100;i+=2)
        a[i] = a[50*i+1];
```

To use the GCD test, this loop must first be “normalized”—written so that the index starts at 1 and increments by 1 on every iteration. Write a normalized version of the loop (change the indices as needed), then use the GCD test to see if there is a dependence.

4.7 [15] <4.1,4.4> Here is another loop:

```
for (i=2,i<=100;i+=2)
        a[i] = a[i-1];
```

Normalize the loop and use the GCD test to detect a dependence. Is there a loop-carried, true dependence in this loop?

4.8 [25] <4.4> Show that if for two array elements A(a × i + b) and A(c × i + d) there is a true dependence, then GCD(c,a) divides (d – b).

4.9 [15] <4.4> Rewrite the software pipelining loop shown in the Example on page 249, so that it can be run by simply decrementing R1 by 16 before the loop starts. After rewriting the loop, show the start-up and finish-up code. *Hint*: To get the loop to run properly when R1 is decremented, the `S.D` should store the result of the *original* first iteration. You can achieve this by adjusting load-store offsets.

4.10 [20] <4.4> Consider the loop that we software pipelined on page 249. Suppose the latency of the `ADD.D` was five cycles. The software pipelined loop now has a stall. Show how this loop can be written using both software pipelining and loop unrolling to eliminate any stalls. The loop should be unrolled as few times as possible (once is enough). You need not show loop start-up or clean-up.

4.11 [15] <4.5> Perform the same transformation (moving up the branch) as the example on page 262, but using only conditional move. Be careful that your loads, which are no longer control dependent, cannot raise an exception if they should not have been executed!

4.12 [Discussion] <4.3-4.5> Discuss the advantages and disadvantages of a superscalar implementation and a VLIW approach in the context of MIPS. What levels of ILP favor each approach? What other concerns would you consider in choosing which type of processor to build? How does speculation affect the results?

5 Memory-Hierarchy Design

Ideally one would desire an indefinitely large memory capacity such that any particular . . . word would be immediately available. . . . We are . . . forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible.

A. W. Burks, H. H. Goldstine, and J. von Neumann
Preliminary Discussion of the Logical Design of an Electronic Computing Instrument (1946)

5.1	Introduction	373
5.2	Review of the ABCs of Caches	376
5.3	Cache Performance	390
5.4	Reducing Cache Miss Penalty	398
5.5	Reducing Miss Rate	408
5.6	Reducing Cache Miss Penalty or Miss Rate via Parallelism	421
5.7	Reducing Hit Time	430
5.8	Main Memory and Organizations for Improving Performance	435
5.9	Memory Technology	442
5.10	Virtual Memory	448
5.11	Protection and Examples of Virtual Memory	457
5.12	Crosscutting Issues in the Design of Memory Hierarchies	467
5.13	Putting It All Together: Alpha 21264 Memory Hierarchy	471
5.14	Another View: The Emotion Engine of the Sony Playstation 2	479
5.15	Another View: The Sun Fire 6800 Server	483
5.16	Fallacies and Pitfalls	488
5.17	Concluding Remarks	495
5.18	Historical Perspective and References	498
	Exercises	504

5.1 Introduction

Computer pioneers correctly predicted that programmers would want unlimited amounts of fast memory. An economical solution to that desire is a *memory hierarchy*, which takes advantage of locality and cost/performance of memory technologies. The *principle of locality*, presented in the first chapter, says that most programs do not access all code or data uniformly (see section 1.6, page 38). This principle, plus the guideline that smaller hardware is faster, led to hierarchies based on memories of different speeds and sizes. Figure 5.1 shows a multilevel memory hierarchy, including typical sizes and speeds of access.

Since fast memory is expensive, a memory hierarchy is organized into several levels—each smaller, faster, and more expensive per byte than the next lower level. The goal is to provide a memory system with cost almost as low as the cheapest level of memory and speed almost as fast as the fastest level. The levels of the hierarchy usually subset one another. All data in one level is also found in the level below, and all data in that lower level is found in the one below it, and so on until we reach the bottom of the hierarchy.

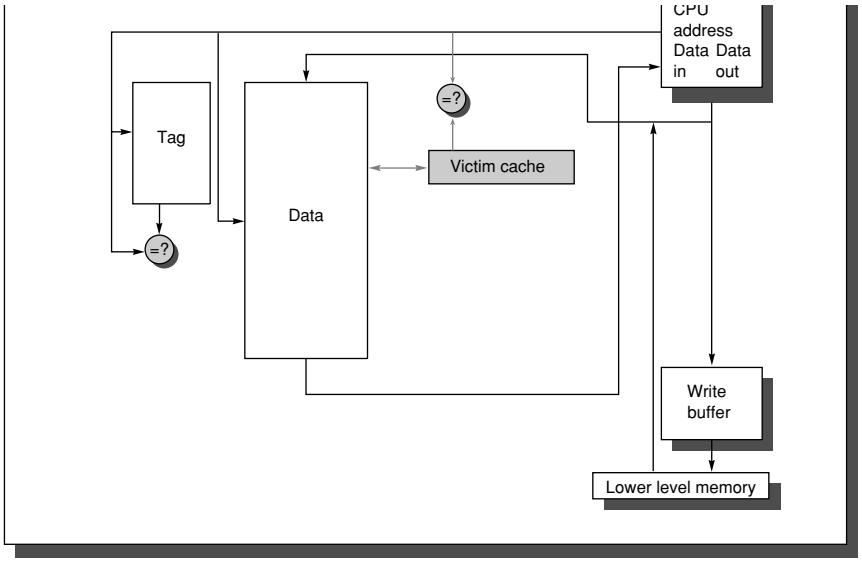


FIGURE 5.1 These are the levels in a typical memory hierarchy in embedded, desktop, and server computers. As we move farther away from the CPU, the memory in the level below becomes slower and larger. Note that the time units change by factors of ten–from picoseconds to milliseconds–and that the size units change by factors of a thousand–from bytes to terabytes. Figure 5.3 shows more parameters for desktops and small servers.

Note that each level maps addresses from a slower, larger memory to a smaller but faster memory higher in the hierarchy. As part of address mapping, the memory hierarchy is given the responsibility of address checking; hence protection schemes for scrutinizing addresses are also part of the memory hierarchy.

The importance of the memory hierarchy has increased with advances in performance of processors. For example, in 1980 microprocessors were often designed without caches, while in 2001 many come with two levels of caches on the chip. As noted in Chapter 1, microprocessor performance improved 55% per year since 1987, and 35% per year until 1986. Figure 5.2 plots CPU performance projections against the historical performance improvement in time to access main memory. Clearly, there is a processor-memory performance gap that computer architects must try to close.

This chapter describes the many ideas invented to overcome the processor-memory performance gap. To put these abstract ideas into practice, throughout the chapter we show examples from the four levels of the memory hierarchy in a computer using the Alpha 21264 microprocessor. Toward the end of the chapter we evaluate the impact of these levels on performance using the SPEC95 benchmark programs.

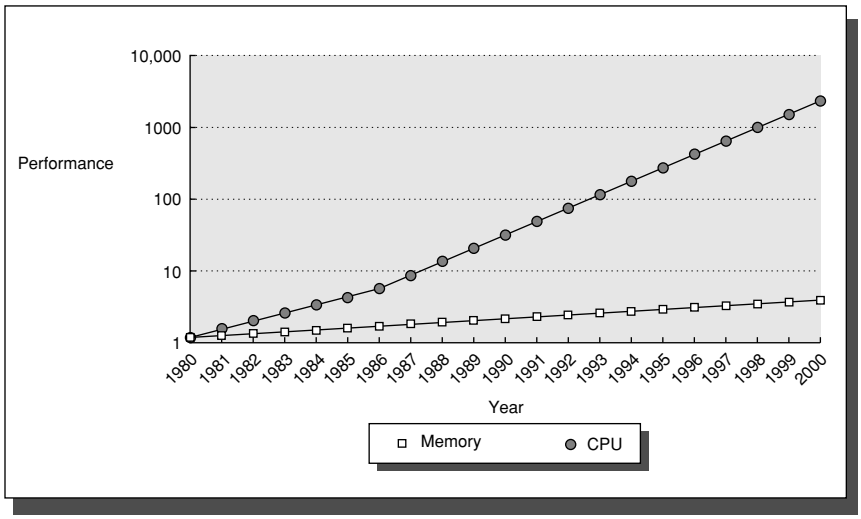


FIGURE 5.2 Starting with 1980 performance as a baseline, the gap in performance between memory and CPUs are plotted over time. Note that the vertical axis must be on a logarithmic scale to record the size of the CPU-DRAM performance gap. The memory baseline is 64-KB DRAM in 1980, with three years to the next generation and a 7% per year performance improvement in latency (see Figure 5.30 on page 444). The CPU line assumes a 1.35 improvement per year until 1986, and a 1.55 improvement thereafter. **<<NOTE: Artist continue the X-axis to 2005: DRAM should be 5.4 in 2005, and CPU should be 25,000 in 2005. This means the Y-axis needs to be extended to 100,000 to accommodate.>>**

The 21264 is a microprocessor designed for desktop and servers. Even these two related classes of computers have different concerns in a memory hierarchy. Desktop computers are primarily running one application at a time on top of an operating system for a single user, whereas server computers may typically have hundreds of users potentially running potentially dozens of applications simultaneously. These characteristics result in more context switches, which effectively increases compulsory miss rates. Thus, desktop computers are concerned more with average latency from the memory hierarchy whereas server computers are also concerned about memory bandwidth. Although protection is important on desktop computers to deter programs from clobbering each other, server computers must prevent one user from accessing another's data, and hence the importance of protection escalates. Server computers also tend to be much larger, with more memory and disk storage, and hence often run much larger applications. In 2001 virtually all servers can be purchased as multiprocessors with hundreds of disks, which places even greater bandwidth demands on the memory hierarchy.

The memory hierarchy of the embedded computers is often quite different from that of the desktop and server. First, embedded computers are often used in real-time applications, and hence programmers must worry about worst case per-

formance. This concern is problematic for caches that improve average case performance, but can degrade worst case performance; we'll mention some techniques to address this later in the chapter. Second, embedded applications are often concerned about power and battery life. The best way to save power is to have less hardware. Hence, embedded computers may not chose hardware-intensive optimizations in the quest of better memory hierarchy performance, as would most desktop and server computers. Third, embedded applications are typically only running one application and use a very simple operating system, if they one at all. Hence, the protection role of the memory hierarchy is often diminished. Finally, the main memory itself may be quite small–less than one megabyte–and there is often no disk storage.

This chapter is a tour of the general principles of memory hierarchy using the desktop as the generic example, but we will take detours to point out where the memory hierarchy of servers and embedded computers diverge from the desktop. Towards the end of the chapter we will pause for two views of the memory hierarchy in addition to the Alpha 21264: the Sony Playstation 2 and the Sun Fire 6800 server. Our first stop is a review.

5.2 Review of the ABCs of Caches

Cache: a safe place for hiding or storing things.

Webster's New World Dictionary of the American Language, Second College Edition (1976)

This section is a quick review of cache basics, covering the following 36 terms:

cache	*cache hit*	*cache miss*	*block*
virtual memory	*page*	*page fault*	*page fault*
memory stall cycles	*miss penalty*	*miss rate*	*address trace*
direct mapped	*fully associative*	*n-way set associative*	*set*
valid bit	*dirty bit*	*least-recently used*	*random replacement*
block address	*block offset*	*tag field*	*index*
write through	*write back*	*write allocate*	*no-write allocate*
instruction cache	*data cache*	*unified cache*	*write buffer*
average memory access time	*hit time*	*misses per instruction*	*write stall*

Readers who know the meaning of such terms should skip to "An Example: The Alpha 21264 Data Cache" on page 387, or even further to section 5.3 on page 390 about cache performance. (If this review goes too quickly, you might want to look at Chapter 7 in *Computer Organization and Design*, which we wrote for readers with less experience.)

For those interested in a review, two particularly important levels of the memory hierarchy are cache and virtual memory.

Cache is the name given to the first level of the memory hierarchy encountered once the address leaves the CPU. Since the principle of locality applies at many levels, and taking advantage of locality to improve performance is popular, the term *cache* is now applied whenever buffering is employed to reuse commonly occurring items. Examples include *file caches*, *name caches*, and so on.

When the CPU finds a requested data item in the cache, it is called a *cache hit*. When the CPU does not find a data item it needs in the cache, a *cache miss* occurs. A fixed-size collection of data containing the requested word, called a *block*, is retrieved from the main memory and placed into the cache. *Temporal locality* tells us that we are likely to need this word again in the near future, so it is useful to place it in the cache where it can be accessed quickly. Because of *spatial locality*, there is high probability that the other data in the block will be needed soon.

The time required for the cache miss depends on both the latency and bandwidth of the memory. Latency determines the time to retrieve the first word of the block, and bandwidth determines the time to retrieve the rest of this block. A cache miss is handled by hardware and causes processors following in-order execution to pause, or stall, until the data are available.

Similarly, not all objects referenced by a program need to reside in main memory. If the computer has *virtual memory*, then some objects may reside on disk. The address space is usually broken into fixed-size blocks, called *pages*. At any time, each page resides either in main memory or on disk. When the CPU references an item within a page that is not present in the cache or main memory, a *page fault* occurs, and the entire page is moved from the disk to main memory. Since page faults take so long, they are handled in software and the CPU is not stalled. The CPU usually switches to some other task while the disk access occurs. The cache and main memory have the same relationship as the main memory and disk.

Figure 5.3 shows the range of sizes and access times of each level in the memory hierarchy for computers ranging from high-end desktops to low-end servers.

Cache Performance Review

Because of locality and the higher speed of smaller memories, a memory hierarchy can substantially improve performance. One method to evaluate cache performance is to expand our CPU execution time equation from Chapter 1. We now account for the number of cycles during which the CPU is stalled waiting for a memory access, which we call the *memory stall cycles*. The performance is then the product of the clock cycle time and the sum of the CPU cycles and the memory stall cycles:

Level	1	2	3	4
Called	Registers	Cache	Main memory	Disk storage
Typical size	< 1 KB	< 16 MB	< 16 GB	> 100 GB
Implementation technology	Custom memory with multiple ports, CMOS	On-chip or off-chip CMOS SRAM	CMOS DRAM	Magnetic disk
Access time (in ns)	0.25 -0.5	0.5 to 25	80-250	5,000,000
Bandwidth (in MB/sec)	20,000-100,000	5,000-10,000	1000-5000	20-150
Managed by	Compiler	Hardware	Operating system	Operating system/operator
Backed by	Cache	Main memory	Disk	CD or Tape

FIGURE 5.3 The typical levels in the hierarchy slow down and get larger as we move away from the CPU for a large workstation or small server. Embedded computers might have no disk storage, and much smaller memories and caches. The access times increase as we move to lower levels of the hierarchy, which makes it feasible to manage the transfer less responsively. The implementation technology shows the typical technology used for these functions. The access time is given in nanoseconds for typical values in 2001; these times will decrease over time. Bandwidth is given in megabytes per second between levels in the memory hierarchy. Bandwidth for disk storage includes both the media and the buffered interfaces.

$$\text{CPU execution time} = (\text{CPU clock cycles} + \text{Memory stall cycles}) \times \text{Clock cycle time}$$

This equation assumes that the CPU clock cycles include the time to handle a cache hit, and that the CPU is stalled during a cache miss. Section 5.3 re-examines this simplifying assumption.

The number of memory stall cycles depends on both the number of misses and the cost per miss, which is called the *miss penalty*:

$$\begin{aligned}\text{Memory stall cycles} &= \text{Number of misses} \times \text{Miss penalty}\\ &= \text{IC} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}\\ &= \text{IC} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss rate} \times \text{Miss penalty}\end{aligned}$$

The advantage of the last form is that the components can be easily measured. We already know how to measure IC (instruction count). Measuring the number of memory references per instruction can be done in the same fashion; every instruction requires an instruction access and we can easily decide if it also requires a data access.

Note that we calculated miss penalty as an average, but we will use it below as if it were a constant. The memory behind the cache may be busy at the time of the miss due prior memory requests or memory refresh (see section 5.9). The number of clock cycles also varies at interfaces between different clocks of the processor, bus, and memory. Thus, please remember that using a single number for miss penalty is a simplification.

The component *miss rate* is simply the fraction of cache accesses that result in a miss (i.e., number of accesses that miss divided by number of accesses). Miss rates can be measured with cache simulators that take an *address trace* of the instruction and data references, simulate the cache behavior to determine which references hit and which miss, and then report the hit and miss totals. Some microprocessors provide hardware to count the number of misses and memory references, which is a much easier and faster way to measure miss rate.

The formula above is an approximation since the miss rates and miss penalties are often different for reads and writes. Memory stall clock cycles could then be defined in terms of the number of memory accesses per instruction, miss penalty (in clock cycles) for reads and writes, and miss rate for reads and writes:

$$\begin{aligned}\text{Memory stall clock cycles} = {} & \text{IC} \times \text{Reads per instruction} \times \text{Read miss rate} \times \text{Read miss penalty} \\ & + \text{IC} \times \text{Writes per instruction} \times \text{Write miss rate} \times \text{Write miss penalty}\end{aligned}$$

We normally simplify the complete formula by combining the reads and writes and finding the average miss rates and miss penalty for reads *and* writes:

$$\text{Memory stall clock cycles} = \text{IC} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss rate} \times \text{Miss penalty}$$

The miss rate is one of the most important measures of cache design, but, as we will see in later sections, not the only measure.

EXAMPLE Assume we have a computer where the clocks per instruction (CPI) is 1.0 when all memory accesses hit in the cache. The only data accesses are loads and stores, and these total 50% of the instructions. If the miss penalty is 25 clock cycles and the miss rate is 2%, how much faster would the computer be if all instructions were cache hits?

ANSWER First compute the performance for the computer that always hits:

$$\begin{aligned}\text{CPU execution time} &= (\text{CPU clock cycles} + \text{Memory stall cycles}) \times \text{Clock cycle} \\ &= (\text{IC} \times \text{CPI} + 0) \times \text{Clock cycle} \\ &= \text{IC} \times 1.0 \times \text{Clock cycle}\end{aligned}$$

Now for the computer with the real cache, first we compute memory stall cycles:

$$\begin{aligned}\text{Memory stall cycles} &= \text{IC} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss rate} \times \text{Miss penalty} \\ &= \text{IC} \times (1 + 0.5) \times 0.02 \times 25 \\ &= \text{IC} \times 0.75\end{aligned}$$

where the middle term (1 + 0.5) represents one instruction access and 0.5

data accesses per instruction. The total performance is thus

$$\begin{aligned}\text{CPU execution time}_{\text{cache}} &= (\text{IC} \times 1.0 + \text{IC} \times 0.75) \times \text{Clock cycle} \\ &= 1.75 \times \text{IC} \times \text{Clock cycle}\end{aligned}$$

The performance ratio is the inverse of the execution times:

$$\begin{aligned}\frac{\text{CPU execution time}_{\text{cache}}}{\cancel{\text{CPU execution time}}} &= \frac{1.75 \times \text{IC} \times \text{Clock cycle}}{\cancel{1.0 \times \text{IC} \times \text{Clock cycle}}} \\ &= 1.75\end{aligned}$$

The computer with no cache misses is 1.75 times faster. n

Some designers prefer measuring miss rate as *misses per instruction* rather than misses per memory reference. These two are related:

$$\frac{\text{Misses}}{\cancel{\text{Instruction}}} = \frac{\text{Miss rate} \times \text{Memory accesses}}{\cancel{\text{Instruction Count}}} = \text{Miss rate} \times \frac{\text{Memory accesses}}{\cancel{\text{Instruction}}}$$

The latter formula is useful when you know the average number of memory accesses per instruction as it allows you to convert miss rate into misses per instruction, and vice versa. For example, we can turn the miss rate per memory reference in the example above into misses per instruction:

$$\frac{\text{Misses}}{\cancel{\text{Instruction}}} = \text{Miss rate} \times \frac{\text{Memory accesses}}{\cancel{\text{Instruction}}} = 0.02 \times 1.5 = 0.030$$

By the way, misses per instruction are often reported as misses per 1000 instructions to show integers instead of fractions. Thus, the answer above could also be expressed as 30 misses per 1000 instructions.

The advantage of misses per instruction is that it is independent of the hardware implementation. For example, the 21264 fetches about twice as many instructions as are actually committed, which can artificially reduce the miss rate if measured as misses per memory reference rather than per instruction. The drawback is that misses per instruction is architecture dependent; for example, the average number of memory accesses per instruction may be very different for an 80x86 versus MIPS. Thus, misses per instruction are most popular with architects working with a single computer family, although the similarity of RISC architectures allows one to give insights into others.

EXAMPLE To show equivalency between the two miss rate equations, let's redo the example above, this time assuming a miss rate per 1000 instructions of 30. What is memory stall time in terms of instruction count?

ANSWER Recomputing the memory stall cycles:

$$\begin{aligned}
\text{Memory stall cycles} &= \text{Number of misses} \times \text{Miss penalty} \\
&= \text{IC} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty} \\
&= \text{IC} / 1000 \times \frac{\text{Misses}}{\text{Instruction} \times 1000} \times \text{Miss penalty} \\
&= \text{IC} / 1000 \times 30 \times 25 \\
&= \text{IC} / 1000 \times 750 \\
&= \text{IC} \times 0.75
\end{aligned}$$

We get the same answer as on page 379. n

Four Memory Hierarchy Questions

We continue our introduction to caches by answering the four common questions for the first level of the memory hierarchy:

Q1: Where can a block be placed in the upper level? (*Block placement*)

Q2: How is a block found if it is in the upper level? (*Block identification*)

Q3: Which block should be replaced on a miss? (*Block replacement*)

Q4: What happens on a write? (*Write strategy*)

The answers to these questions help us understand the different trade-offs of memories at different levels of a hierarchy; hence we ask these four questions on every example.

Q1: Where can a block be placed in a cache?

Figure 5.4 shows that the restrictions on where a block is placed create three categories of cache organization:

- If each block has only one place it can appear in the cache, the cache is said to be *direct mapped*. The mapping is usually

  (Block address) MOD *(Number of blocks in cache)*

- If a block can be placed anywhere in the cache, the cache is said to be *fully associative*.

- If a block can be placed in a restricted set of places in the cache, the cache is *set associative*. A *set* is a group of blocks in the cache. A block is first mapped onto a set, and then the block can be placed anywhere within that set. The set is usually chosen by *bit selection*; that is,

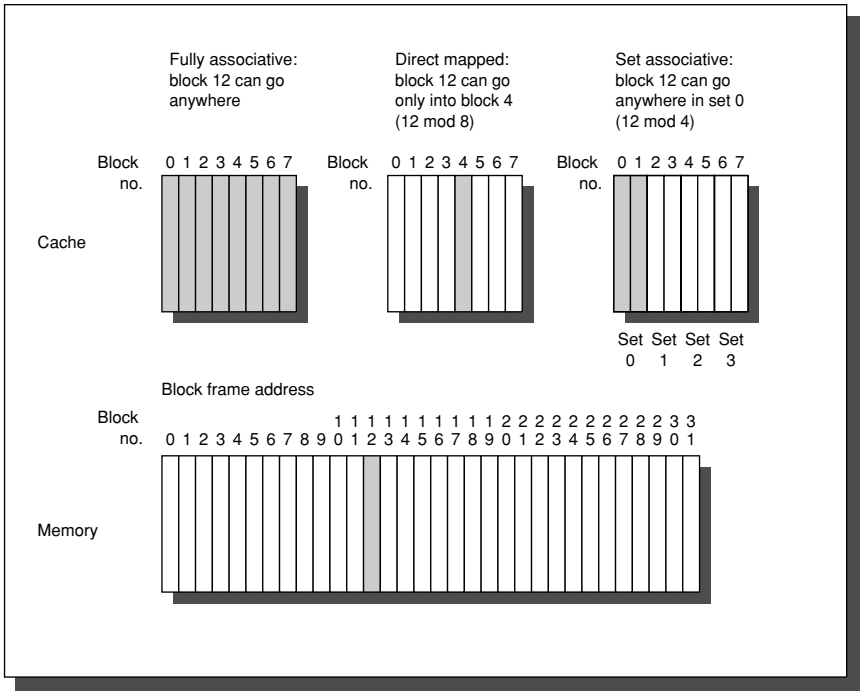


FIGURE 5.4 This example cache has eight block frames and memory has 32 blocks. The three options for caches are shown left to right. In fully associative, block 12 from the lower level can go into any of the eight block frames of the cache. With direct mapped, block 12 can only be placed into block frame 4 (12 modulo 8). Set associative, which has some of both features, allows the block to be placed anywhere in set 0 (12 modulo 4). With two blocks per set, this means block 12 can be placed either in block 0 or in block 1 of the cache. Real caches contain thousands of block frames and real memories contain millions of blocks. The set-associative organization has four sets with two blocks per set, called *two-way set associative*. Assume that there is nothing in the cache and that the block address in question identifies lower-level block 12.

$$\textit{(Block address)} \ \text{MOD} \ \textit{(Number of sets in cache)}$$

If there are n blocks in a set, the cache placement is called *n-way set associative*.

The range of caches from direct mapped to fully associative is really a continuum of levels of set associativity. Direct mapped is simply one-way set associative and a fully associative cache with m blocks could be called m-way set associative. Equivalently, direct mapped can be thought of as having m sets and fully associative as having one set.

The vast majority of processor caches today are direct mapped, two-way set associative, or four-way set associative, for reasons we shall see shortly.

Q2: How is a block found if it is in the cache?

Caches have an address tag on each block frame that gives the block address. The tag of every cache block that might contain the desired information is checked to see if it matches the block address from the CPU. As a rule, all possible tags are searched in parallel because speed is critical.

There must be a way to know that a cache block does not have valid information. The most common procedure is to add a *valid bit* to the tag to say whether or not this entry contains a valid address. If the bit is not set, there cannot be a match on this address.

Before proceeding to the next question, let's explore the relationship of a CPU address to the cache. Figure 5.5 shows how an address is divided. The first division is between the *block address* and the *block offset*. The block frame address can be further divided into the *tag* field and the *index* field. The block-offset field selects the desired data from the block, the index field selects the set, and the tag field is compared against it for a hit. Although the comparison could be made on more of the address than the tag, there is no need because of the following:

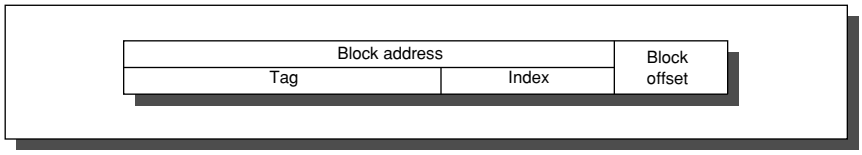


FIGURE 5.5 The three portions of an address in a set-associative or direct-mapped cache. The tag is used to check all the blocks in the set and the index is used to select the set. The block offset is the address of the desired data within the block. Fully associative caches have no index field.

- The offset should not be used in the comparison, since the entire block is present or not, and hence all block offsets result in a match by definition.
- Checking the index is redundant, since it was used to select the set to be checked. An address stored in set 0, for example, must have 0 in the index field or it couldn't be stored in set 0; set 1 must have an index value of 1; and so on. This optimization saves hardware and power by reducing the width of memory size for the cache tag.

If the total cache size is kept the same, increasing associativity increases the number of blocks per set, thereby decreasing the size of the index and increasing the size of the tag. That is, the tag-index boundary in Figure 5.5 moves to the right with increasing associativity, with the end point of fully associative caches having no index field.

Q3: Which block should be replaced on a cache miss?

When a miss occurs, the cache controller must select a block to be replaced with the desired data. A benefit of direct-mapped placement is that hardware decisions are simplified—in fact, so simple that there is no choice: Only one block frame is checked for a hit, and only that block can be replaced. With fully associative or set-associative placement, there are many blocks to choose from on a miss. There are three primary strategies employed for selecting which block to replace:

- *Random*—To spread allocation uniformly, candidate blocks are randomly selected. Some systems generate pseudorandom block numbers to get reproducible behavior, which is particularly useful when debugging hardware.
- *Least-recently used* (LRU)—To reduce the chance of throwing out information that will be needed soon, accesses to blocks are recorded. Relying on the past to predict the future, the block replaced is the one that has been unused for the longest time. LRU relies on a corollary of locality: If recently used blocks are likely to be used again, then a good candidate for disposal is the least-recently used block.
- *First In First Out* (FIFO)—Because LRU can be complicated to calculate, this approximates LRU by determining the *oldest* block rather than the LRU.

A virtue of random replacement is that it is simple to build in hardware. As the number of blocks to keep track of increases, LRU becomes increasingly expensive and is frequently only approximated. Figure 5.6 shows the difference in miss rates between LRU, random, and FIFO replacement.

	Associativity								
	Two-way			Four-way			Eight-way		
Size	LRU	Random	FIFO	LRU	Random	FIFO	LRU	Random	FIFO
16 KB	114.1	117.3	115.5	111.7	115.1	113.3	109.0	111.8	110.4
64 KB	103.4	104.3	103.9	102.4	102.3	103.1	99.7	100.5	100.3
256 KB	92.2	92.1	92.5	92.1	92.1	92.5	92.1	92.1	92.5

FIGURE 5.6 Data cache misses per 1000 instructions comparing least-recently used, random, and first-in, first-out replacement for several sizes and associativities. There is little difference between LRU and random for the largest-size cache, with LRU outpeforming the others for smaller caches. FIFO generally outperforms random in the smaller cache sizes. These data were collected for a block size of 64 bytes for the Alpha architecture using ten SPEC2000 benchmarks. Five are from SPECint2000 (gap, gcc, gzip, mcf, and perl) and five are from SPECfp2000 (applu, art, equake, lucas, and swim). We will use this computer and these benchmarks in most figures in this chapter.

Q4: What happens on a write?

Reads dominate processor cache accesses. All instruction accesses are reads, and most instructions don't write to memory. Figure 2.32 on page 149 in Chapter 2

suggests a mix of 10% stores and 37% loads for MIPS programs, making writes 10%/(100% + 37% + 10%) or about 7% of the overall memory traffic. Of the *data cache* traffic, writes are 10%/(37% + 10%) or about 21%. Making the common case fast means optimizing caches for reads, especially since processors traditionally wait for reads to complete but need not wait for writes. Amdahl's Law (section 1.6, page 29) reminds us, however, that high-performance designs cannot neglect the speed of writes.

Fortunately, the common case is also the easy case to make fast. The block can be read from cache at the same time that the tag is read and compared, so the block read begins as soon as the block address is available. If the read is a hit, the requested part of the block is passed on to the CPU immediately. If it is a miss, there is no benefit—but also no harm in desktop and server computers; just ignore the value read. Embedded's emphasis on power generally means avoiding unnecessary work, which might lead the designer to separate data read from address check so that data is not read on a miss.

Such optimism is not allowed for writes. Modifying a block cannot begin until the tag is checked to see if the address is a hit. Because tag checking cannot occur in parallel, writes normally take longer than reads. Another complexity is that the processor also specifies the size of the write, usually between 1 and 8 bytes; only that portion of a block can be changed. In contrast, reads can access more bytes than necessary without fear; once again, embedded designers might weigh the power benefits of reading less.

The write policies often distinguish cache designs. There are two basic options when writing to the cache:

- *Write through* —The information is written to both the block in the cache *and* to the block in the lower-level memory.
- *Write back* —The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.

To reduce the frequency of writing back blocks on replacement, a feature called the *dirty bit* is commonly used. This status bit indicates whether the block is *dirty* (modified while in the cache) or *clean* (not modified). If it is clean, the block is not written back on a miss, since identical information to the cache is found in lower levels.

Both write back and write through have their advantages. With write back, writes occur at the speed of the cache memory, and multiple writes within a block require only one write to the lower-level memory. Since some writes don't go to memory, write back uses less memory bandwidth, making write back attractive in multiprocessors which are common in servers. Since write back uses the rest of the memory hierarchy and memory buses less than write through, it also saves power, making it attractive for embedded applications.

Write through is easier to implement than write back. The cache is always clean, so unlike write back read misses never result in writes to the lower level. Write through also has the advantage that the next lower level has the most current copy of the data, which simplifies data coherency. Data coherency (see sec-

tion 5.12) is important for multiprocessors and for I/O, which we examine in Chapters 6 and 7.

As we shall see, I/O and multiprocessors are fickle: they want write back for processor caches to reduce the memory traffic and write through to keep the cache consistent with lower levels of the memory hierarchy.

When the CPU must wait for writes to complete during write through, the CPU is said to *write stall*. A common optimization to reduce write stalls is a *write buffer*, which allows the processor to continue as soon as the data is written to the buffer, thereby overlapping processor execution with memory updating. As we shall see shortly, write stalls can occur even with write buffers.

Since the data are not needed on a write, there are two are two options on a write miss:

- *Write allocate* —The block is allocated on a write miss, followed by the write-hit actions above. In this natural option, write misses act like read misses.
- *No-write allocate*—This apparently unusual alternative is write misses do *not* affect the cache. Instead, the block is modified only in the lower level memory.

Thus, blocks stays out of the cache in no-write allocate until the program tries to read the blocks, but even blocks that are only written will still be in the cache with write allocate. Let's look at an example.

EXAMPLE Assume a fully associative write back cache with many cache entries that starts empty. Below is a sequence of five memory operations (the address is in parentheses):

Write Mem[100];
WriteMem[100];
Read Mem[200];
WriteMem[200];
WriteMem[100].

What are the number of hits and misses with using no-write allocate versus write allocate?

ANSWER For no-write allocate, the address 100 is not in the cache, and there is no allocation on write, so the first two writes will result in misses. Address 200 is also not in the cache, so the read is also a miss. The subsequent write to address 200 is a hit. The last write to 100 is still a miss. The result for no write allocate is 4 misses and 1 hit.

For write allocate, the first accesses to 100 and 200 are misses, and the rest are hits since 100 and 200 are both found in the cache. Thus, the result for write allocate is 2 misses and 3 hits. ▫

Either write-miss policy could be used with write through or write back. Normally, write-back caches use write allocate, hoping that subsequent writes to that block will be captured by the cache. Write-through caches often use no-write allocate. The reasoning is that even if there are subsequent writes to that block, the writes must still go to the lower level memory, so what's to be gained?

An Example: The Alpha 21264 Data Cache

To give substance to these ideas, Figure 5.7 shows the organization of the data cache in the Alpha 21264 microprocessor that is found in the Compaq AlphaServer ES40, one of several models that use it. The cache contains 65,536 (64K) bytes of data in 64-byte blocks with two-way set-associative placement, write back, and write allocate on a write miss.

Let's trace a cache hit through the steps of a hit as labeled in Figure 5.7. (The four steps are shown as circled numbers.) As we shall see later (Figure 5.36), the 21264 processor presents a 48-bit virtual address to the cache for tag comparison, which is simultaneously translated into a 44-bit physical address. (It also optionally supports 43-bit virtual addresses with 41-bit physical addresses.)

The reason Alpha doesn't use all 64 bits of virtual address is that its designers don't think anyone needs that big of virtual address space yet, and the smaller size simplifies the Alpha virtual address mapping. The designers planned to grow the virtual address in future microprocessors.

The physical address coming into the cache is divided into two fields: the 38-bit block address and 6-bit block offset ($64 = 2^6$ and $38 + 6 = 44$). The block address is further divided into an address tag and cache index. Step 1 shows this division.

The cache index selects the tag to be tested to see if the desired block is in the cache. The size of the index depends on cache size, block size, and set associativity. For the 21264 cache the set associativity is set to two, and we calculate the index as follows:

$$2^{\text{index}} = \frac{\text{Cache size}}{\cancel{\text{Block size} \times \text{Set associativity}}} = \frac{65536}{\cancel{64 \times 2}} = 512 = 2^9$$

Hence, the index is 9 bits wide, and the tag is 38 – 9 or 29 bits wide. Although that is the index needed to select the proper block, 64 bytes is much more than the CPU wants to consume at once. Hence, it makes more sense to organize the data portion of the cache memory 8 bytes wide, which is the natural data word of the 64-bit Alpha processor. Thus, in addition to 9 bits to index the proper cache block, 3 more bits from the block offset are used to index the proper 8 bytes.

Index selection is step 2 in Figure 5.7. The two tags are compared and the winner is selected. (Section 5.10 explains how the 21264 handles virtual address translation.)

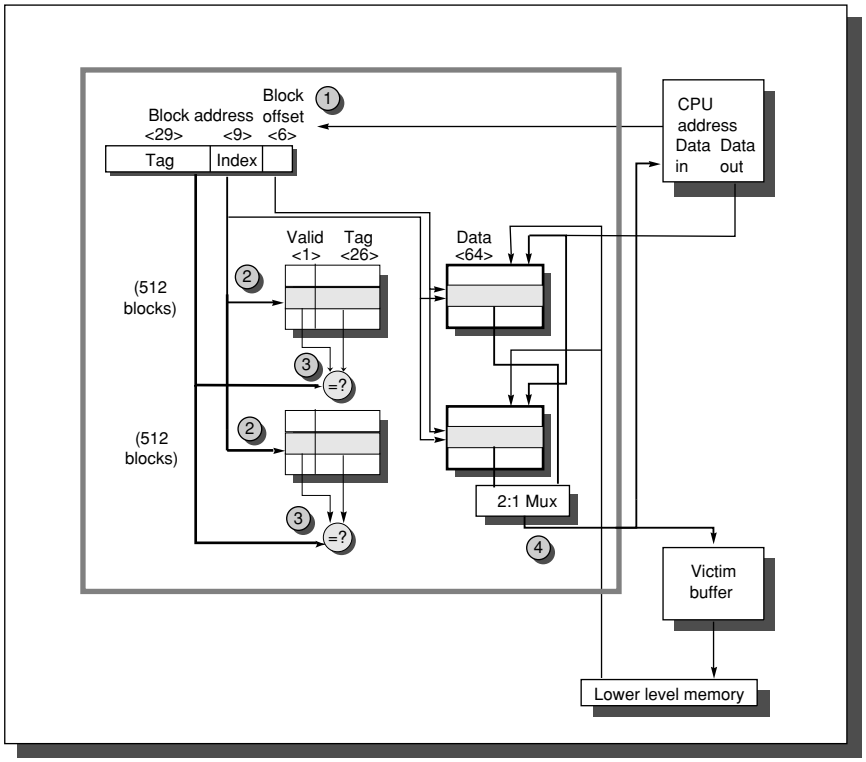


FIGURE 5.7 The organization of the data cache in the Alpha 21264 microprocessor. The 64-KB cache is two-way set associative with 64-byte blocks. The 9-bit index selects between 512 sets. The four steps of a read hit, shown as circled numbers in order of occurrence, label this organization. Three bits of the block offset join the index to supply the RAM address to select the proper 8 bytes. Thus, the cache holds two groups 4096 64-bit words, with each group containing half of the 512 sets. Although not exercised in this example, the line from lower level memory to the cache is used on a miss to load the cache. The size of address leaving the CPU is 44 bits because it is a physical address and not a virtual address. Figure 5.36 on page 454 explains how the Alpha maps from the virtual to physical for a cache access. **<<REVIZE DRAWING, including the more realistic drawing of the memory blocks as in old figure 5.10 in CA:AQA 2/e>>**

After reading the two tags from the cache, they are compared to the tag portion of the block address from the CPU. This comparison is step 3 in the figure. To be sure the tag contains valid information, the valid bit must be set or else the results of the comparison are ignored.

Assuming one tag does match, the final step is to signal the CPU to load the proper data from the cache by using the winning input from a 2:1 multiplexor. The 21264 allows three clock cycles for these four steps, so the instructions in the following two clock cycles would wait if they tried to use the result of the load.

Handling writes is more complicated than handling reads in the 21264, as it is in any cache. If the word to be written is in the cache, the first three steps are the same. Since the 21264 executes out-of-order, only after it signals that the instruction has committed and the cache tag comparison indicates a hit are the data are written to the cache.

So far we have assumed the common case of a cache hit. What happens on a miss? On a read miss, the cache sends a signal to the processor telling it the data is not yet available, and 64 bytes are read from the next level of the hierarchy. The path to the next lower level in the 21264 is 16 bytes wide. In the 667 MHz AlphaServer ES40 it takes 2.25 ns per transfer, or 9 ns for all 64 bytes. Since the data cache is set associative, there is a choice on which block to replace. The 21264 does *round robin* selection, dedicating a bit for every two blocks to remember where to go next. Unlike LRU, which selects the block that was referenced longest ago, round robin selects the block that was filled longest ago. Round robin is easier to implement since it is only updated on a miss rather than on every hit. Replacing a block means updating the data, the address tag, the valid bit, and the round robin bit.

Since the 21264 uses write back, the old data block could have been modified, and hence it cannot simply be discarded. The 21264 keeps one dirty bit per block to record if the block was written. If the "victim" was modified, its data and address are sent to the Victim Buffer. (This structure is similar to a *write buffer* in other computers.) The 21264 has space for eight victim blocks. In parallel with other cache actions, it writes victim blocks to the next level of the hierarchy. If the Victim Buffer is full, the cache must wait.

A write miss is very similar to a read miss, since the 21264 allocates a block on a read or a write miss.

We have seen how it works, but the *data* cache cannot supply all the memory needs of the processor: the processor also needs instructions. Although a single cache could try to supply both, it can be a bottleneck. For example, when a load or store instruction is executed, the pipelined processor will simultaneously request both a data word *and* an instruction word. Hence, a single cache would present a structural hazard for loads and stores, leading to stalls. One simple way to conquer this problem is to divide it: one cache is dedicated to instructions and another to data. Separate caches are found in most recent processors, including the Alpha 21264. Hence, it has a 64-KB instruction cache as well as the 64-KB data cache.

The CPU knows whether it is issuing an instruction address or a data address, so there can be separate ports for both, thereby doubling the bandwidth between the memory hierarchy and the CPU. Separate caches also offer the opportunity of optimizing each cache separately: different capacities, block sizes, and associativities may lead to better performance. (In contrast to the instruction caches and data caches of the 21264, the terms *unified* or *mixed* are applied to caches that can contain either instructions or data.)

Size	Instruction cache	Data cache	Unified cache
8 KB	8.16	44.0	63.0
16 KB	3.82	40.9	51.0
32 KB	1.36	38.4	43.3
64 KB	0.61	36.9	39.4
128 KB	0.30	35.3	36.2
256 KB	0.02	32.6	32.9

FIGURE 5.8 Miss per 1000 instructions for instruction, data, and unified caches of different sizes. The percentage of instruction references is about 78%. The data are for two-way associative caches with 64-byte blocks for the same computer and benchmarks as Figure 5.6.

Figure 5.8 shows that instruction caches have lower miss rates than data caches. Separating instructions and data removes misses due to conflicts between instruction blocks and data blocks, but the split also fixes the cache space devoted to each type. Which is more important to miss rates? A fair comparison of separate instruction and data caches to unified caches requires the total cache size to be the same. For example, a separate 16-KB instruction cache and 16-KB data cache should be compared to a 32-KB unified cache. Calculating the average miss rate with separate instruction and data caches necessitates knowing the percentage of memory references to each cache. Figure 2.32 on page 149 suggests the split is 100%/(100% + 37% + 10%) or about 78% instruction references to (37% + 10%)/(100% + 37% + 10%) or about 22% data references. Splitting affects performance beyond what is indicated by the change in miss rates, as we shall see shortly.

5.3 Cache Performance

Because instruction count is independent of the hardware, it is tempting to evaluate CPU performance using that number. As we saw in Chapter 1, however, such indirect performance measures have waylaid many a computer designer. The corresponding temptation for evaluating memory-hierarchy performance is to concentrate on miss rate, because it, too, is independent of the speed of the hardware. As we shall see, miss rate can be just as misleading as instruction count. A better measure of memory-hierarchy performance is the *average memory access time*:

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

where *Hit time* is the time to hit in the cache; we have seen the other two terms before. The components of average access time can be measured either in absolute time—say, 0.25 to 1.0 nanoseconds on a hit—or in the number of clock cycles that

the CPU waits for the memory—such as a miss penalty of 75 to 100 clock cycles. Remember that average memory access time is still an indirect measure of performance; although it is a better measure than miss rate, it is not a substitute for execution time.

This formula can help us decide between split caches and a unified cache.

EXAMPLE Which has the lower miss rate: a 16-KB instruction cache with a 16-KB data cache or a 32-KB unified cache? Use the miss rates in Figure 5.7 to help calculate the correct answer assuming 47% of the instructions are data transfer instructions. Assume a hit takes 1 clock cycle and the miss penalty is 100 clock cycles. A load or store hit takes 1 extra clock cycle on a unified cache if there is only one cache port to satisfy two simultaneous requests. Using the pipelining terminology of the previous chapter, the unified cache leads to a structural hazard. What is the average memory access time in each case? Assume write-through caches with a write buffer and ignore stalls due to the write buffer.

ANSWER First let's convert misses per 1000 instructions into miss rates. Solving the general formula is from above, miss rate is

$$\text{Miss rate} = \frac{\dfrac{\text{Misses}}{\text{1000 Instructions}} / 1000}{\dfrac{\text{Memory accesses}}{\text{Instruction}}}$$

Since every instruction access has exactly 1 memory access to fetch the instruction, the instruction miss rate is:

$$\text{Miss rate}_{\text{16 KB Instruction}} = \frac{3.82/1000}{1.00} = 0.004$$

Since 47% of the instructions are data transfers, the data miss rate is:

$$\text{Miss rate}_{\text{16 KB Data}} = \frac{40.9/1000}{0.47} = 0.087$$

The unified miss rate needs to account for instruction and data accesses:

$$\text{Miss rate}_{\text{32 KB Unified}} = \frac{43.3/1000}{1.00 + 0.47} = 0.029$$

As stated above, about 78% of the memory accesses are instruction references. Thus, the overall miss rate for the split caches is

$$(78\% \times 0.004) + (22\% \times 0.087) = 0.022$$

Thus, a 32-KB unified cache has a higher effective miss rate than two16-KB caches.

The average memory access time formula can be divided into instruction and data accesses:

$$\begin{aligned}&\text{Average memory access time}\\&= \%\ \text{instructions} \times (\text{Hit time} + \text{Instruction miss rate} \times \text{Miss penalty}) +\\&\%\ \text{data} \times (\text{Hit time} + \text{Data miss rate} \times \text{Miss penalty})\end{aligned}$$

Therefore, the time for each organization is

$$\begin{aligned}&\text{Average memory access time}_{\text{split}}\\&= 78\% \times (1 + 0.004 \times 100) + 22\% \times (1 + 0.087 \times 100)\\&= (78\% \times 1.38) + (22\% \times 9.70) = 1.078 + 2.134 = 3.21\\&\text{Average memory access time}_{\text{unified}}\\&= 78\% \times (1 + 0.029 \times 100) + 22\% \times (1 + 1 + 0.029 \times 100)\\&= (78\% \times 3.95) + (22\% \times 4.95) = 3.080 + 1.089 = 4.17\end{aligned}$$

Hence, the split caches in this example—which offer two memory ports per clock cycle, thereby avoiding the structural hazard—also have a better average memory access time than the single-ported unified cache. n

Average memory access time and Processor Performance

An obvious question is whether average memory access time due to cache misses predicts processor performance.

First, there are other reasons for stalls, such as contention due to I/O devices using memory. Designers often assume that all memory stalls are due to cache misses, since the memory hierarchy typically dominates other reasons for stalls. We use this simplifying assumption here, but beware to account for *all* memory stalls when calculating final performance.

Second, the answer depends also on the CPU. If we have an in-order execution CPU (See Chapter 3), then the answer is basically yes. The CPU stalls during misses, and the memory stall time is strongly correlated to average memory access time. Let's make that assumption for now, but we'll return to out-of-order CPUs in the next subsection.

As stated in the prior section, we can model CPU time as:

$$\text{CPU time} = (\text{CPU execution clock cycles} + \text{Memory stall clock cycles}) \times \text{Clock cycle time}$$

This formula raises the question whether the clock cycles for a cache hit should be considered part of CPU execution clock cycles or part of memory stall clock cycles. Although either convention is defensible, the most widely accepted is to include hit clock cycles in CPU execution clock cycles.

We can now explore the impact of caches on performance.

EXAMPLE Let's use an in-order execution computer for the first example, such as the UltraSPARC III (see section 5.15). Assume the cache miss penalty is 100 clock cycles, and all instructions normally take 1.0 clock cycles (ignoring memory stalls). Assume the average miss rate is 2%, there is an average of 1.5 memory references per instruction, and that the average number of cache misses per 1000 instructions is 30. What is the impact on performance when behavior of the cache is included? Calculate the impact using both misses per instruction and miss rate.

ANSWER

$$\text{CPU time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \frac{\text{Memory stall clock cycles}}{\text{Instruction}} \right) \times \text{Clock cycle time}$$

The performance, including cache misses, is

$$\begin{aligned} \text{CPU time}_{\text{with cache}} &= \text{IC} \times (1.0 + (30 / 1000 \times 100)) \times \text{Clock cycle time} \\ &= \text{IC} \times 4.00 \times \text{Clock cycle time} \end{aligned}$$

Now calculating performance using miss rate:

$$\text{CPU time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss penalty} \right) \times \text{Clock cycle time}$$

$$\begin{aligned} \text{CPU time}_{\text{with cache}} &= \text{IC} \times (1.0 + (1.5 \times 2\% \times 100)) \times \text{Clock cycle time} \\ &= \text{IC} \times 4.00 \times \text{Clock cycle time} \end{aligned}$$

The clock cycle time and instruction count are the same, with or without a cache. Thus, CPU time increases fourfold, with CPI from 1.00 for a "perfect cache" to 4.00 with a cache that can miss. Without any memory hierarchy at all the CPI would increase again to 1.0 + 100 × 1.5 or 151—a factor of almost 40 times longer than a system with a cache! n

As this example illustrates, cache behavior can have enormous impact on performance. Furthermore, cache misses have a double-barreled impact on a CPU with a low CPI and a fast clock:

1. The lower the $\text{CPI}_{\text{execution}}$, the higher the *relative* impact of a fixed number of cache miss clock cycles.
2. When calculating CPI, the cache miss penalty is measured in CPU clock cycles for a miss. Therefore, even if memory hierarchies for two computers are identical, the CPU with the higher clock rate has a larger number of clock cycles per miss and hence a higher memory portion of CPI.

The importance of the cache for CPUs with low CPI and high clock rates is thus greater, and, consequently, greater is the danger of neglecting cache behavior in assessing performance of such computers. Amdahl's Law strikes again!

Although minimizing average memory access time is a reasonable goal—and we will use it in much of this chapter—keep in mind that the final goal is to reduce CPU execution time. The next example shows how these two can differ.

EXAMPLE What is the impact of two different cache organizations on the performance of a CPU? Assume that the CPI with a perfect cache is 2.0, the clock cycle time is 1.0 ns, there are 1.5 memory references per instruction, the size of both caches is 64 KB, and both have a block size of 64 bytes. One cache is direct mapped and the other is two-way set associative. Figure 5.7 on page 388 shows that for set-associative caches we must add a multiplexor to select between the blocks in the set depending on the tag match. Since the speed of the CPU is tied directly to the speed of a cache hit, assume the CPU clock cycle time must be stretched 1.25 times to accommodate the selection multiplexor of the set-associative cache. To the first approximation, the cache miss penalty is 75 ns for either cache organization. (In practice, it is normally rounded up or down to an integer number of clock cycles.) First, calculate the average memory access time, and then CPU performance. Assume the hit time is one clock cycle, the miss rate of a direct-mapped 64-KB cache is 1.4%, and the miss rate for a two-way set-associative cache of the same size is 1.0%.

ANSWER Average memory access time is

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

Thus, the time for each organization is

$$\text{Average memory access time}_{\text{1-way}} = 1.0 + (.014 \times 75) = 2.05 \text{ ns}$$
$$\text{Average memory access time}_{\text{2-way}} = 1.0 \times 1.25 + (.010 \times 75) = 2.00 \text{ ns}$$

The average memory access time is better for the two-way set-associative cache.

CPU performance is

$$\text{CPU time} = \text{IC} \times \left(\text{CPI}_{\text{Execution}} + \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty} \right) \times \text{Clock cycle time}$$

$$= \text{IC} \times \left[(\text{CPI}_{\text{Execution}} \times \text{Clock cycle time}) + \left(\text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss penalty} \times \text{Clock cycle time} \right) \right]$$

Substituting 75 ns for (Miss penalty × Clock cycle time), the performance

of each cache organization is

$$\text{CPU time}_{\text{1-way}} = \text{IC} \times (2 \times 1.0 + (1.5 \times 0.014 \times 75)) = 3.58 \times \text{IC}$$
$$\text{CPU time}_{\text{2-way}} = \text{IC} \times (2 \times 1.0 \times 1.25 + (1.5 \times 0.010 \times 75)) = 3.63 \times \text{IC}$$

and relative performance is

$$\frac{\text{CPU time}_{\text{2-way}}}{\text{CPU time}_{\text{1-way}}} = \frac{3.63 \times \text{Instruction count}}{3.58 \times \text{Instruction count}} = \frac{3.63}{3.58} = 1.01$$

In contrast to the results of average memory access time comparison, the direct-mapped cache leads to slightly better average performance because the clock cycle is stretched for *all* instructions for the two-way set-associative case, even if there are fewer misses. Since CPU time is our bottom-line evaluation, and since direct mapped is simpler to build, the preferred cache is direct mapped in this example. n

Miss Penalty and Out-of-Order Execution Processors

For an out-of-order execution processor, how do you define miss penalty? Is it the full latency of the miss to memory, or is it just the "exposed" or non-overlapped latency when the processor must stall? This question does not arise in processors which stall until the data miss completes.

Let's redefine memory stalls to lead to a new definition of miss penalty as non-overlapped latency:

$$\frac{\text{Memory stall cycles}}{\text{Instruction}} = \frac{\text{Misses}}{\text{Instruction}} \times (\text{Total miss latency} - \text{Overlapped miss latency})$$

Similarly, as some out-of-order CPUs stretch the hit time, that portion of the performance equation could be divided total hit latency less overlapped hit latency. This equation could be further expanded to account for contention for memory resources in an out-of-order processor by dividing total miss latency into latency without contention and latency due to contention. Let's just concentrate on miss latency.

We now have to decide

- *length of memory latency*: what to consider as the start and the end of a memory operation in an out-of-order processor; and
- *length of latency overlap*: what is the start of overlap with for the processor (or equivalently, when do we say a memory operation is stalling the processor).

Given the complexity of out-of-order execution processors, there is no single correct definition.

Since only committed operations are seen at the retirement pipeline stage, we say a processor is stalled in a clock cycle if it does not retire the maximum possible number of instructions in that cycle. We attribute that stall to the first instruction that could not be retired. This definition is by no means foolproof. For example, applying an optimization to improve a certain stall time may not always improve execution time because another type of stall—hidden behind the targeted stall—may now be exposed.

For latency, we could start measuring from the time the memory instruction is queued in the instruction window, or when the address is generated, or when the instruction is actually sent to the memory system. Any option works as long as it is used in a consistent fashion.

EXAMPLE Let's redo the example above, but this time assuming the processor with the longer clock cycle time supports out-of-order execution yet still has a direct mapped cache. Assume 30% of the 75 ns miss penalty can be overlapped; that is, the average CPU memory stall time is now 52.5 ns.

ANSWER Average memory access time for the out-of-order computer is

$$\text{Average memory access time}_{\text{1-way,OOO}} = 1.0 \times 1.25 + (0.014 \times 52.5) = 1.99 \text{ ns}$$

The performance of the OOO cache is

$$\text{CPU time}_{\text{1-way,OOO}} = \text{IC} \times (2 \times 1.0 \times 1.25 + (1.5 \times 0.014 \times 52.5)) = 3.60 \times \text{IC}$$

Hence, despite a much slower clock cycle time and the higher miss rate of a direct mapped cache, the out-of-order computer can be slightly faster if it can hide 30% of the miss penalty. n

In summary, although the state-of-the-art in defining and measuring memory stalls for out-of-order processors is not perfect and is relatively complex, readers should be aware of the issues for they significantly affect performance.

Improving Cache Performance

To help summarize this section and to act as a handy reference, Figure 5.9 lists the cache equations in this chapter.

The increasing gap between CPU and main memory speeds shown in Figure 5.2 has attracted the attention of many architects. A bibliographic search for the years 1989 to 2001 revealed more than 5000 research papers on the subject of caches. Your authors' job was to survey all 5000 papers, decide what is and is not worthwhile, translate the results into a common terminology, reduce the results to their essence, write in an intriguing fashion, and provide just the right amount of detail!

$$2^{\text{index}} = \frac{\text{Cache size}}{\text{Block size} \times \text{Set associativity}}$$

$$\text{CPU execution time} = (\text{CPU clock cycles} + \text{Memory stall cycles}) \times \text{Clock cycle time}$$

$$\text{Memory stall cycles} = \text{Number of misses} \times \text{Miss penalty}$$

$$\text{Memory stall cycles} = \text{IC} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

$$\frac{\text{Misses}}{\text{Instruction}} = \text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}}$$

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

$$\text{CPU execution time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \frac{\text{Memory stall clock cycles}}{\text{Instruction}}\right) \times \text{Clock cycle time}$$

$$\text{CPU execution time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}\right) \times \text{Clock cycle time}$$

$$\text{CPU execution time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss penalty}\right) \times \text{Clock cycle time}$$

$$\frac{\text{Memory stall cycles}}{\text{Instruction}} = \frac{\text{Misses}}{\text{Instruction}} \times (\text{Total miss latency} - \text{Overlapped miss latency})$$

$$\text{Average memory access time} = \text{Hit time}_{\text{L1}} + \text{Miss rate}_{\text{L1}} \times (\text{Hit time}_{\text{L2}} + \text{Miss rate}_{\text{L2}} \times \text{Miss penalty}_{\text{L2}})$$

$$\frac{\text{Memory stall cycles}}{\text{Instruction}} = \frac{\text{Misses}_{\text{L1}}}{\text{Instruction}} \times \text{Hit time}_{\text{L2}} + \frac{\text{Misses}_{\text{L2}}}{\text{Instruction}} \times \text{Miss penalty}_{\text{L2}}$$

FIGURE 5.9 Summary of performance equations in this chapter. The first equation calculates with cache index size, but the rest help evaluates performance. The final two equations deal with multilevel caches, is explained early in the next section. They are included here to help make the figure a useful reference.

Fortunately, this task was simplified by our long standing policy of only including ideas in this book that have made their way into commercially viable computers. In computer architecture, many ideas look much better on paper than in silicon.

The average memory access time formula gave us a framework to present the surviving cache optimizations for improving cache performance or power:

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

Hence, we organize 16 cache optimizations into four categories:

- Reducing the miss penalty (Section 5.4): multilevel caches, critical word first, read miss before write miss, merging write buffers, victim caches;
- Reducing the miss rate (Section 5.5): larger block size, larger cache size, higher associativity, pseudo-associativity, and compiler optimizations;
- Reducing the miss penalty or miss rate via parallelism (Section 5.6): nonblocking caches, hardware prefetching, and compiler prefetching;
- Reducing the time to hit in the cache (Section 5.7): small and simple caches, avoiding address translation, and pipelined cache access.

Figure 5.26 on page 436 concludes with a summary of the implementation complexity and the performance benefits of the 17 techniques presented.

5.4 Reducing Cache Miss Penalty

Reducing cache misses has been the traditional focus of cache research, but the cache performance formula assures us that improvements in miss penalty can be just as beneficial as improvements in miss rate. Moreover, Figure 5.2 shows that technology trends have improved the speed of processors faster than DRAMs, making the relative cost of miss penalties increase over time.

We give five optimizations here to address increasing miss penalty. Perhaps the most interesting optimization is the first, which adds more levels of caches to reduce miss penalty.

First Miss Penalty Reduction Technique: Multi-Level Caches

Many techniques to reduce miss penalty affect the CPU. This technique ignores the CPU, concentrating on the interface between the cache and main memory.

The performance gap between processors and memory leads the architect to this question: Should I make the cache faster to keep pace with the speed of CPUs, or make the cache larger to overcome the widening gap between the CPU and main memory?

One answer is: both. Adding another level of cache between the original cache and memory simplifies the decision. The first-level cache can be small enough to match the clock cycle time of the fast CPU. Yet the second-level cache can be large enough to capture many accesses that would go to main memory, thereby lessening the effective miss penalty.

Although the concept of adding another level in the hierarchy is straightforward, it complicates performance analysis. Definitions for a second level of cache are not always straightforward. Let's start with the definition of *average memory access time* for a two-level cache. Using the subscripts L1 and L2 to refer, respectively, to a first-level and a second-level cache, the original formula is

$$\text{Average memory access time} = \text{Hit time}_{L1} + \text{Miss rate}_{L1} \times \text{Miss penalty}_{L1}$$

and

$$\text{Miss penalty}_{L1} = \text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2}$$

so

$$\text{Average memory access time} = \text{Hit time}_{L1} + \text{Miss rate}_{L1} \times (\text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2})$$

In this formula, the second-level miss rate is measured on the leftovers from the first-level cache. To avoid ambiguity, these terms are adopted here for a two-level cache system:

- *Local miss rate*—This rate is simply the number of misses in a cache divided by the total number of memory accesses to this cache. As you would expect, for the first-level cache it is equal to Miss rate_{L1} and for the second-level cache it is Miss rate_{L2}.
- *Global miss rate*—The number of misses in the cache divided by the total number of memory accesses generated by the CPU. Using the terms above, the global miss rate for the first-level cache is still just Miss rate_{L1} but for the second-level cache it is Miss $\text{rate}_{L1} \times$ Miss rate_{L2}.

This local miss rate is large for second level caches because the first-level cache skims the cream of the memory accesses. This is why the global miss rate is the more useful measure: it indicates what fraction of the memory accesses that leave the CPU go all the way to memory.

Here is a place where the misses per instruction metric shines. Instead of confusion about local or global miss rates, we just expand memory stalls per instruction to add the impact of a second level cache.

Average memory stalls per instruction = Misses per $\text{instruction}_{L1} \times$ Hit time_{L2} + Misses per $\text{instruction}_{L2} \times$ Miss penalty_{L2}.

EXAMPLE Suppose that in 1000 memory references there are 40 misses in the first-level cache and 20 misses in the second-level cache. What are the various miss rates? Assume the miss penalty from L2 cache to Memory is 100 clock cycles, the hit time of L2 cache is 10 clock cycles, the Hit time of L1 is 1 clock cycles, and there are 1.5 memory references per instruction. What is the average memory access time and average stall cycles per instruction? Ignore the impact of writes.

ANSWER The miss rate (either local or global) for the first-level cache is 40/1000 or 4%. The local miss rate for the second-level cache is 20/40 or 50%. The global miss rate of the second-level cache is 20/1000 or 2%. Then

Average memory access time = Hit time_{L1} + Miss $\text{rate}_{L1} \times$ (Hit time_{L2} + Miss $\text{rate}_{L2} \times$ Miss penalty_{L2}

$= 1 + 4\% \times (10 + 50\% \times 100) = 1 + 4\% \times 60 = 3.4$ clock cycles

To see how many misses we get per instruction, we divide 1000 memory references by 1.5 memory references per instruction, which yields 667 instructions. Thus, we need to multiply the misses by 1.5 to get the number of misses per 1000 instructions. We have 40×1.5 or 60 L1 misses 20×1.5 or 30 L2 misses per 1000 instructions. For average memory stalls per instruction, assuming the misses are distributed uniformly between instructions and data:

Average memory stalls per instruction = Misses per instruction$_{L1}$ × Hit time$_{L2}$ + Misses per instruction$_{L2}$ × Miss penalty$_{L2}$
= (60/1000) × 10 + (30/1000) × 100
= 0.060 × 10 + 0.030 × 100 = 3.6 clock cycles
If we subtract the L1 hit time from AMAT and then multiplying by the average number of memory references per instruction we get the same average memory stalls per instruction:
(3.4 - 1.0) x 1.5 = 2.4 x 1.5 = 3.6 clock cycles.
As this example shows, there is less confusion with multilevel caches when calculating using misses per instruction versus miss rates. n

Note that these formulas are for combined reads and writes, assuming a write-back first-level cache. Obviously, a write-through first-level cache will send *all* writes to the second level, not just the misses, and a write buffer might be used.

Figures 5.10 and 5.11 show how miss rates and relative execution time change with the size of a second-level cache for one design. From these figures we can gain two insights. The first is that the global cache miss rate is very similar to the single cache miss rate of the second-level cache, provided that the second-level cache is much larger than the first-level cache. Hence, our intuition and knowledge about the first-level caches apply. The second insight is that the local cache rate is *not* a good measure of secondary caches; it is a function of the miss rate of the first-level cache, and hence can vary by changing the first-level cache. Thus, the global cache miss rate should be used when evaluating second-level caches.

With these definitions in place, we can consider the parameters of second-level caches. The foremost difference between the two levels is that the speed of the first-level cache affects the clock rate of the CPU, while the speed of the second-level cache only affects the miss penalty of the first-level cache. Thus, we can consider many alternatives in the second-level cache that would be ill chosen for the first-level cache. There are two major questions for the design of the second-level cache: Will it lower the average memory access time portion of the CPI, and how much does it cost?

The initial decision is the size of a second-level cache. Since everything in the first-level cache is likely to be in the second-level cache, the second-level cache should be much bigger than the first. If second-level caches are just a little bigger, the local miss rate will be high. This observation inspires design of huge second-level caches—the size of main memory in older computers! One question is whether set associativity makes more sense for second-level caches.

EXAMPLE Given the data below, what is the impact of second-level cache associativity on its miss penalty?

- Hit time$_{L2}$ for direct mapped = 10 clock cycles
- Two-way set associativity increases hit time by 0.1 clock cycles to 10.1 clock cycles

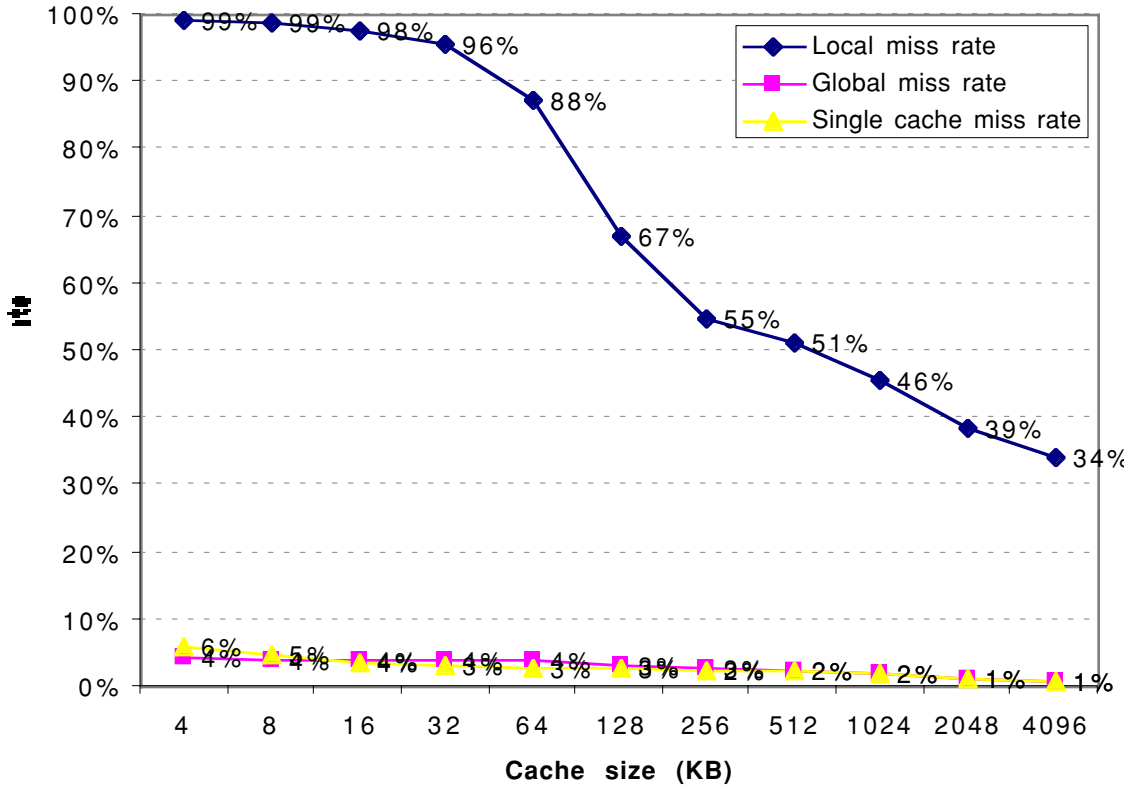


FIGURE 5.10 Miss rates versus cache size for multilevel caches. Second-level caches *smaller* than the sum of the two 64-KB first level make little sense, as reflected in the high miss rates. After 256 KB the single cache is within 10% of the global miss rates. The miss rate of a single-level cache versus size is plotted against the local miss rate and global miss rate of a second-level cache using a 32-KB first-level cache. The L2 Caches (unified) were 2-way set-associative with LRU replacement. Each had split L1instruction and data caches that were 64KB 2-way set-associative with LRU replacement. The block size for both L1 and L2 caches was 64 bytes. Data was collected for as in Figure 5.6.**<<Artist please separate numbers for graphs at bottom>>**

- Local miss rate$_{L2}$ for direct mapped = 25%
- Local miss rate$_{L2}$ for two-way set associative = 20%
- Miss penalty$_{L2}$ = 100 clock cycles

ANSWER For a direct-mapped second-level cache, the first-level cache miss penalty is

$$\text{Miss penalty}_{1\text{- way L2}} = 10 + 25\% \times 100 = 35.0 \text{ clock cycles}$$

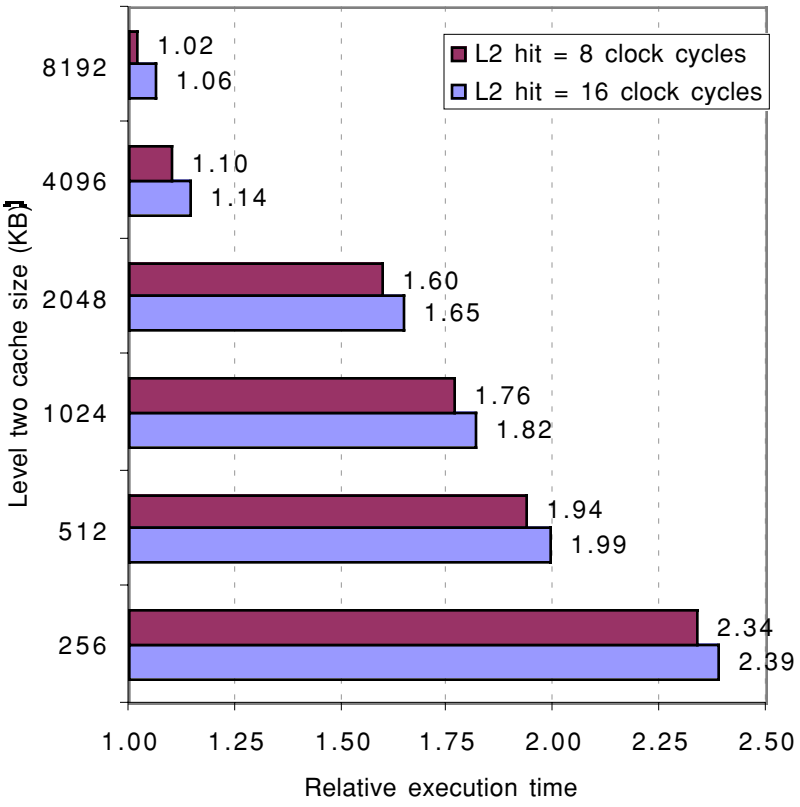


FIGURE 5.11 Relative execution time by second-level cache size. The two bars are for different clock cycles for a L2 cache hit. The reference execution time of 1.00 is for an 8192-KB second-level cache with a one-clock-cycle latency on a second-level hit. These data were collected the same way as in Figure 5.10, using a simulator to imitate the Alpha 21264.

Adding the cost of associativity increases the hit cost only 0.1 clock cycles, making the new first-level cache miss penalty

$$\text{Miss penalty}_{2\text{- way L2}} = 10.1 + 20\% \times 100 = 30.1 \text{ clock cycles}$$

In reality, second-level caches are almost always synchronized with the first-level cache and CPU. Accordingly, the second-level hit time must be an integral number of clock cycles. If we are lucky, we shave the second-level hit time to 10 cycles; if not, we round up to 11 cycles. Either choice is an improvement over the direct-mapped second-level cache:

$$\text{Miss penalty}_{2\text{- way L2}} = 10 + 20\% \times 100 = 30.0 \text{ clock cycles}$$

$$\text{Miss penalty}_{2\text{- way L2}} = 11 + 20\% \times 100 = 31.0 \text{ clock cycles}$$

n

Now we can reduce the miss penalty by reducing the *miss rate* of the second-level caches.

Another consideration concerns whether data in the first-level cache is in the second-level cache. *Multilevel inclusion* is the natural policy for memory hierarchies: L1 data is always present in L2. Inclusion is desirable because consistency between I/O and caches (or among caches in a multiprocessor) can be determined just by checking the second-level cache (see section 8.7).

One drawback to inclusion is that measurements can suggest smaller blocks for the smaller first-level cache and larger blocks for the larger second-level cache. For example, the Pentium 4 has 64-byte blocks in its L1 caches and 128-byte blocks in its L2 cache. Inclusion can still be maintained with more work on a second-level miss. The second-level cache must invalidate all first-level blocks that map onto the second-level block to be replaced, causing a slightly higher first-level miss rate. To avoid such problems, many cache designers keep the block size the same in all levels of caches.

However, what if the designer can only afford an L2 cache that is slightly bigger than the L1 cache? Should a significant portion of its space be used as a redundant copy of the L1 cache? In such cases a sensible opposite policy is *multilevel exclusion*: L1 data is *never* found in L2 cache. Typically, with exclusion a cache miss in L1 results in a swap of blocks between L1 and L2 instead of a replacement of a L1 block with a L2 block. This policy prevents wasting space in L2 cache. For example, the AMD Athlon chip obeys the exclusion property since it has two 64 KB first level caches and only a 256 KB L2 cache.

As these issues illustrate, although a novice might design the first and second-level caches independently, the designer of the first-level cache has a simpler job given a compatible second-level cache. It is less of a gamble to use a write through, for example, if there is a write-back cache at the next level to act as a backstop for repeated writes.

The essence of all cache designs is balancing fast hits and few misses. For second-level caches, there are many fewer hits than in the first-level cache, so the emphasis shifts to fewer misses. This insight leads to much larger caches and techniques to lower the miss rate described in section 5.5, such as higher associativity and larger blocks.

Second Miss Penalty Reduction Technique: Critical Word First and Early Restart

Multilevel caches require extra hardware to reduce miss penalty, but not this second technique. It is based on the observation that the CPU normally needs just one word of the block at a time. This strategy is impatience: Don't wait for the full block to be loaded before sending the requested word and restarting the CPU. Here are two specific strategies:

- *Critical word first*—Request the missed word first from memory and send it to the CPU as soon as it arrives; let the CPU continue execution while filling the rest of the words in the block. Critical-word-first fetch is also called *wrapped* fetch and *requested word first*.
- *Early restart*—Fetch the words in normal order, but as soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution.

Generally these techniques only benefit designs with large cache blocks, since the benefit is low unless blocks are large. The problem is that given spatial locality, there is more than random chance that the next miss is to the remainder of the block. In such cases, the effective miss penalty is the time from the miss until the second piece arrives.

EXAMPLE Let's assume a computer has a 64-byte cache block, an L2 cache that takes 11 clock cycles to get the critical 8 bytes, and then 2 clock cycles per 8 bytes to fetch the rest of the block.(These parameters are similar to the AMD Athlon.) Calculate the average miss penalty for critical word first, assuming that there will be no other accesses to the rest of the block until it is completely fetched. Then calculate assuming the following instructions reads data sequentially 8 bytes at a time from the rest of the block. Compare the times with and without critical word first.

ANSWER The average miss penalty is 11 clock cycles for critical word first. The Athlon can issue two loads per clock cycle, which is faster than the L2 cache can supply data. Thus, it would take 11 + (8-1) x 2 or 25 clock cycles for the CPU to sequentially read a full cache block. Without critical word first, it would take 25 clocks cycle to load the block, and then 8/2 or 4 clocks to issue the loads, giving 29 clock cycles total. ▫

As this example illustrates, the benefits of critical word first and early restart depend on the size of the block and the likelihood of another access to the portion of the block that has not yet been fetched.

The next technique takes overlap between the CPU and cache miss penalty even further to reduce the average miss penalty.

Third Miss Penalty Reduction Technique: Giving Priority to Read Misses over Writes

This optimization serves reads before writes have been completed. We start with looking at complexities of a write buffer.

With a write-through cache the most important improvement is a write buffer (page 386) of the proper size. Write buffers, however, do complicate memory accesses in that they might hold the updated value of a location needed on a read miss.

EXAMPLE Look at this code sequence:

```
SW R2, 512(R0)          ; M[512] ← R3   (cache index 0)
LW R1,1024(R0)          ; R1 ← M[1024]  (cache index 0)
LW R2,512(R0)           ; R2 ← M[512]   (cache index 0)
```

Assume a direct-mapped, write-through cache that maps 512 and 1024 to the same block, and a four-word write buffer. Will the value in R2 always be equal to the value in R3?

ANSWER Using the terminology from Chapter 3, this is a read-after-write data hazard in memory. Let's follow a cache access to see the danger. The data in R3 are placed into the write buffer after the store. The following load uses the same cache index and is therefore a miss. The second load instruction tries to put the value in location 512 into register R2; this also results in a miss. If the write buffer hasn't completed writing to location 512 in memory, the read of location 512 will put the old, wrong value into the cache block, and then into R2. Without proper precautions, R3 would not be equal to R2! n

The simplest way out of this dilemma is for the read miss to wait until the write buffer is empty. The alternative is to check the contents of the write buffer on a read miss, and if there are no conflicts and the memory system is available, let the read miss continue. Virtually all desktop and server processors use the latter approach, giving reads priority over writes.

The cost of writes by the processor in a write-back cache can also be reduced. Suppose a read miss will replace a dirty memory block. Instead of writing the dirty block to memory, and then reading memory, we could copy the dirty block to a buffer, then read memory, and *then* write memory. This way the CPU read, for which the processor is probably waiting, will finish sooner. Similar to the situation above, if a read miss occurs, the processor can either stall until the buffer is empty or check the addresses of the words in the buffer for conflicts.

Fourth Miss Penalty Reduction Technique: Merging Write Buffer

This technique also involves write buffers, this time improving their efficiency.

Write through caches rely on write buffers, as all stores must be sent to the next lower level of the hierarchy. As mentioned above, even write back caches

use a simple buffer when a block is replaced. If the write buffer is empty, the data and the full address are written in the buffer, and the write is finished from the CPU's perspective; the CPU continues working while the write buffer prepares to write the word to memory. If the buffer contains other modified blocks, the addresses can be checked to see if the address of this new data matches the address of the valid write buffer entry. If so, the new data are combined with that entry, called *write merging*.

If the buffer is full and there is no address match, the cache (and CPU) must wait until the buffer has an empty entry. This optimization uses the memory more efficiently since multiword writes are usually faster than writes performed one word at a time.

The optimization also reduces stalls due to the write buffer being full. Figure 5.12 shows a write buffer with and without write merging. Assume we had four entries in the write buffer, and each entry could hold four 64-bit words. Without this optimization, four stores to sequential addresses would fill the buffer at one word per entry, even though these four words when merged exactly fit within a single entry of the write buffer.

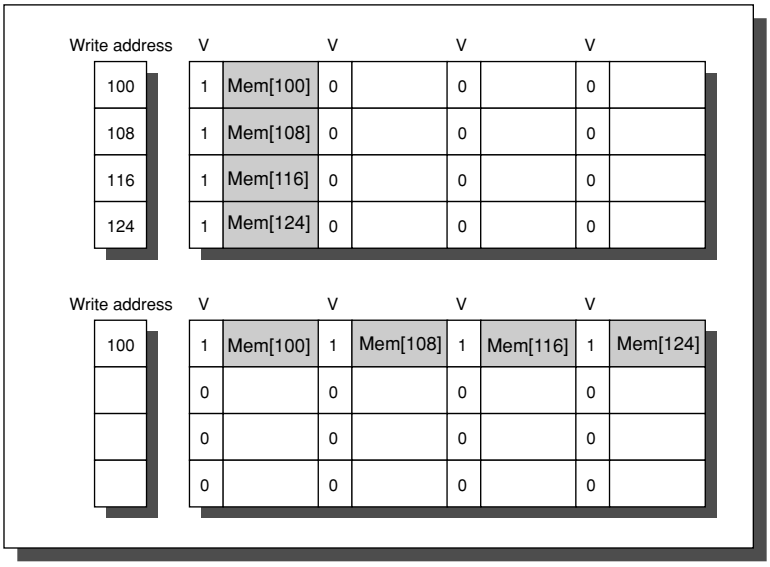


FIGURE 5.12 To illustrate write merging, the write buffer on top does not use it while the write buffer on the bottom does. The four writes are merged into a single buffer entry with write merging; without it, the buffer is full even though three-fourths of each entry is wasted. The buffer has four entries, and each entry holds four 64-bit words. The address for each entry is on the left, with valid bits (V) indicating whether or not the next sequential eight bytes are occupied in this entry. (Without write merging, the words to the right in the upper drawing would only be used for instructions which wrote multiple words at the same time.)

Note that input/output device registers are often mapped into the physical address space, as is the case of the 21264. These I/O addresses cannot allow write merging, as separate I/O registers may not act like an array of words in memory. For example, they may require one addresses and data word per register rather than multiword writes using a single address.

Fifth Miss Penalty Reduction Technique: Victim Caches

One approach to lower miss penalty is to remember what was discarded in case it is needed again. Since the discarded data has already been fetched, it can be used again at small cost.

Such "recycling" requires a small, fully associative cache between a cache and its refill path. Figure 5.13 shows the organization. This *victim cache* contains only blocks that are discarded from a cache because of a miss—"victims"—and are checked on a miss to see if they have the desired data before going to the next lower-level memory. If it is found there, the victim block and cache block are swapped. The AMD Athlon has a victim cache with eight entries.

Jouppi [1990] found that victim caches of one to five entries are effective at reducing misses, especially for small, direct-mapped data caches. Depending on the program, a four-entry victim cache might remove one quarter of the misses in a 4-KB direct-mapped data cache.

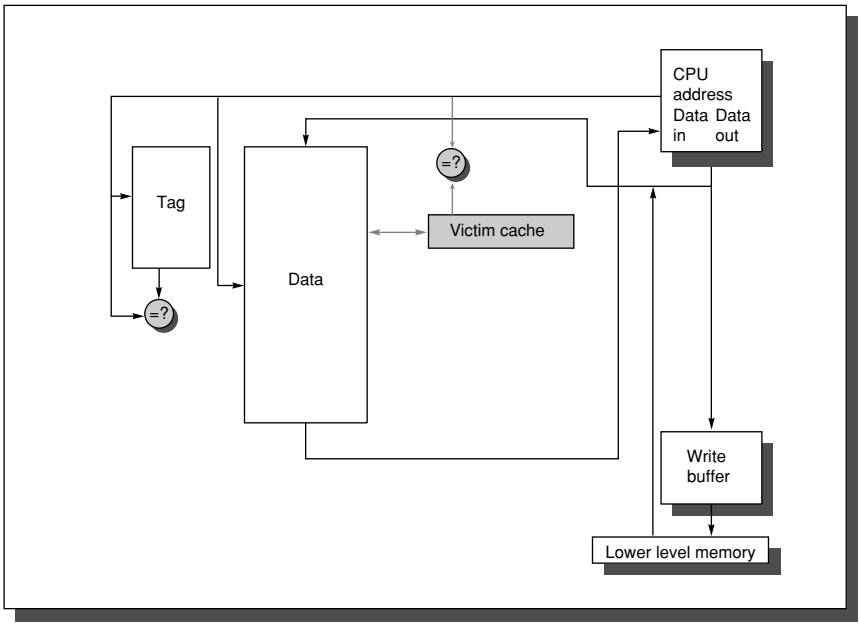


FIGURE 5.13 Placement of victim cache in the memory hierarchy. Although it reduces miss penalty, the victim cache is aimed at reducing the damage done by conflict misses, described in the next section. Jouppi [1990] found the four-entry victim cache could reduce the miss penalty for 20% to 95% of conflict misses.

Summary of Miss Penalty Reduction Techniques

The processor-memory performance gap of Figure 5.2 on page 375 determines the miss penalty, and as the gap grows so do techniques that try to close it. We present five in this section. The first technique follows the proverb "the more the merrier": assuming the principle of locality will keep applying recursively, just keep adding more levels of increasingly larger caches until you are happy. The second technique is impatience: it retrieves the word of the block that caused the miss rather than waiting for the full block to arrive. The next technique is preference. It gives priority to reads over writes since the processor generally waits for reads but continues after launching writes. The fourth technique is companionship, combining writes to sequential words into a single block to create a more efficient transfer to memory. Finally comes a cache equivalent of recycling, as a victim cache keeps a few discarded blocks available for when the fickle primary cache wants a word that it recently discarded. All these techniques help with miss penalty, but multilevel caches is probably the most important.

Testimony of the importance of miss penalty is that most desktop and server computers use the first four of optimizations. Yet most cache research has concentrated on reducing the miss rate, so that is where we go in the next section.

5.5 Reducing Miss Rate

The classical approach to improving cache behavior is to reduce miss rates, and we present five techniques to do so. To gain better insights into the causes of misses, we first start with a model that sorts all misses into three simple categories:

- *Compulsory*—The very first access to a block *cannot be* in the cache, so the block must be brought into the cache. These are also called *cold start misses* or *first reference misses*.
- *Capacity*—If the cache cannot contain all the blocks needed during execution of a program, capacity misses (in addition to compulsory misses) will occur because of blocks being discarded and later retrieved.
- *Conflict*—If the block placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory and capacity misses) will occur because a block may be discarded and later retrieved if too many blocks map to its set. These misses are also called *collision misses* or *interference misses*. The idea is that hits in a fully associative cache which become misses in an N-way set associative cache are due to more than N requests on some popular sets.

Figure 5.14 shows the relative frequency of cache misses, broken down by the "three C's." Compulsory misses are those that occur in an infinite cache. Capacity misses are those that occur in a fully associative cache. Conflict misses are those that occur going from fully associative to 8-way associative, 4-way associative, and so on. Figure 5.15 presents the same data graphically. The top graph

Cache size	Degree associative	Total miss rate	Miss rate components (relative percent) (*Sum = 100% of total miss rate*)					
			Compulsory		Capacity		Conflict	
4 KB	1-way	0.098	0.0001	0.1%	0.070	72%	0.027	28%
4 KB	2-way	0.076	0.0001	0.1%	0.070	93%	0.005	7%
4 KB	4-way	0.071	0.0001	0.1%	0.070	99%	0.001	1%
4 KB	8-way	0.071	0.0001	0.1%	0.070	100%	0.000	0%
8 KB	1-way	0.068	0.0001	0.1%	0.044	65%	0.024	35%
8 KB	2-way	0.049	0.0001	0.1%	0.044	90%	0.005	10%
8 KB	4-way	0.044	0.0001	0.1%	0.044	99%	0.000	1%
8 KB	8-way	0.044	0.0001	0.1%	0.044	100%	0.000	0%
16 KB	1-way	0.049	0.0001	0.1%	0.040	82%	0.009	17%
16 KB	2-way	0.041	0.0001	0.2%	0.040	98%	0.001	2%
16 KB	4-way	0.041	0.0001	0.2%	0.040	99%	0.000	0%
16 KB	8-way	0.041	0.0001	0.2%	0.040	100%	0.000	0%
32 KB	1-way	0.042	0.0001	0.2%	0.037	89%	0.005	11%
32 KB	2-way	0.038	0.0001	0.2%	0.037	99%	0.000	0%
32 KB	4-way	0.037	0.0001	0.2%	0.037	100%	0.000	0%
32 KB	8-way	0.037	0.0001	0.2%	0.037	100%	0.000	0%
64 KB	1-way	0.037	0.0001	0.2%	0.028	77%	0.008	23%
64 KB	2-way	0.031	0.0001	0.2%	0.028	91%	0.003	9%
64 KB	4-way	0.030	0.0001	0.2%	0.028	95%	0.001	4%
64 KB	8-way	0.029	0.0001	0.2%	0.028	97%	0.001	2%
128 KB	1-way	0.021	0.0001	0.3%	0.019	91%	0.002	8%
128 KB	2-way	0.019	0.0001	0.3%	0.019	100%	0.000	0%
128 KB	4-way	0.019	0.0001	0.3%	0.019	100%	0.000	0%
128 KB	8-way	0.019	0.0001	0.3%	0.019	100%	0.000	0%
256 KB	1-way	0.013	0.0001	0.5%	0.012	94%	0.001	6%
256 KB	2-way	0.012	0.0001	0.5%	0.012	99%	0.000	0%
256 KB	4-way	0.012	0.0001	0.5%	0.012	99%	0.000	0%
256 KB	8-way	0.012	0.0001	0.5%	0.012	99%	0.000	0%
512 KB	1-way	0.008	0.0001	0.8%	0.005	66%	0.003	33%
512 KB	2-way	0.007	0.0001	0.9%	0.005	71%	0.002	28%
512 KB	4-way	0.006	0.0001	1.1%	0.005	91%	0.000	8%
512 KB	8-way	0.006	0.0001	1.1%	0.005	95%	0.000	4%

FIGURE 5.14 Total miss rate for each size cache and percentage of each according to the "three C's." Compulsory misses are independent of cache size, while capacity misses decrease as capacity increases, and conflict misses decrease as associativity increases. Figure 5.15 shows the same information graphically. Note that the 2:1 cache rule of thumb (inside front cover) is supported by the statistics in this table through 128 KB: a direct-mapped cache of size *N* has about the same miss rate as a 2-way set-associative cache of size *N*/2. Caches larger than 128 KB do not prove that rule. Note that the Capacity column is also the fully associative miss rate. Data was collected as in Figure 5.6 using LRU replacement.

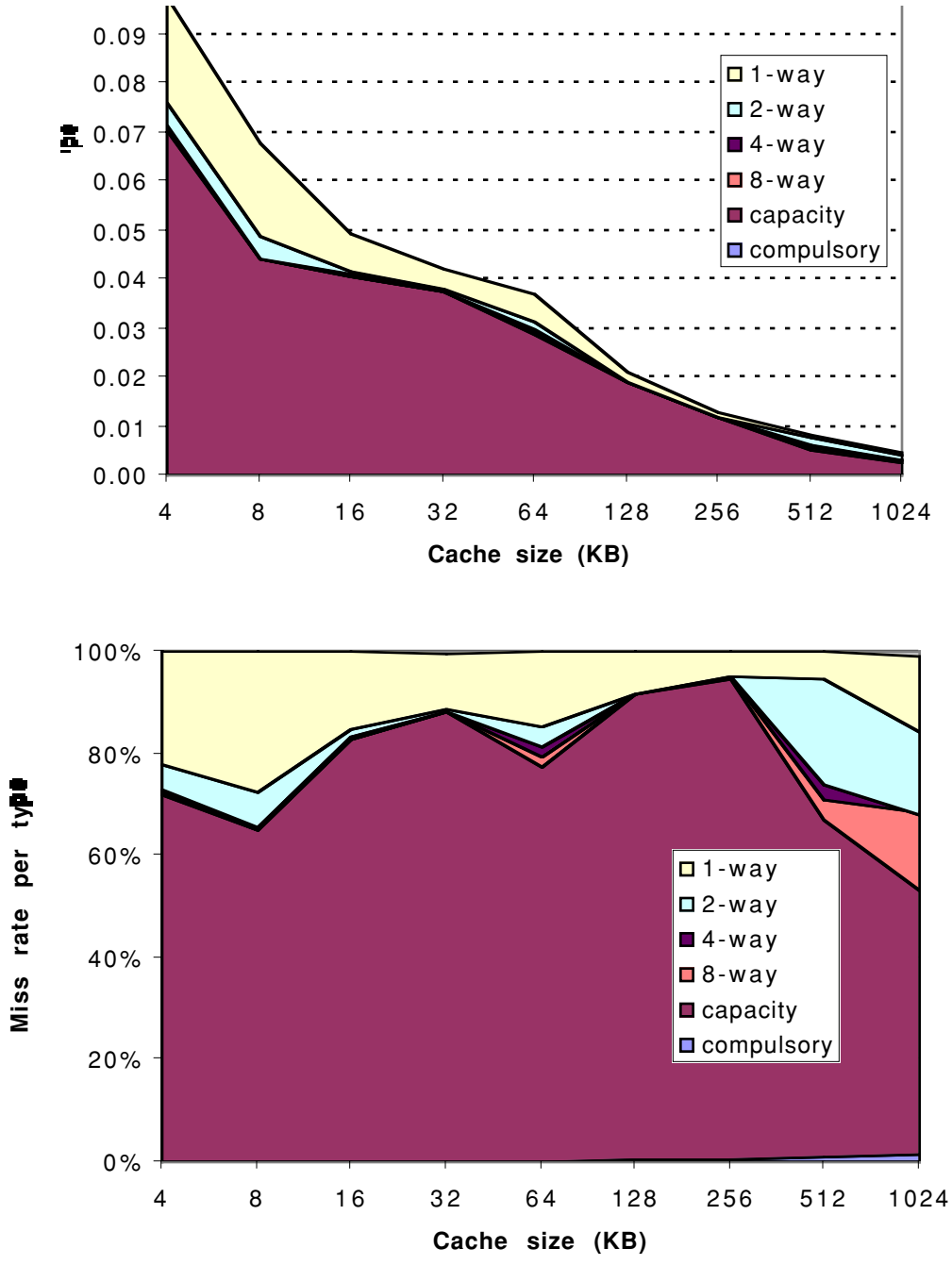


FIGURE 5.15 Total miss rate (top) and distribution of miss rate (bottom) for each size cache according to three C's for the data in Figure 5.14. The top diagram is the actual D-cache miss rates, while the bottom diagram shows percentage in each category. (Space allows the graphs to show one extra cache size than can fit in Figure 5.14.

shows absolute miss rates; the bottom graph plots percentage of all the misses by type of miss as a function of cache size.

To show the benefit of associativity, conflict misses are divided into misses caused by each decrease in associativity. Here are the four divisions of conflict misses and how they are calculated:

- *Eight-way*—conflict misses due to going from fully associative (no conflicts) to eight-way associative
- *Four-way*—conflict misses due to going from eight-way associative to four-way associative
- *Two-way*—conflict misses due to going from four-way associative to two-way associative
- *One-way*—conflict misses due to going from two-way associative to one-way associative (direct mapped)

As we can see from the figures, the compulsory miss rate of the SPEC2000 programs is very small, as it is for many long-running programs.

Having identified the three C's, what can a computer designer do about them? Conceptually, conflicts are the easiest: Fully associative placement avoids all conflict misses. Full associativity is expensive in hardware, however, and may slow the processor clock rate (see the example above), leading to lower overall performance.

There is little to be done about capacity except to enlarge the cache. If the upper-level memory is much smaller than what is needed for a program, and a significant percentage of the time is spent moving data between two levels in the hierarchy, the memory hierarchy is said to *thrash*. Because so many replacements are required, thrashing means the computer runs close to the speed of the lower-level memory, or maybe even slower because of the miss overhead.

Another approach to improving the three C's is to make blocks larger to reduce the number of compulsory misses, but, as we shall see, large blocks can increase other kinds of misses.

The three C's give insight into the cause of misses, but this simple model has its limits; it gives you insight into average behavior but may not explain an individual miss. For example, changing cache size changes conflict misses as well as capacity misses, since a larger cache spreads out references to more blocks. Thus, a miss might move from a capacity miss to a conflict miss as cache size changes. Note that the three C's also ignore replacement policy, since it is difficult to model and since, in general, it is less significant. In specific circumstances the replacement policy can actually lead to anomalous behavior, such as poorer miss rates for larger associativity, which contradicts the three C's model. (Some have proposed using an address trace to determine optimal placement to avoid placement misses from the 3 Cs model; we've not followed that advice here.)

Alas, many of the techniques that reduce miss rates also increase hit time or miss penalty. The desirability of reducing miss rates using the five techniques presented in the rest of this section must be balanced against the goal of making the whole system fast. This first example shows the importance of a balanced perspective.

First Miss Rate Reduction Technique: Larger Block Size

The simplest way to reduce miss rate is to increase the block size. Figure 5.16 shows the trade-off of block size versus miss rate for a set of programs and cache sizes. Larger block sizes will reduce compulsory misses. This reduction occurs because the principle of locality has two components: temporal locality and spatial locality. Larger blocks take advantage of spatial locality.

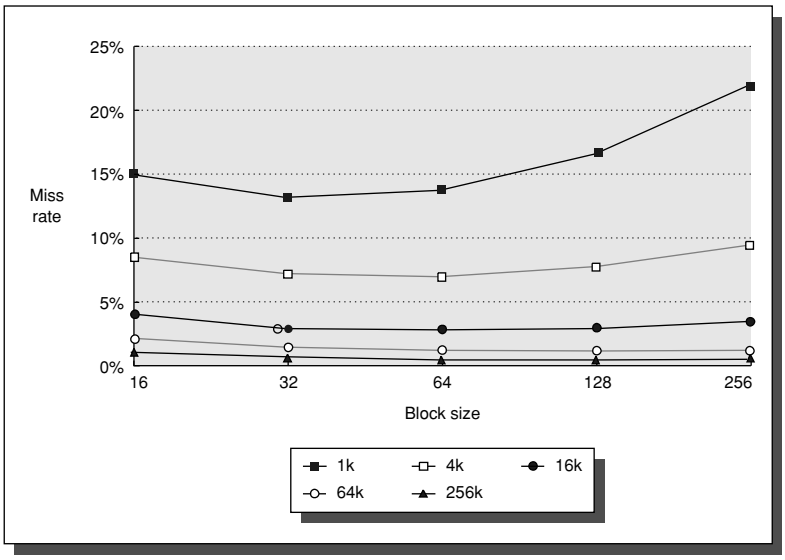


FIGURE 5.16 Miss rate versus block size for five different-sized caches. Note that miss rate actually goes up is the block size is too large relative to the cache size. Each line represents a cache of different size. Figure 5.17 shows the data used to plot these lines. Unfortunately, SPEC2000 traces would take too long if block size was included, so these data are based on SPEC92 on a DECstation 5000 (Gee et al [1993]). **<<Artist: Drop 1K from graph and legend.; then scale Y axis to 0% to 10%>>**

At the same time, larger blocks increase the miss penalty. Since they reduce the number of blocks in the cache, larger blocks may increase conflict misses and even capacity misses if the cache is small. Clearly, there is little reason to increase the block size to such a size that it *increases* the miss rate. There is also no benefit to reducing miss rate if it increases the average memory access time. The increase in miss penalty may outweigh the decrease in miss rate.

Block size	Cache size			
	4K	16K	64K	256K
16	8.57%	3.94%	2.04%	1.09%
32	7.24%	2.87%	1.35%	0.70%
64	7.00%	2.64%	1.06%	0.51%
128	7.78%	2.77%	1.02%	0.49%
256	9.51%	3.29%	1.15%	0.49%

FIGURE 5.17 Actual miss rate versus block size for five different-sized caches in Figure 5.16. Note that for a 4-KB cache, 256-byte blocks have the highest miss rate than 32-byte blocks. In this example, the cache would have to be 256 KB in order for a 256-byte block to decrease misses.

EXAMPLE Figure 5.17 shows the actual miss rates plotted in Figure 5.16. Assume the memory system takes 80 clock cycles of overhead and then delivers 16 bytes every 2 clock cycles. Thus, it can supply 16 bytes in 82 clock cycles, 32 bytes in 84 clock cycles, and so on. Which block size has the smallest average memory access time for each cache size in Figure 5.17?

ANSWER Average memory access time is

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

If we assume the hit time is one clock cycle independent of block size, then the access time for a 16-byte block in a 1-KB cache is

$$\text{Average memory access time} = 1 + (15.05\% \times 82) = 13.341 \text{ clock cycles}$$

and for a 256-byte block in a 256-KB cache the average memory access time is

$$\text{Average memory access time} = 1 + (0.49\% \times 112) = 1.549 \text{ clock cycles}$$

Figure 5.18 shows the average memory access time for all block and cache sizes between those two extremes. The boldfaced entries show the fastest block size for a given cache size: 32 bytes for 1-KB and 4-KB, and 64 bytes for the larger caches. These sizes are, in fact, popular block sizes for processor caches today.

n

As in all of these techniques, the cache designer is trying to minimize both the miss rate and the miss penalty. The selection of block size depends on both the latency and bandwidth of the lower-level memory. High latency and high band-

		Cache size				
Block size	**Miss penalty**	**1K**	**4K**	**16K**	**64K**	**256K**
16	82	13.341	8.027	4.231	2.673	1.894
32	84	**12.206**	**7.082**	3.411	2.134	1.588
64	88	13.109	7.160	**3.323**	**1.933**	**1.449**
128	96	16.974	8.469	3.659	1.979	1.470
256	112	25.651	11.651	4.685	2.288	1.549

FIGURE 5.18 Average memory access time versus block size for five different-sized caches in Figure 5.16. Block sizes of 32 and 64 byte dominate. The smallest average time per cache size is boldfaced.

width encourage large block size since the cache gets many more bytes per miss for a small increase in miss penalty. Conversely, low latency and low bandwidth encourage smaller block sizes since there is little time saved from a larger block. For example, twice the miss penalty of a small block may be close to the penalty of a block twice the size. The larger number of small blocks may also reduce conflict misses. Note that Figures 5.16 and 5.18 above show the difference between selecting a block size based on minimizing miss rate versus minimizing average memory access time.

After seeing the positive and negative impact of larger block size on compulsory and capacity misses, the next two subsections look at the potential of higher capacity and higher associativity.

Second Miss Rate Reduction Technique: Larger caches

The obvious way to reduce capacity misses in Figures 5.14 and 5.15 above is to increases capacity of the cache. The obvious drawback is longer hit time and higher cost. This technique has been especially popular in off-chip caches: The size of second or third level caches in 2001 equals the size of main memory in desktop computers from the first edition of this book, only a decade before!

Third Miss Rate Reduction Technique: Higher Associativity

Figures 5.14 and 5.15 above show how miss rates improve with higher associativity. There are two general rules of thumb that can be gleaned from these figures. The first is that eight-way set associative is for practical purposes as effective in reducing misses for these sized caches as fully associative. You can see the difference by comparing the 8-way entries to the capacity miss column in Figure 5.14, since capacity misses are calculated using fully associative cache. The second observation, called the *2:1 cache rule of thumb* and found on the front inside cover, is that

a direct-mapped cache of size N has about the same miss rate as a 2-way set-associative cache of size $N/2$. This held for cache sizes less than 128 KB.

Like many of these examples, improving one aspect of the average memory access time comes at the expense of another. Increasing block size reduced miss rate while increasing miss penalty, and greater associativity can come at the cost of increased hit time (see Figure 5.24 on page 431 in section 5.7.)Hence, the pressure of a fast processor clock cycle encourages simple cache designs, but the increasing miss penalty rewards associativity, as the following example suggests.

EXAMPLE Assume higher associativity would increase the clock cycle time as listed below:

$$\text{Clock cycle time}_{\text{2-way}} = 1.36 \times \text{Clock cycle time}_{\text{1-way}}$$
$$\text{Clock cycle time}_{\text{4-way}} = 1.44 \times \text{Clock cycle time}_{\text{1-way}}$$
$$\text{Clock cycle time}_{\text{8-way}} = 1.52 \times \text{Clock cycle time}_{\text{1-way}}$$

Assume that the hit time is 1 clock cycle, that the miss penalty for the direct-mapped case is 25 clock cycles to an L2 cache that never misses, and that the miss penalty need not be rounded to an integral number of clock cycles. Using Figure 5.14 for miss rates, for which cache sizes are each of these three statements true?

$$\text{Average memory access time}_{\text{8-way}} < \text{Average memory access time}_{\text{4-way}}$$
$$\text{Average memory access time}_{\text{4-way}} < \text{Average memory access time}_{\text{2-way}}$$
$$\text{Average memory access time}_{\text{2-way}} < \text{Average memory access time}_{\text{1-way}}$$

ANSWER Average memory access time for each associativity is

$$\text{Average memory access time}_{\text{8-way}} = \text{Hit time}_{\text{8-way}} + \text{Miss rate}_{\text{8-way}} \times \text{Miss penalty}_{\text{8-way}} = 1.52 + \text{Miss rate}_{\text{8-way}} \times 25$$
$$\text{Average memory access time}_{\text{4-way}} = 1.44 + \text{Miss rate}_{\text{4-way}} \times 25$$
$$\text{Average memory access time}_{\text{2-way}} = 1.36 + \text{Miss rate}_{\text{2-way}} \times 25$$
$$\text{Average memory access time}_{\text{1-way}} = 1.00 + \text{Miss rate}_{\text{1-way}} \times 25$$

The miss penalty is the same time in each case, so we leave it as 25 clock cycles. For example, the average memory access time for a 4-KB direct-mapped cache is

$$\text{Average memory access time}_{\text{1-way}} = 1.00 + (0.133 \times 25) = 3.44$$

and the time for a 512-KB, eight-way set-associative cache is

$$\text{Average memory access time}_{\text{8-way}} = 1.52 + (0.006 \times 25) = 1.66$$

Using these formulas and the miss rates from Figure 5.14, Figure 5.19 shows the average memory access time for each cache and associativity. The figure shows that the formulas in this example hold for caches less

than or equal to 8 KB for up to 4-way associativity. Starting with 16 KB, the greater hit time of larger associativity outweighs the time saved due to the reduction in misses.

	Associativity			
Cache size (KB)	One-way	Two-way	Four-way	Eight-way
4	3.44	3.25	3.22	**3.28**
8	2.69	2.58	2.55	**2.62**
16	2.23	**2.40**	**2.46**	**2.53**
32	2.06	**2.30**	**2.37**	**2.45**
64	1.92	**2.14**	**2.18**	**2.25**
128	1.52	**1.84**	**1.92**	**2.00**
256	1.32	**1.66**	**1.74**	**1.82**
512	1.20	**1.55**	**1.59**	**1.66**

FIGURE 5.19 Average memory access time using miss rates in Figure 5.14 for parameters in the example. Boldface type means that this time is higher than the number to the left; that is, higher associativity *increases* average memory access time.

Note that we did not account for the slower clock rate on the rest of the program in this example, thereby understating the advantage of direct-mapped cache.

n

Fourth Miss Rate Reduction Technique: Way Prediction and Pseudo-Associative Caches and

Another approach reduces conflict misses and yet maintains the hit speed of direct mapped cache. In *way-prediction*, extra bits are kept in the cache to predict the set of the *next* cache access. This prediction means the multiplexor is set early to select the desired set, and only a single tag comparison is performed that clock cycle. A miss results in checking the other sets for matches in subsequent clock cycles.

The Alpha 21264 uses way prediction in its instruction cache. (Added to each block of the instruction cache is a set predictor bit. The bit is used to select which of the two sets to try on the *next* cache access. If the predictor is correct, the instruction cache latency is one clock cycle. If not, it tries the other set, changes the set predictor, and has a latency of three clock cycles. (The latency of the 21264 data cache, which is very similar to its instruction cache, is also three clock cycles.) Simulations using SPEC95 suggested set prediction accuracy is in excess of 85%, so pseudo associativity saves pipeline stages in more than 85% of the instruction fetches.

In addition to improving performance, way prediction can reduce power for embedded applications. By only supplying power to the half of the tags that are expected to be used, the MIPS R4300 series lowers power consumption with the same benefits.

A related approach is called *pseudo-associative* or *column associative*. Accesses proceed just as in the direct-mapped cache for a hit. On a miss, however, before going to the next lower level of the memory hierarchy, a second cache entry is checked to see if it matches there. A simple way is to invert the most significant bit of the index field to find the other block in the "pseudo set."

Pseudo-associative caches then have one fast and one slow hit time—corresponding to a regular hit and a pseudo hit—in addition to the miss penalty. Figure 5.20 shows the relative times. One danger would be if many fast hit times of the direct-mapped cache became slow hit times in the pseudo-associative cache. The performance would then be *degraded* by this optimization. Hence, it is important to indicate for each set which block should be the fast hit and which should be the slow one. One way is simply to make the upper one fast and swap the contents of the blocks. Another danger is that the miss penalty may become slightly longer, adding the time to check another cache entry.

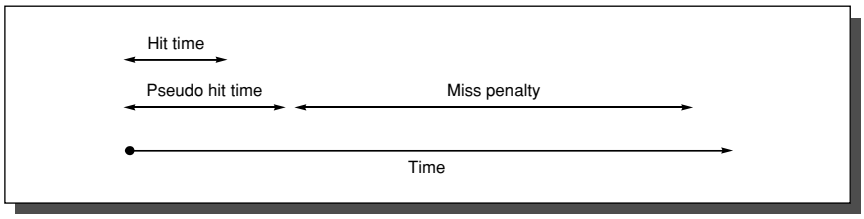


FIGURE 5.20 Relationship between a regular hit time, pseudo hit time, and miss penalty. Basically, pseudoassociativity offers a normal hit and a slow hit rather than more misses.

Fifth Miss Rate Reduction Technique: Compiler Optimizations

Thus far our techniques to reduce misses have required changes to or additions to the hardware: larger blocks, larger caches, higher associativity, or pseudo-associativity. This final technique reduces miss rates without any hardware changes.

This magical reduction comes from optimized software—the hardware designer's favorite solution! The increasing performance gap between processors and main memory has inspired compiler writers to scrutinize the memory hierarchy to see if compile time optimizations can improve performance. Once again research is split between improvements in instruction misses and improvements in data misses.

Code can easily be rearranged without affecting correctness; for example, reordering the procedures of a program might reduce instruction miss rates by re-

ducing conflict misses. McFarling [1989] looked at using profiling information to determine likely conflicts between groups of instructions. Reordering the instructions reduced misses by 50% for a 2-KB direct-mapped instruction cache with 4-byte blocks, and by 75% in an 8-KB cache. McFarling got the best performance when it was possible to prevent some instructions from ever entering the cache. Even without that feature, optimized programs on a direct-mapped cache missed less than unoptimized programs on an eight-way set-associative cache of the same size.

Another code optimization aims for better efficiency from long cache blocks. Aligning basic blocks so that the entry point is at the beginning of a cache block decreases the chance of a cache miss for sequential code.

Data have even fewer restrictions on location than code. The goal of such transformations is to try to improve the spatial and temporal locality of the data. For example, array calculations can be changed to operate on all the data in a cache block rather than blindly striding through arrays in the order the programmer happened to place the loop.

To give a feeling of this type of optimization, we will show two examples, transforming the C code by hand to reduce cache misses.

Loop Interchange

Some programs have nested loops that access data in memory in nonsequential order. Simply exchanging the nesting of the loops can make the code access the data in the order it is stored. Assuming the arrays do not fit in cache, this technique reduces misses by improving spatial locality; reordering maximizes use of data in a cache block before it is discarded.

```
/* Before */
for (j = 0; j < 100; j = j+1)
       for (i = 0; i < 5000; i = i+1)
              x[i][j] = 2 * x[i][j];

/* After */
for (i = 0; i < 5000; i = i+1)
       for (j = 0; j < 100; j = j+1)
              x[i][j] = 2 * x[i][j];
```

The original code would skip through memory in strides of 100 words, while the revised version accesses all the words in one cache block before going to the next block. This optimization improves cache performance without affecting the number of instructions executed, unlike the prior example.

Blocking

This optimization tries to reduce misses via improved temporal locality. We are again dealing with multiple arrays, with some arrays accessed by rows and some by columns. Storing the arrays row by row (*row major order*) or column by column (*column major order*) does not solve the problem because both rows and columns are used in every iteration of the loop. Such orthogonal accesses mean the transformations such as loop interchange are not helpful.

Instead of operating on entire rows or columns of an array, blocked algorithms operate on submatrices or *blocks*. The goal is to maximize accesses to the data loaded into the cache before the data are replaced. The code example below, which performs matrix multiplication, helps motivate the optimization:

```
/* Before */
for (i = 0; i < N; i = i+1)
       for (j = 0; j < N; j = j+1)
              {r = 0;
               for (k = 0; k < N; k  = k + 1)
                     r = r + y[i][k]*z[k][j];
               x[i][j] = r;
              };
```

The two inner loops read all N by N elements of z, read the same N elements in a row of y repeatedly, and write one row of N elements of x. Figure 5.21 gives a

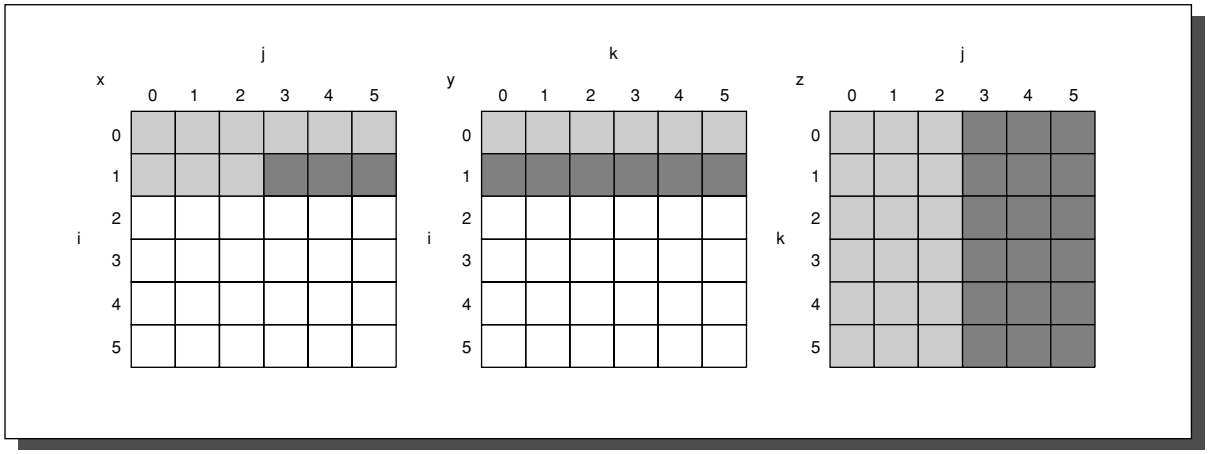


FIGURE 5.21 A snapshot of the three arrays x, y, and z when i = 1. The age of accesses to the array elements is indicated by shade: white means not yet touched, light means older accesses and dark means newer accesses. Compared to Figure 5.22, elements of y and z are read repeatedly to calculate new elements of x. The variables i, j, and k are shown along the rows or columns used to access the arrays.

snapshot of the accesses to the three arrays. A dark shade indicates a recent access, a light shade indicates an older access, and white means not yet accessed.

The number of capacity misses clearly depends on N and the size of the cache. If it can hold all three N by N matrices, then all is well, provided there are no cache conflicts. If the cache can hold one N by N matrix and one row of N, then at least the i-th row of y and the array z may stay in the cache. Less than that and misses may occur for both x and z. In the worst case, there would be $2N^3 + N^2$ memory words accessed for N^3 operations.

To ensure that the elements being accessed can fit in the cache, the original code is changed to compute on a submatrix of size B by B. Two inner loops now compute in steps of size B rather than the full length of x and z. B is called the *blocking factor*. (Assume x is initialized to zero.)

```
/* After */
for (jj = 0; jj < N; jj = jj+B)
for (kk = 0; kk < N; kk = kk+B)
for (i = 0; i < N; i = i+1)
        for (j = jj; j < min(jj+B,N); j = j+1)
            {r = 0;
             for (k = kk; k < min(kk+B,N); k = k + 1)
                    r = r + y[i][k]*z[k][j];
             x[i][j] = x[i][j] + r;
            };
```

Figure 5.22 illustrates the accesses to the three arrays using blocking. Looking only at capacity misses, the total number of memory words accessed is $2N^3/B + N^2$. This total is an improvement by about a factor of B. Hence, blocking exploits a combination of spatial and temporal locality, since y benefits from spatial locality and z benefits from temporal locality.

Although we have aimed at reducing cache misses, blocking can also be used to help register allocation. By taking a small blocking size such that the block can be held in registers, we can minimize the number of loads and stores in the program.

Summary of Reducing Cache Miss Rate

This section first presented the three C's model of cache misses: compulsory, capacity, and conflict. This intuitive model led to three obvious optimizations: larger block size to reduce compulsory misses, larger cache size to reduce capacity misses, and higher associativity to reduce conflict misses. Since higher associativity may affect cache hit time or cache power consumption, way prediction checks only a piece of the cache for hits and then on a miss checks the rest. The final technique is the favorite of the hardware designer, leaving cache optimizations to the compiler.

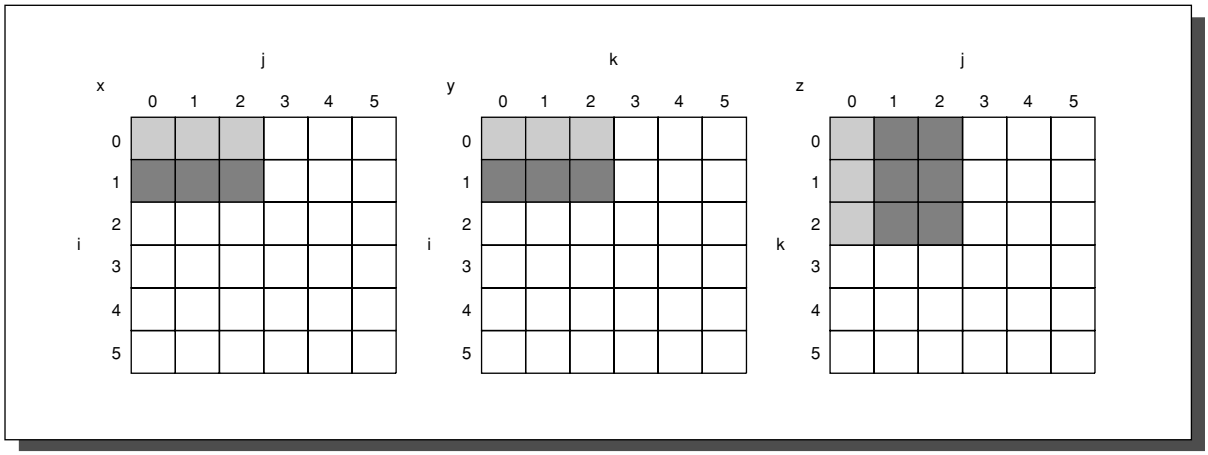


FIGURE 5.22 The age of accesses to the arrays x, y, and z. Note in contrast to Figure 5.21 the smaller number of elements accessed.

The increasing processor-memory gap has meant that cache misses are a primary cause of lower than expected performance. As a result, both algorithms and compilers are changing from the traditional focus of reducing operations to reducing cache misses.

The next section increases performance by having the processor and memory hierarchy operate in parallel, with compilers again playing a significant role in orchestrating this parallelism.

5.6 Reducing Cache Miss Penalty or Miss Rate via Parallelism

This section describes three techniques that overlap the execution of instructions with activity in the memory hierarchy. The first creates a memory hierarchy to match the out-of-order processors, but the second and third work with any type of processor. Although popular in desktop and server computers, the emphasis on efficiency in power and silicon area of embedded computers means such techniques are only found in embedded computers if they are small and reduce power.

First Miss Penalty/Rate Reduction Technique: Nonblocking Caches to Reduce Stalls on Cache Misses

For pipelined computers that allow out-of-order completion (Chapter 3), the CPU need not stall on a cache miss. For example, the CPU could continue fetching in-

structions from the instruction cache while waiting for the data cache to return the missing data. A *nonblocking cache* or *lockup-free cache* escalates the potential benefits of such a scheme by allowing the data cache to continue to supply cache hits during a miss. This "hit under miss" optimization reduces the effective miss penalty by being helpful during a miss instead of ignoring the requests of the CPU. A subtle and complex option is that the cache may further lower the effective miss penalty if it can overlap multiple misses: a "hit under multiple miss" or "miss under miss" optimization. The second option is beneficial only if the memory system can service multiple misses (see page 441). Be aware that hit under miss significantly increases the complexity of the cache controller as there can be multiple outstanding memory accesses.

Figure 5.23 shows the average time in clock cycles for cache misses for an 8-KB data cache as the number of outstanding misses is varied. Floating-point programs benefit from increasing complexity, while integer programs get almost all of the benefit from a simple hit-under-one-miss scheme. Following the discussion in Chapter 3, the number of simultaneous outstanding misses limits achievable instruction level parallelism in programs.

EXAMPLE For the cache described in Figure 5.23, which is more important for floating-point programs: two-way set associativity or hit under one miss? What about integer programs? Assume the following average miss rates for 8-KB data caches: 11.4% for floating-point programs with a direct-mapped cache, 10.7% for these programs with a two-way set-associative cache, 7.4% for integer programs with a direct-mapped cache, and 6.0% for integer programs with a two-way set-associative cache. Assume the average memory stall time is just the product of the miss rate and the miss penalty.

ANSWER The numbers for Figure 5.23 were based on a miss penalty of 16 clock cycles assuming an L2 cache. Although this is low for a miss penalty (we'll see how in the next subsection), let's stick with it for consistency. For floating-point programs the average memory stall times are

$$\text{Miss rate}_{\text{DM}} \times \text{Miss penalty} = 11.4\% \times 16 = 1.84$$

$$\text{Miss rate}_{\text{2-way}} \times \text{Miss penalty} = 10.7\% \times 16 = 1.71$$

The memory stalls of two-way are thus 1.71/1.84 or 93% of direct-mapped cache. The caption of Figure 5.23 says hit under one miss reduces the average memory stall time to 76% of a blocking cache. Hence, for floating-point programs, the direct-mapped data cache supporting hit under one miss gives better performance than a two-way set-associative cache that blocks on a miss.

For integer programs the calculation is

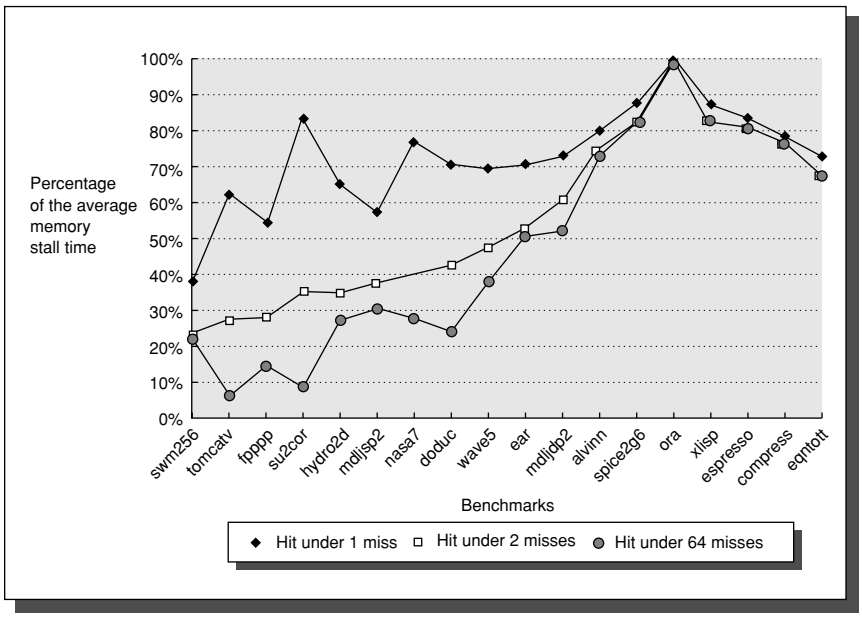


FIGURE 5.23 Ratio of the average memory stall time for a blocking cache to hit-under-miss schemes as the number of outstanding misses is varied for 18 SPEC92 programs. The hit-under-64-misses line allows one miss for every register in the processor. The first 14 programs are floating-point programs: the average for hit under 1 miss is 76%, for 2 misses is 51%, and for 64 misses is 39%. The final four are integer programs, and the three averages are 81%, 78%, and 78%, respectively. These data were collected for an 8-KB direct-mapped data cache with 32-byte blocks and a 16-clock-cycle miss penalty, which today would imply a second level cache. These data were generated using the VLIW Multiflow Compiler, which scheduled loads away from use [Farkas and Jouppi 1994].

$$\text{Miss rate}_{\text{DM}} \times \text{Miss penalty} = 7.4\% \times 16 = 1.18$$

$$\text{Miss rate}_{\text{2-way}} \times \text{Miss penalty} = 6.0\% \times 16 = 0.96$$

The memory stalls of two-way are thus 0.96/1.18 or 81% of direct-mapped cache. The caption of Figure 5.23 says hit under one miss reduces the average memory stall time to 81% of a blocking cache, so the two options give about the same performance for integer programs. One advantage of hit under miss is that it cannot affect the hit time, as associativity can. n

The real difficulty with performance evaluation of nonblocking caches is that that they imply a dynamic-issue CPU. As a cache miss does not necessarily stall the CPU as it would a static issue CPU. As mentioned on page 395, it is difficult to judge the impact of any single miss, and hence difficult to calculate the average

memory access time. As was the case of a second miss to the remaining block with critical word first above, the effective miss penalty is not the sum of the misses but the non-overlapped time that the CPU is stalled. In general, out-of-order CPUs are capable of hiding the miss penalty of an L1 data cache miss which hits in the L2 cache, but not capable of hiding a significant fraction of an L2 cache miss. Changing the program to pipeline L2 misses can help, especially to a banked memory system (see section 5.8).

Chapter 1 discusses the pros and cons of execution-driven simulation versus trace-driven simulation. Cache studies involving an out-of-order CPUs use execution-driven simulation to evaluate innovations, as avoiding a cache miss that is completely hidden by dynamic issue does not help performance.

An added complexity of multiple outstanding misses is that it is now possible for there to be more than one miss request to the same block. For example, with 64 byte blocks there could be a miss to address 1000 and then later a miss to address 1032. Thus, the hardware must check on misses to be sure it is not to block already being requested to avoid possible incoherency problems and to save time.

Second Miss Penalty/Rate Reduction Technique: Hardware Prefetching of Instructions and Data

Nonblocking caches effectively reduce the miss penalty by overlapping execution with memory access. To have value, we need a processor that can allow instructions to execute out-of-order. Another approach is to prefetch items before they are requested by the processor. Both instructions and data can be prefetched, either directly into the caches or into an external buffer that can be more quickly accessed than main memory.

Instruction prefetch is frequently done in hardware outside of the cache. Typically, the processor fetches two blocks on a miss: the requested block and the next consecutive block. The requested block is placed in the instruction cache when it returns, and the prefetched block is placed into the instruction stream buffer. If the requested block is present in the instruction stream buffer, the original cache request is canceled, the block is read from the stream buffer, and the next prefetch request is issued.

Jouppi [1990] found that a single instruction stream buffer would catch 15% to 25% of the misses from a 4-KB direct-mapped instruction cache with 16-byte blocks. With 4 blocks in the instruction stream buffer the hit rate improves to about 50%, and with 16 blocks to 72%.

A similar approach can be applied to data accesses. Jouppi found that a single data stream buffer caught about 25% of the misses from the 4-KB direct-mapped cache. Instead of having a single stream, there could be multiple stream buffers beyond the data cache, each prefetching at different addresses. Jouppi found that four data stream buffers increased the data hit rate to 43%. Palacharla and Kessler [1994] looked at a set of scientific programs and considered stream buffers that could handle either instructions or data. They found that eight stream buffers could capture 50% to 70% of all misses from a processor with two 64-KB four-way set-associative caches, one for instructions and the other for data.

The UltraSPARC III uses such a prefetch scheme. A prefetch cache remembers the address used to prefetch the data. If a load hits in prefetch cache, the block is read from the prefetch cache, and the next prefetch request is issued. It calculates the "stride" of the next prefetched block using the difference between current address and the previous address. There can be up to eight simultaneous prefetches in UltraSPARC III.

EXAMPLE What is the effective miss rate of the UltraSPARC III using instruction prefetching? How much bigger an data cache would be needed in the UltraSPARC III to match the average access time if prefetching were removed? It has an 64-KB data cache. Assume prefetching reduces data miss rate by 20%.

ANSWER We assume it takes 1 extra clock cycle if the data misses the cache but is found in the prefetch buffer. Here is our revised formula:

$$\text{Average memory access time}_{\text{prefetch}} = \text{Hit time} + \text{Miss rate} \times \text{Prefetch hit rate} \times 1 + \text{Miss rate} \times (1 - \text{Prefetch hit rate}) \times \text{Miss penalty}$$

Let's assume the prefetch hit rate is 50%. Figure 5.8 on page 390 gives the misses per 1000 instructions for an 64-KB data cache as 36.9. To convert to a miss rate, is we assume 22% data references, the rate is

$$\frac{36.9}{1000 \times 22/100} = \frac{36.9}{220}$$ or 16.7%. Assume the he hit time is 1 clock cycles, and the miss penalty is 15 clock cycles since UltraSPARC III has an L2 cache:

$$\text{Average memory access time}_{\text{prefetch}} = 1 + (16.7\% \times 20\% \times 1) + (16.7\% \times (1 - 20\%) \times 15) = 1 + 0.034 + 2.013 = 3.046$$

To find the effective miss rate with the equivalent performance, we start with the original formula and solve for the miss rate:

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

$$\text{Miss rate} = \frac{\text{Average memory access time} - \text{Hit time}}{\text{Miss penalty}}$$

$$\text{Miss rate} = \frac{3.046 - 1}{15} = \frac{2.046}{15} = 13.6\%$$

Our calculation suggests that the effective miss rate of prefetching with an 64-KB cache is 13.6%. Figure 5.8 on page 390 gives the misses per 1000 instructions of a 256-KB instruction cache as 32.6, yielding a miss rate of 32.6/(22% x1000) or 14.8%,. If the prefetching reduces miss rate by 20%, then a 64 KB data cache with prefetching outperforms a 256-KB cache without it. n

Prefetching relies on utilizing memory bandwidth that otherwise would be unused, but if it interferes with demand misses it can actually lower performance. Help from compilers can reduce useless prefetching.

Third Miss Penalty/Rate Reduction Technique: Compiler-Controlled Prefetching

An alternative to hardware prefetching is for the compiler to insert prefetch instructions to request the data before they are needed. There are several flavors of prefetch:

- *Register prefetch* will load the value into a register.
- *Cache prefetch* loads data only into the cache and not the register.

Either of these can be *faulting* or *nonfaulting*; that is, the address does or does not cause an exception for virtual address faults and protection violations. Nonfaulting prefetches simply turn into no-ops if they would normally result in an exception. Using this terminology, a normal load instruction could be considered a "faulting register prefetch instruction."

The most effective prefetch is "semantically invisible" to a program: it doesn't change the contents of registers and memory *and* it cannot cause virtual memory faults. Most processors today offer non-faulting cache prefetches. This section assumes nonfaulting cache prefetch, also called *nonbinding* prefetch.

Prefetching makes sense only if the processor can proceed while the prefetched data are being fetched; that is, the caches do not stall but continue to supply instructions and data while waiting for the prefetched data to return. As you would expect, the data cache for such computers is normally nonblocking.

Like hardware-controlled prefetching, the goal is to overlap execution with the prefetching of data. Loops are the important targets, as they lend themselves to prefetch optimizations. If the miss penalty is small, the compiler just unrolls the loop once or twice and it schedules the prefetches with the execution. If the miss penalty is large, it uses software pipelining (page 290 in Chapter 4) or unrolls many times to prefetch data for a future iteration.

Issuing prefetch instructions incurs an instruction overhead, however, so care must be taken to ensure that such overheads do not exceed the benefits. By concentrating on references that are likely to be cache misses, programs can avoid unnecessary prefetches while improving average memory access time significantly.

EXAMPLE For the code below, determine which accesses are likely to cause data cache misses. Next, insert prefetch instructions to reduce misses. Finally, calculate the number of prefetch instructions executed and the misses avoided by prefetching. Let's assume we have an 8-KB direct-mapped

data cache with 16-byte blocks, and it is a write-back cache that does write allocate. The elements of `a` and `b` are 8 bytes long as they are double-precision floating-point arrays. There are 3 rows and 100 columns for `a` and 101 rows and 3 columns for `b`. Let's also assume they are not in the cache at the start of the program.

```
for (i = 0; i < 3; i = i+1)
        for (j = 0; j < 100; j = j+1)
                a[i][j] = b[j][0] * b[j+1][0];
```

ANSWER The compiler will first determine which accesses are likely to cause cache misses; otherwise, we will waste time on issuing prefetch instructions for data that would be hits. Elements of `a` are written in the order that they are stored in memory, so `a` will benefit from spatial locality: the even values of `j` will miss and the odd values will hit. Since `a` has 3 rows and 100 columns, its accesses will lead to $3 \times \left\lceil \frac{100}{2} \right\rceil$ or 150 misses.

The array `b` does not benefit from spatial locality since the accesses are not in the order it is stored. The array `b` does benefit twice from temporal locality: the same elements are accessed for each iteration of `i`, and each iteration of `j` uses the same value of `b` as the last iteration. Ignoring potential conflict misses, the misses due to `b` will be for `b[j+1][0]` accesses when `i` = 0, and also the first access to `b[j][0]` when `j` = 0. Since `j` goes from 0 to 99 when `i` = 0, accesses to `b` lead to 100 + 1 or 101 misses.

Thus, this loop will miss the data cache approximately 150 times `a` for plus a 101 times for `b`, or 251 misses.

To simplify our optimization, we will not worry about prefetching the first accesses of the loop. These may be already in the cache, or we will pay the miss penalty of first few elements of a or b. Nor we will worry about suppressing the prefetches at the end of the loop which try to prefetch beyond the end of `a` (`a[i][100]`...`a[i][106]`) and end of `b` (`b[100][0]`... `b[106][0]`). If these were faulting prefetches, we could not take this luxury. Let's assume that the miss penalty is so large we prefetch need to start at least, say, seven iterations in advance. (Stated alternatively, we assume prefetching has no benefit until the eighth iteration.)

```
for (j = 0; j < 100; j = j+1) {
        prefetch(b[j+7][0]);
        /* b(j,0) for 7 iterations later */
        prefetch(a[0][j+7]);
        /* a(0,j) for 7 iterations later */
```

```
        a[0][j] = b[j][0] * b[j+1][0];};
for (i = 1; i < 3; i = i+1)
        for (j = 0; j < 100; j = j+1) {
                prefetch(a[i][j+7]);
                /* a(i,j) for +7 iterations */
                a[i][j] = b[j][0] *b[j+1][0];}
```

This revised code prefetches `a[i][7]` through `a[i][99]` and `b[7][0]` through `b[99][0]`, reducing the number of nonprefetched misses to:

- 7 misses for elements b[0][0], b[1][0], ..., b[6][0] in the first loop;
- 4 misses ($\lceil 7/2 \rceil$) for elements a[0][0], a[0][1], ..., a[0][6] for in the first loop (spatial locality reduces misses to one per 16 byte cache block);
- 4 misses ($\lceil 7/2 \rceil$) for elements a[1][0], a[1][1], ..., a[1][6] in the second loop;
- 4 misses ($\lceil 7/2 \rceil$) for elements a[2][0], a[2][1], ..., a[2][6] in the second loop;

or a total of 19 nonprefetched misses. The cost of avoiding 232 cache misses is executing 400 prefetch instructions, likely a good trade-off. ▫

EXAMPLE Calculate the time saved in the example above. Ignore instruction cache misses and assume there are no conflict or capacity misses in the data cache. Assume that prefetches can overlap with each other and with cache misses, thereby transferring at the maximum memory bandwidth. Here are the key loop times ignoring cache misses: the original loop takes 7 clock cycles per iteration, the first prefetch loop takes 9 clock cycles per iteration, and the second prefetch loop takes 8 clock cycles per iteration (including the overhead of the outer for loop). A miss takes 100 clock cycles.

ANSWER The original doubly nested loop executes the multiply 3×100 or 300 times. Since the loop takes 7 clock cycles per iteration, the total is 300×7 or 2100 clock cycles plus cache misses. Cache misses add 251×100 or 25,100 clock cycles, giving a total of 27,200 clock cycles. The first prefetch loop iterates 100 times; at 9 clock cycles per iteration the total is 900 clock cycles plus cache misses. They add 11×100 or 1100 clock cycles for cache misses, giving a total of 2000. The second loop executes 2×100 or 200 times, and at 8 clock cycles per iteration it takes 1600 clock cycles plus 8×100 or 800 clock cycles for cache misses. This gives a total of 2400 clock cycles. From the prior example we know that this code executes 400 prefetch instructions during the 2000 + 2400 or

4400 clock cycles to execute these two loops. If we assume that the prefetches are completely overlapped with the rest of the execution, then the prefetch code is 27,200/4400 or 6.2 times faster.

In addition to the nonfaulting prefetch loads, the 21264 offers prefetches to help with writes. In the example above, we prefetched `a[i][j+7]` even though we were not going to read the data. We just wanted it in the cache so that we could write over it. If an aligned cache block is being written in full, the write hint instruction tells the cache to allocate the block but do not bother loading the data, as the CPU will write over it. In the example above, if the array `a` were properly aligned and padded, such an instruction could replace the instructions prefetching `a` with the write hint instructions, thereby saving hundreds of memory accesses. Since these write hints do have side effects, care would also have to be taken not to access memory outside of the memory allocated for `a`.

Although array optimizations are easy to understand, modern programs are more likely to use pointers. Luk and Mowry [1999] have demonstrated that compiler-based prefetching can sometimes be extended to pointers as well. Of ten programs with recursive data structures, prefetching all pointers when a node is visited improved performance by 4% to 31% in half the programs. On the other hand, the remaining programs were still within 2% of their original performance. The issue is both whether prefetches are to data already in the cache and whether they occur early enough for the data to arrive by the time it is needed.

Summary of Reducing Cache Miss Penalty/ Miss Rate via Parallelism

This section first covered non-blocking caches, which enable out-of-order processors. In general such processors cache hide misses to L1 caches that hit in the L2 cache, but not a complete L2 cache miss. However, if miss under miss is supported, nonblocking caches can take advantage of more bandwidth behind the cache by having several outstanding misses operating at once for programs with sufficient instruction level parallelism.

The hardware and software prefetching techniques leverage excess memory bandwidth for performance by trying to anticipate the needs of a cache. Although speculation may not make sense for power sensitive embedded applications, it normally does for desktop and server computers. The potential success of prefetching is either lower miss penalty, or if they are started far in advance of need, reduction of the miss rate. This ambiguity of whether they help miss rate or miss penalty is one reason they are included in separate section.

Now that we have spent nearly 30 pages on techniques that reduce cache misses or miss penalty in sections 5.4 to 5.6, it is time to look at reducing the final component of average memory access time.

5.7 Reducing Hit Time

Hit time is critical because it affects the clock rate of the processor; in many processors today the cache access time limits the clock cycle rate, even for processors that take multiple clock cycles to access the cache. Hence, a fast hit time is multiplied in importance beyond the average memory access time formula because it helps everything. This section gives four general techniques.

First Hit Time Reduction Technique: Small and Simple Caches

A time-consuming portion of a cache hit is using the index portion of the address to read the tag memory and then compare it to the address. Our guideline from Chapter 1 suggests that smaller hardware is faster, and a small cache certainly helps the hit time. It is also critical to keep the cache small enough to fit on the same chip as the processor to avoid the time penalty of going off-chip. The second suggestion is to keep the cache simple, such as using direct mapping (see page 414). A main benefit of direct-mapped caches is that the designer can overlap the tag check with the transmission of the data. This effectively reduces hit time. Hence, the pressure of a fast clock cycle encourages small and simple cache designs for first-level caches. For second level caches, some designs strike a compromise by keeping the tags on-chip and the data off-chip, promising a fast tag check, yet providing the greater capacity of separate memory chips.

One approach to determining the impact on hit time in advance of building a chip is to use CAD tools. CACTI is a program to estimate the access time of alternative cache structures on CMOS microprocessors within 10% of more detailed CAD tools. For a given minimum feature size, it estimates the hit time of caches as you vary cache size, associativity, and number of read/write ports. Figure 5.24 shows the estimated impact on hit time as cache size and associativity are varied. Depending on cache size, for these parameters the model suggests that hit times for direct mapped is 1.2 to 1.5 times faster than 2-way set associative; 2-way is l.02 to 1.11 times faster than 4-way; and 4-way is 1.0 to 1.08 times faster than fully associative (except for a 256 KB cache, which is 1.19 times faster).

Although the amount of on-chip cache increased with new generations of microprocessors, the size of the L1 caches has recently not increased between generations. The L1 caches are the same size between the Alpha 21264 and 21364, UltraSPARC II and III, and AMD K6 and Athlon. The L1 data cache size is actually reduced from 16 KB in Pentium III to 8 KB in Pentium 4. The emphasis recently is on fast clock time while hiding L1 misses with dynamic execution and using L2 caches to avoid going to memory.

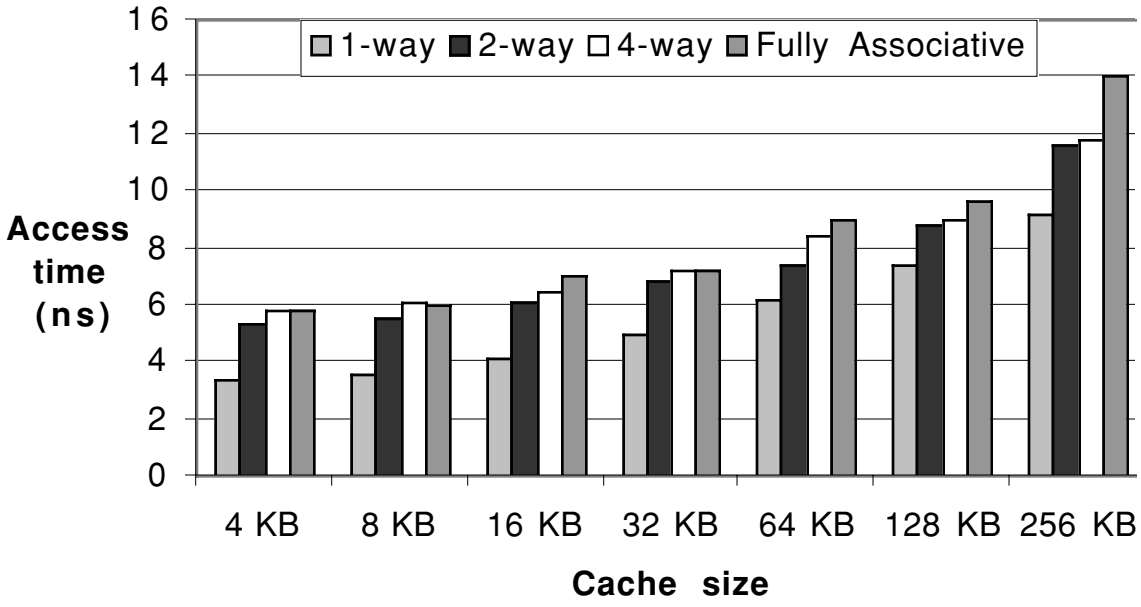


FIGURE 5.24 Access times for as size and associativity vary in a CMOS cache. These data are based on Spice runs used to validate the CACTI model 2.0 by Reinmann and Jouppi [1999]. They assumed 0.80-micron feature size, a single read/write port, 32 address bits, 64 output bits, and 32 byte blocks. The median ratios of access time relative to the direct mapped caches are 1.36, 1.44, and 1.52 for 2-way, 4-way, and 8-way associative caches, respectively.

Second Hit Time Reduction Technique: Avoiding Address Translation During Indexing of the Cache

Even a small and simple cache must cope with the translation of a virtual address from the CPU to a physical address to access memory. As described below in section 5.10, processors treat main memory as just another level of the memory hierarchy, and thus the address of the virtual memory that exists on disk must be mapped onto the main memory.

The guideline of making the common case fast suggests that we use virtual addresses for the cache, since hits are much more common than misses. Such caches are termed *virtual caches*, with *physical cache* used to identify the traditional cache that uses physical addresses. As we shall shortly see, it is important to distinguish two tasks: indexing the cache and the comparing addresses. Thus, the issues are whether a virtual or physical address is used to index the cache and whether a virtual or physical index is used in the tag comparison. Full virtual addressing for both index and tags eliminates address translation time from a cache hit. Then why doesn't everyone build virtually addressed caches?

One reason is protection. Page level protection is checked as part of the virtual to physical address translation, and it must be enforced no matter what. One solu-

tion is to copy the protection information from the TLB on a miss, add a field to hold it, and check it on every access to the virtually addressed cache.

Another reason is that every time a process is switched, the virtual addresses refer to different physical addresses, requiring the cache to be flushed. Figure 5.25 shows the impact on miss rates of this flushing. One solution is to increase the width of the cache address tag with a *process-identifier tag* (PID). If the operating system assigns these tags to processes, it only need flush the cache when a PID is recycled; that is, the PID distinguishes whether or not the data in

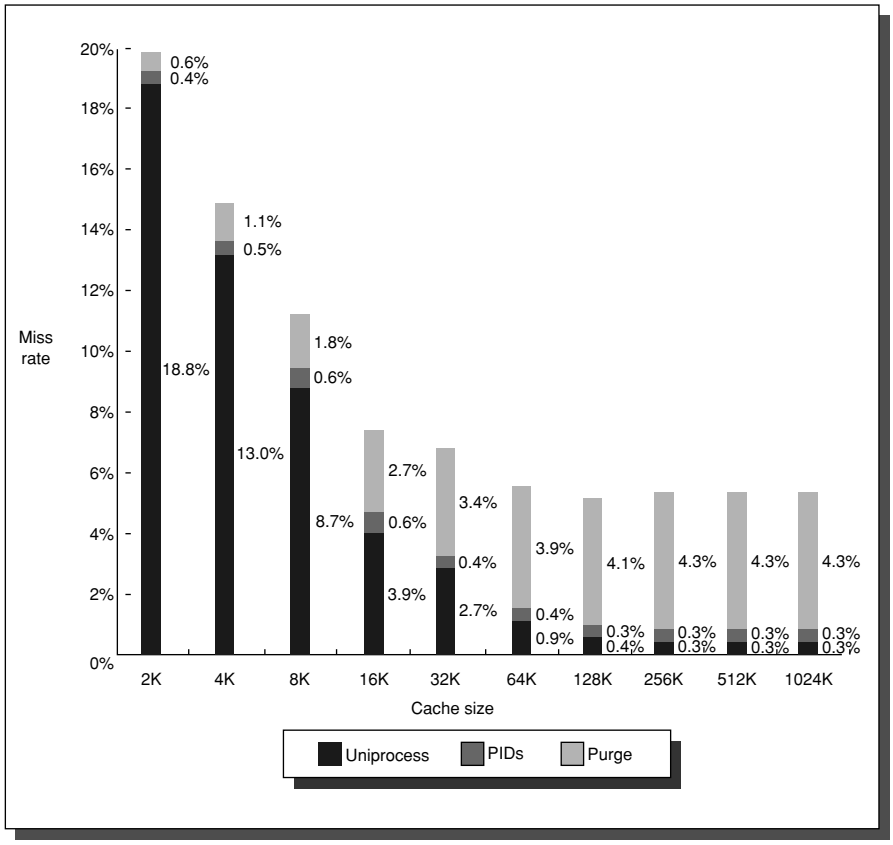


FIGURE 5.25 Miss rate versus virtually addressed cache size of a program measured three ways: without process switches (uniprocess), with process switches using a process-identifier tag (PIDs), and with process switches but without PIDs (purge). PIDs increase the uniprocess absolute miss rate by 0.3% to 0.6% and save 0.6% to 4.3% over purging. Agarwal [1987] collected these statistics for the Ultrix operating system running on a VAX, assuming direct-mapped caches with a block size of 16 bytes. Note that the miss rate goes up from 128K to 256K. Such nonintuitive behavior can occur in caches because changing size changes the mapping of memory blocks onto cache blocks, which can change the conflict miss rate.

the cache are for this program. Figure 5.25 shows the improvement in miss rates by using PIDs to avoid cache flushes.

A third reason why virtual caches are not more popular is that operating systems and user programs may use two different virtual addresses for the same physical address. These duplicate addresses, called *synonyms* or *aliases*, could result in two copies of the same data in a virtual cache; if one is modified, the other will have the wrong value. With a physical cache this wouldn't happen, since the accesses would first be translated to the same physical cache block.

Hardware solutions to the synonym problem, called *anti-aliasing*, guarantee every cache block a unique physical address. The Alpha 21264 uses a 64 KB instruction cache with an 8 KB page and two-way set associativity, hence the hardware must handle aliases involved with the 2 virtual address bits in both sets. It avoids aliases by simply checking all 8 possible locations on a miss–four entries per set–to be sure that none match the physical address of the data being fetched. If one is found, it is invalidated, so when the new data is loaded into the cache its physical address is guaranteed to be unique.

Software can make this problem much easier by forcing aliases to share some address bits. The version of UNIX from Sun Microsystems, for example, requires all aliases to be identical in the last 18 bits of their addresses; this restriction is called *page coloring*. Note that page coloring is simply set-associative mapping applied to virtual memory: the 4-KB (2^{12}) pages are mapped using 64 (2^6) sets to ensure that the physical and virtual addresses match in the last 18 bits. This restriction means a direct-mapped cache that is 2^{18} (256K) bytes or smaller can never have duplicate physical addresses for blocks. From the perspective of the cache, page coloring effectively increases the page offset, as software guarantees that the last few bits of the virtual and physical page address are identical

The final area of concern with virtual addresses is I/O. I/O typically uses physical addresses and thus would require mapping to virtual addresses to interact with a virtual cache. (The impact of I/O on caches is further discussed below in section 5.12.)

One alternative to get the best of both virtual and physical caches is to use part of the page offset—the part that is identical in both virtual and physical addresses—to index the cache. At the same time as the cache is being read using that index, the virtual part of the address is translated, and the tag match uses physical addresses.

This alternative allows the cache read to begin immediately and yet the tag comparison is still with physical addresses. The limitation of this *virtually indexed, physically tagged* alternative is that a direct-mapped cache can be no bigger than the page size. For example, in the data cache in Figure 5.7 on page 388, the index is 9 bits and the cache block offset is 6 bits. To use this trick, the virtual page size would have to be at least $2^{(9+6)}$ bytes or 32 KB. If not, a portion of the index must be translated from virtual to physical address.

Associativity can keep the index in the physical part of the address and yet still support a large cache. Recall that size of index is controlled by this formula:

$$2^{\text{index}} = \frac{\text{Cache size}}{\text{Block size} \times \text{Set associativity}}$$

For example, doubling associativity and doubling the cache size does not change the size of the index. The Pentium III, with 8-KB pages, avoids translation with its16-KB cache by using 2-way set associativity. The IBM 3033 cache, as an extreme example, is 16-way set associative, even though studies show there is little benefit to miss rates above eight-way set associativity. This high associativity allows a 64-KB cache to be addressed with a physical index, despite the handicap of 4-KB pages in the IBM architecture.

Third Hit Time Reduction Technique: Pipelined Cache Access

The final technique is simply to pipeline cache access so that the effective latency of a first level cache hit can be multiple clock cycles, giving fast cycle time and slow hits. For example, the pipeline for the Pentium takes one clock cycle to access the instruction cache, for the Pentium Pro through Pentium III it takes two clocks, and for the Pentium 4 it takes four clocks. This split increases the number of pipeline stages, leading to greater penalty on mispredicted branches and more clock cycles between the issue of the load and the use of the data (see section 3.9).

Note that this technique in reality increases the bandwidth of instructions rather that decreasing the actual latency of a cache hit.

Fourth Hit Time Reduction Technique: Trace Caches

A challenge in the effort to find instruction level parallelism beyond four instructions per cycle is to supply enough instructions every cycle without dependencies. One solution is called a *trace cache*. Instead of limiting the instructions in a static cache block to spatial locality, a trace cache finds a dynamic sequence of instructions *including taken branches* to load into a cache block.

The name comes from the cache blocks containing dynamic traces of the executed instructions as determined by the CPU rather than containing static sequences of instructions as determined by memory. Hence, the branch prediction is folded into cache, and must be validated along with the addresses to have a valid fetch. The Intel Netburst microarchitecture, which is the foundation of the Pentium 4 and its successors, uses a trace cache.

Clearly, trace caches have much more complicated address mapping mechanisms, as the addresses are no longer aligned to power of 2 multiple of the word size. However, they have other benefits for utilization of the data portion of the instruction cache. Very long blocks in conventional caches may be entered from a taken branch, and hence the first portion of the block would occupy space in the cache might not be fetched. Similarly, such blocks may be exited by taken branches, so the last portion of the block might be wasted. Given that taken branches or jumps are one in 5 to 10 instructions, space utilization is a real problem for processors like the AMD Athlon, whose 64 byte block would likely include16 to 24 80x86 instructions. The trend towards even greater instruction issue should make the problem worse. Trace caches store instructions only from the branch entry point to the exit of the trace, thereby avoiding such header and trailer overhead.

The downside of trace caches is that they store the same instructions multiple times in the instruction cache. Conditional branches making different choices result in the same instructions being part of separate traces, which each occupy space in the cache.

Cache Optimization Summary

The techniques in sections 5.4 to 5.7 to improve miss rate, miss penalty, and hit time generally impact the other components of the average memory access equation as well as the complexity of the memory hierarchy. Figure 5.26 summarizes these techniques and estimates the impact on complexity, with + meaning that the technique improves the factor, – meaning it hurts that factor, and blank meaning it has no impact. Generally no technique helps more than one category.

5.8 Main Memory and Organizations for Improving Performance

Main memory is the next level down in the hierarchy. Main memory satisfies the demands of caches and serves as the I/O interface, as it is the destination of input as well as the source for output. Performance measures of main memory emphasize both latency and bandwidth. (Memory bandwidth is the number of bytes read or written per unit time.) Traditionally, main memory latency (which affects the cache miss penalty) is the primary concern of the cache, while main memory bandwidth is the primary concern of I/O and multiprocessors. The relationship of main memory and multiprocessors is discussed in Chapter 6, and relationship of main memory and I/O is discussed in Chapter 7.

Technique	Miss penalty	Miss rate	Hit time	Hardware complexity	Comment
Multi-level caches	+			2	Costly hardware; harder if block size L1 ≠ L2; widely used
Critical word first and early restart	+			2	Widely used
Giving priority to read misses over writes	+			1	Trivial for uniprocessor, and widely used
Merging Write Buffer	+			1	Used with write through; in 21164. UltraSPARC III; widely used
Victim caches	+	+		2	AMD Athlon has 8 entries
Larger block size	–	+		0	Trivial; Pentium 4 L2 uses 128 bytes
Larger cache size		+	–	1	Widely used, esp. for L2 caches
Higher associativity		+	–	1	Widely used
Way-predicting caches		+		2	Used in I-cache of UltraSPARC III; D-cache of MIPS R4300 series
Pseudo-associative		+		2	Used in L2 of MIPS R10000
Compiler techniques to reduce cache misses		+		0	Software is challenge; some computers have compiler option
Nonblocking caches	+			3	Used with all out-of-order CPUs
Hardware prefetching of instructions and data	+	+		2 instr., 3 data	Many prefetch instructions; UltraSPARC III prefetches data
Compiler-controlled prefetching	+	+		3	Needs nonblocking cache too; several processors support it
Small and simple caches		–	+	0	Trivial; widely used
Avoiding address translation during indexing of the cache			+	2	Trivial if small cache; used in Alpha 21164, UltraSPARC III
Pipelined cache access			+	1	Widely used
Trace cache			+	3	Used in Pentium 4

FIGURE 5.26 Summary of cache optimizations showing impact on cache performance and complexity for the techniques in sections 5.4 to 5.7. Although generally a technique helps only one factor, prefetching can reduce misses if done sufficiently early; if not, it can reduce miss penalty. + means that the technique improves the factor, – means it hurts that factor, and blank means it has no impact. The complexity measure is subjective, with 0 being the easiest and 3 being a challenge.

Although caches are interested in low latency memory, it is generally easier to improve memory bandwidth with new organizations than it is to reduce latency. With the popularity of second-level caches and their larger block sizes, main memory bandwidth becomes important to caches as well. In fact, cache designers increase block size to take advantage of the high memory bandwidth.

The previous sections describe what can be done with cache organization to reduce this CPU-DRAM performance gap, but simply making caches larger or

adding more levels of caches may not be a cost-effective way to eliminate the gap. Innovative organizations of main memory are needed as well. In the this section we examine techniques for organizing memory to improve bandwidth.

Let's illustrate these organizations with the case of satisfying a cache miss. Assume the performance of the basic memory organization is

- 4 clock cycles to send the address
- 56 clock cycles for the access time per word
- 4 clock cycles to send a word of data

Given a cache block of four words, and that a word is 8 bytes, the miss penalty is 4 × (4 + 56 + 4) or 256 clock cycles, with a memory bandwidth of one-eighth byte (32/256) per clock cycle. These values are our default case.

Figure 5.27 shows some of the options to faster memory systems. The next three solutions assume generic memory. The next three solutions assume generic memory technology, which we explore in the next section.

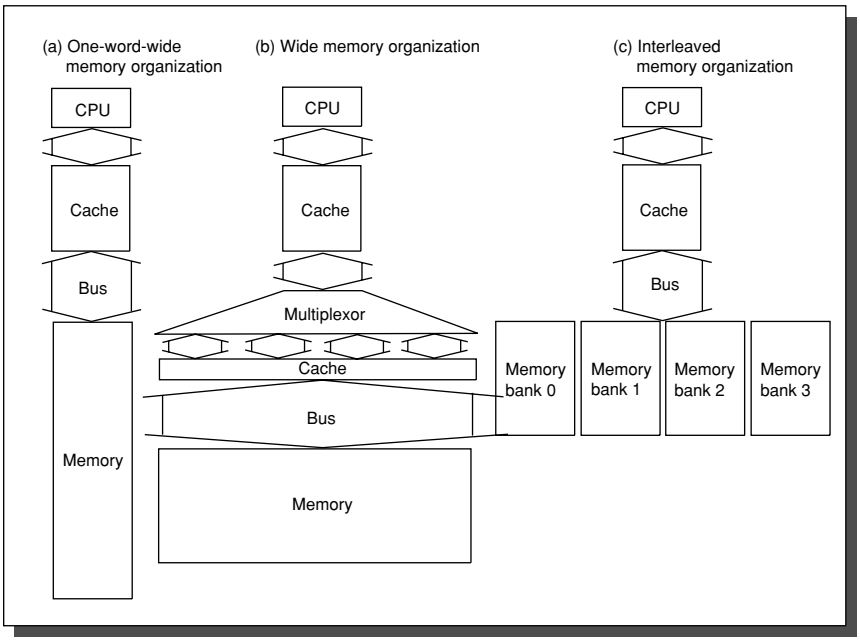


FIGURE 5.27 Three examples of bus width, memory width, and memory interleaving to achieve higher memory bandwidth. (a) is the simplest design, with everything the width of one word; (b) shows a wider memory, bus, and L2 cache with a narrow L1 cache; while (c) shows a narrow bus and cache with an interleaved memory.

The simplest approach to increasing memory bandwidth, then, is to make the memory wider; we examine this first.

First Technique for Higher Bandwidth: Wider Main Memory

First-level caches are often organized with a physical width of one word because most CPU accesses are that size; see Figure 5.27(a). Doubling or quadrupling the width of the cache and the memory will therefore double or quadruple the memory bandwidth. With a main memory width of two words, the miss penalty in our example would drop from 4×64 or 256 clock cycles as calculated above to 2×64 or 128 clock cycles. The reason is at twice the width we need half the memory accesses, and each takes 64 clock cycles. At four words wide the miss penalty is just 1×64 clock cycles. The bandwidth is then one-quarter byte per clock cycle at two words wide and one-half byte per clock cycle when the memory is four words wide.

There is cost in the wider connection between the CPU and memory, typically called a memory *bus*. CPUs will still access the cache a word at a time, so there now needs to be a multiplexor between the cache and the CPU—and that multiplexor may be on the critical timing path. Second-level caches can help since the multiplexing can be between first- and second-level caches, not on the critical path; see Figure 5.27(b).

Since main memory is traditionally expandable by the customer, a drawback to wide memory is that the minimum increment is doubled or quadrupled when the width is doubled or quadrupled. In addition, memories with error correction have difficulties with writes to a portion of the protected block, such as a byte. The rest of the data must be read so that the new error correction code can be calculated and stored when the data are written. (Section 5.15 describes error correction on the Sun Fire 6800 server.) If the error correction is done over the full width, the wider memory will increase the frequency of such "read-modify-write" sequences because more writes become partial block writes. Many designs of wider memory have separate error correction every word since most writes are that size.

Second Technique for Higher Bandwidth: Simple Interleaved Memory

Increasing width is one way to improve bandwidth, but another is to take advantage of the potential parallelism of having many chips in a memory system. Memory chips can be organized in *banks* to read or write multiple words at a time rather than a single word. In general, the purpose of interleaved memory is to try to take advantage of the potential memory bandwidth of *all* the chips in the system; in contrast, most memory systems activate only the chips containing the needed words. The two philosophies affect the power of the memory system,

leading to different decisions depending on the relative importance of power versus performance.

The banks are often one word wide so that the width of the bus and the cache need not change, but sending addresses to several banks permits them all to read simultaneously. Figure 5.27(c) shows this organization. For example, sending an address to four banks (with access times shown on page 437) yields a miss penalty of 4 + 56 + (4 × 4) or 76 clock cycles, giving a bandwidth of about 0.4 bytes per clock cycle. Banks are also valuable on writes. Although back-to-back writes would normally have to wait for earlier writes to finish, banks allow one clock cycle for each write, provided the writes are not destined to the same bank. Such a memory organization is especially important for write through.

The mapping of addresses to banks affects the behavior of the memory system. The example above assumes the addresses of the four banks are interleaved at the word level: bank 0 has all words whose address modulo 4 is 0, bank 1 has all words whose address modulo 4 is 1, and so on. Figure 5.28 shows this interleaving.

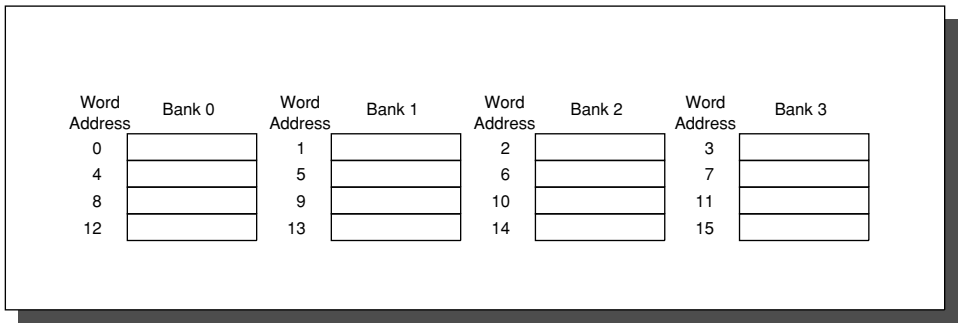


FIGURE 5.28 Four-way interleaved memory. This example assumes word addressing: with byte addressing and eight bytes per word, each of these addresses would be multiplied by eight.

This mapping is referred to as the *interleaving factor*; *interleaved memory* normally means banks of memory that are word interleaved. This interleaving optimizes sequential memory accesses. A cache read miss is an ideal match to word-interleaved memory, as the words in a block are read sequentially. Write-back caches make writes as well as reads sequential, getting even more efficiency from word-interleaved memory.

EXAMPLE What can interleaving and wide memory buy? Consider the following description of a computer and its cache performance:

Block size = 1 word

Memory bus width = 1 word

Miss rate = 3%

Memory accesses per instruction = 1.2

Cache miss penalty = 64 cycles (as above)

Average cycles per instruction (ignoring cache misses) = 2

If we change the block size to two words, the miss rate falls to 2%, and a four-word block has a miss rate of 1.2%. What is the improvement in performance of interleaving two ways and four ways versus doubling the width of memory and the bus, assuming the access times on page 437?

ANSWER The CPI for the base computer using one-word blocks is

$$2 + (1.2 \times 3\% \times 64) = 4.30$$

Since the clock cycle time and instruction count won't change in this example, we can calculate performance improvement by just comparing CPI.

Increasing the block size to two words gives the following options:

$$\text{64-bit bus and memory, no interleaving} = 2 + (1.2 \times 2\% \times 2 \times 64) = 5.07$$

$$\text{64-bit bus and memory, interleaving} = 2 + (1.2 \times 2\% \times (4 + 56 + 8)) = 3.63$$

$$\text{128-bit bus and memory, no interleaving} = 2 + (1.2 \times 2\% \times 1 \times 64) = 3.54$$

Thus, doubling the block size slows down the straightforward implementation (5.07 versus 4.30), while interleaving or wider memory is 1.19 or 1.22 times faster, respectively. If we increase the block size to four, the following is obtained:

$$\text{64-bit bus and memory, no interleaving} = 2 + (1.2 \times 1.2\% \times 4 \times 64) = 5.69$$

$$\text{64-bit bus and memory, interleaving} = 2 + (1.2 \times 1.2\% \times (4 + 56 + 16)) = 3.09$$

$$\text{128-bit bus and memory, no interleaving} = 2 + (1.2 \times 1.2\% \times 2 \times 64) = 3.84$$

Again, the larger block hurts performance for the simple case (5.69 vs. 4.30), although the interleaved 64-bit memory is now fastest—1.39 times faster versus 1.22 for the wider memory and bus. n

This subsection has shown that interleaved memory is logically a wide memory, except that accesses to banks are staged over time to share internal resources—the memory bus in this example.

How many banks should be included? One metric, used in vector computers, is as follows:

$$\text{Number of banks} \geq \text{Number of clock cycles to access word in bank}$$

The memory system goal is to deliver information from a new bank each clock cycle for sequential accesses. To see why this formula holds, imagine there were fewer banks than clock cycles to access a word in a 64-bit bank; say, 8 banks with an access time of 10 clock cycles. After 10 clock cycles the CPU could get a word from bank 0, and then bank 0 would begin fetching the next desired word as the CPU received the following 7 words from the other 7 banks. At clock cycle 18 the CPU would be at the door of bank 0, waiting for it to supply the next word. The CPU would have to wait until clock cycle 20 for the word to appear. Hence, we want more banks than clock cycles to access a bank to avoid waiting.

We will discuss conflicts on nonsequential accesses to banks in the following subsection. For now, we note that having many banks reduces the chance of these bank conflicts.

Ironically, as capacity per memory chip increases, there are fewer chips in the same-sized memory system, making multiple banks much more expensive. For example, a 512-MB main memory takes 256 memory chips of 4 M × 4 bit, easily organized into 16 banks of 16 memory chips. However, it takes only sixteen 64-M × 4-bit memory chips for 64 MB, making one bank the limit. Many manufacturers will want to have a small memory option in the baseline model. This shrinking number of chips is the main disadvantage of interleaved memory banks. Chips organized with wider paths, such as 16 M × 16 bits, postpone this weakness.

A second disadvantage of memory banks is again the difficulty of main memory expansion. Either the memory system must support multiple generations of memory chips, or the memory controller changes the interleaving based on the size of physical memory, or both.

Third Technique for Higher Bandwidth: Independent Memory Banks

The original motivation for memory banks was higher memory bandwidth by interleaving sequential accesses. This hardware is not much more difficult since the banks can share address lines with a memory controller, enabling each bank to use the data portion of the memory bus.

A generalization of interleaving is to allow multiple independent accesses, where multiple memory controllers allow banks (or sets of word-interleaved banks) to operate independently. Each bank needs separate address lines and possibly a separate data bus. For example, an input device may use one controller and one bank, the cache read may use another, and a cache write may use a third. Nonblocking caches (page 421) allow the CPU to proceed beyond a cache miss, potentially allowing multiple cache misses to be serviced simultaneously. Such a design only makes sense with memory banks; otherwise the multiple reads will be serviced by a single memory port and get only a small benefit of overlapping access with transmission. Multiprocessors that share a common memory provide further motivation for memory banks (see Chapter 6).

Independent of memory technology, higher bandwidth is available using memory banks, by making memory and its bus wider, or doing both. The next section examines the underlying memory technology.

5.9 Memory Technology

… the one single development that put computers on their feet was the invention of a reliable form of memory, namely, the core memory. … Its cost was reasonable, it was reliable and, because it was reliable, it could in due course be made large. [p. 209]

Maurice Wilkes, *Memoirs of a Computer Pioneer* (1985)

The prior section described ways to organize memory chips; this section describes the technology inside the memory chips. Before describing the options, let's go over the performance metrics.

Memory latency is traditionally quoted using two measures—access time and cycle time. *Access time* is the time between when a read is requested and when the desired word arrives, while *cycle time* is the minimum time between requests to memory. One reason that cycle time is greater than access time is that the memory needs the address lines to be stable between accesses.

DRAM technology

The main memory of virtually every desktop or server computer sold since 1975 is composed of semiconductor DRAMs,.

As early DRAMs grew in capacity, the cost of a package with all the necessary address lines was an issue. The solution was to multiplex the address lines, thereby cutting the number of address pins in half. Figure 5.29 shows the basic DRAM organization. One half of the address is sent first, called the *row access strobe* or *RAS*. It is followed by the other half of the address, sent during the *column access strobe* or *CAS*. These names come from the internal chip organization, since the memory is organized as a rectangular matrix addressed by rows and columns.

An additional requirement of DRAM derives from the property signified by its first letter, *D*, for *dynamic*. To pack more bits per chip, DRAMs use only a single transistor to store a bit. Reading that bit can disturb the information, however. To prevent loss of information, each bit must be "refreshed" periodically. Fortunately, all the bits in a row can be refreshed simultaneously just by reading that row. Hence, every DRAM in the memory system must access every row within a certain time window, such as 8 milliseconds. Memory controllers include hardware to periodically refresh the DRAMs.

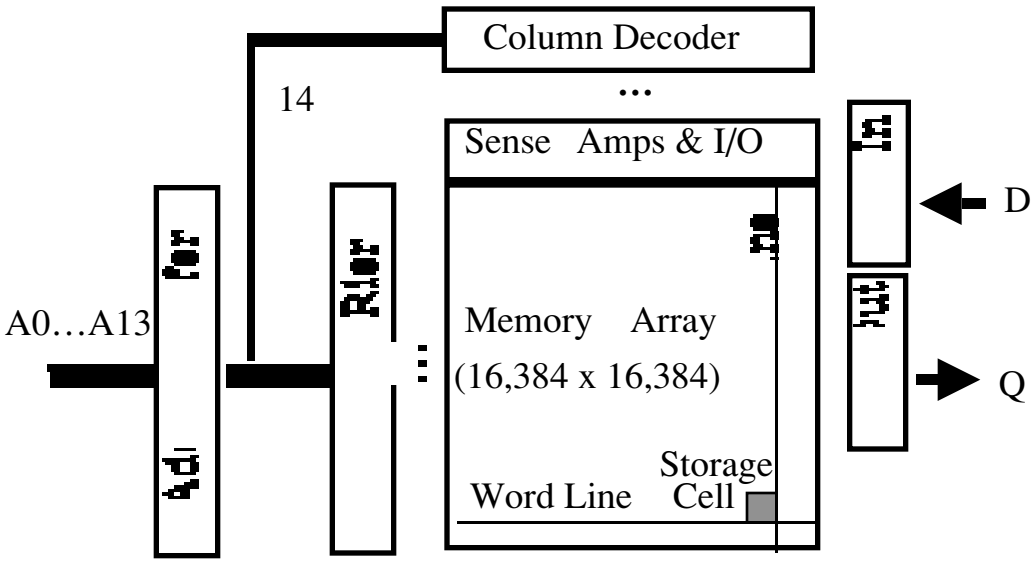


FIGURE 5.29 Internal organization of a 64-Mbit DRAM. DRAMs often use banks of memory arrays internally, and select between them. For example, instead of one 16,384 x 16,384 memory, a DRAM might use 256 1,024 x 1,024 arrays or 16 2,048 x 2,048 arrays.

This requirement means that the memory system is occasionally unavailable because it is sending a signal telling every chip to *refresh*. The time for a refresh is typically a full memory access (RAS and CAS) for each row of the DRAM. Since the memory matrix in a DRAM is conceptually square, the number of steps in a refresh is usually the square root of the DRAM capacity. DRAM designers try to keep time spent refreshing to be less than 5% of the total time.

Earlier sections presented main memory as if operated like a Swiss train, consistently delivering the goods exactly according to schedule. Refresh belies that myth, for some accesses take much longer than others do. Thus, refresh is another reason for variability of memory latency and hence cache miss penalty.

Amdahl suggested a rule of thumb that memory capacity should grow linearly with CPU speed to keep a balanced system, so that a 1000 MIPS processor should have 1000 megabytes of memory. CPU designers rely on DRAMs to supply that demand: in the past they expected a four-fold improvement in capacity every three years, or 55% per year. Unfortunately, the performance of DRAMs is growing at a much slower rate. Figure 5.30 shows a performance improvement in row access time, which is related to latency, of about 5% per year. The CAS or Data Transfer Time, which is related to bandwidth, is growing at more than twice that rate.

Year of introduction	Chip size	Row access strobe (RAS)		Column access strobe (CAS) / Data Transfer Time	Cycle time
		Slowest DRAM	Fastest DRAM		
1980	64 Kbit	180 ns	150 ns	75 ns	250 ns
1983	256 Kbit	150 ns	120 ns	50 ns	220 ns
1986	1 Mbit	120 ns	100 ns	25 ns	190 ns
1989	4 Mbit	100 ns	80 ns	20 ns	165 ns
1992	16 Mbit	80 ns	60 ns	15 ns	120 ns
1996	64 Mbit	70 ns	50 ns	12 ns	110 ns
1998	128 Mbit	70 ns	50 ns	10 ns	100 ns
2000	256 Mbit	65 ns	45 ns	7 ns	90 ns
2002	512 Mbit	60 ns	40 ns	5 ns	80 ns

FIGURE 5.30 Times of fast and slow DRAMs with each generation. Performance improvement of row access time is about 5% per year. The improvement by a factor of two in column access accompanied the switch from NMOS DRAMs to CMOS DRAMs.

Although we have been talking about individual chips, DRAMs are commonly sold on small boards called *DIMMs* for *Dual Inline Memory Modules*. DIMMs typically contain 4 to 16 DRAMs. They are normally organized to be eight bytes wide for desktop systems.

In addition to the DIMM packaging and the new interfaces to improve the data transfer time, discussed in the following subsections, the biggest change to DRAMs has been a slowing down in capacity growth. For 20 years DRAMs obeyed Moore's Law, bringing out a new chip with four times the capacity every three years. As a result of a slowing in demand for DRAMs, since 1998 new chips only double capacity every two years. In 2001, this new slower pace shows no sign of changing.

Just as virtually all desktop or server computer since 1975 used DRAMs for main memory, virtually all use SRAM for cache, the topic of the next subsection.

SRAM Technology

In contrast to DRAMs are SRAMs—the first letter standing for *static*. The dynamic nature of the circuits in DRAM require data to be written back after being read, hence the difference between the access time and the cycle time as well as the need to refresh. SRAMs typically use six transistors per bit to prevent the information from being disturbed when read.

This difference in refresh alone can make a difference for embedded applications. Devices often go into low power or standby mode for long periods. SRAM needs only minimal power to retain the charge in standby mode, but DRAMs must continue to be refreshed occasionally so as to not lose information.

In DRAM designs the emphasis is on cost per bit and capacity, while SRAM designs are concerned with speed and capacity. (Because of this concern, SRAM address lines are not multiplexed.). Thus, unlike DRAMs, there is no difference between access time and cycle time. For memories designed in comparable technologies, the capacity of DRAMs is roughly 4 to 8 times that of SRAMs. The cycle time of SRAMs is 8 to 16 times faster than DRAMs, but they are also 8 to 16 times as expensive.

Embedded Processor Memory Technology: ROM and Flash

Embedded computers usually have small memories, and most do not have a disk to act as non-volatile storage. Two memory technologies are found in embedded computers to address this problem.

The first is *Read-Only Memory* (*ROM*). ROM is programmed at time of manufacture, needing only a single transistor per bit to represent 1 or 0. ROM is used for the embedded program and for constants, often included as part of a larger chip.

In addition to being non-volatile, ROM is also non-destructible; nothing the computer can do can modify the contents of this memory. Hence, ROM also provides a level of protection to the code of embedded computers. Since address-based protection is often not enabled in embedded processors, ROM can fulfill an important role.

The second memory technology offers non-volatility but allows the memory to be modified. *Flash memory* allows the embedded device to alter nonvolatile memory after the system is manufactured, which can shorten product development. Flash memory, described in on page 498 in Chapter 7, allows reading at almost DRAM speeds but writing flash is 10 to 100 times slower. In 2001, the DRAM capacity per chip and the megabytes per dollar is about four to eight times greater than flash memory.

Improving Memory Performance in a standard DRAM Chip

As Moore's Law continues to supply more transistors and as the processor-memory gap increases pressure on memory performance, some of the ideas of the prior section have made their way inside the DRAM chip. Generally the idea has been for greater bandwidth, often at the cost of greater latency. This subsection presents techniques that take advantage of the nature of DRAMs.

As mentioned earlier, a DRAM access is divided into row access and column access. DRAMs must buffer a row of bits inside the DRAM for the column access, and this row is usually the square root of the DRAM size—8 Kbits for 64 Mbits, 16 Kbits for 256 Mbits, and so on.

Although presented logically as a single monolithic array of memory bits, the internal organization of DRAM actually consists of many memory modules. For a variety of manufacturing reasons, these modules are usually 1 to 4 megabits. Thus, if you were to examine a 256 Mbit DRAM under a microscope, you might see 128 2-megabit memory arrays on the chip. This large number of arrays internally presents the opportunity to provide much higher bandwidth off chip.

To improve bandwidth, there have been a variety of evolutionary innovations over time. The first was timing signals that allow repeated accesses to the row buffer without another row access time, typically called *fast page mode*. Such a buffer comes naturally, as each array will buffer 1024 to 2048 bits for each access.

The second major change is that conventional DRAMs have an asynchronous interface to the memory controller, and hence every transfer involves overhead to synchronize with the controller. The solution was to add a clock signal to the DRAM interface, so that the repeated transfers would not bear that overhead. This optimization is called *Synchronous DRAM*, abbreviated *SDRAM*. SDRAMs typically also have a programmable register to hold the number of bytes requested, and hence can send many bytes over several cycles per request.

In 2001 the bus rates are 100 MHz to 150 MHz. SDRAM DIMMs of these speeds are called PC100, PC133, and PC 150, based on the clock speed of the individual chip. Multiplying the eight-byte width of the DIMM times the clock rate, the peak speed per memory module is 800 to 1200 MB/sec.

The third major DRAM innovation to increase bandwidth is to transfer data on both the rising edge and falling edge of the DRAM clock signal, thereby doubling the peak data rate. This optimization is called *Double Data Rate*, and abbreviated *DDR*. The bus speeds for these DRAMs are also 100 to 150 MHz, but these DDR DIMMs are confusingly labeled by the peak *DIMM* bandwidth. The name PC1600 comes from100 MHz $\times$ 2 $\times$ 8 bytes or 1600 Megabytes/second, 133 MHz leads to PC2100, 150 MHz yields PC2400, and so on.

In each of the three cases the advantage of such optimizations is that they add a small amount of logic to exploit the high internal DRAM bandwidth, adding little cost to the system while achieving a significant improvement in bandwidth. Unlike traditional interleaved memories, there is no danger in using such a mode as DRAM chips increase in capacity.

Improving Memory Performance via a new DRAM Interface: RAMBUS

Recently new breeds of DRAMs have been produced that further optimize the interface between the DRAM and CPU. The company RAMBUS takes the standard DRAM core and provides a new interface, making a single chip act more like a memory system than a memory component: each chip has interleaved memory and a high speed interface. RAMBUS licenses its technology to companies that use its interface, both DRAM and microprocessor manufacturers.

The first generation RAMBUS interface dropped RAS/CAS, replacing it with a bus that allows other accesses over the bus between the sending of the address and return of the data. It is typically called *RDRAM*. (Such a bus is called a *packet-switched bus* or *split-transaction* bus, described in Chapters 7 and 8.) This bus allows a single chip to act as a memory bank. A chip can return a variable amount of data from a single request, and even perform its own refresh. RDRAM offered a byte-wide interface, and was one of the first DRAMs to use a clock signal, and it also transfers on both edges of its clock. Inside each chip were four banks, each with their own row buffer. To run at its 300 Mhz clock, the RAMBUS bus is limited to be no more than 4 inches long. Typically a microprocessor uses a single RAMBUS channel, so just one RDRAM is transferring at a time.

The second generation RAMBUS interface, called *Direct RDRAM* or *DRDRAM*, offers up to 1.6 GBytes/second of bandwidth from a single DRAM. Innovations in this interface include a separate row- and column-command buses instead of the conventional multiplexing; an 18-bit data bus; expanding from 4 to 16 internal banks per RDRAM to reduce bank conflicts; increasing the number of row buffers from 4 to 8; increasing the clock to 400 MHz clock; and a much more sophisticated controller on chip. Because of the separation of data, row, and column buses, three transactions can be performed simultaneously.

RAMBUS helped set the new optimistic naming trend, calling the 350 MHz part PC700, the 400 MHz part PC800, and so on. Since each chip is 2 bytes wide, the peak chip bandwidth of PC700 is 1400 MB/second, PC800 is 1600 MB/second, and so on. RAMBUS chips are not sold in DIMMs but in "RIMMs" which is similar in size but incompatible with DIMMs. RIMMs are designed to have a single RAMBUS chip on the RIMM supply the memory bandwidth needs of the computer, and are not interchangeable with DIMMs.

Comparing RAMBUS and DDR SDRAM

How does the RAMBUS interface compare in cost and performance when placed in a system? Most main memory systems already use SDRAM to get more bits per memory access, in the hope of reducing the CPU-DRAM performance gap. Since the most computers use memory in DIMM packages, which are typically at least 64-bits wide, the DIMM memory bandwidth is closer to what RAMBUS provides than you might expect when just comparing DRAM chips.

The one note of caution is that performance of cache based systems are based in part on latency to the first byte and in part on the bandwidth to deliver the rest of the bytes in the block. Although these innovations help with the latter case, none help with latency. Amdahl's Law reminds us of the limits of accelerating one piece of the problem while ignoring another part.

In addition to performance, the new breed of DRAMs such as RDRAM and DRDRAM have price a premium over traditional DRAMs to provide the greater

bandwidth since these chips are larger. The question over time is how much more. In 2001 it is factor of two; Section 5.16 has a detailed price-performance evaluation.

The marketplace will determine whether the more radical DRAMs such as RAMBUS will become popular for main memory, or whether the price premium restricts them to niche markets.

5.10 Virtual Memory

… a system has been devised to make the core drum combination appear to the programmer as a single level store, the requisite transfers taking place automatically.

Kilburn et al. [1962]

At any instant in time computers are running multiple processes, each with its own address space. (Processes are described in the next section.) It would be too expensive to dedicate a full-address-space worth of memory for each process, especially since many processes use only a small part of their address space. Hence, there must be a means of sharing a smaller amount of physical memory among many processes. One way to do this, *virtual memory*, divides physical memory into blocks and allocates them to different processes. Inherent in such an approach must be a *protection* scheme that restricts a process to the blocks belonging only to that process. Most forms of virtual memory also reduce the time to start a program, since not all code and data need be in physical memory before a program can begin.

Although protection provided by virtual memory is essential for current computers, sharing is not the reason that virtual memory was invented. If a program became too large for physical memory, it was the programmer's job to make it fit. Programmers divided programs into pieces, then identified the pieces that were mutually exclusive, and loaded or unloaded these *overlays* under user program control during execution. The programmer ensured that the program never tried to access more physical main memory than was in the computer, and that the proper overlay was loaded at the proper time. As one can well imagine, this responsibility eroded programmer productivity.

Virtual memory was invented to relieve programmers of this burden; it automatically manages the two levels of the memory hierarchy represented by main memory and secondary storage. Figure 5.31 shows the mapping of virtual memory to physical memory for a program with four pages.

In addition to sharing protected memory space and automatically managing the memory hierarchy, virtual memory also simplifies loading the program for execution. Called *relocation*, this mechanism allows the same program to run in any location in physical memory. The program in Figure 5.31 can be placed any-

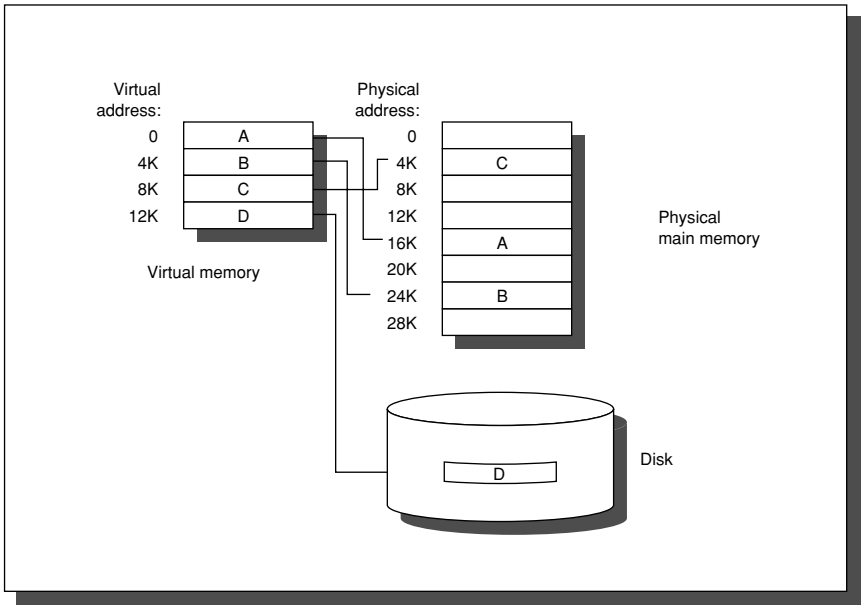


FIGURE 5.31 The logical program in its contiguous virtual address space is shown on the left. It consists of four pages A, B, C, and D. The actual location of three of the blocks is in physical main memory and the other is located on the disk.

where in physical memory or disk just by changing the mapping between them. (Prior to the popularity of virtual memory, processors would include a relocation register just for that purpose.) An alternative to a hardware solution would be software that changed all addresses in a program each time it was run.

Several general memory-hierarchy ideas from Chapter 1 about caches are analogous to virtual memory, although many of the terms are different. *Page* or *segment* is used for block, and *page fault* or *address fault* is used for miss. With virtual memory, the CPU produces *virtual addresses* that are translated by a combination of hardware and software to *physical addresses*, which access main memory. This process is called *memory mapping* or *address translation*. Today, the two memory-hierarchy levels controlled by virtual memory are DRAMs and magnetic disks. Figure 5.32 shows a typical range of memory-hierarchy parameters for virtual memory.

There are further differences between caches and virtual memory beyond those quantitative ones mentioned in Figure 5.32:

- Replacement on cache misses is primarily controlled by hardware, while virtual memory replacement is primarily controlled by the operating system. The longer miss penalty means it's more important to make a good decision, so the operating system can be involved and spend take time deciding what to replace.
- The size of the processor address determines the size of virtual memory, but the cache size is independent of the processor address size.

Parameter	First-level cache	Virtual memory
Block (page) size	16-128 bytes	4096-65,536 bytes
Hit time	1-3 clock cycles	50-150 clock cycles
Miss penalty	8-150 clock cycles	1,000,000-10,000,000 clock cycles
(Access time)	(6-130 clock cycles)	(800,000-8,000,000 clock cycles)
(Transfer time)	(2-20 clock cycles)	(200,000-2,000,000 clock cycles)
Miss rate	0.1-10%	0.00001- 0.001%
Address mapping	25- 45 bit physical address to 14- 20 bit cache address	32-64 bit virtual address to 25-45 bit physical address

FIGURE 5.32 Typical ranges of parameters for caches and virtual memory. Virtual memory parameters represent increases of 10 to 1,000,000 times over cache parameters. Normally first level caches contain at most 1 megabyte of data while physical memory contains 32 megabytes to 1 terabyte.

- In addition to acting as the lower-level backing store for main memory in the hierarchy, secondary storage is also used for the file system. In fact, the file system occupies most of secondary storage. It is not normally in the address space.

Virtual memory also encompasses several related techniques. Virtual memory systems can be categorized into two classes: those with fixed-size blocks, called *pages*, and those with variable-size blocks, called *segments*. Pages are typically fixed at 4096 to 65,536 bytes, while segment size varies. The largest segment supported on any processor ranges from 2^{16} bytes up to 2^{32} bytes; the smallest segment is 1 byte. Figure 5.33 shows how the two approaches might divide code and data.

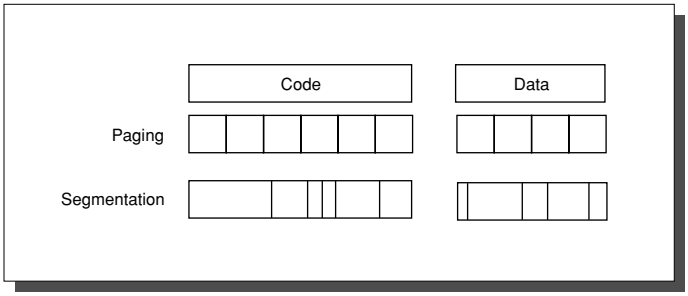


FIGURE 5.33 Example of how paging and segmentation divide a program.

The decision to use paged virtual memory versus segmented virtual memory affects the CPU. Paged addressing has a single fixed-size address divided into page number and offset within a page, analogous to cache addressing. A single address does not work for segmented addresses; the variable size of segments re-

quires one word for a segment number and one word for an offset within a segment, for a total of two words. An unsegmented address space is simpler for the compiler.

The pros and cons of these two approaches have been well documented in operating systems textbooks; Figure 5.34 summarizes the arguments. Because of the replacement problem (the third line of the figure), few computers today use pure segmentation. Some computers use a hybrid approach, called *paged segments*, in which a segment is an integral number of pages. This simplifies replacement because memory need not be contiguous, and the full segments need not be in main memory. A more recent hybrid is for a computer to offer multiple page sizes, with the larger sizes being powers of two times the smallest page size. The IBM 405CR embedded processor, for example, allows 1 KB, 4 KB ($2^2 \times 1$ KB), 16 KB ($2^4 \times 1$ KB), 64 KB ($2^6 \times 1$ KB), 256 KB ($2^8 \times 1$ KB), 1024 KB ($2^{10} \times 1$ KB), and 4096 KB ($2^{12} \times 1$ KB) to act as a single page.

	Page	**Segment**
Words per address	One	Two (segment and offset)
Programmer visible?	Invisible to application programmer	May be visible to application programmer
Replacing a block	Trivial (all blocks are the same size)	Hard (must find contiguous, variable-size, unused portion of main memory)
Memory use inefficiency	Internal fragmentation (unused portion of page)	External fragmentation (unused pieces of main memory)
Efficient disk traffic	Yes (adjust page size to balance access time and transfer time)	Not always (small segments may transfer just a few bytes)

FIGURE 5.34 Paging versus segmentation. Both can waste memory, depending on the block size and how well the segments fit together in main memory. Programming languages with unrestricted pointers require both the segment and the address to be passed. A hybrid approach, called *paged segments*, shoots for the best of both worlds: segments are composed of pages, so replacing a block is easy, yet a segment may be treated as a logical unit.

We are now ready to answer the four memory-hierarchy questions for virtual memory.

Q1: Where can a block be placed in main memory?

The miss penalty for virtual memory involves access to a rotating magnetic storage device and is therefore quite high. Given the choice of lower miss rates or a simpler placement algorithm, operating systems designers normally pick lower miss rates because of the exorbitant miss penalty. Thus, operating systems allow blocks to be placed anywhere in main memory. According to the terminology in Figure 5.4 (page 382), this strategy would be labeled fully associative.

Q2: How is a block found if it is in main memory?

Both paging and segmentation rely on a data structure that is indexed by the page or segment number. This data structure contains the physical address of the block. For segmentation, the offset is added to the segment's physical address to obtain the final physical address. For paging, the offset is simply concatenated to this physical page address (see Figure 5.35).

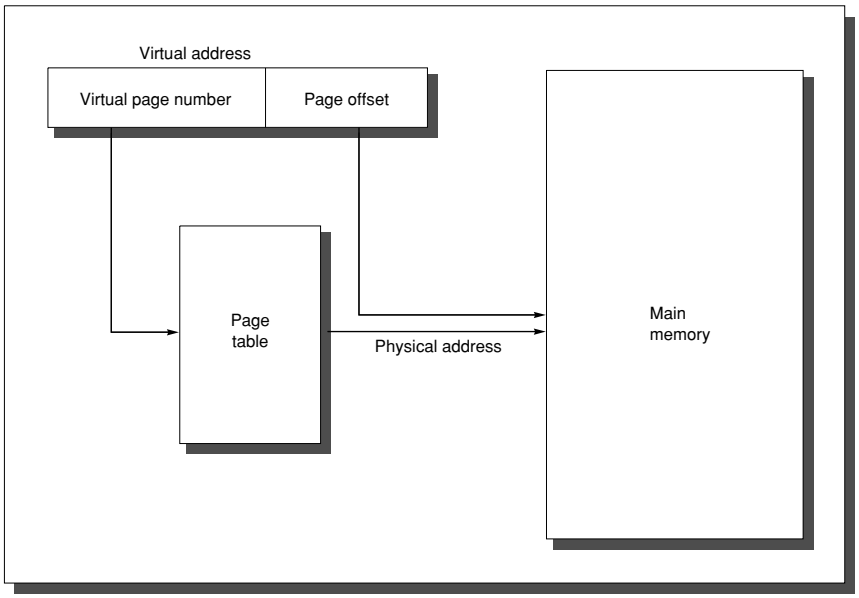


FIGURE 5.35 The mapping of a virtual address to a physical address via a page table.

This data structure, containing the physical page addresses, usually takes the form of a *page table*. Indexed by the virtual page number, the size of the table is the number of pages in the virtual address space. Given a 32-bit virtual address, 4-KB pages, and 4 bytes per page table entry, the size of the page table would be $(2^{32}/2^{12}) \times 2^2 = 2^{22}$ or 4 MB.

To reduce the size of this data structure, some computers apply a hashing function to the virtual address. The hash allows the data structure to be the length of the number of *physical* pages in main memory. This number could be much smaller than the number of virtual pages. Such a structure is called an *inverted page table*. Using the example above, a 512-MB physical memory would only need 1 MB (8×512 MB/4 KB) for an inverted page table; the extra 4 bytes per page table entry is for the virtual address. The HP/Intel IA-64 covers both bases by offering both traditional pages tables *and* inverted page tables, leaving the choice of mechanism to the operating system programmer.

To reduce address translation time, computers use a cache dedicated to these address translations, called a *translation look-aside buffer*, or simply *translation buffer*. They are described in more detail shortly.

Q3: Which block should be replaced on a virtual memory miss?

As mentioned above, the overriding operating system guideline is minimizing page faults. Consistent with this guideline, almost all operating systems try to replace the least-recently used (LRU) block, because if the past predicts the future, that is the one less likely to be needed.

To help the operating system estimate LRU, many processors provide a *use bit* or *reference bit*, which is logically set whenever a page is accessed. (To reduce work, it is actually set only on a translation buffer miss, which is described shortly.) The operating system periodically clears the use bits and later records them so it can determine which pages were touched during a particular time period. By keeping track in this way, the operating system can select a page that is among the least-recently referenced.

Q4: What happens on a write?

The level below main memory contains rotating magnetic disks that take millions of clock cycles to access. Because of the great discrepancy in access time, no one has yet built a virtual memory operating system that writes through main memory to disk on every store by the CPU. (This remark should not be interpreted as an opportunity to become famous by being the first to build one!) Thus, the write strategy is always write back.

Since the cost of an unnecessary access to the next-lower level is so high, virtual memory systems usually include a dirty bit. It allows blocks to be written to disk only if they have been altered since being read from the disk.

Techniques for Fast Address Translation

Page tables are usually so large that they are stored in main memory, and sometimes paged themselves. Paging means that every memory access logically takes at least twice as long, with one memory access to obtain the physical address and a second access to get the data. This cost is far too dear.

One remedy is to remember the last translation, so that the mapping process is skipped if the current address refers to the same page as the last one. A more general solution is to again rely on the principle of locality; if the accesses have locality, then the *address translations* for the accesses must also have locality. By keeping these address translations in a special cache, a memory access rarely requires a second access to translate the data. This special address translation cache is referred to as a *translation look-aside buffer* or TLB, also called a *translation buffer* or TB.

A TLB entry is like a cache entry where the tag holds portions of the virtual address and the data portion holds a physical page frame number, protection field, valid bit, and usually a use bit and dirty bit. To change the physical page

frame number or protection of an entry in the page table, the operating system must make sure the old entry is not in the TLB; otherwise, the system won't behave properly. Note that this dirty bit means the corresponding *page* is dirty, not that the address translation in the TLB is dirty nor that a particular block in the data cache is dirty. The operating system resets these bits by changing the value in the page table and then invalidating the corresponding TLB entry. When the entry is reloaded from the page table, the TLB gets an accurate copy of the bits.

Figure 5.36 shows the Alpha 21264 data TLB organization, with each step of a translation labeled. The TLB uses fully associative placement; thus, the translation begins (steps 1 and 2) by sending the virtual address to all tags. Of course, the tag must be marked valid to allow a match. At the same time, the type of memory access is checked for a violation (also in step 2) against protection information in the TLB.

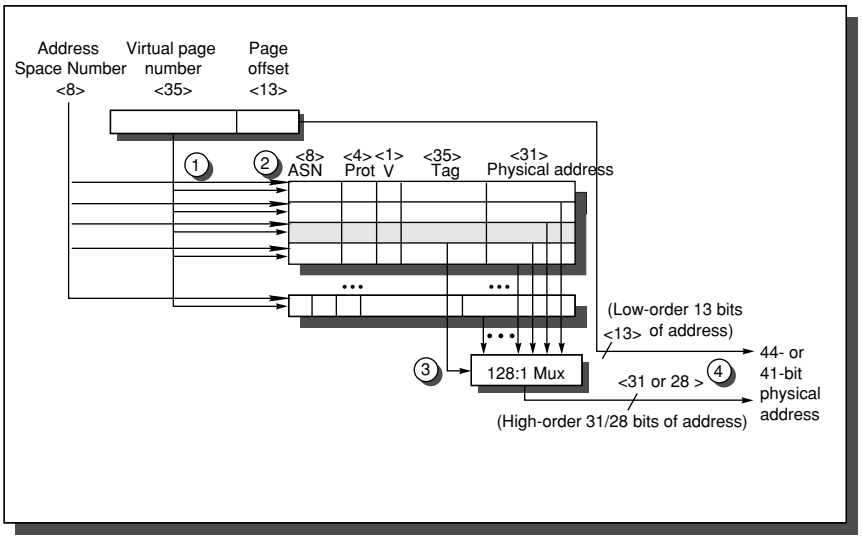


FIGURE 5.36 Operation of the Alpha 21264 data TLB during address translation. The four steps of a TLB hit are shown as circled numbers. The Address Space Number (ASN) is used like a Process ID for virtual caches, in that the TLB is not flushed on a context switch, only when ASNs are recycled. The next fields of an entry are protection permissions (Prot) and the valid bit (V). Note that there is no specific reference, use bit, or dirty bit. Hence, a page replacement algorithm such as LRU must rely on disabling reads and writes occasionally to record reads and writes to pages to measure usage and whether or not pages are dirty. The advantage of these omissions is that the TLB need not be written during normal memory accesses nor during a TLB miss. Alpha 21264 has an option of either 44-bit or 41-bit physical addresses. This TLB has 128 entries.

To reduce TLB misses due to context switches, each entry has an 8-bit Address Space Number or ASN, which plays the same role as a Process ID number

mentioned in Figure 5.25 on page 432. If the context switching returns to the process with the same ASN, it can still match the TLB. Thus, the process ASN and the PTE ASN must also match for a valid tag.

For reasons similar to those in the cache case, there is no need to include the 13 bits of the Alpha 21264 page offset in the TLB. The matching tag sends the corresponding physical address through effectively a 128:1 multiplexor (step 3). The page offset is then combined with the physical page frame to form a full physical address (step 4). The address size is 44 or 41 bits depending on a physical address mode bit (see section 5.11).

Address translation can easily be on the critical path determining the clock cycle of the processor, so the 21264 uses a virtually addressed instruction cache, thus the TLB is only accessed during an instruction cache miss.

Selecting a Page Size

The most obvious architectural parameter is the page size. Choosing the page is a question of balancing forces that favor a larger page size versus those favoring a smaller size. The following favor a larger size:

- The size of the page table is inversely proportional to the page size; memory (or other resources used for the memory map) can therefore be saved by making the pages bigger.
- As mentioned on page 433 in section 5.7, a larger page size can allow larger caches with fast cache hit times.
- Transferring larger pages to or from secondary storage, possibly over a network, is more efficient than transferring smaller pages.
- The number of TLB entries are restricted, so a larger page size means that more memory can be mapped efficiently, thereby reducing the number of TLB misses.

It is for this final reason that recent microprocessors have decided to support multiple page sizes; for some programs, TLB misses can be as significant on CPI as the cache misses.

The main motivation for a smaller page size is conserving storage. A small page size will result in less wasted storage when a contiguous region of virtual memory is not equal in size to a multiple of the page size. The term for this unused memory in a page is *internal fragmentation*. Assuming that each process has three primary segments (text, heap, and stack), the average wasted storage per process will be 1.5 times the page size. This amount is negligible for computers with hundreds of megabytes of memory and page sizes of 4 KB to 8 KB. Of course, when the page sizes become very large (more than 32 KB), lots of storage (both main and secondary) may be wasted, as well as I/O bandwidth. A final concern is process start-up time; many processes are small, so a large page size would lengthen the time to invoke a process

Summary of Virtual Memory and Caches

With virtual memory, TLBs, first level caches, and second levels caches all mapping portions of the virtual and physical address space, it can get confusing what bits go where. Figure 5.37 gives a hypothetical example going from a 64-bit virtual address to a 41 bit physical address with two levels of cache. This L1 cache is virtually indexed, physically tagged since both the cache size and the page size are 8 KB. The L2 cache is 4 MB. The block size for both is 64 bytes.

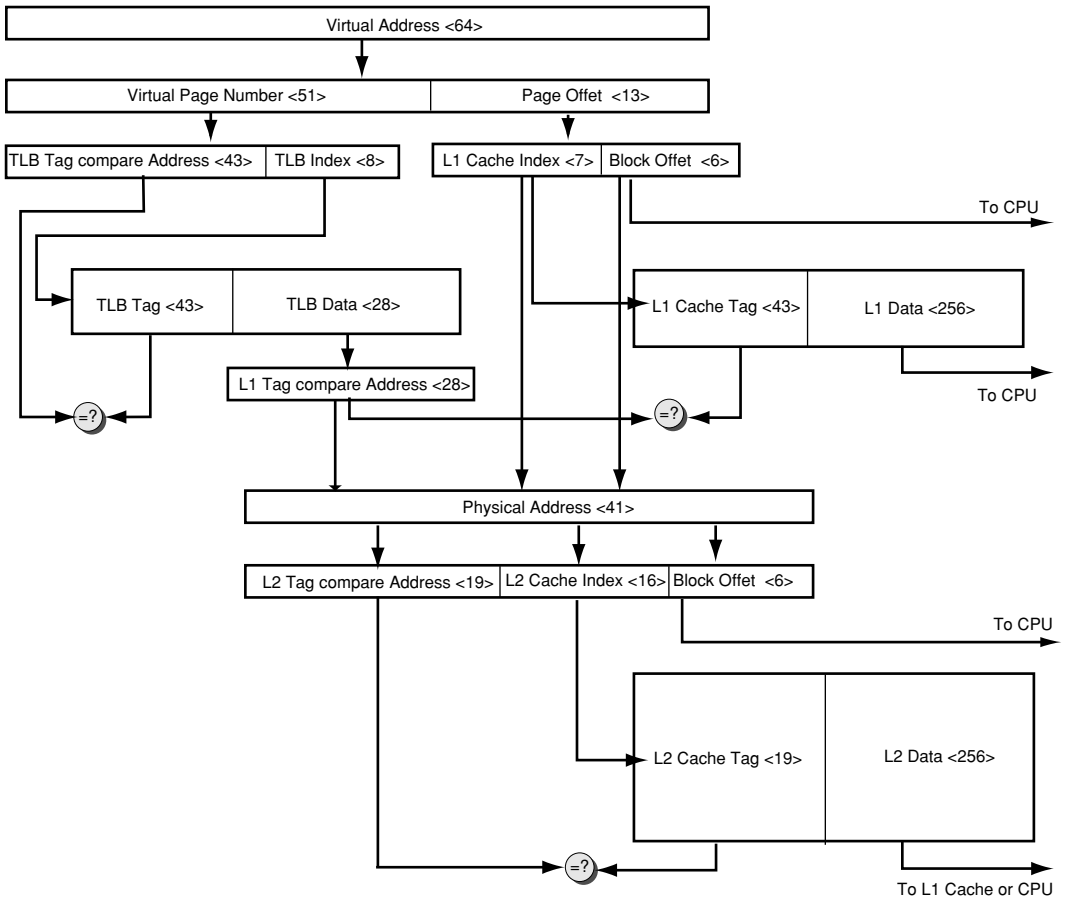


FIGURE 5.37 The overall picture of an hypothetical memory hierarchy going from virtual address to L2 cache access. The page size is 8 KB. The TLB is direct mapped with 256 entries. The L1 cache is a direct-mapped 8 KB and the L2 cache is a direct-mapped 4 MB. Both using 64 byte blocks. The virtual address is 64 bits and the physical address is 41 bits. The primary difference between this simple figure and a real cache, as in Figure 5.43 on page 472 , is replication of pieces of this figure.

First, the 64-bit virtual address is logically divided into a virtual page number and page offset. The former is sent to the TLB to be translated into a physical address, and the latter is sent to the L1 cache act as an index. If the TLB match is a hit, then the physical page number is sent to the L1 cache tag to check for a match. If it matches, its a L1 cache hit. The block offset then selects the word for the CPU.

If the L1 cache check results in a miss, the physical address is then used to try the L2 cache. The middle portion of the physical address is used as an index to the 4 MB L2 cache. The resulting L2 Cache Tag is compared to the upper part of the physical address to check for a match. If it matches, we have a L2 cache hit, and the data is sent to the CPU, which uses the block offset to select the desired word. On an L2 miss, the physical address is then used to get the block from memory.

Although this is a simple example, the major difference between this drawing and a real cache is replication. First, there is only one L1 cache. When there are two L1 caches, the top half of the diagram is duplicated. Note this would lead to two TLBs, which is typical. Hence, one cache and TLB is for instructions, driven from the PC, and one cache and TLB is for data, driven from the effective address. The second simplification is that all the caches and TLBs are direct mapped. If any were N-way set associative, then we would replicate each set of tag memory, comparators, and data memory N times and connect data memories with a N:1 multiplexor to select a hit. Of course, if the total cache size remained the same, the cache index would also shrink by N bits according to the formula in Figure 5.9 on page 397 .

5.11 Protection and Examples of Virtual Memory

The invention of multiprogramming, where a computer would be shared by several programs running concurrently, led to new demands for protection and sharing among programs. These demands are closely tied to virtual memory in computers today, and so we cover the topic here along with two examples of virtual memory.

Multiprogramming leads to the concept of a *process*. Metaphorically, a process is a program's breathing air and living space—that is, a running program plus any state needed to continue running it. Time-sharing is a variation of multiprogramming that shares the CPU and memory with several interactive users at the same time, giving the illusion that all users have their own computers. Thus, at any instant it must be possible to switch from one process to another. This exchange is called a *process switch* or *context switch*.

A process must operate correctly whether it executes continuously from start to finish, or is interrupted repeatedly and switched with other processes. The responsibility for maintaining correct process behavior is shared by designers of

the computer and the operating system. The computer designer must ensure that the CPU portion of the process state can be saved and restored. The operating system designer must guarantee that processes do not interfere with each others' computations.

The safest way to protect the state of one process from another would be to copy the current information to disk. However, a process switch would then take seconds—far too long for a time-sharing environment.

This problem is solved by operating systems partitioning main memory so that several different processes have their state in memory at the same time. This division means that the operating system designer needs help from the computer designer to provide protection so that one process cannot modify another. Besides protection, the computers also provide for sharing of code and data between processes, to allow communication between processes or to save memory by reducing the number of copies of identical information.

Protecting Processes

The simplest protection mechanism is a pair of registers that checks every address to be sure that it falls between the two limits, traditionally called *base* and *bound*. An address is valid if

$$\text{Base} \leq \text{Address} \leq \text{Bound}$$

In some systems, the address is considered an unsigned number that is always added to the base, so the limit test is just

$$(\text{Base} + \text{Address}) \leq \text{Bound}$$

If user processes are allowed to change the base and bounds registers, then users can't be protected from each other. The operating system, however, must be able to change the registers so that it can switch processes. Hence, the computer designer has three more responsibilities in helping the operating system designer protect processes from each other:

1. Provide at least two modes, indicating whether the running process is a user process or an operating system process. This latter process is sometimes called a *kernel* process, a *supervisor* process, or an *executive* process.
2. Provide a portion of the CPU state that a user process can use but not write. This state includes the base/bound registers, a user/supervisor mode bit(s), and the exception enable/disable bit. Users are prevented from writing this state because the operating system cannot control user processes if users can change the address range checks, give themselves supervisor privileges, or disable exceptions.
3. Provide mechanisms whereby the CPU can go from user mode to supervisor

mode and vice versa. The first direction is typically accomplished by a *system call*, implemented as a special instruction that transfers control to a dedicated location in supervisor code space. The PC is saved from the point of the system call, and the CPU is placed in supervisor mode. The return to user mode is like a subroutine return that restores the previous user/supervisor mode.

Base and bound constitute the minimum protection system, while virtual memory offers a more fine-grained alternative to this simple model. As we have seen, the CPU address must be mapped from virtual to physical address. This mapping provides the opportunity for the hardware to check further for errors in the program or to protect processes from each other. The simplest way of doing this is to add permission flags to each page or segment. For example, since few programs today intentionally modify their own code, an operating system can detect accidental writes to code by offering read-only protection to pages. This page-level protection can be extended by adding user/kernel protection to prevent a user program from trying to access pages that belong to the kernel. As long as the CPU provides a read/write signal and a user/kernel signal, it is easy for the address translation hardware to detect stray memory accesses before they can do damage. Such reckless behavior simply interrupts the CPU and invokes the operating system.

Processes are thus protected from one another by having their own page tables, each pointing to distinct pages of memory. Obviously, user programs must be prevented from modifying their page tables or protection would be circumvented.

Protection can be escalated, depending on the apprehension of the computer designer or the purchaser. Rings added to the CPU protection structure expand memory access protection from two levels (user and kernel) to many more. Like a military classification system of top secret, secret, confidential, and unclassified, concentric *rings* of security levels allow the most trusted to access anything, the second most trusted to access everything except the innermost level, and so on. The "civilian" programs are the least trusted and, hence, have the most limited range of accesses. There may also be restrictions on what pieces of memory can contain code—execute protection—and even on the entrance point between the levels. The Intel Pentium protection structure, which uses rings, is described later in this section. It is not clear whether rings are an improvement in practice over the simple system of user and kernel modes.

As the designer's apprehension escalates to trepidation, these simple rings may not suffice. Restricting the freedom given a program in the inner sanctum requires a new classification system. Instead of a military model, the analogy of this system is to keys and locks: A program can't unlock access to the data unless it has the key. For these keys, or *capabilities*, to be useful, the hardware and operating system must be able to explicitly pass them from one program to another without

allowing a program itself to forge them. Such checking requires a great deal of hardware support if time for checking keys is to be kept low.

A Paged Virtual Memory Example: The Alpha Memory Management and the 21264 TLB

The Alpha architecture uses a combination of segmentation and paging, providing protection while minimizing page table size. With 48-bit virtual addresses, the 64-bit address space is first divided into three segments: *seg0* (bits 63 - 47 = 0...00), *kseg* (bits 63 - 46 = 0...10), and *seg1* (bits 63 to 46 = 1...11). kseg is reserved for the operating system kernel, has uniform protection for the whole space, and does not use memory management. User processes use seg0, which is mapped into pages with individual protection. Figure 5.38 shows the layout of seg0 and seg1. seg0 grows from address 0 upward, while seg1 grows downward to 0. Many systems today use some such combination of predivided segments and paging. This approach provides many advantages: segmentation divides the address space and conserves page table space, while paging provides virtual memory, relocation, and protection.

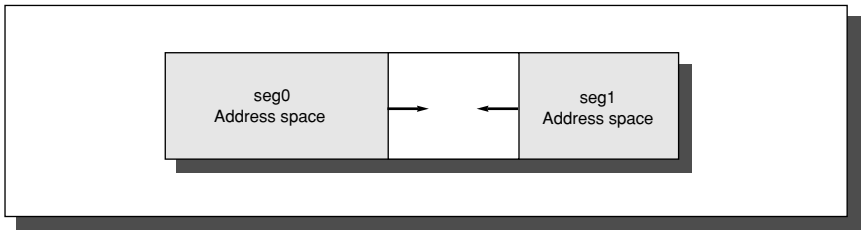


FIGURE 5.38 The organization of seg0 and seg1 in the Alpha. User processes live in seg0, while seg1 is used for portions of the page tables. seg0 includes a downward growing stack, text and data, and an upward growing heap.

Even with this division, the size of page tables for the 64-bit address space is alarming. Hence, the Alpha uses a three-level hierarchical page table to map the address space to keep the size reasonable. Figure 5.39 shows address translation in the Alpha. The addresses for each of these page tables come from three "level" fields, labeled level1, level2, and level3. Address translation starts with adding the level1 address field to the page table base register and then reading memory from this location to get the base of the second-level page table. The level2 address field is in turn added to this newly fetched address, and memory is accessed again to determine the base of the third page table. The level3 address field is added to this base address, and memory is read using this sum to (finally) get the physical address of the page being referenced. This address is concatenated with the page offset to get the full physical address. Each page table in the Alpha ar-

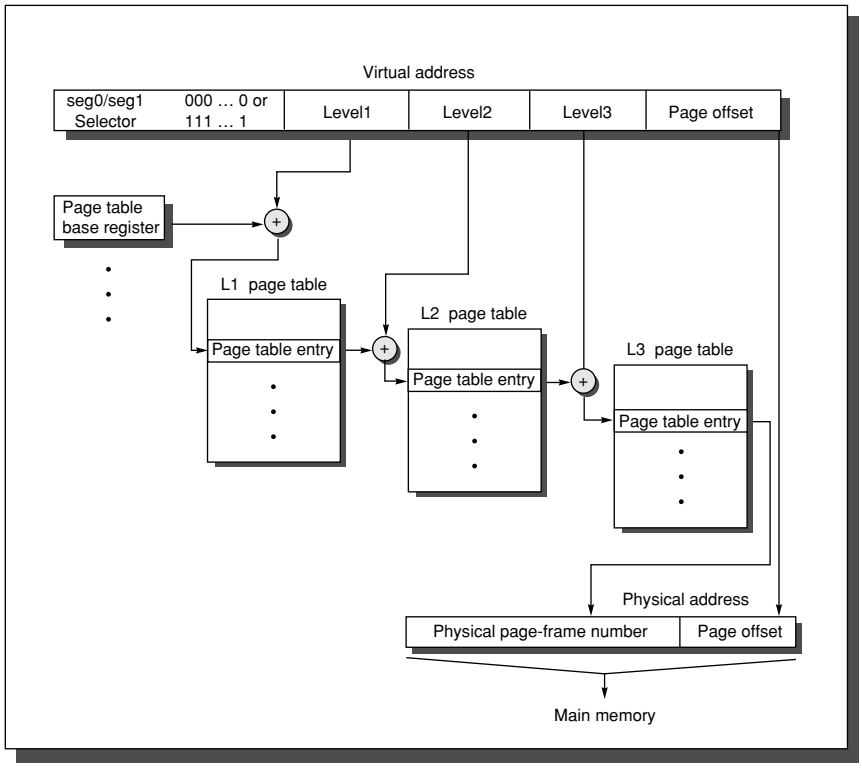


FIGURE 5.39 The mapping of an Alpha virtual address. This figure/description shows the 21264's virtual memory implementation with 3 page table levels, which supports an effective physical address size of 41 bits (allowing access up to 2^40 bytes of memory and 2^40 I/O addresses). Each page table is exactly one page long, so each level field is *n* bits wide where 2^n = page size/8. The Alpha architecture document allows the page size to grow from 8 KB in the current implementations to 16 KB, 32 KB, or 64 KB in the future. The virtual address for each page size grows from the original 43 bits to 47, 51, or 55 bits and the maximum physical address size grows from the current 41 bits to 45, 47, or 48 bits. The 21264 also can support a 4-level page table structure (with a level 0 page table field in virtual address bits 43-52) that can allow full access to its 44-bit physical address and 48-bit virtual address while keeping page sizes to 8 KB. That mode is not depicted here. In addition, the size depends on the operating system. VMS does not require KSEG like UNIX does, so VMS could reach the entire 44-bit physical address space with 3-level page tables. The 41-bit physical address restriction comes from the fact that some operating systems need the KSEG section.

chitecture is constrained to fit within a single page. The first three levels (0, 1, and 2) use physical addresses that need no further translation, but Level 3 is mapped virtually. These normally hit the TLB, but if not, the table is accessed a second time with physical addresses.

The Alpha uses a 64-bit *page table entry* (*PTE*) in each of these page tables. The first 32 bits contain the physical page frame number, and the other half includes the following five protection fields:

- *Valid*—Says that the page frame number is valid for hardware translation
- *User read enable*—Allows user programs to read data within this page
- *Kernel read enable*—Allows the kernel to read data within this page
- *User write enable*—Allows user programs to write data within this page
- *Kernel write enable*—Allows the kernel to write data within this page

In addition, the PTE has fields reserved for systems software to use as it pleases. Since the Alpha goes through three levels of tables on a TLB miss, there are three potential places to check protection restrictions. The Alpha obeys only the bottom-level PTE, checking the others only to be sure the valid bit is set.

Since the PTEs are 8 bytes long, the page tables are exactly one page long, and the Alpha 21264 has 8-KB pages, each page table has 1024 PTEs. Each of the three level fields are 10 bits long and the page offset is 13 bits. This derivation leaves $64 - (3 \times 10 + 13)$ or 21 bits to be defined. If this is a seg0 address, the most-significant bit of the level 1 field is a 0, and for seg1 the two most-significant bits of the level 1 field are 11_{two}. Alpha requires all bits to the left of the level1 field to be identical. For seg0 these 21 bits are all zeros and for seg1 they are all ones. This restriction means the 21264 virtual addresses are really much shorter than the full 64 bits found in registers.

The maximum virtual address and physical address is then tied to the page size. The original architecture document allows for the Alpha to expand the minimum page size from 8 KB up to 64 KB, thereby increasing the virtual address to $3 \times 13 + 16$ or 55 bits and the maximum physical address to 32 + 16 or 48 bits. In fact, the upcoming 21364 supports both. It will be interesting to see whether or not operating systems accommodate such expansion plans.

Although we have explained translation of legal addresses, what prevents the user from creating illegal address translations and getting into mischief? The page tables themselves are protected from being written by user programs. Thus, the user can try any virtual address, but by controlling the page table entries the operating system controls what physical memory is accessed. Sharing of memory between processes is accomplished by having a page table entry in each address space point to the same physical memory page.

The Alpha 21264 employs two TLBs to reduce address translation time, one for instruction accesses and another for data accesses. Figure 5.40 shows the important parameters. The Alpha allows the operating system to tell the TLB that contiguous sequences of pages can act as one: the options are 8, 64, and 512 times the minimum page size. Thus, the variable page size of a PTE mapping makes the match more challenging, as the size of the space being mapped in the PTE also must be checked to determine the match. Figure 5.36 above describes the data TLB.

Parameter	Description
Block size	1 PTE (8 bytes)
Hit time	1 clock cycle
Miss penalty (average)	20 clock cycles
TLB size	Same for Instruction and Data TLBs: 128 PTEs per TLB, each of which can map 1, 8, 64, or 512 pages
Block selection	Round robin
Write strategy	(Not applicable)
Block placement	Fully associative

FIGURE 5.40 Memory-hierarchy parameters of the Alpha 21264 TLB.

Memory management in the Alpha 21264 is typical of most desktop or server computers today, relying on page-level address translation and correct operation of the operating system to provide safety to multiple processes sharing the computer. In the next section we see a protection scheme for individuals who want to trust the operating system as little as possible.

A Segmented Virtual Memory Example: Protection in the Intel Pentium

The second system is the most dangerous system a man ever designs… . The general tendency is to over-design the second system, using all the ideas and frills that were cautiously sidetracked on the first one.

F. P. Brooks, Jr., *The Mythical Man-Month* (1975)

The original 8086 used segments for addressing, yet it provided nothing for virtual memory or for protection. Segments had base registers but no bound registers and no access checks, and before a segment register could be loaded the corresponding segment had to be in physical memory. Intel's dedication to virtual memory and protection is evident in the successors to the 8086 (today called IA-32), with a few fields extended to support larger addresses. This protection scheme is elaborate, with many details carefully designed to try to avoid security loopholes. The next few pages highlight a few of the Intel safeguards; if you find the reading difficult, imagine the difficulty of implementing them!

The first enhancement is to double the traditional two-level protection model: the Pentium has four levels of protection. The innermost level (0) corresponds to Alpha kernel mode and the outermost level (3) corresponds to Alpha user mode. The IA-32 has separate stacks for each level to avoid security breaches between the levels. There are also data structures analogous to Alpha page tables that con-

tain the physical addresses for segments, as well as a list of checks to be made on translated addresses.

The Intel designers did not stop there. The IA-32 divides the address space, allowing both the operating system and the user access to the full space. The IA-32 user can call an operating system routine in this space and even pass parameters to it while retaining full protection. This safe call is not a trivial action, since the stack for the operating system is different from the user's stack. Moreover, the IA-32 allows the operating system to maintain the protection level of the *called* routine for the parameters that are passed to it. This potential loophole in protection is prevented by not allowing the user process to ask the operating system to access something indirectly that it would not have been able to access itself. (Such security loopholes are called *Trojan horses*.)

The Intel designers were guided by the principle of trusting the operating system as little as possible, while supporting sharing and protection. As an example of the use of such protected sharing, suppose a payroll program writes checks and also updates the year-to-date information on total salary and benefits payments. Thus, we want to give the program the ability to read the salary and year-to-date information, and modify the year-to-date information but not the salary. We shall see the mechanism to support such features shortly. In the rest of this subsection, we will look at the big picture of the IA-32 protection and examine its motivation.

Adding Bounds Checking and Memory Mapping

The first step in enhancing the Intel processor was getting the segmented addressing to check bounds as well as supply a base. Rather than a base address, as in the 8086, segment registers in the IA-32 contain an index to a virtual memory data structure called a *descriptor table*. Descriptor tables play the role of page tables in the Alpha. On the IA-32 the equivalent of a page table entry is a *segment descriptor*. It contains fields found in PTEs:

- A *present bit*—equivalent to the PTE valid bit, used to indicate this is a valid translation
- A *base field*—equivalent to a page frame address, containing the physical address of the first byte of the segment
- An *access bit*—like the reference bit or use bit in some architectures that is helpful for replacement algorithms
- An *attributes field*—specifies the valid operations and protection levels for operations that use this segment

There is also a *limit field*, not found in paged systems, which establishes the upper bound of valid offsets for this segment. Figure 5.41 shows examples of IA-32 segment descriptors.

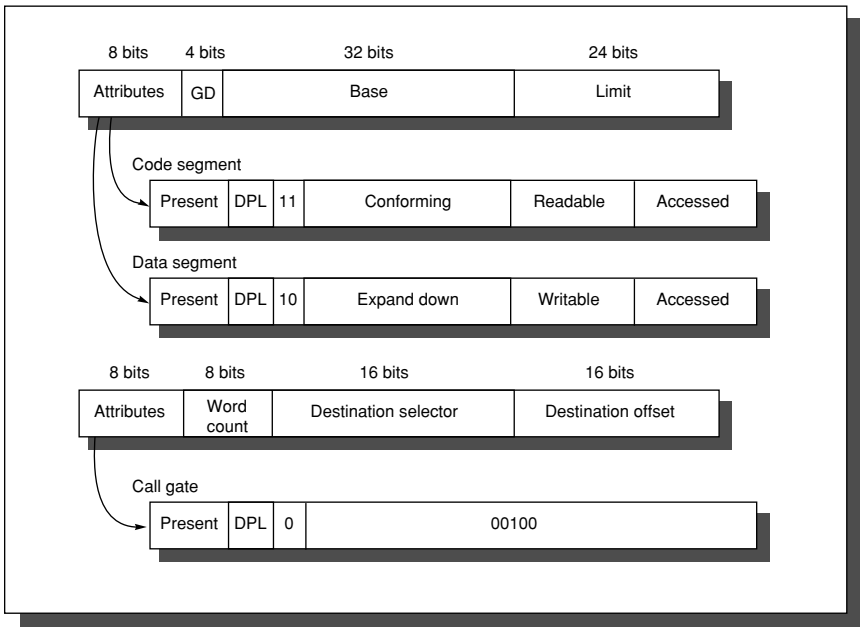


FIGURE 5.41 The IA-32 segment descriptors are distinguished by bits in the attributes field. *Base*, *limit, present*, *readable*, and *writable* are all self-explanatory. D gives the default addressing size of the instructions: 16 bits or 32 bits. G gives the granularity of the segment limit: 0 means in bytes and 1 means in 4-KB pages. G is set to 1 when paging is turned on to set the size of the page tables. DPL means *descriptor privilege level*—this is checked against the code privilege level to see if the access will be allowed. *Conforming* says the code takes on the privilege level of the code being called rather than the privilege level of the caller; it is used for library routines. The *expand-down field* flips the check to let the base field be the high-water mark and the limit field be the low-water mark. As one might expect, this is used for stack segments that grow down. *Word count* controls the number of words copied from the current stack to the new stack on a call gate. The other two fields of the call gate descriptor, *destination selector* and *destination offset*, select the descriptor of the destination of the call and the offset into it, respectively. There are many more than these three segment descriptors in the IA-32 protection model.

IA-32 provides an optional paging system in addition to this segmented addressing. The upper portion of the 32-bit address selects the segment descriptor and the middle portion is an index into the page table selected by the descriptor. We describe below the protection system that does not rely on paging.

Adding Sharing and Protection

To provide for protected sharing, half of the address space is shared by all processes and half is unique to each process, called *global address space* and *local address space*, respectively. Each half is given a descriptor table with the appropriate name. A descriptor pointing to a shared segment is placed in the global

descriptor table, while a descriptor for a private segment is placed in the local descriptor table.

A program loads a IA-32 segment register with an index to the table *and* a bit saying which table it desires. The operation is checked according to the attributes in the descriptor, the physical address being formed by adding the offset in the CPU to the base in the descriptor, provided the offset is less than the limit field. Every segment descriptor has a separate 2-bit field to give the legal access level of this segment. A violation occurs only if the program tries to use a segment with a lower protection level in the segment descriptor.

We can now show how to invoke the payroll program mentioned above to update the year-to-date information without allowing it to update salaries. The program could be given a descriptor to the information that has the writable field clear, meaning it can read but not write the data. A trusted program can then be supplied that will only write the year-to-date information. It is given a descriptor with the writable field set (Figure 5.41). The payroll program invokes the trusted code using a code segment descriptor with the conforming field set. This setting means the called program takes on the privilege level of the code being called rather than the privilege level of the caller. Hence, the payroll program can read the salaries and call a trusted program to update the year-to-date totals, yet the payroll program cannot modify the salaries. If a Trojan horse exists in this system, to be effective it must be located in the trusted code whose only job is to update the year-to-date information. The argument for this style of protection is that limiting the scope of the vulnerability enhances security.

Adding Safe Calls from User to OS Gates and Inheriting Protection Level for Parameters

Allowing the user to jump into the operating system is a bold step. How, then, can a hardware designer increase the chances of a safe system without trusting the operating system or any other piece of code? The IA-32 approach is to restrict where the user can enter a piece of code, to safely place parameters on the proper stack, and to make sure the user parameters don't get the protection level of the called code.

To restrict entry into others' code, the IA-32 provides a special segment descriptor, or *call gate*, identified by a bit in the attributes field. Unlike other descriptors, call gates are full physical addresses of an object in memory; the offset supplied by the CPU is ignored. As stated above, their purpose is to prevent the user from randomly jumping anywhere into a protected or more-privileged code segment. In our programming example, this means the only place the payroll program can invoke the trusted code is at the proper boundary. This restriction is needed to make conforming segments work as intended.

What happens if caller and callee are "mutually suspicious," so that neither trusts the other? The solution is found in the word count field in the bottom descriptor in Figure 5.41. When a call instruction invokes a call gate descriptor, the descriptor copies the number of words specified in the descriptor from the local

stack onto the stack corresponding to the level of this segment. This copying allows the user to pass parameters by first pushing them onto the local stack. The hardware then safely transfers them onto the correct stack. A return from a call gate will pop the parameters off both stacks and copy any return values to the proper stack. Note that this model is incompatible with the current practice of passing parameters in registers.

This scheme still leaves open the potential loophole of having the operating system use the user's address, passed as parameters, with the operating system's security level, instead of with the user's level. The IA-32 solves this problem by dedicating 2 bits in every CPU segment register to the *requested protection level*. When an operating system routine is invoked, it can execute an instruction that sets this 2-bit field in all address parameters with the protection level of the user that called the routine. Thus, when these address parameters are loaded into the segment registers, they will set the requested protection level to the proper value. The IA-32 hardware then uses the requested protection level to prevent any foolishness: No segment can be accessed from the system routine using those parameters if it has a more-privileged protection level than requested.

Summary: Protection on the Alpha versus the IA-32

If the IA-32 protection model looks harder to build than the Alpha model, that's because it is. This effort must be especially frustrating for the IA-32 engineers, since few customers use the elaborate protection mechanism. In addition, the fact that the protection model is a mismatch for the simple paging protection of UNIX-like systems means it will be used only by someone writing an operating system especially for this computer.

In the last edition we wondered whether the popularity of the Internet would lead to demands for increased support for security, and hence put this elaborate protection model to good use. Despite widely documented security breaches and the ubiquity of this architecture, no one has proposed a new operating system to leverage the 80x86 protection features.

5.12 Crosscutting Issues in the Design of Memory Hierarchies

This section describes four topics discussed in other chapters that are fundamental to memory-hierarchy design.

Superscalar CPU and Number of Ports to the Cache

One complexity of the advanced designs of Chapters 3 and 4 is that multiple instructions can be issued within a single clock cycle. Clearly, if there is not sufficient peak bandwidth from the cache to match the peak demands of the instructions, there is little benefit to designing such parallelism in the processor.

Some processors increase complexity of instruction fetch by allowing instructions to be issued to be found on any boundary instead of, say, a multiple of four words. As mentioned above, similar reasoning applies to CPUs that want to continue executing instructions on a cache miss: clearly the memory hierarchy must also be nonblocking or the CPU cannot benefit.

For example, the UltraSPARC III fetches up to 4 instructions per clock cycle, and executes up to 4, with up to 2 being loads or stores. Hence, the instruction cache must deliver 128 bits per clock cycle and the data cache must support two 64-bit accesses per clock cycle.

Speculative Execution and the Memory System

Inherent in CPUs that support speculative execution or conditional instructions is the possibility of generating invalid addresses that would not occur without speculative execution. Not only would this be incorrect behavior if exceptions were taken, the benefits of speculative execution would be swamped by false exception overhead. Hence, the memory system must identify speculatively executed instructions and conditionally executed instructions and suppress the corresponding exception.

By similar reasoning, we cannot allow such instructions to cause the cache to stall on a miss, for again unnecessary stalls could overwhelm the benefits of speculation. Hence, these CPUs must be matched with nonblocking caches (see page 421).

In reality, the penalty of the an L2 miss is so large that compilers normally only speculate on L1 misses. Figure 5.23 on page 423 shows that for some well-behaved scientific programs the compiler can sustain multiple outstanding L2 misses (“miss under miss”) so as to effectively cut the L2 miss penalty. Once again, for this to work the memory system behind the cache must match the desires of the compiler in number of simultaneous memory accesses.

Combining the Instruction Cache with Instruction Fetch and Decode Mechanisms

With Moore’s Law continuing to offer more transistors and increasing demands for instruction level parallelism and clock rate, increasingly the instruction cache and first part of instruction execution are merging (see Chapter 3).

The leading example is the Netburst microarchitecture of the Pentium 4 and its successors. Not only does it use a trace cache (see page 434), which combines branch prediction with instruction fetch, it stores the internal RISC operations (see Chapter 3) in the trace cache. Hence, cache hits save 5 of 25 pipeline stages for decoding and translation. The downside of caching decoded instructions is impact on die size. It appears on the die that the 12000 RISC operations in the trace cache take equivalent of 96 KB of SRAM, which suggests that the RISC operations are about 64-bits long. 80x86 instructions would surely be two to three times more efficient.

Embedded computers also have bigger instructions in the cache, but for another reason. Given the importance of code size for such applications, several keep a compressed version of the instruction in main memory and then expand to the full size in the instruction cache (see page 130 in Chapter 2.)

Embedded Computer Caches and Real Time Performance

As mentioned before, embedded computers often are placed in real time environments where a set of tasks must be completed every time period. In such situations performance variability is of more concern than average case performance. Since caches were invented to improve average case performance at the cost of greater variability, they would seem to be a problem for real time computing.

In practice, instruction caches are widely used in embedded computers since most code has predictable behavior. Data caches then are the real issue.

To cope with that challenge, some embedded computers allow a portion of the cache to be "locked down." That is, a portion of the cache acts like a small scratchpad memory under program control. In a set associative data cache, one block of an entry would be locked down while the others could still buffer accesses to main memory. If it was direct mapped, then every address that maps onto that locked down block would result in a miss and later is passed to the CPU.

Embedded Computer Caches and Power

Although caches were invented to reduce memory access time, they also save power. It is much more power efficient to access on chip memory than it is to drive the pins of the chip, drive the memory bus, activate the external memory chips and then make the return trip.

To further improve power efficiency of caches on chip, some of the optimizations in sections 5.4 to 5.7 are reoriented for power. For example, the MIPS 4300 uses way prediction to only power half of the address checking hardware for its two-way set associative cache.

I/O and Consistency of Cached Data

Because of caches, data can be found in memory and in the cache. As long as the CPU is the sole device changing or reading the data and the cache stands between the CPU and memory, there is little danger in the CPU seeing the old or *stale* copy. I/O devices give the opportunity for other devices to cause copies to be inconsistent or for other devices to read the stale copies. Figure 5.42 illustrates the problem, generally referred to as the *cache-coherency* problem.

The question is this: Where does the I/O occur in the computer—between the I/O device and the cache or between the I/O device and main memory? If input puts data into the cache and output reads data from the cache, both I/O and the CPU see the same data, and the problem is solved. The difficulty in this approach

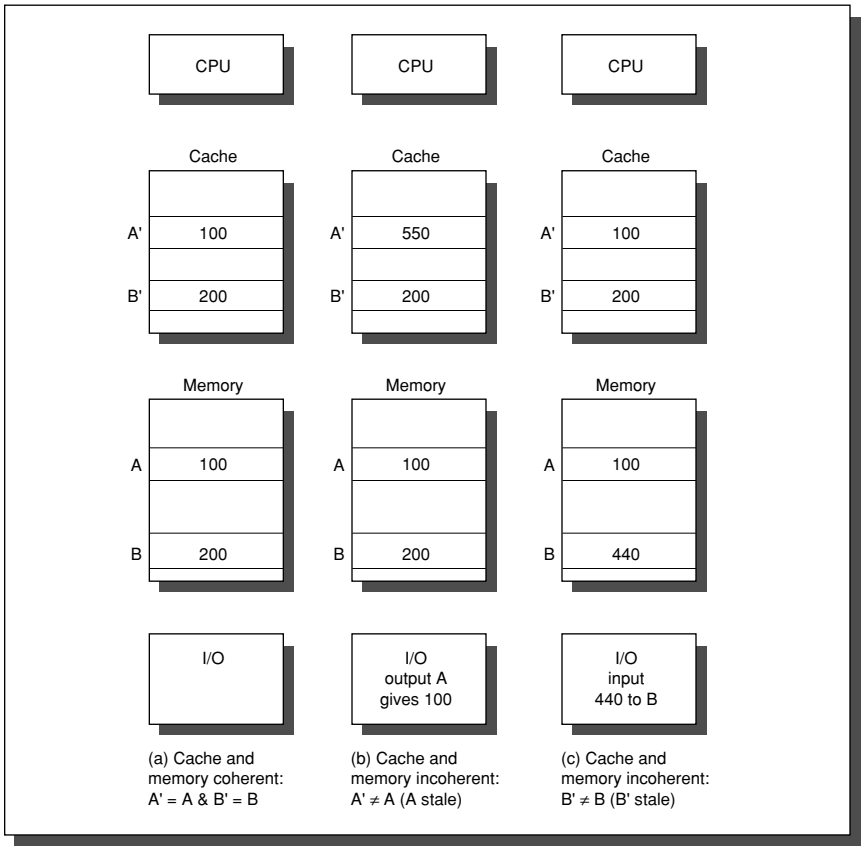


FIGURE 5.42 The cache-coherency problem. A' and B' refer to the cached copies of A and B in memory. (a) shows cache and main memory in a coherent state. In (b) we assume a write-back cache when the CPU writes 550 into A. Now A' has the value but the value in memory has the old, stale value of 100. If an output used the value of A from memory, it would get the stale data. In (c) the I/O system inputs 440 into the memory copy of B, so now B' in the cache has the old, stale data.

is that it interferes with the CPU. I/O competing with the CPU for cache access will cause the CPU to stall for I/O. Input may also interfere with the cache by displacing some information with new data that is unlikely to be accessed soon. For example, on a page fault the CPU may need to access a few words in a page, but a program is not likely to access every word of the page if it were loaded into the cache. Given the integration of caches onto the same integrated circuit, it is also difficult for that interface to be visible.

The goal for the I/O system in a computer with a cache is to prevent the stale-data problem while interfering with the CPU as little as possible. Many systems,

therefore, prefer that I/O occur directly to main memory, with main memory acting as an I/O buffer. If a write-through cache were used, then memory would have an up-to-date copy of the information, and there would be no stale-data issue for output. (This benefit is a reason processors used write through.) Alas, write-through usually found only today in first level data caches backed by a L2 cache which uses write back. Even embedded caches avoid write through for reasons of power efficiency

Input requires some extra work. The software solution is to guarantee that no blocks of the I/O buffer designated for input are in the cache. In one approach, a buffer page is marked as noncachable; the operating system always inputs to such a page. In another approach, the operating system flushes the buffer addresses from the cache after the input occurs. A hardware solution is to check the I/O addresses on input to see if they are in the cache. To avoid slowing down the cache to check addresses, a duplicate set of tags may be used to allow checking of I/O addresses in parallel with processor cache accesses. If there is a match of I/O addresses in the cache, the cache entries are invalidated to avoid stale data. All these approaches can also be used for output with write-back caches. More about this is found in Chapter 7.

The cache-coherency problem applies to multiprocessors as well as I/O. Unlike I/O, where multiple data copies are a rare event—one to be avoided whenever possible—a program running on multiple processors will *want* to have copies of the same data in several caches. Performance of a multiprocessor program depends on the performance of the system when sharing data. The protocols to maintain coherency for multiple processors are called *cache-coherency protocols,* and are described in Chapter 6.

5.13 Putting It All Together: Alpha 21264 Memory Hierarchy

Thus far we have given glimpses of the Alpha 21264 memory hierarchy; this section unveils the full design and shows the performance of its components for the SPEC95 programs. Figure 5.43 gives the overall picture of this design. The 21264 is an out-of-order execution processor that fetches up to four instructions per clock cycle and executes up to six instructions per clock cycle. It uses either a 48-bit virtual address and a 44-bit physical address or 43-bit virtual address and 41-bit physical; thus far, all systems just use 41 bits. In either case, Alpha halves the physical address space, with the lower half for memory addresses and the upper half for I/O addresses. For the rest of this section, we assume use of the 43-bit virtual address and the 41-bit physical address.

Let's really start at the beginning, when the Alpha is turned on. Hardware on the chip loads the instruction cache serially from an external PROM. This initialization fills up to 64-KB worth of instructions (16K instructions) into the cache. The same serial interface (and PROM) also loads configuration information that specifies L2 cache speed/timing, system port speed/timing, and much other infor-

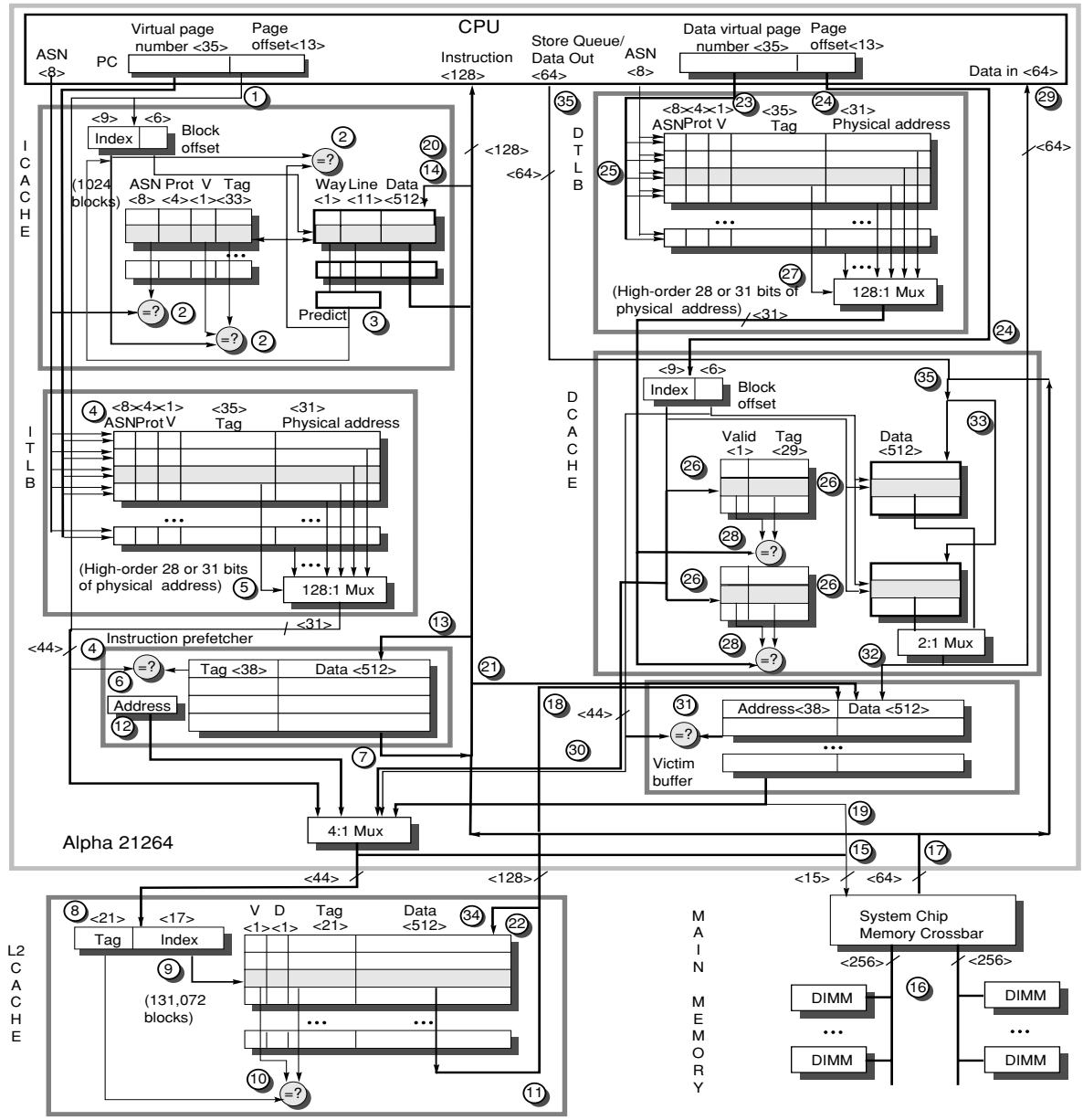


FIGURE 5.43 The overall picture of the Alpha 21264 memory hierarchy. Individual components can be seen in greater detail in Figures 5.7 (page 388) and 5.36 (page 454). The instruction cache is virtually indexed and tagged, but the data cache has virtual index but physical tags. Hence, every data address must be sent to the data TLB at the same time as it is sent to the data cache. Both the instruction and data TLB's have 128 entries. Each TLB entry can map a page of size 8KB, 64KB, 512KB, or 4MB. The 21264 supports a 48-bit or 43-bit virtual address and a 44-bit or 41-bit physical address.

mation necessary for the 21264 to communicate with the external logic. This code completes the remainder of the processor and system initialization.

The preloaded instructions execute in Privileged Architecture Library (PAL) mode. The software executed in PAL mode is simply machine language routines with some implementation-specific extensions to allow access to low-level hardware, such as the TLB. PAL code runs with exceptions disabled, and the instruction addresses are not translated. Since PAL code avoids the TLB, instruction accesses are not checked for memory management violations.

One of the first steps is to update the instruction TLB with valid page table entries (PTEs) for this process. Kernel code updates the appropriate page table entry (in memory) for each page to be mapped. A miss in the TLB is handled by PAL code, since normal code that relies on the TLB cannot change the TLB.

Once the operating system is ready to begin executing a user process, it sets the PC to the appropriate address in segment seg0.

We are now ready to follow memory hierarchy in action: Figure 5.43 is labeled with the steps of this narrative. First, a 12-bit address is sent to the 64-KB instruction cache, along with a 35-bit page number. An 8-bit Address Space Number (ASN) is also sent, for the same purpose as using ASN's in the TLB (step 1).The instruction cache is virtually indexed and virtually tagged, so instruction TLB translations are only required on cache misses. As mentioned in section 5.5, the Instruction cache uses way prediction, so a 1-bit way predict bit is prepended to the 9-bit index. The effective index is then 10 bits, similar to a 64-KB direct mapped cache with 1024 blocks. Thus, the effective instruction cache index is 10 bits (see page 387), and the instruction cache tag is then 48 – 9 bits (actual index) – 6 bits (block offset) or 33 bits. As the 21264 expects 4 instructions (16 bytes) each instruction fetch, an additional 2 bits is used from the 6-bit block offset to select the appropriate 16 bytes. Hence, 10 + 2 or 12 bits to read 16 bytes of instructions.

To reduce latency, the instruction cache includes two mechanisms to begin early access of the next block. As mentioned in section 5.5, the way predicting cache relies on a 1-bit field for every 16 bytes to predict which of two sets will be used next, offering the hit time of direct mapped with miss rate of two-way associativity. It also includes 11 bits to predict the next group of 16 bytes to be read. This field is loaded with the address of the next sequential group on a cache miss, and updated to a nonsequential address by the dynamic branch predictor. These two techniques are called *way prediction* and *line prediction*.

Thus, the index field of the PC is compared to the predicted block address, the tag field of the PC is compared to the address from the tag portion of the cache, and the 8-bit process ASN to the tag ASN field (step 2). The valid bit is also checked. If any field has the wrong value, it is a miss. On a hit in the instruction cache, the proper fetch block is supplied, and the next way and line prediction is loaded to read the next block (step 3). There is also a protection field in the tag, to

ensure that instruction fetch does not violate protection barriers. The instruction stream access is now done.

An instruction cache miss causes a simultaneous check of the instruction TLB and the instruction prefetcher (step 4). The fully associative TLB simultaneously searches all 128 entries to find a match between the address and a valid PTE (step 5). In addition to translating the address, the TLB checks to see if the PTE demands that this access result in an exception, and if the address space number of the processor matches the address space number in the field. An exception might occur if either this access violates the protection on the page or if the page is not in main memory. If the desired instruction address is found in the instruction prefetcher (step 6), the instructions are (eventually) supplied directly by the prefetcher (step 7). Otherwise, if there is no TLB exception, an access to the second-level cache is started (step 8).

In the case of a complete miss, the second-level cache continues trying to fetch the block. The 21264 microprocessor is designed to work with direct-mapped second-level caches from 1MB to 16 MB. For this section we use the memory system of the 667 Mhz Compaq AlphaserverES40, a shared memory system with from 1 to 4 processors. It has a 444 Mhz, 8-MB direct-mapped, second-level cache. (The data rate is 444 Mhz; the L2 SRAM parts use the double data rate technique of DRAMs, so they are clocked at only half that rate.)The L2 index is

$$2^{\text{index}} = \frac{\text{Cache size}}{\cancel{\text{Block size} \times \text{Set associativity}}} = \frac{8192K}{\cancel{64 \times 1}} = 128K = 2^{17}$$

so the 35-bit block address (41-bit physical address – 6-bit block offset) is divided into a 18-bit tag and a 17-bit index (step 9).The cache controller reads the tag from that index and if it matches and is valid (step 10), it returns the critical 16 bytes (step 11), with the remaining 48 bytes of the cache block supplied 16 bytes per 2.25 ns. The 21264 L2 interface does not require that the L2 cache clock be an integer multiple of the processor clock, so it can be loaded faster than 3.00 ns that you might expect from a 667 MHz processor. At the same time, a request is made for the next sequential 64-byte block (step 12), which is loaded into the instruction prefetcher in the next 6 clock cycles (step 13). Each miss can cause a prefetch of up to 4 cache blocks. An instruction cache miss costs approximately 15 CPU cycles (22 ns), depending on clock alignments.

By the way, the instruction prefetcher does not rely on the TLB for address translation. It simply increments the physical address of the miss by 64 bytes, checking to make sure that the new address is within the same page. If the incremented address crosses a page boundary, then the prefetch is suppressed. To save time, the prefetched instructions are passed around the CPU and then written to the instruction cache while the instructions execute in the CPU (step 14).

If the instruction is not found in the secondary cache, the physical address command is sent to the ES40 system chip set via four consecutive transfer cycles

on a narrow, 15-bit outbound address bus (step 15). The address and command use the address bus for 8 CPU cycles. The ES40 connects the microprocessor to memory via a crossbar to one of two 256-bit memory busses to service the request (step 16). Each bus contains a variable number of DIMMs (dual inline memory modules). The size and number of DIMMs can vary to give a total of 32 GB of memory in the 667 Mhz ES40. Since the 21264 provides single error correction/double error detection checking on data cache (see section 5.15), L2 cache, busses, and main memory, the data busses actually include an additional 32-bits for ECC bits.

Although the crossbar has two 256-bit buses, the path to the microprocessor is much narrower. 64 data bits. Thus, the 21264 has two off-chip paths: 128 data bits for the L2 cache and 64 data bits for memory crossbar. Separate paths allows a point-to-point connection and hence a high clock rate interface for both the L2 cache and the crossbar.

The total latency of the instruction miss that is serviced by main memory is approximately 130 CPU cycles for the critical instructions. The system logic fills the remainder of the 64-byte cache block at a rate of 8 bytes per 2 CPU cycles (step 17).

Since the second-level cache is a write-back cache, any miss can lead to an old block being written back to memory. The 21264 places this "victim" block into a victim file (step 18), as it does with a victim dirty block in the data cache, to get out of the way of new data when the new cache reference determined first read the L2 cache; that is, the original instruction fetch read that missed (step 8). The 21264 sends out the address of the victim out the system address bus following the address of the new request (step 19). The system chip set later extracts the victim data and writes it to the memory DIMMs.

The new data are loaded into the instruction cache as soon as they arrive (step 20). It also goes into a (L1) victim buffer (step 21), and is later written to the L2 cache (step 22). The victim buffer is of size 8, so many victims can be queued before being written back either to the L2 or to memory. The 21264 can also manage up to 8 simultaneous cache block misses, allowing it to hit under 8 misses as described in section 5.4.

If this initial instruction is a load, the data addresses is also sent to the data cache. It is 64 KB, 2-way set-associative, and write-back with a 64-byte block size. Unlike the instruction cache, the data cache is virtually indexed and *physically* tagged. Thus, the page frame of the instruction's data address is sent to the data TLB (step 23) at the same time as the (9+3)-bit index from the virtual address is sent to the data cache (step 24). The data TLB is a fully associative cache containing 128 PTEs (step 25), each of which represents page sizes from 8 KB to 4 MB. A TLB miss will trap to PAL code to load the valid PTE for this address. In the worst case, the page is not in memory, and the operating system gets the page from disk, just as before. Since millions of instructions could execute during a page fault, the operating system will swap in another process if one is waiting to run.

The index field of the address is sent of both sets of the data cache (step 26). Assuming that we have a valid PTE in the data TLB (step 27), the two tags and valid bits are compared to the physical page frame (step 28), with a match sending the desired 8 bytes to the CPU (step 29). A miss goes to the second-level cache, which proceeds similar to an instruction miss (Step 30), except that it must check the victim buffer first to be sure the block is not there (Step 31).

As mentioned in section 5.7, the data cache can be virtually addressed and physically tagged. On a miss, the cache controller must check for a synonym (two different virtual addresses that reference the same physical address). Hence, the data cache tags are examined in parallel with the L2 cache tags during an L2 lookup. As the minimum page size is 8 KB or 13 bits and the cache index plus block offset is 15 bits, the cache must check 2^2 or 4 blocks per set for synonyms. If it finds a synonym, the offending block is invalidated. This guarantees that a cache block can reside in one of the 8 possible data cache locations at any given time.

A write back victim can be produced on a data cache miss. The victim data is extracted from the data cache simultaneously with the fill of the data cache with the L2 data, and sent to the victim buffer (Step 32). I

In the case of an L2 miss, the fill data from the system is written directly into the (L1) data cache (step 33). The L2 is written only with L1 victims (step 34). They appear either because they were modified by the CPU, or because they had been loaded from memory directly into the data cache but not yet written into the L2 cache.

Suppose the instruction is a store instead of a load. When the store issues it does a data cache lookup just like a load. A store miss causes the block to be filled into the data cache very much as with a load miss. The store does not update the cache until later, after it is known to be non-speculative. During this time the store resides in a store queue, part of the out-of-order control mechanism of the CPU. Stores write from the store queue into the data cache on idle cache cycles (step 35). The data cache is ECC protected, so a read-modify-write operation is required to update the data cache on stores.

Performance of the 21264 Memory Hierarchy

How well does the 21264 work? The bottom line in this evaluation is the percentage of time lost while the CPU is waiting for the memory hierarchy. The major components are the instruction and data caches, instruction and data TLBs, and the secondary cache. Alas, in an out-of-order execution processors like the 21264 it is very hard to isolate the time waiting for memory, since a memory stall for one instruction may be completely hidden by successful completion of a later instruction.

How well does out-or-order perform compared in in-order? Figure 5.44 shows relative performance for SPECint2000 benchmarks for the out-of-order 21264 and its predecessor, the in-order Alpha 21164. The clock rates are similar in the

figure, but other differences in addition include the on-chip caches (two 64 KB L1 caches vs. two 8 KB L1 caches plus one 96 KB L2 cache). The miss rate for the 21164 on-chip L2 cache is also plotted in the figure along with the miss rate

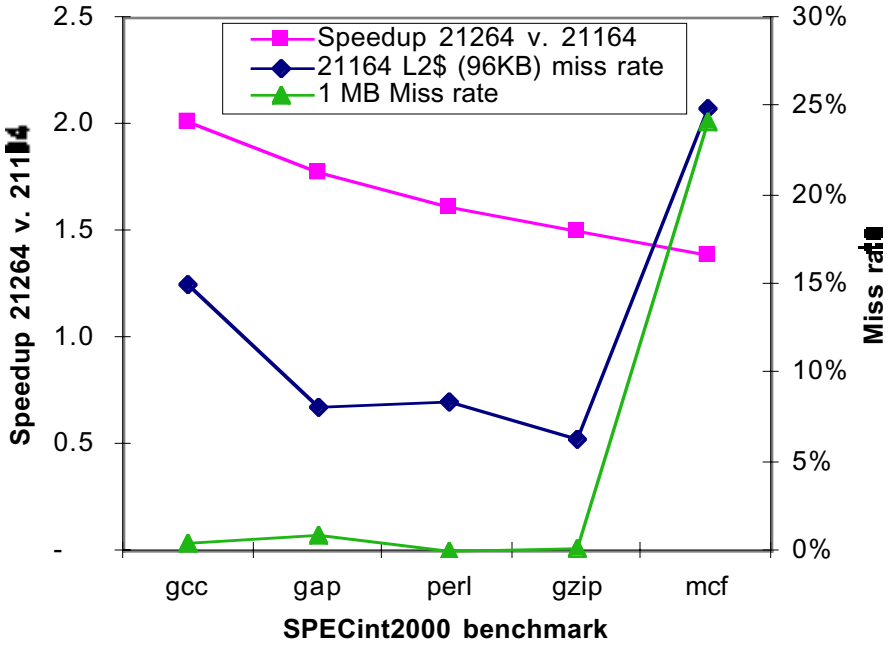


FIGURE 5.44 Alpha 21264/21164 Performance speedup vs. miss rates for SPECint2000. The left axis shows the speedup of the out-of-order 21264 is greatest with the highest miss rate of the 21164 L2 cache (right axis) as long as the access is a hit in the 21264 L2 cache. If it misses the L2 cache of the 21264, out-of-order execution is not as helpful. The 21264 is running at 500 MHz and the earlier 21164 is running at 533MHz.

of a 1MB cache. Figure 5.44 shows that speedup generally tracks its miss rate; the higher the 21164 miss rate, the higher the speedup of the 21264 over the 21164. The only exception is MCF, which is also the only program with a high miss rate for the a 1MB cache.

This result is likely explained by the 21264's ability to continue to execute during caches misses which stall the 21164 but hit in the L2 cache of the 21264. If the miss also misses in the L2 cache, then the 21264 must also stall, hence the lower speedup for MCF.

Figure 5.45 shows the CPI and various misses per 1000 instructions for a benchmark similar to TPC-C on a commercial database and the SPEC95 programs. Clearly, the SPEC95 programs do not tax the 21264 memory hierarchy, with instruction cache misses per instruction of 0.001% to 0.343% and second-level cache misses per instruction of 0.001% to 1%. The commercial benchmark

Program	CPI	Cache misses per 1000 instructions		TLB misses per 1000 instructions
		I cache	L2 cache	D TLB
TPC-C like	2.23	11.15	7.30	1.21
go	0.58	0.53	0.00	0.00
m88ksim	0.38	0.16	0.04	0.01
gcc	0.63	3.43	0.25	0.30
compress	0.70	0.00	0.40	0.00
li	0.49	0.07	0.00	0.01
ijpeg	0.49	0.03	0.02	0.01
perl	0.56	1.66	0.09	0.26
vortex	0.58	1.19	0.63	1.98
Avg. SPECint95	0.55	0.88	0.18	0.03
tomcatv	0.52	0.01	5.16	0.12
swim	0.40	0.00	5.99	0.10
su2cor	0.59	0.03	1.64	0.11
hydro2d	0.64	0.01	0.46	0.19
mgrid	0.44	0.02	0.05	0.10
applu	0.94	0.01	10.20	0.18
turb3d	0.44	0.01	1.60	0.10
apsi	0.67	0.05	0.01	0.04
fpppp	0.52	0.13	0.00	0.00
wave5	0.74	0.07	1.72	0.89
Avg. SPECfp95	0.59	0.03	2.68	0.09

FIGURE 5.45 CPI and misses per 1000 instructions for a running a TPC-C like database workload and the SPEC95 benchmarks (see Chapter 1) on the Alpha 21264 in the Compaq ES40. In addition to the worse miss rates shown here, the TPC-C like benchmark also has a branch misprediction rate of about 19 per 1000 instructions retired. This rate is 1.7 times worse than the average SPECint95 program and 25 times worse than the average SPECfp95. Since the 21264 uses an out-of-order instruction execution, the statistics are calculated as the number of misses per 1000 instructions successfully committed. Cvetnovic and Kessler [2000] collected this data, but unfortunately did not include miss rates of the L1 data cache or data TLB. Note that their hardware performance monitor could not isolate the benefits of successful hardware prefetching to the Instruction Cache. Hence, compulsory misses are likely very low.

does exercise the memory hierarchy more, with misses per instruction of 1.1% and 0.7%, respectively.

How do the CPIs compare to the peak rate of 0.25, or 4 instructions per clock cycle? For SPEC95 the 21264 completes almost 2 instructions per clock cycle, with an average CPI of 0.55 to 0.59. For the database benchmark, the combination of higher miss rates for caches and TLBs and a higher branch misprediction

rate (not shown) yield a CPI of 2.23, or less than one instruction every two clock cycles. This factor of four slowdown in CPI suggest that microprocessors designed to be used in servers may see much heavier demands on the memory systems than do microprocessors for desktops.

5.14 Another View: The Emotion Engine of the Sony Playstation 2

Desktop computers and severs rely on the memory hierarchy to reduce average access time to relatively static data, but there are embedded applications where data is often a continuos stream. In such applications there is still spatial locality, but temporal locality is much more limited.

To give another look at memory performance beyond the desktop, this section examines the microprocessor at the heart of Sony Playstation 2. As we shall see, the steady stream of graphics and audio demanded by electronic games leads to a different approach to memory design. The style is high bandwidth via many dedicated independent memories.

Figure 5.46 shows the 3Cs for the MP3 decode kernel. Compared to SPEC2000 results in Figure 5.15 on page 410, much smaller caches capture the misses for multimedia applications. Hence, we expect small caches.

Figure 5.47 shows a block diagram of the Sony Playstation 2 (PS2). Not surprisingly for a game machine, there are interfaces for video, sound, and a DVD player. Surprisingly, there are two standard computer I/O busses, USB and IEEE 1394, a PCMCIA slot as found in portable PCs, and a modem. These additions suggest Sony has suggest greater plans for the PS2 beyond traditional games. Although it appears the that I/O processor (IOP) simply handles the I/O devices and the game console, it includes a 34 MHz MIPS processor which also acts as the emulation computer to run games for earlier Sony Playstations. It also connects to a standard PC audio card to provide the sound for the games.

Thus, one challenge for the memory system of this embedded application is to act as source or destination for the extensive number of I/O devices. The PS2 designers met this challenge with two PC800 (400 MHz) DRDRAM chips using two channels, offering 32 MB of storage and a peak memory bandwidth of 3.2 MB/second (see section 5.8).

What's left in the figure is basically two big chips: the Graphics Synthesizer and the Emotion Engine.

The Graphics Synthesizer takes rendering commands from the Emotion Engine in what are commonly called *display lists*. These are lists of 32-bit commands that tell the renderer what shape to use and where to place them, plus what colors and textures to fill them.

This chip also has the highest bandwidth portion of the memory system. By using embedded DRAM on the Graphics Synthesizer, the chip contains the full video buffer *and* has a 2048-bit wide interface so that pixel filling is not a bottleneck. This embedded DRAM greatly reduces the bandwidth demands on the

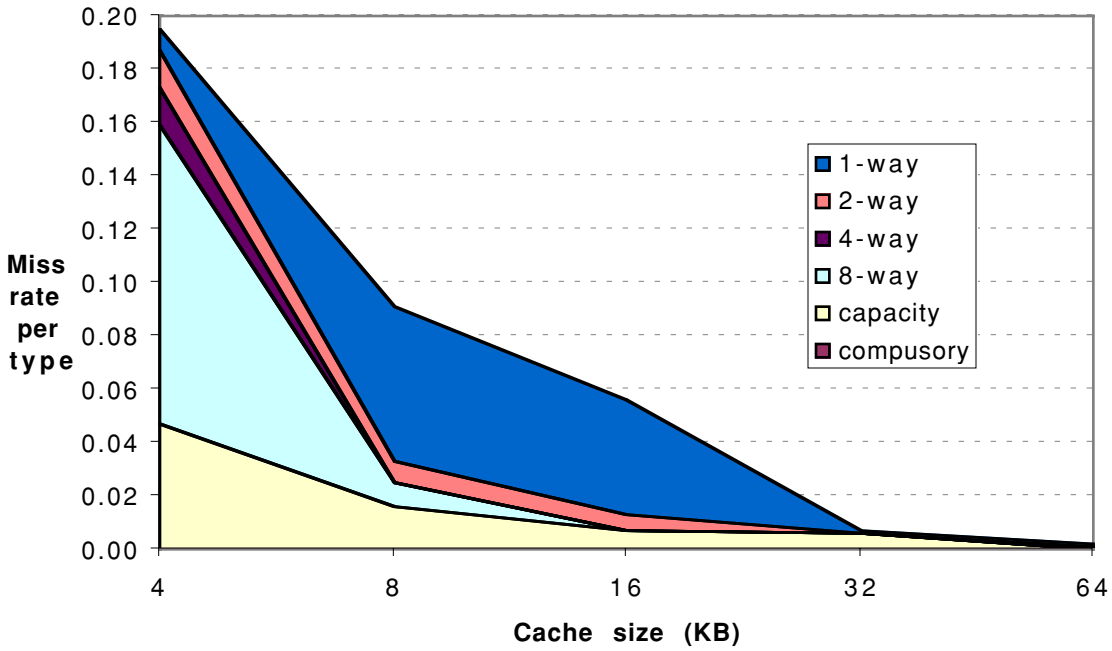


FIGURE 5.46 Three Cs for MPEG3 decode. A 2-way set associative 16-KB data cache has a total miss rate of 0.013, compared to in 0.041 in Figure 5.14 on page 409. The compulsory misses are too small to see on the graph. From Hughes et al [2001].

DRDRAM. It illustrates a common technique found in embedded applications: separate memories dedicated to individual functions to inexpensively achieve greater memory bandwidth for the entire system.

The remaining large chip is the Emotion Engine, and its job is to accept inputs from the IOP and create the display lists of a video game to enable 3D video transformations in real time. A major insight shaped the design of the Emotion Engine: generally in a racing car game there are foreground objects that are constantly changing and background objects that change less in reaction to the events, although the background can be most of the screen. This observation led to a split of responsibilities.

The CPU works with VPU0 as a tightly-coupled coprocessor, in that every VPU0 instruction is a standard MIPS coprocessor instruction, and the addresses are generated by the MIPS CPU. VPU0 is called a vector processor, but it is similar to a 128-bit, SIMD extensions for multimedia found in several desktop processors (see Chapter 2).

VPU1, in contrast, fetches its own instructions and data and acts in parallel with CPU-VPU0, acting more like a traditional vector unit. With this split, the

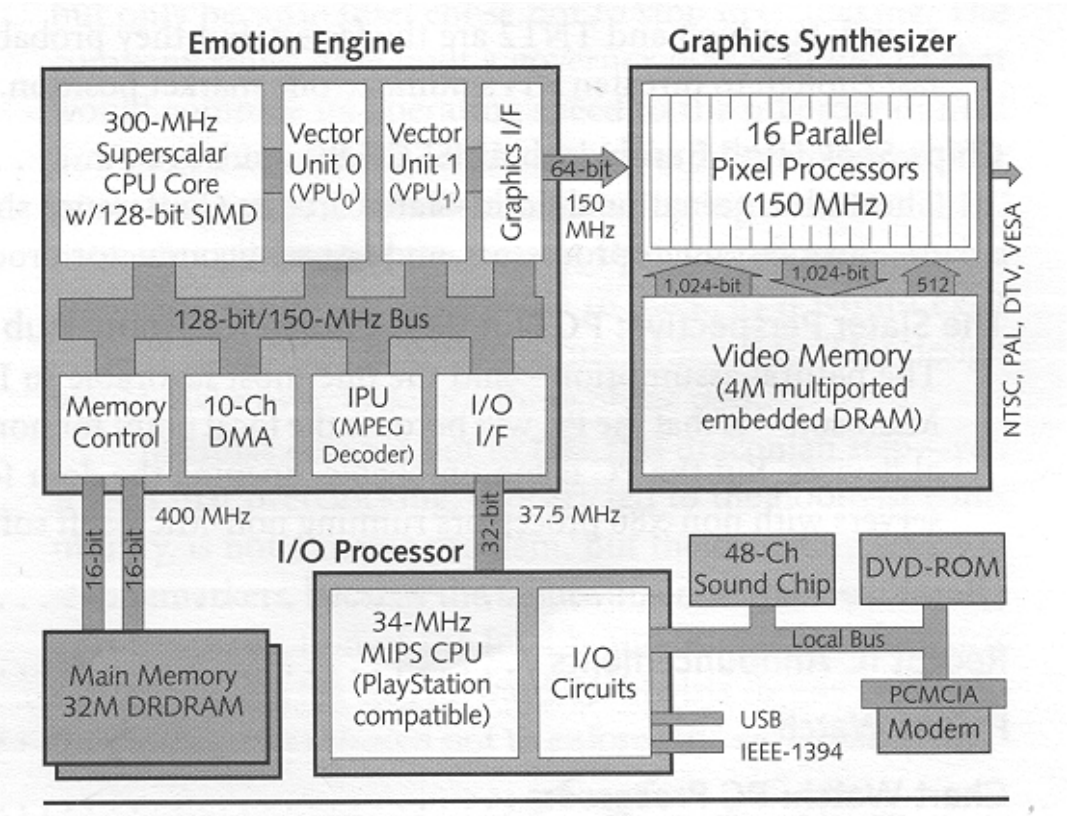


FIGURE 5.47 Block diagram of the Sony Playstation 2. The 10 DMA channels orchestrate the transfers between all the small memories on the chip, which when completed all head towards the Graphics Interface so as to be rendered by the Graphics Synthesizer. The Graphics Synthesized uses DRAM on-chip to provide an entire frame buffer plus graphics processors to perform the rendering desired based on the display commands given from the Emotion Engine. The Embedded DRAM allows 1024-bit transfers between the pixel processors and the display buffer. The Supescalar CPU is a 64-bit MIPS III with two-instruction issue, and comes with a 2-way set associative, 16-KB instruction cache, a 2-way set associative 8-KB data cache, and 16 KB of scatchpad memory. It has been extended with 128-bit SIMD instructions for multimedia applications (see Chapter 2). Vector Unit 0 is a primarily a DSP-like coprocessor for the CPU (see Chapter 2), which can an operator on 128-bit registers in SIMD manner between 8 bits and 32 bits per word. It has 4 KB of instruction memory and 4 KB of data memory. Vector Unit 1 has similar functions to VPU0, but it normally operates independently of the CPU, and contains 16 KB of instruction memory and 16 KB of data memory. All three units can communicate over the 128-bit system bus, but there are is also a 128-bit dedicated path between the CPU and VPU0 and a 128-bit dedicated path between VPU1 and the Graphics Interface. Although VPU0 and VPU1 have identical microarchitectures, the differences in memory size and units to which they have direct connections affect the roles that they take in a game. At 0.25- micron line widths, the Emotion Engine chip uses 13.5M transistors and 225 mm^2 and the Graphics Synthesizer is 279 mm^2. To put this in perspective, the Alpha 21264 microprocessor in 0.25-micron technology is about 160 mm^2 and uses 15M transistors. (This figure is based on a Figure 1 in “Sony’s Emotionally Charged Chip,” Microprocessor Report, 13:5.)

more flexible CPU-VPU0 handles the foreground action and the VPU1 handles the background. Both deposit their resulting display lists into the Graphics Interface to send the lists to the Graphics Synthesizer.

Thus, the programmer of the Emotion Engine thus has three processors sets to chose from to implement his program: the traditional 64-bit MIPS architecture including a floating point unit, the MIPS architecture extended with multimedia instructions (VPU0), and an independent vector processor (VPU1). To accelerate MPEG decoding, the is another coprocessor (Image Processing Unit) that can act independent of the other two.

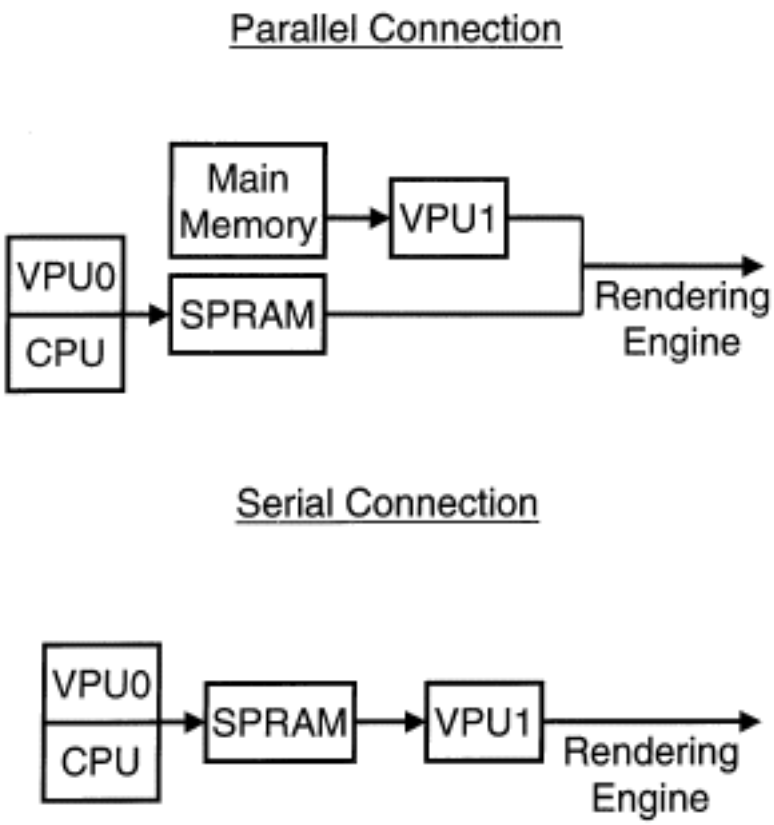


FIGURE 5.48 Two modes of using Emotion Engine organization. The first mode divides the work between the two units, and then allows the Graphics Interface to properly merge the display lists. The second more uses CPU/VPU0 as a filter of what to send to VPU1, which then does all the display lists. Its up to the programmer to chose between serial and parallel data flow. SPRAM is the scratchpad memory. **<<Redraw with Serial on top, Parallel on Bottom; can be much smaller>>**

With this split of function, the question then is how to connect the units together, how to make the data flow between units, and how to provide the memory

bandwidth needed by all these units. As mentioned above, the Emotion Engine designers chose many dedicated memories. The CPU has a 16-KB scatchpad memory (SPRAM) in addition to a 16-KB instruction cache and an 8-KB data cache. VPU0 has a 4-KB instruction memory and a 4-KB data memory, and VPU1 has a 16-KB instruction memory and a 16-KB data memory. Note that these are four *memories*, not caches of a larger memory elsewhere. In each memory the latency is just one clock cycle. VPU1 has more memory than VPU0 because it creates the bulk of the display lists and because it largely acts independently.

The programmer organizes all memories as two double buffers, one pair for the incoming DMA data and one pair for the outgoing DMA data. The programmer then uses the various processors to transform the data from the input buffer to the output buffer. To keep the data flowing among the units, the programmer next sets up the 10 DMA channels, taking care to meet the real time deadline for realistic animation of 15 frame per second.

Figure 5.48 shows that this organization supports two main operating modes: serial where CPU/VPU0 act as a preprocessor on what to give VPU1 for it to create for the Graphics Interface using the scratchpad memory as the buffer, and parallel where both the CPU/VPU0 and VPU1 create display lists. The display lists and the Graphics Synthesizer have multiple context identifiers to distinguish the parallel display lists to produce a coherent final image.

All units in the Emotion Engine are linked by a common 150 Mhz, 128-bit wide bus. To offer greater bandwidth, there are also two dedicated buses: a 128-bit path between the CPU and VPU0, and a 128-bit path between VPU1 and the Graphics Interface. The programmer also chooses which bus to use when setting up the DMA channels.

Taking the big picture, if a server-oriented designer had been given the problem we might see a single common bus with many local caches and cache coherent mechanism to keep data consistent. In contrast, the Playstation 2 followed the tradition of embedded designers and has at least nine distinct memory modules. To keep the data flowing in real time from memory to the display the PS2 uses dedicated memories, dedicated buses, and DMA channels. Coherency is the responsibility of the programmer, and given the continuous flow from main memory to the graphics interface and the real time requirements, programmer controlled coherency works well for this application.

5.15 Another View: The Sun Fire 6800 Server

The Sun Fire 6800 is a mid range multiprocessor server with particular attention paid to the memory system. The emphasis of this server is cost-performance for both commercial computing, running data base applications such data warehousing and data mining, as well as high performance computing. This server also includes special features to improve availability and maintainability.

Given these goals, what should be the size of the caches? Looking at the SPEC2000 results in Figure 5.17 on page 413 suggests miss rates of 0.5% for a 1-MB data cache, with infinitesimal instruction cache miss rates at those sizes. It would seem that a 1-MB off chip cache should be sufficient.

Commercial workloads, however, get very different results. Figure 5.49 shows the impact on the off chip cache for an Alpha 21164 server running commercial workloads. Unlike the results for SPEC2000, commercial workloads running database applications have significant misses just for instructions with a 1-megabyte cache. The reason is that code size of commercial databases is measured in millions of lines of code, unlike any SPEC2000 benchmark. Second, note that capacity and conflict misses remain significant until cache size becomes 4 to 8 megabyte. Note that even compulsory misses lead to a measurable causes higher CPI; this is because servers often run many processes, which results in many context switches and thus more compulsory misses. Finally, there is a new category of misses in a multiprocessor, and these are due to having to keep all the caches of a multiprocessor coherent, a problem mentioned in section 5.12. These are sometimes called *coherency misses*, adding a fourth C to our three C model from section 5.5. Chapter 6 explains a good deal more about coherence or sharing traffic in multiprocessors. The data suggests that commercial workloads need considerably bigger off-chip caches than do SPEC2000.

Figure 5.50 shows the essential characteristics of the Sun Fire 6800 that the designers selected. Note the 8 MB L2 cache, which is in response to the commercial needs.

The microprocessor that drives this server is the UltraSPARC III. One striking feature of the chip is the number of pins: 1368 in a Ball Grid Array. Figure 5.51 shows how one chip could use so many pins. The L2 caches bus operates at 200 MHz, local memory at 75 MHz, and the rest operate at 150 MHz. The combination of UltraSPARC III and the data switch yields a peak bandwidth to off chip memory of 11 Gbytes/second.

Note that the several wide buses include error correction bits in Figure 5.51. Error correction codes enable buses and memories to both detect and correct errors. The idea is to calculate and storage parity over different subsets of the bits in the protected word. When parity does not match it indicates an error. By looking at which of the overlapping subsets have a parity error and which don't, its possible to determine the bit that failed. The Sun Fire ECC was also designed to detect any pair of bit errors, and also to detect if a whole DRAM chip failed turning all the bits of an 8-bit wide chip to zero. Such codes are generally classified as *Single Error Correcting/Double Error Detecting* (*SEC/DED*). The UltraSPARC sends these ECC bits between the memory chips and the microprocessor, so that errors that occur on the high-speed buses are also detected and corrected.

In addition to several wide busses for memory bandwidth, the designers of UltraSPARC were concerned about latency. Hence, the chip includes a DRAM controller on the chip, which they claim saved 80 ns of latency. The result is 220 ns to the local memory and 280 ns to the non-local memory. (This server supports non-uniform memory access shared memory, described in Chapter 6.) Since

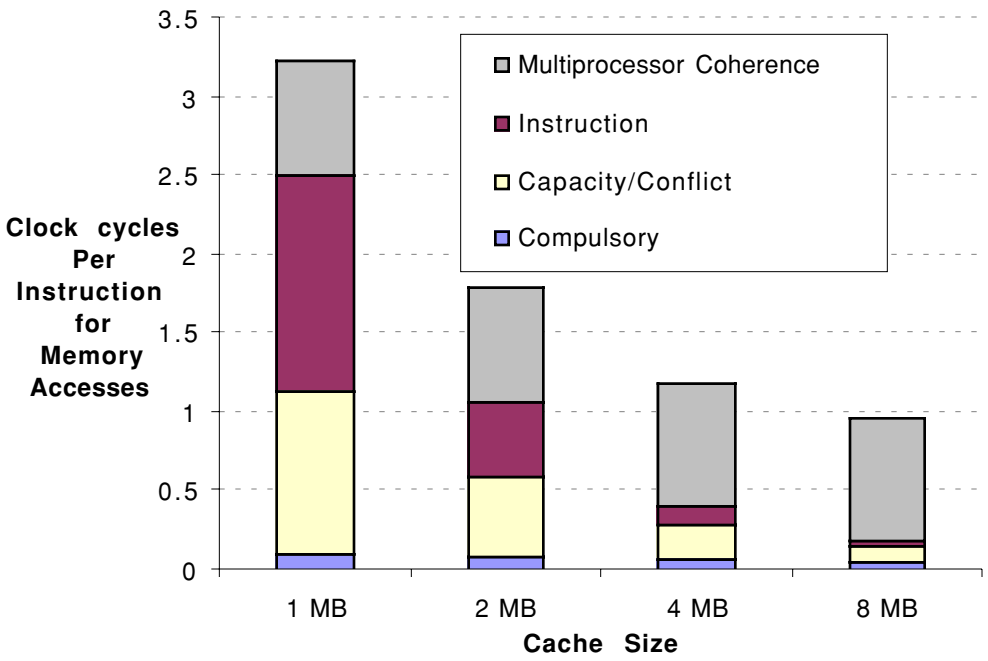


FIGURE 5.49 Clock cycles per instruction for memory accesses versus off-chip cache size for a four-processor server. Note how much higher the performance impact is for large caches than for the SPEC2000 programs in Figure 5.15 on page 410. The workload includes the Oracle commercial database engine for the online transaction processing (OLTP) and decision support systems and the AltaVista search engine for the Web index search. This data was collected by Barroso et al [1998] using the Alpha 21164 microprocessor.

memory is connected directly to the processor to lower latency, its size is a function of the number of processors. The limit in 2001 is 8 GB per processor.

For similar latency reasons, UltraSPARC also includes the tags for the L2 cache on chip. The designers claim this saved 10 clock cycles off a miss. At a total of almost 90 KB of tag, it is comparable in size to the data cache.

The on-chip caches are both 4-way set associative, with the instruction cache being 32 KB and the data cache being 64 KB. The block size for these caches is 32 bytes. To reduce latency to the data cache, it combines an address adder with the word line decoder. This combination largely eliminates the adder's latency. Compared to UltraSPARC II at the same cache size and clock rate, sum-addressed memory reduced latency from three to two clock cycles.

The L1 data cache uses write through (no write allocate) and the L2 cache uses write back (write allocate). Both L1 caches provide parity to detect errors; since the data cache is write through, there is always a redundant copy of the data elsewhere, so parity errors only require prefetching the good data. The memory system behind the cache supports to up 15 outstanding memory accesses.

Processors	2 to 24
Processors	2 to 24 UltraSPARC III processors
Processor Clock rate	900 MHz
Pipeline	14 stages
Superscalar	4-issue, 4-way sustained
L1 I cache	32 KB, 4-way set associative (S.A.), pseudorandom replacement
L1 I cache latency	2 clocks
L1 D cache	64 KB, 4-way SA; write through, no write allocate, pseudorandom replacement
L1 D cache latency	2 clocks
L1 I/D miss penalty	20 ns (15 to 18 clock cycles, depending on clock rate)
L2 cache	8 MB, direct mapped; write back, write allocate, multilevel inclusion
L2 miss penalty	220 to 280 ns (198 to 252 clock cycles, depending whether memory is local)
Write Cache	2 KB, 4-way SA, LRU, 64 byte block, no write allocate
Prefetch Cache	2 KB, 4-way SA, LRU, 64 byte block
Block size	32 bytes
Processor Address space	64 bit
Maximum Memory	8 GB/processor, or up to 192 GB total
System Bus, peak speed	Sun Fire Interconnect, 9.6 GB/sec
I/O cards	up to 8 66-MHz, 64-bit PCI, 24 33-MHz 64-bit PCI
Domains	1 to 4
Processor Power	70 W at 750 MHz
Processor Package	1368 pin flip-chip ceramic Ball Grid Array
Processor Technology	29 M transistors (75% SRAM cache), die size is 217 mm^2; 0.15 micron, 7-layer CMOS

FIGURE 5.50 Technical summary of Sun Fire 6800 server and UltraSPARC III microprocessor.

Between the two levels of cache is a 2-KB write cache. The write cache act as a write buffer and merges writes to consecutive locations. It keeps a bit per byte to indicate if it is valid and does not read the block from the L2 cache when the block is allocated. Often the entire block is written, there by avoiding a read access to the L2 cache. The designers claim more than 90% of the time UltraSPARC III can merge a store into an existing dirty block of the write cache. The write cache is also a convenient place to calculate ECC.

UltraSPARC III handles address translation with multiple levels of on-chip TLBs, with the smaller ones being fastest. A cache access starts with a virtual address selecting four blocks and four microtags which checks 8 bits of the virtual address to select the set. In parallel with the cache access the 64-bit virtual address is translated to the 43-bit physical address using two TLBs: a 16 entry fully

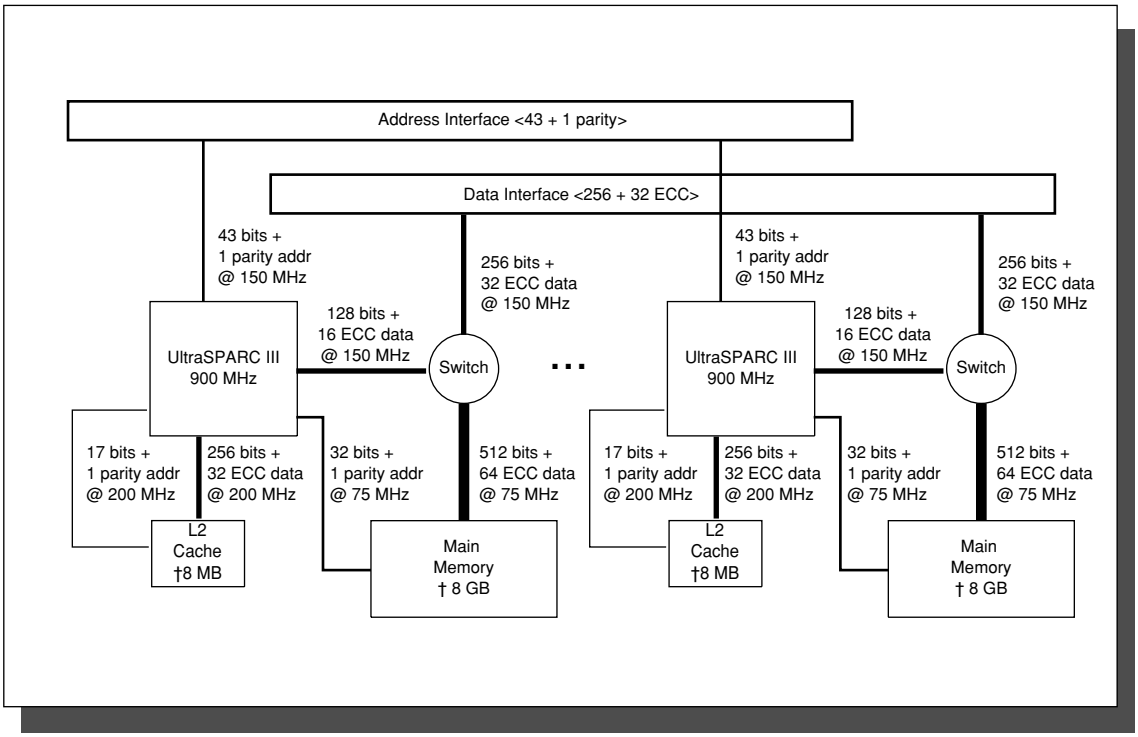


FIGURE 5.51 Sun Fire 6800 server block diagram. Note the large number of wide memory paths per processors. Up to 24 processors can be connected. Up to 12 share a single Sun Fire Data Interconnect. With more than 12, a second data interconnect is added. The number of separate paths to different memories are 256 data pins + 32 bits of error correction code (ECC) and 17 address bits + 1 bit parity for the off-chip L2 cache (200 MHz); 43 address pins + 1 bit of parity for addresses to external main memory; 32 address pins + 1 bit of parity for addresses to local main memory; 128 data pins + 16 bits of ECC to a data switch (150 MHz); 256 data pins + 32 bits of ECC between the data switch and the data interconnect (150 MHz); and 512 data pins + 64 bits of ECC between the data switch and local memory (75 MHz).

associative cache and a 128 entry 4-way associative cache. The physical address is then compared to the cache full tag, and only if they match is a cache hit allowed to proceed.

To get even more memory performance, UltraSPARC III also has a data prefetch cache, essentially the same as the streaming buffers described in section 5.6. It supports up to eight prefetch requests initiated either by hardware or by software. The prefetch cache remembers the address used to prefetch the data. If a load hits in prefetch cache, it automatically prefetches next load address. It calculates the stride using the current address and the previous address. Each prefetch entry is 64 bytes, and it is filled in 14 clock cycles from the L2 cache. Loads from the prefetch cache complete at the rate of 2 every 3 clock cycles versus 2 every 4 clock cycles from the data cache. This multiported memory can support two 8-byte reads and one 16-byte write every clock cycle.

In addition to prefetching data, UltraSPARC III has a small instruction prefetch buffer of 32 bytes that tries to stay one block ahead of the instruction decoder. On an instruction cache miss, two blocks are requested: one for the instruction cache and one for the instruction prefetch buffer. The buffer is then used to fill the cache if a sequential access also misses. In addition to parity and ECC to help with dependability, Sun Fire 6800 server offers up to four dynamic system domains. This option allows the computer to be divided into quarters or halves, with each piece running its own version of the operating system independently. Thus, a hardware or software failure in one piece does not affect applications running on the rest of the computer. The dynamic portion of the name means the reconfiguration can occur without rebooting the system.

To help diagnose problems, every UltraSPARC III has an 8-bit ``back door'' bus that runs independently from the main buses. If the system bus has an error, processors can still boot and run diagnostic programs over this back door bus to diagnose the problem.

Among the other availability features of the 6800 is a redundant path between the processors. Each system has two networks to connect the 24 processors together so that if one fails the system still works. Similarly, each Sun Fire interconnect has two crossbar chips to link it to the processor board, so that one crossbar chip can fail and yet the board can still communicate. There are also dual redundant system controllers that monitors server operation, and so they are able to notify administrators when problems are detected. Administrators can use the controllers to remotely initiate diagnostics and corrective actions.

In summary, the Sun Fire 6800 server and its processor pay much greater attention to dependability, memory latency and bandwidth, and system scalability than do desktop computers and processors.

5.16 Fallacies and Pitfalls

As the most naturally quantitative of the computer architecture disciplines, memory hierarchy would seem to be less vulnerable to fallacies and pitfalls. Yet the authors were limited here not by lack of warnings, but by lack of space!

Fallacy: Predicting cache performance of one program from another.

Figure 5.14 on page 409 shows the instruction miss rates and data miss rates for three programs from the SPEC2000 benchmark suite as cache size varies. Depending on the program, the data misses per thousand instructions for a 4096-KB cache is 9, 2, or 90, and the instruction misses per thousand instructions for a 4-KB cache is 55, 19, or 0.0004. Figure 5.45 on page 478 shows that commercial programs such as databases will have significant miss rates even in a 8-MB second-level cache, which is generally not the case for the SPEC programs. Similarly, MPEG3 decode in Figure 5.46 on page 480 fits entirely in a 64 KB data cache,

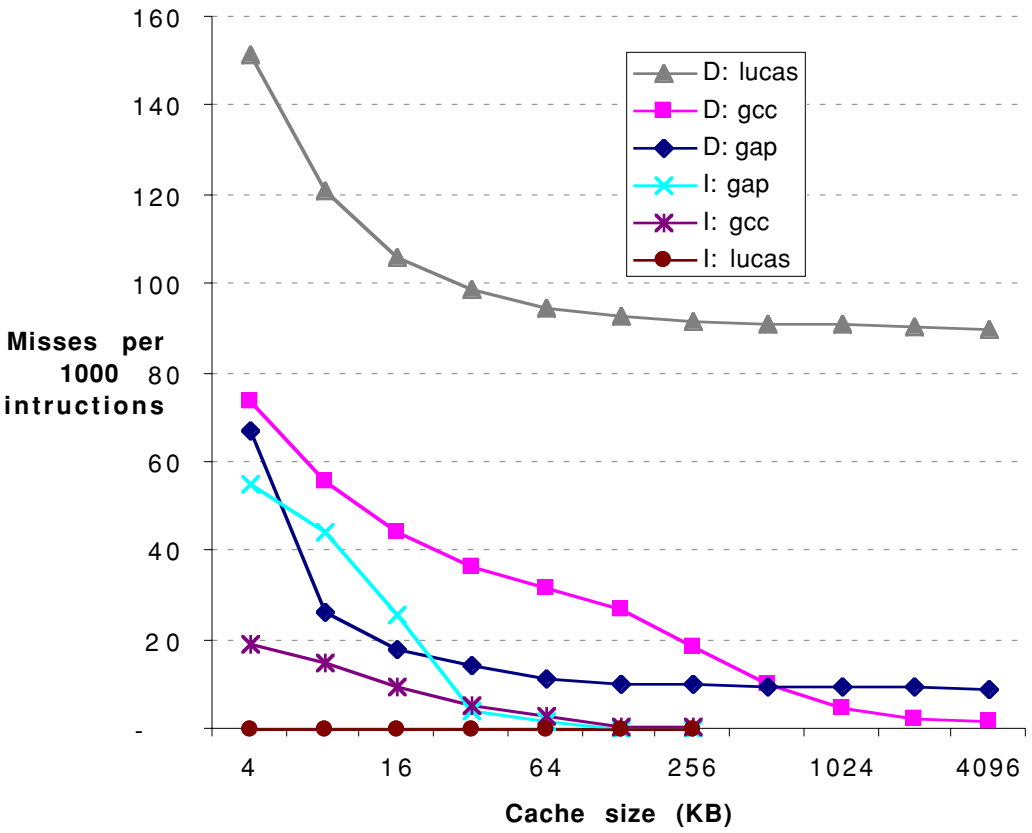


FIGURE 5.52 Instruction and data misses per thousand instructions for as cache size varies from 4 KB to 4096 KB. Instruction misses for gcc are 30,000 to 40,000 times larger than lucas, and conversely, data misses for lucas are 2 to 60 times larger than gcc. The programs gap, gcc, and lucas are from the SPEC2000 benchmark suite. These data are from the same experiment as in Figure 5.10.

while SPEC doesn't get such low miss rates until 1024 KB. Clearly, generalizing cache performance from one programs to another is unwise.

Pitfall: Simulating enough instructions to get accurate performance measures of the memory hierarchy.

There are really three pitfalls here. One is trying to predict performance of a large cache using a small trace. Another is that a program's locality behavior is not constant over the run of the entire program. The third is that a program's locality behavior may vary depending on the input.

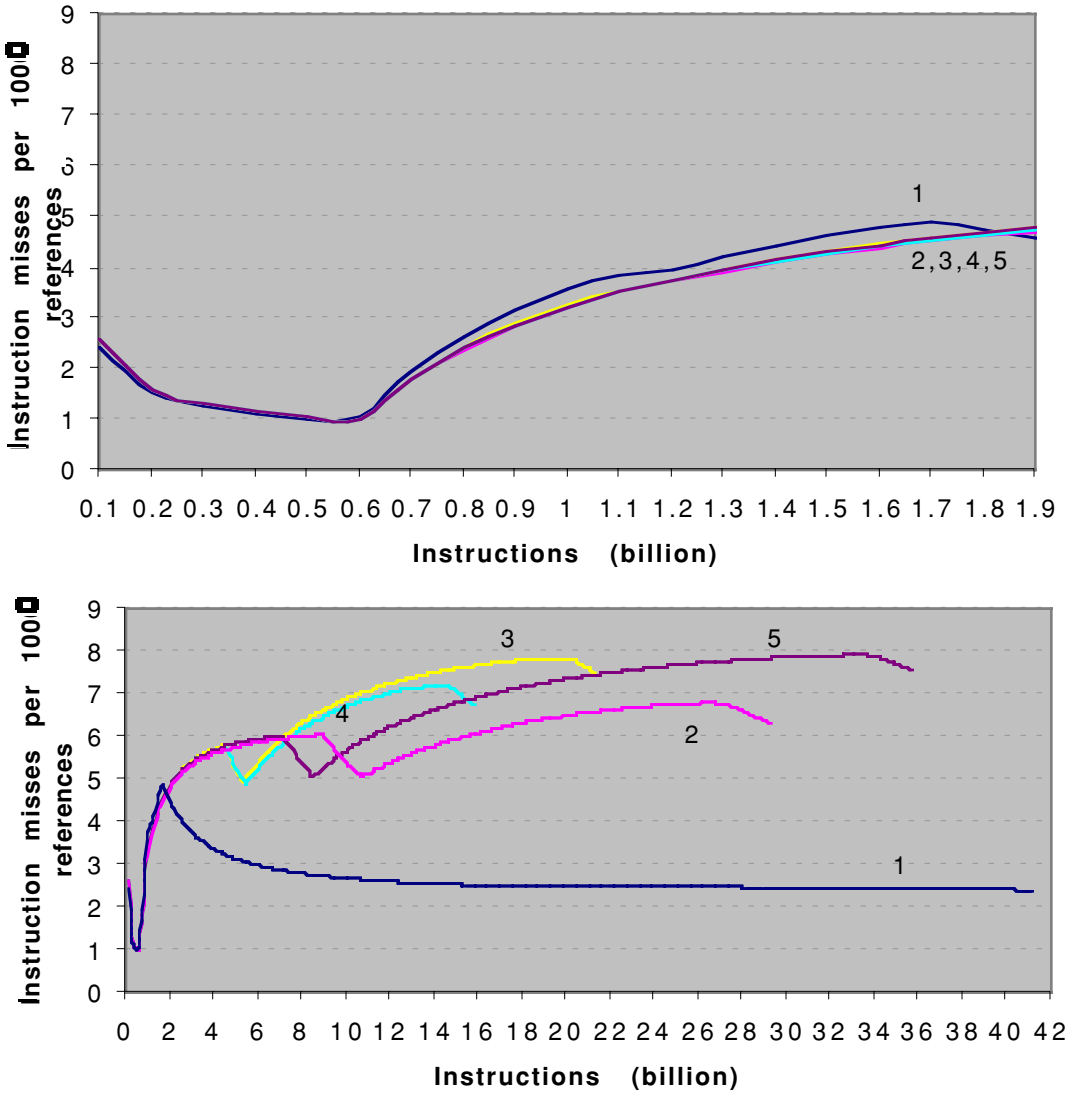


FIGURE 5.53 Instruction misses per 1000 references for five inputs to perl benchmark from SPEC2000. There is little variation in misses and little difference between the five inputs for the first 1.9 billion instructions, but running to completion shows how misses vary over the life of the program and how they depend on the input. The top graph shows the running average misses for the first 1.9 billion instructions, which starts at about 2.5 and ends at about 4.7 misses per 1000 references for all five inputs. The bottom graph shows the running average misses to run to competition, which takes 16 to 41 billion instructions depending on the input. After the first 1.9 billion instructions, the misses per 1000 references vary from 2.4 to 7.9 depending on the input. The simulations were for the Alpha processor using separate L1 caches for instructions and data, each 2-way 64KB with LRU, and a unified1MB direct-mapped L2 cache.

Figure 5.53 shows the cumulative average instruction misses per thousand instructions for five inputs to a single SPEC2000 program. For these inputs, the average memory rate for the first 1.9 billion instructions is very different from their average miss rate for the rest of the execution.

The first edition of this book included another example of this pitfall. The compulsory miss ratios were erroneously high (e.g., 1%) because of tracing too few memory accesses. A program with an compulsory cache miss ratio of 1% running on a computer accessing memory 10 million times per second (at the time of the first edition) would access hundreds of megabytes of memory per second:

$$\frac{10{,}000{,}000 \text{ accesses}}{\text{Second}} \times \frac{0.01 \text{ misses}}{\text{Access}} \times \frac{32 \text{ bytes}}{\text{Miss}} \times \frac{60 \text{ seconds}}{\text{Minute}} = \frac{192{,}000{,}000 \text{ bytes}}{\text{Minute}}$$

Data on typical page fault rates and process sizes do not support the conclusion that memory is touched at this rate.

Pitfall: Too small an address space.

Just five years after DEC and Carnegie Mellon University collaborated to design the new PDP-11 computer family, it was apparent that their creation had a fatal flaw. An architecture announced by IBM six years *before* the PDP-11 was still thriving, with minor modifications, 25 years later. And the DEC VAX, criticized for including unnecessary functions, sold millions of units after the PDP-11 went out of production. Why?

The fatal flaw of the PDP-11 was the size of its addresses (16 bits) as compared to the address sizes of the IBM 360 (24 to 31 bits) and the VAX (32 bits). Address size limits the program length, since the size of a program and the amount of data needed by the program must be less than $2^{\text{address size}}$. The reason the address size is so hard to change is that it determines the minimum width of anything that can contain an address: PC, register, memory word, and effective-address arithmetic. If there is no plan to expand the address from the start, then the chances of successfully changing address size are so slim that it normally means the end of that computer family. Bell and Strecker [1976] put it like this:

There is only one mistake that can be made in computer design that is difficult to recover from—not having enough address bits for memory addressing and memory management. The PDP-11 followed the unbroken tradition of nearly every known computer. [p. 2]

A partial list of successful computers that eventually starved to death for lack of address bits includes the PDP-8, PDP-10, PDP-11, Intel 8080, Intel 8086, Intel 80186, Intel 80286, Motorola AMI 6502, Zilog Z80, CRAY-1, and CRAY X-MP.

Even the venerable 80x86 line is showing danger signs, with Intel justifying migration to IA-64 in part to provide a larger flat address space than 32 bits, and AMD proposing its own 64-bit address extension called x86-64.

As we expected, by this third edition every desktop and server microprocessor manufacturer offers computers with 64-bit flat addresses. DSPs and embedded applications, however, may yet be condemned to repeat history as memories grow and desired functions multiply.

Pitfall: Emphasizing memory bandwidth in DRAMs versus memory latency.

Direct RDRAM offers up to 1.6 GBytes/second of bandwidth from a single DRAM. When announced, the peak bandwidth was 8 times faster than individual conventional SDRAM chips.

PCs do most memory accesses through a two-level cache hierarchy, so its unclear how much benefit is gained from high bandwidth without also improving memory latency. According to Pabst [2000], when comparing PCs with 400 MHz DRDRAMs to PCs 133 MHz SDRAM, for office applications they had identical average performance. For games, DRDRAM was 1% to 2% faster. For professional graphics applications, it was 10% to 15% faster. The tests used a 800 MHz Pentium III (which integrates a 256-KB L2 cache), chip sets that support a 133 MHz system bus, and 128 MB of main memory.

	Modules		Dell XPS PCs					
ECC?	**No ECC**	**ECC**	**No ECC**			**ECC**		
Label	**DIMM**	**RIMM**	**A**	**B**	**B - A**	**C**	**D**	**D - C**
Memory or System?	**DRAM**		**System**		**DRAM**	**System**		**DRAM**
Memory Size (MB)	256	256	128	512	384	128	512	384
SDRAM PC100	$175	$259	$1,519	$2,139	$620	$1,559	$2,269	$710
DRDRAM PC700	$725	$826	$1,689	$3,009	$1,320	$1,789	$3,409	$1,620
Price Ratio DRDRAM/SDRAM	4.1	3.2	1.1	1.4	2.1	1.1	1.5	2.3

FIGURE 5.54 Comparison of price of SDRAM v. DRDRAM in memory modules and in systems in 2000. DRDRAM memory modules cost about a factor of four more without ECC and three more with ECC. Looking at the cost of the extra 384 MB of memory in PCs in going from 128 MB to 512 MB, DRDRAM costs twice as much. Except for differences in bandwidths of the DRAMs, the systems were identically configured. The Dell XPS PCs were identical except for memory: 800 MHz Pentium III, 20 GB ATA disk, 48X CDROM, 17" monitor, and Microsoft Windows 95/98 and Office 98. The module prices were the lowest found at pricewatch.com in June 2000. By September 2001 PC800 DRDRAM cost $76 for 256 MB while PC100 to PC150 SDRAM cost $15 to $23, or about a factor of 3.3 to 5.0 less expensive. (In September 2001 Dell did not offer systems whose only difference was type of DRAMs, hence we stick with the comparison from 2000.)

One measure of the RDRAM cost is about a 20% larger die for the same capacity compared to SDRAM. DRAM designers use redundant rows and columns to significantly improve yield on the memory portion of the DRAM, so a much

larger interface might have a disproportionate impact on yield. Yields are a closely guarded secret, but prices are not. Figure 5.54 compares prices of various versions of DRAM, in memory modules and in systems. Using this evaluation, in 2000 the price is about a factor of two to three higher for RDRAM.

RDRAM is at its strongest in small memory systems that need high bandwidth. The low cost of the Sony Playstation 2, for example, limits the amount of memory in the system to just two chips, yet its graphics has an appetite for high memory bandwidth. RDRAM is at its weakest in servers, where the large number of DRAM chips needed in even the minimal memory configuration make it easy to achieve high bandwidth with ordinary DRAMs.

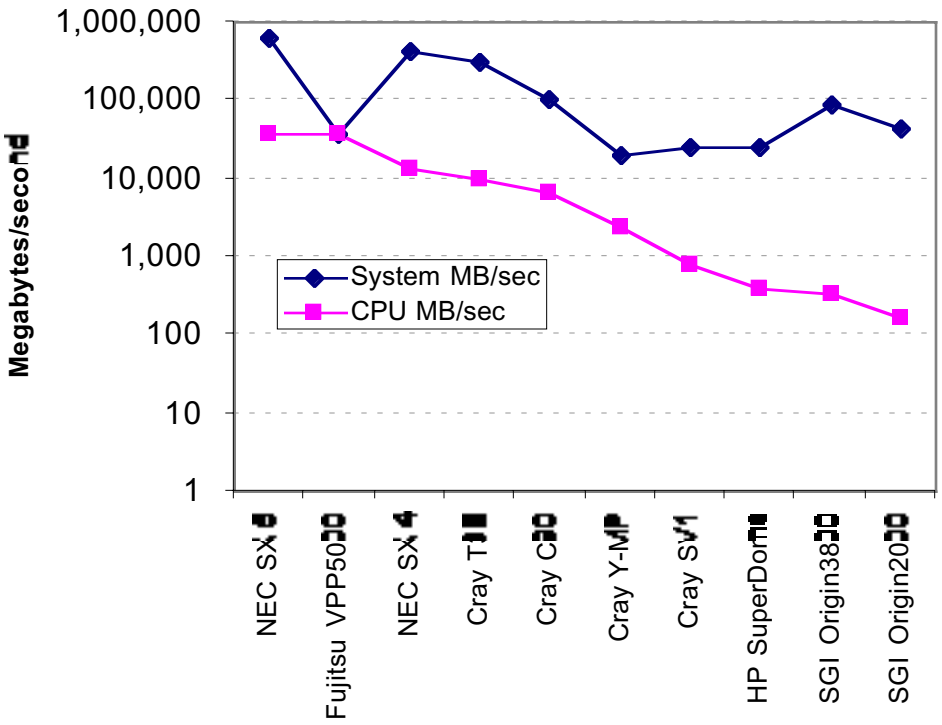


FIGURE 5.55 Top 10 in memory bandwidth as measured by the copy portion of the stream benchmark [McCalpin 2001]. Note that the last three computers are the only cache-based systems on the list, and that six of the top seven are vector computers. Systems use between 8 and 256 processors to achieve higher memory bandwidth. System bandwidth is bandwidth of all CPUs collectively. CPU bandwidth is simply system bandwidth divided by the number of CPUs. The STREAM benchmark is a simple synthetic benchmark program that measures sustainable memory bandwidth (in MB/s) for simple vector kernels. It is specifically works with data sets much larger than the available cache on any given system.

Pitfall: Delivering high memory bandwidth in a cache-based system.

Caches help with average cache memory latency but may not deliver high memory bandwidth to an application that needs it. Figure 5.55 shows the top ten results from the Stream benchmark as of 2001, which measures bandwidth to copy data [McCalpin 2001].The NEC SX 5 memory offers up to 16384 SDRAM memory banks to achieve its top ranking.

Only three computers rely on data caches, and they are the three slowest of the top ten, about a factor of a hundred slower than the fastest processor. Stated another way, a processor from 1988–the Cray YMP–still has a factor of 10 in memory bandwidth advantage over cache based processors from 2001.

Pitfall: Ignoring the impact of the operating system on the performance of the memory hierarchy.

Figure 5.56 shows the memory stall time due to the operating system spent on three large workloads. About 25% of the stall time is either spent in misses in the operating system or results from misses in the application programs because of interference with the operating system.

	Misses		Time						
			% time due to appl. misses	% time due directly to OS misses					% time OS misses & appl. conflicts
Workload	% in appl	% in OS	Inherent appl. misses	OS conflicts w. appl.	OS instr misses	Data misses for migration	Data misses in block operations	Rest of OS misses	
Pmake	47%	53%	14.1%	4.8%	10.9%	1.0%	6.2%	2.9%	25.8%
Multipgm	53%	47%	21.6%	3.4%	9.2%	4.2%	4.7%	3.4%	24.9%
Oracle	73%	27%	25.7%	10.2%	10.6%	2.6%	0.6%	2.8%	26.8%

FIGURE 5.56 Misses and time spent in misses for applications and operating system. The operating system adds about 25% to the execution time of the application. Each CPU has a 64-KB instruction cache and a two-level data cache with 64 KB in the first level and 256 KB in the second level; all caches are direct mapped with 16-byte blocks. Collected on Silicon Graphics POWER station 4D/340, a multiprocessor with four 33-MHz R3000 CPUs running three application workloads under a UNIX System V—Pmake: a parallel compile of 56 files; Multipgm: the parallel numeric program MP3D running concurrently with Pmake and five-screen edit session; and Oracle: running a restricted version of the TP-1 benchmark using the Oracle database. Data from Torrellas, Gupta, and Hennessy [1992].

Pitfall: Relying on the operating systems to change the page size over time.

The Alpha architects had an elaborate plan to grow the architecture over time by growing its page size, even building it into the size of its virtual address. When it

became time to grow page sizes with later Alphas, the operating system designers balked and the virtual memory system was revised to grow the address space while maintaining the 8-KB page.

Architects of other computers noticed very high TLB miss rates, and so added multiple, larger page sizes to the TLB. The hope was that operating systems programmers would allocate an object to the largest page that made sense, thereby preserving TLB entries. After a decade of trying, most operating systems use these "superpages" only for handpicked functions: mapping the display memory or other I/O devices, or using very large pages for the database code.

5.17 Concluding Remarks

Figure 5.57 compares the memory hierarchy of microprocessors aimed at desktop, server, and embedded applications. The L1 caches are similar across applications, with the primary differences being L2 cache size, die size, processor clock rate, and instructions issued per clock.

In contrast to showing the state of the art in a given year, Figure 5.56 shows evolution over a decade of the memory hierarchy of Alpha microprocessors and systems. The primary change between the Alpha 21064 and 21364 is the hundredfold increase in on-chip cache size, which tries to compensate for the sixfold increase in main memory latency as measured in instructions.

The difficulty of building a memory system to keep pace with faster CPUs is underscored by the fact that the raw material for main memory is the same as that found in the cheapest computer. It is the principle of locality that saves us here—its soundness is demonstrated at all levels of the memory hierarchy in current computers, from disks to TLBs. One question is whether increasing scale breaks any of our assumptions. Are L3 caches bigger than prior main memories a cost-effective solution? Do 8 KB pages makes sense with terabyte main memories?

The design decisions at all these levels interact, and the architect must take the whole system view to make wise decisions. The primary challenge for the memory-hierarchy designer is in choosing parameters that work well together, not in inventing new techniques. The increasingly fast CPUs are spending a larger fraction of time waiting for memory, which has led to new inventions that have increased the number of choices: prefetching, cache-aware compilers, and increasing page size. Fortunately, there tends to be a technological "sweet spot" in balancing cost, performance, power, and complexity: missing the target wastes performance, power, hardware, design time, debug time, or possibly all five. Architects hit the target by careful, quantitative analysis.

MPU	AMD Athlon	Intel Pentium III	Intel Pentium 4	IBM Power-PC 405CR	Sun UltraSPARC III
Instruction set architecture	80x86	80x86	80x86	PowerPC	SPARC v9
Intended application	desktop	desktop, server	desktop	embedded core	server
CMOS Process	0.18	0.18	0.18	0.25	0.15
Die size (mm^2)	128	106 to 385	217	37	210
Instructions issued/clock	3	3	3 RISC ops	1	4
Clock Rate (2001)	1400 MHz	900 - 1200 MHz	2000 MHz	266 MHz	900 MHz
Instruction Cache	64 KB, 2-way S.A.	16 KB, 2-way S.A.	12000 RISC op trace cache (~96 KB)	16 KB, 2-way S.A.	32 KB, 4-way S.A.
Latency (clocks)	3	3	4	1	2
Data Cache	64 KB, 2-way S.A.	16 KB, 2-way S.A.	8 KB, 4-way S.A.	8 KB, 2-way S.A.	64 KB, 4-way S.A.
Latency (clocks)	3	3	2	1	2
TLB entries (I/D/L2 TLB)	280/288	32/64	128	4/8/64	128/512
Min. page size	8 KB	8 KB	8 KB	1 KB	8 KB
On Chip L2 Cache	256 KB, 16-way S.A.	256 - 2048 KB, 8-way S.A.	256 KB, 8-way S.A.	--	--
Off Chip L2 Cache	--	--	--	--	8MB, 1-wayS.A.
Latency (clocks)	11	7	?	--	15
Block Size (L1/L2, bytes)	64	32	64/128	32	32
Memory bus width (bits)	64	64	64	64	128
Memory bus clock	133 MHz	133 MHz	400 MHz	133 MHz	150 MHz

FIGURE 5.57 Desktop, embedded and server microprocessors in 2001. From a memory hierarchy perspective, the primary differences between applications is L2 cache. There is no L2 cache for embedded, 256 KB on chip for desktop, and servers use 2MB on chip or 8 MB off chip. The processor clock rates also vary: 266 MHz for embedded, 900 MHz for servers, and 1200 to 2000 MHz for desktop. The Intel Pentium III includes the Xeon chip set for multiprocessor servers. It has the same processor core as the standard Pentium III, but a much larger on-chip L2 cache (up to 2 MB) and die size (385 mm^2) but a slower clock rate (900 MHz).

CPU	21064	21164	21264	21364
CMOS Process Feature Size	0.68	0.50	0.35	0.18
Clock Rate (Initial)	200	300	525	~ 1000
1st System Ship Date	3000 / 800	8400 5/300	ES40	2002-2003?
CPI gcc (SPECInt92/95)	2.51	1.27	0.63	~ 0.6
Instruction Cache	8 KB, 1-way	8 KB, 1-way	64 KB, 2-way	64 KB, 2-way
Latency (clocks)	2	2	2 or 3	2 or 3
Data Cache	8 KB, 1-way, Write Through	8 KB, 1-way, Write Through	64 KB, 2-way, Write Back	64 KB, 2-way, Write Back
Latency	2	2	3	3
Write/Victim Buffer	4 blocks	6 blocks	8 blocks	32 blocks
Block Size (bytes, all caches)	32	32	32 or 64	64
Virtual/Physical Address Size	43/34	43/40	48/44 or 43/41	48/44 or 43/41
Page size	8 KB	8 KB	8 KB	8 KB or 64 KB
Instruction TLB	12 entry, F.A	48 entry, F.A	128 entry, F.A	128 entry, F.A
Data TLB	32 entry, F.A	64 entry, F.A	128 entry, F.A	128 entry, F.A
Path Width Off Chip (bits)	128	128	128 to L2, 64 to memory	128?
On Chip Unified L2 Cache	---	96 KB, 3-way, Write Back	---	1536 KB, 6-way, Write Back
Latency (clocks)	---	7	---	12
Off Chip Unified L2 or L3 Cache	2 MB, 1-way, Write Back	4 MB, 1-way, Write Back	8 MB, 1-way, Write Back	---
Latency (clocks)	5	12	16	---
Memory size	.008 - 1 GB	0.125 - 14 GB	0.5 - 32 GB	0.5 - 4 GB / proc.
Latency (clocks)	68	80	122	~ 90
Latency (instructions)	27	63	194	~ 150

FIGURE 5.58 Four generations of Alpha microprocessors and systems. Instruction latency was calculated by dividing the latency in clock cycles by average CPI for SPECint programs. The 21364 integrates a large on-chip cache and a memory controller to connect directly to DRDRAM chips, thereby significantly lowering memory latency. The large on-chip cache and low latency to memory make an off-chip cache unnecessary. A network allows processors to access non-local memory with non-uniform access times (see Chapter 6): 30 clocks per network hop, so 120 clocks in the nearest group of 4 and 200 in a group of 16. Memory latency in instructions is calculated by dividing clocks by average CPI.

5.18 Historical Perspective and References

Although the pioneers of computing knew of the need for a memory hierarchy and coined the term, the automatic management of two levels was first proposed by Kilburn et al. [1962]. It was demonstrated with the Atlas computer at the University of Manchester. This computer appeared the year *before* the IBM 360 was announced. Although IBM planned for its introduction with the next generation (System/370), the operating system TSS wasn't up to the challenge in 1970. Virtual memory was announced for the 370 family in 1972, and it was for this computer that the term "translation look-aside buffer" was coined [Case and Padegs 1978]. The only computers today without virtual memory are a few supercomputers, embedded processors, and older personal computers.

Both the Atlas and the IBM 360 provided protection on pages, and the GE 645 was the first system to provide paged segmentation. The earlier Burroughs computers provided virtual memory using segmentation, similar to the segmented address scheme of the Intel 8086. The 80286, the first 80x86 to have the protection mechanisms described on pages 463 to 467, was inspired by the Multics protection software that ran on the GE 645. Over time, computers evolved more elaborate mechanisms. The most elaborate mechanism was *capabilities*, which reached its highest interest in the late 1970s and early 1980s [Fabry 1974; Wulf, Levin, and Harbison 1981]. Wilkes [1982], one of the early workers on capabilities, had this to say:

Anyone who has been concerned with an implementation of the type just described [capability system], or has tried to explain one to others, is likely to feel that complexity has got out of hand. It is particularly disappointing that the attractive idea of capabilities being tickets that can be freely handed around has become lost ….

Compared with a conventional computer system, there will inevitably be a cost to be met in providing a system in which the domains of protection are small and frequently changed. This cost will manifest itself in terms of additional hardware, decreased runtime speed, and increased memory occupancy. It is at present an open question whether, by adoption of the capability approach, the cost can be reduced to reasonable proportions.

Today there is little interest in capabilities either from the operating systems or the computer architecture communities, despite growing interest in protection and security.

Bell and Strecker [1976] reflected on the PDP-11 and identified a small address space as the only architectural mistake that is difficult to recover from. At the time of the creation of PDP-11, core memories were increasing at a very slow rate. In addition, competition from 100 other minicomputer companies meant that DEC might not have a cost-competitive product if every address had to go

through the 16-bit datapath twice. Hence, the architect's decision to add only 4 more address bits than found in the predecessor of the PDP-11.

The architects of the IBM 360 were aware of the importance of address size and planned for the architecture to extend to 32 bits of address. Only 24 bits were used in the IBM 360, however, because the low-end 360 models would have been even slower with the larger addresses in 1964. Unfortunately, the architects didn't reveal their plans to the software people, and programmers who stored extra information in the upper 8 "unused" address bit foiled the expansion effort. (Apple made a similar mistake 20 years later with the 24-bit address in the Motorola 68000, which required a procedure to later determine "32-bit clean" programs for the Macintosh when later 68000s used the full 32-bit virtual address.) Virtually every computer since then, will check to make sure the unused bits stay unused, and trap if the bits have the wrong value.

A few years after the Atlas paper, Wilkes published the first paper describing the concept of a cache [1965]:

The use is discussed of a fast core memory of, say, 32,000 words as slave to a slower core memory of, say, one million words in such a way that in practical cases the effective access time is nearer that of the fast memory than that of the slow memory. [p. 270]

This two-page paper describes a direct-mapped cache. Although this is the first publication on caches, the first implementation was probably a direct-mapped instruction cache built at the University of Cambridge. It was based on tunnel diode memory, the fastest form of memory available at the time. Wilkes states that G. Scarott suggested the idea of a cache memory.

Subsequent to that publication, IBM started a project that led to the first commercial computer with a cache, the IBM 360/85 [Liptay 1968]. Gibson [1967] describes how to measure program behavior as memory traffic as well as miss rate and shows how the miss rate varies between programs. Using a sample of 20 programs (each with 3 million references!), Gibson also relied on average memory access time to compare systems with and without caches. This precedent is more than 30 years old, and yet many used miss rates until the early 1990s.

Conti, Gibson, and Pitkowsky [1968] describe the resulting performance of the 360/85. The 360/91 outperforms the 360/85 on only 3 of the 11 programs in the paper, even though the 360/85 has a slower clock cycle time (80 ns versus 60 ns), less memory interleaving (4 versus 16), and a slower main memory (1.04 microsecond versus 0.75 microsecond). This paper was also the first to use the term "cache."

Others soon expanded the cache literature. Strecker [1976] published the first comparative cache design paper examining caches for the PDP-11. Smith [1982] later published a thorough survey paper, using the terms "spatial locality" and "temporal locality"; this paper has served as a reference for many computer designers.

Although most studies relied on simulations, Clark [1983] used a hardware monitor to record cache misses of the VAX-11/780 over several days. Clark and Emer [1985] later compared simulations and hardware measurements for translations.

Hill [1987] proposed the three C's used in section 5.5 to explain cache misses. Jouppi [1998] retrospectively says that Hill's three Cs model led directly to his invention of the victim cache to take advantage of faster direct map caches and yet avoid most of the cost of conflict misses. Ugumar and Abraham' [1993] argue that the baseline cache for the three C's model should use optimal replacement; this eliminates the anomalies of LRU-based miss classification, and allows conflict misses to be broken down into those caused by mapping and those caused by a non-optimal replacement algorithm.

One of the first papers on nonblocking caches is by Kroft [1981]. Kroft [1998] later explained that he was the first to design a computer with a cache at Control Data Corporation, and when using old concepts for new mechanisms, he hit upon the idea of allowing his two-ported cache to continue to service other accesses on a miss.

Baer and Wang [1988] did one of the first examinations of multilevel inclusion property. Wang, Baer, and Levy [1989] then produced an early paper on performance evaluation of multilevel caches. Later, Jouppi and Wilton [1994] proposed multilevel exclusion for multilevel caches on chip.

In addition to victim caches, Jouppi [1990] also examined prefetching via streaming buffers. His work was extended by Farkas et al [1995] to that streaming buffers work well with non-blocking loads and speculative execution for in-order processors, and later Farkas et al [1997] showed that while out-of-order processors can tolerate unpredictable latency better, they still benefit. They also refined memory bandwidth demands of stream buffers.

Proceedings of the Symposium on Architectural Support for Compilers and Operating Systems (ASPLOS) and the International Computer Architecture Symposium (ISCA) from the 1990s are filled with papers on caches. (In fact some wags claimed ISCA really stood for the International *Cache* Architecture Symposium.)

This chapter relies on the measurements of SPEC2000 benchmarks collected by Cantin and Hill [2001]. There are several other papers used in this chapter that are cited in the captions of the figures that use the data: Agarwal and Pudar [1993]; Barroso, Gharachorloo, and Bugnion [1998]; Farkas and Jouppi [1994]; Jouppi [1990]; Lam, Rothberg, and Wolf [1991]; Mowry, Lam, and Gupta [1992]; Lebeck and Wood [1994]; McCalpin [2001]; and Torrellas, Gupta, and Hennessy [1992].

The Alpha architecture is described in detail by Bhandarkar [1995] and by Sites [1992], and sources of information on the 21264 are Compaq [1999], Cvetanovic and Kessler [2000], and Kessler[1999]. Two Emotion Engine references are Kunimatsu et al [2000] and Oka and Suzuoki [1999]. Information on the Sun Fire 6800 server is found primarily on Sun's web site, but Horel and Lauterbach [1999] and Heald, R. *et al* [2000] published detailed information on UltraSPARC III.

References

AGARWAL, A. [1987]. *Analysis of Cache Performance for Operating Systems and Multiprogramming*, Ph.D. Thesis, Stanford Univ., Tech. Rep. No. CSL-TR-87-332 (May).

AGARWAL, A. AND S. D. PUDAR [1993]. "Column-associative caches: A technique for reducing the miss rate of direct-mapped caches," 20th Annual Int'l Symposium on Computer Architecture ISCA '20, San Diego, Calif., May 16–19. *Computer Architecture News* 21:2 (May), 179–90.

BAER, J.-L. AND W.-H. WANG [1988]. "On the inclusion property for multi-level cache hierarchies," *Proc. 15th Annual Symposium on Computer Architecture* (May–June), Honolulu, 73–80.

Barroso, L.A., Gharachorloo, K. and E. Bugnion [1998]. "Memory System Characterization of Commercial Workloads," Proceedings 25th International Symposium on Computer Architecture, Barcelona (July), 3-14.

BELL, C. G. AND W. D. STRECKER [1976]. "Computer structures: What have we learned from the PDP-11?," *Proc. Third Annual Symposium on Computer Architecture* (January), Pittsburgh, 1–14.

BHANDARKAR, D. P. [1995]. *Alpha Architecture Implementations,* Digital Press, Newton, Mass.

BORG, A., R. E. KESSLER, AND D. W. WALL [1990]. "Generation and analysis of very long address traces," *Proc. 17th Annual Int'l Symposium on Computer Architecture*, Seattle, May 28–31, 270–9.

CABTIN, J. F. AND M. D.HILL, [2001]. *Cache Performance for Selected SPEC CPU2000 Benchmark*s, http://www.jfred.org/cache-data.html, (June).

CASE, R. P. AND A. PADEGS [1978]. "The architecture of the IBM System/370," *Communications of the ACM* 21:1, 73–96. Also appears in D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples* (1982), McGraw-Hill, New York, 830–855.

CLARK, D. W. [1983]. "Cache performance of the VAX-11/780," *ACM Trans. on Computer Systems* 1:1, 24–37.

D. W. CLARK and J. S. EMER [1985], "Performance of the VAX-11/780 Translation Buffer: Simulation and Measurement," *ACM Trans. on Computer Systems*, 3, 1 (February 1985),31-62.

COMPAQ COMPUTER CORPORATION [1999] *Compiler Writer's Guide for the Alpha 21264* Order Number EC-RJ66A-TE, June, 112 pages. http://www1.support.compaq.com/alpha-tools/documentation/current/21264_EV67/ec-rj66a-te_comp_writ_gde_for_alpha21264.pdf

CONTI, C., D. H. GIBSON, AND S. H. PITKOWSKY [1968]. "Structural aspects of the System/360 Model 85, Part I: General organization," *IBM Systems J*. 7:1, 2–14.

CRAWFORD, J. H. AND P. P. GELSINGER [1987]. *Programming the 80386*, Sybex, Alameda, Calif.

CVETANOVIC, Z. and R.E. KESSLER [2000] "Performance Analysis of the Alpha 21264-based Compaq ES40 System." *Proc. 27th Annual Int'l Symposium on Computer Architecture*, Vancouver, Canada, June 10-14, IEEE Computer Society Press, 192-202.

FABRY, R. S. [1974]. "Capability based addressing," *Comm. ACM* 17:7 (July), 403–412.

Farkas, K.I., P. Chow, N.P. Jouppi,; Z. Vranesic [1997]. "Memory-system design considerations for dynamically-scheduled processors."24th Annual International Symposium on Computer Architecture. Denver, CO, USA, 2-4 June, 133-43.

FARKAS, K. I. AND N. P. JOUPPI [1994]. "Complexity/performance trade-offs with non-blocking loads," *Proc. 21st Annual Int'l Symposium on Computer Architecture*, Chicago (April).

Farkas, K.I., N.P.Jouppi, and P. Chow [1995]. "How useful are non-blocking loads, stream buffers and speculative execution in multiple issue processors?," Proceedings. First IEEE Symposium on High-Performance Computer Architecture, Raleigh, NC, USA, 22-25 Jan.,78-89.

GAO, Q. S. [1993]. "The Chinese remainder theorem and the prime memory system," 20th Annual Int'l Symposium on Computer Architecture ISCA '20, San Diego, May 16–19, 1993. *Computer Architecture News* 21:2 (May), 337–40.

GEE, J. D., M. D. HILL, D. N. PNEVMATIKATOS, AND A. J. SMITH [1993]. "Cache performance of the SPEC92 benchmark suite," *IEEE Micro* 13:4 (August), 17–27.

GIBSON, D. H. [1967]. "Considerations in block-oriented systems design," *AFIPS Conf. Proc.* 30, SJCC, 75–80.

HANDY, J. [1993]. *The Cache Memory Book*, Academic Press, Boston.

Heald, R. et al [2000]. " A third-generation SPARC V9 64-b microprocessor," *IEEE Journal of Solid-State Circuits*, 35:11 (Nov), 1526-38.

HILL, M. D. [1987]. *Aspects of Cache Memory and Instruction Buffer Performance*, Ph.D. Thesis, University of Calif. at Berkeley, Computer Science Division, Tech. Rep. UCB/CSD 87/381 (November).

HILL, M. D. [1988]. "A case for direct mapped caches," *Computer* 21:12 (December), 25–40.

Horel, T. and G. Lauterbach [1999]. "UltraSPARC-III: designing third-generation 64-bit performance," *IEEE Micro*, 19:3 (May-June), 73-85.

Hughes, C.J.; Kaul, P.; Adve, S.V.; Jain, R.; Park, C.; Srinivasan, J. [2001]. "Variability in the execution of multimedia applications and implications for architecture, " *Proc. 28th Annual International Symposium on Computer Architecture*, Goteborg, Sweden, 30 June-4 July , 254-65.

JOUPPI, N. P. [1990]. "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," *Proc. 17th Annual Int'l Symposium on Computer Architecture*, 364–73.

JOUPPI, N. P. [1998]. "Retrospective: Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers."*25 Years of the International Symposia on Computer Architecture (Selected Papers)*. ACM, 71-73.

Jouppi, N.P. and S.J.E. Wilton [1994]. "Trade-offs in two-level on-chip caching. Proceedings the 21st Annual International Symposium on Computer Architecture, Chicago, IL, USA, (18-21 April).34-45.

KESSLER, R.E. [1999] "The Alpha 21264 microprocessor." *IEEE Micro*, vol.19, (no.2), March-April, 24-36.

KILBURN, T., D. B. G. EDWARDS, M. J. LANIGAN, AND F. H. SUMNER [1962]. "One-level storage system," *IRE Trans. on Electronic Computers* EC-11 (April) 223–235. Also appears in D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples* (1982), McGraw-Hill, New York, 135–148.

KROFT, D. [1981]. "Lockup-free instruction fetch/prefetch cache organization," *Proc. Eighth Annual Symposium on Computer Architecture* (May 12–14), Minneapolis, 81–87.

KROFT, D. [1998]. "Retrospective: Lockup-Free Instruction Fetch/Prefetch Cache Organization."*25 Years of the International Symposia on Computer Architecture (Selected Papers)*. ACM, 20-21.

KUNIMATSU, A., N. IDE, T. SATO, et al. [2000] "Vector unit architecture for emotion synthesis." *IEEE Micro*, vol.20, (no.2), IEEE, March-April, 40-7.

LAM, M. S., E. E. ROTHBERG, AND M. E. WOLF [1991]. "The cache performance and optimizations of blocked algorithms," Fourth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems, Santa Clara, Calif., April 8–11. *SIGPLAN Notices* 26:4 (April), 63–74.

LEBECK, A. R. AND D. A. WOOD [1994]. "Cache profiling and the SPEC benchmarks: A case study," *Computer* 27:10 (October), 15–26.

LIPTAY, J. S. [1968]. "Structural aspects of the System/360 Model 85, Part II: The cache," *IBM Systems J.* 7:1, 15–21.

LUK, C.-K. and T.C MOWRY[1999]. "Automatic compiler-inserted prefetching for pointer-based applications." *IEEE Transactions on Computer*s, vol.48, (no.2), IEEE, Feb. p.134-41.

MCFARLING, S. [1989]. "Program optimization for instruction caches," *Proc. Third Int'l Conf. on*

Architectural Support for Programming Languages and Operating Systems (April 3–6), Boston, 183–191.

MCCALPIN, J.D. [2001]. *STREAM: Sustainable Memory Bandwidth in High Performance Computers* http://www.cs.virginia.edu/stream/.

MOWRY, T. C., S. LAM, AND A. GUPTA [1992]. "Design and evaluation of a compiler algorithm for prefetching," Fifth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V), Boston, October 12–15, *SIGPLAN Notices* 27:9 (September), 62–73.

OKA, M. and M. SUZUOKI. [1999] "Designing and programming the emotion engine." *IEEE Micro*, vol.19, (no.6), Nov.-Dec., 20-8.

PABST, T. [2000], "Performance Showdown at 133 MHz FSB - The Best Platform for Coppermine," http://www6.tomshardware.com/mainboard/00q1/000302/.

PALACHARLA, S. AND R. E. KESSLER [1994]. "Evaluating stream buffers as a secondary cache replacement," *Proc. 21st Annual Int'l Symposium on Computer Architecture*, Chicago, April 18–21, 24–33.

PRZYBYLSKI, S. A. [1990]. *Cache Design: A Performance-Directed Approach,* Morgan Kaufmann Publishers, San Francisco, Calif.

PRZYBYLSKI, S. A., M. HOROWITZ, AND J. L. HENNESSY [1988]. "Performance trade-offs in cache design," *Proc. 15th Annual Symposium on Computer Architecture* (May–June), Honolulu, 290–298.

REINMAN, G. and N. P. JOUPPI. [1999]. "Extensions to CACTI," http://research.compaq.com/wrl/people/jouppi/CACTI.html.

SAAVEDRA-BARRERA, R. H. [1992]. *CPU Performance Evaluation and Execution Time Prediction Using Narrow Spectrum Benchmarking*, Ph.D. Dissertation, University of Calif., Berkeley (May).

SAMPLES, A. D. AND P. N. HILFINGER [1988]. "Code reorganization for instruction caches," Tech. Rep. UCB/CSD 88/447 (October), University of Calif., Berkeley.

SITES, R. L. (ED.) [1992]. *Alpha Architecture Reference Manual,* Digital Press, Burlington, Mass.

SMITH, A. J. [1982]. "Cache memories," *Computing Surveys* 14:3 (September), 473–530.

SMITH, J. E. AND J. R. GOODMAN [1983]. "A study of instruction cache organizations and replacement policies," *Proc. 10th Annual Symposium on Computer Architecture* (June 5–7), Stockholm, 132–137.

STOKES, J. [2000], "Sound and Vision: A Technical Overview of the Emotion Engine," http://arstechnica.com/reviews/1q00/playstation2/ee-1.html.

STRECKER, W. D. [1976]. "Cache memories for the PDP-11?," *Proc. Third Annual Symposium on Computer Architecture* (January), Pittsburgh, 155–158.

Sugumar, R.A. and S.G. Abraham [1993]. "Efficient simulation of caches under optimal replacement with applications to miss characterization." 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems, Santa Clara, CA, USA, 17-21 May, p.24-35.

TORRELLAS, J., A. GUPTA, AND J. HENNESSY [1992]. "Characterizing the caching and synchronization performance of a multiprocessor operating system," Fifth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V), Boston, October 12–15, *SIGPLAN Notices* 27:9 (September), 162–174.

WANG, W.-H., J.-L. BAER, AND H. M. LEVY [1989]. "Organization and performance of a two-level virtual-real cache hierarchy," *Proc. 16th Annual Symposium on Computer Architecture* (May 28–June 1), Jerusalem, 140–148.

WILKES, M. [1965]. "Slave memories and dynamic storage allocation," *IEEE Trans. Electronic Computers* EC-14:2 (April), 270–271.

WILKES, M. V. [1982]. "Hardware support for memory protection: Capability implementations,"

Proc. Symposium on Architectural Support for Programming Languages and Operating Systems (March 1–3), Palo Alto, Calif., 107–116.

WULF, W. A., R. LEVIN, AND S. P. HARBISON [1981]. *Hydra/C.mmp: An Experimental Computer System,* McGraw-Hill, New York.

EXERCISES

- Jouppi comments, 2nd pass: drop prime number exercises; mention that "L2 misses can be overlapped on 21264 when doing block copies; e.g., overlap reading of source block, writing of destination block. Stream benchmark uses this; put 21064 on the web, then still use exercise 5.11.
- Reinhart comments, 2nd pass: "In the prefetching section, it might be interesting to compare & contrast standard prefetches with the speculative load support found in IA-64 (or maybe this would make a good exercise)."
- Technical reviewer, 2nd pass: Suggests deriving components of accesses to arrays X, Y, and Z before and after in blocking example on page 419 to show how came up with "$2N^3 + N^2$ memory words accessed for N^3 operations" before blocking and "total number of memory words accessed is $2N^3/B + N^2$" after blocking. (See page 10 of technical review)
- The CACTI program mentioned in the section allows people to design caches and compare hit times as well as miss rates; I would propose projects that look at both, or possible use an existing exercise and look at the cache access times in different technologies: 0.18, 0.13, 0.10 to see how/if the trade-offs differ. Also, see how access times differ with different sized caches.
- Jouppi said his teaching experience at Stanford was that it was important to show a simple in-order CPU so that simple memory performance questions could be answered using spreadsheets before being exposed to the OOO simulators and traces. Thus he suggests saving the 21064 design in some detail and then include a series of exercises about it so that they can get the ideas here first. Perhaps just include a section in the exercises that describes the 21064 caches, and keep some of the old exercises?
- To really account for the impact of cache misses on OOO computers, there needs to be an OOO instruction simulator to go with the cache. A popular one is Simple Scalar from Doug Berger, originally from Wisconsin. Another is RSIM from Rice. I'd add the URLs and propose exercises which use them.
- Add a discussion question: given an Out-of-order CPU, how do you quantiatively evaluate memory options? Why is it different from an in-order CPU?
- Thus these exercises need to be sorted into whether to assume CPU in-order, and can use spreadsheet since it stalls, or out-of-order, and use a simulator since you cannot account for overlap. Perhaps can simply look at L2 cache

misses, assuming L1 are overlapped?

- One advanced exercise (level [25]) is to study the impact of out of order execution on temporal locality in an L1 data cache. The hypothesis is that there may be more conflict misses due to OOO execution, and that hence multiway set associativity may be more impact for OOO than for in-order CPUs. The idea would be to vary out-of-orderness in terms of the size of the load and store queues and to look at different miss rates as you vary associativity, and see if the CPU execution model makes much difference.
- Some of these examples with a short miss time (e.g., next one where miss is 10X a hit), state that there is a second level cache and for purposes of this equation assume that it doesn't miss, hence it makes sense to talk about "small" miss penalties. Real misses all the way to memory should be no less than 100 clock cycles
- Mark Hill and Norm Jouppi think we should emphasize "misses per 1000 instructions" vs. "miss rate" in this edition, so we need several exercises comparing them. I intend to replace some of the figures, or show it both ways, so this should be supported by exercises using MPI. This is especially true for L2 caches, which is much less confusing.
- Next is a great exercise. I'd just change/modernize the numbers so the answers are different. Perhaps make up another one that is similar, just to have some that aren't in the answer book?

5.1 [15/15/12/12] <5.1,5.2> Let's try to show how you can make *unfair* benchmarks. Here are two computers with the same processor and main memory but different cache organizations. Assume the miss time is 10 times a cache hit time for both computers. Assume writing a 32-bit word takes 5 times as long as a cache hit (for the write-through cache) and that writing a whole 32-byte block takes 10 times as long as a cache-read hit (for the write-back cache). The caches are unified; that is, they contain both instructions and data.

Cache A: 128 sets, two elements per set, each block is 32 bytes, and it uses write through and no-write allocate.

Cache B: 256 sets, one element per set, each block is 32 bytes, and it uses write back and does allocate on write misses.

a. [15] <1.5,5.2> Describe a program that makes computer A run as much faster as possible than computer B. (Be sure to state any further assumptions you need, if any.)

b. [15] <1.5,5.2> Describe a program that makes computer B run as much faster as possible than computer A. (Be sure to state any further assumptions you need, if any.)

c. [12] <1.5,5.2> Approximately how much faster is the program in part (a) on computer A than computer B?

d. [12] <1.5,5.2> Approximately how much faster is the program in part (b) on computer B than on computer A?

- Next is another interesting exercise. Just update graph example. 21264 would be great, if you had access to it.

5.2 [15/10/12/12/12/12/12/12/12/12/12] <5.5,5.4> In this exercise, we will run a program to evaluate the behavior of a memory system. The key is having accurate timing and then having the program stride through memory to invoke different levels of the hierarchy. Below is the code in C for UNIX systems. The first part is a procedure that uses a standard UNIX utility to get an accurate measure of the user CPU time; this procedure may need to change to work on some systems. The second part is a nested loop to read and write memory at different strides and cache sizes. To get accurate cache timing, this code is repeated many times. The third part times the nested loop overhead only so that it can be subtracted from overall measured times to see how long the accesses were. The last part prints the time per access as the size and stride varies. You may need to change `CACHE_MAX` depending on the question you are answering and the size of memory on the system you are measuring. The code below was taken from a program written by Andrea Dusseau of U.C. Berkeley, and was based on a detailed description found in Saavedra-Barrera [1992].

```
#include <stdio.h>
#include <sys/times.h>
#include <sys/types.h>
#include <time.h>
#define CACHE_MIN (1024) /* smallest cache */
#define CACHE_MAX (1024*1024) /* largest cache */
#define SAMPLE 10 /* to get a larger time sample */
#ifndef CLK_TCK
#define CLK_TCK 60 /* number clock ticks per second */
#endif
int x[CACHE_MAX]; /* array going to stride through */

double get_seconds() { /* routine to read time */
    struct tms rusage;
    times(&rusage); /* UNIX utility: time in clock ticks */
    return (double) (rusage.tms_utime)/CLK_TCK;
}
void main() {
int register i, index, stride, limit, temp;
int steps, tsteps, csize;
double sec0, sec; /* timing variables */

for (csize=CACHE_MIN; csize <= CACHE_MAX; csize=csize*2)
    for (stride=1; stride <= csize/2; stride=stride*2) {
        sec = 0; /* initialize timer */
        limit = csize-stride+1; /* cache size this loop */

        steps = 0;
            do { /* repeat until collect 1 second */
        sec0 = get_seconds(); /* start timer */
        for (i=SAMPLE*stride;i!=0;i=i-1) /* larger sample */
          for (index=0; index < limit; index=index+stride)
                x[index] = x[index] + 1; /* cache access */
        steps = steps + 1; /* count while loop iterations */
        sec = sec + (get_seconds() - sec0);/* end timer */
```

```
        } while (sec < 1.0); /* until collect 1 second */

        /* Repeat empty loop to subtract loop overhead */
        tsteps = 0; /* used to match no. while iterations */
            do { /* repeat until same no. iterations as above */
        sec0 = get_seconds(); /* start timer */
        for (i=SAMPLE*stride;i!=0;i=i-1) /* larger sample */
          for (index=0; index < limit; index=index+stride)
                temp = temp + index; /* dummy code */
        tsteps = tsteps + 1; /* count while iterations */
        sec = sec - (get_seconds() - sec0);/* - overhead */
        } while (tsteps<steps); /* until = no. iterations */

        printf("Size:%7d Stride:%7d read+write:%l4.0f ns\n",
            csize*sizeof(int), stride*sizeof(int), (double)
        sec*1e9/(steps*SAMPLE*stride*((limit-1)/stride+1)));
    }; /* end of both outer for loops */
}
```

The program above assumes that program addresses track physical addresses, which is true on the few computers that use virtually addressed caches such as the Alpha 21264. In general, virtual addresses tend to follow physical addresses shortly after rebooting, so you may need to reboot the computer in order to get smooth lines in your results.

To answer the questions below, assume that the sizes of all components of the memory hierarchy are powers of 2.

a. [15] <5.5,5.4> Plot the experimental results with elapsed time on the y-axis and the memory stride on the x-axis. Use logarithmic scales for both axes, and draw a line for each cache size.

b. [10] <5.5,5.4> How many levels of cache are there?

c. [12] <5.5,5.4> What is the size of the first-level cache? Block size? *Hint*: Assume the size of the page is much larger than the size of a block in a secondary cache (if any), and the size of a second-level cache block is greater than or equal to the size of a block in a first-level cache.

d. [12] <5.5,5.4> What is the size of the second-level cache (if any)? Block size?

e. [12] <5.5,5.4> What is the associativity of the first-level cache? Second-level cache?

f. [12] <5.5,5.4> What is the page size?

g. [12] <5.5,5.4> How many entries are in the TLB?

h. [12] <5.5,5.4> What is the miss penalty for the first-level cache? Second-level?

i. [12] <5.5,5.4> What is the time for a page fault to secondary memory? *Hint*: A page fault to magnetic disk should be measured in milliseconds.

j. [12] <5.5,5.4> What is the miss penalty for the TLB?

k. [12] <5.5,5.4> Is there anything else you have discovered about the memory hierarchy from these measurements?

- Replace Figure below with a newer computer, perhaps by getting results from someone who has run this on a recent computer, or worst case from the paper by Saveerda and Smith.

5.3 [10/10/10] <5.2> Figure 5.59 shows the output from running the program in Exercise 5.2 on a SPARCstation 1+, which has a single unified cache.

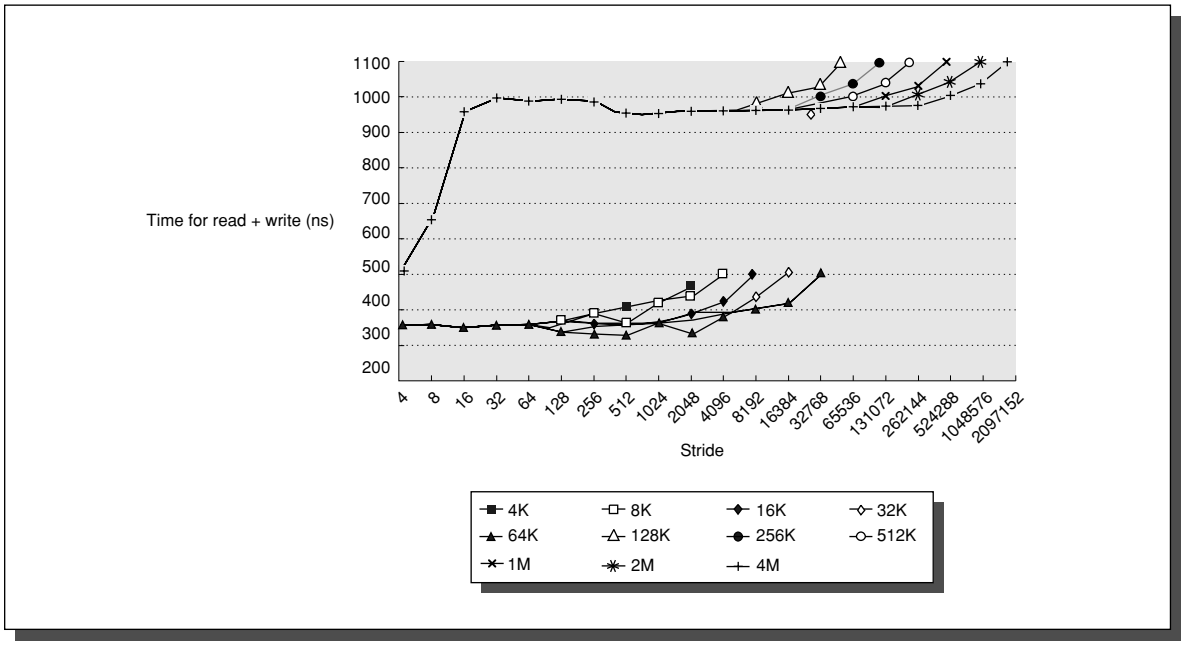


FIGURE 5.59 Results of running program in Exercise 5.2 on a SPARCstation 1+.

a. [10] <5.2> What is the size of the cache?

b. [10] <5.2> What is the block size of the cache?

c. [10] <5.2> What is the miss penalty for the first-level cache?

5.4 [15/15] <5.2> You purchased an Acme computer with the following features:

- 95% of all memory accesses are found in the cache.
- Each cache block is two words, and the whole block is read on any miss.
- The processor sends references to its cache at the rate of 10^9 words per second.
- 25% of those references are writes.
- Assume that the memory system can support 10^9 words per second, reads or writes.
- The bus reads or writes a single word at a time (the memory system cannot read or write two words at once).

- Assume at any one time, 30% of the blocks in the cache have been modified.
- The cache uses write allocate on a write miss.

You are considering adding a peripheral to the system, and you want to know how much of the memory system bandwidth is already used. Calculate the percentage of memory system bandwidth used on the average in the two cases below. Be sure to state your assumptions.

a. [15] <5.2> The cache is write through.

b. [15] <5.2> The cache is write back.

5.5 [15/15] <5.7> One difference between a write-through cache and a write-back cache can be in the time it takes to write. During the first cycle, we detect whether a hit will occur, and during the second (assuming a hit) we actually write the data. Let's assume that 50% of the blocks are dirty for a write-back cache. For this question, assume that the write buffer for write through will never stall the CPU (no penalty). Assume a cache read hit takes 1 clock cycle, the cache miss penalty is 50 clock cycles, and a block write from the cache to main memory takes 50 clock cycles. Finally, assume the instruction cache miss rate is 0.5% and the data cache miss rate is 1%.

a. [15] <5.7> Using statistics for the average percentage of loads and stores from MIPS in Figure 2.32 on page 149, estimate the performance of a write-through cache with a two-cycle write versus a write-back cache with a two-cycle write for each of the programs.

b. [15] <5.7> Do the same comparison, but this time assume the write-through cache pipelines the writes, so that a write hit takes just one clock cycle.

5.6 [20] <5.5> Improve on the compiler prefetch Example found on page 425: Try to eliminate both the number of extraneous prefetches and the number of non-prefetched cache misses. Calculate the performance of this refined version using the parameters in the Example.

- The following example isn't there any more

5.7 [15/12] <5.5> The example evaluation of a pseudo-associative cache on page 399 assumed that on a hit to the slower block the hardware swapped the contents with the corresponding fast block so that subsequent hits on this address would all be to the fast block. Assume that if we don't swap, a hit in the slower block takes just one extra clock cycle instead of two extra clock cycles.

a. [15] <5.5> Derive a formula for the average memory access time using the terminology for direct-mapped and two-way set-associative caches as given on page 399.

b. [12] <5.5> Using the formula from part (a), recalculate the average memory access times for the two cases found on page 399 (8-KB cache and 256-KB cache). Which pseudo-associative scheme is faster for the given configurations and data?

- Perhaps next is a good in-order v. out-of-order exercise?

5.8 [15/20/15] <5.10> If the base CPI with a perfect memory system is 1.5, what is the CPI for these cache organizations? Use Figure 5.14 (page 409):

- 16-KB direct-mapped unified cache using write back.

- 16-KB two-way set-associative unified cache using write back.
- 32-KB direct-mapped unified cache using write back.

Assume the memory latency is 40 clocks, the transfer rate is 4 bytes per clock cycle and that 50% of the transfers are dirty. There are 32 bytes per block and 20% of the instructions are data transfer instructions. There is no write buffer. Add to the assumptions above a TLB that takes 20 clock cycles on a TLB miss. A TLB does not slow down a cache hit. For the TLB, make the simplifying assumption that 0.2% of all references aren't found in TLB, either when addresses come directly from the CPU or when addresses come from cache misses.

a. [15] <5.5> Compute the effective CPI for the three caches assuming an ideal TLB.

b. [20] <5.5> Using the results from part (a), compute the effective CPI for the three caches with a real TLB.

c. [15] <5.5> What is the impact on performance of a TLB if the caches are virtually or physically addressed?

5.9 [10] <5.4> What is the formula for average access time for a three-level cache?

- prime number of memory banks were dropped for the 3/e. Thus we must modify Exercise 5.10 which refers to the prime number of banks, at least to explain the ideas. Perhaps even add exercises, including an explanation of the ideas dropped from the second edition by using text from the second edition?

5.10 [15/15] <5.8> The section on avoiding bank conflicts by having a prime number of memory banks mentioned that there are techniques for fast modulo arithmetic, especially when the prime number can be represented as $2^N - 1$. The idea is that by understanding the laws of modulo arithmetic we can simplify the hardware. The key insights are the following:

1. Modulo arithmetic obeys the laws of distribution:

   $((a \text{ modulo } c) + (b \text{ modulo } c)) \text{ modulo } c = (a + b) \text{ modulo } c$
   $((a \text{ modulo } c) \times (b \text{ modulo } c)) \text{ modulo } c = (a \times b) \text{ modulo } c$

2. The sequence 2^0 modulo $2^N - 1$, 2^1 modulo $2^N - 1$, 2^2 modulo $2^N - 1$, . . . is a repeating pattern 2^0, 2^1, 2^2, and so on for powers of 2 less than 2^N. For example, if $2^N - 1 = 7$, then

   2^0 modulo 7 = 1
   2^1 modulo 7 = 2
   2^2 modulo 7 = 4
   2^3 modulo 7 = 1
   2^4 modulo 7 = 2
   2^5 modulo 7 = 4

3. Given a binary number a, the value of (a mod 7) can be expressed as

   $a_i \times 2^i + \ldots + a_2 \times 2^2 + a_1 \times 2^1 + a_0 \times 2^0 \text{ modulo } 7 =$
   $((a_0 + a_3 + \ldots) \times 1 + (a_1 + a_4 + \ldots) \times 2 + (a_2 + a_5 + \ldots) \times 4) \text{ modulo } 7$

   where $i = \log_2 a$ and $a_j = 0$ for $j > i$

   This is possible because 7 is a prime number of the form $2^N - 1$. Since the multiplica-

tions in the expression above are by powers of two, they can be replaced by binary shifts (a very fast operation).

4. The address is now small enough to find the modulo by looking it up in a read-only memory (ROM) to get the bank number.

Finally, we are ready for the questions.

a. [15] <5.8> Given $2^N - 1$ memory banks, what is the approximate reduction in size of an address that is M bits wide as a result of the intermediate result in step 3 above? Give the general formula, and then show the specific case of $N = 3$ and $M = 32$.

b. [15] <5.8> Draw the block structure of the hardware that would pick the correct bank out of seven banks given a 32-bit address. Assume that each bank is 8 bytes wide. What is the size of the adders and ROM used in this organization?

n Old, so drop?

5.11 [25/10/15] <5.8> The CRAY X-MP instruction buffers can be thought of as an instruction-only cache. The total size is 1 KB, broken into four blocks of 256 bytes per block. The cache is fully associative and uses a first-in, first-out replacement policy. The access time on a miss is 10 clock cycles, with the transfer time of 64 bytes every clock cycle. The X-MP takes 1 clock cycle on a hit. Use the cache simulator to determine the following:

a. [25] <5.8> Instruction miss rate.

b. [10] <5.8> Average instruction memory access time measured in clock cycles.

c. [15] <5.8> What does the CPI of the CRAY X-MP have to be for the portion due to instruction cache misses to be 10% or less?

n The next 5 exercises all refer to traces. Since we no longer have traces readily available, these exercises should be changed to work with something like Burger's simulator running a program and producing addresses vs. an address trace, unless there are some traces left online someplace?

5.12 [25] <5.8> Traces from a single process give too high estimates for caches used in a multiprocess environment. Write a program that merges the uniprocess DLX traces into a single reference stream. Use the process-switch statistics in Figure 5.25 (page 432) as the average process-switch rate with an exponential distribution about that mean. (Use the number of clock cycles rather than instructions, and assume the CPI of DLX is 1.5.) Use the cache simulator on the original traces and the merged trace. What is the miss rate for each, assuming a 64-KB direct-mapped cache with 16-byte blocks? (There is a process-identified tag in the cache tag so that the cache doesn't have to be flushed on each switch.)

5.13 [25] <5.8> One approach to reducing misses is to prefetch the next block. A simple but effective strategy, found in the Alpha 21064, is when block i is referenced to make sure block $i + 1$ is in the cache, and if not, to prefetch it. Do you think automatic prefetching is more or less effective with increasing block size? Why? Is it more or less effective with increasing cache size? Why? Use statistics from the cache simulator and the traces to support your conclusion.

5.14 [20/25] <5.8> Smith and Goodman [1983] found that for a *small instruction* cache, a

cache using direct mapping could consistently outperform one using fully associative with LRU replacement.

a. [20] <5.8> Explain why this would be possible. (*Hint*: You can't explain this with the three C's model because it ignores replacement policy.)

b. [25] <5.8> Use the cache simulator to see if their results hold for the traces.

5.15 [30] <5.10> Use the cache simulator and traces to calculate the effectiveness of a four-bank versus eight-bank interleaved memory. Assume each word transfer takes one clock on the bus and a random access is eight clocks. Measure the bank conflicts and memory bandwidth for these cases:

a. <5.10> No cache and no write buffer.

b. <5.10> A 64-KB direct-mapped write-through cache with four-word blocks.

c. <5.10> A 64-KB direct-mapped write-back cache with four-word blocks.

d. <5.10> A 64-KB direct-mapped write-through cache with four-word blocks but the "interleaving" comes from a page-mode DRAM.

e. <5.10> A 64-KB direct-mapped write-back cache with four-word blocks but the "interleaving" comes from a page-mode DRAM.

5.16 [25/25/25] <5.10> Use a cache simulator and traces to calculate the effectiveness of early restart and out-of-order fetch. What is the distribution of first accesses to a block as block size increases from 2 words to 64 words by factors of two for the following:

a. [25] <5.10> A 64-KB instruction-only cache?

b. [25] <5.10> A 64-KB data-only cache?

c. [25] <5.10> A 128-KB unified cache?

Assume direct-mapped placement.

5.17 [25/25/25/25/25/25] <5.2> Use a cache simulator and traces with a program you write yourself to compare the effectiveness of these schemes for fast writes:

a. [25] <5.2> One-word buffer and the CPU stalls on a data-read cache miss with a write-through cache.

b. [25] <5.2> Four-word buffer and the CPU stalls on a data-read cache miss with a write-through cache.

c. [25] <5.2> Four-word buffer and the CPU stalls on a data-read cache miss only if there is a potential conflict in the addresses with a write-through cache.

d. [25] <5.2> A write-back cache that writes dirty data first and then loads the missed block.

e. [25] <5.2> A write-back cache with a one-block write buffer that loads the miss data first and then stalls the CPU on a clean miss if the write buffer is not empty.

f. [25] <5.2> A write-back cache with a one-block write buffer that loads the miss data first and then stalls the CPU on a clean miss only if the write buffer is not empty and

there is a potential conflict in the addresses.

Assume a 64-KB direct-mapped cache for data and a 64-KB direct-mapped cache for instructions with a block size of 32 bytes. The CPI of the CPU is 1.5 with a perfect memory system and it takes 14 clocks on a cache miss and 7 clocks to write a single word to memory.

5.18 [25] <5.4> Using the UNIX pipe facility, connect the output of one copy of the cache simulator to the input of another. Use this pair to see at what cache size the global miss rate of a second-level cache is approximately the same as a single-level cache of the same capacity for the traces provided.

5.19 [Discussion] <5.10> Second-level caches now contain several megabytes of data. Although new TLBs provide for variable length pages to try to map more memory, most operating systems do not take advantage of them. Does it make sense to miss the TLB on data that are found in a cache? How should TLBs be reorganized to avoid such misses?

5.20 [Discussion] <5.10> Some people have argued that with increasing capacity of memory storage per chip, virtual memory is an idea whose time has passed, and they expect to see it dropped from future computers. Find reasons for and against this argument.

5.21 [Discussion] <5.10> So far, few computer systems take advantage of the extra security available with gates and rings found in a CPU like the Intel Pentium. Construct some scenario whereby the computer industry would switch over to this model of protection.

5.22 [Discussion] <5.17> Many times a new technology has been invented that is expected to make a major change to the memory hierarchy. For the sake of this question, let's suppose that biological computer technology becomes a reality. Suppose biological memory technology has the following unusual characteristic: It is as fast as the fastest semiconductor DRAMs and it can be randomly accessed, but its per byte costs are the same as magnetic disk memory. It has the further advantage of not being any slower no matter how big it is. The only drawback is that you can only write it once, but you can read it many times. Thus it is called a *WORM* (write once, read many) memory. Because of the way it is manufactured, the WORM memory module can be easily replaced. See if you can come up with several new ideas to take advantage of WORMs to build better computers using "biotechnology."

5.23 [Discussion] <3,4,5> Chapters 3 and 4 showed how execution time is being reduced by pipelining and by superscalar and VLIW organizations: even floating-point operations may account for only a fraction of a clock cycle in total execution time. On the other hand, Figure 5.2 on page 375 shows that the memory hierarchy is increasing in importance. The research on algorithms, data structures, operating systems, and even compiler optimizations were done in an era of simpler computers, with no pipelining or caches. Classes and textbooks may still reflect those simpler computers. What is the impact of the changes in computer architecture on these other fields? Find examples where textbooks suggest the solution appropriate for old computers but inappropriate for modern computers. Talk to people in other fields to see what they think about these changes.

6 Multiprocessors and Thread-Level Parallelism

The turning away from the conventional organization came in the middle 1960s, when the law of diminishing returns began to take effect in the effort to increase the operational speed of a computer. … Electronic circuits are ultimately limited in their speed of operation by the speed of light… and many of the circuits were already operating in the nanosecond range.

Bouknight et al., *The Illiac IV System* [1972]

… sequential computers are approaching a fundamental physical limit on their potential computational power. Such a limit is the speed of light...

A. L. DeCegama, *The Technology of Parallel Processing, Volume I* (1989)

… today's multiprocessors… are nearing an impasse as technologies approach the speed of light. Even if the components of a sequential processor could be made to work this fast, the best that could be expected is no more than a few million instructions per second.

Mitchell, *The Transputer: The Time Is Now* [1989]

6.1	Introduction	635
6.2	Characteristics of Application Domains	649
6.3	Symmetric Shared-Memory Architectures	658
6.4	Performance of Symmetric Shared-Memory Multiprocessors	670
6.5	Distributed Shared-Memory Architectures	687
6.6	Performance of Distributed Shared-Memory Multiprocessors	697
6.7	Synchronization	705
6.8	Models of Memory Consistency: An Introduction	719
6.9	Multithreading: Exploiting Thread-Level Parallelism within a Processor	723
6.10	Crosscutting Issues	728
6.11	Putting It All Together: Sun's Wildfire Prototype	735
6.13	Another View: Embedded Multiprocessors	751
6.14	Fallacies and Pitfalls	752
6.15	Concluding Remarks	758
6.16	Historical Perspective and References	765
	Exercises	780

Major changes

1. split up the longest sections
2. clearer discussion of the concept of thread and process
3. SMT and multithreading section
4. two another views
5. reordered the cross cutting issues--no big changes, just reordered

6.1 Introduction

As the quotations that open this chapter show, the view that advances in uniprocessor architecture were nearing an end has been widely held at varying times. To counter this view, we observe that during the period 1985–2000, uni-

processor performance growth, driven by the microprocessor, was at its highest rate since the first transistorized computers in the late 1950s and early 1960s.

On balance, though, your authors believe that parallel processors will definitely have a bigger role in the future. This view is driven by three observations. First, since microprocessors are likely to remain the dominant uniprocessor technology, the logical way to improve performance beyond a single processor is by connecting multiple microprocessors together. This combination is likely to be more cost-effective than designing a custom processor. Second, it is unclear whether the pace of architectural innovation that has been based for more than fifteen years on increased exploitation of instruction-level parallelism can be sustained indefinitely. As we saw in Chapters 3 and 4, modern multiple-issue processors have become incredibly complex, and the increases achieved in performance for increasing complexity, increasing silicon, and increasing power seem to be diminishing. Third, there appears to be slow but steady progress on the major obstacle to widespread use of parallel processors, namely software. This progress is probably faster in the server and embedded markets, as we discussed in Chapter 3 and 4. Server and embedded applications exhibit natural parallelism that can be exploited without some of the burdens of rewriting a gigantic software base. This is more of a challenge in the desktop space.

Your authors, however, are extremely reluctant to predict the death of advances in uniprocessor architecture. Indeed, we believe that the rapid rate of performance growth will continue at least for the next five years. Whether this pace of innovation can be sustained longer is difficult to predict but hard to bet against. Nonetheless, if the pace of progress in uniprocessors does slow down, multiprocessor architectures will become increasingly attractive.

That said, we are left with two problems. First, multiprocessor architecture is a large and diverse field, and much of the field is in its youth, with ideas coming and going and, until very recently, more architectures failing than succeeding. Given that we are already on page 636, full coverage of the multiprocessor design space and its trade-offs would require another volume. (Indeed, Culler, Singh, and Gupta [1999] cover *only* multiprocessors in their 1000 page book!) Second, such coverage would necessarily entail discussing approaches that may not stand the test of time, something we have largely avoided to this point. For these reasons, we have chosen to focus on the mainstream of multiprocessor design: multiprocessors with small to medium numbers of processors (≤128). Such designs vastly dominate in terms of both units and dollars. We will pay only slight attention to the larger-scale multiprocessor design space (≥128 processors). At the present, the future architecture of such multiprocessors is unsettled and even the viability of that marketplace is in doubt. We will return to this topic briefly at the end of the chapter, in section 6.15.

Edited by Foxit Reader
Copyright(C) by Foxit Software Company,2005-2008
For Evaluation Only.

A Taxonomy of Parallel Architectures

We begin this chapter with a taxonomy so that you can appreciate both the breadth of design alternatives for multiprocessors and the context that has led to the development of the dominant form of multiprocessors. We briefly describe the alternatives and the rationale behind them; a longer description of how these different models were born (and often died) can be found in the historical perspectives at the end of the chapter.

The idea of using multiple processors both to increase performance and to improve availability dates back to the earliest electronic computers. About 30 years ago, Flynn proposed a simple model of categorizing all computers that is still useful today. He looked at the parallelism in the instruction and data streams called for by the instructions at the most constrained component of the multiprocessor, and placed all computers in one of four categories:

1. *Single instruction stream, single data stream* (SISD)—This category is the uniprocessor.
2. *Single instruction stream, multiple data streams* (SIMD)—The same instruction is executed by multiple processors using different data streams. Each processor has its own data memory (hence multiple data), but there is a single instruction memory and control processor, which fetches and dispatches instructions. The multimedia extensions we considered in Chapter 2 are a limited form of SIMD parallelism. Vector architectures are the largest class of processors of this type.
3. *Multiple instruction streams, single data stream* (MISD)—No commercial multiprocessor of this type has been built to date, but may be in the future. Some special purpose stream processors approximate a limited form of this (there is only a single data stream that is operated on by successive functional units).
4. *Multiple instruction streams, multiple data streams* (MIMD)—Each processor fetches its own instructions and operates on its own data. The processors are often off-the-shelf microprocessors.

This is a coarse model, as some multiprocessors are hybrids of these categories. Nonetheless, it is useful to put a framework on the design space.

As discussed in the historical perspectives, many of the early multiprocessors were SIMD, and the SIMD model received renewed attention in the 1980s, and except for vector processors, was gone by the mid 1990s. MIMD has clearly emerged as the architecture of choice for general-purpose multiprocessors. Two factors are primarily responsible for the rise of the MIMD multiprocessors:

1. MIMDs offer flexibility. With the correct hardware and software support, MIMDs can function as single-user multiprocessors focusing on high performance for one application, as multiprogrammed multiprocessors running many tasks simultaneously, or as some combination of these functions.
2. MIMDs can build on the cost/performance advantages of off-the-shelf microprocessors. In fact, nearly all multiprocessors built today use the same microprocessors found in workstations and single-processor servers.

With an MIMD, each processor is executing its own instruction stream. In many cases, each processor executes a different process. Recall from the last chapter, that a process is an segment of code that may be run independently, and that the state of the process contains all the information necessary to execute that program on a processor. In a multiprogrammed environment, where the processors may be running independent tasks, each process is typically independent of the processes on other processors.

It is also useful to be able to have multiple processors executing a single program and sharing the code and most of their address space. When multiple processes share code and data in this way, they are often called *threads*. Today, the term thread is often used in a casual way to refer to multiple loci of execution that may run on different processors, even when they do not share an address space.

To take advantage of an MIMD multiprocessor with n processors, we must usually have at least n threads or processes to execute. The independent threads are typically identified by the programmer or created by the compiler. Since the parallelism in this situation is contained in the threads, it is called *thread-level parallelism*.

Threads may vary from large-scale, independent processes–for example, independent programs running in a multiprogrammed fashion on different processors–to parallel iterations of a loop, automatically generated by a compiler and each executing for perhaps less than a thousand instructions. Although the size of a thread is important in considering how to exploit thread-level parallelism efficiently, the important qualitative distinction is that such parallelism is identified at a high-level by the software system and that the threads consist of hundreds to millions of instructions that may be executed in parallel. In contrast, instruction-level parallelism is identified by primarily by the hardware, though with software help in some cases, and is found and exploited one instruction at a time.

Existing MIMD multiprocessors fall into two classes, depending on the number of processors involved, which in turn dictate a memory organization and interconnect strategy. We refer to the multiprocessors by their memory organization, because what constitutes a small or large number of processors is likely to change over time.

Edited by Foxit Reader
Copyright(C) by Foxit Software Company,2005-2008
For Evaluation Only.

The first group, which we call *centralized shared-memory architectures*, have at most a few dozen processors in 2000. For multiprocessors with small processor counts, it is possible for the processors to share a single centralized memory and to interconnect the processors and memory by a bus. With large caches, the bus and the single memory, possibly with multiple banks, can satisfy the memory demands of a small number of processors. By replacing a single bus with multiple buses, or even a switch, a centralized shared memory design can be scaled to a few dozen processors. Although scaling beyond that is technically conceivable, sharing a centralized memory, even organized as multiple banks, becomes less attractive as the number of processors sharing it increases.

Because there is a single main memory that has a symmetric relationship to all processors and a uniform access time from any processor, these multiprocessors are often called *symmetric (shared-memory) multiprocessors* (*SMPs*), and this style of architecture is sometimes called *UMA* for *uniform memory access*. This type of centralized shared-memory architecture is currently by far the most popular organization. Figure 6.1 shows what these multiprocessors look like. The architecture of such multiprocessors is the topic of section 6.3.

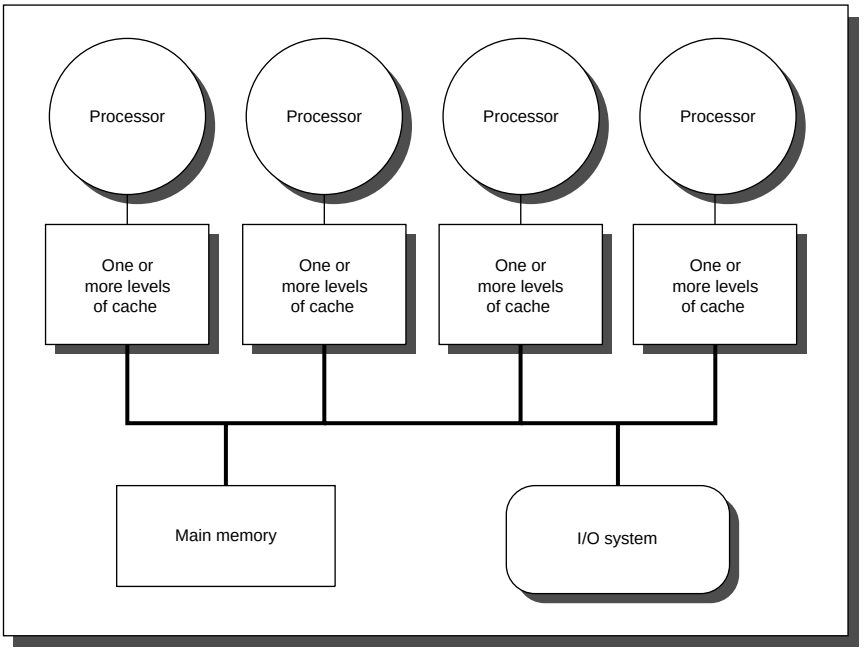


FIGURE 6.1 Basic structure of a centralized shared-memory multiprocessor. Multiple processor-cache subsystems share the same physical memory, typically connected by a bus. In larger designs, multiple buses, or even a switch may be used, but the key architectural property: uniform access time o all memory from all processors remains.

Edited by Foxit Reader
Copyright(C) by Foxit Software Company,2005-2008
For Evaluation Only.

The second group consists of multiprocessors with physically distributed memory. To support larger processor counts, memory must be distributed among the processors rather than centralized; otherwise the memory system would not be able to support the bandwidth demands of a larger number of processors without incurring excessively long access latency. With the rapid increase in processor performance and the associated increase in a processor's memory bandwidth requirements, the scale of multiprocessor for which distributed memory is preferred over a single, centralized memory continues to decrease in number (which is another reason not to use small and large scale). Of course, the larger number of processors raises the need for a high bandwidth interconnect, of which we saw examples in Chapter 7. Both direct interconnection networks (i.e., switches) and indirect networks (typically multidimensional meshes) are used. Figure 6.2 shows what

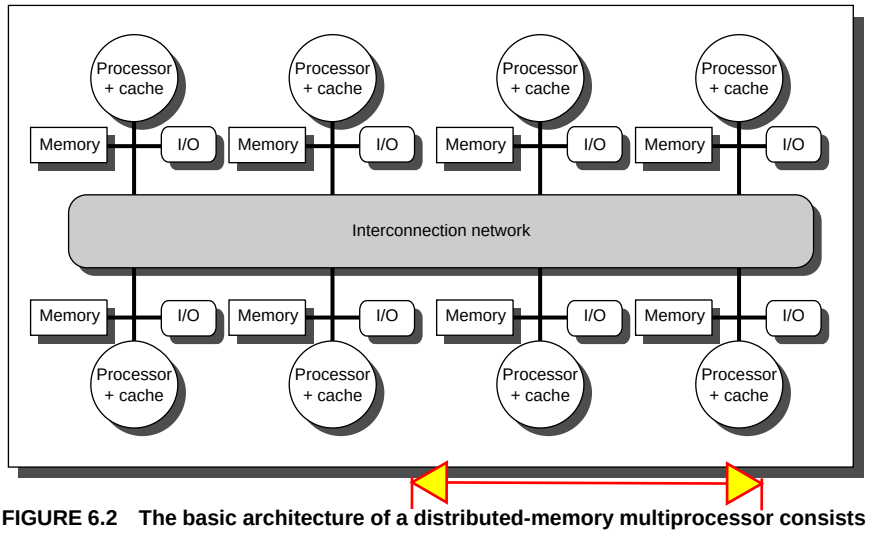


FIGURE 6.2 The basic architecture of a distributed-memory multiprocessor consists of individual nodes containing a processor, some memory, typically some I/O, and an interface to an interconnection network that connects all the nodes. Individual nodes may contain a small number of processors, which may be interconnected by a small bus or a different interconnection technology, which is less scalable than the global interconnection network.

these multiprocessors look like.

Distributing the memory among the nodes has two major benefits. First, it is a cost-effective way to scale the memory bandwidth, if most of the accesses are to the local memory in the node. Second, it reduces the latency for accesses to the local memory. These two advantages make distributed memory attractive at smaller processor counts as processors get ever faster and require more memory

bandwidth and lower memory latency. The key disadvantage for a distributed memory architecture is that communicating data between processors becomes somewhat more complex and has higher latency, at least when there is no contention, because the processors no longer share a single centralized memory. As we will see shortly, the use of distributed memory leads to two different paradigms for interprocessor communication.

Typically, I/O as well as memory is distributed among the nodes of the multiprocessor, and the nodes may be small SMPs (2–8 processors). Although the use of multiple processors in a node together with a memory and a network interface may be quite useful from a cost-efficiency viewpoint, it is not fundamental to how these multiprocessors work, and so we will focus on the one-processor-per-node design for most of this chapter.

Models for Communication and Memory Architecture

As discussed earlier, any large-scale multiprocessor must use multiple memories that are physically distributed with the processors. There are two alternative architectural approaches that differ in the method used for communicating data among processors.

In the first method, communication occurs through a shared address space. That is, the physically separate memories can be addressed as one logically shared address space, meaning that a memory reference can be made by any processor to any memory location, assuming it has the correct access rights. These multiprocessors are called *distributed shared-memory* (*DSM*) architectures. The term *shared memory* refers to the fact that the *address space* is shared; that is, the same physical address on two processors refers to the same location in memory. Shared memory does *not* mean that there is a single, centralized memory. In contrast to the symmetric shared-memory multiprocessors, also known as UMAs (uniform memory access), the DSM multiprocessors are also called *NUMAs*, *non-uniform memory access*, since the access time depends on the location of a data word in memory.

Alternatively, the address space can consist of multiple private address spaces that are logically disjoint and cannot be addressed by a remote processor. In such multiprocessors, the same physical address on two different processors refers to two different locations in two different memories. Each processor-memory module is essentially a separate computer; therefore these parallel processors have been called *multicomputers*. As pointed out in the previous chapter, a multicomputer can even consist of completely separate computers connected on a local area network, which, today, are popularly called *clusters*. For applications that require little or no communication and can make use of separate memories, such clusters of processors, whether using a standardized or customized interconnect, can form a very cost-effective approach (see Section 7.x).

With each of these organizations for the address space, there is an associated communication mechanism. For a multiprocessor with a shared address space, that address space can be used to communicate data implicitly via load and store operations; hence the name *shared memory* for such multiprocessors. For a multiprocessor with multiple address spaces, communication of data is done by explicitly passing messages among the processors. Therefore, these multiprocessors are often called *message passing multiprocessors*.

In message passing multiprocessors, communication occurs by sending messages that request action or deliver data just as with the network protocols discussed in section 7.2. For example, if one processor wants to access or operate on data in a remote memory, it can send a message to request the data or to perform some operation on the data. In such cases, the message can be thought of as a *remote procedure call* (*RPC*). When the destination processor receives the message, either by polling for it or via an interrupt, it performs the operation or access on behalf of the remote processor and returns the result with a reply message. This type of message passing is also called *synchronous,* since the initiating processor sends a request and waits until the reply is returned before continuing. Software systems have been constructed to encapsulate the details of sending and receiving messages, including passing complex arguments or return values, presenting a clean RPC facility to the programmer.

Communication can also occur from the viewpoint of the writer of data rather than the reader, and this can be more efficient when the processor producing data knows which other processors will need the data. In such cases, the data can be sent directly to the consumer process without having to be requested first. It is often possible to perform such message sends asynchronously, allowing the sender process to continue immediately. Often the receiver will want to block if it tries to receive the message before it has arrived; in other cases, the reader may check whether a message is pending before actually trying to perform a blocking receive. Also the sender must be prepared to block if the receiver has not yet consumed an earlier message and no buffer space is available. The message passing facilities offered in different multiprocessors are fairly diverse. To ease program portability, standard message passing libraries (for example, message passing interface, or MPI) have been proposed. Such libraries sacrifice some performance to achieve a common interface.

Performance Metrics for Communication Mechanisms

Three performance metrics are critical in any communication mechanism:

1. *Communication bandwidth*—Ideally the communication bandwidth is limited by processor, memory, and interconnection bandwidths, rather than by some aspect of the communication mechanism. The bisection bandwidth (see Section 7.x) is determined by the interconnection network. The bandwidth in or

out of a single node, which is often as important as bisection bandwidth, is affected both by the architecture within the node and by the communication mechanism. How does the communication mechanism affect the communication bandwidth of a node? When communication occurs, resources within the nodes involved in the communication are tied up or occupied, preventing other outgoing or incoming communication. When this occupancy is incurred for each word of a message, it sets an absolute limit on the communication bandwidth. This limit is often lower than what the network or memory system can provide. Occupancy may also have a component that is incurred for each communication event, such as an incoming or outgoing request. In the latter case, the occupancy limits the communication rate, and the impact of the occupancy on overall communication bandwidth depends on the size of the messages.

2. *Communication latency*—Ideally the latency is as low as possible. As we will see in Chapter 8, communication latency is equal to

   Sender overhead + Time of flight + Transmission time + Receiver overhead

   Time of flight is fixed and transmission time is determined by the interconnection network. The software and hardware overheads in sending and receiving messages are largely determined by the communication mechanism and its implementation. Why is latency crucial? Latency affects both performance and how easy it is to program a multiprocessor. Unless latency is hidden, it directly affects performance either by tying up processor resources or by causing the processor to wait. Overhead and occupancy are closely related, since many forms of overhead also tie up some part of the node, incurring an occupancy cost, which in turn limits bandwidth. Key features of a communication mechanism may directly affect overhead and occupancy. For example, how is the destination address for a remote communication named, and how is protection implemented? When naming and protection mechanisms are provided by the processor, as in a shared address space, the additional overhead is small. Alternatively, if these mechanisms must be provided by the operating system for each communication, this increases the overhead and occupancy costs of communication, which in turn reduce bandwidth and increase latency.

3. *Communication latency hiding*—How well can the communication mechanism hide latency by overlapping communication with computation or with other communication? Although measuring this is not as simple as measuring the first two metrics, it is an important characteristic that can be quantified by measuring the running time on multiprocessors with the same communication latency but different support for latency hiding. We will see examples of latency hiding techniques for shared memory in sections 6.8 and 6.10. Although hiding latency is certainly a good idea, it poses an additional burden on the software system and ultimately on the programmer. Furthermore, the amount of latency that can be hidden is application dependent. Thus, it is usually best to reduce latency wherever possible.

Each of these performance measures is affected by the characteristics of the communications needed in the application. The size of the data items being communicated is the most obvious, since it affects both latency and bandwidth in a direct way, as well as affecting the efficacy of different latency hiding approaches. Similarly, the regularity in the communication patterns affects the cost of naming and protection, and hence the communication overhead. In general, mechanisms that perform well with smaller as well as larger data communication requests, and irregular as well as regular communication patterns, are more flexible and efficient for a wider class of applications. Of course, in considering any communication mechanism, designers must consider cost as well as performance.

Advantages of Different Communication Mechanisms

Each of these two primary communication mechanisms has its advantages. For shared-memory communication, advantages include

- Compatibility with the well-understood mechanisms in use in centralized multiprocessors, which all use shared-memory communication.
- Ease of programming when the communication patterns among processors are complex or vary dynamically during execution. Similar advantages simplify compiler design.
- The ability to develop applications using the familiar shared-memory model, focusing attention only on those accesses that are performance critical.
- Lower overhead for communication and better use of bandwidth when communicating small items. This arises from the implicit nature of communication and the use of memory mapping to implement protection in hardware, rather than through the I/O system.
- The ability to use hardware-controlled caching to reduce the frequency of remote communication by supporting automatic caching of all data, both shared and private. As we will see, caching reduces both latency and contention for accessing shared data. This advantage also comes with a disadvantage, which we mention below.

The major advantages for message-passing communication include

- The hardware can be simpler, especially by comparison with a scalable shared-memory implementation that supports coherent caching of remote data.
- Communication is explicit, which means it is simpler to understand; in shared memory models, it can be difficult to know when communication is occurring and when it is not, as well as how costly the communication is.

- Explicit communication focuses programmer attention on this costly aspect of parallel computation, sometimes leading to improved structure in a multiprocessor program.
- Synchronization is naturally associated with sending messages, reducing the possibility for errors introduced by incorrect synchronization.
- It makes it easier to use sender-initiated communication, which may have some advantages in performance,

Of course, the desired communication model can be created on top of a hardware model that supports either of these mechanisms. Supporting message passing on top of shared memory is considerably easier: Because messages essentially send data from one memory to another, sending a message can be implemented by doing a copy from one portion of the address space to another. The major difficulties arise from dealing with messages that may be misaligned and of arbitrary length in a memory system that is normally oriented toward transferring aligned blocks of data organized as cache blocks. These difficulties can be overcome either with small performance penalties in software or with essentially no penalties, using a small amount of hardware support.

Supporting shared memory efficiently on top of hardware for message passing is much more difficult. Without explicit hardware support for shared memory, all shared-memory references need to involve the operating system to provide address translation and memory protection, as well as to translate memory references into message sends and receives. Loads and stores usually move small amounts of data, so the high overhead of handling these communications in software severely limits the range of applications for which the performance of software-based shared memory is acceptable. An ongoing area of research is the exploration of when a software-based model is acceptable and whether a software-based mechanism is usable for the highest level of communication in a hierarchically structured system. One possible direction is the use of virtual memory mechanisms to share objects at the page level, a technique called *shared virtual memory;* we discuss this approach in section 6.10.

In distributed-memory multiprocessors, the memory model and communication mechanisms distinguish the multiprocessors. Originally, distributed-memory multiprocessors were built with message passing, since it was clearly simpler and many designers and researchers did not believe that a shared address space could be built with distributed memory. Shared-memory communication has been sup-

ported in virtually every multiprocessor designed since 1995. What hardware communication mechanisms will be supported in the very largest multiprocessors (called *massively parallel processors*, or *MPPs*), which typically have far more than 100 processors, is unclear; shared memory, message passing, and hybrid approaches are all contenders. Despite the symbolic importance of the MPPs, such multiprocessors are a small portion of the market and have little or no influence on the mainstream multiprocessors with tens of processors. We will return to a discussion of the possibilities and trends for MPPs in the concluding remarks and historical perspectives at the end of this chapter.

SMPs, which we focus on in Section 6.3, vastly dominate DSM multiprocessors in terms of market size (both units and dollars), and SMPs will probably be the architecture of choice for on-chip multiprocessors. For moderate scale multiprocessors (>8 processors) long-term technical trends favor distributing memory, which is also likely to be the dominant approach when on-chip SMPs are used as the building blocks in the future. These distributed shared-memory multiprocessors are a natural extension of the centralized multiprocessors that dominate the market, so we discuss these architectures in section 6.5. In contrast, multicomputers or message-passing multiprocessors build on advances in network technology, as we discussed in the last chapter. Since the technologies employed were well described in the last chapter, we focus our attention on shared-memory approaches in the rest of this chapter.

Challenges of Parallel Processing

Two important hurdles, both explainable with Amdahl's Law, make parallel processing challenging. The first has to do with the limited parallelism available in programs and the second arises from the relatively high cost of communications. Limitations in available parallelism make it difficult to achieve good speedups in any parallel processor, as our first Example shows.

EXAMPLE Suppose you want to achieve a speedup of 80 with 100 processors. What fraction of the original computation can be sequential?

ANSWER Amdahl's Law is

$$\text{Speedup} = \frac{1}{\dfrac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} + (1 - \text{Fraction}_{\text{enhanced}})}$$

For simplicity in this example, assume that the program operates in only two modes: parallel with all processors fully used, which is the enhanced

mode, or serial with only one processor in use. With this simplification, the speedup in enhanced mode is simply the number of processors, while the fraction of enhanced mode is the time spent in parallel mode. Substituting into the equation above:

$$80 = \frac{1}{\frac{\text{Fraction}_{\text{parallel}}}{100} + (1 - \text{Fraction}_{\text{parallel}})}$$

Simplifying this equation yields

$$\begin{aligned} 0.8 \times \text{Fraction}_{\text{parallel}} + 80 \times (1 - \text{Fraction}_{\text{parallel}}) &= 1 \\ 80 - 79.2 \times \text{Fraction}_{\text{parallel}} &= 1 \\ \text{Fraction}_{\text{parallel}} &= \frac{80 - 1}{79.2} \\ \text{Fraction}_{\text{parallel}} &= 0.9975 \end{aligned}$$

Thus to achieve a speedup of 80 with 100 processors, only 0.25% of original computation can be sequential. Of course, to achieve linear speedup (speedup of *n* with *n* processors), the entire program must usually be parallel with no serial portions. (One exception to this is *superlinear speedup* that occurs due to the increased memory and cache available when the processor count is increased. This effect is usually not very large and rarely scales linearly with processor count.) In practice, programs do not just operate in fully parallel or sequential mode, but often use less than the full complement of the processors when running in parallel mode. Exercise 6.2 asks you to extend Amdahl's Law to deal with such a case.

n

The second major challenge in parallel processing involves the large latency of remote access in a parallel processor. In existing shared-memory multiprocessors, communication of data between processors may cost anywhere from 100 clock cycles to over 1,000 clock cycles, depending on the communication mechanism, the type of interconnection network, and the scale of the multiprocessor. Figure 6.3 shows the typical round-trip delays to retrieve a word from a remote memory for several different shared-memory parallel processors.

The effect of long communication delays is clearly substantial. Let's consider a simple Example.

EXAMPLE Suppose we have an application running on a 32-processor multiprocessor, which has a 400 ns time to handle reference to a remote memory. For this application, assume that all the references except those involving communication hit in the local memory hierarchy, which may be only

Multiprocessor	Year shipped	SMP or NUMA	Max. processors	Interconnection network	Typical remote memory access time
Sun Starfire servers	1996	SMP	64	Multiple buses	500 ns
SGI Origin 3000	1999	NUMA	512	Fat hypercube	500 ns
Cray T3E	1996	NUMA	2,048	2-way 3D torus	300 ns
HP V series	1998	SMP	32	8x8 crossbar	1000 ns
Compaq Alphaserver GS	1999	SMP	32	Switched busses	400 ns

FIGURE 6.3 Typical remote access times to retrieve a word from a remote memory in shared-memory multiprocessors.

slightly pessimistic. Processors are stalled on a remote request, and the processor clock rate is 1GHz. If the base IPC (assuming that all references hit in the cache) is 2, how much faster is the multiprocessor if there is no communication versus if 0.2% of the instructions involve a remote communication reference?

ANSWER It is simpler to first calculate the CPI. The effective CPI for the multiprocessor with 0.2% remote references is

$$\begin{aligned} \text{CPI} &= \text{Base CPI} + \text{Remote request rate} \times \text{Remote request cost} \\ &= \frac{1}{\text{Base IPC}} + 0.2\% \times \text{Remote request cost} \\ &= 0.5 + 0.2\% \times \text{Remote request cost} \end{aligned}$$

The Remote request cost is

$$\frac{\text{Remote access cost}}{\text{Cycle time}} = \frac{400\text{ns}}{1\text{ ns}} = 400 \text{ cycles}$$

Hence we can compute the CPI:

$$\text{CPI} = 0.5 + 0.8 = 1.3$$

The multiprocessor with all local references is 1.3/0.5 = 2.6 times faster. In practice, the performance analysis is much more complex, since some fraction of the noncommunication references will miss in the local hierarchy and the remote access time does not have a single constant value. For example, the cost of a remote reference could be quite a bit worse, since contention caused by many references trying to use the global interconnect can lead to increased delays. n

These problems—insufficient parallelism and long latency remote communi-

cation—are the two biggest challenges in using multiprocessors. The problem of inadequate application parallelism must be attacked primarily in software with new algorithms that can have better parallel performance. Reducing the impact of long remote latency can be attacked both by the architecture and by the programmer. For example, we can reduce the frequency of remote accesses with either hardware mechanisms, such as caching shared data, or with software mechanisms, such as restructuring the data to make more accesses local. We can try to tolerate the latency by using prefetching or multithreading, which we examined in Chapters 4 and 5.

Much of this chapter focuses on techniques for reducing the impact of long remote communication latency. For example, sections 6.3 and 6.5 discuss how caching can be used to reduce remote access frequency, while maintaining a coherent view of memory. Section 6.7 discusses synchronization, which, because it inherently involves interprocessor communication, is an additional potential bottleneck. Section 6.8 talks about latency hiding techniques and memory consistency models for shared memory. Before we wade into these topics, it is helpful to have some understanding of the characteristics of parallel applications, both for better comprehension of the results we show using some of these applications and to gain a better understanding of the challenges in writing efficient parallel programs.

6.2 Characteristics of Application Domains

In earlier chapters, we examined the performance and characteristics of applications with only a small amount of insight into the structure of the applications. For understanding the key elements of uniprocessor performance, such as caches and pipelining, general knowledge of an application is often adequate, although we saw that deeper application knowledge was necessary to exploit higher levels of ILP.

In parallel processing, the additional performance-critical characteristics—such as load balance, synchronization, and sensitivity to memory latency—typically depend on high-level characteristics of the application. These characteristics include factors like how data is distributed, the structure of a parallel algorithm, and the spatial and temporal access patterns to data. Therefore at this point we take the time to examine the three different classes of workloads.

The three different domains of multiprocessor workloads we explore are a commercial workload, consisting of transaction processing, decision support, and web searching; a multiprogrammed workload with operating systems behavior included; and a workload consisting of individual parallel programs from the technical computing domain.

A Commercial Workload

Our commercial workload consists of three applications:

1. An online transaction processing workload (OLTP) modeled after TPC-B (which has similar memory behavior to its newer cousin TPC-C) and using Oracle 7.3.2 as the underlying database. The workload consists of a set of client processes that generate requests and a set of servers that handle them. The server processes consume 85% of the user time, with the remaining going to the clients. Although the I/O latency is hidden by careful tuning and enough requests to keep the CPU busy, the server processes typically block for I/O after about 25,000 instructions.
2. A decision support system (DSS) workload based on TPC-D and also using Oracle 7.3.2 as the underlying database. The workload includes only six of the 17 read queries in TPC-D, although the six queries examined in the benchmark span the range of activities in the entire benchmark. To hide the I/O latency, parallelism is exploited both within queries, where parallelism is detected during a query formulation process, and across queries. Blocking calls are much less frequent than in the OLTP benchmark; the six queries average about 1.5 million instructions before blocking.
3. A web index search (Altavista) benchmark based on a search of a memory mapped version of the Altavista database (200 GB). The inner loop is heavily optimized. Because the search structure is static, little synchronization is needed among the threads.

The fraction of time spent in user mode, in the kernel, and in the idle loop are shown in Figure 6.4. The frequency of I/O increases both the kernel time and the idle time (see the OLTP entry, which has the largest I/O to computation ratio). Altavista, which maps the entire search database into memory and has been extensively tuned, shows the least kernel or idle time.

Benchmark	% Time User Mode	% Time Kernel	% Time CPU Idle
OLTP	71%	18%	11%
DSS (range for the six queries)	82–94%	3–5%	4–13%
DSS (average across all queries)	87%	3.7%	9.3%
Altavista	> 98%	< 1%	<1%

FIGURE 6.4 The distribution of execution time in the commercial workloads. The OLTP benchmark has the largest fraction of both OS time and CPU idle time (which is I/O wait time). The DSS benchmark shows much less OS time, since it does less I/O, but still more than 9% idle time. The extensive tuning of the Altavista search engine is clear in these measurement. The data for this workload were collected by Barroso et. al. [1998] on a 4-processor Alphaserver 4100.

Multiprogramming and OS Workload

For small-scale multiprocessors we will also look at a multiprogrammed workload consisting of both user activity and OS activity. The workload used is two independent copies of the compile phase of the Andrew benchmark. The compile phase consists of a parallel make using eight processors. The workload runs for 5.24 seconds on eight processors, creating 203 processes and performing 787 disk requests on three different file systems. The workload is run with 128 MB of memory, and no paging activity takes place.

The workload has three distinct phases: compiling the benchmarks, which involves substantial compute activity; installing the object files in a library; and removing the object files. The last phase is completely dominated by I/O and only two processes are active (one for each of the runs). In the middle phase, I/O also plays a major role and the CPU is largely idle.

Because both CPU idle time and instruction cache performance are important in this workload, we examine these two issues here, focusing on the data cache performance later in the chapter. For the workload measurements, we assume the following memory and I/O systems:

I/O system	Memory
Level 1 instruction cache	32K bytes, two-way set associative with a 64-byte block, one clock cycle hit time
Level 1 data cache	32K bytes, two-way set associative with a 32-byte block, one clock cycle hit time
Level 2 cache	1M bytes unified, two-way set associative with a 128-byte block, hit time 10 clock cycles
Main memory	Single memory on a bus with an access time of 100 clock cycles
Disk system	Fixed access latency of 3 ms (less than normal to reduce idle time)

Figure 6.5 shows how the execution time breaks down for the eight processors using the parameters just listed. Execution time is broken into four components: idle—execution in the kernel mode idle loop; user—execution in user code; synchronization—execution or waiting for synchronization variables; and kernel—execution in the OS that is neither idle nor in synchronization access.

Unlike the parallel scientific workload, this multiprogramming workload has a significant instruction cache performance loss, at least for the OS. The instruction cache miss rate in the OS for a 64-byte block size, two set-associative cache varies from 1.7% for a 32-KB cache to 0.2% for a 256-KB cache. User-level, instruction cache misses are roughly one-sixth of the OS rate, across the variety of cache sizes.

	User execution	Kernel execution	Synchronization wait	CPU Idle (waiting for I/O)
% instructions executed	27%	3%	1%	69%
% execution time	27%	7%	2%	64%

FIGURE 6.5 The distribution of execution time in the multiprogrammed parallel make workload. The high fraction of idle time is due to disk latency when only one of the eight processes is active. These data and the subsequent measurements for this workload were collected with the SimOS system [Rosenblum 1995]. The actual runs and data collection were done by M. Rosenblum, S. Herrod, and E. Bugnion of Stanford University, using the SimOS simulation system.

Scientific/Technical Applications

Our scientific/technical parallel workload consists of two applications and two computational kernels. The kernels are an FFT (fast Fourier transformation) and an LU decomposition, which were chosen because they represent commonly used techniques in a wide variety of applications and have performance characteristics typical of many parallel scientific applications. In addition, the kernels have small code segments whose behavior we can understand and directly track to specific architectural characteristics. Like many scientific application, I/O is essentially nonexistent in this workload.

The two applications that we use in this chapter are Barnes and Ocean, which represent two important but very different types of parallel computation. We briefly describe each of these applications and kernels and characterize their basic behavior in terms of parallelism and communication. We describe how the problem is decomposed for a distributed shared-memory multiprocessor; certain data decompositions that we describe are not necessary on multiprocessors that have a single centralized memory.

The FFT Kernel

The *fast Fourier transform* (FFT) is the key kernel in applications that use spectral methods, which arise in fields ranging from signal processing to fluid flow to climate modeling. The FFT application we study here is a one-dimensional version of a parallel algorithm for a complex-number FFT. It has a sequential execution time for n data points of $n \log n$. The algorithm uses a high radix (equal to $\sqrt{n}$) that minimizes communication. The measurements shown in this chapter are collected for a million-point input data set.

There are three primary data structures: the input and output arrays of the data being transformed and the roots of unity matrix, which is precomputed and only read during the execution. All arrays are organized as square matrices. The six steps in the algorithm are as follows:

1. Transpose data matrix.

2. Perform 1D FFT on each row of data matrix.
3. Multiply the roots of unity matrix by the data matrix and write the result in the data matrix.
4. Transpose data matrix.
5. Perform 1D FFT on each row of data matrix.
6. Transpose data matrix.

The data matrices and the roots of unity matrix are partitioned among processors in contiguous chunks of rows, so that each processor's partition falls in its own local memory. The first row of the roots of unity matrix is accessed heavily by all processors and is often replicated, as we do, during the first step of the algorithm just shown. The data transposes ensure good locality during the individual FFT steps, which would otherwise access nonlocal data.

The only communication is in the transpose phases, which require all-to-all communication of large amounts of data. Contiguous subcolumns in the rows assigned to a processor are grouped into blocks, which are transposed and placed into the proper location of the destination matrix. Every processor transposes one block locally and sends one block to each of the other processors in the system. Although there is no reuse of individual words in the transpose, with long cache blocks it makes sense to block the transpose to take advantage of the spatial locality afforded by long blocks in the source matrix.

The LU Kernel

LU is an LU factorization of a dense matrix and is representative of many dense linear algebra computations, such as QR factorization, Cholesky factorization, and eigenvalue methods. For a matrix of size $n \times n$ the running time is n^3 and the parallelism is proportional to n^2. Dense LU factorization can be performed efficiently by blocking the algorithm, using the techniques in Chapter 5, which leads to highly efficient cache behavior and low communication. After blocking the algorithm, the dominant computation is a dense matrix multiply that occurs in the innermost loop. The block size is chosen to be small enough to keep the cache miss rate low, and large enough to reduce the time spent in the less parallel parts of the computation. Relatively small block sizes (8×8 or 16×16) tend to satisfy both criteria.

Two details are important for reducing interprocessor communication. First, the blocks of the matrix are assigned to processors using a 2D tiling: the $\frac{n}{B} \times \frac{n}{B}$ (where each block is $B \times B$) matrix of blocks is allocated by laying a grid of size $p \times p$ over the matrix of blocks in a cookie-cutter fashion until all the blocks are allocated to a processor. Second, the dense matrix multiplication is performed by the processor that owns the *destination* block. With this blocking and allocation scheme, communication during the reduction is both regular and predictable. For

the measurements in this chapter, the input is a 512×512 matrix and a block of 16×16 is used.

A natural way to code the blocked LU factorization of a 2D matrix in a shared address space is to use a 2D array to represent the matrix. Because blocks are allocated in a tiled decomposition, and a block is not contiguous in the address space in a 2D array, it is very difficult to allocate blocks in the local memories of the processors that own them. The solution is to ensure that blocks assigned to a processor are allocated locally and contiguously by using a 4D array (with the first two dimensions specifying the block number in the 2D grid of blocks, and the next two specifying the element in the block).

The Barnes Application

Barnes is an implementation of the Barnes-Hut n-body algorithm solving a problem in galaxy evolution. *N-body algorithms* simulate the interaction among a large number of bodies that have forces interacting among them. In this instance the bodies represent collections of stars and the force is gravity. To reduce the computational time required to model completely all the individual interactions among the bodies, which grow as n^2, n-body algorithms take advantage of the fact that the forces drop off with distance. (Gravity, for example, drops off as $1/d^2$, where d is the distance between the two bodies.) The Barnes-Hut algorithm takes advantage of this property by treating a collection of bodies that are "far away" from another body as a single point at the center of mass of the collection and with mass equal to the collection. If the body is far enough from any body in the collection, then the error introduced will be negligible. The collections are structured in a hierarchical fashion, which can be represented in a tree. This algorithm yields an $n \log n$ running time with parallelism proportional to n.

The Barnes-Hut algorithm uses an octree (each node has up to eight children) to represent the eight cubes in a portion of space. Each node then represents the collection of bodies in the subtree rooted at that node, which we call a *cell*. Because the density of space varies and the leaves represent individual bodies, the depth of the tree varies. The tree is traversed once per body to compute the net force acting on that body. The force-calculation algorithm for a body starts at the root of the tree. For every node in the tree it visits, the algorithm determines if the center of mass of the cell represented by the subtree rooted at the node is "far enough away" from the body. If so, the entire subtree under that node is approximated by a single point at the center of mass of the cell, and the force this center of mass exerts on the body is computed. On the other hand, if the center of mass is not far enough away, the cell must be "opened" and each of its subtrees visited. The distance between the body and the cell, together with the error tolerances, determines which cells must be opened. This force calculation phase dominates the execution time. This chapter takes measurements using 16K bodies; the criterion for determining whether a cell needs to be opened is set to the middle of the range typically used in practice.

Obtaining effective parallel performance on Barnes-Hut is challenging because the distribution of bodies is nonuniform and changes over time, making partitioning the work among the processors and maintenance of good locality of reference difficult. We are helped by two properties: the system evolves slowly; and because gravitational forces fall off quickly, with high probability, each cell requires touching a small number of other cells, most of which were used on the last time step. The tree can be partitioned by allocating each processor a subtree. Many of the accesses needed to compute the force on a body in the subtree will be to other bodies in the subtree. Since the amount of work associated with a subtree varies (cells in dense portions of space will need to access more cells), the size of the subtree allocated to a processor is based on some measure of the work it has to do (e.g., how many other cells does it need to visit), rather than just on the number of nodes in the subtree. By partitioning the octree representation, we can obtain good load balance and good locality of reference, while keeping the partitioning cost low. Although this partitioning scheme results in good locality of reference, the resulting data references tend to be for small amounts of data and are unstructured. Thus this scheme requires an efficient implementation of shared-memory communication.

The Ocean Application

Ocean simulates the influence of eddy and boundary currents on large-scale flow in the ocean. It uses a restricted red-black Gauss-Seidel multigrid technique to solve a set of elliptical partial differential equations. *Red-black Gauss-Seidel* is an iteration technique that colors the points in the grid so as to consistently update each point based on previous values of the adjacent neighbors. *Multigrid methods* solve finite difference equations by iteration using hierarchical grids. Each grid in the hierarchy has fewer points than the grid below, and is an approximation to the lower grid. A finer grid increases accuracy and thus the rate of convergence, while requiring more execution time, since it has more data points. Whether to move up or down in the hierarchy of grids used for the next iteration is determined by the rate of change of the data values. The estimate of the error at every time-step is used to decide whether to stay at the same grid, move to a coarser grid, or move to a finer grid. When the iteration converges at the finest level, a solution has been reached. Each iteration has n^2 work for an $n \times n$ grid and the same amount of parallelism.

The arrays representing each grid are dynamically allocated and sized to the particular problem. The entire ocean basin is partitioned into square subgrids (as close as possible) that are allocated in the portion of the address space corresponding to the local memory of the individual processors, which are assigned responsibility for the subgrid. For the measurements in this chapter we use an input that has 130×130 grid points. There are five steps in a time iteration. Since data are exchanged between the steps, all the processors present synchronize at the end of each step before proceeding to the next. Communication occurs when the boundary points of a subgrid are accessed by the adjacent subgrid in nearest-neighbor fashion.

Computation/Communication for the Parallel Programs

A key characteristic in determining the performance of parallel programs is the ratio of computation to communication. If the ratio is high, it means the application has lots of computation for each datum communicated. As we saw in section 6.1, communication is the costly part of parallel computing; therefore high computation-to-communication ratios are very beneficial. In a parallel processing environment, we are concerned with how the ratio of computation to communication changes as we increase either the number of processors, the size of the problem, or both. Knowing how the ratio changes as we increase the processor count sheds light on how well the application can be sped up. Because we are often interested in running larger problems, it is vital to understand how changing the data set size affects this ratio.

To understand what happens quantitatively to the computation-to-communication ratio as we add processors, consider what happens separately to computation and to communication as we either add processors or increase problem size. Figure 6.6 shows that as we add processors, for these applications, the amount of computation per processor falls proportionately and the amount of communication per processor falls more slowly. As we increase the problem size, the computation scales as the O( ) complexity of the algorithm dictates. Communication scaling is more complex and depends on details of the algorithm; we describe the basic phenomena for each application in the caption of Figure 6.6.

The overall computation-to-communication ratio is computed from the individual growth rate in computation and communication. In general, this ratio rises slowly with an increase in data set size and decreases as we add processors. This reminds us that performing a fixed-size problem with more processors leads to increasing inefficiencies because the amount of communication among processors grows. It also tells us how quickly we must scale data set size as we add processors, to keep the fraction of time in communication fixed. The following example illustrates this tradeoffs.

EXAMPLE Suppose we know that for a given multiprocessor the Ocean application spends 20% of its execution time waiting for communication when run on four processors. Assume that the cost of each communication event is independent on processor count, which is not true in general, since communication costs rise with processor count. How much faster might we expect Ocean to run on a 32-processor machine with the same problem size? What fraction of the execution time is spent on communication in this case? How much larger a problem should we run if we want the fraction of time spent communicating to be the same?

ANSWER The computation to communication ratio for Ocean is $\sqrt{n}/\sqrt{p}$, so if the problem size is the same, the communication frequency scales by $\sqrt{p}$.

Application	Scaling of computation	Scaling of communication	Scaling of computation-to-communication
FFT	$\frac{n\log n}{p}$	$\frac{n}{p}$	$\log n$
LU	$\frac{n}{p}$	$\frac{\sqrt{n}}{\sqrt{p}}$	$\frac{\sqrt{n}}{\sqrt{p}}$
Barnes	$\frac{n\log n}{p}$	Approximately $\frac{\sqrt{n}(\log n)}{\sqrt{p}}$	Approximately $\frac{\sqrt{n}}{\sqrt{p}}$
Ocean	$\frac{n}{p}$	$\frac{\sqrt{n}}{\sqrt{p}}$	$\frac{\sqrt{n}}{\sqrt{p}}$

FIGURE 6.6 Scaling of computation, of communication, and of the ratio are critical factors in determining performance on parallel multiprocessors. In this table p is the increased processor count and n is the increased data set size. Scaling is on a per-processor basis. The computation scales up with n at the rate given by O( ) analysis and scales down linearly as p is increased. Communication scaling is more complex. In FFT all data points must interact, so communication increases with n and decreases with p. In LU and Ocean, communication is proportional to the boundary of a block, so it scales with data set size at a rate proportional to the side of a square with n points, namely $\sqrt{n}$; for the same reason communication in these two applications scales inversely to $\sqrt{p}$. Barnes has the most complex scaling properties. Because of the fall-off of interaction between bodies, the basic number of interactions among bodies, which require communication, scales as $\sqrt{n}$. An additional factor of log n is needed to maintain the relationships among the bodies. As processor count is increased, communication scales inversely to $\sqrt{p}$.

This means that communication time increase by $\sqrt{8}$. We can use a variation on Amdahl's Law, recognizing that the computation is decreased but the communication time is increased. If T_4 is the total execution time for 4 processors, then the execution time for 32 processors is:

$$\begin{aligned} T_{32} &= \text{Compute time} + \text{Communicaton time} \\ &= \frac{0.8 \times T_4}{8} + (0.2 \times T_4) \times \sqrt{8} \\ &= 0.1 \times T_4 + 0.57 \times T_4 = 0.67 \times T_4 \end{aligned}$$

Hence the speed-up is:

$$\text{Speedup} = \frac{T_4}{T_{32}} = \frac{T_4}{0.67 \times T_4} = 1.49$$

And the fraction of time spent in communication goes from 20% to 0.57/ 0.67 = 85%.

For the fraction of the communication time to remain the same, we must keep the computation to communication ratio the same, so the problem size must scale at the same rate as the processor count. Notice that because we have changed the problem size, we cannot measure of the scaled problem. We will return to the critical issue of scaling applications for multiprocessors in both in the Cross Cutting Issues and the Fallacies and Pitfalls. n

6.3 Symmetric Shared-Memory Architectures

Multis are a new class of computers based on multiple microprocessors. The small size, low cost, and high performance of microprocessors allow design and construction of computer structures that offer significant advantages in manufacture, price-performance ratio, and reliability over traditional computer families.... Multis are likely to be the basis for the next, the fifth, generation of computers. [p. 463]

Bell [1985]

As we saw in Chapter 5, the use of large, multilevel caches can substantially reduce the memory bandwidth demands of a processor. If the main memory bandwidth demands of a single processor are reduced, multiple processors may be able to share the same memory. Starting in the 1980s, this observation, combined with the emerging dominance of the microprocessor, motivated many designers to create small-scale multiprocessors where several processors shared a single physical memory connected by a shared bus. This type of design is called symmetric shared memory, because each processor has the same relationship to one single shared memory. Because of the small size of the processors and the significant reduction in the requirements for bus bandwidth achieved by large caches, such symmetric multiprocessors are extremely cost-effective, provided that a sufficient amount of memory bandwidth exists. Early designs of such multiprocessors were able to place an entire CPU and cache subsystem on a board, which plugged into the bus backplane. More recent designs have placed up to four processors per board; and a recent announcement by IBM includes 2 processors on the same die. Figure 6.1 on page 639 shows a simple diagram of such a multiprocessor.

Small-scale shared-memory machines usually support the caching of both shared and private data. *Private data* is used by a single processor, while *shared data* is used by multiple processors, essentially providing communication among the processors through reads and writes of the shared data. When a private item is cached, its location is migrated to the cache, reducing the average access time as well as the memory bandwidth required. Since no other processor uses the data,

the program behavior is identical to that in a uniprocessor. When shared data are cached, the shared value may be replicated in multiple caches. In addition to the reduction in access latency and required memory bandwidth, this replication also provides a reduction in contention that may exist for shared data items that are being read by multiple processors simultaneously. Caching of shared data, however, introduces a new problem: cache coherence.

What Is Multiprocessor Cache Coherence?

As we saw in Chapter 6, the introduction of caches caused a coherence problem for I/O operations, since the view of memory through the cache could be different from the view of memory obtained through the I/O subsystem. The same problem exists in the case of multiprocessors, because the view of memory held by two different processors is through their individual caches. Figure 6.7 illustrates the problem and shows how two different processors can have two different values for the same location. This difficulty s generally referred to as the *cache-coherence* problem.

Time	Event	Cache contents for CPU A	Cache contents for CPU B	Memory contents for location X
0				1
1	CPU A reads X	1		1
2	CPU B reads X	1	1	1
3	CPU A stores 0 into X	0	1	0

FIGURE 6.7 The cache-coherence problem for a single memory location (X), read and written by two processors (A and B). We initially assume that neither cache contains the variable and that X has the value 1. We also assume a write-through cache; a write-back cache adds some additional but similar complications. After the value of X has been written by A, A's cache and the memory both contain the new value, but B's cache does not, and if B reads the value of X, it will receive 1!

Informally, we could say that a memory system is coherent if any read of a data item returns the most recently written value of that data item. This definition, although intuitively appealing, is vague and simplistic; the reality is much more complex. This simple definition contains two different aspects of memory system behavior, both of which are critical to writing correct shared-memory programs. The first aspect, called *coherence,* defines what values can be returned by a read. The second aspect, called *consistency,* determines when a written value will be returned by a read. Let's look at coherence first.

A memory system is coherent if

1. A read by a processor, P, to a location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P.
2. A read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated in time and no other writes to X occur between the two accesses.
3. Writes to the same location are *serialized*: that is, two writes to the same location by any two processors are seen in the same order by all processors. For example, if the values 1 and then 2 are written to a location, processors can never read the value of the location as 2 and then later read it as 1.

The first property simply preserves program order—we expect this property to be true even in uniprocessors. The second property defines the notion of what it means to have a coherent view of memory: If a processor could continuously read an old data value, we would clearly say that memory was incoherent.

The need for write serialization is more subtle, but equally important. Suppose we did not serialize writes, and processor P1 writes location X followed by P2 writing location X. Serializing the writes ensures that every processor will see the write done by P2 at some point. If we did not serialize the writes, it might be the case that some processor could see the write of P2 first and then see the write of P1, maintaining the value written by P1 indefinitely. The simplest way to avoid such difficulties is to serialize writes, so that all writes to the same location are seen in the same order; this property is called *write serialization*.

Although the three properties just described are sufficient to ensure coherence, the question of when a written value will be seen is also important. To see why, observe that we cannot require that a read of X instantaneously see the value written for X by some other processor. If, for example, a write of X on one processor precedes a read of X on another processor by a very small time, it may be impossible to ensure that the read returns the value of the data written, since the written data may not even have left the processor at that point. The issue of exactly *when* a written value must be seen by a reader is defined by a *memory consistency model*—a topic discussed in section 6.8.

Coherence and consistency are complementary: Coherence defines the behavior of reads and writes to the same memory location, while consistency defines the behavior of reads and writes with respect to accesses to other memory locations. For simplicity, and because we cannot explain the problem in full detail at this point, assume that we require that a write does not complete until all processors have seen the effect of the write and that the processor does not change the order of any write with any other memory access. This allows the processor to reorder reads, but forces the processor to finish a write in program order. We will rely on this assumption until we reach section 6.8, where we will see exactly the meaning of this definition, as well as the alternatives.

Basic Schemes for Enforcing Coherence

The coherence problem for multiprocessors and I/O, although similar in origin, has different characteristics that affect the appropriate solution. Unlike I/O, where multiple data copies are a rare event—one to be avoided whenever possible—a program running on multiple processors will normally have copies of the same data in several caches. In a coherent multiprocessor, the caches provide both *migration* and *replication* of shared data items.

Coherent caches provide migration, since a data item can be moved to a local cache and used there in a transparent fashion. This migration reduces both the latency to access a shared data item that is allocated remotely and the bandwidth demand on the shared memory.

Coherent caches also provide replication for shared data that is being simultaneously read, since the caches make a copy of the data item in the local cache. Replication reduces both latency of access and contention for a read shared data item. Supporting this migration and replication is critical to performance in accessing shared data. Thus, rather than trying to solve the problem by avoiding it in software, small-scale multiprocessors adopt a hardware solution by introducing a protocol to maintain coherent caches.

The protocols to maintain coherence for multiple processors are called *cache-coherence protocols*. Key to implementing a cache-coherence protocol is tracking the state of any sharing of a data block. There are two classes of protocols, which use different techniques to track the sharing status, in use:

- *Directory based*—The sharing status of a block of physical memory is kept in just one location, called the *directory;* we focus on this approach in section 6.5, when we discuss scalable shared-memory architecture.
- *Snooping*—Every cache that has a copy of the data from a block of physical memory also has a copy of the sharing status of the block, and no centralized state is kept. The caches are usually on a shared-memory bus, and all cache controllers monitor or *snoop* on the bus to determine whether or not they have a copy of a block that is requested on the bus. We focus on this approach in this section.

Snooping protocols became popular with multiprocessors using microprocessors and caches attached to a single shared memory because these protocols can use a preexisting physical connection—the bus to memory—to interrogate the status of the caches.

Snooping Protocols

There are two ways to maintain the coherence requirement described in the previous subsection. One method is to ensure that a processor has exclusive access to a data item before it writes that item. This style of protocol is called a *write invalidate protocol* because it invalidates other copies on a write. It is by far the most

common protocol, both for snooping and for directory schemes. Exclusive access ensures that no other readable or writable copies of an item exist when the write occurs: all other cached copies of the item are invalidated.

Figure 6.8 shows an example of an invalidation protocol for a snooping bus with write-back caches in action To see how this protocol ensures coherence, consider a write followed by a read by another processor: Since the write requires exclusive access, any copy held by the reading processor must be invalidated (hence the protocol name). Thus, when the read occurs, it misses in the cache and is forced to fetch a new copy of the data. For a write, we require that the writing processor have exclusive access, preventing any other processor from being able to write simultaneously. If two processors do attempt to write the same data simultaneously, one of them wins the race (we'll see how we decide who wins shortly), causing the other processor's copy to be invalidated. For the other processor to complete its write, it must obtain a new copy of the data, which must now contain the updated value. Therefore, this protocol enforces write serialization.

Processor activity	Bus activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of memory location X
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes a 1 to X	Invalidation for X	1		0
CPU B reads X	Cache miss for X	1	1	1

FIGURE 6.8 An example of an invalidation protocol working on a snooping bus for a single cache block (X) with write-back caches. We assume that neither cache initially holds X and that the value of X in memory is 0. The CPU and memory contents show the value after the processor and bus activity have both completed. A blank indicates no activity or no copy cached. When the second miss by B occurs, CPU A responds with the value canceling the response from memory. In addition, both the contents of B's cache and the memory contents of X are updated. This update of memory, which occurs when a block becomes shared, is typical in most protocols and simplifies the protocol, as we will see shortly.

The alternative to an invalidate protocol is to update all the cached copies of a data item when that item is written. This type of protocol is called a *write update* or *write broadcast* protocol. To keep the bandwidth requirements of this protocol under control it is useful to track whether or not a word in the cache is shared—that is, is contained in other caches. If it is not, then there is no need to broadcast or update any other caches. Figure 6.8 shows an example of a write update protocol in operation. In the decade since these protocols were developed, invalidate has emerged as the winner for the vast majority of designs. To understand why, let's look at the qualitative performance differences.

The performance differences between write update and write invalidate protocols arise from three characteristics:

Processor activity	Bus activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of memory location X
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes a 1 to X	Write broadcast of X	1	1	1
CPU B reads X		1	1	1

FIGURE 6.9 An example of a write update or broadcast protocol working on a snooping bus for a single cache block (X) with write-back caches. We assume that neither cache initially holds X and that the value of X in memory is 0. The CPU and memory contents show the value after the processor and bus activity have both completed. A blank indicates no activity or no copy cached. When CPU A broadcasts the write, both the cache in CPU B and the memory location of X are updated.

1. Multiple writes to the same word with no intervening reads require multiple write broadcasts in an update protocol, but only one initial invalidation in a write invalidate protocol.
2. With multiword cache blocks, each word written in a cache block requires a write broadcast in an update protocol, although only the first write to any word in the block needs to generate an invalidate in an invalidation protocol. An invalidation protocol works on cache blocks, while an update protocol must work on individual words (or bytes, when bytes are written). It is possible to try to merge writes in a write broadcast scheme, just as we did for write buffers in Chapter 5, but the basic difference remains.
3. The delay between writing a word in one processor and reading the written value in another processor is usually less in a write update scheme, since the written data are immediately updated in the reader's cache (assuming that the reading processor has a copy of the data). By comparison, in an invalidation protocol, the reader is invalidated first, then later reads the data and is stalled until a copy can be read and returned to the processor.

Because bus and memory bandwidth is usually the commodity most in demand in a bus-based multiprocessor and invalidation protocols generate less bus and memory traffic, invalidation has become the protocol of choice for almost all multiprocessors. Update protocols also cause problems for memory consistency models, reducing the potential performance gains of update, mentioned in point 3, even further. In designs with very small processor counts (2, or at most, 4) where the processors are tightly coupled (perhaps even on the same chip), the larger bandwidth demands of update may be acceptable. Nonetheless, given the trends in increasing processor performance and the related increase in bandwidth

demands, we can expect update schemes to be used very infrequently. For this reason, we will focus only on invalidate protocols for the rest of the chapter.

Basic Implementation Techniques

The key to implementing an invalidate protocol in a small-scale multiprocessor is the use of the bus to perform invalidates. To perform an invalidate the processor simply acquires bus access and broadcasts the address to be invalidated on the bus. All processors continuously snoop on the bus watching the addresses. The processors check whether the address on the bus is in their cache. If so, the corresponding data in the cache is invalidated.

The serialization of access enforced by the bus also forces serialization of writes, since when two processors compete to write to the same location, one must obtain bus access before the other. The first processor to obtain bus access will cause the other processor's copy to be invalidated, causing writes to be strictly serialized. One implication of this scheme is that a write to a shared data item cannot complete until it obtains bus access.

In addition to invalidating outstanding copies of a cache block that is being written into, we also need to locate a data item when a cache miss occurs. In a write-through cache, it is easy to find the recent value of a data item, since all written data are always sent to the memory, from which the most recent value of a data item can always be fetched. (Write buffers can lead to some additional complexities, which are discussed in section 6.8.)

For a write-back cache, however, the problem of finding the most recent data value is harder, since the most recent value of a data item can be in a cache rather than in memory. Happily, write-back caches can use the same snooping scheme both for caches misses and for writes: Each processor snoops every address placed on the bus. If a processor finds that it has a dirty copy of the requested cache block, it provides that cache block in response to the read request and causes the memory access to be aborted. Since write-back caches generate lower requirements for memory bandwidth, they are greatly preferable in a multiprocessor, despite the slight increase in complexity. Therefore, we focus on implementation with write-back caches.

The normal cache tags can be used to implement the process of snooping, and the valid bit for each block makes invalidation easy to implement. Read misses, whether generated by an invalidation or by some other event, are also straightforward since they simply rely on the snooping capability. For writes we'd like to know whether any other copies of the block are cached, because, if there are no other cached copies, then the write need not be placed on the bus in a write-back cache. Not sending the write reduces both the time taken by the write and the required bandwidth.

To track whether or not a cache block is shared we can add an extra state bit associated with each cache block, just as we have a valid bit and a dirty bit. By adding a bit indicating whether the block is shared, we can decide whether a write must generate an invalidate. When a write to a block in the shared state oc-

curs, the cache generates an invalidation on the bus and marks the block as private. No further invalidations will be sent by that processor for that block. The processor with the sole copy of a cache block is normally called the *owner* of the cache block.

When an invalidation is sent, the state of the owner's cache block is changed from shared to unshared (or exclusive). If another processor later requests this cache block, the state must be made shared again. Since our snooping cache also sees any misses, it knows when the exclusive cache block has been requested by another processor and the state should be made shared.

Every bus transaction must check the cache-address tags, which could potentially interfere with CPU cache accesses. This potential interference is reduced by one of two techniques: duplicating the tags or employing a multilevel cache with *inclusion*, whereby the levels closer to the CPU are a subset of those further away. If the tags are duplicated, then the CPU and the snooping activity may proceed in parallel. course, on a cache miss the processor needs to arbitrate for and update both sets of tags. Likewise, if the snoop finds a matching tag entry, it needs to arbitrate for and access both sets of cache tags (to perform an invalidate or to update the shared bit), as well as possibly the cache data array to retrieve a copy of a block. Thus with duplicate tags the processor only needs to be stalled when it does a cache access at the same time that a snoop has detected a copy in the cache. Furthermore, snooping activity is delayed only when the cache is dealing with a miss.

If the CPU uses a multilevel cache with the inclusion property, then every entry in the primary cache is required to be in the secondary cache. Thus the snoop activity can be directed to the second-level cache, while most of the processor's activity is directed to the primary cache. If the snoop gets a hit in the secondary cache, then it must arbitrate for the primary cache to update the state and possibly retrieve the data, which usually requires a stall of the processor. Since many multiprocessors use a multilevel cache to decrease the bandwidth demands of the individual processors, this solution has been adopted in many designs. Sometimes it may even be useful to duplicate the tags of the secondary cache to further decrease contention between the CPU and the snooping activity. We discuss the inclusion property in more detail in section 6.10 on page 728.

An Example Protocol

A bus-based coherence protocol is usually implemented by incorporating a finite state controller in each node. This controller responds to requests from the processor and from the bus, changing the state of the selected cache block, as well as using the bus to access data or to invalidate it. Figure 6.10 shows the requests generated by the processor-cache module in a node, in the top half of the table, as well as those coming from the bus, in the bottom half of the table. For simplicity,

the protocol we explain does not distinguish between a write hit and a write miss to a shared cache block: in both cases, we treat such an access as a write miss. When the write miss is placed on the bus, any processors with copies of the cache block invalidate it. In a write-back cache, if the block is exclusive in just one cache, that cache also writes back the block. Treating write hits to shared blocks as cache misses reduces the number of different bus transactions and simplifies the controller. In more sophisticated protocols, these "misses" are treated as upgrade requests that generate a bus transaction and an invalidate, but do not actually transfer the data, since the copy in he cache is up-to-date.

Request	Source	State of addressed cache block	Function and explanation
Read hit	Processor	Shared or Exclusive	Read data in cache
Read miss	Processor	Invalid	Place read miss on bus.
Read miss	Processor	Shared	Address conflict miss: place read miss on bus
Read miss	Processor	Exclusive	Address conflict miss: write back block, then place read miss on bus
Write hit	Processor	Exclusive	Write data in cache.
Write hit	Processor	Shared	Place write miss on bus.
Write miss	Processor	Invalid	Place write miss on bus.
Write miss	Processor	Shared	Address conflict miss: place write miss on bus
Write miss	Processor	Exclusive	Address conflict miss: write back block, then place write miss on bus
Read Miss	Bus	Shared	No action; allow memory to service read miss.
Read Miss	Bus	Exclusive	Attempt to share data: place cache block on bus and change state to Shared.
Write miss	Bus	Shared	Attempt to write shared block; invalidate the block.
Write miss	Bus	Exclusive	Attempt to write block that is exclusive elsewhere: write back the cache block and make its state Invalid.

FIGURE 6.10 The cache-coherence mechanism receives requests from both the processor and the bus and responds to these based on the type of request, whether it hits or misses in the cache, and the state of the cache block specified in the request. For read or write misses snooped from the bus, an action is required *only* if the read or write addresses matches a block in the cache and the block is valid. Placing a write miss on the bus when a write hits in the Shared state, ensures an exclusive copy, though the data need not actually be transferred. This is referred to as an upgrade, and some protocols distinguish it from a write miss to avoid the data transfer.

Figure 6.11 shows a finite-state transition diagram for a single cache block using a write-invalidation protocol and a write-back cache. For simplicity, the three states of the protocol are duplicated to represent transitions based on CPU requests (on the left, which corresponds to the top half of the table in Figure 6.11), as opposed to transitions based on bus requests (on the right, which corresponds

to the bottom half of the table in Figure 6.11). Boldface type is used to distinguish the bus actions, as opposed to the conditions on which a state transition depends. The state in each node represents the state of the selected cache block specified by the processor or bus request.

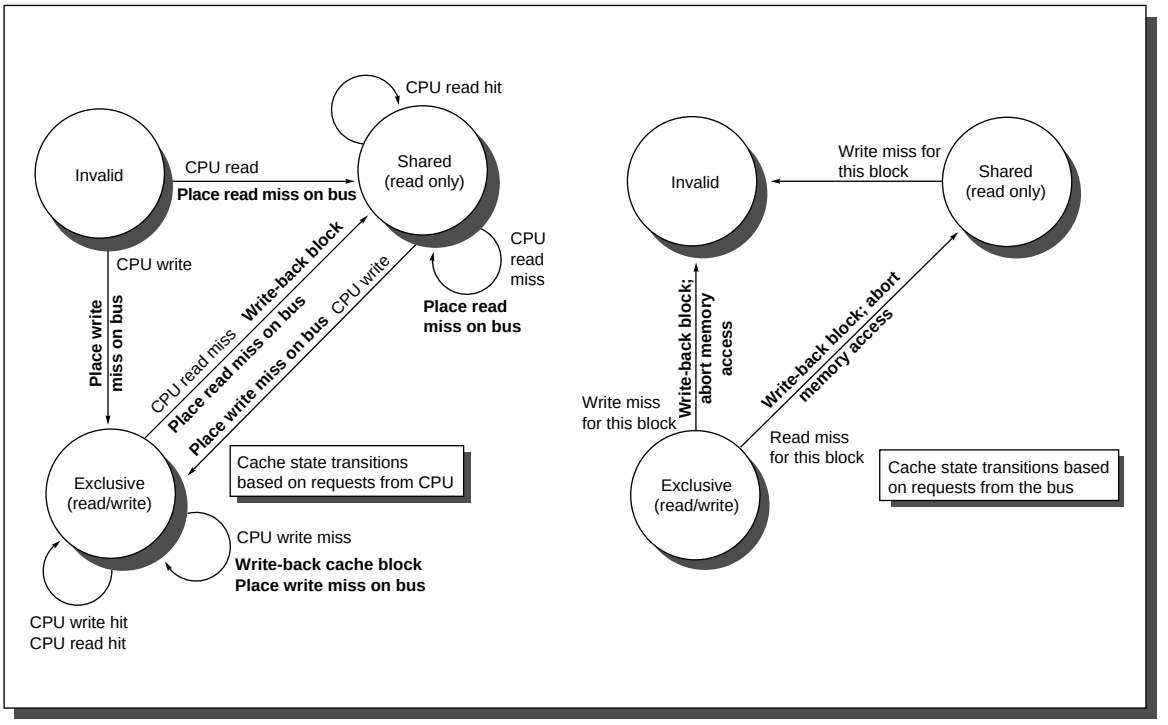


FIGURE 6.11 A write-invalidate, cache-coherence protocol for a write-back cache showing the states and state transitions for each block in the cache. The cache states are shown in circles with any access permitted by the CPU without a state transition shown in parenthesis under the name of the state. The stimulus causing a state change is shown on the transition arcs in regular type, and any bus actions generated as part of the state transition are shown on the transition arc in bold. The stimulus actions apply to a block in the cache, not to a specific address in the cache. Hence, a read miss to a block in the shared state is a miss for that cache block but for a different address. The left side of the diagram shows state transitions based on actions of the CPU associated with this cache; the right side shows transitions based on operations on the bus. A read miss in the exclusive or shared state and a write miss in the exclusive state occur when the address requested by the CPU does not match the address in the cache block. Such a miss is a standard cache replacement miss. An attempt to write a block in the shared state always generates a miss, even if the block is present in the cache, since the block must be made exclusive. Whenever a bus transaction occurs, all caches that contain the cache block specified in the bus transaction take the action dictated by the right half of the diagram. The protocol assumes that memory provides data on a read miss for a block that is clean in all caches. In actual implementations, these two sets of state diagrams are combined. This protocol is somewhat simpler than those in use in existing multiprocessors.

All of the states in this cache protocol would be needed in a uniprocessor cache, where they would correspond to the invalid, valid (and clean), and dirty

states. All of the state changes indicated by arcs in the left half of Figure 6.11 would be needed in a write-back uniprocessor cache; the only difference in a multiprocessor with coherence is that the controller must generate a write miss when the controller has a write hit for a cache block in the shared state. The state changes represented by the arcs in the right half of Figure 6.11 are needed only for coherence and would not appear at all in a uniprocessor cache controller.

In reality, there is only one finite-state machine per cache, with stimuli coming either from the attached CPU or from the bus. Figure 6.12 shows how the state

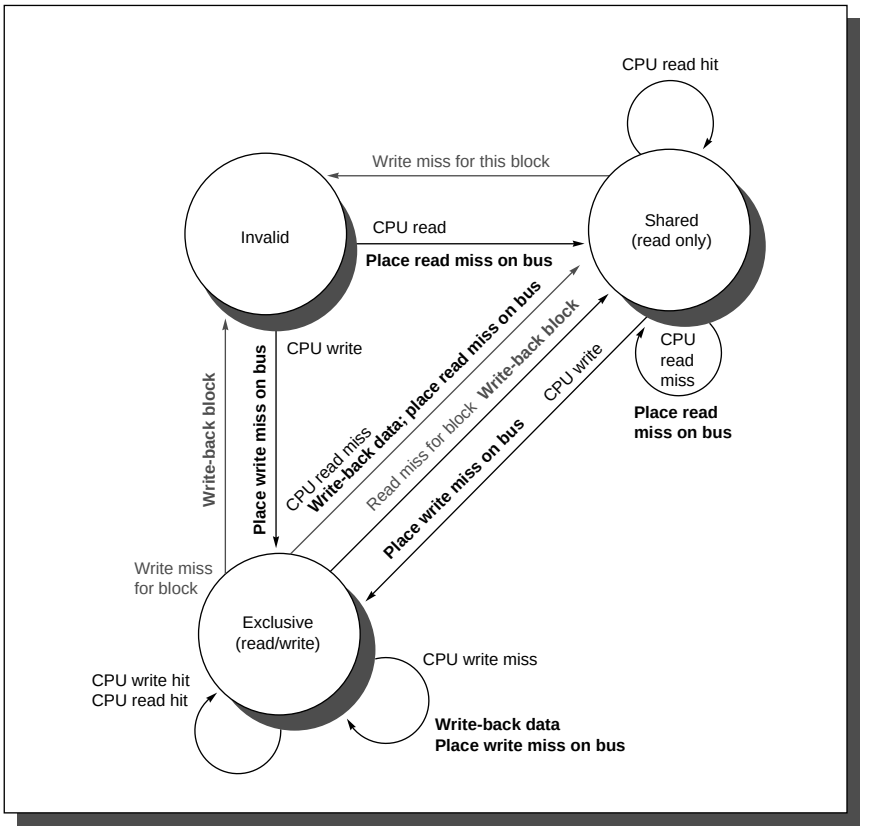


FIGURE 6.12 Cache-coherence state diagram with the state transitions induced by the local processor shown in black and by the bus activities shown in gray. As in Figure 6.11, the activities on a transition are shown in bold.

transitions in the right half of Figure 6.11 are combined with those in the left half of the figure to form a single state diagram for each cache block.

To understand why this protocol works, observe that any valid cache block is either in the shared state in multiple caches or in the exclusive state in exactly one cache. Any transition to the exclusive state (which is required for a processor to write to the block) requires a write miss to be placed on the bus, causing all caches to make the block invalid. In addition, if some other cache had the block in exclusive state, that cache generates a write back, which supplies the block containing the desired address. Finally, if a read miss occurs on the bus to a block in the exclusive state, the owning cache also makes its state shared, forcing a subsequent write to require exclusive ownership.

The actions in gray in Figure 6.12, which handle read and write misses on the bus, are essentially the snooping component of the protocol. One other property that is preserved in this protocol, and in most other protocols, is that any memory block in the shared state is always up to date in the memory. This simplifies the implementation, as we will see in detail in section 6.7.

Although our simple cache protocol is correct, it omits a number of complications that make the implementation much trickier. The most important of these is that the protocol assumes that operations are *atomic*—that is, an operation can be done in such a way that no intervening operation can occur. For example, the protocol described assumes that write misses can be detected, acquire the bus, and receive a response as a single atomic action. In reality this is not true. Similarly, if we used a split transaction bus (see Chapter 6, section 6.3), as most modern bus-based multiprocessors do, then even read misses would also not be atomic.

Nonatomic actions introduce the possibility that the protocol can *deadlock*, meaning that it reaches a state where it cannot continue. Appendix E deals with these complex issues, showing how the protocol can be modified to deal with nonatomic writes without introducing deadlock.

As stated earlier, this coherence protocol is actually simpler than those used in practice. There are two major simplifications. First, in this protocol all transitions to the exclusive state generate a write miss on the bus, and we assume that the requesting cache always fills the block with the contents returned. This simplifies the detailed implementation. Most real protocols distinguish between a write miss and a write hit, which can occur when the cache block is initially in the shared state. Such misses are called *ownership* or *upgrade* misses, since they involve changing the state of the block, but do not actually require a data fetch. To support such state changes, the protocol uses an *invalidate operation*, in addition to a write miss. With such operations, however, the actual implementation of the protocol becomes slightly more complex.

The second major simplification is that many multiprocessors distinguish between a cache block that is really shared and one that exists in the clean state in exactly one cache. This addition of a “clean and private” state eliminates the need to generate a bus transaction on a write to such a block. Another enhancement in wide use allows other caches to supply data on a miss to a shared block.

Constructing small (2-4) processor bus-based multiprocessors has become very easy. Many modern microprocessors provide basic support for cache coherency and also allow the construction of a shared memory bus by direct connection of the external memory bus of two processors. These capabilities reduce the number of chips required to build a small-scale multiprocessor. For example, the Intel Pentium III Xeon and Pentium 4 Xeon processors are designed for use in cache coherent multiprocessors and have an external interface that supports snooping and allows two processors to be directly connected. They also have larger on-chip caches to reduce bus utilization. A system chip set containing an external memory controller is used to connect the shared processor memory bus with a set of memory chips. The memory controller also implements the coherency protocol. Since different size multiprocessors generate different demands for bus bandwidth, Intel has two different system chip sets designed for dual processor systems and for midrange range systems (2-4 processors). A small-scale multiprocessor may be built with only two additional system chips: the memory controller memory controller mentioned above and an I/O hub chip that interfaces standard I/O buses, such as PCI, to the memory bus.

The next part of this section examines the performance of these protocols for our parallel and multiprogrammed workloads; the value of these extensions to a basic protocol will be clear when we examine the performance.

6.4 Performance of Symmetric Shared-Memory Multiprocessors

In a bus-based multiprocessor using an invalidation protocol, several different phenomena combine to determine performance. In particular, the overall cache performance is a combination of the behavior of uniprocessor cache miss traffic and the traffic caused by communication, which results in invalidations and subsequent cache misses. Changing the processor count, cache size, and block size can affect these two components of the miss rate in different ways, leading to overall system behavior that is a combination of the two effects.

In Chapter 5, we saw how breaking the uniprocessor miss rate into the 3C classification could provide insight into both application behavior and potential improvements to the cache design. Similarly, the misses that arise from interprocessor communication, which are often called *coherence misses*, can be broken into two separate sources.

The first source are the so-called *true sharing misses* that arise from the communication of data through the cache coherence mechanism. In an invalidation-based protocol, the first write by a processor to a shared cache block causes an invalidation to establish ownership of that block. Additionally, when another processor attempts to read a modified word in that cache block, a miss occurs and the

resultant block is transferred. Both these misses are classified as true sharing misses since they directly arise from the sharing of data among processors.

The second effect, called *false sharing*, arises from the use of an invalidation-based coherence algorithm with a single valid bit per cache block. False sharing occurs when a block is invalidated (and a subsequent reference causes a miss) because some word in the block, other than the one being read, is written into. If the word written into is actually used by the processor that received the invalidate, then the reference was a true sharing reference and would have caused a miss independent of the block size or position of words. If, however, the word being written and the word read are different and the invalidation does not cause a new value to be communicated, but only causes an extra cache miss, then it is a false sharing miss. In a false sharing miss, the block is shared, but no word in the cache is actually shared, and the miss would not occur if the block size were a single word. The following Example makes the sharing patterns clear.

EXAMPLE Assume that words x1 and x2 are in the same cache block, which is in the shared state in the caches of P1 and P2. Assuming the following sequence of events, identify each miss as a true sharing miss, a false sharing miss, or a hit. Any miss that would occur if the block size were one word is designated a true sharing miss.

Time	P1	P2
1	Write x1	
2		Read x2
3	Write x1	
4		Write x2
5	Read x2	

ANSWER Here are classifications by time step:

1. This event is a true sharing miss, since x1 was read by P2 and needs to be invalidated from P2.
2. This event is a false sharing miss, since x2 was invalidated by the write of x1 in P1, but that value of x1 is not used in P2.
3. This event is a false sharing miss, since the block containing x1 is marked shared due to the read in P2, but P2 did not read x1. The cache block containing x1 will be in the shared state after the read by P2; a write miss is required to obtain exclusive access to the block. In some protocols this will be handled as an *upgrade request*, which

generates a bus invalidate, but does not transfer the cache block.

4. This event is a false sharing miss for the same reason as step 3.
5. This event is a true sharing miss, since the value being read was written by P2. n

True sharing and false sharing miss rates can be affected by a variety of changes in the cache architecture. Thus, we will find it useful to decompose not only the uniprocessor and multiprocessor miss rates, but also the true-sharing and false-sharing miss rates.

Performance Measurements of the Commercial Workload

The performance measurements of the commercial workload, which we examine in this section, were taken either on a Alphaserver 4100, or using a configurable simulator modeled after the Alphaserver 4100. The Alphaserver 4100 used for these measurements has four processors, each of which is an Alpha 21164 running at 300 MHz. Each processor has a three-level cache hierarchy:

- L1 consist of a pair of 8 KB direct-mapped on-chip caches, one for instruction and one for data. The block size is 32-bytes, and the data cache is write-through to L2, using a write buffer.
- L2 is a 96 KB on-chip unified 3-way set associative cache with a 32-byte block size, using write-back.
- L3 is an off-chip, combined, direct-mapped 2 MB caches with 64-byte blocks also using write-back.

The latency for an access to L2 is 7 cycles, to L3 it is 21 cycles, and to main memory it is 80 clock cycles (typical without contention). Cache to cache transfers, which occur on a miss to an exclusive block held in another cache, require 125 clock cycles. All the measurement shown in this section were collected by Barroso, Gharachorloo, and Bugnion [1998].

We start by looking at the overall CPU execution for these benchmarks on the 4-processor system; as discussed on page 650, these benchmarks include substantial I/O time, which is ignored in the CPU time measurements. We group the six DSS queries as a single benchmark, reporting the average behavior. The effective CPI varies widely for these benchmarks, from a CPI of 1.3 for the Altavista web search to an average CPI of 1.6 for the DSS workload, to 7.0 for the OLTP workload. Figure 6.13 shows how the execution time breaks down into instruction execution, cache and memory system access time, and other stalls (which are primarily pipeline resource stalls, but also include TLB and branch mispredict stalls). Although the performance of the DSS and Altavista workloads is reasonable, the performance of the OLTP workload is very poor, due to a poor performance of the memory hierarchy.

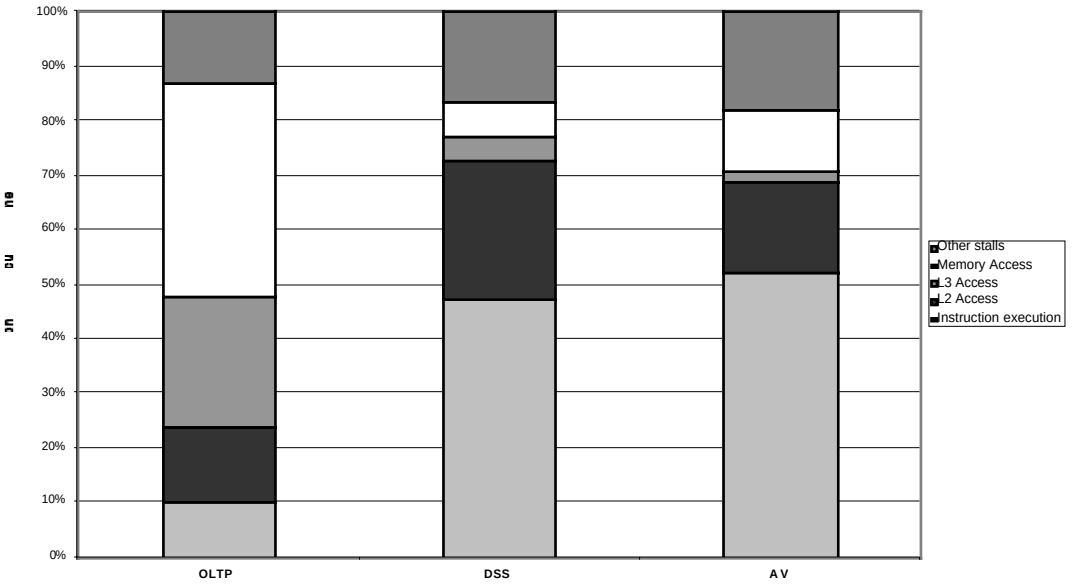


FIGURE 6.13 The execution time breakdown for the three programs (OLTP, DSS, and Altavista) in the commercial workload. The DSS numbers are the average across six different queries. The CPI varies widely from a low of 1.3 for Altavista, to 1.61 for the DSS queries, to 7.0 for Oracle. (Individually, the DSS queries show a CPI range of 1.3 to 1.9.) Other stalls includes: resource stalls (implemented with replay traps on the 21164), branch mispredict, memory barrier, and TLB misses. For these benchmarks resource-based pipeline stalls are the dominant factor.This data combines the behavior of user and kernel accesses. Only OLTP has a significant fraction of kernel accesses, and the kernel accesses tend to be better behaved than the user accesses!

Since the OLTP workload demands the most from the memory system with large numbers of expensive L3 misses, we focus on examining the impact of L3 cache size, processor count, and block size on the OLTP benchmark. Figure 6.14 shows the effect of increasing the cache size, using 2-way set associative caches, which reduces the large number of conflict misses. The execution time is improved as the L3 cache grows due to the reduction in L3 misses. The idle time also grows, reducing some of the performance gains. This growth occurs because with fewer memory system stalls, more server processes are needed to cover the I/O latency. The workload could be retuned to increase the computation/communication balance, holding the idle time in check.

To better understand how the L3 miss rate responds, we ask: What factors contribute to the L3 miss rate and how do they change as the L3 cache grows? Figure 6.15 shows this data, displaying the number of memory access cycles contributed per instruction from five sources. The two largest sources of memory access cycles (due to L3 misses) with a 1 MB L3 are instruction and capacity/conflict misses. With a larger L3 these two sources shrink to be minor contributors. Un-

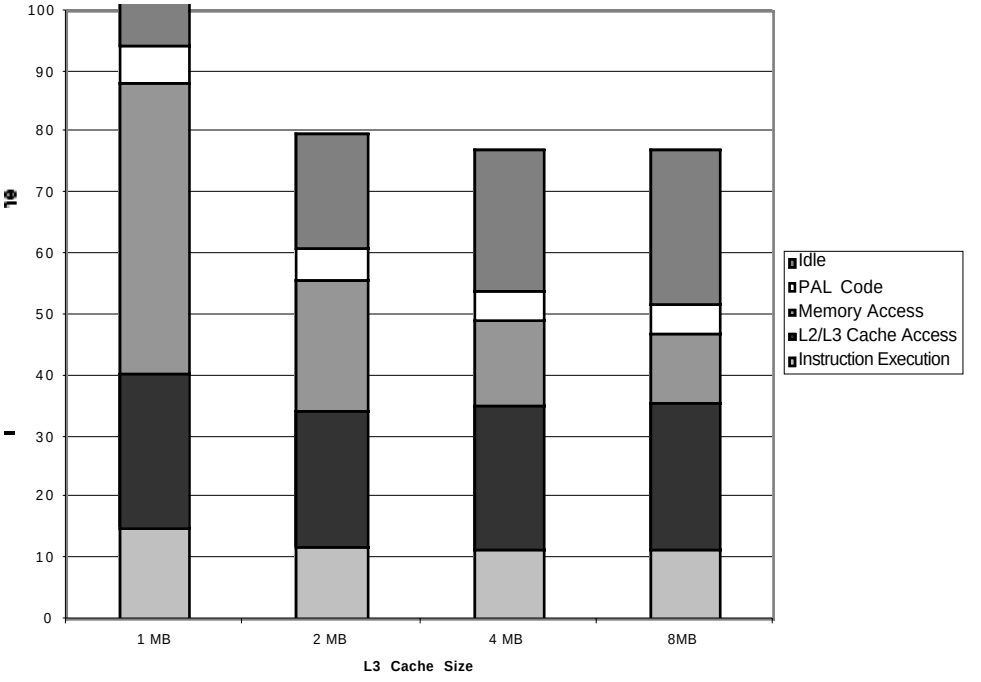


FIGURE 6.14 The relative performance of the OLTP workload as the size of the L3 cache, which is set as 2-way set associative, is grown from 1 MB to 8MB. Interestingly, the performance of the 1 MB, 2-way set associative cache is very similar to the direct-mapped 2 MB cache that is used in the Alphaserver 4100.

fortunately, the cold, false sharing, and true sharing misses are unaffected by a larger L3. Thus, at 4 and 8 MB, the true sharing misses generate the dominant fraction of the misses.

Clearly, increasing the cache size eliminates most of the uniprocessor misses, while leaving the multiprocessor misses untouched. How does increasing the processor count affect different types of misses? Figure 6.16 shows this data assuming a base configuration with a 2 MB, 2-way set associative L3 cache. As we might expect, the increase in the true sharing miss rate, which is not compensated for by any decrease in the uniprocessor misses, leads to an overall increase in the memory access cycles per instruction.

The final question we examine is whether increasing the block size, which should decrease the instruction and cold miss rate and, within limits, also reduce the capacity/ conflict miss rate, is helpful for this workload. Figure 6.17 shows

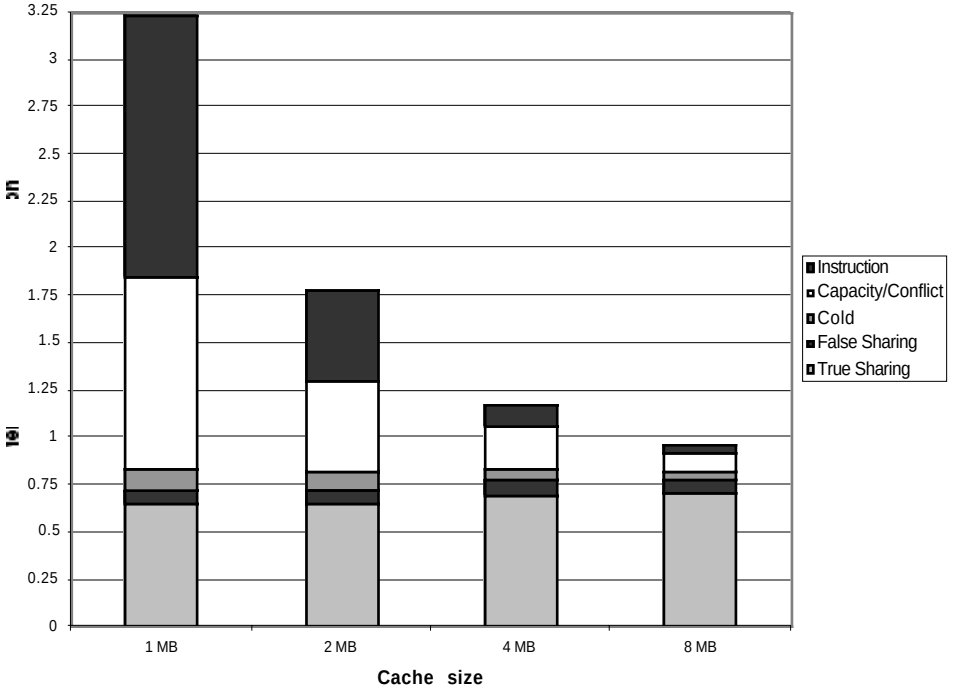


FIGURE 6.15 The contributing causes of memory access cycles shift as the cache size is increased. The L3 cache is simulated as 2-way set associative.

the number of misses per one-thousand instructions as the block size is increased from 32 to 256. Increasing the block size from 32 to 256 affects four of the miss rate components:

- the true sharing miss rate decreases by more than a factor of 2, indicating locality in the true sharing patterns,
- the cold start miss rate significantly decreases, as we would expect,
- the conflict/capacity misses show a small decrease (a factor of 1.26 compared to a factor of 8 increase in block size), indicating that the spatial locality is not high in the uniprocessor misses, and
- the false sharing miss rate. although small in absolute terms, nearly doubles.

The lack of a significant effect on the instruction miss rate is startling and clearly indicates that the large instruction footprint has very poor spatial locality! Over-

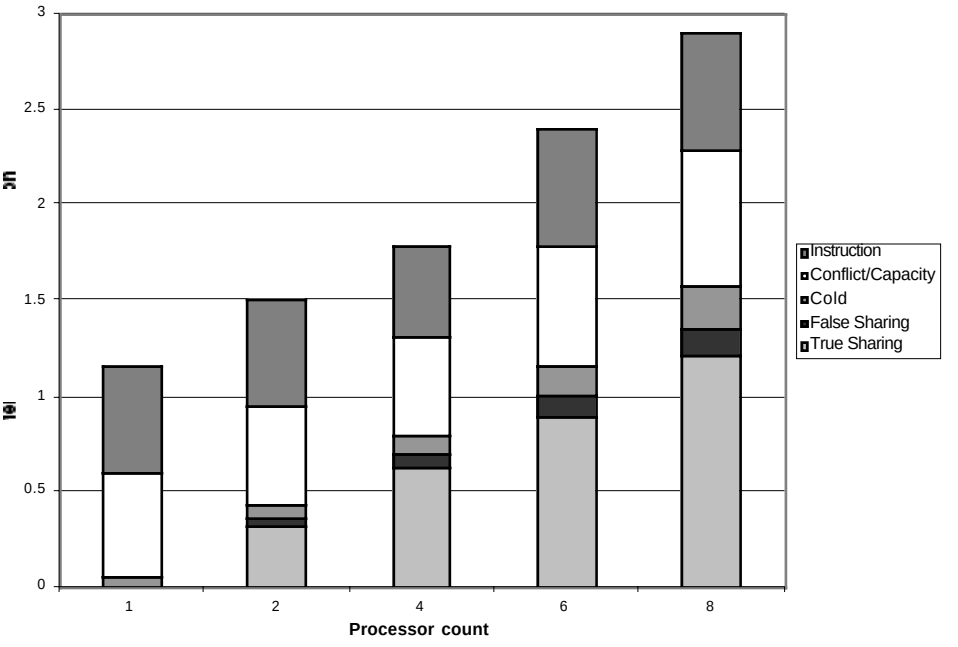


FIGURE 6.16 The contribution to memory access cycles increases as processor count increases primarily due to increased true sharing. The cold misses slightly increase since each processor must now handle more cold misses.

all, increasing the block size of the of the third-level cache to 128 or possibly 256 bytes seems appropriate.

Performance of the Multiprogramming and OS Workload

In this subsection we examine the cache performance of the multiprogrammed workload as the cache size and block size are changed. The workload remains the same as described in the previous section: two independent parallel makes, each using up to eight processors. Because of differences between the behavior of the kernel and that of the user processes, we keep these two components separate. Remember, though, that the user processes execute more than eight times as many instructions, so that the overall miss rate is determined primarily by the miss rate in user code, which, as we will see, is often one-fifth of the kernel miss rate.

Figure 6.18 shows the data miss rate versus data cache size for the kernel and user components. The misses can be broken into three significant classes:

- Compulsory, or cold, misses represent the first access to this block by this pro-

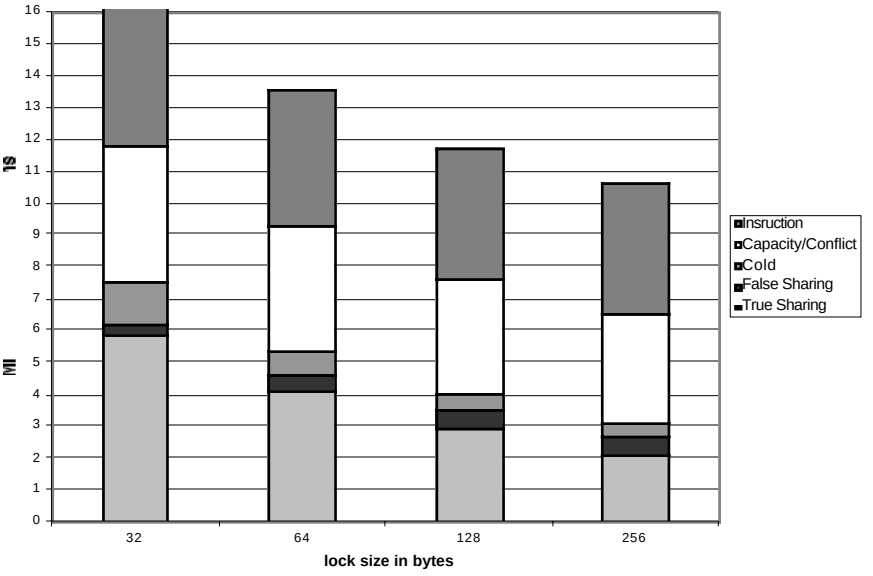


FIGURE 6.17 The number of misses per one-thousand instructions drops steadily as the block size of the L3 cache is increased making a good case for an L3 block size of at least 128 bytes. The L3 cache is a 2MB, 2-way set associative,

cessor and are significant in this workload.

- Coherence misses represent misses due to invalidations.
- Normal capacity misses include misses caused by interference between the OS and the user process and between multiple user processes. Conflict misses are included in this category.

For this workload the behavior of the operating system is more complex than the user processes. This is for two reasons. First, the kernel initializes all pages before allocating them to a user, which significantly increases the compulsory component of the kernel's miss rate. Second, the kernel actually shares data and thus has a nontrivial coherence miss rate. In contrast, user processes cause coherence misses only when the process is scheduled on a different processor; this component of the miss rate is small. Figure 6.19 shows the breakdown of the kernel miss rate as the cache size is increased.

Increasing the block size is likely to have beneficial effects for this workload, since a larger fraction of the misses arise from compulsory and capacity, both of which can be potentially improved with larger block sizes. Since coherence misses are relatively more rare, the negative effects of increasing block size should be

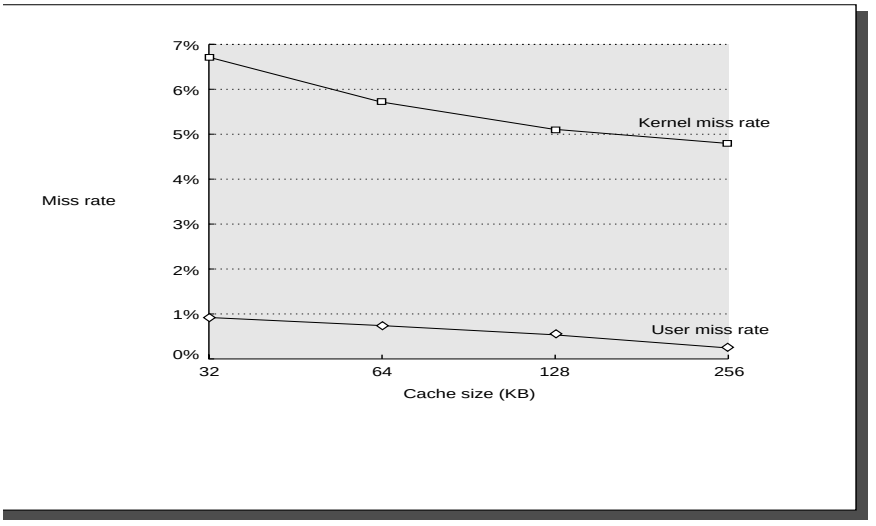


FIGURE 6.18 The data miss rate drops faster for the user code than for the kernel code as the data cache is increased from 32 KB to 256 KB with a 32-byte block. Although the user level miss rate drops by a factor of 3, the kernel level miss rate drops only by a factor of 1.3. As Figure 6.19 shows, this is due to a higher rate of compulsory misses and coherence misses. This multiprogramming workload is run on eight processors.

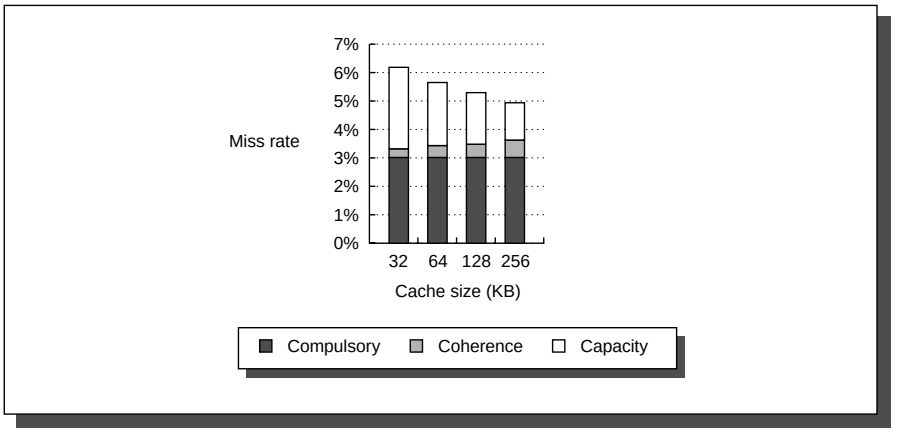


FIGURE 6.19 The components of the kernel data miss rate change as the data cache size is increased from 32KB to 256 KB, when the multiprogramming workload is run on eight processors. The compulsory miss rate component stays constant, since it is unaffected by cache size. The capacity component drops by more than a factor of two, while the coherence component nearly doubles. The increase in coherence misses occurs because the probability of a miss being caused by an invalidation increases with cache size, since fewer entries are bumped due to capacity.

small. Figure 6.20 shows how the miss rate for the kernel and user references changes as the block size is increased, assuming a 32 KB two-way set-associative data cache. Figure 6.21 confirms that, for the kernel references, the largest im-

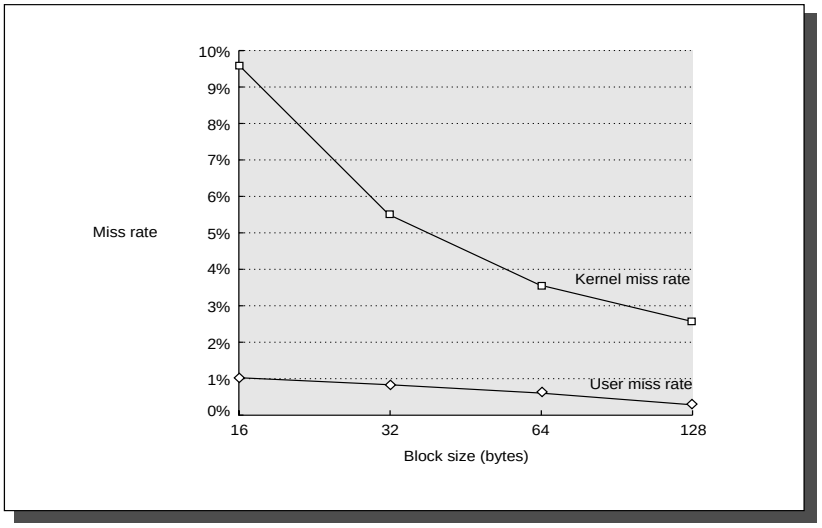


FIGURE 6.20 Miss rate for the multiprogramming workload drops steadily as the block size is increased for a 32-KB two-way set-associative data cache and an eight-CPU multiprocessor. As we might expect based on the higher compulsory component in the kernel, the improvement in miss rate for the kernel references is larger (almost a factor of 4 for the kernel references when going from 16-byte to 128-byte blocks versus just under a factor of 3 for the user references).

provement is the reduction of the compulsory miss rate. The absence of large increases in the coherence miss rate as block size is increased means that false sharing effects are insignificant.

If we examine the number of bytes needed per data reference, as in Figure 6.22, we see that the e kernel has a higher traffic ratio that grows quickly with block size. This is despite the significant reduction in compulsory misses; the smaller reduction in capacity and coherence misses drives an increase in total traffic. The user program has a much smaller traffic ratio that grows very slowly.

For the multiprogrammed workload, the OS is a much more demanding user of the memory system. If more OS or OS-like activity is included in the workload, it will become very difficult to build a sufficiently capable memory system.

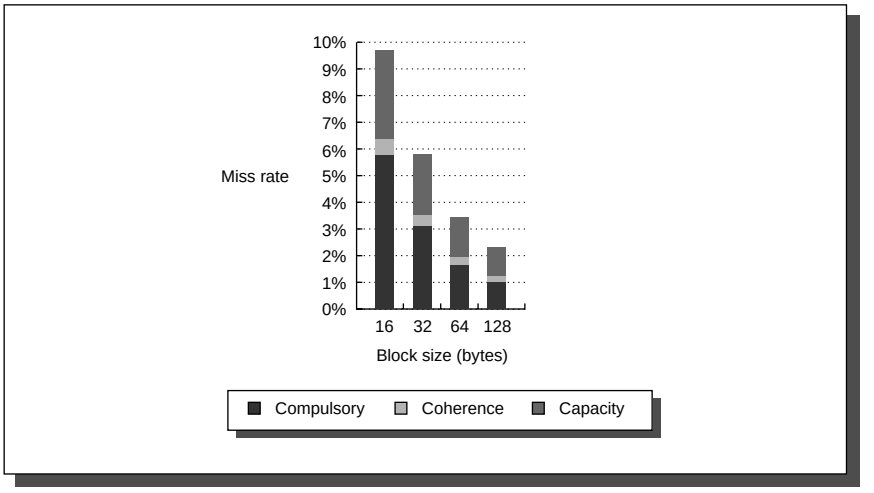


FIGURE 6.21 As we would expect, the increasing block size substantially reduces the compulsory miss rate in the kernel references. It also has a significant impact on the capacity miss rate, decreasing it by a factor of 2.4 over the range of block sizes. The increased block size has a small reduction in coherence traffic, which appears to stabilize at 64 bytes, with no change in the coherence miss rate in going to 128-byte lines. Because there are not significant reductions in the coherence miss rate as the block size increases, the fraction of the miss rate due to coherence grows from about 7% to about 15%.

Performance for the Scientific/Technical Workload

In this section, we use a simulator to study the performance of our four scientific parallel programs. For these measurements, the problem sizes are as follows:

- *Barnes-Hut*—16K bodies run for six time steps (the accuracy control is set to 1.0, a typical, realistic value);
- *FFT*—1 million complex data points
- *LU*—A 512×512 matrix is used with 16×16 blocks
- *Ocean*—A 130×130 grid with a typical error tolerance

In looking at the miss rates as we vary processor count, cache size, and block size, we decompose the total miss rate into *coherence misses* and normal uniprocessor misses. The normal uniprocessor misses consist of capacity, conflict, and compulsory misses. We label these misses as capacity misses, because that is the dominant cause for these benchmarks. For these measurements, we include as a coherence miss any write misses needed to upgrade a block from shared to exclusive, even though no one is sharing the cache block. This measurement reflects a protocol that does not distinguish between a private and shared cache block.

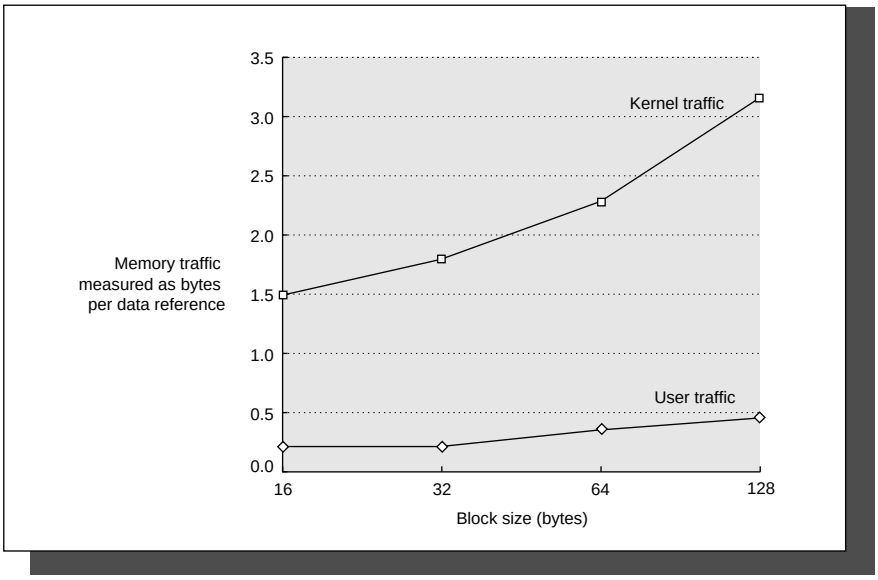


FIGURE 6.22 The number of bytes needed per data reference grows as block size is increased for both the kernel and user components. It is interesting to compare this chart against the same chart for the parallel program workload shown in Figure 6.26.

Figure 6.23 shows the data miss rates for our four applications, as we increase the number of processors from one to sixteen, while keeping the problem size constant. As we increase the number of processors, the total amount of cache increases, usually causing the capacity misses to drop. In contrast, increasing the processor count usually causes the amount of communication to increase, in turn causing the coherence misses to rise. The magnitude of these two effects differs by application.

In FFT, the capacity miss rate drops (from nearly 7% to just over 5%) but the coherence miss rate increases (from about 1% to about 2.7%), leading to a constant overall miss rate. Ocean shows a combination of effects, including some that relate to the partitioning of the grid and how grid boundaries map to cache blocks. For a typical 2D grid code the communication-generated misses are proportional to the boundary of each partition of the grid, while the capacity misses are proportional to the area of the grid. Therefore, increasing the total amount of cache while keeping the total problem size fixed will have a more significant effect on the capacity miss rate, at least until each subgrid fits within an individual processor's cache. The significant jump in miss rate between one and two proces-

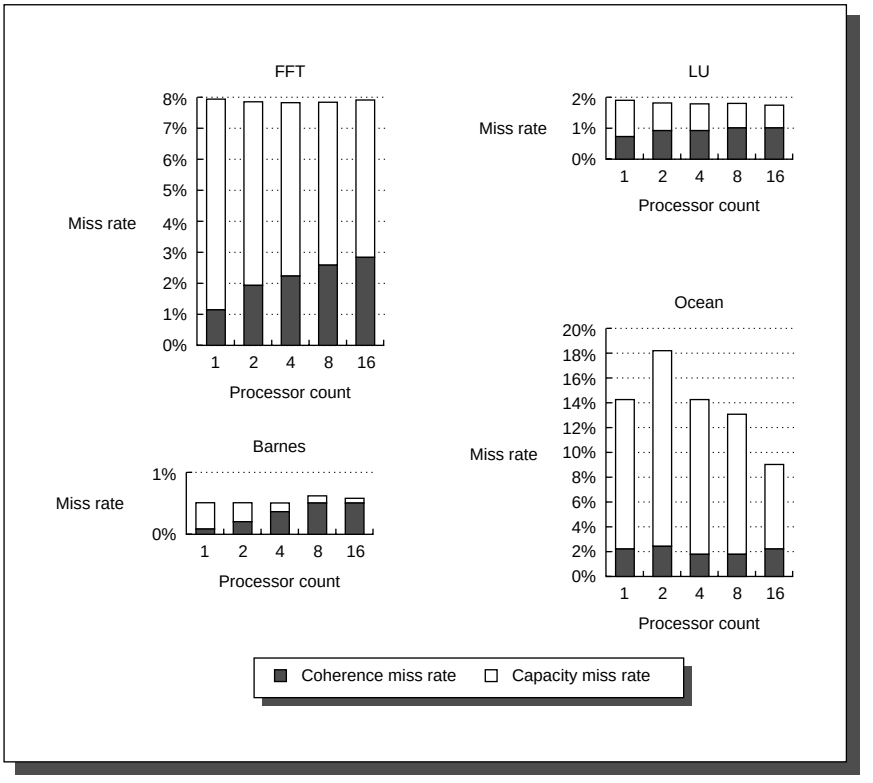


FIGURE 6.23 Data miss rates can vary in nonobvious ways as the processor count is increased from one to sixteen. The miss rates include both coherence and capacity miss rates. The compulsory misses in these benchmarks are all very small and are included in the capacity misses. Most of the misses in these applications are generated by accesses to data that is potentially shared, although in the applications with larger miss rates (FFT and Ocean), it is the capacity misses rather than the coherence misses that comprise the majority of the miss rate. Data is potentially shared if it is allocated in a portion of the address space used for shared data. In all except Ocean, the potentially shared data is heavily shared, while in Ocean only the boundaries of the subgrids are actually shared, although the entire grid is treated as a potentially shared data object. Of course, since the boundaries change as we increase the processor count (for a fixed-size problem), different amounts of the grid become shared. The anomalous increase in capacity miss rate for Ocean in moving from one to two processors arises because of conflict misses in accessing the subgrids. In all cases except Ocean, the fraction of the cache misses caused by coherence transactions rises when a fixed-size problem is run on an increasing number of processors. In Ocean, the coherence misses initially fall as we add processors due to a large number of misses that are write ownership misses to data that is potentially, but not actually, shared. As the subgrids begin to fit in the aggregate cache (around 16 processors), this effect lessens. The single processor numbers include write upgrade misses, which occur in this protocol even if the data is not actually shared, since it is in the shared state. For all these runs, the cache size is 64 KB, two-way set associative, with 32-byte blocks. Notice that the scale on the y-axis for each benchmark is different, so that the behavior of the individual benchmarks can be seen clearly.

sors occurs because of conflicts that arise from the way in which the multiple grids are mapped to the caches. This conflict is present for direct-mapped and two-way set associative caches, but fades at higher associativities. Such conflicts are not unusual in array-based applications, especially when there are multiple grids in use at once. In Barnes and LU the increase in processor count has little effect on the miss rate, sometimes causing a slight increase and sometimes causing a slight decrease.

Increasing the cache size usually has a beneficial effect on performance, since it reduces the frequency of costly cache misses. Figure 6.24 illustrates the change in miss rate as cache size is increased for 16 processors, showing the portion of the miss rate due to coherence misses and to uniprocessor capacity misses. Two effects can lead to a miss rate that does not decrease—at least not as quickly as we might expect—as cache size increases: inherent communication and plateaus in the miss rate. Inherent communication leads to a certain frequency of coherence misses that are not significantly affected by increasing cache size. Thus if the cache size is increased while maintaining a fixed problem size, the coherence miss rate eventually limits the decrease in cache miss rate. This effect is most obvious in Barnes, where the coherence miss rate essentially becomes the entire miss rate.

A less important effect is a temporary plateau in the capacity miss rate that arises when the application has some fraction of its data present in cache but some significant portion of the data set does not fit in the cache or in caches that are slightly bigger. In LU, a very small cache (about 4 KB) can capture the pair of 16×16 blocks used in the inner loop; beyond that the next big improvement in capacity miss rate occurs when both matrices fit in the caches, which occurs when the total cache size is between 4 MB and 8 MB. This effect, sometimes called a *working set effect*, is partly at work between 32 KB and 128 KB for FFT, where the capacity miss rate drops only 0.3%. Beyond that cache size, a faster decrease in the capacity miss rate is seen, as a major data structure begins to reside in the cache. These plateaus are common in programs that deal with large arrays in a structured fashion.

Increasing the block size is another way to change the miss rate in a cache. In uniprocessors, larger block sizes are often optimal with larger caches. In multiprocessors, two new effects come into play: a reduction in spatial locality for shared data and a potential increase in miss rate due to false sharing. Several studies have shown that shared data have lower spatial locality than unshared data. Poorer locality means that for shared data, fetching larger blocks is less effective than in a uniprocessor, because the probability is higher that the block will be replaced before all its contents are referenced. Likewise, increasing the basic size also increases the potential frequency of false sharing, increasing the miss rate.

Figure 6.25 shows the miss rates as the cache block size is increased for a 16-processor run with a 64-KB cache. The most interesting behavior is in Barnes, where the miss rate initially declines and then rises due to an increase in the num-

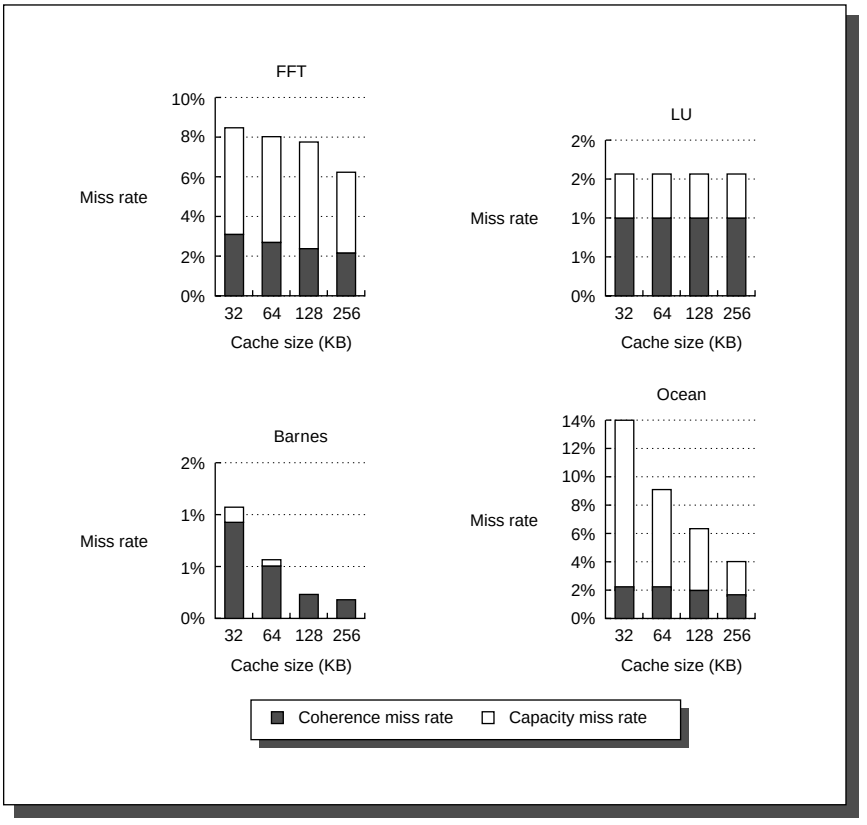


FIGURE 6.24 The miss rate usually drops as the cache size is increased, although coherence misses dampen the effect. The block size is 32 bytes and the cache is two-way set-associative. The processor count is fixed at 16 processors. Observe that the scale for each graph is different.

ber of coherence misses, which probably occurs because of false sharing. In the other benchmarks, increasing the block size decreases the overall miss rate. In Ocean and LU, the block size increase affects both the coherence and capacity miss rates about equally. In FFT, the coherence miss rate is actually decreased at a faster rate than the capacity miss rate. This reduction occurs because the communication in FFT is structured to be very efficient. In less optimized programs, we would expect more false sharing and less spatial locality for shared data, resulting in more behavior like that of Barnes.

Although the drop in miss rates with longer blocks may lead you to believe that choosing a longer block size is the best decision, the bottleneck in bus-based

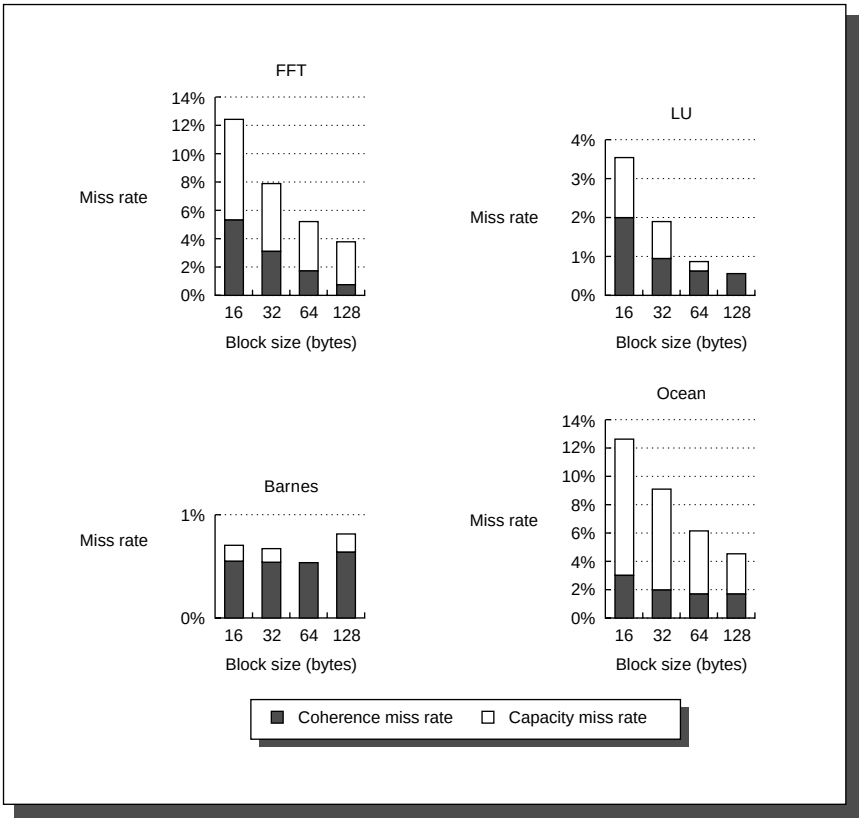


FIGURE 6.25 The data miss rate drops as the cache block size is increased. All these results are for a 16-processor run with a 64-KB cache and two-way set associativity. Once again we use different scales for each benchmark.

multiprocessors is often the limited memory and bus bandwidth. Larger blocks mean more bytes on the bus per miss. Figure 6.26 shows the growth in bus traffic as the block size is increased. This growth is most serious in the programs that have a high miss rate, especially Ocean. The growth in traffic can actually lead to performance slowdowns due both to longer miss penalties and to increased bus contention.

Summary: Performance of Snooping Cache Schemes

In this section we examined the cache performance of three very different workloads. We saw that the coherence traffic can introduce new behaviors in the memory system that do not respond as easily to changes in cache size or block size that are normally used to improve uniprocessor cache performance.

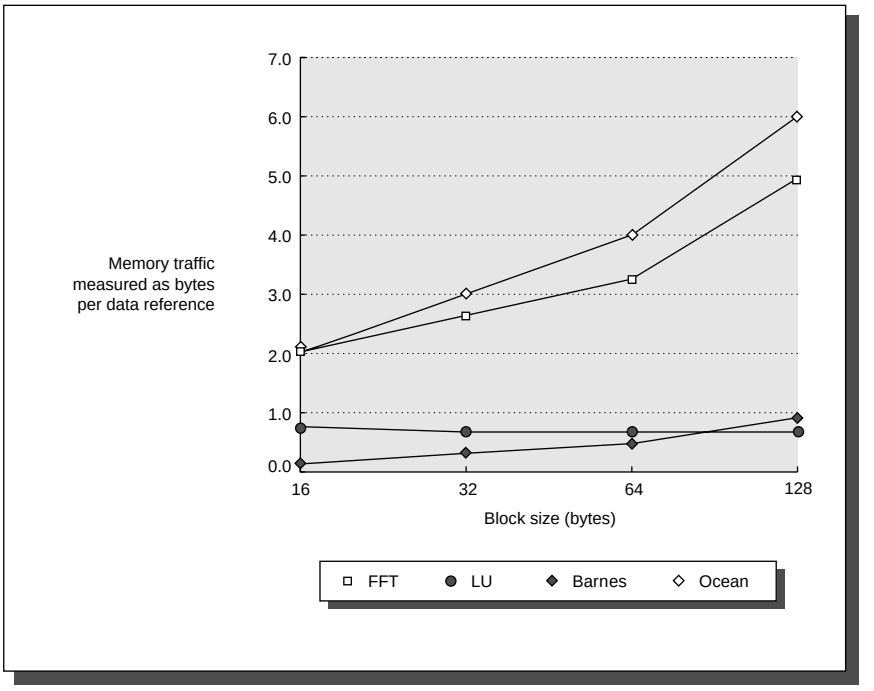


FIGURE 6.26 Bus traffic for data misses climbs steadily as the block size in the data cache is increased. The factor of 3 increase in traffic for Ocean is the best argument against larger block sizes. Remember that our protocol treats ownership or upgrade misses the same as other misses, slightly increasing the penalty for large cache blocks; in both Ocean and FFT this simplification accounts for less than 10% of the traffic.

In the commercial workload, the performance of the web searching and DSS benchmarks is reasonable (CPI of 1.3 and 1.6, respectively), while the OLTP benchmark is much worse (CPI=7.0). For OLTP, the large instruction working set demands a large cache to achieve acceptable performance. Increasing the cache size reduces the execution time, but is limited by the true and false sharing misses, which do not decrease as the cache grows. Similarly, increasing the processor counts increases true and false sharing, leading to an increase in memory access cycles. Fortunately, this workload responds favorably to an increase in block size, although the instruction miss rate remains similar. For these large workloads, it appears that very large (≥4 MB) off-chip caches with large block sizes (64-128 bytes) could work reasonably well.

In the multiprogrammed workload, the user and OS portions perform very differently, although neither has significant coherence traffic. In the OS portion, the compulsory and capacity contributions to the miss rate are much larger, leading to overall miss rates that are comparable to the worst programs in the parallel

program workload. User cache performance, on the other hand, is very good and compares to the best programs in the parallel program workload.

Coherence requests are a significant but not overwhelming component in the scientific processing workload. We can expect, however, that coherence requests will be more important in parallel programs that are less optimized.

The question of how these cache miss rates affect CPU performance depends on the rest of the memory system, including the latency and bandwidth of the interconnect and memory, a topic we return to in Section 6.11.

6.5 Distributed Shared-Memory Architectures

A scalable multiprocessor supporting shared memory could choose to exclude or include cache coherence. The simplest scheme for the hardware is to exclude cache coherence, focusing instead on a scalable memory system. Several companies have built this style of multiprocessor; the Cray T3D/E is best-known example. In such multiprocessors, memory is distributed among the nodes and all nodes are interconnected by a network. Access can be either local or remote—a controller inside each node decides, on the basis of the address, whether the data resides in the local memory or in a remote memory. In the latter case a message is sent to the controller in the remote memory to access the data.

These systems have caches, but to prevent coherence problems, shared data is marked as uncacheable and only private data is kept in the caches. Of course, software can still explicitly cache the value of shared data by copying the data from the shared portion of the address space to the local private portion of the address space that is cached. Coherence is then controlled by software. The advantage of such a mechanism is that little hardware support is required, although support for features such as block copy may be useful, since remote accesses fetch only single words (or double words) rather than cache blocks.

There are several disadvantages to this approach. First, compiler mechanisms for transparent software cache coherence are very limited. The techniques that currently exist apply primarily to programs with well-structured loop-level parallelism or a very strict form of object-oriented programming, and these techniques have significant overhead arising from explicitly copying data. For irregular problems or problems involving dynamic data structures and pointers (including operating systems, for example), compiler-based software cache coherence is currently impractical. The basic difficulty is that software-based coherence algorithms must be conservative: every block that *might* be shared must be treated as

if it *is* shared. Being conservative results in excess coherence overhead, because the compiler cannot predict the actual sharing accurately enough. Due to the complexity of the possible interactions, asking programmers to deal with coherence is unworkable.

Second, without cache coherence, the multiprocessor loses the advantage of being able to fetch and use multiple words in a single cache block for close to the cost of fetching one word. The benefits of spatial locality in shared data cannot be leveraged when single words are fetched from a remote memory for each reference. Support for a DMA mechanism among memories can help, but such mechanisms are often either costly to use (since they may require OS intervention) or expensive to implement since special-purpose hardware support and a buffer are needed. For message-passing programs, however, such mechanisms can be extremely useful, since programmers can overcome the usage penalties by using large messages.

Third, mechanisms for tolerating latency such as prefetch are more useful when they can fetch multiple words, such as a cache block, and where the fetched data remain coherent; we will examine this advantage in more detail later.

These disadvantages are magnified by the large latency of access to remote memory versus a local cache. For example, on the Cray T3E a local cache access has a latency of two cycles and is pipelined. A remote memory access takes up to 400 processor clock cycles for a remote memory using the 450 MHz Alpha processor in the T3E-900.

For these reasons, cache coherence is an accepted requirement in small-scale multiprocessors. For larger-scale architectures, there are new challenges to extending the cache-coherent shared-memory model. Although the bus can certainly be replaced with a more scalable interconnection network (the SUN Enterprise servers use up to four buses, e.g.), and we could certainly distribute the memory so that the memory bandwidth could also be scaled, the lack of scalability of the snooping coherence scheme needs to be addressed.

A snooping protocol requires communication with all caches on every cache miss, including writes of potentially shared data. The absence of any centralized data structure that tracks the state of the caches is both the fundamental advantage of a snooping-based scheme, since it allows it to be inexpensive, as well as its Achilles' heel when it comes to scalability. For example, with only 16 proces-

sors and a block size of 64 bytes and a 512 KB data cache, the total bus bandwidth demand (ignoring stall cycles) for the four programs in the scientific/technical workload ranges from about 1 GB/sec (for Barnes) to about 42 GB/sec (for FTT), assuming a processor that sustains a data reference every 1 ns, which is what a 2000 superscalar processor with nonblocking caches might generate. In comparison, the Sun Enterprise system with the Starfire interconnect, the highest bandwidth SMP system in 2000, can support about 12 GB/sec of random accesses for the 16x16 crossbar and has a maximum bandwidth of 21.3 GB/sec at the memory system. Although the cache size used in these simulations is moderate (though large enough to eliminate much of the uniprocessor miss traffic), so is the problem size.

Alternatively, we could build scalable shared-memory architectures that include cache coherency. The key is to find an alternative coherence protocol to the snooping protocol. One alternative protocol is a directory protocol. A *directory* keeps the state of every block that may be cached. Information in the directory includes which caches have copies of the block, whether it is dirty, and so on. (Section 6.11 on page 735 describes a hybrid approach that uses directories to extend a snooping protocol.)

Existing directory implementations associate an entry in the directory with each memory block. In typical protocols, the amount of information is proportional to the product of the number of memory blocks and the number of processors. This overhead is not a problem for multiprocessors with less than about two hundred processors, because the directory overhead will be tolerable. For larger multiprocessors, we need methods to allow the directory structure to be efficiently scaled. The methods that have been proposed either try to keep information for fewer blocks (e.g., only those in caches rather than all memory blocks) or try to keep fewer bits per entry.

To prevent the directory from becoming the bottleneck, directory entries can be distributed along with the memory, so that different directory accesses can go to different locations, just as different memory requests go to different memories. A distributed directory retains the characteristic that the sharing status of a block is always in a single known location. This property is what allows the coherence protocol to avoid broadcast. Figure 6.27 shows how our distributed-memory multiprocessor looks with the directories added to each node.

Directory-Based Cache-Coherence Protocols: The Basics

Just as with a snooping protocol, there are two primary operations that a directory protocol must implement: handling a read miss and handling a write to a shared,

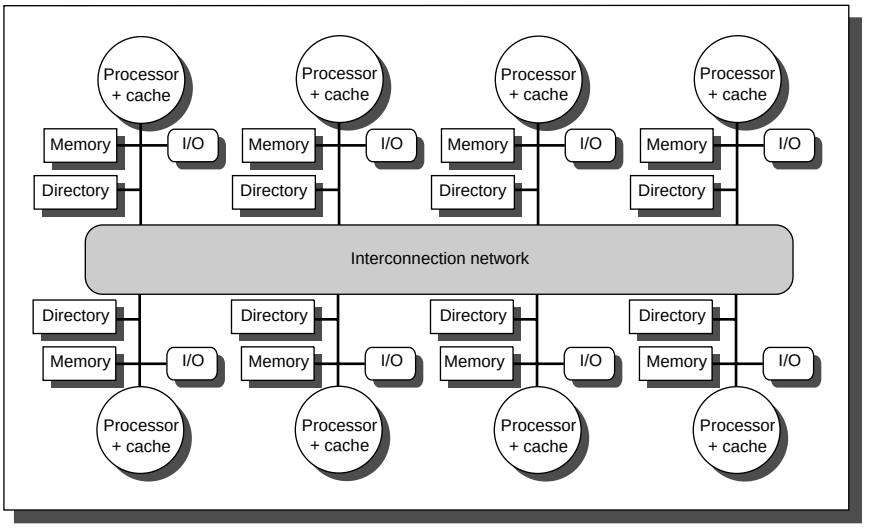


FIGURE 6.27 A directory is added to each node to implement cache coherence in a distributed-memory multiprocessor. Each directory is responsible for tracking the caches that share the memory addresses of the portion of memory in the node. The directory may communicate with the processor and memory over a common bus, as shown, or it may have a separate port to memory, or it may be part of a central node controller through which all intranode and internode communications pass.

clean cache block. (Handling a write miss to a shared block is a simple combination of these two.) To implement these operations, a directory must track the state of each cache block. In a simple protocol, these states could be the following:

- *Shared*—One or more processors have the block cached, and the value in memory is up to date (as well as in all the caches).
- *Uncached*—No processor has a copy of the cache block.
- *Exclusive*—Exactly one processor has a copy of the cache block and it has written the block, so the memory copy is out of date. The processor is called the *owner* of the block.

In addition to tracking the state of each cache block, we must track the processors that have copies of the block when it is shared, since they will need to be invalidated on a write. The simplest way to do this is to keep a bit vector for each memory block. When the block is shared, each bit of the vector indicates whether the corresponding processor has a copy of that block. We can also use the bit vec-

tor to keep track of the owner of the block when the block is in the exclusive state. For efficiency reasons, we also track the state of each cache block at the individual caches.

The states and transitions for the state machine at each cache are identical to what we used for the snooping cache, although the actions on a transition are slightly different. We make the same simplifying assumptions that we made in the case of the snooping cache: attempts to write data that is not exclusive in the writer's cache always generate write misses, and the processors block until an access completes. Since the interconnect is no longer a bus and since we want to avoid broadcast, there are two additional complications. First, we cannot use the interconnect as a single point of arbitration, a function the bus performed in the snooping case. Second, because the interconnect is message oriented (unlike the bus, which is transaction oriented), many messages must have explicit responses.

Before we see the protocol state diagrams, it is useful to examine a catalog of the message types that may be sent between the processors and the directories. Figure 6.28 shows the type of messages sent among nodes. The *local* node is the node where a request originates. The *home* node is the node where the memory location and the directory entry of an address reside. The physical address space is statically distributed, so the node that contains the memory and directory for a given physical address is known. For example, the high-order bits may provide the node number, while the low-order bits provide the offset within the memory on that node. The local node may also be the home node. The directory must be accessed when the home node is the local node, since copies may exist in yet a third node, called a remote node.

A *remote* node is the node that has a copy of a cache block, whether exclusive (in which case it is the only copy) or shared. A remote node may be the same as either the local node or the home node. In such cases, the basic protocol does not change, but interprocessor messages may be replaced with intraprocessor messages.

In this section, we assume a simple model of memory consistency. To minimize the type of messages and the complexity of the protocol, we make an assumption that messages will be received and acted upon in the same order they are sent. This assumption may not be true in practice, and can result in additional complications, some of which we address in section 6.8 when we discuss memory consistency models. In this section, we use this assumption to ensure that invalidates sent by a processor are honored immediately.

Message type	Source	Destination	Message contents	Function of this message
Read miss	Local cache	Home directory	P, A	Processor P has a read miss at address A; request data and make P a read sharer.
Write miss	Local cache	Home directory	P, A	Processor P has a write miss at address A; — request data and make P the exclusive owner.
Invalidate	Home directory	Remote cache	A	Invalidate a shared copy of data at address A.
Fetch	Home directory	Remote cache	A	Fetch the block at address A and send it to its home directory; change the state of A in the remote cache to shared.
Fetch/invalidate	Home directory	Remote cache	A	Fetch the block at address A and send it to its home directory; invalidate the block in the cache.
Data value reply	Home directory	Local cache	D	Return a data value from the home memory.
Data write back	Remote cache	Home directory	A, D	Write back a data value for address A.

FIGURE 6.28 The possible messages sent among nodes to maintain coherence are shown with the source and destination node, the contents (where P=requesting processor number), A=requested address, and D=data contents), and the function of the message. The first two messages are miss requests sent by the local cache to the home. The third through fifth messages are messages sent to a remote cache by the home when the home needs the data to satisfy a read or write miss request. Data value replies are used to send a value from the home node back to the requesting node. Data value write backs occur for two reasons: when a block is replaced in a cache and must be written back to its home memory, and also in reply to fetch or fetch/invalidate messages from the home. Writing back the data value whenever the block becomes shared simplifies the number of states in the protocol, since any dirty block must be exclusive and any shared block is always available in the home memory.

An Example Directory Protocol

The basic states of a cache block in a directory-based protocol are exactly like those in a snooping protocol, and the states in the directory are also analogous to those we showed earlier. Thus we can start with simple state diagrams that show

the state transitions for an individual cache block and then examine the state diagram for the directory entry corresponding to each block in memory. As in the snooping case, these state transition diagrams do not represent all the details of a coherence protocol; however, the actual controller is highly dependent on a number of details of the multiprocessor (message delivery properties, buffering structures, and so on). In this section we present the basic protocol state diagrams. The knotty issues involved in implementing these state transition diagrams are examined in Appendix E, along with similar problems that arise for snooping caches.

Figure 6.29 shows the protocol actions to which an individual cache responds.

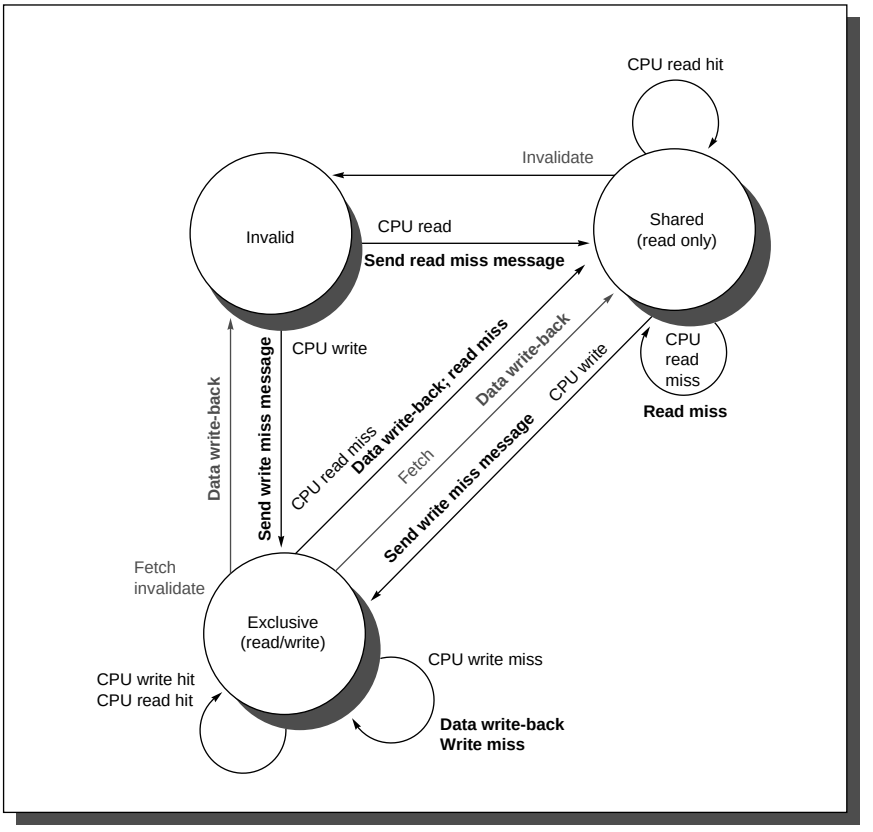


FIGURE 6.29 State transition diagram for an individual cache block in a directory-based system. Requests by the local processor are shown in black and those from the home directory are shown in gray. The states are identical to those in the snooping case, and the transactions are very similar, with explicit invalidate and write-back requests replacing the write misses that were formerly broadcast on the bus. As we did for the snooping controller, we assume that an attempt to write a shared cache block is treated as a miss; in practice, such a transaction can be treated as an ownership request or upgrade request and can deliver ownership without requiring that the cache block be fetched.

We use the same notation as in the last section, with requests coming from outside the node in gray and actions in bold. The state transitions for an individual cache are caused by read misses, write misses, invalidates, and data fetch requests; these operations are all shown in Figure 6.29. An individual cache also generates read and write miss messages that are sent to the home directory. Read

and write misses require data value replies, and these events wait for replies before changing state.

The operation of the state transition diagram for a cache block in Figure 6.29 is essentially the same as it is for the snooping case: the states are identical, and the stimulus is almost identical. The write miss operation, which was broadcast on the bus in the snooping scheme, is replaced by the data fetch and invalidate operations that are selectively sent by the directory controller. Like the snooping protocol, any cache block must be in the exclusive state when it is written and any shared block must be up to date in memory.

In a directory-based protocol, the directory implements the other half of the coherence protocol. A message sent to a directory causes two different types of actions: updates of the directory state, and sending additional messages to satisfy the request. The states in the directory represent the three standard states for a block; unlike in a snoopy scheme, however, the directory state indicates the state of all the cached copies of a memory block, rather than for a single cache block. The memory block may be uncached by any node, cached in multiple nodes and readable (shared), or cached exclusively and writable in exactly one node. In addition to the state of each block, the directory must track the set of processors that have a copy of a block; we use a set called *Sharers* to perform this function. In multiprocessors with less than 64 nodes (which may represent 2-4 times as many processors), this set is typically kept as a bit vector. In larger multiprocessors, other techniques, which we discuss in the Exercises, are needed. Directory requests need to update the set Sharers and also read the set to perform invalidations.

Figure 6.30 shows the actions taken at the directory in response to messages received. The directory receives three different requests: read miss, write miss, and data write back. The messages sent in response by the directory are shown in bold, while the updating of the set Sharers is shown in bold italics. Because all the stimulus messages are external, all actions are shown in gray. Our simplified protocol assumes that some actions are atomic, such as requesting a value and sending it to another node; a realistic implementation cannot use this assumption.

To understand these directory operations, let's examine the requests received and actions taken state by state. When a block is in the uncached state the copy in memory is the current value, so the only possible requests for that block are

- *Read miss*—The requesting processor is sent the requested data from memory and the requestor is made the only sharing node. The state of the block is made shared.
- *Write miss*—The requesting processor is sent the value and becomes the Sharing node. The block is made exclusive to indicate that the only valid copy is cached. Sharers indicates the identity of the owner.

When the block is in the shared state the memory value is up-to-date, so the same two requests can occur:

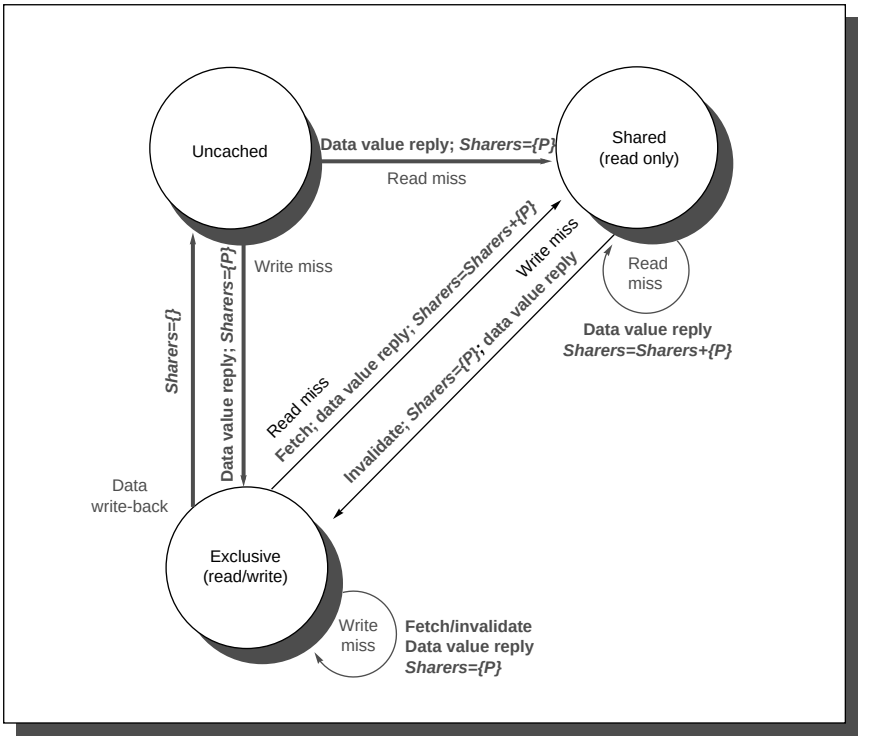


FIGURE 6.30 The state transition diagram for the directory has the same states and structure as the transition diagram for an individual cache. All actions are in gray because they are all externally caused. Bold indicates the action taken by the directory in response to the request. Bold italics indicate an action that updates the sharing set, Sharers, as opposed to sending a message.

- *Read miss*—The requesting processor is sent the requested data from memory and the requesting processor is added to the sharing set.
- *Write miss*—The requesting processor is sent the value. All processors in the set Sharers are sent invalidate messages, and the Sharers set is to contain the identity of the requesting processor. The state of the block is made exclusive.

When the block is in the exclusive state the current value of the block is held in the cache of the processor identified by the set sharers (the owner), so there are three possible directory requests:

- *Read miss*—The owner processor is sent a data fetch message, which causes the state of the block in the owner's cache to transition to shared and causes the owner to send the data to the directory, where it is written to memory and sent back to the requesting processor. The identity of the requesting processor is added to the set sharers, which still contains the identity of the processor that

was the owner (since it still has a readable copy).

- *Data write-back*—The owner processor is replacing the block and therefore must write it back. This write-back makes the memory copy up to date (the home directory essentially becomes the owner), the block is now uncached, and the sharer set is empty.
- *Write miss*—The block has a new owner. A message is sent to the old owner causing the cache to invalidate the block and send the value to the directory, from which it is sent to the requesting processor, which becomes the new owner. Sharers is set to the identity of the new owner, and the state of the block remains exclusive.

This state transition diagram in Figure 6.30 is a simplification, just as it was in the snooping cache case. In the directory case it is a larger simplification, since our assumption that bus transactions are atomic no longer applies. Appendix E explores these issues in depth.

In addition, the directory protocols used in real multiprocessors contain additional optimizations. In particular, in this protocol when a read or write miss occurs for a block that is exclusive, the block is first sent to the directory at the home node. From there it is stored into the home memory and also sent to the original requesting node. Many of the protocols in use in commercial multiprocessors forward the data from the owner node to the requesting node directly (as well as performing the write back to the home). Such optimizations often add complexity by increasing the possibility of deadlock and by increasing the types of messages that must be handled.

6.6 Performance of Distributed Shared-Memory Multiprocessors

The performance of a directory-based multiprocessor depends on many of the same factors that influence the performance of bus-based multiprocessors (e.g., cache size, processor count, and block size), as well as the distribution of misses to various locations in the memory hierarchy. The location of a requested data item depends on both the initial allocation and the sharing patterns. We start by examining the basic cache performance of our scientific/technical workload and then look at the effect of different types of misses.

Because the multiprocessor is larger and has longer latencies than our snooping-based multiprocessor, we begin with a slightly larger cache (128 KB) and a larger block size of 64 bytes.

In distributed memory architectures, the distribution of memory requests between local and remote is key to performance, because it affects both the consumption of global bandwidth and the latency seen by requests. Therefore, for the figures in this section we separate the cache misses into local and remote requests. In looking at the figures, keep in mind that, for these applications, most of the remote misses that arise are coherence misses, although some capacity misses can also be remote, and in some applications with poor data distribution, such misses can be significant (see the Pitfall on page 758).

As Figure 6.31 shows, the miss rates with these cache sizes are not affected much by changes in processor count, with the exception of Ocean, where the miss rate rises at 64 processors. This rise results from two factors: an increase in mapping conflicts in the cache that occur when the grid becomes small, which leads to a rise in local misses, and an increase in the number of the coherence misses, which are all remote.

Figure 6.32 shows how the miss rates change as the cache size is increased, assuming a 64-processor execution and 64-byte blocks. These miss rates decrease at rates that we might expect, although the dampening effect caused by little or no reduction in coherence misses leads to a slower decrease in the remote misses than in the local misses. By the time we reach the largest cache size shown, 512 KB, the remote miss rate is equal to or greater than the local miss rate. Larger caches would amplify this trend.

We examine the effect of changing the block size in Figure 6.33. Because these applications have good spatial locality, increases in block size reduce the miss rate, even for large blocks, although the performance benefits for going to the largest blocks are small. Furthermore, most of the improvement in miss rate comes from a reduction in the local misses.

Rather than plot the memory traffic, Figure 6.34 plots the number of bytes required per data reference versus block size, breaking the requirement into local and global bandwidth. In the case of a bus, we can simply aggregate the demands of each processor to find the total demand for bus and memory bandwidth. For a scalable interconnect, we can use the data in Figure 6.34 to compute the required per-node global bandwidth and the estimated bisection bandwidth, as the next Example shows.

EXAMPLE Assume a 64-processor multiprocessor with 1GHz processors that sustain one memory reference per processor clock. For a 64-byte block size, the remote miss rate is 0.7%. Find the per-node and estimated bisection bandwidth for FFT. Assume that the processor does not stall for remote memory requests; this might be true if, for example, all remote data were prefetched. How do these bandwidth requirements compare to various interconnection technologies?

ANSWER The per-node bandwidth is simply the number of data bytes per reference

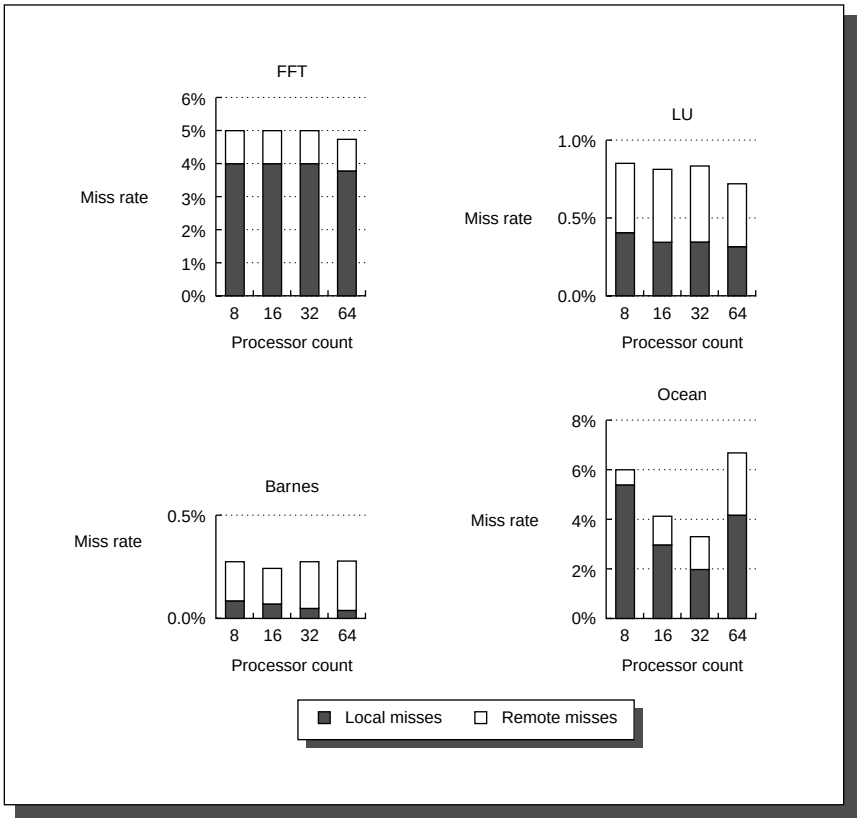


FIGURE 6.31 The data miss rate is often steady as processors are added for these benchmarks. Because of its grid structure, Ocean has an initially decreasing miss rate, which rises when there are 64 processors. For Ocean, the local miss rate drops from 5% at 8 processors to 2% at 32, before rising to 4% at 64. The remote miss rate in Ocean, driven primarily by communication, rises monotonically from 1% to 2.5%. Note that to show the detailed behavior of each benchmark, different scales are used on the y-axis. The cache for all these runs is 128 KB, two-way set associative, with 64-byte blocks. Remote misses include any misses that require communication with another node, whether to fetch the data or to deliver an invalidate. In particular, in this figure and other data in this section, the measurement of remote misses includes write upgrade misses where the data is up to date in the local memory but cached elsewhere and, therefore, requires invalidations to be sent. Such invalidations do indeed generate remote traffic, but may or may not delay the write, depending on the consistency model (see section 6.8).

times the reference rate: $0.7\% \times 1000 \times 64$ = 448 MB/sec. This rate is somewhat higher than the hardware sustainable transfer rate for the CrayT3E (using a block prefetch) and lower than that for an SGI Origin 3000 (1.6 GB/processor pair). The FFT per-node bandwidth demand ex-

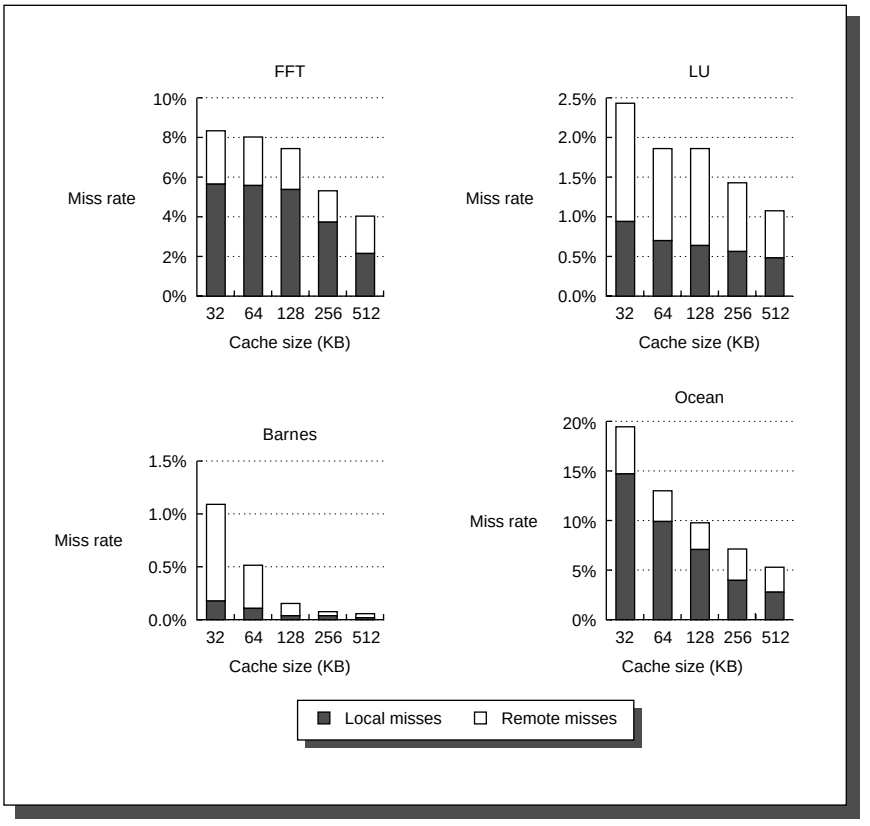


FIGURE 6.32 Miss rates decrease as cache sizes grow. Steady decreases are seen in the local miss rate, while the remote miss rate declines to varying degrees, depending on whether the remote miss rate had a large capacity component or was driven primarily by communication misses. In all cases, the decrease in the local miss rate is larger than the decrease in the remote miss rate. The plateau in the miss rate of FFT, which we mentioned in the last section, ends once the cache exceeds 128 KB. These runs were done with 64 processors and 64-byte cache blocks.

ceeds the bandwidth sustainable from the fastest standard networks by more than a factor of 5.

FFT performs all-to-all communication, so the bisection bandwidth is equal to the number of processors times the per-node bandwidth, or about 64 x 448 MB/sec = 28.7 GB/sec. The SGI Origin 3000 with 64-processors has a bisection bandwidth of about 50 GB/sec. No standard networking technology comes close. n

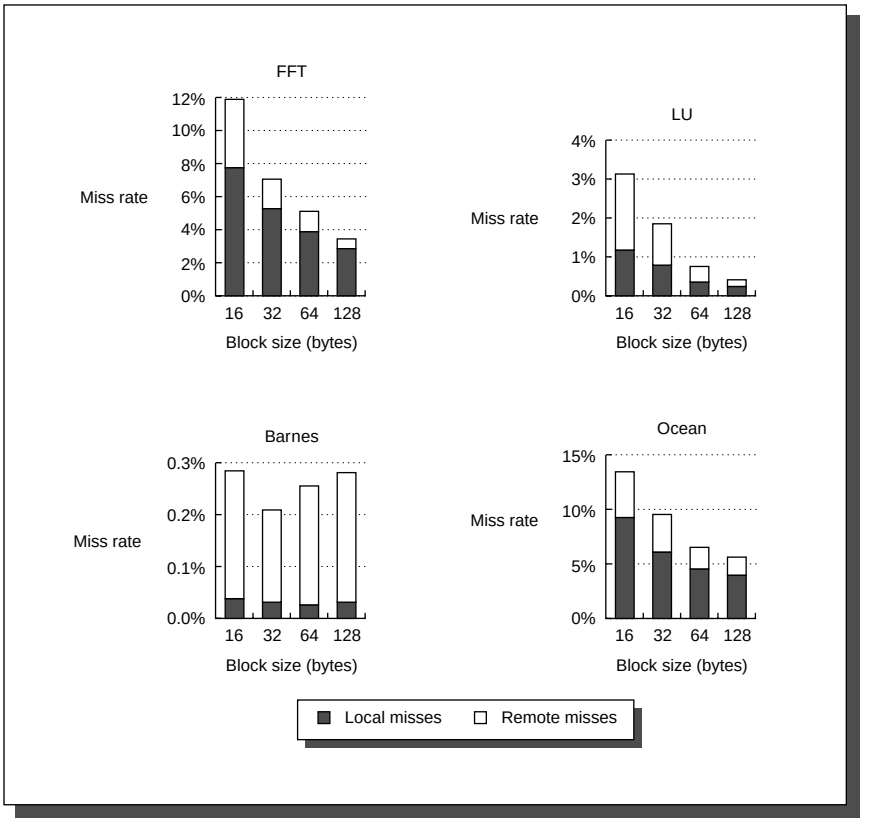


FIGURE 6.33 Data miss rate versus block size assuming a 128-KB cache and 64 processors in total. Although difficult to see, the coherence miss rate in Barnes actually rises for the largest block size, just as in the last section.

The previous Example looked at the bandwidth demands. The other key issue for a parallel program is remote memory access time, or latency. To get insight into this, we use a simple example of a directory-based multiprocessor. Figure 6.35 shows the parameters we assume for our simple multiprocessor model. It assumes that the time to first word for a local memory access is 85 processor cycles and that the path to local memory is 16 bytes wide, while the network interconnect is 4 bytes wide. This model ignores the effects of contention, which are probably not too serious in the parallel benchmarks we examine, with the possible exception of FFT, which uses all-to-all communication. Contention could have a serious performance impact in other workloads.

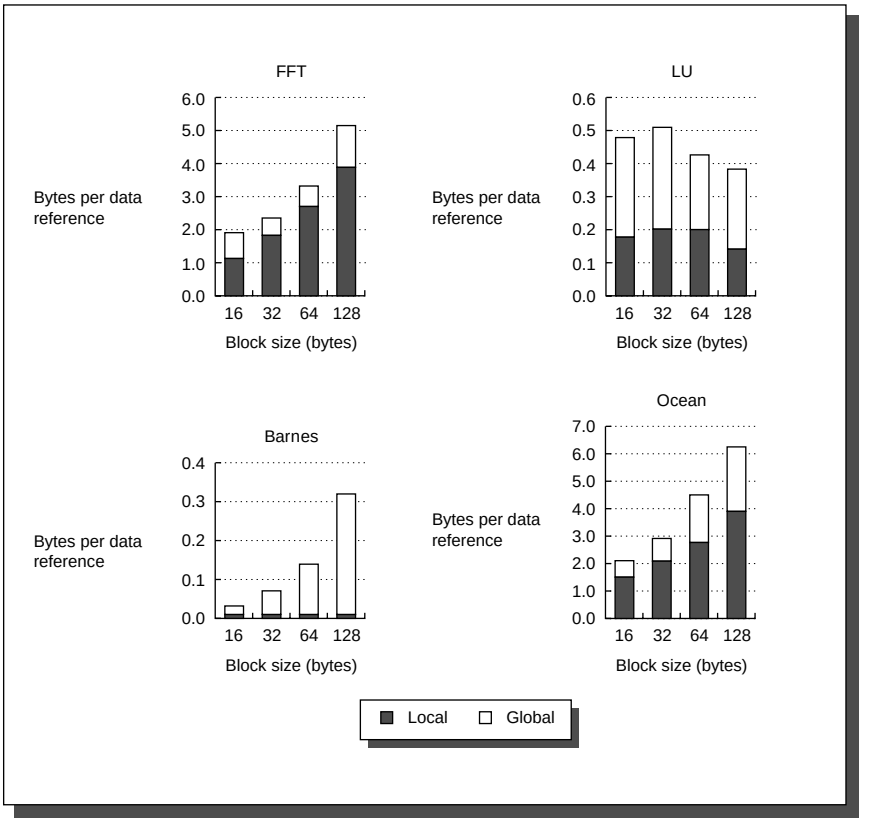


FIGURE 6.34 The number of bytes per data reference climbs steadily as block size is increased. These data can be used to determine the bandwidth required per node both internally and globally. The data assumes a 128-KB cache for each of 64 processors.

Figure 6.36 shows the cost in cycles for the average memory reference, assuming the parameters in Figure 6.35. Only the latencies for each reference type are counted. Each bar indicates the contribution from cache hits, local misses, remote misses, and 3-hop remote misses. The cost is influenced by the total frequency of cache misses and upgrades, as well as by the distribution of the location where the miss is satisfied. The cost for a remote memory reference is fairly steady as the processor count is increased, except for Ocean. The increasing miss rate in Ocean for 64 processors is clear in Figure 6.31. As the miss rate increases, we should expect the time spent on memory references to increase also.

Although Figure 6.36 shows the memory access cost, which is the dominant multiprocessor cost in these benchmarks, a complete performance model would

Characteristic	Processor clock cycles ≤ 16 processor	Processor clock cycles 17–64 processor
Cache hit	1	1
Cache miss to local memory	85	85
Cache miss to remote home directory	125	150
Cache miss to remotely cached data (3-hop miss)	140	170

FIGURE 6.35 Characteristics of the example directory-based multiprocessor. Misses can be serviced locally (including from the local directory), at a remote home node, or using the services of both the home node and another remote node that is caching an exclusive copy. This last case is called a 3-hop miss and has a higher cost because it requires interrogating both the home directory and a remote cache. Note that this simple model does not account for invalidation time, but does include some factor for increasing interconnect time. These remote access latencies are based on those in an SGI Origin 3000, the fastest scalable interconnect system in 2000, and assume a 500 MHz processor.

need to consider the effect of contention in the memory system, as well as the losses arising from synchronization delays.

The coherence protocols that we have discussed so far have made several simplifying assumptions. In practice, real protocols must deal with two realities: nonatomicity of operations and finite buffering. We have seen why certain operations (such as a write miss) cannot be atomic. In DSM multiprocessors the presence of only a finite number of buffers to hold message requests and replies introduces additional possibilities for deadlock. The challenge for the designer is to create a protocol that works correctly and without deadlock, using nonatomic actions and finite buffers as the building blocks. These factors are fundamental challenges in all parallel multiprocessors, and the solutions are applicable to a wide variety of protocol design environments, both in hardware and in software.

Because this material is extremely complex and not necessary to comprehend the rest of the chapter, we have placed it in Appendix E. For the interested reader, Appendix E shows how the specific problems in our coherence protocols are solved and illustrates the general principles that are more globally applicable. It describes the problems arising in snooping cache implementations, as well as the more complex problems that arise in more distributed systems using directories. If you want to understand how either state-of-the-art SMPs (which use split transactions buses and nonblocking memory accesses) or DSM multiprocessors really work and why designing them is such a challenge, go read Appendix E!

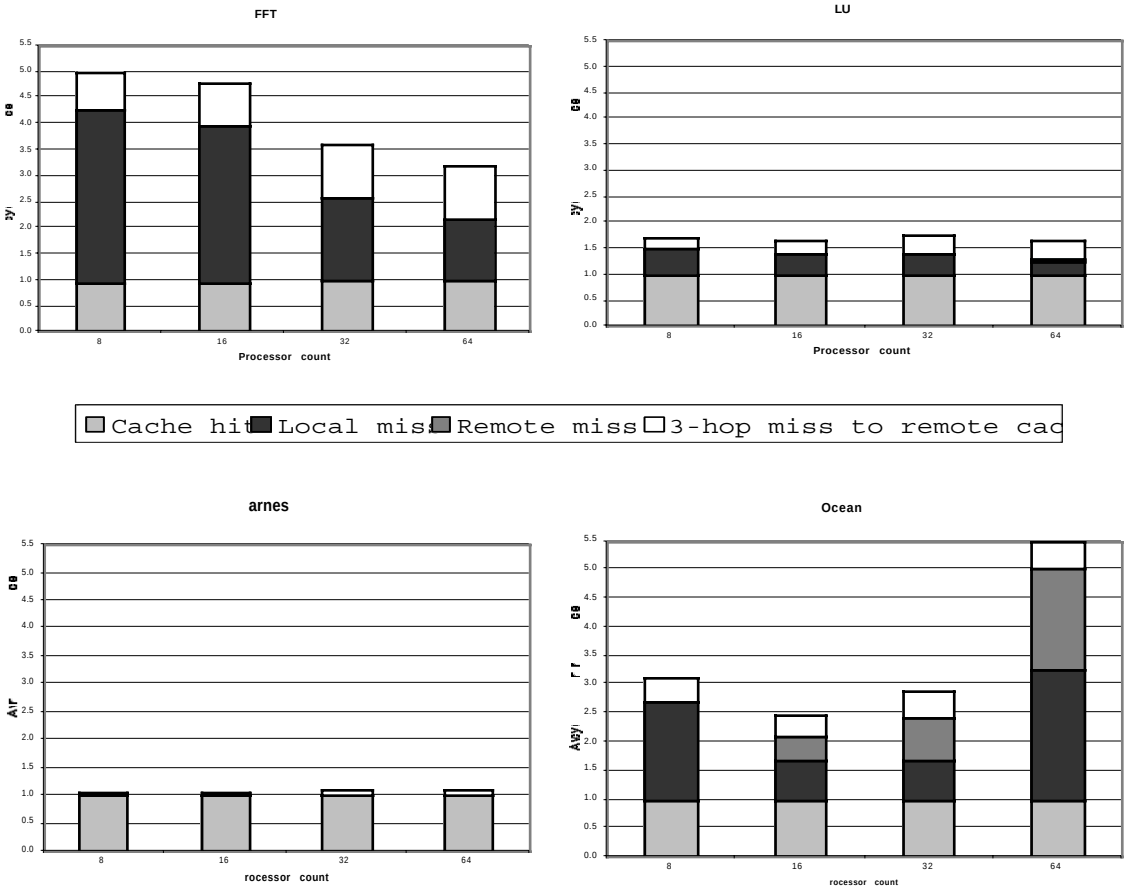


FIGURE 6.36 The effective latency of memory references in a DSM multiprocessor depends both on the relative frequency of cache misses and on the location of the memory where the accesses are served. These plots show the memory access cost (a metric called average memory access time in Chapter 5) for each of the benchmarks for 8, 16, 32, and 64 processors, assuming a 512KB data cache that is two-way set associative with 64-byte blocks. The average memory access cost is composed of four different types of accesses, with the cost of each type given in Figure 6.35. For the Barnes and LU benchmarks, the low miss rates lead to low overall access times. In FFT, the higher access cost is determined by a higher local miss rate (1-4%) and a significant 3-hop miss rate (1%). The improvement in FFT comes from the reduction in local miss rate from 4% to 1%, as the aggregate cache increases. Ocean shows the biggest change in the cost of memory accesses, and the highest overall cost at 64 processors. The high cost is driven primarily by a high local miss rate (average 1.6%). The memory access cost drops from 8 to 16 processors as the grids more easily fit in the individual caches. At 64 processors, the data set size is too small to map properly and both local misses and coherence misses rise, as we saw in Figure 6.31.

6.7 Synchronization

Synchronization mechanisms are typically built with user-level software routines that rely on hardware-supplied synchronization instructions. For smaller multiprocessors or low-contention situations, the key hardware capability is an uninterruptible instruction or instruction sequence capable of atomically retrieving and changing a value. Software synchronization mechanisms are then constructed using this capability. For example, we will see how very efficient spin locks can be built using a simple hardware synchronization instruction and the coherence mechanism. In larger-scale multiprocessors or high-contention situations, synchronization can become a performance bottleneck, because contention introduces additional delays and because latency is potentially greater in such a multiprocessor. We will see how contention can arise in implementing some common user-level synchronization operations and examine more powerful hardware-supported synchronization primitives that can reduce contention as well as latency.

We begin by examining the basic hardware primitives, then construct several well-known synchronization routines with the primitives, and then turn to performance problems in larger multiprocessors and solutions for those problems.

Basic Hardware Primitives

The key ability we require to implement synchronization in a multiprocessor is a set of hardware primitives with the ability to atomically read and modify a memory location. Without such a capability, the cost of building basic synchronization primitives will be too high and will increase as the processor count increases. There are a number of alternative formulations of the basic hardware primitives, all of which provide the ability to atomically read and modify a location, together with some way to tell if the read and write were performed atomically. These hardware primitives are the basic building blocks that are used to build a wide variety of user-level synchronization operations, including things such as locks and barriers. In general, architects do not expect users to employ the basic hardware primitives, but instead expect that the primitives will be used by system programmers to build a synchronization library, a process that is often complex and tricky. Let's start with one such hardware primitive and show how it can be used to build some basic synchronization operations.

One typical operation for building synchronization operations is the *atomic exchan*ge, which interchanges a value in a register for a value in memory. To see how to use this to build a basic synchronization operation, assume that we want to build a simple lock where the value 0 is used to indicate that the lock is free and a 1 is used to indicate that the lock is unavailable. A processor tries to set the lock by doing an exchange of 1, which is in a register, with the memory address corresponding to the lock.The value returned from the exchange instruction is 1 if some other processor had already claimed access and 0 otherwise. In the latter

case, the value is also changed to be 1, preventing any competing exchange from also retrieving a 0.

For example, consider two processors that each try to do the exchange simultaneously: This race is broken since exactly one of the processors will perform the exchange first, returning 0, and the second processor will return 1 when it does the exchange. The key to using the exchange (or swap) primitive to implement synchronization is that the operation is atomic: the exchange is indivisible and two simultaneous exchanges will be ordered by the write serialization mechanisms. It is impossible for two processors trying to set the synchronization variable in this manner to both think they have simultaneously set the variable.

There are a number of other atomic primitives that can be used to implement synchronization. They all have the key property that they read and update a memory value in such a manner that we can tell whether or not the two operations executed atomically. One operation, present in many older multiprocessors, is *test-and-set*, which tests a value and sets it if the value passes the test. For example, we could define an operation that tested for 0 and set the value to 1, which can be used in a fashion similar to how we used atomic exchange. Another atomic synchronization primitive is *fetch-and-increment:* it returns the value of a memory location and atomically increments it. By using the value 0 to indicate that the synchronization variable is unclaimed, we can use fetch-and-increment, just as we used exchange. There are other uses of operations like fetch-and-increment, which we will see shortly.

A slightly different approach to providing this atomic read-and-update operation has been used in some recent multiprocessors. Implementing a single atomic memory operation introduces some challenges, since it requires both a memory read and a write in a single, uninterruptible instruction. This requirement complicates the implementation of coherence, since the hardware cannot allow any other operations between the read and the write, and yet must not deadlock.

An alternative is to have a pair of instructions where the second instruction returns a value from which it can be deduced whether the pair of instructions was executed as if the instructions were atomic. The pair of instructions is effectively atomic if it appears as if all other operations executed by any processor occurred before or after the pair. Thus, when an instruction pair is effectively atomic, no other processor can change the value between the instruction pair.

The pair of instructions includes a special load called a *load linked* or *load locked* and a special store called a *store conditional*. These instructions are used in sequence: If the contents of the memory location specified by the load linked are changed before the store conditional to the same address occurs, then the store conditional fails. If the processor does a context switch between the two instructions, then the store conditional also fails. The store conditional is defined to return a value indicating whether or not the store was successful. Since the load linked returns the initial value and the store conditional returns 1 if it succeeds and 0 otherwise, the following sequence implements an atomic exchange on the memory location specified by the contents of R1:

```
try:    MOV     R3,R4,R0        ;mov exchange value
        LL      R2,0(R1)        ;load linked
        SC      R3,0(R1)        ;store conditional
        BEQZ    R3,try          ;branch store fails
        MOV     R4,R2           ;put load value in R4
```

At the end of this sequence the contents of R4 and the memory location specified by R1 have been atomically exchanged (ignoring any effect from delayed branches). Any time a processor intervenes and modifies the value in memory between the LL and SC instructions, the SC returns 0 in R3, causing the code sequence to try again.

An advantage of the load linked/store conditional mechanism is that it can be used to build other synchronization primitives. For example, here is an atomic fetch-and-increment:

```
try:    LL      R2,0(R1)        ;load linked
        DADDUI  R3,R2,#1        ;increment
        SC      R3,0(R1)        ;store conditional
        BEQZ    R3,try          ;branch store fails
```

These instructions are typically implemented by keeping track of the address specified in the LL instruction in a register, often called the *link register.* If an interrupt occurs, or if the cache block matching the address in the link register is invalidated (for example, by another SC), the link register is cleared. The SC instruction simply checks that its address matches that in the link register; if so, the SC succeeds; otherwise, it fails. Since the store conditional will fail after either another attempted store to the load linked address or any exception, care must be taken in choosing what instructions are inserted between the two instructions. In particular, only register-register instructions can safely be permitted; otherwise, it is possible to create deadlock situations where the processor can never complete the SC. In addition, the number of instructions between the load linked and the store conditional should be small to minimize the probability that either an unrelated event or a competing processor causes the store conditional to fail frequently.

Implementing Locks Using Coherence

Once we have an atomic operation, we can use the coherence mechanisms of a multiprocessor to implement *spin locks:* locks that a processor continuously tries to acquire, spinning around a loop until it suceeds. Spin locks are used when a programmer expects the lock to be held for a very short amount of time and when

she wants the process of locking to be low latency when the lock is available. Because spin locks tie up the processor, waiting in a loop for the lock to become free, they are inappropriate in some circumstances.

The simplest implementation, which we would use if there were no cache coherence, would keep the lock variables in memory. A processor could continually try to acquire the lock using an atomic operation, say exchange, and test whether the exchange returned the lock as free. To release the lock, the processor simply stores the value 0 to the lock. Here is the code sequence to lock a spin lock whose address is in R1 using an atomic exchange:

```
          DADDUI R2,R0,#1
lockit:   EXCH   R2,0(R1)    ;atomic exchange
          BNEZ   R2,lockit   ;already locked?
```

If our multiprocessor supports cache coherence, we can cache the locks using the coherence mechanism to maintain the lock value coherently. Caching locks has two advantages. First, it allows an implementation where the process of "spinning" (trying to test and acquire the lock in a tight loop) could be done on a local cached copy rather than requiring a global memory access on each attempt to acquire the lock. The second advantage comes from the observation that there is often locality in lock accesses: that is, the processor that used the lock last will use it again in the near future. In such cases, the lock value may reside in the cache of that processor, greatly reducing the time to acquire the lock.

Obtaining the first advantage—being able to spin on a local cached copy rather than generating a memory request for each attempt to acquire the lock—requires a change in our simple spin procedure. Each attempt to exchange in the loop directly above requires a write operation. If multiple processors are attempting to get the lock, each will generate the write. Most of these writes will lead to write misses, since each processor is trying to obtain the lock variable in an exclusive state.

Thus we should modify our spin-lock procedure so that it spins by doing reads on a local copy of the lock until it successfully sees that the lock is available. Then it attempts to acquire the lock by doing a swap operation. A processor first reads the lock variable to test its state. A processor keeps reading and testing until the value of the read indicates that the lock is unlocked. The processor then races against all other processes that were similarly "spin waiting" to see who can lock the variable first. All processes use a swap instruction that reads the old value and stores a 1 into the lock variable. The single winner will see the 0, and the losers will see a 1 that was placed there by the winner. (The losers will continue to set the variable to the locked value, but that doesn't matter.) The winning processor executes the code after the lock and, when finished, stores a 0 into the lock variable to release the lock, which starts the race all over again. Here is the code to perform this spin lock (remember that 0 is unlocked and 1 is locked):

```
lockit: LD      R2,0(R1)     ;load of lock
        BNEZ    R2,lockit    ;not available-spin
        DADDUI  R2,R0,#1     ;load locked value
        EXCH    R2,0(R1)     ;swap
        BNEZ    R2,lockit    ;branch if lock wasn't 0
```

Let's examine how this "spin-lock" scheme uses the cache-coherence mechanisms. Figure 6.37 shows the processor and bus or directory operations for multiple processes trying to lock a variable using an atomic swap. Once the processor with the lock stores a 0 into the lock, all other caches are invalidated and must fetch the new value to update their copy of the lock. One such cache gets the copy of the unlocked value (0) first and performs the swap. When the cache miss of other processors is satisfied, they find that the variable is already locked, so they must return to testing and spinning.

Step	Processor P0	Processor P1	Processor P2	Coherence state of lock	Bus/directory activity
1	Has lock	Spins, testing if lock = 0	Spins, testing if lock = 0	Shared	None
2	Set lock to 0	(Invalidate received)	(Invalidate received)	Exclusive	Write invalidate of lock variable from P0
3		Cache miss	Cache miss	Shared	Bus/directory services P2 cache miss; write back from P0
4		(Waits while bus/ directory busy)	Lock = 0	Shared	Cache miss for P2 satisfied
5		Lock = 0	Executes swap, gets cache miss	Shared	Cache miss for P1 satisfied
6		Executes swap, gets cache miss	Completes swap: returns 0 and sets Lock =1	Exclusive	Bus/directory services P2 cache miss; generates invalidate
7		Swap completes and returns 1	Enter critical section	Shared	Bus/directory services P1 cache miss; generates write back
8		Spins, testing if lock = 0			None

FIGURE 6.37 Cache-coherence steps and bus traffic for three processors, P0, P1, and P2. This figure assumes write-invalidate coherence. P0 starts with the lock (step 1). P0 exits and unlocks the lock (step 2). P1 and P2 race to see which reads the unlocked value during the swap (steps 3–5). P2 wins and enters the critical section (steps 6 and 7), while P1's attempt fails so it starts spin waiting (steps 7 and 8). In a real system, these events will take many more than eight clock ticks, since acquiring the bus and replying to misses takes much longer.

This example shows another advantage of the load-linked/store-conditional primitives: the read and write operation are explicitly separated. The load linked need not cause any bus traffic. This fact allows the following simple code sequence, which has the same characteristics as the optimized version using exchange (R1 has the address of the lock):

```
lockit:  LL     R2,0(R1)      ;load linked
         BNEZ   R2,lockit     ;not available-spin
         DADDUI R2,R0,#1      ;locked value
         SC     R2,0(R1)      ;store
         BEQZ   R2,lockit     ;branch if store fails
```

The first branch forms the spinning loop; the second branch resolves races when two processors see the lock available simultaneously.

Although our spin lock scheme is simple and compelling, it has difficulty scaling up to handle many processors because of the communication traffic generated when the lock is released. The next section discusses these problems in more detail, as well as techniques to overcome these problems in larger multiprocessors.

Synchronization Performance Challenges

To understand why the simple spin-lock scheme of the previous section does not scale well, imagine a large multiprocessor with all processors contending for the same lock. The directory or bus acts as a point of serialization for all the processors, leading to lots of contention, as well as traffic. The following Example shows how bad things can be.

EXAMPLE Suppose there are 10 processors on a bus that each try to lock a variable simultaneously. Assume that each bus transaction (read miss or write miss) is 100 clock cycles long. You can ignore the time of the actual read or write of a lock held in the cache, as well as the time the lock is held (they won't matter much!). Determine the number of bus transactions required for all 10 processors to acquire the lock, assuming they are all spinning when the lock is released at time 0. About how long will it take to process the 10 requests? Assume that the bus is totally fair so that every pending request is serviced before a new request and that the processors are equally fast.

ANSWER Figure 6.38 shows the sequence of events from the time of the release to the time to the next release. Of course, the number of processors contending for the lock drops by one each time the lock is acquired, which reduces the average cost to 1550 cycles. Thus for 10 lock-unlock pairs it will take over 15,000 cycles for the processors to pass through the lock. Fur-

thermore, the average processor will spend half this time idle, simply trying to get the lock. The number of bus transactions involved is over 200!

Event	Duration
Read miss by all waiting processors to fetch lock (10×100)	1000
Write miss by releasing processor and invalidates	100
Read miss by all waiting processors (10×100)	1000
Write miss by all waiting processors, one successful lock (100), and invalidation of all lock copies (9×100)	1000
Total time for one processor to acquire and release lock	3100 clocks

FIGURE 6.38 The time to acquire and release a single lock when 10 processors contend for the lock, assuming each bus transaction takes 100 clock cycles. Because of fair bus arbitration, the releasing processor must wait for *all* other 9 processors to try to get the lock in vain!

n

The difficulty in this Example arises from contention for the lock and serialization of lock access, as well as the latency of the bus access. (The fairness property of the bus actually makes things worse, since it delays the processor that claims the lock from releasing it; unfortunately, for any bus arbitration scheme some worst-case scenario does exist.) The key advantages of spin locks, namely that they have low overhead in terms of bus or network cycles and offer good performance when locks are reused by the same processor, are both lost in this example. We will consider alternative implementations in the next section, but before we do that, let's consider the use of spin locks to implement another common high-level synchronization primitive.

Barrier Synchronization

One additional common synchronization operation in programs with parallel loops is a *barrier*. A barrier forces all processes to wait until all the processes reach the barrier and then releases all of the processes. A typical implementation of a barrier can be done with two spin locks: one used to protect a counter that tallies the processes arriving at the barrier and one used to hold the processes until the last process arrives at the barrier. To implement a barrier we usually use the ability to spin on a variable until it satisfies a test; we use the notation `spin(condition)` to indicate this. Figure 6.40 is a typical implementation, assuming that lock and unlock provide basic spin locks and `total` is the number of processes that must reach the barrier.

In practice, another complication makes barrier implementation slightly more complex. Frequently a barrier is used within a loop, so that processes released from the barrier would do some work and then reach the barrier again. Assume that one of the processes never actually leaves the barrier (it stays at the spin op-

```
lock (counterlock);/* ensure update atomic */
if (count==0) release=0;/*first=>reset release */
count = count +1;/* count arrivals */
unlock(counterlock);/* release lock */
if (count==total) {/* all arrived */
       count=0;/* reset counter */
       release=1;/* release processes */
}
else {/* more to come */

       spin (release==1);/* wait for arrivals */
}
```

FIGURE 6.39 Code for a simple barrier. The lock `counterlock` protects the counter so that it can be atomically incremented. The variable `count` keeps the tally of how many processes have reached the barrier. The variable `release` is used to hold the processes until the last one reaches the barrier.The operation `spin (release==1)` causes a process to wait until all processes reach the barrier.

eration), which could happen if the OS scheduled another process, for example. Now it is possible that one process races ahead and gets to the barrier again before the last process has left. The "fast" process then traps the remaining "slow" process in the barrier by resetting the flag `release`. Now all the processes will wait infinitely at the next instance of this barrier, because one process is trapped at the last instance, and the number of processes can never reach the value of total.

The important observation in this example is that the programmer did nothing wrong. Instead, the implementer of the barrier made some assumptions about forward progress that cannot be assumed. One obvious solution to this is to count the processes as they exit the barrier (just as we did on entry) and not to allow any process to reenter and reinitialize the barrier until all processes have left the prior instance of this barrier. This extra step would significantly increase the latency of the barrier and the contention, which as we will see shortly are already large. An alternative solution is a *sense-reversing barrier*, which makes use of a private per-process variable, `local_sense`, which is initialized to 1 for each process. Figure 6.40 shows the code for the sense-reversing barrier. This version of a barrier is

safely usable; as the next example shows, however, its performance can still be quite poor.

```
local_sense =! local_sense; /*toggle local_sense*/
lock (counterlock);/* ensure update atomic */
count=count+1;/* count arrivals */
unlock (counterlock);/* unlock */
if (count==total) {/* all arrived */
       count=0;/* reset counter */
       release=local_sense;/* release processes */
}
else {/* more to come */
       spin (release==local_sense);/*wait for signal*/
}
```

FIGURE 6.40 Code for a sense-reversing barrier. The key to making the barrier reusable is the use of an alternating pattern of values for the flag release, which controls the exit from the barrier. If a process races ahead to the next instance of this barrier while some other processes are still in the barrier, the fast process cannot trap the other processes, since it does not reset the value of `release` as it did in Figure 6.40.

EXAMPLE Suppose there are 10 processors on a bus that each try to execute a barrier simultaneously. Assume that each bus transaction is 100 clock cycles, as before. You can ignore the time of the actual read or write of a lock held in the cache as the time to execute other nonsynchronization operations in the barrier implementation. Determine the number of bus transactions required for all 10 processors to reach the barrier, be released from the barrier, and exit the barrier. Assume that the bus is totally fair, so that every pending request is serviced before a new request and that the processors are equally fast. Don't worry about counting the processors out of the barrier. How long will the entire process take?

ANSWER The following table shows the sequence of events for one processor to traverse the barrier, assuming that the first process to grab the bus does not have the lock.

Event	Duration in clocks for one processor	Duration in clocks for 10 processors
Time for each processor to grab lock, increment, release lock	3100	31,000
Time to execute release	100	100
Time for each processor to get the release flag	100	1000
Total	3300	31,100

Our barrier operation takes about as long as the 10-processor lock-unlock sequence we considered earlier. The total number of bus transactions is about 220. n

As we can see from these examples, synchronization performance can be a real bottleneck when there is substantial contention among multiple processes. When there is little contention and synchronization operations are infrequent, we are primarily concerned about the latency of a synchronization primitive—that is, how long it takes an individual process to complete a synchronization operation. Our basic spin-lock operation can do this in two bus cycles: one to initially read the lock and one to write it. We could improve this to a single bus cycle by a variety of methods. For example, we could simply spin on the swap operation. If the lock were almost always free, this could be better, but if the lock were not free, it would lead to lots of bus traffic, since each attempt to lock the variable would lead to a bus cycle. In practice, the latency of our spin lock is not quite as bad as we have seen in this example, since the write miss for a data item present in the cache is treated as an upgrade and will be cheaper than a true read miss.

The more serious problem in these examples is the serialization of each process's attempt to complete the synchronization. This serialization is a problem when there is contention, because it greatly increases the time to complete the synchronization operation. For example, if the time to complete all 10 lock and unlock operations depended only on the latency in the uncontended case, then it would take 1000 rather than 15,000 cycles to complete the synchronization operations. The barrier situation is as bad, and in some ways worse, since it is highly likely to incur contention. The use of a bus interconnect exacerbates these problems, but serialization could be just as serious in a directory-based multiprocessor, where the latency would be large. The next section presents some solutions that are useful when either the contention is high or the processor count is large.

Synchronization Mechanisms for Larger-Scale Multiprocessors

What we would like are synchronization mechanisms that have low latency in uncontended cases and that minimize serialization in the case where contention is significant. We begin by showing how software implementations can improve the performance of locks and barriers when contention is high; we then explore two basic hardware primitives that reduce serialization while keeping latency low.

Software Implementations

The major difficulty with our spin-lock implementation is the delay due to contention when many processes are spinning on the lock. One solution is to artificially delay processes when they fail to acquire the lock. The best performance is obtained by increasing the delay exponentially whenever the attempt to acquire

the lock fails. Figure 6.41 shows how a spin lock with *exponential back-off* is implemented. Exponential back-off is a common technique for reducing contention in shared resources, including access to shared networks and buses (see section 7.7). This implementation still attempts to preserve low latency when contention is small by not delaying the initial spin loop. The result is that if many processes are waiting, the back-off does not affect the processes on their first attempt to acquire the lock. We could also delay that process, but the result would be poorer performance when the lock was in use by only two processes and the first one happened to find it locked.

```
         ADDUI  R3,R0,#1     ;R3 = initial delay
lockit:  LL     R2,0(R1)     ;load linked
         BNEZ   R2,lockit    ;not available-spin
         DADDUI R2,R2,#1     ;get locked value
         SC     R2,0(R1)     ;store conditional
         BNEZ   R2,gotit     ;branch if store succeeds
         DSLL   R3,R3,#1     ;increase delay by factor of 2
         PAUSE  R3           ;delays by value in R3
         J      lockit
gotit:   use data protected by lock
```

FIGURE 6.41 A spin lock with exponential back-off. When the store conditional fails, the process delays itself by the value in R3. The delay can be implemented by decrementing R3 until it reaches 0. The exact timing of the delay is multiprocessor dependent, although it should start with a value that is approximately the time to perform the critical section and release the lock. The statement `pause R3` should cause a delay of R3 of these time units. The value in R3 is increased by a factor of 2 every time the store conditional fails, which causes the process to wait twice as long before trying to acquire the lock again. The small variations in the rate at which competing processors execute instructions are usually sufficient to ensure that processes will not continually collide. If the natural perturbation in execution time was insufficient, R3 could be initialized with a small random value, increasing the variance in the successive delays an reducing the probability of successive collisions.

Another technique for implementing locks is to use queuing locks. Queuing locks work by constructing a queue of waiting processors; whenever a processor frees up the lock, it causes the next processor in the queue to attempt access. This eliminates contention for a lock when it is freed. We show how queuing locks operate in the next section using a hardware implementation, but software implementations using arrays can achieve most of the same benefits (see Exercise 6.25). Before we look at hardware primitives, let's look at a better mechanism for barriers.

Our barrier implementation suffers from contention both during the *gather* stage, when we must atomically update the count, and at the *release* stage, when

all the processes must read the release flag. The former is more serious because it requires exclusive access to the synchronization variable and thus creates much more serialization; in comparison, the latter generates only read contention. We can reduce the contention by using a *combining tree,* a structure where multiple requests are locally combined in tree fashion. The same combining tree can be used to implement the release process, reducing the contention there; we leave the last step for the Exercises.

Our combining tree barrier uses a predetermined *n*-ary tree structure. We use the variable *k* to stand for the fan-in; in practice $k = 4$ seems to work well. When the *k*th process arrives at a node in the tree, we signal the next level in the tree. When a process arrives at the root, we release all waiting processes. As in our earlier example, we use a sense-reversing technique. A tree-based barrier, as shown in Figure 6.42, uses a tree to combine the processes and a single signal to release the barrier, Exercises 6.23 and 6.24 ask you to analyze the time for the combining barrier versus the noncombining version. Some MPPs (e.g., the T3D and CM-5) have also included hardware support for barriers, but more recent machines have relied on software libraries for this support.

Hardware Primitives

In this section we look at two hardware synchronization primitives. The first primitive deals with locks, while the second is useful for barriers and a number of other user-level operations that require counting or supplying distinct indices. In both cases we can create a hardware primitive where latency is essentially identical to our earlier version, but with much less serialization, leading to better scaling when there is contention.

The major problem with our original lock implementation is that it introduces a large amount of unneeded contention. For example, when the lock is released all processors generate both a read and a write miss, although at most one processor can successfully get the lock in the unlocked state. This sequence happens on each of the 20 lock/unlock sequences, as we saw in the example on page 710.

We can improve this situation by explicitly handing the lock from one waiting processor to the next. Rather than simply allowing all processors to compete every time the lock is released, we keep a list of the waiting processors and hand the lock to one explicitly, when its turn comes. This sort of mechanism has been called a *queuing lock.* Queuing locks can be implemented either in hardware, which we describe here, or in software using an array to keep track of the waiting processes. The basic concepts are the same in either case. Our hardware implementation assumes a directory-based multiprocessor where the individual processor caches are addressable. In a bus-based multiprocessor, a software implementation would be more appropriate and would have each processor using

```
struct node{/* a node in the combining tree */
       int counterlock; /* lock for this node */
       int count; /* counter for this node */
       int parent; /* parent in the tree = 0..P-1cep except for root
};
struct node tree [0..P-1]; /* the tree of nodes */
int local_sense; /* private per processor */
int release; /* global release flag */

/* function to implement barrier */
barrier (int mynode) {
       lock (tree[mynode].counterlock); /* protect count */
       tree[mynode].count=tree[mynode].count+1;
              /* increment count */
       unlock (tree[mynode].counterlock); /* unlock */
       if (tree[mynode].count==k) {/* all arrived at mynode */
              if (tree[mynode].parent >=0) {
                     barrier(tree[mynode].parent);
              } else{
                     release = local_sense;
              };
              tree[mynode].count = 0; /* reset for the next time */
       } else{
              spin (release==local_sense); /* wait */
       };
};
/* code executed by a processor to join barrier */
local_sense =! local_sense;
barrier (mynode);
```

FIGURE 6.42 An implementation of a tree-based barrier reduces contention considerably. The tree is assumed to be prebuilt statically using the nodes in the array tree. Each node in the tree combines k processes and provides a separate counter and lock, so that at most k processes contend at each node. When the kth process reaches a node in the tree it goes up to the parent, incrementing the count at the parent. When the count in the parent node reaches k, the release flag is set. The count in each node is reset by the last process to arrive. Sense-reversing is used to avoid races as in the simple barrier.

a different address for the lock, permitting the explicit transfer of the lock from one process to another.

How does a queuing lock work? On the first miss to the lock variable, the miss is sent to a synchronization controller, which may be integrated with the memory controller (in a bus-based system) or with the directory controller. If the lock is free, it is simply returned to the processor. If the lock is unavailable, the controller creates a record of the node's request (such as a bit in a vector) and sends the

processor back a locked value for the variable, which the processor then spins on. When the lock is freed, the controller selects a processor to go ahead from the list of waiting processors. It can then either update the lock variable in the selected processor's cache or invalidate the copy, causing the processor to miss and fetch an available copy of the lock.

EXAMPLE How many bus transaction and how long does it take to have 10 processors lock and unlock the variable using a queuing lock that updates the lock on a miss? Make the other assumptions about the system the same as those in the earlier example on page 710.

ANSWER Each processor misses once on the lock initially and once to free the lock, so it takes only 20 bus cycles. The first 10 initial misses take 1000 cycles, followed by a 100-cycle delay for each of the 10 releases. This sequence yields a total of 2100 cycles—significantly better than the case with conventional coherence-based spin locks. n

There are a couple of key insights in implementing such a queuing lock capability. First, we need to be able to distinguish the initial access to the lock, so we can perform the queuing operation, and also the lock release, so we can provide the lock to another processor. The queue of waiting processes can be implemented by a variety of mechanisms. In a directory-based multiprocessor, this queue is akin to the sharing set, and similar hardware can be used to implement the directory and queuing lock operations. One complication is that the hardware must be prepared to reclaim such locks, since the process that requested the lock may have been context-switched and may not even be scheduled again on the same processor.

Queuing locks can be used to improve the performance of our barrier operation (see Exercise 6.17). Alternatively, we can introduce a primitive that reduces the amount of time needed to increment the barrier count, thus reducing the serialization at this bottleneck, which should yield comparable performance to using queuing locks. One primitive that has been introduced for this and for building other synchronization operations is *fetch-and-increment*, which atomically fetches a variable and increments its value. The returned value can be either the incremented value or the fetched value. Using fetch-and-increment we can dramatically improve our barrier implementation, compared to the simple code-sensing barrier.

EXAMPLE Write the code for the barrier using fetch-and-increment. Making the same assumptions as in our earlier example and also assuming that a

fetch-and-increment operation takes 100 clock cycles, determine the time for 10 processors to traverse the barrier. How many bus cycles are required?

ANSWER Figure 6.40 shows the code for the barrier. This implementation requires 10 fetch-and-increment operations and 10 cache misses for the release operation for a total time of 2000 cycles and 20 bus/interconnect operations versus an earlier implementation that took over 15 times longer and 10 times more bus operations to complete the barrier. Of course, fetch-and-increment can also be used in implementing the combining tree barrier, reducing the serialization at each node in the tree.

```
local_sense =! local_sense; /*toggle local_sense*/
fetch_and_increment(count);/* atomic update*/
if (count==total) {/* all arrived */
       count=0;/* reset counter */
       release=local_sense;/* release processes */
}
else {/* more to come */
       spin (release==local_sense);/*wait for signal*/
}
```

FIGURE 6.43 Code for a sense-reversing barrier using fetch-and-increment to do the counting.

n

As we have seen, synchronization problems can become quite acute in larger-scale multiprocessors. When the challenges posed by synchronization are combined with the challenges posed by long memory latency and potential load imbalance in computations, we can see why getting efficient usage of large-scale parallel processors is very challenging.

6.8 Models of Memory Consistency: An Introduction

Cache coherence ensures that multiple processors see a consistent view of memory. It does not answer the question of *how* consistent the view of memory must be. By ":how consistent" we mean, when must a processor see a value that has been updated by another processor? Since processors communicate through

shared variables (used both for data values and for synchronization), the question boils down to this: In what order must a processor observe the data writes of another processor? Since the only way to "observe the writes of another processor" is through reads, the question becomes, what properties must be enforced among reads and writes to different locations by different processors?

Although the question of how consistent memory be seems simple, it is remarkably complicated, as we can see with a simple example. Here are two code segments from processes P1 and P2, shown side by side:

```
P1:     A = 0;                  P2:     B = 0;
          .....                           .....
        A = 1;                          B = 1;
L1:     if (B == 0) ...         L2:     if (A == 0)...
```

Assume that the processes are running on different processors, and that locations `A` and `B` are originally cached by both processors with the initial value of 0. If writes always take immediate effect and are immediately seen by other processors, it will be impossible for *both* if-statements (labeled `L1` and `L2`) to evaluate their conditions as true, since reaching the if-statement means that either `A` or `B` must have been assigned the value 1. But suppose the write invalidate is delayed, and the processor is allowed to continue during this delay; then it is possible that both P1 and P2 have not seen the invalidations for `B` and `A` (respectively) *before* they attempt to read the values. The question is, should this behavior be allowed, and if so, under what conditions?

The most straightforward model for memory consistency is called *sequential consistency.* Sequential consistency requires that the result of any execution be the same as if the memory accesses executed by each processor were kept in order and the accesses among different processors were arbitrarily interleaved. Sequential consistency eliminates the possibility of some nonobvious execution in the previous example, because the assignments must be completed before the if statements are initiated.

The simplest way to implement sequential consistency is to require a processor to delay the completion of any memory access until all the invalidations caused by that access are completed. Of course, it is equally effective to delay the next memory access until the previous one is completed. Remember that memory consistency involves operations among different variables: the two accesses that must be ordered are actually to different memory locations. In our example, we must delay the read of `A` or `B` (`A==0` or `B==0`) until the previous write has completed (`B=1` or `A=1`). Under sequential consistency, we cannot, for example, simply place the write in a write buffer and continue with the read. Although sequential consistency presents a simple programming paradigm, it reduces potential

performance, especially in a multiprocessor with a large number of processors or long interconnect delays, as we can see in the following Example.

EXAMPLE Suppose we have a processor where a write miss takes 40 cycles to establish ownership, 10 cycles to issue each invalidate after ownership is established, and 50 cycles for an invalidate to complete and be acknowledged once it is issued. Assuming that four other processors share a cache block, how long does a write miss stall the writing processor if the processor is sequentially consistent? Assume that the invalidates must be explicitly acknowledged before the directory controller knows they are completed. Suppose we could continue executing after obtaining ownership for the write miss without waiting for the invalidates; how long would the write take?

ANSWER When we wait for invalidates, each write takes the sum of the ownership time plus the time to complete the invalidates. Since the invalidates can overlap, we need only worry about the last one, which starts 10 + 10 + 10 + 10 = 40 cycles after ownership is established. Hence the total time for the write is 40 + 40 + 50 = 130 cycles. In comparison, the ownership time is only 40 cycles. With appropriate write-buffer implementations it is even possible to continue before ownership is established. n

To provide better performance, researchers and architects have explored two different routes. First, they developed ambitious implementations that preserve sequential consistency but use latency hiding techniques to reduce the penalty; we discuss these in the section on cross-cutting issues (see page 731). Second, they developed less restrictive memory consistency models that allow for faster hardware. Such models can affect how the programmer sees the multiprocessor, so before we discuss these less restrictive models, let's look at what the programmer expects.

The Programmer's View

Although the sequential consistency model has a performance disadvantage, from the viewpoint of the programmer it has the advantage of simplicity. The challenge is to develop a programming model that is simple to explain and yet allows a high performance implementation.

One such programming model that allows us to have a more efficient implementation is to assume that programs are *synchronized*. A program is synchronized if all access to shared data is ordered by synchronization operations. A data reference is ordered by a synchronization operation if, in every possible execution, a write of a variable by one processor and an access (either a read or a write)

of that variable by another processor are separated by a pair of synchronization operations, one executed after the write by the writing processor and one executed before the access by the second processor. Cases where variables may be updated without ordering by synchronization are called *data races*, because the execution outcome depends on the relative speed of the processors, and like races in hardware design, the outcome is unpredictable, which leads to another name for synchronized programs: *data-race-free*.

As a simple example, consider a variable being read and updated by two different processors. Each processor surrounds the read and update with a lock and an unlock, both to ensure mutual exclusion for the update and to ensure that the read is consistent. Clearly, every write is now separated from a read by the other processor by a pair of synchronization operations: one unlock (after the write) and one lock (before the read). Of course, if two processors are writing a variable with no intervening reads, then the writes must also be separated by synchronization operations.

It is a broadly accepted observation that most programs are synchronized. This observation is true primarily because if the accesses were unsynchronized, the behavior of the program would be quite difficult to determine because the speed of execution would determine which processor won a data race and thus affect the results of the program. Even with sequential consistency, reasoning about such programs is very difficult. Programmers could attempt to guarantee ordering by constructing their own synchronization mechanisms, but this is extremely tricky, can lead to buggy programs, and may not be supported architecturally, meaning that they may not work in future generations of the multiprocessor. Instead, almost all programmers will choose to use synchronization libraries that are correct and optimized for the multiprocessor and the type of synchronization. Finally, the use of standard synchronization primitives ensures that even if the architecture implements a more relaxed consistency model than sequential consistency, a synchronized program will behave as if the hardware implemented sequential consistency.

Relaxed Consistency Models: The Basics

The key idea in relaxed consistency models is to allow reads and writes to complete out of order, but to use synchronization operations to enforce ordering, so that a synchronized program behaves as if the processor were sequentially consistent. There are a variety of relaxed models that are classified according to what orderings they relax. The three major sets of orderings that are relaxed are:

1. The W→R ordering: which yields a model known as total store ordering or processor consistency. Because this ordering retains ordering among writes, many programs that operate under sequential consistency operate under this model, without additional synchronization.

2. The W→W ordering: which yields a model known as partial store order.
3. The R→W and R→R orderings: which yields a variety of models including weak ordering, the Alpha consistency model, the PowerPC consistency model, and release consistency depending on the details of the ordering restrictions and how synchronization operations enforce ordering.

By relaxing these orderings, the processor can possibly obtain significant performance advantages. There are, however, many complexities in describing relaxed consistency models, including the advantages and complexities of relaxing different orders, defining precisely what it means for a write to complete, and deciding when processors can see values that the processor itself has written. These complexities, as well as an assessment of the performance of relaxed model and a discussion of the implementation issues, are described in more detail in Appendix F.

Final Remarks on Consistency Models

At the present time, many multiprocessors being built support some sort of relaxed consistency model, varying from processor consistency to release consistency. Since synchronization is highly multiprocessor specific and error prone, the expectation is that most programmers will use standard synchronization libraries and will write synchronized programs, making the choice of a weak consistency model invisible to the programmer and yielding higher performance.

An alternative viewpoint, which we discuss more extensively in the next section (specifically on page 731), argues that with speculation much of the performance advantage of relaxed consistency models can be obtained with sequential or processor consistency.

A key part of this argument in favor of relaxed consistency revolves the role of the compiler and its ability to optimize memory access to potentially shared variables. This topic is also discussed on page 731.

6.9 Multithreading: Exploiting Thread-Level Parallelism within a Processor

Multithreading allows multiple threads to share the functional units of a single processor in an overlapping fashion. To permit this sharing, the processor must duplicate the independent state of each thread. For example, a separate copy of the register file, a separate PC, and a separate page table are required for each thread. The memory itself can be shared through the virtual memory mechanisms, which already support multiprogramming. In addition, the hardware must

support the ability to change to a different thread relatively quickly; in particular, a thread switch should be much more efficient than a process switch, which typically requires hundreds to thousands of processor cycles.

There are two main approaches to multithreading. *Fine-grained multithreading* switches between threads on each instruction, causing the execution of multiples threads to be interleaved. This interleaving is often done in a round-robin fashion, skipping any threads that are stalled at that time. To make fine-grained multithreading practical, the CPU must be able to switch threads on every clock cycle. One key advantage of fine-grained multithreading is that it can hide the throughput losses that arise from both short and long stalls, since instructions from other threads can be executed when one thread stalls. The primary disadvantage of fine-grained multithreading is that it slows down the execution of the individual threads, since a thread that is ready to execute without stalls will be delayed by instructions from other threads.

Coarse-grained multithreading was invented as an alternative to fine-grained multithreading. Coarse-grained multithreading switches threads only on costly stalls, such as level two cache misses. This change relieves the need to have thread-switching be essentially free and is much less likely to slow the processor down, since instructions from other threads will only be issued, when a thread encounters a costly stall. Coarse-grained multithreading suffers, however, from a major drawback: it is limited in its ability to overcome throughput losses, especially from shorter stalls. This limitation arises from the pipeline start-up costs of coarse-grain multithreading. Because a CPU with coarse-grained multithreading issues instructions from a single thread, when a stall occurs, the pipeline must be emptied or frozen. The new thread that begins executing after the stall must fill the pipeline before instructions will be able to complete. Because of this start-up overhead, coarse-grained multithreading is much more useful for reducing the penalty of high cost stalls, where pipeline refill is negligible compared to the stall time.

The next section explores a variation on fine-grained multithreading that enables a superscalar processor to exploit ILP and multithreading in an integrated and efficient fashion. Section 6.12 examines a commercial processor using coarse-grained multithreading.

Simultaneous Multithreading: Converting Thread-Level Parallelism into Instruction-Level Parallelism

Simultaneous multithreading (SMT) is a variation on multithreading that uses the resources of a multiple-issue, dynamically-scheduled processor to exploit TLP at the same time it exploits ILP. The key insight that motivates SMT is that modern multiple-issue processors often have more functional unit parallelism available than a single thread can effectively use. Furthermore, with register renaming and dynamic scheduling, multiple instructions from independent threads can be is-

sued without regard to the dependences among them; the resolution of the dependences can be handled by the dynamic scheduling capability.

Figure 6.44 conceptually illustrates the differences in a processor's ability to exploit the resources of a superscalar for the following processor configurations:

- a superscalar with no multithreading support,
- a superscalar with coarse-grained multithreading,
- a superscalar with fine-grained multithreading, and
- a superscalar with simultaneous multithreading.

In the superscalar without multithreading support, the use of issue slots is limited by a lack of ILP, a topic we discussed extensively in Chapter 3. In addition, a major stall, such as an instruction cache miss, can leave the entire processor idle.

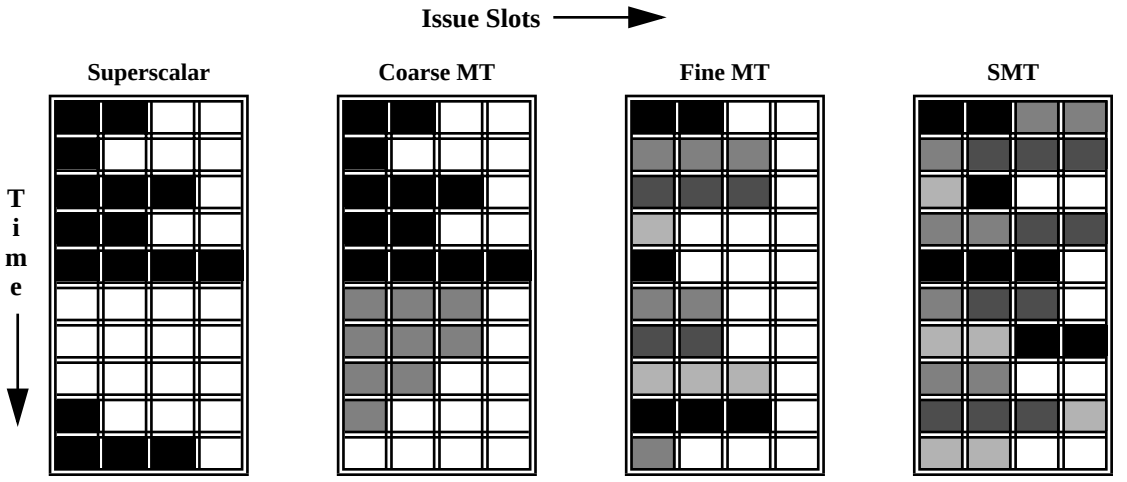


FIGURE 6.44 This illustration shows how these four different approaches use the issue slots of a superscalar processor. The horizontal dimension represents the instruction issue capability in each clock cycle. The vertical dimension represents a sequence of clock cycles. An empty (white) box indicates that the corresponding issue slot is unused in that clock cycle. The shades of grey and black correspond to four different threads in the multithreading processors. Black is also used to indicate the occupied issue slots in the case of the superscalar without multithreading support.

In the coarse-grained multithreaded superscalar, the long stalls are partially hidden by switching to another thread that uses the resources of the processor. Although this reduces the number of completely idle clock cycles, within each clock cycle, the ILP limitations still lead to idle cycles. Furthermore, in a coarse-grained multithreaded processor, since thread switching only occurs when there is a stall and the new thread has a start-up period, there are likely to be some fully idle cycles remaining.

In the fine-grained case, the interleaving of threads eliminates fully empty slots. Because only one thread issues instructions in a given clock cycle, however, ILP limitations still lead to a significant number of idle slots within individual clock cycles.

In the SMT case, thread-level parallelism (TLP) and instruction-level parallelism (ILP) are exploited simultaneously; with multiple threads using the issue slots in a single clock cycle. Ideally, the issue slot usage is limited by imbalances in the resource needs and resource availability over multiple threads. In practice, other factors–including how many active threads are considered, finite limitations on buffers, the ability to fetch enough instructions from multiple threads, and practical limitations of what instruction combinations can issue from one thread and from multiple threads–can also restrict how many slots are used. Although Figure 6.44 greatly simplifies the real operation of these processors it does illustrate the potential performance advantages of multithreading in general and SMT in particular.

As mentioned above, simultaneous multithreading uses the insight that a dynamically scheduled processor already has many of the hardware mechanisms needed to support the integrated exploitation of TLP through multithreading. In particular, dynamically scheduled superscalars have a large set of virtual registers that can be used to hold the register sets of independent threads (assuming separate renaming tables are kept for each thread). Because register renaming provides unique register identifiers, instructions from multiple threads can be mixed in the datapath without confusing sources and destinations across the threads. This observation leads to the insight that multithreading can be built on top of an out-of-order processor by adding a per thread renaming table, keeping separate PCs, and providing the capability for instructions from multiple threads to commit. There are complications in handling instruction commit, since we would like instructions from independent threads to be able to commit independently. The independent commitment of instructions from separate threads can be supported by logically keeping a separate reorder buffer for each thread.

Design Challenges in SMT processors

Because a dynamically scheduled superscalar processor is likely to have a deep pipeline, SMT will be unlikely to gain much in performance if it were coarse-grained. Since SMT will likely make sene only in a fine-grained implementation, we must worry about the impact of fine-grained scheduling on single thread performance. This effect can be minimized by having a preferred thread, which still permits multithreading to preserve some of its performance advantage with a smaller compromise in single thread performance. At first glance, it might appear that a preferred thread approach sacrifices neither throughput nor single-thread performance. Unfortunately, with a preferred thread, the processor is likely to sacrifice some throughput, when the preferred thread encounters a stall. The rea-

son is that the pipeline is less likely to have a mix of instructions from several threads, resulting in greater probability that either empty slots or a stall will occur. Throughput is maximized by having a sufficient number of independent threads to hide all stalls in any combination of threads.

Unfortunately, mixing many threads will inevitably compromise the execution time of individual threads. Similar problems exist in instruction fetch. To maximize single thread performance, we should fetch as far ahead as possible in that single thread and always have the fetch unit free when a branch is mispredicted and a miss occurs in the prefetch buffer. Unfortunately, this limits the number of instructions available for scheduling from other threads, reducing throughput. All multithreaded processor must seek to balance this tradeoff.

In practice, the problems of dividing resources and balancing single-thread and multiple-thread performance turn out not to be as challenging as they sound, at least for current superscalar back-ends. In particular, for current machines that issue four to eight instructions per cycle, it probably suffices to have a small number of active threads, and an even smaller number of "preferred" threads. Whenever possible, the processor acts on behalf of a preferred thread. This starts with prefetching instructions: whenever the prefetch buffers for the preferred threads are not full, instructions are fetched for those threads. Only when the preferred thread buffers are full is the instruction unit directed to prefetch for other threads. Note that having two preferred threads means that we are simultaneously prefetching for two instruction streams and this adds complexity to the instruction fetch unit and the instruction cache. Similarly, the instruction issue unit can direct its attention to the preferred threads, considering other threads only if the preferred threads are stalled and cannot issue. In practice, having four to eight threads and two to four preferred threads is likely to completely utilize the capability of a superscalar back-end that is roughly double the capability of those available in 2001.

There are a variety of other design challenges for an SMT processor, including:

- dealing with a larger register file needed to hold multiple contexts,
- maintaining low overhead on the clock cycle, particularly in critical steps such as instruction issue, where more candidate instructions need to be considered, and in instruction completion, where choosing what instructions to commit may be challenging, and
- ensuring that the cache conflicts generated by the simultaneous execution of multiple threads do not cause significant performance degradation.

In viewing these problems, two observation are important. In many cases, the potential performance overhead due to multithreading is small, and simple choices work well enough. Second, the efficiency of current superscalars is low enough that there is room for significant improvement, even at the cost of some overhead.

SMT appears to be the most promising way to achieve that improvement in throughput.

Because SMT exploits thread-level parallelism on a multiple-issue superscalar, it is most likely to be included in high-end processors targeted at server markets. In addition, it is likely that there will be some mode to restrict the multithreading, so as to maximize the performance of a single thread.

Prior to deciding to abandon the Alpha architecture in mid 2001, Compaq had announced that the Alpha 21364 would have SMT capability when it became available in 2002 In July 2001, Intel announced that a future processor based on the Pentium 4 microarchitecture and targeted at the server market, most likely Pentium 4 Xenon, would support SMT, initially with two-thread implementation. Intel claims a 30% improvement in throughput for server applications with this new support.

6.10 Crosscutting Issues

Because multiprocessors redefine many system characteristics (e.g., performance assessment, memory latency, and the importance of scalability), they introduce interesting design problems that cut across the spectrum, affecting both hardware and software. In this section we give several examples including: measuring and reporting the performance of multiprocessors, enhancing latency tolerance in memory systems, and a method for using virtual memory support to implement shared memory.

Memory System Issues

As we have seen in this chapter, memory system issues are at the core of the design of shared-memory multiprocessors. Indeed, multiprocessing introduces many new memory system complications that do not exist in uniprocessors. In this section we look at two implementation issues that have a significant impact on the design and implementation of a memory system in a multiprocessor context.

Inclusion and Its Implementation

Many multiprocessors use multilevel cache hierarchies to reduce both the demand on the global interconnect and the latency of cache misses. If the cache also provides *multilevel inclusion*—every level of cache hierarchy is a subset of the level further away from the processor—then we can use the multilevel structure to reduce the contention between coherence traffic and processor traffic, as explained earlier. Thus most multiprocessors with multilevel caches enforce the

inclusion property. This restriction is also called the *subset property,* because each cache is a subset of the cache below it in the hierarchy.

At first glance, preserving the multilevel inclusion property seems trivial. Consider a two-level example: any miss in L1 either hits in L2 or generates a miss in L2, causing it to be brought into both L1 and L2. Likewise, any invalidate that hits in L2 must be sent to L1, where it will cause the block to be invalidated, if it exists.

The catch is what happens when the block size of L1 and L2 are different. Choosing different block sizes is quite reasonable, since L2 will be much larger and have a much longer latency component in its miss penalty, and thus will want to use a larger block size. What happens to our "automatic" enforcement of inclusion when the block sizes differ? A block in L2 represents multiple blocks in L1, and a miss in L2 causes the replacement of data that is equivalent to multiple L1 blocks. For example, if the block size of L2 is four times that of L1, then a miss in L2 will replace the equivalent of four L1 blocks. Let's consider a detailed example.

EXAMPLE Assume that L2 has a block size four times that of L1. Show how a miss for an address that causes a replacement in L1 and L2 can lead to violation of the inclusion property.

ANSWER Assume that L1 and L2 are direct mapped and that the block size of L1 is b bytes and the block size of L2 is $4b$ bytes. Suppose L1 contains two blocks with starting addresses x and $x + b$ and that $x \bmod 4b = 0$, meaning that x also is the starting address of a block in L2, then that single block in L2 contains the L1 blocks x, $x + b$, $x + 2b$, and $x + 3b$. Suppose the processor generates a reference to block y that maps to the block containing x in both caches and hence misses. Since L2 missed, it fetches $4b$ bytes and replaces the block containing x, $x + b$, $x + 2b$, and $x + 3b$, while L1 takes b bytes and replaces the block containing x. Since L1 still contains $x + b$, but L2 does not, the inclusion property no longer holds. ▫

To maintain inclusion with multiple block sizes, we must probe the higher levels of the hierarchy when a replacement is done at the lower level to ensure that any words replaced in the lower level are invalidated in the higher-level caches. Most systems chose this solution rather than the alternative of not relying on inclusion and snooping the higher-level caches. In the Exercises we explore inclusion further and show that similar problems exist if the associativity of the levels is different. Baer and Wang [1988] describe the advantages and challenges of inclusion in detail.

Nonblocking Caches and Latency Hiding

We saw the idea of nonblocking or lockup-free caches in Chapter 5, where the concept was used to reduce cache misses by overlapping them with execution and by pipelining misses. There are additional benefits in the multiprocessor case. The first is that the miss penalties are likely to be larger, meaning there is more latency to hide, and the opportunity for pipelining misses is also probably larger, since the memory and interconnect system can often handle multiple outstanding memory references also.

Second, a multiprocessor needs nonblocking caches to take advantage of weak consistency models. For example, to implement a model like processor consistency requires that writes be nonblocking with respect to reads so that a processor can continue either immediately, by buffering the write, or as soon as it establishes ownership of the block and updates the cache. Relaxed consistency models allow further reordering of misses, but nonblocking caches are needed to take full advantage of this flexibility.

Finally, nonblocking support is critical to implementing prefetching. Prefetching, which we also discussed in Chapter 5, is even more important in multiprocessors than in uniprocessors, again due to longer memory latencies. In Chapter 5 we described why it is important that prefetches not affect the semantics of the program, since this allows them to be inserted anywhere in the program without changing the results of the computation.

In a multiprocessor, maintaining the absence of any semantic impact from the use of prefetches requires that prefetched data be kept coherent. A prefetched value is kept coherent if, when the value is actually accessed by a load instruction, the most recently written value is returned, even if that value was written after the prefetch. This result is exactly the property that cache coherence gives us for other variables in memory. A prefetch that brings a data value closer, and guarantees that on the actual memory access to the data (a load of the prefetched value) the most recent value of the data item is obtained, is called *nonbinding*, since the data value is not bound to a local copy, which would be incoherent. By contrast, a prefetch that moves a data value into a general-purpose register is binding, since the register value is a new variable, as opposed to a cache block, which is a coherent copy of a variable. A nonbinding prefetch maintains the coherence properties of any other value in memory, while a binding prefetch appears more like a register load, since it removes the data from the coherent address space.

Why is nonbinding prefetch critical? Consider a simple but typical example: a data value written by one processor and used by another. In this case, the consumer would like to prefetch the value as early as possible; but suppose the producing process is delayed for some reason. Then the prefetch may fetch the old value of the data item. If the prefetch is nonbinding, the copy of the old data is invalidated when the value is written, maintaining coherence. If the prefetch is binding, however, then the old, incoherent value of the data is used by the prefetching process. Because of the long memory latencies, a prefetch may need to be placed a hundred or more instructions earlier than the data use, if we aim to

hide the entire latency. This requirement makes the nonbinding property vital to ensure coherent usage of the prefetch in multiprocessors.

Implementing prefetch requires the same sort of support that a lockup-free cache needs, since there are multiple outstanding memory accesses. This requirement causes several complications:

1. A local node will need to keep track of the multiple outstanding accesses, since the replies may return in a different order than they were sent. This accounting can be handled by adding tags to the requests, or by incorporating the address of the memory block in the reply.
2. Before issuing a request (either a normal fetch or a prefetch), the node must ensure that it has not already issued a request for the same block, since two write requests for the same block could lead to incorrect operation of the protocol. For example, if the node issues a write prefetch to a block, while it has a write miss or write prefetch outstanding, both our snooping protocol and directory protocol can fail to operate properly.
3. Our implementation of the directory and snooping controllers assumes that the processor stalls on a miss. Stalling allows the cache controller to simply wait for a reply when it has generated a request. With a nonblocking cache or with prefetching, a processor can generate additional requests while it is waiting for replies. This complicates the directory and snooping controllers; Appendix E shows how these issues can be addressed.

Compiler Optimization and the Consistency Model

Another reason for defining a model for memory consistency is to specify the range of legal compiler optimizations that can be performed on shared data. In explicitly parallel programs, unless the synchronization points are clearly defined and the programs are synchronized, the compiler could not interchange a read and a write of two different shared data items, because such transformations might affect the semantics of the program. This prevents even relatively simple optimizations, such as register allocation of shared data, because such a process usually interchanges reads and writes. In implicitly parallelized programs—for example, those written in High Performance FORTRAN (HPF)—programs must be synchronized and the synchronization points are known, so this issue does not arise.

Using Speculation to Hide Latency in Strict Consistency Models

As we saw in Chapters 4 and 5, speculation can be used to hide memory latency. It can also be used to hide latency arising from a strict consistency model, giving much of the benefit of a relaxed memory model. The key idea is for: the processor to use dynamic scheduling to reorder memory references, letting them possibly execute out-of-order. Executing the memory references out-of-order may generate violations of sequential consistency, which might affect the execution of the program. This possibility is avoided by using the delayed commit feature of a

speculative processor. Assume the coherency protocol is based on invalidation. If the processor receives an invalidation for a memory reference before the memory reference is committed, the processor uses speculation recovery to back-out the computation and restart with the memory reference whose address was invalidated.

If the reordering of memory requests by the processor yields an execution order that could result in an outcome that differs from what would have been seen under sequential consistency, the processor will redo the execution. The key to using this approach is that the processor need only guarantee that the result would be the same as if all access were completed in order, and it can achieve this by detecting when the results might differ. The approach is attractive because the speculative restart will rarely be triggered. It will only be triggered when there are unsynchronized access that actually cause a race. Gharachorloo, et. al. made this observation in a 1992 paper.

Hill in a 1998 paper advocates the combination of sequential or processor consistency together with speculative execution as the consistency model of choice. His argument has three parts. First, that an aggressive implementation of either sequential consistency or processor consistency will gain most of the advantage of a more relaxed model. Second, that such an implementation adds very little to the implementation cost of a speculative processor. Third, that such an approach allows the programmer to reason using the simpler programming models of either sequential or processor consistency.

The MIPS R10000 design team had this insight in the mid 1990s and used the R10000's out-of-order capability to support this type of aggressive implementation of sequential consistency. Hill's arguments are likely to motivate others to follow this approach.

One open question is how successful compiler technology will be in optimizing memory references to shared variables. The state of optimization technology and the fact that shared data is often accessed via pointers or array indexing has limited the use of such optimizations. If this technology became available and led to significant performance advantages, compiler writers would want to be able to take advantage of a more relaxed programming model.

Using Virtual Memory Support to Build Shared Memory

Suppose we wanted to support a shared address space among a group of workstations connected to a network. One approach is to use the virtual memory mechanism and operating system (OS) support to provide shared memory. This approach, which was first explored more than 10 years ago, has been called *distributed virtual memory (DVM)* or *shared virtual memory (SVM)*. The key observation that this idea builds on is that the virtual memory hardware has the ability to control access to portions of the address space for both reading and writing. By using the hardware to check and intercept accesses and the operating system to ensure coherence, we can create a coherent, shared address space across the distributed memory of multiple processors.

In SVM, pages become the units of coherence, rather than cache blocks. The OS can allow pages to be replicated in read-only fashion, using the virtual memory support to protect the pages from writing. When a process attempts to write such a page, it traps to the operating system. The operating system on that processor can then send messages to the OS on each node that shares the page, requesting that the page be invalidated. Just as in a directory system, each page has a home node, and the operating system running in that node is responsible for tracking who has copies of the page.

The mechanisms are quite similar to those at work in coherent shared memory. The key differences are that the unit of coherence is a page and that software is used to implement the coherence algorithms. It is exactly these two differences that lead to the major performance differences. A page is considerably bigger than a cache block, and the possibilities for poor usage of a page and for false sharing are very high. Such events can lead to much less stable performance and sometimes even lower performance than a uniprocessor. Because the coherence algorithms are implemented in software, they have much higher overhead.

The result of this combination is that shared virtual memory has become an acceptable substitute for loosely coupled message passing, since in both cases the frequency of communication must be low, and communication that is structured in larger blocks is favored. Distributed virtual memory is not currently competitive with schemes that have hardware-supported, coherent memory, such as the distributed shared-memory schemes we examined in section 6.5: Most programs written for coherent shared memory cannot be run efficiently on shared virtual memory without changes.

Several factors could change the attractiveness of shared virtual memory. Better implementation and small amounts of hardware support could reduce the overhead in the operating system. Compiler technology, as well as the use of smaller or multiple page sizes, could allow the system to reduce the disadvantages of coherence at a page-level granularity. The concept of software-supported

shared memory remains an active area of research, and such techniques may play an important role in connecting more loosely coupled multiprocessors, such as networks of workstations.

Performance Measurement of Parallel Processors

One of the most controversial issues in parallel processing has been how to measure the performance of parallel processors. Of course, the straightforward answer is to measure a benchmark as supplied and to examine wall-clock time. Measuring wall-clock time obviously makes sense; in a parallel processor, measuring CPU time can be misleading because the processors may be idle but unavailable for other uses.

Users and designers are often interested in knowing not just how well a multiprocessor performs with a certain fixed number of processors, but also how the performance scales as more processors are added. In many cases, it makes sense to scale the application or benchmark, since if the benchmark is unscaled, effects arising from limited parallelism and increases in communication can lead to results that are pessimistic when the expectation is that more processors will be used to solve larger problems. Thus, it is often useful to measure the speedup as processors are added both for a fixed-size problem and for a scaled version of the problem, providing an unscaled and a scaled version of the speedup curves. The choice of how to measure the uniprocessor algorithm is also important to avoid anomalous results, since using the parallel version of the benchmark may understate the uniprocessor performance and thus overstate the speedup, as discussed with an example in section 6.14.

Once we have decided to measure scaled speedup, the question is *how* to scale the application. Let's assume that we have determined that running a benchmark of size n on p processors makes sense. The question is how to scale the benchmark to run on $m \times p$ processors. There are two obvious ways to scale the problem: keeping the amount of memory used per processor constant; and keeping the total execution time, assuming perfect speedup, constant. The first method, called *memory-constrained scaling,* specifies running a problem of size $m \times n$ on $m \times p$ processors. The second method, called *time-constrained scaling,* requires that we know the relationship between the running time and the problem size, since the former is kept constant. For example, suppose the running time of the application with data size n on p processors is proportional to n^2/p. Then with time-constrained scaling, the problem to run is the problem whose ideal running time on $m \times p$ processors is still n^2/p. The problem with this ideal running time has size $\sqrt{m} \times n$.

EXAMPLE Suppose we have a problem whose execution time for a problem of size n is proportional to n^3. Suppose the actual running time on a 10-processor multiprocessor is 1 hour. Under the time-constrained and memory-constrained scaling models, find the size of the problem to run and the effec-

tive running time for a 100-processor multiprocessor.

ANSWER For the time-constrained problem, the ideal running time is the same, 1 hour, so the problem size is $\sqrt[3]{10} \times n$ or 2.15 times larger than the original. For memory-constrained scaling, the size of the problem is 10*n* and the ideal execution time is 10^3/10, or 100 hours! Since most users will be reluctant to run a problem on an order of magnitude more processors for 100 times longer, this size problem is probably unrealistic. n

In addition to the scaling methodology, there are questions as to how the program should be scaled when increasing the problem size affects the quality of the result. Often, we must change other application parameters to deal with this effect. As a simple example, consider the effect of time to convergence for solving a differential equation. This time typically increases as the problem size increases, since, for example, we often require more iterations for the larger problem. Thus when we increase the problem size, the total running time may scale faster than the basic algorithmic scaling would indicate.

For example, suppose that the number of iterations grows as the log of the problem size. Then for a problem whose algorithmic running time is linear in the size of the problem, the effective running time actually grows proportional to n log n. If we scaled from a problem of size m on 10 processors, purely algorithmic scaling would allow us to run a problem of size 10 m on 100 processors. Accounting for the increase in iterations means that a problem of size $k \times m$, where k log k = 10, will have the same running time on 100 processors. This problem size yields a scaling of 5.72 m, rather than 10 m.

In practice, scaling to deal with error requires a good understanding of the application and may involve other factors, such as error tolerances (for example, it affects the cell-opening criteria in Barnes-Hut). In turn, such effects often significantly affect the communication or parallelism properties of the application as well as the choice of problem size.

Scaled speedup is not the same as unscaled (or true) speedup; confusing the two has led to erroneous claims (e.g., see the fallacy on page 753). Scaled speedup has an important role, but only when the scaling methodology is sound and the results are clearly reported as using a scaled version of the application. Singh et. al.[1993] describe these issues in detail.

6.11 Putting It All Together: Sun's Wildfire Prototype

In Sections 6.3 and 6.5 we examined centralized memory architectures (also known as SMPs or symmetric multiprocessors) and distributed memory architectures (also known as DSMs or distributed shared memory multiprocessors). SMPs have the advantage of maintaining a single centralized memory with uni-

form access time, and although cache hit rates are crucial, memory placement is not. In comparison, DSMs have a nonuniform memory architecture and memory placement can be important; in return, they can achieve far greater scalability.

The question is whether there is a way to combine the advantages of the two approaches: maximizing the uniform memory access property while simultaneously allowing greater scalability. The answer is a partial yes, if we accept some compromises on the uniformity of the memory model and some limits of scalability. The machine we discuss in this section, an experimental prototype multiprocessor called Wildfire, built by Sun Microsystems, attempts to do exactly this.

One key motivation for an approach that maximizes the uniformity of memory access while accepting some limits on scalability is the rising importance of OLTP and web server markets for large-scale multiprocessors. In comparison to scientific applications, which played a key role in driving both SMP and DSM development, commercial server applications have both less predictable memory access patterns and less demand for scalability to hundreds or thousands of processors.

The Wildfire Architecture

Wildfire attempts to maximize the benefits of SMP, while allowing scalability by creating a DSM architecture using large SMPs as the nodes. The individual nodes in the Wildfire design are Sun E series multiprocessors (E6x00, E5x00, E4x00, or E3x00. Our measurements in this section are all done with E6000 multiprocessors as the nodes. An E6000 can accept up to 15 processor or I/O boards on the Gigaplane bus interconnect, which supports 50 million bus transactions per second, up to 112 outstanding transactions, and has a peak bandwidth of 3.2 GB/sec. Each processor board contains 2 UltraSPARC III processors.

Wildfire can connect 2 to 4 E6000 multiprocessors by replacing one dual processor (or I/O) board with a Wildfire Interface board (WFI), yielding up to 112 processors (4 x 28), as shown in Figure 6.45. The WFI board supports one coherent address space across all four multiprocessor nodes with the two high-order address bits used to designate which node contains a memory address. Hence, Wildfire is a cache-coherent nonuniform memory access architecture (cc-NUMA) with large nodes. Within each node of 28 processors, memory is uniformly accessible, only processes that span nodes need to worry about the non uniformity in memory access times.

The WFI plugs into the bus and sees all memory requests; it implements the global coherence across up to four nodes. Each WFI has three ports that connect to up to three additional Wildfire nodes, each with a dual directional 800 MB/sec connection. WFI uses a simple directory scheme, similar to that discussed in Section 6.7. To keep the amount of directory state small, the directory is actually a cache, which is backed by the main memory in the node. When an miss occurs, the request is routed to the home node for the requested address. When the re-

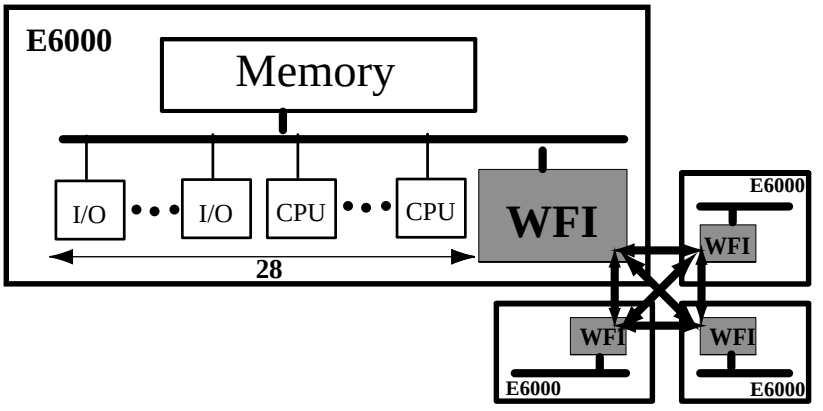


FIGURE 6.45 The Wildfire Architecture uses a bus-based SUN Enterprise server as its building blocks. The WFI replaces a processor or I/O board and provides internode coherency and communication, as well as hardware support for page replication.

quest arrives at the WFI of the home node, the WFI does a directory look-up. If the address is cached locally or clean in memory, a bus transaction is used to retrieve it. If the requested data is cached exclusively in a remote node, a request is sent to that remote node, where the WFI on that node generates a bus request to fetch the data. When the data is returned from either the remote owner or the home node, it is placed on the bus by the WFI and returned to the requesting processor.

We can see from this discussion one major disadvantage of this design: each remote request requires either four or five bus transactions. Two transactions are required at the local node and two or three others are required elsewhere, depending on where the data is cached:

- If the referenced data is cached only in the home node, then two additional bus transactions in the home are sufficient to retrieve the data.
- If the referenced data is cached exclusively in a third remote node, then two bus transactions are required at the remote node and one is required at the home node (to write the shared data back into the home memory).

These are one-way transactions and the E6000 bus is a split transaction bus, so even a normal memory access takes two bus transactions. Nonetheless, there is an increase in the required bus bandwidth of between 1.5 and 2.5. This increase

means that the processor count at which the buses within the nodes become saturated is lowered by a factor of at least 1.5, so that if a 28 processor design saturated the bus bandwidth of an E6000, a four-node Wildfire design could accommodate about 18 processors per node before saturating the bus bandwidth, assuming an even distribution of remote requests. A significant fraction of requests to data cached remotely from its home would further lower the useful processor count in each node. We will return to a further discussion of the advantages and disadvantages of this approach shortly, but first, let us look at how the Wildfire design reduces the fraction of costly remote memory accesses.

Using Page Replication and Migration to Reduce NUMA Effects

Wildfire uses special support, called CMR for *Coherent Memory Replication*, for page migration and replication. The idea is inspired by a more sophisticated hardware scheme for supporting migration and replication, called *COMA* for *Cache Only Memory Architecture*. COMA is an approach that treats all main memory as a cache allowing replication and migration of memory blocks. Full COMA implementations are quite complex, so a variety of simplifications have been proposed. CMR is based on one of these simplifications called S-COMA, for Simple COMA. S-COMA, like CMR, uses page-level mechanisms for migrating and replicating pages in memory, although coherence is still maintained at the cache-block level. We discuss the COMA ideas, as well as other approaches to migration and replication, in more detail in the historical perspectives and in the exercises.

To decide when to replicate or migrate pages, CMR uses a set of page counters that record the frequency of misses to remote pages. Migration is preferred when a page is primarily used by a node other than the one where the page is currently allocated. Replication is useful when multiple nodes share a page; the drawback of replication is that it requires extra memory. When the node sizes in a DSM are small, page migration and replication can lead to both excessive overhead for moving pages and excessive memory overhead from duplication of pages. With the large nodes in Wildfire, however, page-level migration and replication are much more attractive.

CMR, like S-COMA, maintains coherence at the unit of a cache-block, rather than at the page level. This choice is important for two reasons. First, maintaining coherence at the page level is likely to lead to a significant numbers of false sharing misses; we saw this increase in false sharing misses with increases in block size in Section 6.3. Second, the large size of a page means that even true sharing misses are likely to end up moving many bytes of data that are never used. These two drawbacks have limited the usefulness of the Shared Virtual Memory approach, which we discussed on page 733. CMR avoids these problems by making the unit of coherence a cache block and by selectively migrating and replicating some pages, while leaving others as standard NUMA pages that are accessed remotely when a cache miss occurs.

In addition to the page counters that the operating system uses to decide when to migrate or replicate a page, CMR requires special support to map between physical and virtual addresses of replicated pages. First, when a page is replicated the page tables are changed to refer to the local physical memory address of the duplicated page. To maintain coherence, however, a miss to this page must be sent to the home node to check the directory entry in that node. Thus, the WFI maintains a structure that maps the address of a replicated page (the local physical address) to its original physical address (called the global address) and generates the appropriate remote memory request, just as if the page were never replicated. When a write-back request or invalidation request is received, the global address must be translated to the local address, and the WFI maintains such a mapping for all pages that have been replicated. By maintaining these two maps, pages can be replicated while maintaining coherence at the unit of a cache block, which increases the usefulness of page replication.

Performance of Wildfire

In this section we look at the performance of the Wildfire prototype starting first with basic performance measures such as latency for memory accesses and bandwidth and then turning to application performance. Since Wildfire is a research prototype, rather than a product, its performance evaluation for applications is limited, but some interesting experiments that evaluate the use of page migration and replication are available.

Basic Performance Measures: Latency and Bandwidth

To better understand the design trade-offs between DSM architectures with nodes that have small, medium, and large processor counts, we compare the latency and bandwidth measurements of two different machines: the Sun Wildfire and the SGI Origin 2000.

The SGI Origin 2000 is a highly scalable cc-NUMA architecture capable of accommodating up to 2,048 processors. Each node consists of a pair of MIPS R1000 processors sharing a single memory module. An interface processor called the Hub (see Figure 6.46) provides an interface to the memory and directory in each node and implements the coherence protocol. The Hub interfaces directly to the routing chip, which provides a hypercube interconnection network that maintains a a bisection bandwidth of 200 MB/sec per processor. The high dimension of the router also reduces hop counts leading to a lower ratio of remote to local access.

The Origin and Wildfire designs have significantly different motivations, so a comparison of the design trade-offs must acknowledge this fact. Among the most important differences are:

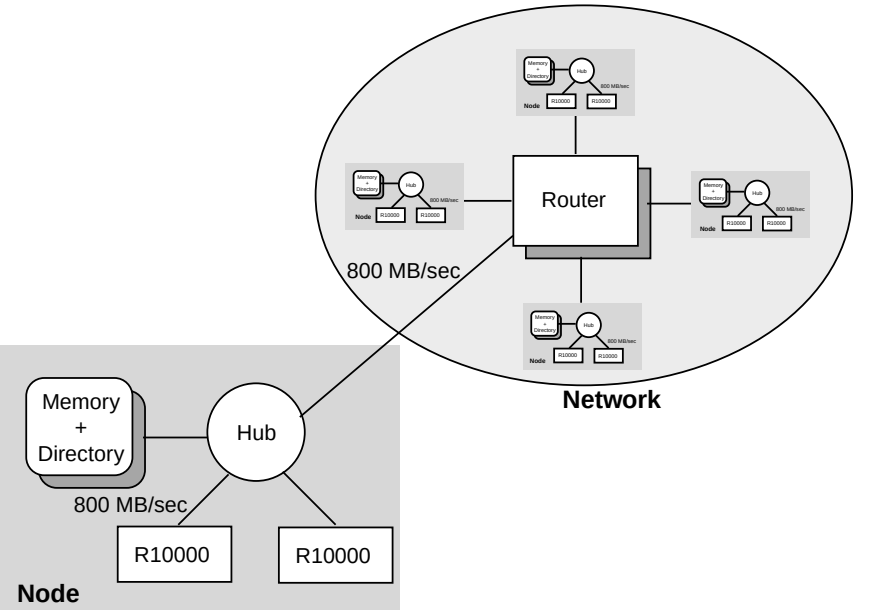


FIGURE 6.46 The SGI Origin 2000 uses an architecture that contains two processors per node and a scalable interconnection network that can handle up to 2,048 processors. A higher dimension network leads to scalable bisection bandwidth and a low ratio per out-of-node and in-node references.

- The range of scalability: Origin can scale to thousands of processors, while the Wildfire design can scale to 112. Practically, the Wildfire design limit is likely to be closer to 64 to 80 processors, since bus bandwidth limits and the need for I/O boards will reduce the effective size of each node.
- The Origin is designed primarily, though not exclusively, for scientific computation and the Wildfire design is oriented primarily for commercial processing. For the Origin design, this means that scalable bandwidth is crucial, and for the Wildfire design, it means that hiding more of the NUMA-ness is crucial.
- The processors are also different in ways that affect both the bandwidth and latency of the nodes, including the block sizes of the L2 caches. We try to reduce this artifact by supplying multiple comparison numbers (e.g., latency to restart and back-to-back worst-case latency).

In Figure 6.47 we compare a variety of latency measurements for the two machines showing the variation arising both from local versus remote accesses and the variation arising from the cache organization. The first portion of the table concentrates on local memory accesses, which remain within one node. We compare both the restart latency, which is the time from miss detection to pipeline re-

start, and a worst-case, back-to-back measurement, which is measured by a sequence of dependent loads. The performance differences arise from the cache architecture (including a factor of two difference in block size), the pipeline architecture, and the main memory access time. Local memory latency also depends on the state of the cache block. We show three cases:

Characteristic	How measured?	Target status?	Sun Wildfire	SGI Origin 2000
Local memory latency	Restart	Unowned	342	338
Local memory latency	Back-to-back	Unowned	330	472
Local memory latency	Restart	Exclusive	362	656
Local memory latency	Back-to-back	Exclusive	350	707
Local memory latency	Restart	Dirty	482	892
Local memory latency	Back-to-back	Dirty	470	1036
Remote memory latency to nearest node	Restart	Unowned	1774	570
Remote memory latency to nearest node	Restart	Dirty	2162	1128
Remote memory latency to furthest node (< 128)	Restart	Unowned	1774	1219
Remote memory latency to furthest node (< 128)	Restart	Dirty	2162	1787
Avg. remote memory latency processors (< 128)	Restart	Unowned	1774	973
Avg. remote memory latency: processors (< 128)	Restart	Dirty	2162	1531
Average memory latency all processors (< 128)	Restart	Unowned	1416	963
Average memory latency all processors (< 128)	Restart	Dirty	1742	1520
Three hop miss to nearest node	Restart	Dirty	2550	953
Three hop miss to furthest node (worst case)	Restart	Dirty	2550	1967
Average three hop miss	Restart	Dirty	2453	1582

FIGURE 6.47 A comparison of memory access latencies (in ns) between the Sun Wildfire prototype (using E6000 nodes) and a SGI Origin 2000 shows significant differences in both local and remote access times. This table has four parts corresponding to local memory accesses (which are within the node), remote memory access involving only the requesting and home node, a third section that compares the average memory latency for the combination of local and remote (but not 3-hop) misses, and a final section showing the 3-hop latencies. The second column describes whether the latency is measured by time to restart the pipeline or by the back-to-back miss cost. For local accesses we show both; for remote accesses, we show the restart latency, which is the more likely case. The third column indicates the state of the remote data. Unowned means that it is in the shared or invalid state in the other caches. Exclusive means exclusive but clean, which requires an intervention to be completed before the memory access can complete, so that write serialization may be maintained. Dirty indicates that the data is exclusive and has been updated; an access, therefore, requires retrieving the data from the cache. In the local case, we show all three possibilities, to show the effect of the processor architecture (e.g., intervention cost and cache block size both affect the access times), while for remote accesses we show the unowned and dirty case, which are likely to be the most frequent.

1. the accessed block is unowned or it is in the shared state
2. the accessed block is owned exclusively but clean, which requires that the block be invalidated,
3. the accessed block is owned and dirty, which requires that the block be retrieved from the cache to satisfy the miss.

These 6 combinations (3 possible states of the target block x 2 possible miss timings) are the most likely cases of a local miss, though there are several other possibilities. These latencies are primarily dominated by choices in the microprocessor design (such as minimizing time to restart or minimizing total miss time) as well as in the local memory system and coherence implementation. These choices increase the difficulty in comparing memory latency for a multiprocessor, since some of these design choices affect the remote latencies as well.

The second section of the table compares the remote access times under a variety of different circumstances but all assuming that the home address is in a different node and that any cached copies are in the home node. For these numbers we use restart latency and consider the two most probable coherence states for a remotely accessed datum: unowned and dirty. The first two entries describe the time to access a datum whose home is in the nearest node; for the Wildfire system all remote nodes are equidistant, while for the Origin, the nearest node is one router hop away. The second pair of numbers deals with the latency when the home is as far away as possible for Origin. Finally, the third and last pair provide the average latency for a uniform distribution of the home address across a multiprocessor with 128 processors.

The fourth set of numbers deals with 3-hop misses, assuming that the owner is in a different node from either the home or the originating node. Here the most likely case is that the data is Dirty, and we show the restart latency for this case under the best, worst, and average assumptions.

From these measurements, we can see several of the trade-offs at work in a design that uses large nodes versus one that uses smaller nodes. Large nodes increase the number of processors reachable with a local access, but also typically have a longer remote access time. The latter is driven primarily by the higher overhead of acquiring access to the bus either for the directory or to access a remote cached copy. Of course, access latency is only part of the picture, bandwidth is also affected by these design decisions.

As Figure 6.48 shows, the pipeline memory bandwidth can be measured in many different ways. The Origin design supports greater memory bandwidth by every measure except local bandwidth to dirty data. Local bandwidth and bisection bandwidth are almost three times higher on a per processor basis for Origin.

Application performance of Wildfire

In this section, we examine the performance of Wildfire, first on an OLTP application and then on a scientific application. We look at both the basic performance

Characteristic	Sun Wildfire MB/sec	SGI Origin 2000 MB/sec
Pipelined local memory bandwidth: unowned data	312	554
Pipelined local memory bandwidth: exclusive data	266	340
Pipelined local memory bandwidth: dirty data	246	182
Total local memory bandwidth (per node)	2,700	631
Local memory bandwidth per processor	96	315
Aggregate local memory bandwidth (all nodes, 112 processors)	10,800	39,088
Remote memory bandwidth, unowned data		508
Remote 3-hop bandwidth, dirty data		238
Total bisection bandwidth (112 processors)	9,600	25,600
Bisection bandwidth per processor (112 processors)	86	229

FIGURE 6.48 A comparison of memory bandwidth measurements (in MB/sec) between the Sun Wildfire prototype (using E6000 nodes) and a SGI Origin 2000 shows significant differences in both local and remote memory bandwidth. The first section of the table compares pipelined local memory bandwidth, which is defined as the sustainable bandwidth for independent accesses generated by a single processor; like restart latency, this measure depends on the state of the addressed cache block. The second section of the table compares the total local memory bandwidth (i.e., within a node) on a per processor basis and system wide. The third section compares the memory bandwidth for remote accesses, both a two-hop access to an unowned cache block and a three-hop access to a dirty cache block. The final section compares the total bisection bandwidth for the entire system and on a per processor basis.

of the architecture versus alternatives such as a strict SMP or a small-node NUMA and then consider the effect of Wildfire's support for replication and migration.

Performance of the OLTP Workload

In this study an OLTP application supporting 900 warehouses was run on a 16-processor E6000 and on a two-node, 16-processor Wildfire configuration. I/O was supplied by 240 disks connected by fiber-channel. To examine the performance of Wildfire and the effect of its support for replication and migration, we consider six system alternatives:

1. Ideal SMP: a 16-processor SMP design, modeled using the E6000.
2. Wildfire with CMR and locality scheduling: a 2-node, 16-processor Wildfire with replication and migration enabled and using the locality scheduling in the OS.
3. Wildfire with CMR only.
4. Wildfire base with neither CMR nor locality scheduling.
5. Unoptimized Wildfire with poor data placement: Wildfire with poor data

placement and unintelligent scheduling. Poor data placement is modeled by assuming that 50% of the cache misses are remote, which in practice is unrealistic.

6. Unoptimized Wildfire with thin nodes (2 processors per node) and poor data placement. This system assumes Wildfires interconnection characteristics, but with eight two-processor nodes. Poor data placement is modeled by assuming that 87.5% (i.e., 14/16) 1of the cache misses are remote, which in practice is unrealistic.

To examine performance we first look at the fraction of cache misses satisfied within a node. Figure 6.49 shows the fraction of local accesses for each of these configurations. For this OLTP application the Wildfire optimizations improve fraction of local accesses by a factor of 1.23 over unoptimized Wildfire, bringing the fraction of local accesses to 87%.

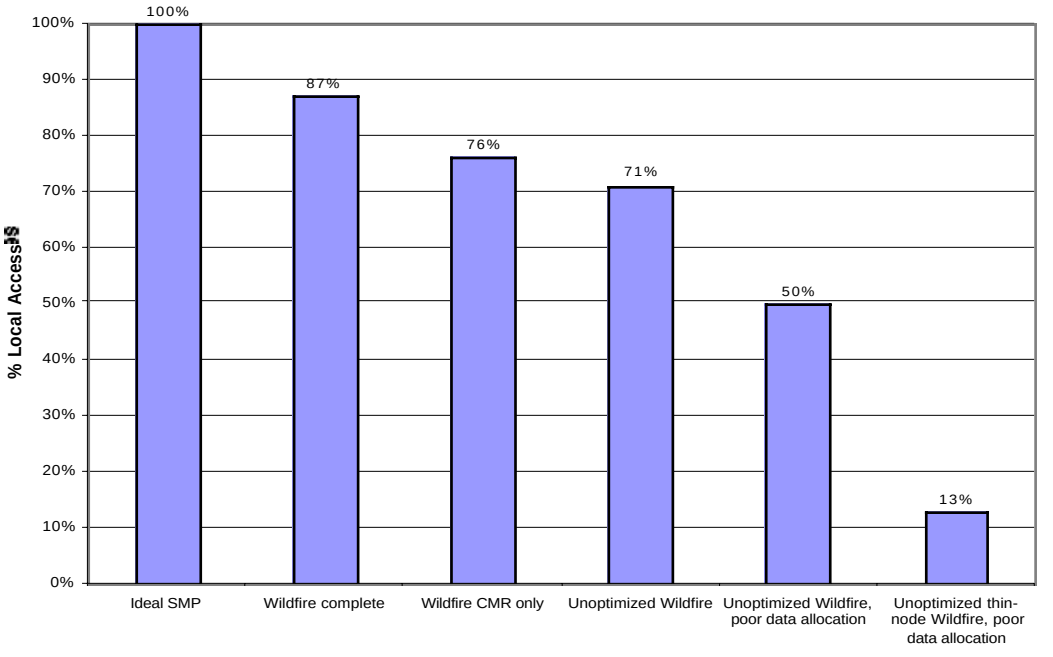


FIGURE 6.49 The fraction of local accesses (defined as within the node) is shown for six different configurations, ranging from an ideal SMP (with only one node and 16 processors) to four configurations with 8-processor nodes, to a configuration with thin, 2-processor nodes. The fraction of remote accesses is set as a parameter for the two rightmost data points, while the other numbers are measured.

Figure 6.50 shows how these changes in local versus remote access fractions translate to performance for this OLTP application. The performance of each system in Figure 6.50 is relative to the E6000; however, as we will see when we examine a scientific application, the E6000 can encounter performance losses from bus contention at 16 processors, so that, in fact, the performance of the E6000 does not represent an upper bound for a multiprocessor using sixteen of the same processors. The E6000 performance is probably within 10-20% of contention-free performance for this benchmark. As we can see from the data the penalty for off-node accesses translates directly to reduced performance. The next section examine how Wildfire performs for a scientific application.

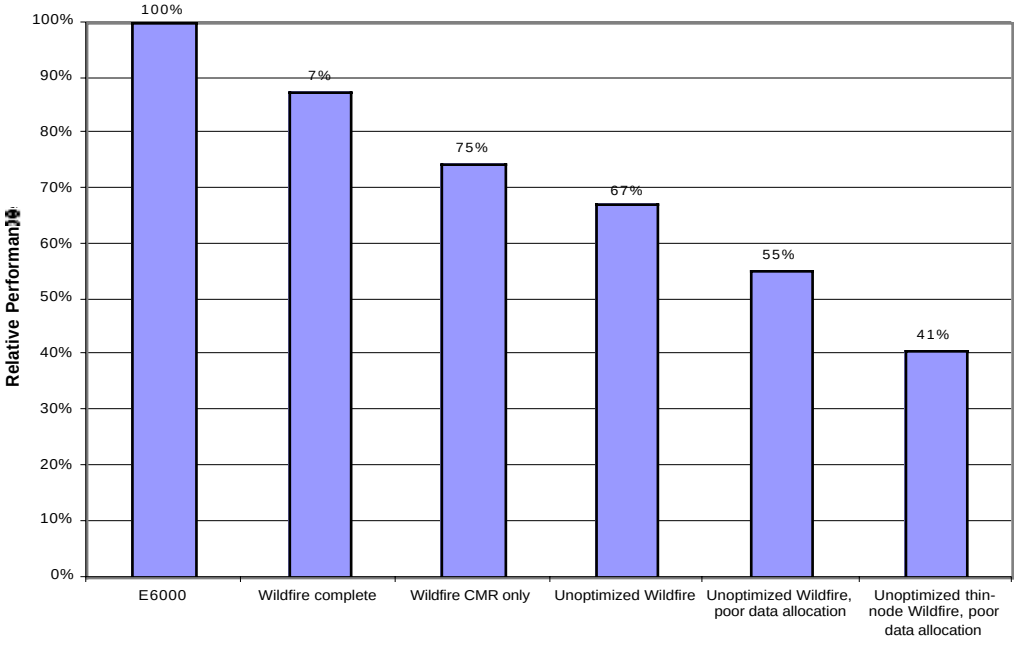


FIGURE 6.50 The performance of the OLTP application using 16 processors is highest for the E6000, and drops off as remote memory accesses become a major performance loss.

Performance of Wildfire on a Scientific Application

In this section we examine a performance study of Wildfire using a Red-Black finite difference solver to solve a 2-dimensional Poisson equation for a square grid. In this implementation, each 2x2 block of grid points is assigned either a red or

black color, so that the overall grid looks like a checkerboard. Red data points are updated based on values of black data points and vice versa, which allows all red points to be updated in parallel and all black points to be updated in parallel. A point is updated by accessing the four neighboring points (all of which are a different color). This data access pattern is common in two-dimensional solvers.

Our first performance comparisons examine the performance of Wildfire versus the E6000 and the E10000. The E 10000 uses a two-level interconnect. Four processors are connected with a 4x4 cross-bar to four memory modules, creating a 4-processor SMP. Up to 16 of these 4-processor nodes can be connected with the Starfire interconnect, which uses a 16x16 cross-bar. Coherence is maintained by a global broadcast scheme.

Figure 6.51 shows the performance of the generalized red-black (GRB) solver for six different configurations. The performance is given in terms of iterations per second with more iterations being better. The leftmost group of columns compares 24-processor measurements on an E6000, E10000, and Wildfire; while the rightmost bars compare 36-processor runs on two different Wildfire configurations and an E10000. Both the 24 and 36 processor runs use the same processor (250 MHZ UltraSPARC II) with 4MB secondary caches.

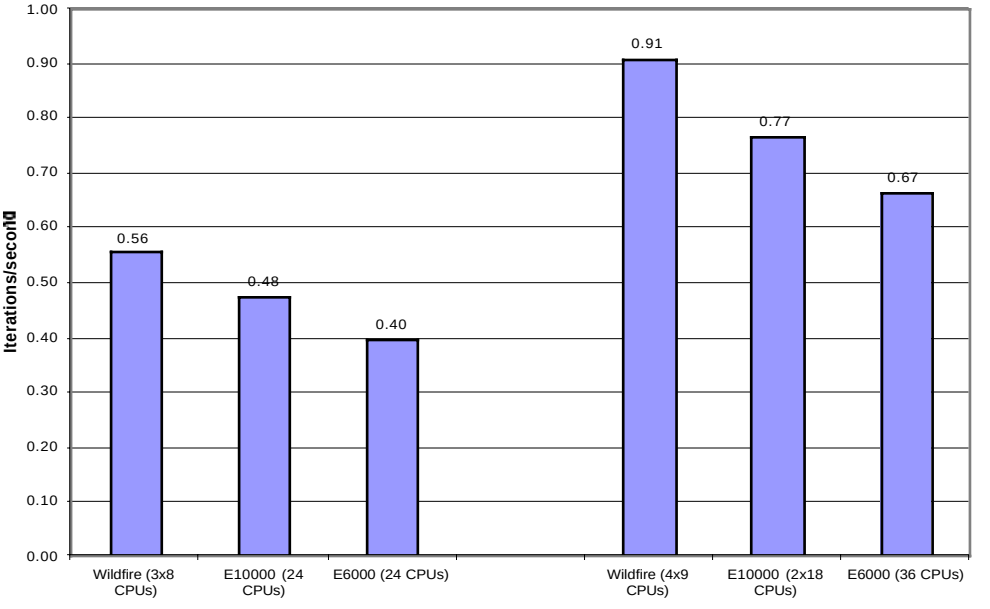


FIGURE 6.51 Wildfire performance for the Red-Black solver measured as iterations per second shows the performance for three different 24-processor and three different 36-processor machines. Iterations per second is directly proportional to performance.

The 24-processor runs include a 3-node Wildfire configuration (with an 8-processor E6000 in each Wildfire node), a 6-node E10000 and a 24-processor E6000. The performance differences among the 24-processor runs on Wildfire, the E1000, and the E6000 arise primarily from bus and interconnect differences. The global broadcast of the E10000 has nontrivial overhead. Thus, despite the fact that the E10000 interconnect has performance equal to that of Wildfire, the performance of Wildfire is about 1.17 times better. For the E6000, the measured bus usage for the 24-processor runs is between 90% and 100%, leading to a significant bottleneck and lengthened memory access time. Overall, Wildfire has a performance advantage of about 1.19 versus the E6000. Equally importantly, these measurements tell us that configurations of Wildfire with larger processor counts per node will not have good performance, at least for applications with behavior similar to this solver. The 36 processor runs confirm this view.

The 36-processor runs compare three alternatives: a 9-node E10000, a 2x18 configure of Wildfire (each Wildfire node is an 18-processor E6000) and a 4x9 configuration of Wildfire (each Wildfire node is an 9-processor E6000). The most interesting comparison here involve the 36-processor versus 24-processor results. The E10000 shows a faster than linear speedup (1.67 in runtime versus 1.5 in processor count); this probably results from improved cache behavior due to the smaller data set that each processor must access in the 36-processor case. The Wildfire results are even more interesting; the 4x9 configuration also shows faster than linear speed-up versus the 24-processor result. The 2x18 configuration, however, shows speed-up that is slower than linear (1.38 vs. 1.5), most probably because the bus has become a major bottleneck.

How well do the migration and replication capabilities of Wildfire work for scientific applications? To examine this question, this solver was executed starting with a memory allocation that placed all the data on a single node. Wildfire's migration and replication capabilities were used to allow data to migrate and replicate to one of the other nodes. Figure 6.52 shows the performance in iterations per second over time for a 1, 2, 3, and 4 node Wildfire, each with 24 processors/node. As shown, the 2, 3, and 4 node runs converge to stable and best performance after somewhere between 120 and 180 seconds. Since during the initial time period, the application averages about 0.2 iterations per second, it requires between 600 and 900 iterations to reach the stable performance levels.

Although the eventual convergence to a good operating point from an initial pathological memory allocation is impressive, the number of iterations required is rather large, and leaves open the question of how well the migration and replication strategies might work in problems where the memory allocation continued to change over time.

A key question is what the relative benefits of migration and replication are? Figure 6.53 examines this question by showing the iteration rate and time to reach that rate. We also show the number of replications and migrations. The pri-

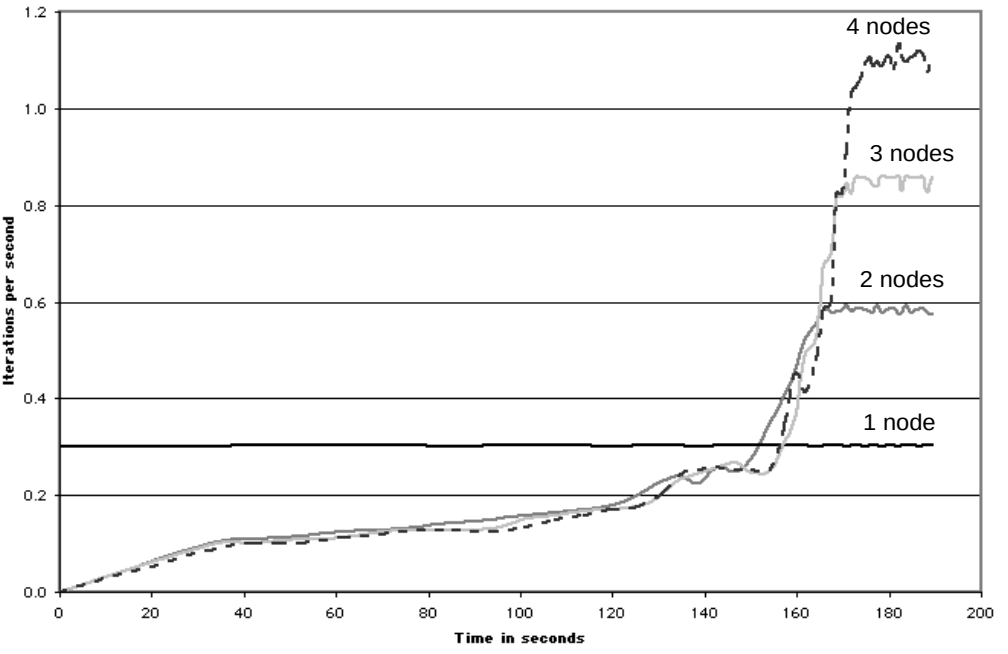


FIGURE 6.52 T**he replication and migration support of Wildfire allows an application to start with a pathological memory allocation (all memory on one node) and converge to a stable allocation that gives nearly linear speed-up.** The final iterations/second number shows that the 96-processor, 4-node version achieves 90% of linear speedup. As expected, the two node runs converge slightly faster than three or four node runs.

mary conclusion we can draw from the performance of these three cases is that the stable performance level for migration is competitive with the combination of migration and replication. Since supporting migration had much lower hardware costs that supporting replication (because the reverse memory maps are not needed), a design that supports migration may be equally or more cost effective than supporting both migration and replication. The large data set coupled with well

defined access patterns by the "owner" of each portion of the grid means that replication buys little over only migration.

Policy	Iterations per second	Iterations needed to reach stability	# Migrations	# Replications
No migration or replication	0.10	0	0	0
Migration only	1.06	154 sec.	99,251	
Replication only	1.15	61 sec.		98.545
Migration + replication	1.09	151 sec.	98,543	85

FIGURE 6.53 Migration only, replication only, and the combination of all three achieve about the same performance given enough execution time and that number is roughly 10 times the performance achieved with the initial data allocation and no replication or migration. For this experiment, which used a 96-processor, 4-node Wildfire, the pages were allocated in a cyclic fashion, meaning that roughly 25% of were allocated to the correct location initially. A large data set size (16K x 8K) that exceeds the capacity of the secondary caches, leads to a high miss rate, which requires migration, replication, or careful initial data placement to reduce the miss penalty.

Concluding Remarks on Wildfire

Wildfire represents an alternative to thin-node NUMAs with 2 to 4 processors per node, while permitting greater scalability than strict SMP designs. The shift in market interest from scientific and supercomputing applications to large-scale servers for database and web applications may favor a fat-node design with 8 to 16 processors per node. The two primary reasons for this are:

1. Although a moderate range of scalability, up to a few hundred processors may be of interest, the "sweet spot" of the server market is likely to be tens of processors. Few, if any, customers will express interest in the thousand processor machines that are a key part of the supercomputer marketplace.
2. The memory access patterns of commercial applications tend to have less sharing and less predictable sharing and data access. The lower rates of sharing are key because a fat node design will tend to have lower bisection bandwidth per processor than a thin-node design. Since a fat-node design has somewhat less dependence on exact memory allocation and data placement, it is likely to perform better for applications with irregular or changing data access patterns. Furthermore, fat-nodes make it easier for migration and replication to work well.

The drawbacks of a fat node design are essentially the dual of its advantages. These include: less scalability, lower bisection bandwidth per processor, and higher internode latencies. For applications that require significant amounts of internode communication even with fat nodes, a fat-node design will face a more challenging programming and optimization task, since the ratio of local to remote accesses times is likely to be quite a bit larger. To read more on Wildfire see:

Hagersten and Koster [1998] and Noordergraaf and van der Pas [1999], which are also the sources for the data in this section.

Considering the growing significance of the commercial server market with its less predictable memory access patterns, its reduced emphasis on ultimate scalability, and its lower interprocess communication requirements, it is likely the "plump" node designs will become more attractive. Growing processor demands and avoidance of bus limits, is likely to lead to designs with 4-8 processors per node rather than the 16-24 limit in Wildfire. Although fatter nodes are likely to be beneficial, the nonuniform access time to memory cannot be ignored when the local node provide SMP-style access to only 3-7 other nodes.

.

6.12 Another View: Multithreading in a Commercial Server

As we have seen, dynamic scheduling can be used to make a single program run faster, as we saw in the Pentium III. Alternatively, multithreading can use a different form of dynamic scheduling (scheduling across multiple threads) to increase the throughput of multiple simultaneously executing programs. This is the approach used in the IBM RS64 III.

The IBM RS64 III processor, also called Pulsar, is a PowerPC microprocessor that supports two different IBM product lines: the RS/6000 series, where it is called the RS64 III processor, and the AS/400 series, where it is called the A50. Both product lines are aimed at commercial servers and focus on throughput in common commercial applications.

Motivated by the observation that such applications encounter high cache and TLB miss rates and thus degraded CPI, the designers decided to include a multithreading capability to enhance throughput and make use of the processor during long TLB or cache-miss stalls. In deciding how to support multithreading, the designers considered three facts:

1. The Pulsar processor, which was based on the earlier Northstar, is a statically scheduled processor.
2. The performance penalty for multithreading must be small both in silicon area and in clock rate.
3. Single thread performance on Pulsar must not suffer.

This combination of considerations led to a multithreading architecture with the following characteristics:

1. Pulsar supports precisely two threads: this minimizes both the incremental silicon area and the potential clock rate impact.

2. The multithreading is coarsely scheduled; that is, threads are not interleaved, instead a thread switch occurs only when a long latency stall is encountered. Coarse multithreading was chosen to maximize single thread performance and make use of the statically scheduled pipeline structure, which makes SMT an impractical choice.

To implement the multithreading architecture, Pulsar includes two copies of the register files and PC register, which resulted in relatively minor silicon overhead (< 10%). In addition, a special register that determines the maximum number of cycles between a thread switch ensures that no thread is ever completely starved for cycles. The overall architecture provides a significant improvement in multithreaded throughput, a key metric for the commercial server workloads. The Pulsar microprocessor is the first widely available, mainline microprocessor to support multithreading; it is likely that future microprocessors will include such a capability either a coarse-grained form or using the SMT approach.

6.13 Another View: Embedded Multiprocessors

Multiprocessors are now common in server environments, and several desktop multiprocessors are available from vendors, such as Sun, Compaq, and Apple. In the embedded space, a number of special-purpose designs have used customized multiprocessors, including the Sony Playstation described in Chapters 2 and 5. Many special-purpose embedded designs consist of a general-purpose programmable processor with special purpose finite-state machines that are used for stream-oriented I/O. In applications ranging from computer graphics and media processing to telecommunications, this style of special-purpose multiprocessor is becoming common. Although the interprocessor interactions in such deigns is highly regimented and relatively simple–consisting primarily of a simple communication channel–because much of the design is committed to silicon, ensuring that the communication protocols among the input/output processors and the general-purpose processor are correct is a major challenge in such designs.

More recently, we have seen the first appearance, in the embedded space, of embedded multiprocessors built from several general-purpose processors. These multiprocessors have been focused primarily on the high-end telecommunications and networking market, where scalability is critical. An example of such a design is the MXP processor designed by empowerTel Networks for use in voice over IP systems. The MXP processor consists of four main components:

1. An interface to serial voice streams, including support for handling jitter.
2. Support for fast packet routing and channel lookup.

3. A complete Ethernet interface, including the MAC layer.
4. Four MIPS32 R4000-class processors each with its own caches (a total of 48 KB or 12 KB per processor).

The MIPS processors are used to run the code responsible for maintaining the voice over IP channels, including the assurance of quality of service, echo cancellation, simple compression, and packet encoding. Since the goal is to run as many independent voice streams as possible, a multiprocessor is an ideal solution.

Because of the small size of the MIPS cores, the entire chip takes only 13.5M transistors. Future generations of the chip are expected to handle more voice channels, as well as do more sophisticated echo cancellation, voice activity detection, and more sophisticated compression.

Your authors expect that multiprocessing will become widespread in the embedded computing arena in the future for two primary reasons. First, the issues of binary software compatibility, which plague desktop and server systems, are less relevant in the embedded space. Often software in an embedded application is written from scratch for an application or significant modified. Second, the applications often have natural parallelism, especially at the high-end of the embedded space. Examples of this natural parallelism abound in applications such as a set-top box, a network switch, or a game system. The lower barriers to use of thread-level parallelism together with the greater sensitivity to die cost (and hence efficient use of silicon) will likely lead to more ready adoption of multiprocessing in the embedded space, as the application needs grow to demand more performance.

6.14 Fallacies and Pitfalls

Given the lack of maturity in our understanding of parallel computing, there are many hidden pitfalls that will be uncovered either by careful designers or by unfortunate ones. Given the large amount of hype that has surrounded multiprocessors, especially at the high end, common fallacies abound. We have included a selection of these.

Pitfall: Measuring performance of multiprocessors by linear speedup versus execution time.

"Mortar shot" graphs—plotting performance versus number of processors showing linear speedup, a plateau, and then a falling off—have long been used to judge the success of parallel processors. Although speedup is one facet of a parallel program, it is not a direct measure of performance. The first question is the power of the processors being scaled: A program that linearly improves performance to equal 100 Intel 486s may be slower than the sequential version on a workstation. Be especially careful of floating-point-intensive programs; process-

ing elements without hardware assist may scale wonderfully but have poor collective performance.

Comparing execution times is fair only if you are comparing the best algorithms on each computer. Comparing the identical code on two processors may seem fair, but it is not; the parallel program may be slower on a uniprocessor than a sequential version. Developing a parallel program will sometimes lead to algorithmic improvements, so that comparing the previously best-known sequential program with the parallel code—which seems fair—will not compare equivalent algorithms. To reflect this issue, the terms *relative speedup* (same program) and *true speedup* (best program) are sometimes used.

Results that suggest *super-linear* performance, when a program on n processors is more than n times faster than the equivalent uniprocessor, may indicate that the comparison is unfair, although there are instances where "real" superlinear speedups have been encountered. For example, when Ocean is run on two processors, the combined cache produces a small superlinear speedup (2.1 vs. 2.0).

In summary, comparing performance by comparing speedups is at best tricky and at worst misleading. Comparing the speedups for two different multiprocessors does not necessarily tell us anything about the relative performance of the multiprocessors. Even comparing two different algorithms on the same multiprocessor is tricky, since we must use true speedup, rather than relative speedup, to obtain a valid comparison.

Fallacy: Amdahl's Law doesn't apply to parallel computers.

In 1987, the head of a research organization claimed that Amdahl's Law (see section 1.6) had been broken by an MIMD multiprocessor. This statement hardly meant, however, that the law has been overturned for parallel computers; the neglected portion of the program will still limit performance. To understand the basis of the media reports, let's see what Amdahl [1967] originally said:

A fairly obvious conclusion which can be drawn at this point is that the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude. [p. 483]

One interpretation of the law was that since portions of every program must be sequential, there is a limit to the useful economic number of processors—say 100. By showing linear speedup with 1000 processors, this interpretation of Amdahl's Law was disproved.

The basis for the statement that Amdahl's Law had been "overcome" was the use of scaled speedup.The researchers scaled the benchmark to have a data set size that is 1000 times larger and compared the uniprocessor and parallel execution times of the scaled benchmark. For this particular algorithm the sequential

portion of the program was constant independent of the size of the input, and the rest was fully parallel—hence, linear speedup with 1000 processors.

We have already described the dangers of relating scaled speedup as true speedup. Additional problems with this sort of scaling methodology, which can result in unrealistic running times, were examined in section 6.10.

Fallacy: Linear speedups are needed to make multiprocessors cost-effective.

It is widely recognized that one of the major benefits of parallel computing is to offer a "shorter time to solution" than the fastest uniprocessor. Many people, however, also hold the view that parallel processors cannot be as cost-effective as uniprocessors unless they can achieve perfect linear speedup. This argument says that because the cost of the multiprocessor is a linear function of the number of processors, anything less than linear speedup means that the ratio of performance/cost decreases, making a parallel processor less cost-effective than using a uniprocessor.

The problem with this argument is that cost is not only a function of processor count, but also depends on memory and I/O. The effect of including memory in the system cost was pointed out by Wood and Hill [1995], and we use an example from their article to demonstrate the effect of looking at a complete system. They compare a uniprocessor server, the Challenge DM (a deskside unit with one processor and up to 6 GB of memory), against a multiprocessor Challenge XL, a rack-mounted, bus-based multiprocessor holding up to 32-processors. (The XL also has faster processors than those of the Challenge DM—150 MHz versus 100 MHz—but we will ignore this difference.)

First, Wood and Hill introduce a cost function: $cost(p, m)$, which equals the list price of a multiprocessor with p processors and m megabytes of memory. For the Challenge DM:

$$cost(1, m) = \$38{,}400 + \$100 \times m$$

For the Challenge XL:

$$cost(p, m) = \$81{,}600 + \$20{,}000 \times p + \$100 \times m$$

Suppose our computation requires 1 GB of memory on either multiprocessor. Then the cost of the DM is \$138,400, while the cost of the Challenge XL is $\$181{,}600 + \$20{,}000 \times p$.

For different numbers of processors, we can compute what speedups are necessary to make the use of parallel processing on the XL *more* cost effective than that of the uniprocessor. For example, the cost of an 8-processor XL is \$341,600, which is about 2.5 times higher than the DM, so if we have a speedup on 8 processors of more than 2.5, the multiprocessor is actually *more* cost effective than the uniprocessor. If we are able to achieve linear speedup, the 8-processor XL

system is actually more than *three times* more cost effective! Things get better with more processors: On 16 processors, we need to achieve a speedup of only 3.6, or less than 25% parallel efficiency, to make the multiprocessor as cost effective as the uniprocessor.

The use of a multiprocessor may involve some additional memory overhead, although this number is likely to be small for a shared-memory architecture. If we assume an extremely conservative number of 100% overhead (i.e., double the memory is required on the multiprocessor), the 8-processor multiprocessor needs to achieve a speedup of 3.2 to break even, and the 16-processor multiprocessor needs to achieve a speedup of 4.3 to break even.

Surprisingly, the XL can even be cost effective when compared against a headless workstation used as a server. For example, the cost function for a Challenge S, which can have at most 256 MB of memory, is

$$cost(1, m) = \$16{,}600 + \$100 \times m$$

For problems small enough to fit in 256 MB of memory on both multiprocessors, the XL breaks even with a speedup of 6.3 on 8 processors and 10.1 on 16 processors.

In comparing the cost/performance of two computers, we must be sure to include accurate assessments of both total system cost and what performance is achievable. For many applications with larger memory demands, such a comparison can dramatically increase the attractiveness of using a multiprocessor.

Fallacy: Multiprocessors are "free."

This fallacy has two different interpretations, and both are erroneous. The first is, given that modern microprocessors contain support for snooping caches, we can build small-scale, bus-based multiprocessors for no additional cost in dollars (other than the microprocessor cost) or sacrifice of performance. Many designers believed this to be true and have even tried to build multiprocessors to prove it.

To understand why this doesn't work, you need to compare a design with no multiprocessing extensibility against a design that allows for a moderate level of multiprocessing (say 2–4 processors). The 2–4 processor design requires some sort of bus and a coherence controller that is more complicated than the simple memory controller required for the uniprocessor design. Furthermore, the memory access time is almost always faster in the uniprocessor case, since the processor can be directly connected to memory with only a simple single-master bus. Thus the strictly uniprocessor solution typically has better performance and lower cost than the 1-processor configuration of even a very small multiprocessor.

It also became popular in the 1980s to believe that the multiprocessor design was free in the sense that an MP could be quickly constructed from state-of-the-art microprocessors and then quickly updated using newer processors as they

became available. This viewpoint ignores the complexity of cache coherence and the challenge of designing high-bandwidth, low-latency memory systems, which for modern processors is extremely difficult. Moreover, there is additional software effort: compilers, operating systems, and debuggers all must be adapted for a parallel system. The next two fallacies are closely related to this one.

Fallacy: Scalability is almost free.

The goal of scalable parallel computing was a focus of much of the research and a significant segment of the high-end multiprocessor development from the mid-1980s through the late 1990s. In the first half of that period, it was widely held that you could build scalability into a multiprocessor and then simply offer the multiprocessor at any point on the scale from a small to large number of processors without sacrificing cost effectiveness. The difficulty with this view is that multiprocessors that scale to larger processor counts require substantially more investment (in both dollars and design time) in the interprocessor communication network, as well as in aspects such as operating system support, reliability, and reconfigurability.

As an example, consider the Cray T3E, which uses 3D torus capable of scaling to 2,048 processors as an interconnection network. At 128 processors, it delivers a peak bisection bandwidth of 38.4 GB/s, or 300 MB/s per processor. But for smaller configurations, the Compaq Alphaserver ES40 can accept up to 4 processors and has 5.6 GB/s of interconnect bandwidth, or almost four times the bandwidth per processor. Furthermore, the cost per CPU in a Cray T3E is several times higher than the cost in the ES40.

The cost of scalability can be seen even in more limited design ranges, such as the Sun Enterprise server line that all use the same basic Ultraport interconnect, scaling the amount of interconnect for different systems. For example, the 4 processor Enterprise 450 places all four processors on a single board and uses an onboard crossbar. The midrange system, designed to support 6 to 30 processors, uses a single address bus and a 32-byte wide data bus to connect the processors. The Enterprise 10000 series uses four addresses buses (memory address interleaved) and a 16x16 crossbar to connect the processors. Although the solution gives getter scalability across the product range than forcing the low-end systems to accommodate four address buses and a multiboard crossbar, the cost of the interconnect system grows faster than linear as the number of processors grows, leading to a higher per processor cost for the 6000 series versus the 450 and for the 10000 series versus the 6000 series.

Scalability is also not free in software: To build software applications that scale requires significantly more attention to load balance, locality, potential contention for shared resources, and the serial (or partly parallel) portions of the program. Obtaining scalability for real applications, as opposed to toys or small kernels, across factors of more than 10 in processor count, is a *major* challenge.

In the future, better compiler technology and performance analysis tools may help with this critical problem.

Pitfall: Not developing the software to take advantage of, or optimize for, a multiprocessor architecture.

There is a long history of software lagging behind on massively parallel processors, possibly because the software problems are much harder. Two examples from mainstream, bus-based multiprocessors illustrate the difficulty of developing software for new multiprocessors. The first has to do with not being able to take advantage of a potential architectural capability, and the second arises from the need to optimize the software for a multiprocessor.

The SUN SPARCCenter was an earlier bus-based multiprocessor with one or two buses. Memory is distributed on the boards with the processors to create a simple building block consisting of processor, cache, and memory. With this structure, the multiprocessor could also have a fast local access and use the bus only to access remote memory. The SUN operating system, however, was not able to deal with the NUMA (non-uniform memory access) aspect of memory, including such issues as controlling where memory was allocated (local versus global). If memory pages were allocated randomly, then successive runs of the same application could have substantially different performance, and the benefits of fast local access might be small or nonexistent. In addition, providing both a remote and a local access path to memory slightly complicated the design because of timing. Since neither the system software nor the application software would not have been able to take advantage of faster local memory and the design was believed to be more complicated, the designers decided to require all requests to go over the bus.

Our second example shows the subtle kinds of problems that can arise when software designed for a uniprocessor is adapted to a multiprocessor environment. The SGI operating system protects the page table data structure with a single lock, assuming that page allocation is infrequent. In a uniprocessor this does not represent a performance problem. In a multiprocessor situation, it can become a major performance bottleneck for some programs. Consider a program that uses a large number of pages that are initialized at start-up, which UNIX does for statically allocated pages. Suppose the program is parallelized so that multiple processes allocate the pages. Because page allocation requires the use of the page table data structure, which is locked whenever it is in use, even an OS kernel that allows multiple threads in the OS will be serialized if the processes all try to allocate their pages at once (which is exactly what we might expect at initialization time!).

This page table serialization eliminates parallelism in initialization and has significant impact on overall parallel performance. This performance bottleneck persists even under multiprogramming. For example, suppose we split the parallel program apart into separate processes and run them, one process per proces-

sor, so that there is no sharing between the processes. (This is exactly what one user did, since he reasonably believed that the performance problem was due to unintended sharing or interference in his application.) Unfortunately, the lock still serializes all the processes—so even the multiprogramming performance is poor. This pitfall indicates the kind of subtle but significant performance bugs that can arise when software runs on multiprocessors. Like many other key software components, the OS algorithms and data structures must be rethought in a multiprocessor context. Placing locks on smaller portions of the page table effectively eliminates the problem.

Pitfall: Neglecting data distribution in a distributed shared-memory multiprocessor.

Consider the Ocean benchmark running on a 32-processor DSM architecture. As Figure 6.31 (page 699) shows, the miss rate is 3.1% for a 64KB cache. Because the grid used for the calculation is allocated in a tiled fashion (as described on page 658), 2.5% of the accesses are local capacity misses and 0.6% are remote communication misses needed to access data at the boundary of each grid. Assuming a 50-cycle local memory access cost and a 150-cycle remote memory access cost, the average miss has a cost of 69.3 cycles.

If the grid was allocated in a straightforward fashion by round-robin allocation of the pages, we could expect 1/32 of the misses to be local and the rest to be remote, which would lead to local miss rate of $3.1\% \times 1/32 = 0.1\%$ and a remote miss rate of 3.0%, for an average miss cost of 146.7 cycles. If the average CPI without cache misses is 0.6, and 45% of the instructions are data references, the version with tiled allocation is

$$\frac{0.6 + 45\% \times 3.1\% \times 146.7}{0.6 + 45\% \times 3.1\% \times 69.3} = \frac{0.6 + 2.05}{0.6 + 0.97} = \frac{2.65}{1.57} = 1.69 \text{ times faster}$$

This analysis only considers latency, and assumes that contention effects do not lead to increased latency, which is very optimistic. Round-robin is also not the worst possible data allocation: if the grid fit in a subset of the memory and was allocated to only a subset of the nodes, contention for memory at those nodes could easily lead to a difference in performance of more than a factor of 2.

6.15 Concluding Remarks

For over a decade prophets have voiced the contention that the organization of a single computer has reached its limits and that truly significant advances can be made only by interconnection of a multiplicity of computers in such a manner as to permit cooperative solution. …Demonstration is made of the continued validity of the single processor approach. … [p. 483]

Amdahl [1967]

The dream of building computers by simply aggregating processors has been around since the earliest days of computing. Progress in building and using effective and efficient parallel processors, however, has been slow. This rate of progress has been limited by difficult software problems as well as by a long process of evolving architecture of multiprocessors to enhance usability and improve efficiency. We have discussed many of the software challenges in this chapter, including the difficulty of writing programs that obtain good speedup due to Amdahl's law, dealing with long remote access or communication latencies, and minimizing the impact of synchronization. The wide variety of different architectural approaches and the limited success and short life of many of the architectures to date has compounded the software difficulties. We discuss the history of the development of these multiprocessors in section 6.16.

Despite this long and checkered past, progress in the last fifteen years leads to some reasons to be optimistic about the future of parallel processing and multiprocessors. This optimism is based on a number of observations about this progress and the long-term technology directions:

1. The use of parallel processing in some domains is beginning to be understood. Probably first among these is the domain of scientific and engineering computation. This application domain has an almost limitless thirst for more computation. It also has many applications that have lots of natural parallelism. Nonetheless, it has not been easy: programming parallel processors even for these applications remains very challenging. Another important, and much larger (in terms of market size), application area is large-scale data base and transaction processing systems. This application domain also has extensive natural parallelism available through parallel processing of independent requests, but its needs for large-scale computation, as opposed to purely access to large-scale storage systems, are less well understood. There are also several contending architectural approaches that may be viable—a point we discuss shortly.

2. It is now widely held that the most effective way to build a computer that offers more performance than that achieved with a single-chip microprocessor is by building a multiprocessor the or a cluster at leverages the significant price/performance advantages of mass-produced microprocessors.

3. Multiprocessors are highly effective for multiprogrammed workloads, which are often the dominant use of mainframes and large servers, as well as for file servers or web servers, which are effectively a restricted type of parallel workload. In the future, such workloads may well constitute a large portion of the market for higher-performance multiprocessors. When a workload wants to share resources, such as file storage, or can efficiently timeshare a resource, such as a large memory, a multiprocessor can be a very efficient host. Further-

more, the OS software needed to efficiently execute multiprogrammed workloads is commonplace.

4. More recently, multiprocessors have proved very effective for certain intensive commercial workloads, such as OLTP (assuming the system supports enough I/O to be CPU-limited), DSS applications (where query optimization is critical), and large-scale, web searching applications. For commercial applications with undemanding communication requirements, little need for very large memories (typically used to cache databases), or limited demand for computation, clusters are likely to be more cost-effective than multiprocessors. The commercial space is currently a mix of clusters of basic PCs, SMPs, and clustered SMPs with different architectural styles appearing to hold some lead in different application spaces.

5. On-chip multiprocessing appears to be growing in importance for two reasons. First, in the embedded market where natural parallelism often exists, such approaches are an obvious alternative to faster, and possibly less silicon efficient, processors. Second, diminishing returns in high-end microprocessor design will encourage designers to pursue on-chip multiprocessing as a potentially more cost-effective direction. We explore the challenges to this direction at the end of this section.

Although there is reason to be optimistic about the growing importance of multiprocessors, many areas of parallel architecture remain unclear. Two particularly important questions are, How will the largest-scale multiprocessors (the massively parallel processors, or MPPs) be built? and What is the role of multiprocessing as a long-term alternative to higher-performance uniprocessors?

The Future of MPP Architecture

Hennessy and Patterson should move MPPs to Chapter 11.

> Jim Gray, when asked about coverage of MPPs in the second edition of this book, alludes to Chapter 11 bankruptcy protection in U.S. law (1995)

Small-scale multiprocessors built using snooping-bus schemes are extremely cost-effective. Microprocessors traditionally have even included much of the logic for cache coherence in the processor chip, and several allow the buses of two or more processors to be directly connected—implementing a coherent bus with no additional logic. With modern integration levels, multiple processors can be placed on a board, on a single multi-chip module (MCM), or even within a single die (as we saw in Section 6.13) resulting in a highly cost-effective multiprocessor. Recent microprocessors have been including support for DSM approaches,

making it possible to connect small to moderate numbers of processors with little overhead. It is premature to predict that such architectures will dominate the middle range of processor counts (16–64), but it appears at the present that this approach is the most attractive.

What is totally unclear at the present is how the very largest multiprocessors will be constructed. The difficulties that designers face include the relatively small market for very large multiprocessors (> 64 nodes and often > $5 million) and the need for multiprocessors that scale to larger processor counts to be extremely cost-effective at the lower processor counts where most of the multiprocessors will be sold. At the present there appear to be four slightly different alternatives for large-scale multiprocessors:

1. Large-scale multiprocessors that simply scale up naturally, using proprietary interconnect and communications controller technology. This approach has been followed in multiprocessors like the Cray T3E and the SGI Origin. There are two primary difficulties with such designs. First, the multiprocessors are not cost-effective at small scales, where the cost of scalability is not valued. Second, these multiprocessors have programming models that are incompatible, in varying degrees, with the mainstream of smaller and midrange multiprocessors.

2. Large-scale multiprocessors constructed from clusters of midrange multiprocessors with combinations of proprietary and standard technologies to interconnect such multiprocessors. The Wildfire design is just such a system. This cluster approach gets its cost-effectiveness through the use of cost-optimized building blocks. In some approaches, the basic architectural model (e.g., coherent shared memory) is extended. Many companies offer a high-end version of such a machine including HP, Sun, and SGI. Due to the two-level nature of the design, the programming model sometimes must be changed from shared memory to message passing or to a different variation on shared memory, among clusters. The migration and replication features in Wildfire offer a way to minimize this disadvantage. This class of machines has made important inroads, especially in commercial applications.

3. Designing clustered multicomputers that use off-the-shelf uniprocessor nodes and a custom interconnect. The advantage of such a design is the cost-effectiveness of the standard uniprocessor node, which is often a repackaged workstation; the disadvantage is that the programming model will probably need to be message passing even at very small node counts. In some application environments where little or no sharing occurs, this may be acceptable. In addition, the cost of the interconnect, because it is custom, can be significant, making the multiprocessor costly, especially at small node counts. The IBM SP-2 is the best example of this approach today.

4. Designing a cluster using *all* off-the-shelf components, which promises the

lowest cost. The leverage in this approach lies in the use of commodity technology everywhere: in the processors (PC or workstation nodes), in the interconnect (high-speed local area network technology, such as ATM or Gigabit Ethernet), and in the software (standard operating systems and programming languages). Of course, such multiprocessors will use message passing, and communication is likely to have higher latency and lower bandwidth than in the alternative designs. Like the previous class of designs, for applications that do not need high bandwidth or low-latency communication, this approach can be extremely cost-effective. Web servers, for example, may be a good match to these multicomputers, as we saw for the Google cluster in Chapter 8.

Each of these approaches has advantages and disadvantages, and the importance of the shortcomings of any one approach are dependent on the application class. In 2000 it is unclear which if any of these models will win out for larger-scale multiprocessors, although the growth of the market for web servers has made "racks of PCs" the dominant form at least by processor count. It is likely that the current bifurcation by market and scale will continue for some time, although in some area a hybridization of these ideas may emerge, given the similarity in several of the approaches.

The Future of Microprocessor Architecture

As we saw in Chapters 3 and 4, architects are using ever more complex techniques to try to exploit more instruction-level parallelism. As we also saw in that chapter, the prospects for finding ever-increasing amounts of instruction-level parallelism in a manner that is efficient to exploit are somewhat limited. Likewise, there are increasingly difficult problems to be overcome in building memory hierarchies for high-performance processors. Of course, continued technology improvements will allow us to continue to advance clock rate. But the use of technology improvements that allow a faster gate speed alone is not sufficient to maintain the incredible growth of performance that the industry has experienced for over 15 years. Maintaining a rapid rate of performance growth will depend to an increasing extent on exploiting the dramatic growth in effective silicon area, which will continue to grow much faster than the basic speed of the process technology.

Unfortunately, for almost ten years, increases in performance have come at the cost of ever-increasing inefficiencies in the use of silicon area, external connections, and power. This diminishing-returns phenomenon has only recently (as of 2001) appeared to have slowed the rate of performance growth. Whether or not this is slowdown temporary is unclear. What is clear, is that we cannot sustain the rapid rate of performance improvements without significant new innovations in computer architecture.

Unlike the prophets quoted at the beginning of the chapter, your authors do not believe that we are about to "hit a brick wall" in our attempts to improve single-

processor performance. Instead, we may see a gradual slowdown in performance growth, especially for integer performance, with the eventual growth being limited primarily by improvements in the speed of the technology. When these limitation will become serious is hard to say, but possibly as early as 2005 and likely by 2010. Even if such a slowdown were to occur, performance might well be expected to grow at the annual rate of 1.35 that we saw prior to 1985 at least until fundamental limitations in silicon are become serious in th 2015 time frame.

Furthermore, we do not want to rule out the possibility of a breakthrough in uniprocessor design. In the early 1980s, many people predicted the end of growth in uniprocessor performance, only to see the arrival of RISC technology and an unprecedented 15-year growth in performance averaging 1.5 times per year!

With this in mind, we cautiously ask whether the long-term direction will be to use increased silicon to build multiple processors on a single chip. Such a direction is appealing from the architecture viewpoint—it offers a way to scale performance without increasing hardware complexity. It also offers an approach to easing some of the challenges in memory-system design, since a distributed memory can be used to scale bandwidth while maintaining low latency for local accesses. The challenge lies in software and in what architecture innovations may be used to make the software easier.

In 2000, IBM announced the first commercial chips with two general-purpose processors on a single die, the Power4 processor. Each Power4 contains two Power3 microprocessors, a shared secondary cache, an interface to an off-chip tertiary cache or main memory, and chip-to-chip communication system, which allows a four processor cross-bar connected module to be built with no additional logic. Using 4 Power4 chips and the appropriate DRAMS, an eight-processor system can be integrated onto a board about 8 inches on a side. The board would contain 700 million transistors, not including the third level cache or main memory, and would have a peak instruction execution rate of 32 billion instructions per second!

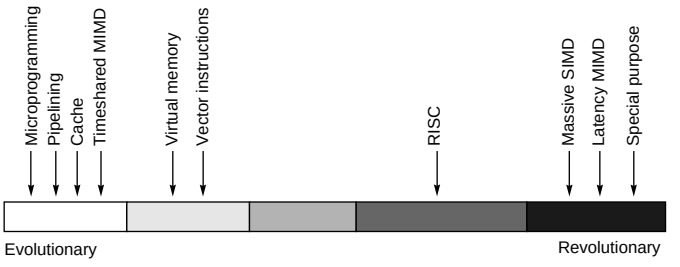
Evolution Versus Revolution and the Challenges to Paradigm Shifts in the Computer Industry

Figure 6.54 shows what we mean by the *evolution-revolution spectrum* of computer architecture innovation. To the left are ideas that are invisible to the user (presumably excepting better cost, better performance, or both) and are at the evolutionary end of the spectrum. At the other end are revolutionary architecture ideas. These are the ideas that require new applications from programmers who must learn new programming languages and models of computation, and must invent new data structures and algorithms.

Revolutionary ideas are easier to get excited about than evolutionary ideas, but to be adopted they must have a much higher payoff. Caches are an example of an evolutionary improvement. Within 5 years after the first publication about caches, almost every computer company was designing a computer with a cache. The

RISC ideas were nearer to the middle of the spectrum, for it took more than eight years for most companies to have a RISC product and more than fifteen year for the last hold out to announce their product. Most multiprocessors have tended to the revolutionary end of the spectrum, with the largest-scale multiprocessors (MPPs) being more revolutionary than others. Most programs written to use multiprocessors as parallel engines have been written especially for that class of multiprocessors, if not for the specific architecture.

The challenge for both hardware and software designers that would propose that multiprocessors and parallel processing become the norm, rather than the exception, is the disruption to the established base of programs. There are two possible ways this paradigm shift could be facilitated: if parallel processing offers the only alternative to enhance performance, and if advances in hardware and software technology can construct a gentle ramp that allows the movement to parallel processing, at least with small numbers of processors, to be more evolutionary.



User compatibility	Binary	Upward binary	Assembly	High-level language	New programs, extended or new HLL, new algorithms
Example	VAX-11/780 vs. 8800	Intel 8086 vs. 80286 vs. 80386 vs. 80486	MIPS 1000 vs. DEC 3100	Sun 3 vs. Sun 4	SISD vs. Intel Paragon
Difference	Microcode, TLB, caches, pipelining, MIMD	Some new instructions	Byte order (Big vs. Little Endian)	Full instruction set (same data representation)	Algorithms, extended HLL, programs

FIGURE 6.54 The evolution-revolution spectrum of computer architecture. The second through fourth columns are distinguished from the final column in that applications and operating systems can be ported from other computers rather than written from scratch. For example, RISC is listed in the middle of the spectrum because user compatibility is only at the level of high-level languages, while microprogramming allows binary compatibility, and latency-oriented MIMDs require changes to algorithms and extending HLLs. Timeshared MIMD means MIMDs justified by running many independent programs at once, while latency MIMD means MIMDs intended to run a single program faster.

6.16 Historical Perspective and References

There is a tremendous amount of history in parallel processing; in this section we divide our discussion by both time period and architecture. We start with the SIMD approach and the Illiac IV. We then turn to a short discussion of some other early experimental multiprocessors and progress to a discussion of some of the great debates in parallel processing. Next we discuss the historical roots of the present multiprocessors and conclude by discussing recent advances.

SIMD Computers: Several Attempts, No Lasting Successes

The cost of a general multiprocessor is, however, very high and further design options were considered which would decrease the cost without seriously degrading the power or efficiency of the system. The options consist of recentralizing one of the three major components.... Centralizing the [control unit] gives rise to the basic organization of [an]... array processor such as the Illiac IV.

Bouknight et al. [1972]

The SIMD model was one of the earliest models of parallel computing, dating back to the first large-scale multiprocessor, the Illiac IV. The key idea in that multiprocessor, as in more recent SIMD multiprocessors, is to have a single instruction that operates on many data items at once, using many functional units.

The earliest ideas on SIMD-style computers are from Unger [1958] and Slotnick, Borck, and McReynolds [1962]. Slotnick's Solomon design formed the basis of the Illiac IV, perhaps the most infamous of the supercomputer projects. Although successful in pushing several technologies that proved useful in later projects, it failed as a computer. Costs escalated from the $8 million estimate in 1966 to $31 million by 1972, despite construction of only a quarter of the planned multiprocessor. Actual performance was at best 15 MFLOPS, versus initial predictions of 1000 MFLOPS for the full system [Hord 1982]. Delivered to NASA Ames Research in 1972, the computer took three more years of engineering before it was usable. These events slowed investigation of SIMD, with Danny Hillis [1985] resuscitating this style in the Connection Machine, which had 65,636 1-bit processors.

Real SIMD computers need to have a mixture of SISD and SIMD instructions. There is an SISD host computer to perform operations such as branches and address calculations that do not need parallel operation. The SIMD instructions are broadcast to all the execution units, each of which has its own set of registers. For flexibility, individual execution units can be disabled during a SIMD instruction.

In addition, massively parallel SIMD multiprocessors rely on interconnection or communication networks to exchange data between processing elements.

SIMD works best in dealing with arrays in for-loops. Hence, to have the opportunity for massive parallelism in SIMD there must be massive amounts of data, or *data parallelism*. SIMD is at its weakest in case statements, where each execution unit must perform a different operation on its data, depending on what data it has. The execution units with the wrong data are disabled so that the proper units can continue. Such situations essentially run at 1/*n*th performance, where *n* is the number of cases.

The basic trade-off in SIMD multiprocessors is performance of a processor versus number of processors. Recent multiprocessors emphasize a large degree of parallelism over performance of the individual processors. The Connection Multiprocessor 2, for example, offered 65,536 single bit-wide processors, while the Illiac IV had 64 64-bit processors.

After being resurrected in the 1980s, first by Thinking Machines and then by MasPar, the SIMD model has once again been put to bed as a general-purpose multiprocessor architecture, for two main reasons. First, it is too inflexible. A number of important problems cannot use such a style of multiprocessor, and the architecture does not scale down in a competitive fashion; that is, small-scale SIMD multiprocessors often have worse cost/performance compared with that of the alternatives. Second, SIMD cannot take advantage of the tremendous performance and cost advantages of microprocessor technology. Instead of leveraging this low-cost technology, designers of SIMD multiprocessors must build custom processors for their multiprocessors.

Although SIMD computers have departed from the scene as general-purpose alternatives, this style of architecture will continue to have a role in special-purpose designs. Many special-purpose tasks are highly data parallel and require a limited set of functional units. Thus designers can build in support for certain operations, as well as hardwire interconnection paths among functional units. Such organizations are often called *array processors*, and they are useful for tasks like image and signal processing.

Other Early Experiments

It is difficult to distinguish the first MIMD multiprocessor. Surprisingly, the first computer from the Eckert-Mauchly Corporation, for example, had duplicate units to improve availability. Holland [1959] gave early arguments for multiple processors.

Two of the best-documented multiprocessor projects were undertaken in the 1970s at Carnegie Mellon University. The first of these was C.mmp [Wulf and Bell 1972; Wulf and Harbison 1978], which consisted of 16 PDP-11s connected by a crossbar switch to 16 memory units. It was among the first multiprocessors with more than a few processors, and it had a shared-memory programming model. Much of the focus of the research in the C.mmp project was on software, espe-

cially in the OS area. A later multiprocessor, Cm* [Swan et al. 1977], was a cluster-based multiprocessor with a distributed memory and a nonuniform access time. The absence of caches and a long remote access latency made data placement critical. This multiprocessor and a number of application experiments are well described by Gehringer, Siewiorek, and Segall [1987]. Many of the ideas in these multiprocessors would be reused in the 1980s when the microprocessor made it much cheaper to build multiprocessors.

Great Debates in Parallel Processing

The quotes at the beginning of this chapter give the classic arguments for abandoning the current form of computing, and Amdahl [1967] gave the classic reply in support of continued focus on the IBM 370 architecture. Arguments for the advantages of parallel execution can be traced back to the 19th century [Menabrea 1842]! Yet the effectiveness of the multiprocessor for reducing latency of individual important programs is still being explored. Aside from these debates about the advantages and limitations of parallelism, several hot debates have focused on how to build multiprocessors.

Predictions of the Future

It's hard to predict the future, yet in 1989 Gordon Bell made two predictions for 1995. We included these predictions in the first edition of the book, when the outcome was completely unclear. We discuss them in this section, together with an assessment of the accuracy of the prediction.

The first is that a computer capable of sustaining a teraFLOPS—one million MFLOPS—will be constructed by 1995, either using a multicomputer with 4K to 32K nodes or a Connection Multiprocessor with several million processing elements [Bell 1989]. To put this prediction in perspective, each year the Gordon Bell Prize acknowledges advances in parallelism, including the fastest real program (highest MFLOPS). In 1989 the winner used an eight-processor Cray Y-MP to run at 1680 MFLOPS. On the basis of these numbers, multiprocessors and programs would have to have improved by a factor of 3.6 each year for the fastest program to achieve 1 TFLOPS in 1995. In 1999, the first Gordon Bell prize winner crossed the 1 TF bar, using a 5,832 processor IBM RS/6000 SST system designed specially for Livermore Laboratories, they achieved 1.18 Teraflops on a shock-wave simulation. This ratio represents a year-to-year improvement of 1.93, which is still quite impressive.

What has become recognized since 1989 is that although we may have the technology to build a teraFLOPS multiprocessor, it is not clear that the machine is cost-effective, except perhaps for a few very specialized and critically important application related to national security. Your authors estimated in 1990 that to achieve 1 TF would require a machine with about 5,000 processors and would cost about $100 million. The 5,832 processor IBM system at Livermore cost

$110 million. As might be expected, improvements in the performance of individual microprocessors both in cost and performance directly affect the cost and performance of large-scale multiprocessors, but a 5000 processor system will cost more than 5000 times the price of a desktop system using the same processor.

The second Bell prediction concerned the number of data streams in supercomputers shipped in 1995. Danny Hillis believed that although supercomputers with a small number of data streams may be the best sellers, the biggest multiprocessors will be multiprocessors with many data streams, and these will perform the bulk of the computations. Bell bet Hillis that in the last quarter of calendar year 1995 more sustained MFLOPS will be shipped in multiprocessors using few data streams (≤100) rather than many data streams (≥1000). This bet concerned only supercomputers, defined as multiprocessors costing more than $1 million and used for scientific applications. Sustained MFLOPS was defined for this bet as the number of floating-point operations per *month*, so availability of multiprocessors affects their rating.

In 1989, when this bet was made, it was totally unclear who would win. In 1995, a survey of the current publicly known supercomputers showed only six multiprocessors in existence in the world with more than 1000 data streams, so Bell's prediction was a clear winner. In fact, in 1995, much smaller microprocessor-based multiprocessors (≤ 20 processors) were becoming dominant. 1n 1995, a survey of the 500 highest-performance multiprocessors in use (based on Linpack ratings), called the Top 500, showed that the largest number of multiprocessors were bus-based shared-memory multiprocessors! By 2000, the picture had become less clear: the top four vendors were IBM (144 SP systems), Sun (121 Enterprise systems), SGI (62 Origin systems), and Cray (54 T3E systems). Although IBM holds the largest number of spots, almost all the other systems on the TOP 500 list are shared-memory systems or clusters of such systems.

More Recent Advances and Developments

With the primary exception of the parallel vector multiprocessors (see Appendix B), all other recent MIMD computers have been built from off-the-shelf microprocessors using a bus and logically central memory or an interconnection network and a distributed memory. A number of experimental multiprocessors built in the 1980s further refined and enhanced the concepts that form the basis for many of today's multiprocessors.

The Development of Bus-Based Coherent Multiprocessors

Although very large mainframes were built with multiple processors in the 1970s, multiprocessors did not become highly successful until the 1980s. Bell [1985] suggests the key was that the smaller size of the microprocessor allowed the memory bus to replace the interconnection network hardware, and that porta-

ble operating systems meant that multiprocessor projects no longer required the invention of a new operating system. In this paper, Bell defines the terms *multiprocessor* and *multicomputer* and sets the stage for two different approaches to building larger-scale multiprocessors.

The first bus-based multiprocessor with snooping caches was the Synapse N+1 described by Frank [1984]. Goodman [1983] wrote one of the first papers to describe snooping caches. The late 1980s saw the introduction of many commercial bus-based, snooping-cache architectures, including the Silicon Graphics 4D/240 [Baskett et al. 1988], the Encore Multimax [Wilson 1987], and the Sequent Symmetry [Lovett and Thakkar 1988]. The mid 1980s saw an explosion in the development of alternative coherence protocols, and Archibald and Baer [1986] provide a good survey and analysis, as well as references to the original papers. Figure 6.55 summarizes several snooping cache-coherence protocols and shows some multiprocessors that have used or are using that protocol.

Name	Protocol type	Memory-write policy	Unique feature	Multiprocessors using
Write Once	Write invalidate	Write back after first write	First snooping protocol described in literature	
Synapse N+1	Write invalidate	Write back	Explicit state where memory is the owner	Synapse multiprocessors; first cache-coherent multiprocessors available
Berkeley (MOESI)	Write invalidate	Write back	Owned shared state	Berkeley SPUR multiprocessor; SUN Enterprise servers
Illinois (MESI)	Write invalidate	Write back	Clean private state; can supply data from any cache with a clean copy	SGI Power and Challenge series
"Firefly"	Write broadcast	Write back when private, write through when shared	Memory updated on broadcast	No current multiprocessors; SPARCCenter 2000 closest.

FIGURE 6.55 Five snooping protocols summarized. Archibald and Baer [1986] use these names to describe the five protocols, and Eggers [1989] summarizes the similarities and differences as shown in this figure. The Firefly protocol was named for the experimental DEC Firefly multiprocessor, in which it appeared. The alternative names for protocols are based on the states they support: M=Modified, E=Exclusive (shared clean), S=Shared, I=Invalid, O=Owner (shared dirty).

The early 1990s saw the beginning of an expansion of such systems with the use of very wide, high speed buses (the SGI Challenge system used a 256-bit, packet-oriented bus supporting up to 8 processor boards and 32 processors) and later, the use of multiple buses and crossbar interconnects, e.g. in the SUN SPARCCenter and Enterprise systems (Charlesworth [1998] discusses the interconnect architecture of these multiprocessors). In 2001, the Sun Enterprise serv-

ers represent the primary example of large-scale (> 16 processors), symmetric multiprocessors in active use.

Toward Large-Scale Multiprocessors

In the effort to build large-scale multiprocessors, two different directions were explored: message passing multicomputers and scalable shared-memory multiprocessors. Although there had been many attempts to build mesh and hypercube-connected multiprocessors, one of the first multiprocessors to successfully bring together all the pieces was the Cosmic Cube built at Caltech [Seitz 1985]. It introduced important advances in routing and interconnect technology and substantially reduced the cost of the interconnect, which helped make the multicomputer viable. The Intel iPSC 860, a hypercube-connected collection of i860s, was based on these ideas. More recent multiprocessors, such as the Intel Paragon, have used networks with lower dimensionality and higher individual links. The Paragon also employed a separate i860 as a communications controller in each node, although a number of users have found it better to use both i860 processors for computation as well as communication. The Thinking Multiprocessors CM-5 made use of off-the-shelf microprocessors and a fat tree interconnect (see Chapter 7). It provided user-level access to the communication channel, thus significantly improving communication latency. In 1995, these two multiprocessors represent the state of the art in message-passing multicomputers.

Early attempts at building a scalable shared-memory multiprocessor include the IBM RP3 [Pfister et al. 1985], the NYU Ultracomputer [Schwartz 1980; Elder et al. 1985], the University of Illinois Cedar project [Gajksi et al. 1983], and the BBN Butterfly and Monarch [BBN Laboratories 1986; Rettberg et al. 1990]. These multiprocessors all provided variations on a nonuniform distributed-memory model (and hence are distributed shared memory or DSM multiprocessors), but did not support cache coherence, which substantially complicated programming. The RP3 and Ultracomputer projects both explored new ideas in synchronization (fetch-and-operate) as well as the idea of combining references in the network. In all four multiprocessors, the interconnect networks turned out to be more costly than the processing nodes, raising problems for smaller versions of the multiprocessor. The Cray T3D/E (see Arpaci et. al. [1995] for an evaluation of the T3D and Scott [1996] for a description of the T3E enhancements) builds on these ideas, using a noncoherent shared address space but building on the advances in interconnect technology developed in the multicomputer domain (see Scott and Thorson [1996]).

Extending the shared-memory model with scalable cache coherence was done by combining a number of ideas. Directory-based techniques for cache coherence were actually known before snooping cache techniques. In fact, the first cache-coherence protocols actually used directories, as described by Tang [1976] and implemented in the IBM 3081. Censier and Feautrier [1978] described a directory coherence scheme with tags in memory. The idea of distributing directories

with the memories to obtain a scalable implementation of cache coherence was first described by Agarwal et al. [1988] and served as the basis for the Stanford DASH multiprocessor (see Lenoski et al. [1990, 1992]), which was the first operational cache-coherent DSM multiprocessor. DASH was a "plump" node cc-NUMA machine that used 4-processor SMPs as its nodes; interconnecting them in a style similar to that of Wildfire but using a more scalable 2-dimension al grid rather than a crossbar for the interconnect.

The Kendall Square Research KSR-1 [Burkhardt et al. 1992] was the first commercial implementation of scalable coherent shared memory. It extended the basic DSM approach to implement a concept called *COMA* (*cache-only memory architecture*), which makes the main memory a cache. Like the Wildfire CMR scheme, in the KSR-1 memory blocks could be replicated in the main memories of each node with hardware support to handle the additional coherence requirements for these replicated blocks. (The KSR-1 was not strictly a pure COMA because it did not migrate the home location of a data item, but always kept a copy at home. Essentially, it implemented only replication.)

In parallel, researchers at the Swedish Institute for Computer Science [Hagersten et. al. 1992.] developed a concept called DDM (for Data Diffusion Machine) which is a true COMA, since all memory operates as a cache, and a memory block does not exist in a predefined node. The absence of a designated home for a memory block significantly complicates the protocols, since it means that there is no static look-up scheme to find the location and status of a block. Furthermore, a true COMA must contend with the problem of finding a place to move a memory block when it conflicts with another block for the same location in memory (which happens because the memory is a cache with a limited associativity). In the event that the displaced block is the last copy of a memory block, which in itself may be difficult to know precisely, the displaced block must be migrated to some other memory location, since it cannot be destroyed (as it is the only copy of the data). This migration process can be very complex requiring a potentially unbounded number of memory blocks to be displaced!

Although no pure COMA machines were ever built, the COMA idea has inspired many variations. COMA-F, or FLAT COMA was proposed by Stenström, Joe, and Gupta in 1992 as a simpler alternative to the original COMA proposals. By allocating a home location COMA-F eliminated the need for multilevel hierarchical look-ups and possible displacement misses, since the block status could always be looked up in the home and the home location always had space for the block. In 1995, Saulsbury et. al. proposed Simple COMA (S-COMA), which implemented COMA using the virtual memory mechanisms for replication and migration, rather than hardware support at the cache-level. Reactive NUMA [Falsafi and Wood 1997] is a proposal to develop a protocol that merges the best of CC-NUMA protocols with S-COMA protocols. At the same time, several groups (see Chandra et. al. 1994 and Soundararajan 1996] explored the use of page-level replication and migration, both to assist in reducing remote misses and as an alternative to other schemes such as strict COMA or remote access caches. Wildfire

builds on many of these ideas to create a blend of hardware and software mechanisms.

The Convex Exemplar implemented scalable coherent shared memory using a two-level architecture: at the lowest level eight-processor modules are built using a crossbar. A ring can then connect up to 32 of these modules, for a total of 256 processors (see Thekkath et. al. [1997] for an evaluation). Lenoski and Laudon [1997] describe the SGI Origin, which was first delivered in 1996 and is closely based on the original Stanford DASH machine, though including a number of innovations for scalability and ease of programming. Origin uses a bit-vector for the directory structure, which is either 16 or 32 bits long. Each bit represents a node, which consists of two processors; a coarse bit vector representation allows each bit to represent up to 8 nodes for a total of 1,024 processors. As Galles [1996] describes, a high performance fat hypercube is used for the global interconnect. Hristea et. al [1997] is a thorough evaluation of the performance of the Origin memory system.

More recent research has focused on enhanced scalability for cache-coherent designs, flexible and adaptable techniques for implementing coherency, and approaches that merge hardware and software schemes. The MIT Alewife machine [Agarwal et. al. 1995] incorporated several innovations including processor support for multithreading and the use of cooperative mechanisms for handling coherence. The Stanford FLASH multiprocessor [Kuskin et. al. 1994, Gibson et. al. 2000] makes use of a programmable processor that implements the coherence scheme, as well as alternative schemes for message-passing, synchronization primitives, or performance instrumentation. Reinhadt and his colleagues at the University of Wisconsin [1994] explored an alternative for a combination of user and-base software and hardware support for coherent shared-memory. The Star-T [Nikhil et. al 1992] and Star-T Voyager [Ang, et. al. 1998] projects at MIT explored the use of multithreading and combining customized and commodity approaches to building scalable multiprocessors.

Developments in Synchronization and Consistency Models

A wide variety of synchronization primitives have been proposed for shared-memory multiprocessors. Mellor-Crummey and Scott [1991] provide an overview of the issues as well as efficient implementations of important primitives, such as locks and barriers. An extensive bibliography supplies references to other important contributions, including developments in spin locks, queuing locks, and barriers.

Lamport [1979] introduced the concept of sequential consistency and what correct execution of parallel programs means. Dubois, Scheurich, and Briggs [1988] introduced the idea of weak ordering (originally in 1986). In 1990, Adve and Hill provided a better definition of weak ordering and also defined the concept of data-race-free; at the same conference, Gharachorloo [1990] and his colleagues introduced release consistency and provided the first data on the

performance of relaxed consistency models. More relaxed consistency models have been widely adopted in microprocessor architectures, including the Sun SPARC, Alpha, and IA-64. Adve and Gharachorloo [1996] is an excellent tutorial on memory consistency and the differences among these models.

Other References

The concept of using virtual memory to implement a shared address space among distinct machines was pioneered in Kai Li's Ivy system in 1988. There have been subsequent papers exploring both hardware support issues, software mechanisms, and programming issues. Amza et. al. [1996] describe a system built on workstations using a new consistency model, L. Kontothanassis, et. al. [1997] describe a software shared memory scheme using remote writes, and Erlichson et. al. [1996] describe the use of shared virtual memory to build large-scale multiprocessors using SMPs as nodes.

There is an almost unbounded amount of information on multiprocessors and multicomputers: Conferences, journal papers, and even books seem to appear faster than any single person can absorb the ideas. No doubt many of these papers will go unnoticed—not unlike the past. Most of the major architecture conferences contain papers on multiprocessors. An annual conference, *Supercomputing XY* (where X and Y are the last two digits of the year), brings together users, architects, software developers, and vendors and publishes the proceedings in book, CD-ROM, and online (see www.scXY.org) form. Two major journals, *Journal of Parallel and Distributed Computing* and the *IEEE Transactions on Parallel and Distributed Systems*, contain papers on all aspects of parallel processing. Several books focusing on parallel processing are included in the following references with Culler, Singh, and Gupta [1999] being the most recent, large-scale effort. For years, Eugene Miya of NASA Ames has collected an online bibliography of parallel-processing papers. The bibliography, which now contains that contains more than 35,000 entries, is available online at:

http://liinwww.ira.uka.de/bibliography/Parallel/Eugene/index.html.

In addition to documenting the discovery of concepts now used in practice, these references also provide descriptions of many ideas that have been explored and found wanting, as well as ideas whose time has just not yet come.

Multithreading and Simultaneous Multithreading

The concept of multithreading dates back to one of the earliest transistorized computers, the TX-2. TX-2 was one of the earliest transistorized computers and is also famous for being the computer on which Ivan Sutherland created Sketchpad, the first computer graphics system. TX-2 was built at MIT's Lincoln Laboratory and became operational in 1959. It used multiple threads to support fast context switching to handle I/O functions. Clark [1957] describes the basic architecture and Forgie [1957] describes the I/O architecture. Multithreading was also

used in the CDC 6600, where a fine-grained multithreading scheme with interleaved scheduling among threads was used as the architecture of the I/O processors. The HEP processor, a pipelined multiprocessor, designed by Denelcor. and shipped in 1982 used fine-grained multithreading to hide the pipeline latency as well as to hide the latency to a large memory shared among all the processors. Because the HEP had no cache, this hiding of memory latency was critical. Burton Smith, one the primary architects, describes the HEP architecture in a 1978 paper and Jordan [1983] published a performance evaluation. The Tera processor extends the multithreading ideas and is described by Alverson et. al. in a 1992 paper.

In the late 1980s and early 1990s, researchers explored the concept of coarse-grained (also called block multithreading), as a way to tolerate latency, especially in multiprocessor environments. The SPARCLE processor in the Alewife system used such a scheme, switching threads whenever a high latency exceptional event, such as a long cache miss, occurred. Agarwal et. al. describe SPARCLE in a 1993 paper. The IBM Pulsar processor uses similar ideas.

By the early 1990s, several research groups had arrived at two key insights. First, they realized that fine-grained multithreading was needed to get the maximum performance benefit, since in a coarse-grained approach, the overhead of thread switching and thread start-up (e.g., filling the pipeline from the new thread) negated much of the performance advantage (see Laudon et. al. 1994). Second, several groups realized that to effectively use large numbers of functional units would require both ILP and thread-level parallelism (TLP). These insights led to several architectures that used combinations of multithreading and multiple issue. Wolfe and Shen [1991] describe an architecture called XIMD that statically interleaves threads scheduled for a VLIW processor. Mirata et. al. (1992) describe a proposed processor for media use that combines a static superscalar pipeline with support for multithreading; they report speed-ups from combining both forms of parallelism. Keckler and Dally [1992] combine static scheduling of ILP nd dynamic scheduling of threads combing the two forms for a processor with multiple functional units. The question of how to balance the allocation of functional units between ILP and TLP and how to schedule the two forms of parallelism remained open.

When it became clear in the middle of the 1990s that dynamically-scheduled superscalars would be delivered shortly, several research groups proposed using the dynamic scheduling capability to mix instructions from several threads on the fly. Yamamoto, Searing, Talcott, Wood, and Nemirosky [1994] appears to be the first such proposal, though the simulation results for their multithreaded superscalar architecture use simplistic assumptions. This work was quickly followed by Tullsen, Eggers, and Levy [1995], which was the first realistic simulation assessment and coined the name simultaneous multithreading. Subsequent work by the same group together with industrial coauthors addressed many of the open questions about SMT. For example, Tullsen et. al. [1996] addressed questions about the challenges of scheduling ILP vs. TLP. Lo et. al. [1997] is an extensive

discussion of the SMT concept and an evaluation of its performance potential, and Lo et. al. [1998] evaluates database performance on an SMT processor.

References

A. AGARWAL, A., KUBIATOWICZ, J., KRANZ, D., LIM, B.-H, YEUNG, D., D'SOUZA, G. AND M. PARKIN [1993], "Sparcle: An evolutionary processor design for large-scale multiprocessors," IEEE Micro 13 (June), pp. 48--61.

ALVERSON, G. ALVERSON, R., CALLAHAN, D. , KOBLENZ, B., PORTERFIELD, A. AND B. SMITH [1992]. "Exploiting heterogeneous parallelism on a multithreaded multiprocessor," Proc. 1992 International Conf. on Supercomputing (November) , pp. 188--197.

ADVE, S. V. AND K. GHARACHORLOO[1996]. "Shared Memory Consistency Models: A Tutorial," *IEEE Computer* 29:12 (December), 66–76.

ADVE, S. V. AND M. D. HILL [1990]. "Weak ordering—A new definition," *Proc. 17th Int'l Symposium on Computer Architecture* (June), Seattle, 2–14.

AGARWAL, A., BIANCHINI, R., CHAIKEN, D., JOHNSON, K., AND D. KRANZ [1995]."THE MIT ALEWIFE MACHINE: ARCHITECTURE AND PERFORMANCE", INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, DENVER, JUNE, 2–13

AGARWAL, A., J. L. HENNESSY, R. SIMONI, AND M.A. HOROWITZ [1988]. "An evaluation of directory schemes for cache coherence," *Proc. 15th Int'l Symposium on Computer Architecture* (June), 280–289.

ALMASI, G. S. AND A. GOTTLIEB [1989]. *Highly Parallel Computing*, Benjamin/Cummings, Redwood City, Calif.

AMDAHL, G. M. [1967]. "Validity of the single processor approach to achieving large scale computing capabilities," *Proc. AFIPS Spring Joint Computer Conf.* 30, Atlantic City, N.J. (April), 483–485.

AMZA C., COX, A. L., DWARKADAS, S., KELEHER, P., LU, H., RAJAMONY, R., YU, W. AND W. ZWAENEPOEL.[1996]. "TREADMARKS: SHARED MEMORY COMPUTING ON NETWORKS OF WORKSTATIONS". IEEE COMPUTER, 29(2) (FEBRUARY), 18–28.

ANG, B., CHIOU, D., ROSENBAND, D., EHRLICH, M., AND RUDOLPH, L., AND ARVIND [1998]. "STARTT-VOYAGER: A FLEXIBLE PLATFORM FOR EXPLORING SCALABLE SMP ISSUES", PROCEEDINGS OF SC'98, ORLANDO, FLORIDA, NOV.

ARCHIBALD, J. AND J.-L. BAER [1986]. "Cache coherence protocols: Evaluation using a multiprocessor simulation model," *ACM Trans. on Computer Systems* 4:4 (November), 273–298.

ARPACI, R.H., CULLER, D.E., KRISHNAMURTHY, A., STEINBERG, S.G. AND K. YELICK [1995]."Empirical evaluation of the CRAY-T3D: A compiler perspective," Proceedings of the International Symposium on Computer Architecture, Denver (June), pages 320-331.

BAER J-L. AND W-H. WANG [1988]. "On the Inclusion Properties for Multi-Level Cache Hierarchies." In Proceedings of the 15th Annual International Symposium on Computer Architecture, Honolulu, June, 73--80.

BARROSO, L.A., GHARACHORLOO, K. AND E. BUGNION [1998]. "Memory System Characterization of Commercial Workloads," Proceedings 25th International Symposium on Computer Architecture, Barcelona (July), 3-14.

BASKETT, F., T. JERMOLUK, AND D. SOLOMON [1988]. "The 4D-MP graphics superworkstation: Computing + graphics = 40 MIPS + 40 MFLOPS and 10,000 lighted polygons per second," *Proc. COMPCON Spring*, San Francisco, 468–471.

BBN LABORATORIES [1986]. "Butterfly parallel processor overview," Tech. Rep. 6148, BBN Labo-

ratories, Cambridge, Mass.

BELL, C. G. [1985]. "Multis: A new class of multiprocessor computers," *Science* 228 (April 26), 462–467.

BELL, C. G. [1989]. "The future of high performance computers in science and engineering," *Comm. ACM* 32:9 (September), 1091–1101.

BOUKNIGHT, W. J, S. A. DENEBERG, D. E. MCINTYRE, J. M. RANDALL, A. H. SAMEH, AND D. L. SLOTNICK [1972]. "The Illiac IV system," *Proc. IEEE* 60:4, 369–379. Also appears in D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples*, McGraw-Hill, New York (1982), 306–316.

BURKHARDT, H. III, S. FRANK, B. KNOBE, AND J. ROTHNIE [1992]. "Overview of the KSR1 computer system," Tech. Rep. KSR-TR-9202001, Kendall Square Research, Boston (February).

CENSIER, L. AND P. FEAUTRIER [1978]. "A new solution to coherence problems in multicache systems," *IEEE Trans. on Computers* C-27:12 (December), 1112–1118.

CHANDRA, R., DEVINE, S., VERGHESE, B., GUPTA, A. AND MENDEL ROSENBLUM [1994]. "Scheduling and Page Migration for Multiprocessor Compute Servers." In Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI). ACM,Santa Clara, CA. October, 12-24. .

CHARLESWORTH, A [1998]. "STARFIRE: EXTENDING THE SMP ENVELOPE," IEEE MICRO 18:1 (JAN/FEB), P 39-49.

CLARK. W.A. [1957]. "The Lincoln TX-2 Computer Development." Proceedings of the Western Joint Computer Conference (February), Institute of Radio Engineers, Los Angeles, 143-145.

.CULLER, D. E., SINGH, J. P., AND A. GUPTA [1999]. Parallel Computer Architecture A Hardware/Software Approach. Morgan Kaufmann Publishers,

1 EDITION, 1999.DUBOIS, M., C. SCHEURICH, AND F. BRIGGS [1988]. "Synchronization, coherence, and event ordering," *IEEE Computer* 9-21 (February).

EGGERS, S. [1989]. *Simulation Analysis of Data Sharing in Shared Memory Multiprocessors*, Ph.D. Thesis, Univ. of California, Berkeley. Computer Science Division Tech. Rep. UCB/CSD 89/501 (April).

ELDER, J., A. GOTTLIEB, C. K. KRUSKAL, K. P. MCAULIFFE, L. RANDOLPH, M. SNIR, P. TELLER, AND J. WILSON [1985]. "Issues related to MIMD shared-memory computers: The NYU Ultracomputer approach," *Proc. 12th Int'l Symposium on Computer Architecture* (June), Boston, 126–135.

ERLICHSON, A., NUCKOLLS, N., CHESSON, G. AND J. L. HENNESSY [1996]. "SoftFLASH: Analyzing the performance of clustered distributed virtual shared memory." In Proc. of the 7th Symp.on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII), pages 210--220, October.

FLYNN, M. J. [1966]. "Very high-speed computing systems," *Proc. IEEE* 54:12 (December), 1901–1909.

FALSAFI , B. ANDWOOD, D.A. [1997]. "Reactive NUMA: a design for unifying S-COMA and CC-NUMA," Proceedings of the 24th international symposium on Computer architecture, June, Denver, CO, 229-240.

FORGIE, J.W [1957]. "The Lincoln TX-2 Input-Output System," Proceedings of the Western Joint Computer Conference (February), Institute of Radio Engineers, Los Angeles, 156-160.

FRANK, S. J. [1984] "TIGHTLY COUPLED MULTIPROCESSOR SYSTEMS SPEED MEMORY ACCESS TIME," *ELECTRONICS* 57:1 (JANUARY), 164–169.

GALLES, M. [1996]. "Scalable Pipelined Interconnect for Distributed Endpoint Routing: The SGI SPIDER chip" . Proceedings Hot Interconnects '96, Stanford University, August.

GAJSKI, D., D. KUCK, D. LAWRIE, AND A. SAMEH [1983]. "CEDAR—A large scale multiprocessor," *Proc. Int'l Conf. on Parallel Processing* (August), 524–529.

GEHRINGER, E. F., D. P. SIEWIOREK, AND Z. SEGALL [1987]. *Parallel Processing: The Cm* Experience*, Digital Press, Bedford, Mass.

GHARACHORLOO, K., GUPTA, A., AND J.L. HENNESSY [1992]. “Hiding memory latency using dynamic scheduling in shared-memory multiprocessors.” In Proc. of the 19th Annual Int. Symp. on Computer Architecture, FGold Coast, Austrailia, June.

GHARACHORLOO, K., D. LENOSKI, J. LAUDON, P. GIBBONS, A. GUPTA, AND J. L. HENNESSY [1990]. “Memory consistency and event ordering in scalable shared-memory multiprocessors,” *Proc. 17th Int’l Symposium on Computer Architecture* (June), Seattle, 15–26.

GIBSON, J, KUNZ, R, OFELT, D, HOROWITZ,M, HENNESSY, J, AND M. HEINRICH [2000]. "FLASH vs. (Simulated) FLASH: Closing the Simulation Loop". Proc. of the 9th Conference on Architectural Support for Programming Languages and Operating Systems (November), San Jose, 49-58.

2000. GOODMAN, J. R. [1983]. “Using cache memory to reduce processor memory traffic,” *Proc. 10th Int’l Symposium on Computer Architecture* (June), Stockholm, Sweden, 124–131.

HAGERSTEN E. AND M. KOSTER [1998]. “WILDFIRE: A SCALABLE PATH FOR SMPS,” ROCEEDINGS OF THE THE FIFTH INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE COMPUTER ARCHITECTURE , 1998.

HAGERSTEN, E., LANDIN, A. AND S. HARIDI. DDM --- A Cache-Only Memory Architecture. IEEE Computer, 25(9):44-54, September, 1992.

HILL, M.D. [1998]. “Multiprocessors should support simple memory consistency models,” *IEEE Computer*, 31:8 (August), 28–34.

HILLIS, W. D. [1985]. *The Connection Multiprocessor*, MIT Press, Cambridge, Mass.

HIRATA, H., KIMURA, K., NAGAMINE, S., MOCHIZUKI, Y., NISHIMURA, A., NAKASE, Y., AND NISHIZAWA, T. [1992]. “An elementary processor architecture with simultaneous instruction issuing from multiple threads,” Proc. 19th Annual International Symposium on Computer Architecture (May). 136–145.

HOCKNEY, R. W. AND C. R. JESSHOPE [1988]. *Parallel Computers-2, Architectures, Programming and Algorithms*, Adam Hilger Ltd., Bristol, England.

HOLLAND, J. H. [1959]. “A universal computer capable of executing an arbitrary number of subprograms simultaneously,” *Proc. East Joint Computer Conf.* 16, 108–113.

HORD, R. M. [1982]. *The Illiac-IV, The First Supercomputer*, Computer Science Press, Rockville, Md.

HRISTEA, C., LENOSKI, D., AND J. KEEN [1997]. Measuring Memory Hierarchy Performance of Cache-Coherent Multiprocessors Using Micro Benchmarks, Proc. Supercomputing 97, San Jose, CA, November.

HWANG, K. [1993]. *Advanced Computer Architecture and Parallel Programming*, McGraw-Hill, New York.

KECKLER, S.W. AND DALLY, W. J. [1992]. “Processor coupling: Integrating compile time and runtime scheduling for parallelism,”. Proc. 19th Annual International Symposium on Computer Architecture (May). 202–213.

KONTOTHANASSIS, L., HUNT, G., STETS, R.,HARDAVELLAS, N., CIERNIAK, M., PARTHASARATHY,S., MEIRA, W., DWARKADAS, S. AND M. SCOTT [1997]. “VM-based shared memory on low-latency, remote-memory-access networks”, . Proc., 24th Annual Int'l. Symp. on Computer Architecture, June, Denver.

KUSKIN, J., OFELT, D., HEINRICH, M., HEINLEIN, J., SIMONI, R., GHARACHORLOO, K., CHAPIN, J., NAKAHIRA, D., BAXTER, J., HOROWITZ, M., GUPTA, A., ROSENBLUM, M., AND J.L. HENNESSY [1994]. "The Stanford FLASH Multiprocessor”, Proceedings of the 21th International Symposium on Computer Architecture, Chicago, April.

LAMPORT, L. [1979]. “How to make a multiprocessor computer that correctly executes multiprocess programs,” *IEEE Trans. on Computers* C-28:9 (September), 241–248.

LAUDON, J., GUPTA, A., AND M. HOROWITZ [1994]. “Interleaving: A multithreading technique target-

ing multiprocessors and work-stations.," Proc Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (October), Boston, 308–318.

LAUDON J. AND D. LENOSKI [1997]. "THE SGI ORIGIN: A CCNUMA HIGHLY SCALABLE SERVER," Proceedings of the 24th international symposium on Computer architecture , June, Denver, p 241-251

LENOSKI, D., J. LAUDON, K. GHARACHORLOO, A. GUPTA, AND J. L. HENNESSY [1990]. "The Stanford DASH multiprocessor," *Proc. 17th Int'l Symposium on Computer Architecture* (June), Seattle, 148–159.

LENOSKI, D., J. LAUDON, K. GHARACHORLOO, W.-D. WEBER, A. GUPTA, J. L. HENNESSY, M. A. HOROWITZ, AND M. LAM [1992]. "The Stanford DASH multiprocessor," *IEEE Computer* 25:3 (March).

LI,.K., [1988] "IVY: A Shared Virtual Memory System for Parallel Computing," Proceedings of the 1988 International Conference on Parallel Processing, Pennsylvania State University Press.

LO,J., EGGERS, S., EMER, J., LEVY, H., STAMM, R., AND D. TULLSEN [1997]. "Converting Thread-Level Parallelism Into Instruction-Level Parallelism via Simultaneous Multithreading." *ACM Transactions on Computer Systems 15:2* (August), 322-354.

LO,J., BARROSO, L., EGGERS, S., GHARACHORLOO, K., LEVY,H., AND S. PAREKH [1998]. "An Analysis of Database Workload Performance on Simultaneous Multithreaded Processors." *Proceedings of the 25th International Symposium on Computer Architecture* (June), 39-50.

LOVETT, T. AND S. THAKKAR [1988]. "The Symmetry multiprocessor system," *Proc. 1988 Int'l Conf. of Parallel Processing*, University Park, Penn., 303–310.

MELLOR-CRUMMEY, J. M. AND M. L. SCOTT [1991]. "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. on Computer Systems* 9:1 (February), 21–65.

MENABREA, L. F. [1842]. "Sketch of the analytical engine invented by Charles Babbage," Bibiothèque Universelle de Genève (October).

MITCHELL, D. [1989]. "The Transputer: The time is now," *Computer Design* (RISC supplement), 40–41.

MIYA, E. N. [1985]. "Multiprocessor/distributed processing bibliography," *Computer Architecture News* (ACM SIGARCH) 13:1, 27–29.

NIKHIL, R.S., PAPADOPOULOS, G.M. AND ARVIND [1992]. "*T: A Multithreaded Massively Parallel Architecture." In Proceedings of the 19th International Symposium on Computer Architecture, Gold Coast, Australia, May, 156–167.

NOORDERGRAAF, L. AND R VAN DER PAS [1999]. "Performance Experiences on Sun's WildFire Prototype," Proc. Supercomputing 99, Portand, Oregon, November.

PFISTER, G. F., W. C. BRANTLEY, D. A. GEORGE, S. L. HARVEY, W. J. KLEINFEKDER, K. P. MCAULIFFE, E. A. MELTON, V. A. NORTON, AND J. WEISS [1985]. "The IBM research parallel processor prototype (RP3): Introduction and architecture," *Proc. 12th Int'l Symposium on Computer Architecture* (June), Boston, 764–771.

REINHARDT, S.K., LARUS, J.R., AND D. A. WOOD[1994]. "Tempest and Typhoon: User-Level Shared Memory." In Proceedings of the 21st Annual International Symposium on Computer Architecture, . Chicago, April, 325--336.

RETTBERG, R. D., W. R. CROWTHER, P. P. CARVEY, AND R. S. TOWLINSON [1990]. "The Monarch parallel processor hardware design," *IEEE Computer* 23:4 (April).

ROSENBLUM, M., S. A. HERROD, E. WITCHEL, AND A. GUTPA [1995]. "Complete computer simulation: The SimOS approach," to appear in *IEEE Parallel and Distributed Technology* 3:4 (fall).

SAULSBURY, A., WILKINSON, T., CARTER, J. AND A. LANDIN [1995]. "An Argument for Simple COMA," *Proc. First Conf. on High Performance Computer Architectures* (January), Raleigh, N. Carolina,, 276-285

SCHWARTZ, J. T. [1980]. “Ultracomputers,” *ACM Trans. on Programming Languages and Systems* 4:2, 484–521.

SCOTT S. L. [1996] “SYNCHRONIZATION AND COMMUNICATION IN THE T3E MULTIPROCESSOR,“ Proceeding Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII), Cambridge, Massachusetts, October, pp. 26--36.

SCOTT S. L. AND G. M. THORSON. “The Cray T3E Network: Adaptive Routing in a High Performance 3D Torus,” In Proceedings of the Symposium on High Performance Interconnects (Hot Interconnects 4), Stanford University, August, pages 14-156.

SEITZ, C. [1985]. “The Cosmic Cube,” *Comm. ACM* 28:1 (January), 22–31.

SINGH, J. P, HENNESSY, J. L. AND A. GUPTA., "Scaling Parallel Programs for Multiprocessors: Methodology and Examples," Computer 26: 7 (July), 22–33.

SLOTNICK, D. L., W. C. BORCK, AND R. C. MCREYNOLDS [1962]. “The Solomon computer,” *Proc. Fall Joint Computer Conf.* (December), Philadelphia, 97–107.

SMITH, B.J. [1978] “A pipelined, shared resource MIMD computer,” Proc. 1978 ICPP (August) pp. 6--8.

SOUNDARARAJAN, V., HEINRICH, M., VERGHESE, B., GHARACHORLOO, K., GUPTA, A., AND J.L. HENNESSY [1998]. “FLEXIBLE USE OF MEMORY FOR REPLICATION/MIGRATION IN CACHE-COHERENT DSM MULTIPROCESSORS,” . *Proc. 25th Int’l Symposium on Computer Architecture* (June), Barcelona, Spain, 342-355.

STENSTRÖM, P., JOE, T. AND A. GUPTA [1992]. “Comparative performance evaluation of cache-coherent NUMA and COMA architectures.” Proceedings of the 19th annual international symposium on Computer architecture, May, Queensland Australia, 80-91.

STONE, H. [1991]. *High Performance Computers*, Addison-Wesley, New York.

SWAN, R. J., A. BECHTOLSHEIM, K. W. LAI, AND J. K. OUSTERHOUT [1977]. “The implementation of the Cm* multi-microprocessor,” *Proc. AFIPS National Computing Conf.*, 645–654.

SWAN, R. J., S. H. FULLER, AND D. P. SIEWIOREK [1977]. “Cm*—A modular, multi-microprocessor,” *Proc. AFIPS National Computer Conf.* 46, 637–644.

TANG, C. K. [1976]. “Cache design in the tightly coupled multiprocessor system,” *Proc. AFIPS National Computer Conf.*, New York (June), 749–753.

THEKKATH, R. SINGH, A.P. SINGH, J.P., JOHN, S. AND J.L. HENNESSY [1997]. “An Evaluation of a Commercial CC-NUMA Architecture---The CONVEX Exemplar SPP1200,” Proceedings of the 11th International Parallel Processing Symposium (IPPS '97), Geneva, Switzerland, April.

TULLSEN, D.M., EGGERS, S.J.,. EMER, J.S.,. LEVY, H.M.. LO, J.L. AND R.L. STAMM [1996]. “Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor.” Proceedings of the 23rd Annual International Symposium on Computer Architecture (May), pages 191--202.

TULLSEN, D.M., EGGERS, S.J., AND H.M. LEVY [1995], “Simultaneous multithreading: Maximizing on-chip parallelism,” *Proc. 22nd International Symposium on Computer Architecture* (June), pp.392-403.

UNGER, S. H. [1958]. “A computer oriented towards spatial problems,” *Proc. Institute of Radio Enginers* 46:10 (October), 1744–1750.

WILSON, A. W., JR. [1987]. “Hierarchical cache/bus architecture for shared-memory multiprocessors,” *Proc. 14th Int’l Symposium on Computer Architecture* (June), Pittsburgh, 244–252.

WOOD, D. A. AND M. D. HILL [1995]. “Cost-effective parallel computing,” *IEEE Computer* 28:2 (February).

WOLFE, A. AND J. P. SHEN [1991]. “A variable instruction stream extension to the VLIW architecture.” *Proc. of the Fourth Conference on Architectural Support for Programming Languages and*

Operating Systems (April), Santa Clara, 2-14.

WULF, W. AND C. G. BELL [1972]. "C.mmp—A multi-mini-processor," *Proc. AFIPS Fall Joint Computing Conf.* 41, part 2, 765–777.

WULF, W. AND S. P. HARBISON [1978]. "Reflections in a pool of processors—An experience report on C.mmp/Hydra," *Proc. AFIPS 1978 National Computing Conf.* 48 (June), Anaheim, Calif., 939–951.

Yamamoto, W., Serrano, M.J., Talcott, A.R., Wood, R.C., and M. Nemirosky [1992].. "Performance estimation of multistreamed, superscalar processors," Proc. Twenty-Seventh Hawaii International Conference on System Sciences (January), pages I:195–204.

EXERCISES

6.1 [10] <6.1> Suppose we have an application that runs in three modes: all processors used, half the processors in use, and serial mode. Assume that 0.02% of the time is serial mode, and there are 100 processors in total. Find the maximum time that can be spent in the mode when half the processors are used, if our goal is a speedup of 80.

6.2 [15] <6.1> Assume that we have a function for an application of the form F(*i*,*p*), which gives the fraction of time that exactly *i* processors are usable given that a total of *p* processors are available. This means that

$$\sum_{i=1}^{p} F(i,p) = 1$$

Assume that when *i* processors are in use, the application runs *i* times faster. Rewrite Amdahl's Law so that it gives the speedup as a function of *p* for some application.

6.3 [10] <6.1, 6.2> The Transaction Processing Council (TPC) has several different benchmarks. Visit their website at www.tpc.org and look at the top 10 performers in each benchmark class. Determine whether each of the top 10 configurations is a multiprocessor or if so what types (SMP, NUMA, cluster, e.g.). Does the ordering look different if price-performance is used as the metric?

6.4 [10] <6.1, 6.2> The Top 500 list categorizes the fastest scientific machines in the world according to their performance on the Linpack benchmark. Visit their website at www.top500.org and look at the top 100 performers (there are many repeats of a particular vendor product, since individual supercomputer sites rather than a product are counted). Determine how many different supercomputer products occur among the top 100 configurations and what type (SMP, NUMA, cluster, e.g.) each different supercomputer is. Try to obtain cost information and see how the data changes when cost-performance is considered.

6.5 [15] <6.3> In small bus-based multiprocessors, write-through caches are sometimes used. One reason is that a write-through cache has a slightly simpler coherence protocol. Show how the basic snooping cache coherence protocol of Figure 6.12 on page 668 can be changed for a write-through cache. From the viewpoint of an implementor, what is the major hardware functionality that is not needed with a write-through cache compared with a

write-back cache?

6.6 [20] <6.3> Add a clean private state to the basic snooping cache-coherence protocol (Figure 6.12 on page 668). Show the protocol in the format of Figure 6.12.

6.7 [15] <6.3> One proposed solution for the problem of false sharing is to add a valid bit per word (or even for each byte). This would allow the protocol to invalidate a word without removing the entire block, allowing a cache to keep a portion of a block in its cache while another processor wrote a different portion of the block. What extra complications are introduced into the basic snooping cache coherency protocol (Figure 6.12) if this capability is included? Remember to consider all possible protocol actions.

6.8 [12/10/15] <6.3> The performance differences for write invalidate and write update schemes can arise from both bandwidth consumption and latency. Assume a memory system with 64-byte cache blocks. Ignore the effects of contention.

a. [12] <6.3> Write two parallel code sequences to illustrate the bandwidth differences between invalidate and update schemes. One sequence should make update look much better and the other should make invalidate look much better.

b. [10] <6.3> Write a parallel code sequence to illustrate the latency advantage of an update scheme versus an invalidate scheme.

c. [15] <6.3> Show, by example, that when contention is included, the latency of update may actually be worse. Assume a bus-based multiprocessor with 50-cycle memory and snoop transactions.

6.9 Use the data on miss rates versus block size for the scientific applications in Section 6.3 to compute AMAT and bus bandwidth making some assumptions about memory access time based n block size.

6.10 [15/15] <6.3–6.5> Restructure this exercise to use timing from E6000 series.

One possible approach to achieving the scalability of distributed shared memory and the cost-effectiveness of a bus design is to combine the two approaches, using a set of processors with memories attached directly to the processors, and interconnected with a bus. The argument in favor of such a design is that the use of local memories and a coherence scheme with limited broadcast results in a reduction in bus traffic, allowing the bus to be used for a larger number of processors. For these Exercises, assume the same parameters as for the Challenge bus. Assume that remote snoops and memory accesses take the same number of cycles as a memory access on the Challenge bus. Ignore the directory processing time for these Exercises. Assume that the coherency scheme works as follows on a miss: If the data are up-to-date in the local memory, it is used there. Otherwise, the bus is used to snoop for the data. Assume that local misses take 25 bus clocks.

a. [15] <6.3–6.5> Find the time for a read or write miss to data that are remote.

b. [15] <6.3–6.5> Ignoring contention and using the data from the Ocean benchmark run on 16 processors for the frequency of local and remote misses (Figure 6.31 on page 699), estimate the average memory access time versus that for a Challenge using the same total miss rate.

6.11 [12/15] <6.3,6.5,6.11> Restructure this exercise using the data comparing Origin to

E6000.

Although it is widely believed that buses are the ideal interconnect for small-scale multiprocessors, this may not always be the case. For example, increases in processor performance are lowering the processor count at which a more distributed implementation becomes attractive. Because a standard bus-based implementation uses the bus both for access to memory and for interprocessor coherency traffic, it has a uniform memory access time for both. In comparison, a distributed memory implementation may sacrifice on remote memory access, but it can have a much better local memory access time.

Consider the design of a DSM multiprocessor with 16 processors. Assume the R4400 cache miss overheads shown for the Challenge design (see pages 730–731). Assume that a memory access takes 150 ns from the time the address is available from either the local processor or a remote processor until the first word is delivered.

a. [12] <6.3,6.5,6.11> How much faster is a local access than on the Challenge?

b. [15] <6.3,6.5,6.11> Assume that the interconnect is a 2D grid with links that are 16 bits wide and clocked at 100 MHz, with a start-up time of five cycles for a message. Assume one clock cycle between nodes in the network, and ignore overhead in the messages and contention (i.e., assume that the network bandwidth is not the limit). Find the average remote memory access time, assuming a uniform distribution of remote requests. How does this compare to the Challenge case? What is the largest fraction of remote misses for which the DSM multiprocessor will have a lower average memory access time than that of the Challenge multiprocessor?

6.12 [20/15/30] <6.5> One downside of a straightforward implementation of directories using fully populated bit vectors is that the total size of the directory information scales as the product: Processor count × Memory blocks. If memory is grown linearly with processor count, then the total size of the directory grows quadratically in the processor count. In practice, because the directory needs only 1 bit per memory block (which is typically 32 to 128 bytes), this problem is not serious for small to moderate processor counts. For example, assuming a 128-byte block, the amount of directory storage compared to main memory is Processor count/1024, or about 10% additional storage with 100 processors. This problem can be avoided by observing that we only need to keep an amount of information that is proportional to the cache size of each processor. We explore some solutions in these Exercises.

a. [20] <6.5> One method to obtain a scalable directory protocol is to organize the multiprocessor as a logical hierarchy with the processors at the leaves of the hierarchy and directories positioned at the root of each subtree. The directory at each subtree root records which descendents cache which memory blocks, as well as which memory blocks with a home in that subtree are cached outside of the subtree. Compute the amount of storage needed to record the processor information for the directories, assuming that each directory is fully associative. Your answer should incorporate both the number of nodes at each level of the hierarchy as well as the total number of nodes.

b. [15] <6.5> Assume that each level of the hierarchy in part (a) has a lookup cost of 50 cycles plus a cost to access the data or cache of 50 cycles, when the point is reached. We want to compute the AMAT (average memory access time—see Chapter 5) for a 64-processor multiprocessor with four-node subtrees. Use the data from the Ocean benchmark run on 64 processors (Figure 6.31) and assume that all noncoherence miss-

es occur within a subtree node and that coherence misses are uniformly distributed across the multiprocessor. Find the AMAT for this multiprocessor. What does this say about hierarchies?

c. [30] <6.5> An alternative approach to implementing directory schemes is to implement bit vectors that are not dense. There are two such strategies: one reduces the number of bit vectors needed and the other reduces the number of bits per vector. Using traces, you can compare these schemes. First, implement the directory as a four-way set-associative cache storing full bit vectors, but only for the blocks that are cached outside of the home node. If a directory cache miss occurs, choose a directory entry and invalidate the entry. Second, implement the directory so that every entry has 8 bits. If a block is cached in only one node outside of its home, this field contains the node number. If the block is cached in more than one node outside its home, this field is a bit vector with each bit indicating a group of eight processors, at least one of which caches the block. Using traces of 64-processor execution, simulate the behavior of these two schemes. Assume a perfect cache for nonshared references, so as to focus on coherency behavior. Determine the number of extraneous invalidations as the directory cache size is increased.

6.13 [25/40] <6.10> Prefetching and relaxed consistency models are two methods of tolerating the latency of longer access in multiprocessors. Another scheme, originally used in the HEP multiprocessor and incorporated in the MIT Alewife multiprocessor, is to switch to another activity when a long-latency event occurs. This idea, called *multiple context* or *multithreading*, works as follows:

- The processor has several register files and maintains several PCs (and related program states). Each register file and PC holds the program state for a separate parallel thread.
- When a long-latency event occurs, such as a cache miss, the processor switches to another thread, executing instructions from that thread while the miss is being handled.

a. [25] <6.10> Using the data for the Ocean benchmark running on 64 processors (Figure 6.31), determine how many contexts are needed to hide all the latency of remote accesses. Assume that local cache misses take 40 cycles and that remote misses take 120 cycles. Assume that the increased demands due to a higher request rate do not affect either the latency or the bandwidth of communications.

b. [40] <6.10> Implement a simulator for a multiple-context directory-based multiprocessor. Use the simulator to evaluate the performance gains from multiple context. How significant are contention and the added bandwidth demands in limiting the gains?

6.14 [25] <6.10> Prove that in a two-level cache hierarchy, where L1 is closer to the processor, inclusion is maintained with no extra action if L2 has at least as much associativity as L1, both caches use LRU replacement, and both caches have the same block size.

6.15 [20] <6.5,6.11> As we saw in the *Putting it All Together* and in *Fallacies and Pitfalls*, data distribution can be important when an application has a nontrivial private data miss rate caused by capacity misses. This problem can be attacked with compiler technology (distributing the data in blocks) or through architectural support, as we saw in the descrip-

tion of CMR on Wildfire.

Assume that we have two DSM multiprocessors: one with CMR support and one without such support. Both multiprocessors have one processor per node and remote coherence misses, which are uniformly distributed, take 1 μS. Assume that all capacity misses on the CMR multiprocessor hit in the local memory and require 250 ns. Assume that capacity misses take 200 ns when they are local on the DSM multiprocessor without CMR and 800 ns, otherwise. Using the Ocean data for 32 processors (Figure 6.23), find what fraction of the capacity misses on the DSM multiprocessor must be local if the performance of the two multiprocessors is identical.

6.16 [15] <6.7> Some multiprocessors have implemented a special broadcast coherence protocol just for locks, sometimes even using a different bus. Evaluate the performance of the spin lock in the Example on page 710 assuming a write broadcast protocol.

6.17 [15] <6.7> Implement the barrier in Figure 6.40 on page 713, using queuing locks. Compare the performance to the spin-lock barrier.

6.18 [15] <6.7> Implement the barrier in Figure 6.40 on page 713, using fetch-and-increment. Compare the performance to the spin-lock barrier.

6.19 [15] <6.7> Implement the barrier on page 717, so that barrier release is also done with a combining tree.

6.20 [30] <6.3–6.7,6.11> Using an available shared-memory multiprocessor, see if you can determine the organization and latencies of its memory hierarchy. For each level of the hierarchy, you can look at the total size, block size, and associativity, as well as the latency of each level of the hierarchy. If the multiprocessor uses a nonbus interconnection network, see if you can discover the topology and latency characteristics of the network. Try to make a table like that in Figure 6.47 for the machine. The lmbench (www.bitmover.com/lmbench/) and stream (http://www.cs.virginia.edu/stream/) benchmark may prove useful in this exercise.

6.21 [30] <6.3–6.7,6.11> Perform exercise 6.20 but looking at the bandwidth characteristics rather than latency. See if you can prepare a table like that in Figure 6.48. Extend the table by looking at the effect of strided accesses, as well as sequential and unrelated accesses.

6.22 [20] <6.5> As we discussed earlier, the directory controller can send invalidates for lines that have been replaced by the local cache controller. To avoid such messages, and to keep the directory consistent, replacement hints are used. Such messages tell the controller that a block has been replaced. Modify the directory coherence protocol of section 6.5 to use such replacement hints.

6.23 [15] <6.7> Find the time for n processes to synchronize using a standard barrier. Assume that the time for a single process to update the count and release the lock is c.

6.24 [15] <6.7> Find the time for n processes to synchronize using a combining tree barrier. Assume that the time for a single process to update the count and release the lock is c.

6.25 [25] <6.7> Implement a software version of the queuing lock for a bus-based system. Using the model in the Example on page 710, how long does it take for 20 processors to acquire and release the lock? You need only count bus cycles.

6.26 [20/30] <6.2–6.7> Both researchers and industry designers have explored the idea of having the capability to explicitly transfer data between memories. The argument in favor of such facilities is that the programmer can achieve better overlap of computation and communication by explicitly moving data when it is available. The first part of this exercise explores the potential on paper; the second explores the use of such facilities on real multiprocessors.

a. [20] <6.2–6.7> Assume that cache misses stall the processor, and that block transfer occurs into the local memory of a DSM node. Assume that remote misses cost 100 cycles and that local misses cost 40 cycles. Assume that each DMA transfer has an overhead of 10 cycles. Assuming that all the coherence traffic can be replaced with DMA into main memory followed by a cache miss, find the potential improvement for Ocean running on 64 processors (Figure 6.31).

b. [30] <6.2–6.7> Find a multiprocessor that implements both shared memory (coherent or incoherent) and a simple DMA facility. Implement a blocked matrix multiply using only shared memory and using the DMA facilities with shared memory. Is the latter faster? How much? What factors make the use of a block data transfer facility attractive?

6.27 [Discussion] <6.11> Construct a scenario whereby a truly revolutionary architecture—pick your favorite candidate—will play a significant role. *Significant* is defined as 10% of the computers sold, 10% of the users, 10% of the money spent on computers, or 10% of some other figure of merit.

6.28 [40] <6.2,6.10,6.14> A multiprocessor or cluster is typically marketed using programs that can scale performance linearly with the number of processors. The project here is to port programs written for one multiprocessor to the others and to measure their absolute performance and how it changes as you change the number of processors. What changes need to be made to improve performance of the ported programs on each multiprocessor? What is the ratio of processor performance according to each program?

6.29 [35] <6.2,6.10,6.14> Instead of trying to create fair benchmarks, invent programs that make one multiprocessor or cluster look terrible compared with the others, and also programs that always make one look better than the others. It would be an interesting result if you couldn't find a program that made one multiprocessor or cluster look worse than the others. What are the key performance characteristics of each organization?

6.30 [40] <6.2,6.10,6.14> Multiprocessors and cluster usually show performance increases as you increase the number of processors, with the ideal being n times speedup for n processors. The goal of this biased benchmark is to make a program that gets worse performance as you add processors. For example, this means that one processor on the multiprocessor or cluster runs the program fastest, two are slower, four are slower than two, and so on. What are the key performance characteristics for each organization that give inverse linear speedup?

6.31 [50] <6.2,6.10,6.14> Networked workstations can be considered multicomputers or clusters, albeit with somewhat slower, though perhaps cheaper, communication relative to computation. Port some cluster benchmarks to a network using remote procedure calls for communication. How well do the benchmarks scale on the network versus the cluster? What are the practical differences between networked workstations and a commercial clus-

ter, such as the IBM-SP series?

7 Storage Systems

I/O certainly has been lagging in the last decade.

Seymour Cray
Public Lecture (1976)

Also, I/O needs a lot of work.

David Kuck
Keynote Address, 15th Annual Symposium on Computer Architecture (1988)

Combining bandwidth and storage ... enables swift and reliable access to the ever expanding troves of content on the proliferating disks and ... repositories of the Internet. ... the capacity of storage arrays of all kinds is rocketing ahead of the advance of computer performance.

George Gilder
"The End Is Drawing Nigh" Forbes ASAP (April 4, 2000)

7.1	Introduction	485
7.2	Types of Storage Devices	487
7.3	Buses—Connecting I/O Devices to CPU/Memory	500
7.4	Reliability, Availability, and Dependability	509
7.5	RAID: Redundant Arrays of Inexpensive Disks	514
7.6	Errors and Failures in Real Systems	520
7.7	I/O Performance Measures	524
7.8	A Little Queuing Theory	530
7.9	Benchmarks of Storage Performance and Availability	541
7.10	Crosscutting Issues	547
7.11	Designing an I/O System in Five Easy Pieces	552
7.12	Putting It All Together: EMC Symmetrix and Celerra	565
7.13	Another View: Sanyo DSC-110 Digital Camera	572
7.14	Fallacies and Pitfalls	575
7.15	Concluding Remarks	581
7.16	Historical Perspective and References	582
	Exercises	590

7.1 Introduction

Input/output has been the orphan of computer architecture. Historically neglected by CPU enthusiasts, the prejudice against I/O is institutionalized in the most widely used performance measure, CPU time (page 32). The performance of a computer's I/O system cannot be measured by CPU time, which by definition ignores I/O. The second-class citizenship of I/O is even apparent in the label *peripheral* applied to I/O devices.

This attitude is contradicted by common sense. A computer without I/O devices is like a car without wheels—you can't get very far without them. And while CPU time is interesting, response time—the time between when the user types a command and when results appear—is surely a better measure of performance. The customer who pays for a computer cares about response time, even if the CPU designer doesn't.

Does I/O Performance Matter?

Some suggest that the prejudice against I/O is well founded. I/O speed doesn't matter, they argue, since there is always another process to run while one process waits for a peripheral.

There are several points to make in reply. First, this is an argument that performance is measured as *throughput*—number of tasks completed per hour—versus response time. Plainly, if users didn't care about response time, interactive software never would have been invented, and there would be no workstations or personal computers today; section 7.7 gives experimental evidence of the importance of response time. It may also be expensive to rely on running other processes, since paging traffic from process switching might actually increase I/O. Furthermore, with mobile devices and desktop computing, there is only one person per computer and thus fewer processes than in timesharing. Many times the only waiting process is the human being! Moreover, applications such as transaction processing (section 7.7) place strict limits on response time as part of the performance analysis.

I/O's revenge is at hand. Suppose response time is just 10% longer than CPU time. First we speed up the CPU by a factor of 10, while neglecting I/O. Amdahl's Law tells us the speedup is only 5 times, half of what we would have achieved if both were sped up tenfold. Similarly, making the CPU 100 times faster without improving the I/O would obtain a speedup of only 10 times, squandering 90% of the potential. If, as predicted in Chapter 1, performance of CPUs improves at 55% per year and I/O did not improve, every task would become I/O-bound. There would be no reason to buy faster CPUs—and no jobs for CPU designers. Thus, I/O performance increasingly limits system performance and effectiveness.

Does CPU Performance Matter?

Moore's Law leads to both large, fast CPUs but also to very small, cheap CPUs. Especially for systems using the latter CPU, it is increasingly unlikely that the most important goal is keeping the CPU busy versus keeping I/O devices busy, as the bulk of the costs may not be with the CPU.

This change in importance is also reflected by the names of our times. Whereas the 1960s to 1980s were called the Computing Revolution, the period since 1990 is been called the Information Age, with concerns focussed on advances in information technology versus raw computational power.

This shift in focus from computation to communication and storage of information emphasizes reliability and scalability as well as cost-performance. To reflect the increasing importance of I/O, the third edition of this book has twice as many I/O chapters as the first edition and half as many on instruction set architecture. This chapter covers storage I/O and the next covers communication I/O. Although two chapters cannot fully vindicate I/O, they may at least atone for some of the sins of the past and restore some balance.

Does Performance Matter?

After 15 years of doubling processor performance every 18 months, processor performance is not the problem it once was. Many would find highly dependable systems much more attractive than faster versions of today's systems with today's level of unreliability. Although it is frustrating when a program crashes, people become hysterical if they lose their data. Hence, storage systems are typically held to a higher standard of dependability than the rest of the computer. Because of traditional demands placed on storage–and because a new century needs new challenges–this chapter defines reliability, availability, and dependability and shows how to improve them.

Dependability is the bedrock of storage, yet it also has its own rich performance theory–queueing theory–that balances throughput versus response time. The software that determines which processor features get used is the compiler, but the operating system usurps that role for storage.

Thus, storage has a different, multifaceted culture from processors, yet it is still found within the architecture tent. We start our exploration of storage with the hardware building blocks.

7.2 Types of Storage Devices

Rather than discuss the characteristics of all storage devices, we will concentrate on those most commonly found: magnetic disks, magnetic tapes, automated tape libraries, CDs, and DVDs. As these I/O devices are generally too large for embedded applications, we conclude with a description of Flash memory, a storage device commonly used in portable devices. (Experienced readers should skip the following subsections with which they are already familiar.)

Magnetic Disks

I think Silicon Valley was misnamed. If you look back at the dollars shipped in products in the last decade, there has been more revenue from magnetic disks than from silicon. They ought to rename the place Iron Oxide Valley.

Al Hoagland, One of the Pioneers of Magnetic Disks (1982)

Despite repeated attacks by new technologies, magnetic disks have dominated nonvolatile storage since 1965. Magnetic disks play two roles in computer systems:

- Long-term, nonvolatile storage for files, even when no programs are running
- A level of the memory hierarchy below main memory used as a backing store for virtual memory during program execution (see section 5.10)

In this section, we are not talking about floppy disks, but the original “hard” disks.

As descriptions of magnetic disks can be found in countless books, we will only list the essential characteristics, with the terms illustrated in Figure 7.1. (Readers who recall these terms might want to skip to the section entitled "The Future of Magnetic Disks" on page 492; those interested in more detail should see Hospodor and Hoagland [1993]) A magnetic disk consists of a collection of *platters* (generally 1 to 12), rotating on a spindle at 3,600 to 15,000 revolutions per minute (RPM). These platters are metal or glass disks covered with magnetic recording material on both sides, so 10 platters have 20 recording surfaces. Disk diameters in 2001 vary by almost a factor of four, from 1.0 to 3.5 inches, although more than 95% of sales are either 2.5- or 3.5- inch diameter disks. Traditionally, the biggest disks have the highest performance and the smallest disks have the lowest price per disk drive. Price per gigabyte often goes to the disks sold in highest volume, which today are 3.5-inch disks.

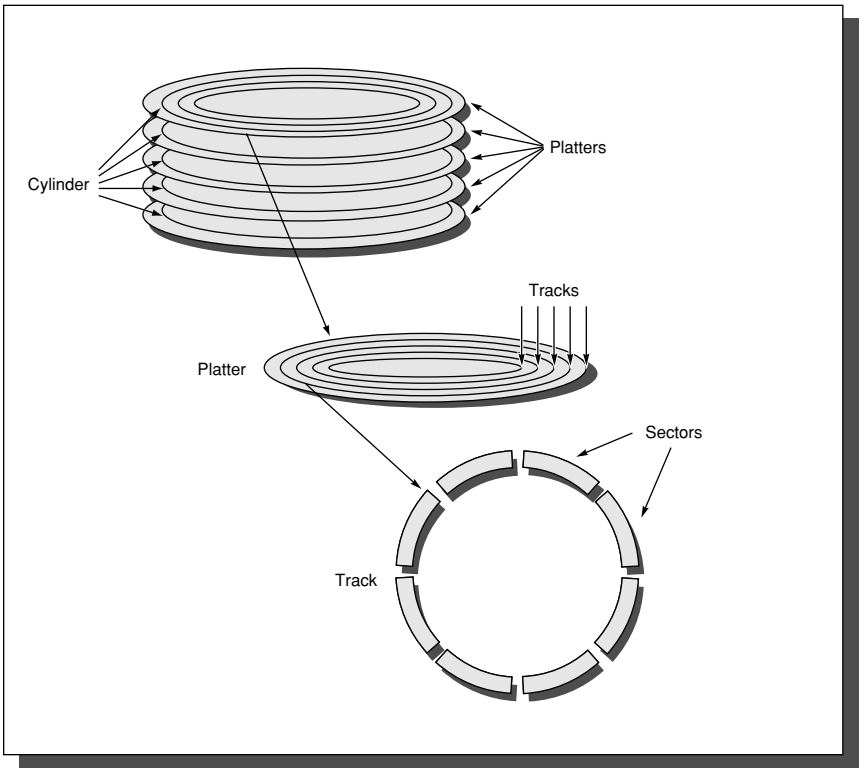


FIGURE 7.1 Disks are organized into platters, tracks, and sectors. Both sides of a platter are coated so that information can be stored on both surfaces. A cylinder refers to a track at the same position on every platter.

The disk surface is divided into concentric circles, designated *tracks*. There are typically 5,000 to 30,000 tracks on each surface. Each track in turn is divided into *sectors* that contain the information; a track might have 100 to 500 sectors. A sector is the smallest unit that can be read or written. IBM mainframes allow users to select the size of the sectors, although most systems fix their size, typically at 512 bytes of data. The sequence recorded on the magnetic media is a sector number, a gap, the information for that sector including error correction code, a gap, the sector number of the next sector, and so on. Occasionally people forget this sequence—confusing the recording density with the density that a user's data can be stored—leading to fallacies about disks (see section 7.14).

In the past, all tracks had the same number of sectors; the outer tracks, which are longer, recorded information at a much lower density than the inner tracks. Recording more sectors on the outer tracks than on the inner tracks, called *constant bit density*, is the standard today. This name is misleading, as the bit density is not really constant. Typically, the inner tracks are recorded at the highest density and the outer tracks at the lowest, but the outer tracks might record, say, 1.7 times more bits despite being 2.1 times longer.

Figure 7.2 shows the characteristics of three magnetic disks in 2000. Large-diameter drives have many more gigabytes to amortize the cost of electronics, so the traditional wisdom used to be that they had the lowest cost per gigabyte. This advantage can be offset, however, if the small drives have much higher sales volume, which lowers manufacturing costs. The 3.5-inch drive, which is the largest surviving drive in 2001, also has the highest sales volume, so it unquestionably has the best price per gigabyte

To read and write information into a sector, a movable *arm* containing a *read/write head* is located over each surface. Rather than represent each recorded bit individually, groups of bits are recorded using a run-length-limited code. Run-length limited codes ensure that there is both a minimum and maximum number of bits in a group that the reader must decipher before seeing synchronization signals, which enables higher recording density as well as reducing error rates. The arms for all surfaces are connected together and move in conjunction, so that all arms are over the same track of all surfaces. The term *cylinder* is used to refer to all the tracks under the arms at a given point on all surfaces.

To read or write a sector, the disk controller sends a command to move the arm over the proper track. This operation is called a *seek*, and the time to move the arm to the desired track is called *seek time*.

Average seek time is the subject of considerable misunderstanding. Disk manufacturers report minimum seek time, maximum seek time, and average seek time in their manuals. The first two are easy to measure, but the average was open to wide interpretation. The industry decided to calculate average seek time as the sum of the time for all possible seeks divided by the number of possible seeks. Average seek times are advertised to be 5 ms to 12 ms. Depending on the application and operating system, however, the actual average seek time may be only 25% to 33% of the advertised number. The explanation is locality of disk references. Section 7.14 has a detailed example.

Characteristics	Seagate Cheetah ST173404LC Ultra160 SCSI Drive	IBM Travelstar 32GH DJSA - 232 ATA-4 Drive	IBM 1GB Microdrive DSCM-11000
Disk diameter (inches)	3.5	2.5	1.0
Formatted data capacity (GB)	73.4	32.0	1.0
Cylinders	14,100	21,664	7,167
Disks	12	4	1
Recording Surfaces (or Heads)	24	8	2
Bytes per sector	512 to 4096	512	512
Average Sectors per track (512 byte)	≈ 424	≈ 360 (256-469)	≈ 140
Maximum areal density (Gbit/sq.in.)	6.0	14.0	15.2
Rotation speed (RPM)	10033	5411	3600
Average seek random cylinder to cylinder (read/write) in ms	5.6/6.2	12.0	12.0
Minimum seek in ms (read/write)	0.6/0.9	2.5	1.0
Maximum seek in ms	14.0/15.0	23.0	19.0
Data transfer rate in MB/second	27 to 40	11 to 21	2.6 to 4.2
Link speed to disk buffer in MB/second	160	67	13
Power idle/operating in Watts	16.4 / 23.5	2.0 / 2.6	0.5 / 0.8
Buffer size in MB	4.0	2.0	0.125
Size: height x width x depth in inches	1.6 x 4.0 x 5.8	0.5 x 2.7 x 3.9	0.2 x 1.4 x 1.7
Weight in pounds	2.00	0.34	0.035
Rated MTTF in powered-on hours	1,200,000	(see caption)	(see caption)
% of powered on hours (POH) per month	100%	45%	20%
% of POH seeking, reading, writing	90%	20%	20%
Load/Unload cycles (disk powered on/off)	250 per year	300,000	300,000
Nonrecoverable read errors per bits read	<1 per 10^{15}	< 1 per 10^{13}	< 1 per 10^{13}
Seek errors	<1 per 10^{7}	not available	not available
Shock tolerance: Operating, Not operating	10 G, 175 G	150 G, 700 G	175 G, 1500 G
Vibration tolerance: Operating, Not operating (sine swept, 0 to peak)	5-400 Hz @ 0.5G, 22-400 Hz @ 2G	5-500 Hz @ 1G, 2.5-500 Hz @ 5G	5-500 Hz @ 1G, 10-500 Hz @ 5G

FIGURE 7.2 Characteristics of three magnetic disks of 2000. To help the reader gain intuition about disks, this table gives typical values for disk parameters. The 2.5-inch drive is a factor of 6 to 9 better in weight, size, and power than the 3.5-inch drive. The 1.0-inch drive is a factor 10 to 11 better than the 2.5-inch drive in weight and size, and a factor of 3-4 better in power. Note that 3.5-inch drives are designed to be used almost continuously, and so rarely turned on and off, while the smaller drives spend most of their time unused and thus are turned on and off repeatedly. In addition, these mobile drives must handle much larger shocks and vibrations, especially when turned off. These requirements affect the relative cost of these drives. Note that IBM no longer quotes MTBF for 2.5 inch drives, but when they last did it was 300,000 hours. IBM quotes the service life as 5 years or 20,000 powered on hours, whichever is first. The service life for the 1.0-inch drives is 5 years or 8800 powered on hours, whichever is first.

The time for the requested sector to rotate under the head is the *rotation latency* or *rotational delay.* The average latency to the desired information is obviously halfway around the disk; if a disk rotates at 10,000 revolutions per minute (RPM), the average rotation time is therefore

$$\text{Average rotation time} = \frac{0.5}{\cancel{10000\ \text{RPM}}} = \frac{0.5}{\cancel{(10000/60)\ \text{RPS}}} = 0.0030\ \text{sec} = 3.0\ \text{ms}$$

Note that there are two mechanical components to a disk access. It takes several milliseconds on average for the arm to move over the desired track and several milliseconds on average for the desired sector to rotate under the read/write head. A simple performance model is to allow one-half rotation of the disk to find the desired data after the proper track is found. Of course, the disk is always spinning, so seeking and rotating actually overlap.

The next component of disk access, *transfer time,* is the time it takes to transfer a block of bits, typically a sector, under the read-write head. This time is a function of the block size, disk size, rotation speed, recording density of the track, and speed of the electronics connecting the disk to computer. Transfer rates in 2001 range from 3 MB per second for the 3600 RPM, 1-inch drives to 65 MB per second for the 15000 RPM, 3.5-inch drives.

Between the disk controller and main memory is a hierarchy of controllers and data paths, whose complexity varies. For example, whenever the transfer time is a small portion of the time of a full access, the designer will want to disconnect the memory device during the access so that other devices can transfer their data. (The default is to hold the datapath for the full access.) This desire is true for high-performance disk controllers, and, as we shall see later, for buses and networks.

There is also a desire to amortize this long access by reading more than simply what is requested; this is called *read ahead*. Read ahead is another case of computer designs trying to leverage spatial locality to enhance performance (see Chapter 5). The hope is that a nearby request will be for the nearby sectors, which will already be available. These sectors go into buffers on the disk that act as a cache. As Figure 7.2 shows, the size of this buffer varies from 0.125 to 4 MB. The hit rate presumably comes solely from spatial locality, but disk-caching algorithms are proprietary and so their techniques and hit rates are unknown. Transfers to and from the buffer operate at the speed of the I/O bus versus the speed of the disk media. In 2001, the I/O bus speeds vary from 80 to 320 MB per second.

To handle the complexities of disconnect/connect and read ahead, there is usually, in addition to the disk drive, a device called a *disk controller*. Thus, the final component of disk-access time is *controller time*, which is the overhead the controller imposes in performing an I/O access. When referring to the performance of a disk in a computer system, the time spent waiting for a disk to become free (*queuing delay*) is added to this time.

EXAMPLE What is the average time to read or write a 512-byte sector for a disk? The advertised average seek time is 5 ms, the transfer rate is 40 MB/second, it rotates at 10000 RPM, and the controller overhead is 0.1 ms. Assume the disk is idle so that there is no queuing delay. In addition, calculate the time assuming the advertised seek time is three times longer than the measured seek time.

ANSWER Average disk access is equal to average seek time + average rotational delay + transfer time + controller overhead. Using the calculated, average seek time, the answer is

$$5 \text{ ms} + \frac{0.5}{10000 \text{ RPM}} + \frac{0.5 \text{ KB}}{40.0 \text{ MB/sec}} + 0.1 \text{ ms} = 5.0 + 3.0 + 0.013 + 0.1 = 8.11 \text{ ms}$$

Assuming the measured seek time is 33% of the calculated average, the answer is

$$1.67 \text{ ms} + 3.0 \text{ ms} + 0.013 \text{ ms} + 0.1 \text{ ms} = 4.783 \text{ ms}$$

Note that only $0.013/4.783$ or 0.3% of the time is the disk transferring data in this example. Even page-sized transfers often take less than 5%, so disks normally spend most of their time waiting for the head to get over the data rather than reading or writing the data.

n

Many disks today are shipped in *disk arrays*. These arrays contain dozens of disks, and may look like a single large disk to the computer. Hence, there is often another level to the storage hierarchy, the *array controller*. They are often key in dependability and performance of storage systems, implementing functions such as RAID (see section 7.5) and caching (see section 7.12).

The Future of Magnetic Disks

The disk industry has concentrated on improving the capacity of disks. Improvement in capacity is customarily expressed as improvement in *areal density*, measured in bits per square inch:

$$\text{Areal density} = \frac{\text{Tracks}}{\text{Inch}} \text{ on a disk surface} \times \frac{\text{Bits}}{\text{Inch}} \text{ on a track}$$

Through about 1988 the rate of improvement of areal density was 29% per year, thus doubling density every three years. Between then and about 1996, the rate improved to 60% per year, quadrupling density every three years and matching the traditional rate of DRAMs. From 1997 to 2001 the rate increased to 100%, or doubling every year. In 2001, the highest density in commercial products is 20 billion bits per square inch, and the lab record is 60 billion bits per square inch.

Cost per gigabyte has dropped at least as fast as areal density has increased, with smaller drives playing the larger role in this improvement. Figure 7.3 on

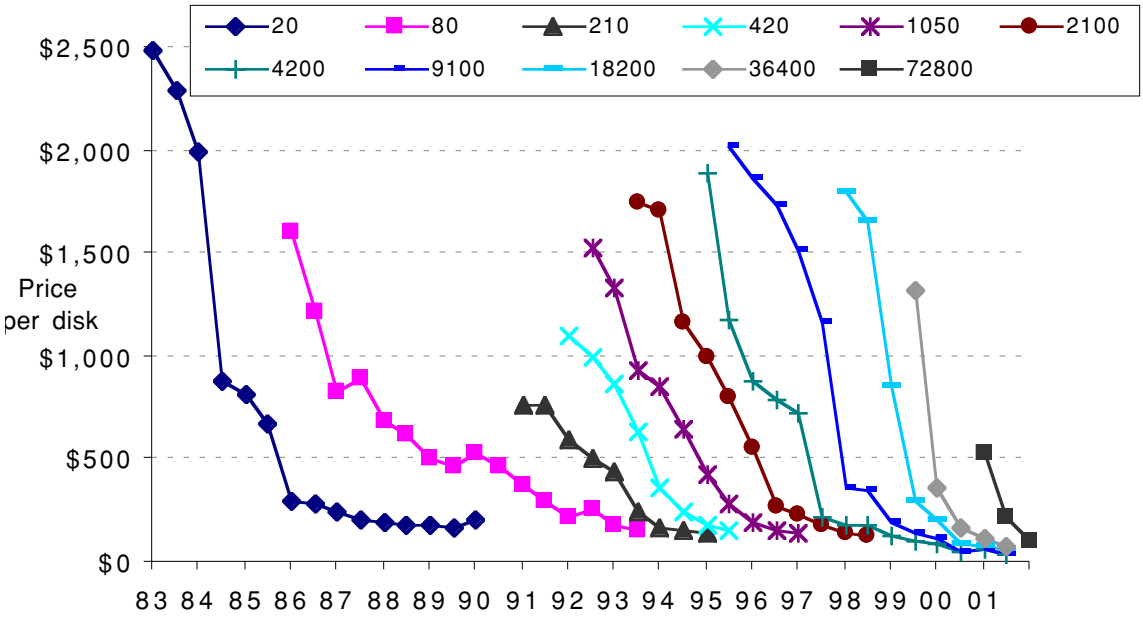


FIGURE 7.3 Price per personal computer disk by capacity (in megabytes) between 1983 and 2001. Note that later the price declines become steeper as the industry increases its rate of improvement from 30% per year to 100% per year. The capacity per disk increased almost 4000 times in 18 years. Although disks come in many sizes, we picked a small number of fixed sizes to show the trends. The price was adjusted to get a consistent disk capacity (e.g., shrinking the price of an 86-MB disk by 80/86 to get a point for the 80-MB line). The prices are in July 2001 dollars, adjusted for inflation using the Producer Price Index for manufacturing industries. The prices through 1995 were collected by Mike Dahlin from advertisements from the January and July editions of Byte magazine, using the lowest price of a disk of a particular size in that issue. Between January 1996 and January 2000, the advertisements come from PC Magazine, as Byte ceased publication. Since July 2000, the results came from biannual samples of pricewatch.com. (See http://www.cs.utexas.edu/users/dahlin/techTrends/data/diskPrices)

page 493 plots price per personal computer disk between 1983 and 2000, showing both the rapid drop in price and the increase in capacity. Figure 7.4 on page 494 above translates these costs into price per gigabyte, showing that it has improved by a factor of 10,000 over those 17 years. Notice the much quicker drop in prices per disk over time, reflecting faster decrease in price per gigabyte.

Because it is more efficient to spin smaller mass, smaller-diameter disks save power as well as volume. In 2001, 3.5-inch or 2.5-inch drives are the leading technology. In the largest drives, rotation speeds have improved from the 3600 RPM standard of the 1980s to 5400–7200 RPM in the 1990s to 10000-15000 RPM in 2001. When combined with increasing density (bits per inch on a track), transfer rates have improved recently by almost 40% per year. There has been some small improvement in seek speed, typically less than 10% per year.

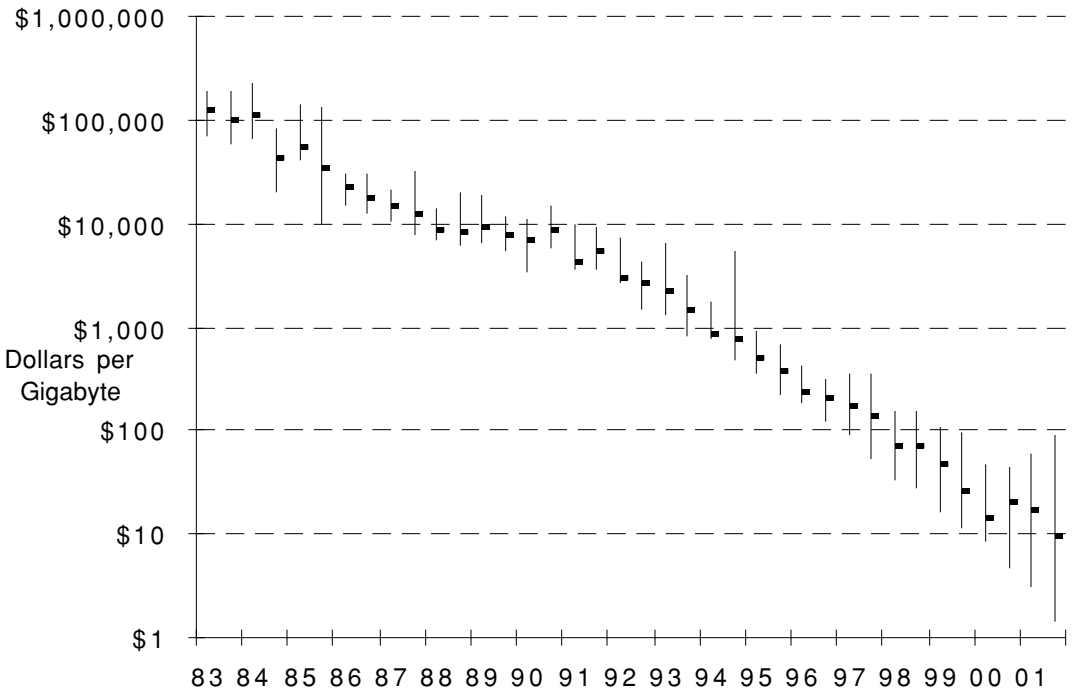


FIGURE 7.4 Price per gigabyte of personal computer disk over time, dropping a factor of 10000 between 1983 and 20001. The center point is the median price per GB, with the low point on the line being the minimum and the high point being the maximum. Note that the graph drops starting in about 1991, and that in January 1997 the spread from minimum to maximum becomes large. This spread is due in part to the increasing difference in price between ATA.IDE and SCSI disks; see section 7.14. The data collection method changed in 2001 to collect more data, which may explain the larger spread between minimum and maximum. These data were collected in the same way as for Figure 7.3, except that more disks are included on this graph. The prices were adjusted for inflation as in Figure 7.3.

Magnetic disks have been challenged many times for supremacy of secondary storage. One reason has been the fabled *access time gap* between disks and DRAM, as shown in Figure 7.5. DRAM latency is about 100,000 times less than disk, although bandwidth is only about 50 times larger. That performance gain costs 100 times more per gigabyte in 2001.

Many have tried to invent a technology cheaper than DRAM but faster than disk to fill that gap, but thus far, all have failed. So far, challengers have never had a product to market at the right time. By the time a new product would ship, DRAMs and disks have made advances as predicted earlier, costs have dropped accordingly, and the challenging product is immediately obsolete.

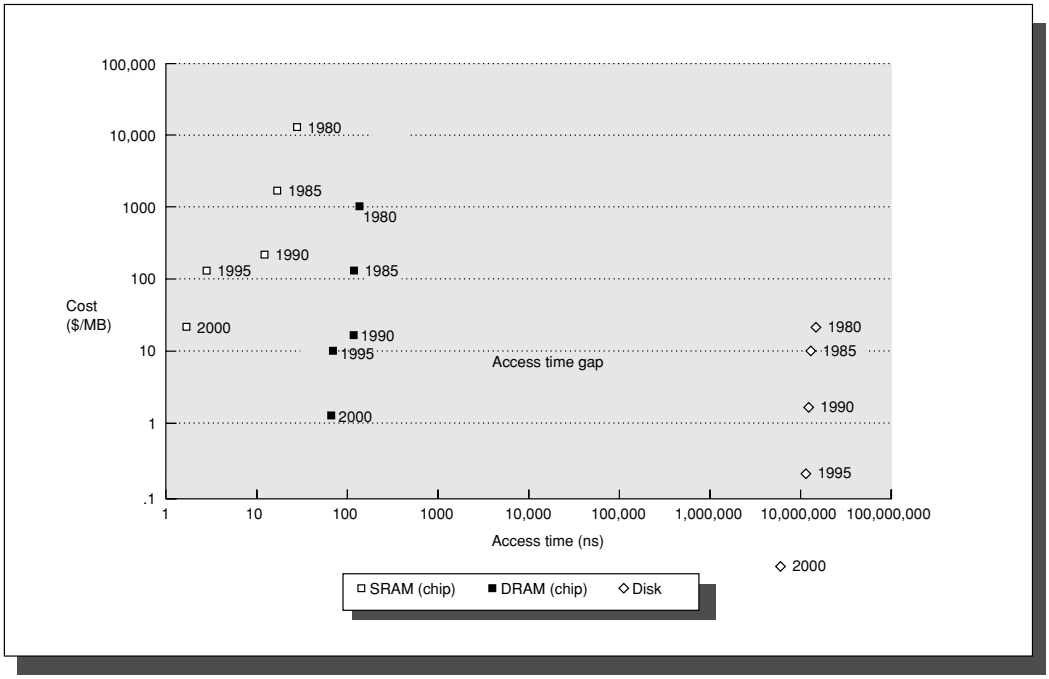


FIGURE 7.5 Cost versus access time for SRAM, DRAM, and magnetic disk in 1980, 1985, 1990, 1995, and 2000. The two-order-of-magnitude gap in cost and access times between semiconductor memory and rotating magnetic disks has inspired a host of competing technologies to try to fill it. So far, such attempts have been made obsolete before production by improvements in magnetic disks, DRAMs, or both. Note that between 1990 and 2000 the cost per megabyte of SRAM and DRAM chips made less improvement, while disk cost made dramatic improvement. <<Note to artist: Need to change the Y-axis scale to go to 0.001, so that can add 2000 disk point at about 0.01 and 8,000,0000 ns.>>

Optical Disks

One challenger to magnetic disks is *optical compact disks*, or *CDs*, and its successor, called *Digital Video Discs* and then *Digital Versatile Discs* or just *DVDs*. Both the *CD-ROM* and *DVD-ROM* are removable and inexpensive to manufacture, but they are read-only mediums. These 4.7-inch diameter disks hold 0.65 and 4.7 GB, respectively, although some DVDs write on both sides to double their capacity. Their high capacity and low cost have led to CD-ROMs and DVD-ROMs replacing floppy disks as the favorite medium for distributing software and other types of computer data.

The popularity of CDs and music that can be downloaded from the WWW led to a market for rewritable CDs, conveniently called CD-RW, and write once CDs, called CD-R. In 2001, there is a small cost premium for drives that can record on CD-RW. The media itself costs about $0.20 per CD-R disk or $0.60 per CD-RW disk. CD-RWs and CD-Rs read at about half the speed of CD-ROMs and CD-RWs and CD-Rs write at about a quarter the speed of CD-ROMs.

The are also write-once and rewritable DVDs, called DVD-R and (alas) DVD-RAM. Rewritable DVD drives cost ten times as much as DVD-ROM drives. The media cost is about $10 per DVD-R disk to $15 per DVD-RAM disk. DVD-RAM reads and writes at about a third of the speed of DVD-ROMs, and DVD-R writes at the speed of DVD-RAM and reads at the speed of DVD-ROMs.

As CDs and DVDs are the replaceable media for the consumer mass market, their rate of improvement is governed by standards committees. It appears that magnetic storage grows more quickly than human beings can agree on standards. Writable optical disks may have the potential to compete with new tape technologies for archival storage, as tape also improves much more slowly than disks.

Magnetic Tapes

Magnetic tapes have been part of computer systems as long as disks because they use the similar technology as disks, and hence historically have followed the same density improvements. The inherent cost/performance difference between disks and tapes is based on their geometries:

- Fixed rotating platters offer random access in milliseconds, but disks have a limited storage area and the storage medium is sealed within each reader.
- Long strips wound on removable spools of "unlimited" length mean many tapes can be used per reader, but tapes require sequential access that can take seconds.

One of the limits of tapes had been the speed at which the tapes can spin without breaking or jamming. A technology called *helical scan tapes* solves this problem by keeping the tape speed the same but recording the information on a diagonal to the tape with a tape reader that spins much faster than the tape is moving. This technology increases recording density by about a factor of 20 to 50. Helical scan tapes were developed for low-cost VCRs and camcorders, which brought down the cost of the tapes and readers.

One drawback to tapes is that they wear out; Helical tapes last for hundreds of passes, while the traditional longitudinal tapes wear out in thousands to millions of passes. The helical scan read/write heads also wear out quickly, typically rated for 2000 hours of continuous use. Finally, there are typically long rewind, eject, load, and spin-up times for helical scan tapes. In the archival backup market, such performance characteristics have not mattered, and hence there has been more engineering focus on increasing density than on overcoming these limitations.

Traditionally, tapes enjoyed a 10X-100X advantage over disks in price per gigabyte, and were the technology of choice for disk backups. In 2001, it appears that tapes are falling behind the rapid advance in disk technology. Whereas in the past the contents of several disks could be stored on a single tape, the largest disk has greater capacity than the largest tapes. Amazingly, the prices of magnetic disks and tape media have crossed: in 2001, the price of a 40 GB IDE disk is about the same as the price of a 40 GB tape!

In the past, the claim was that magnetic tapes must track disks since innovations in disks must help tapes. This claim was important, because tapes are a small market and cannot afford a separate large research and development effort. One reason the market is small is that PC owners generally do not back up disks onto tape, and so while PCs are by far the largest market for disks, PCs are a small market for tapes.

Recently the argument has changed to that tapes have compatibility requirements that are not imposed on disks; tape readers must read or write the current and previous generation of tapes, and must read the last four generations of tapes. As disks are a closed system, the disk heads need only read the platters that are enclosed with them, and this advantage explains why disks are improving at rates that much more rapid.

In addition to the issue of capacity, another challenge is recovery time. Tapes are also not keeping up in bandwidth of disks. Thus, as disks continue to grow, its is not only more expensive to use tapes for backups, it will also take much longer to recover if a disaster occurs.

This growing gap between rate of improvement in disks and tapes calls into question the sensibility of tape backup for disk storage.

Some bold organizations get rid of tapes altogether, using networks and remote disks to replicate the data geographically. The sites are picked so that disasters would not take out both sites, enabling instantaneous recovery time. These sites typically use a file system that does not overwrite data, which allows accidentally discarded files to be recovered. Such a solution depends on advances in disk capacity and network bandwidth to make economic sense, but these two are getting much more investment and hence have better records of accomplishment than tape.

Automated Tape Libraries

Tape capacities are enhanced by inexpensive robots to automatically load and store tapes, offering a new level of storage hierarchy. These *nearline* tapes mean access to terabytes of information in tens of seconds, without the intervention of a human operator. Figure 7.6 shows the Storage Technologies Corporation (STC) PowderHorn, which loads up to 6000 tapes, giving a total capacity of 300 terabytes. Putting this capacity into perspective, the Library of Congress is estimated to have 30 terabytes of text, if books could be magically transformed into ASCII characters.

There are many versions of tape libraries, but these mechanical marvels are not as reliable as other parts of the computer; its not uncommon for tape libraries to have failure rates a factor of 10 higher than other storage devices.

Flash Memory

Embedded devices also need nonvolatile storage, but premiums placed on space and power normally lead to the use of Flash memory instead of magnetic record-

FIGURE 7.6 The StorageTek PowderHorn 9310. This storage silo holds 2000 to 6000 tape cartridges per Library Storage Module (LSM); using the 9840 cartridge, the total uncompressed capacity is 300 terabytes. Each cartridge holds 20 GB of uncompressed data. Depending on the block size and compression, reader transfer at 1.6 to 7.7 MB/second in tests, with a peak speed of 20 MB/second of compressed data. Each LSM has up to 10 tape readers, and can exchange up to 450 cartridges per hour. One LSM is 7.7 feet tall, 10.7 feet in diameter, uses about 1.1 kilowatts, and weighs 8200 pounds. Sixteen LSMs can be linked together to pass cartridges between modules, increasing storage capacity another order of magnitude (Courtesy STC.)

ing. Flash memory is also used as a rewritable ROM in embedded system, typically to allow software to be upgraded without having to replace chips. Applications are typically prohibited from writing to Flash memory in such circumstances.

Like electrically erasable and programmable read-only memories (EEPROM), Flash memory is written by inducing the tunneling of charge from transistor gain to a floating gate. The floating gate acts as a potential well which stores the charge, and the charge cannot move from there without applying an external force. The primary difference between EEPROM and Flash memory is that Flash restricts write to multi-kilobyte blocks, increasing memory capacity per chip by reducing area dedicated to control.

Compared to disks, Flash memories offer low power consumption (less than 50 milliwatts), can be sold in small sizes, and offer read access times comparable to DRAMs. In 2001, a 16 Mbit Flash memory has a 65 ns access time, and a 128 Mbit Flash memory has a 150 ns access time. Some memories even borrow the page mode assesses acceleration from DRAM to bring the time per word down in block transfers to 25 to 40 ns. Unlike DRAMs, writing is much slower and more complicated, sharing characteristics with the older electrically programmable read-only memories (EPROM) and electrically erasable and programmable read-only memories (EEPROM). A block of Flash memory are first electrically erased, and then written with 0s and 1s.

If the logical data is smaller than the Flash block size, the good data that should survive must be copied to another block before the old block can be erased. Thus, information is organized in Flash as linked lists of blocks. Such concerns lead to software that collects good data into fewer blocks so that the rest

can be erased. The linked list structure is also used by some companies to map out bad blocks and offer reduced memory parts at half price rather than discard flawed chips.

The electrical properties of Flash memory are not as well understood as DRAM. Each company's experience, including whether it manufactured EPROM or EEPROM before Flash, affects the organization that it selects. The two basic types of Flash are based on the whether the building blocks for the bits are NOR or NAND gates. NOR Flash devices in 2000 typically take one to two seconds to erase 64 KB to 128 KB blocks, while NAND Flash devices take 5 to 6 millisecond to erase smaller blocks of 4 KB to 8 KB. Programming takes 10 microseconds per byte for NOR devices and 1.5 microseconds per byte for NAND devices. The number of times bits can be erased and still retain information is also often limited, typically about 100,000 cycles for NOR devices and 1,000,000 for some NAND devices.

An example illustrates read and write performance of Flash versus disks.

EXAMPLE Compare the time to read and write a 64-KB block to Flash memory, and magnetic disk. For Flash, assume it takes 65 nanoseconds to read one byte, 1.5 microseconds to write one byte, and 5 milliseconds to erase 4 KB. For disk, use the parameters of the Microdrive in Figure 7.2 on page 490. Assume the measured seek time is one-third of the calculated average, the controller overhead is 0.1 ms, and the data is stored in the outer tracks giving it the fastest transfer rate.

ANSWER Average disk access is equal to average seek time + average rotational delay + transfer time + controller overhead. The average time to read or write 64 KB in a Microdrive disk is:

$$\frac{12 \text{ ms}}{3} + \frac{0.5}{3600 \text{ RPM}} + \frac{64 \text{ KB}}{4.2 \text{ MB/sec}} + 0.1 \text{ ms} = 4.0 + 8.3 + 14.9 + 0.1 = 27.3 \text{ ms}$$

To read 64 KB in Flash you simply divide the 64 KB by the read bandwidth:

$$\text{Flash read time} = \frac{64 \text{ KB}}{1\text{B}/65 \text{ nanoseconds}} = 4,259,840 \text{ ns} = 4.3 \text{ ms}$$

To write 64 KB, first erase it and then divide 64 KB by the write bandwidth:

$$\text{Flash write time} = \frac{64 \text{ KB}}{4\text{KB}/5 \text{ ms}} + \frac{64 \text{ KB}}{1\text{B}/1.5 \text{ microseconds}} = 80 \text{ ms} + 98,304 \text{ us} = 178.3 \text{ ms}$$

Thus, Flash memory is about 6 times faster than disk for reading 64KB, and disk is about 6 times faster than Flash memory for writing 64KB. Note that this example assumes the Microdrive is already operating. If it was powered off to save energy, we should add time for it to resume.

n

The price per megabyte of Flash memory is about 6 times more than DRAM in 2001, making it 600 times more expensive per megabyte than disk. Of course Flash does has its uses, for example when the designer may need only tens of megabytes or less of storage, not provided economically by disks.

Now that we have described several storage devices, we must discover how to connect them to a computer.

7.3 Buses—Connecting I/O Devices to CPU/Memory

In a computer system, the various subsystems must have interfaces to one another; for instance, the memory and CPU need to communicate, and so do the CPU and I/O devices. This communication is commonly done using a *bus*. The bus serves as a shared communication link between the subsystems. The two major advantages of the bus organization are low cost and versatility. By defining a single interconnection scheme, new devices can be added easily and peripherals may even be moved between computer systems that use a common bus. The cost of a bus is low, since a single set of wires is shared among multiple devices.

The major disadvantage of a bus is that it creates a communication bottleneck, possibly limiting the maximum I/O throughput. When I/O must pass through a central bus, this bandwidth limitation is as real as—and sometimes more severe than—memory bandwidth. In server systems, where I/O is frequent, designing a bus system capable of meeting the demands of the processor is a major challenge.

As Moore's Law marches on, buses are increasingly being replaced by networks and switches (see section 7.10). To avoid the bus bottleneck, some I/O devices are connected to computers via *Storage Area Networks* (*SANs*). SANs are covered in the next chapter, so this section concentrates on buses.

One reason bus design is so difficult is that the maximum bus speed is largely limited by physical factors: the length of the bus and the number of devices (and, hence, bus loading). These physical limits prevent arbitrary bus speedup. The desire for high I/O rates (low latency) and high I/O throughput can also lead to conflicting design requirements.

Buses were traditionally classified as *CPU-memory buses* or *I/O buses*. I/O buses may be lengthy, may have many types of devices connected to them, have a wide range in the data bandwidth of the devices connected to them, and normally follow a bus standard. CPU-memory buses, on the other hand, are short, generally high speed, and matched to the memory system to maximize memory-CPU bandwidth. During the design phase, the designer of a CPU-memory bus knows all the types of devices that must connect together, while the I/O bus designer must accept devices varying in latency and bandwidth capabilities. To lower costs, some computers have a single bus for both memory and I/O devices. In the quest for higher I/O performance, some buses are a hybrid of the two. For example, PCI is relatively short, and is used to connect to more traditional I/O buses via bridges that speak both PCI on one end and the I/O bus protocol on the other. To indicate its intermediate state, such buses are sometimes called *mezzanine buses*.

Let's review a typical *bus transaction*, as seen in Figure 7.7. A bus transaction includes two parts: sending the address and receiving or sending the data. Bus transactions are usually defined by what they do to memory: A *read* transaction transfers data *from* memory (to either the CPU or an I/O device), and a *write* transaction writes data to the memory.

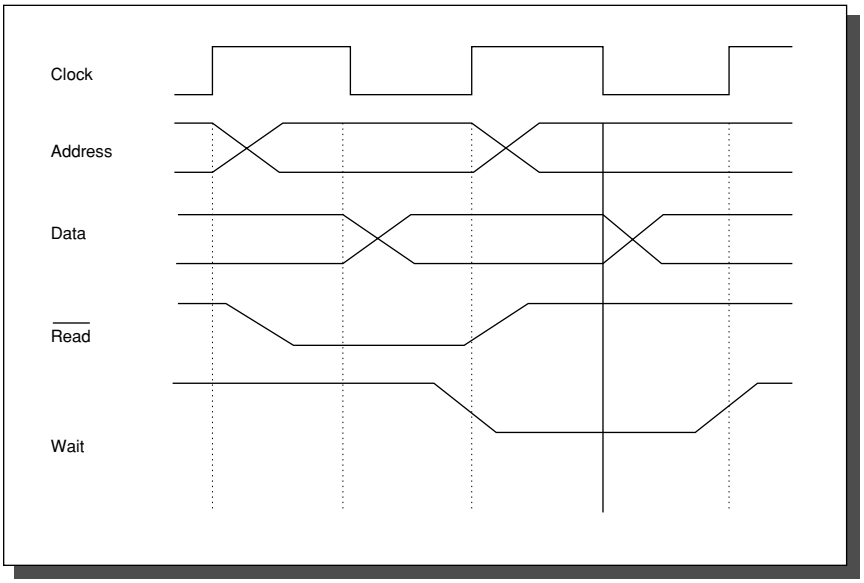


FIGURE 7.7 Typical bus read transaction. The diagonal lines show when the data is changing with respect to the clock signal. This bus is synchronous. The read begins when the Not Read signal is asserted, and data are not ready until the wait signal is deasserted. The vertical bar shows when the data is ready to be read by the CPU.

In a read transaction, the address is first sent down the bus to the memory, together with the appropriate control signals indicating a read. In Figure 7.7, this means asserting the read signal. The memory responds by returning the data on the bus with the appropriate control signals, in this case deasserting the wait signal. A write transaction requires that the CPU or I/O device send both address and data and requires no return of data. Usually the CPU must wait between sending the address and receiving the data on a read, but the CPU often does not wait between sending the address and sending the data on writes.

Bus Design Decisions

The design of a bus presents several options, as Figure 7.8 shows. Like the rest of the computer system, decisions depend on cost and performance goals. The first three options in the figure are clear—separate address and data lines, wider data lines, and multiple-word transfers all give higher performance at more cost.

Option	High performance	Low cost
Bus width	Separate address and data lines	Multiplex address and data lines
Data width	Wider is faster (e.g., 64 bits)	Narrower is cheaper (e.g., 8 bits)
Transfer size	Multiple words have less bus overhead	Single-word transfer is simpler
Bus masters	Multiple (requires arbitration)	Single master (no arbitration)
Split transaction?	Yes—separate request and reply packets get higher bandwidth (need multiple masters)	No—continuous connection is cheaper and has lower latency
Clocking	Synchronous	Asynchronous

FIGURE 7.8 The main options for a bus. The advantage of separate address and data buses is primarily on writes.

The next item in the table concerns the number of *bus masters*. These devices can initiate a read or write transaction; the CPU, for instance, is always a bus master. A bus has multiple masters when there are multiple CPUs or when I/O devices can initiate a bus transaction. If there are multiple masters, an arbitration scheme is required among the masters to decide which one gets the bus next. Arbitration is often a fixed priority for each device, as is the case with daisy-chained devices, or an approximately fair scheme that randomly chooses which master gets the bus.

With multiple masters, a bus can offer higher bandwidth by using packets, as opposed to holding the bus for the full transaction. This technique is called *split transactions*. (Some systems call this ability *connect/disconnect*, a *pipelined bus*, a *pended bus*, or a *packet-switched bus*; the next chapter goes into more detail on packet switching.) Figure 7.9 shows the split-transaction bus. The idea is to divide bus events into requests and replies, so that bus can be used in the time between the request and the reply.

The read transaction is broken into a read-request transaction that contains the address and a memory-reply transaction that contains the data. Each transaction must now be tagged so that the CPU and memory can tell which reply is for which request. Split transactions make the bus available for other masters while the memory reads the words from the requested address. It also normally means that the CPU must arbitrate for the bus to send the data and the memory must arbitrate for the bus to return the data. Thus, a split-transaction bus has higher bandwidth, but it usually has higher latency than a bus that is held during the complete transaction.

The final item in Figure 7.8, *clocking*, concerns whether a bus is synchronous or asynchronous. If a bus is *synchronous,* it includes a clock in the control lines and a fixed protocol for sending address and data relative to the clock. Since little or no logic is needed to decide what to do next, these buses can be both fast and inexpensive. They have two major disadvantages, however. Because of clock-skew problems, synchronous buses cannot be long, and everything on the bus must run at the same clock rate. Some buses allow multiple speed devices on a

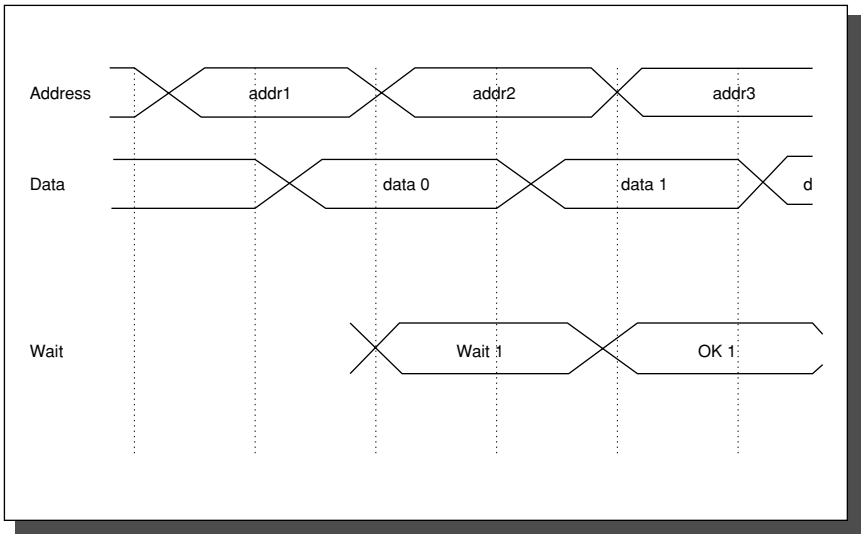


FIGURE 7.9 A split-transaction bus. Here the address on the bus corresponds to a later memory access.

bus, but they all run at the rate of the slowest device. CPU-memory buses are typically synchronous.

An *asynchronous* bus, on the other hand, is not clocked. Instead, self-timed, handshaking protocols are used between bus sender and receiver. Figure 7.10 shows the steps of a master performing a write on an asynchronous bus.

Asynchrony makes it much easier to accommodate a variety of devices and to lengthen the bus without worrying about clock skew or synchronization problems. If a synchronous bus can be used, it is usually faster than an asynchronous bus because it avoids the overhead of synchronizing the bus for each transaction. The choice of synchronous versus asynchronous bus has implications not only for data bandwidth, but also for an I/O system's physical distance and the number of devices that can be connected to the bus. Hence, I/O buses are more likely to be asynchronous than are memory buses. Figure 7.11 suggests when to use one over the other.

Bus Standards

The number and variety of I/O devices is flexible on many computers, permitting customers to tailor computers to their needs. The I/O bus is the interface to which devices are connected. Standards that let the computer designer and I/O-device designer work independently play a large role in buses. As long as both designers meet the requirements, any I/O device can connect to any computer. The I/O bus standard is the document that defines how to connect devices to computers.

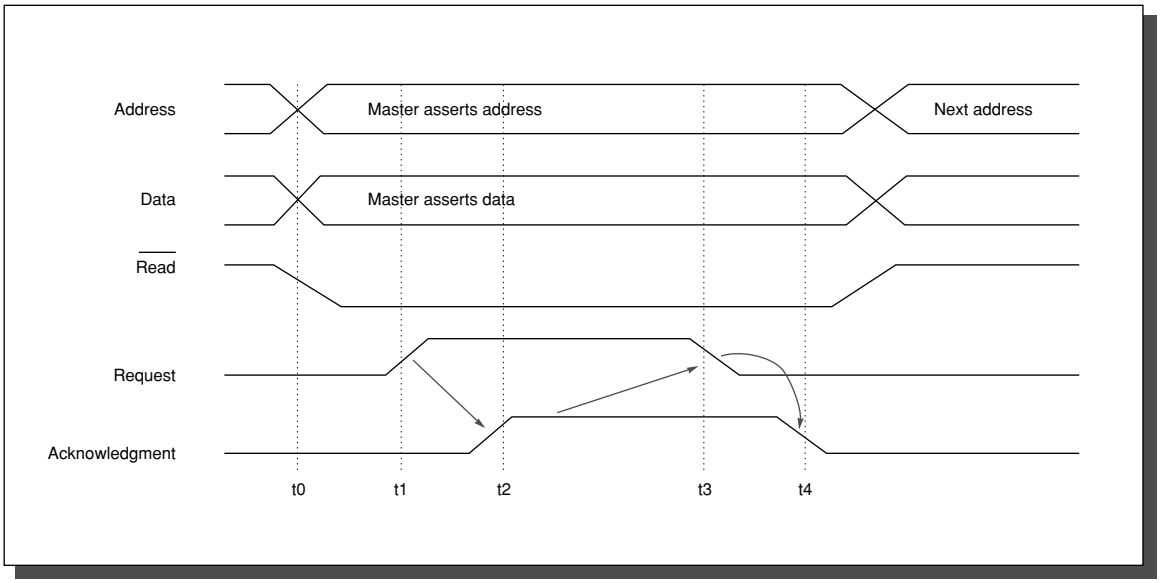


FIGURE 7.10 A master performs a write on an asynchronous bus. The state of the transaction at each time step is as follows. The master has obtained control and asserts address, read/write, and data. It then waits a specified amount of time for slaves to decode target: t1: Master asserts request line; t2: Slave asserts ack, indicating data received; t3: Master releases req; t4: Slave releases ack.

Machines sometimes grow to be so popular that their I/O buses become de facto standards; examples are the PDP-11 Unibus and the IBM PC-AT Bus. Once many I/O devices have been built for a popular machine, other computer designers will build their I/O interface so that those devices can plug into their machines as well. Sometimes standards also come from an explicit standards effort on the part of I/O device makers. Ethernet is an example of a standard that resulted from the cooperation of manufacturers. If standards are successful, they are eventually blessed by a sanctioning body like ANSI or IEEE. A recent variation on traditional standards bodies is trade associations. In that case a limited number of companies agree to produce a standard without cooperating with standards bodies, yet it is still done by committee. PCI is one example of a trade association standard.

Examples of Buses

Figures 7.12 to 7.14 summarize characteristics of common desktop I/O buses, I/O buses found in embedded devices, and CPU-memory interconnects found in servers.

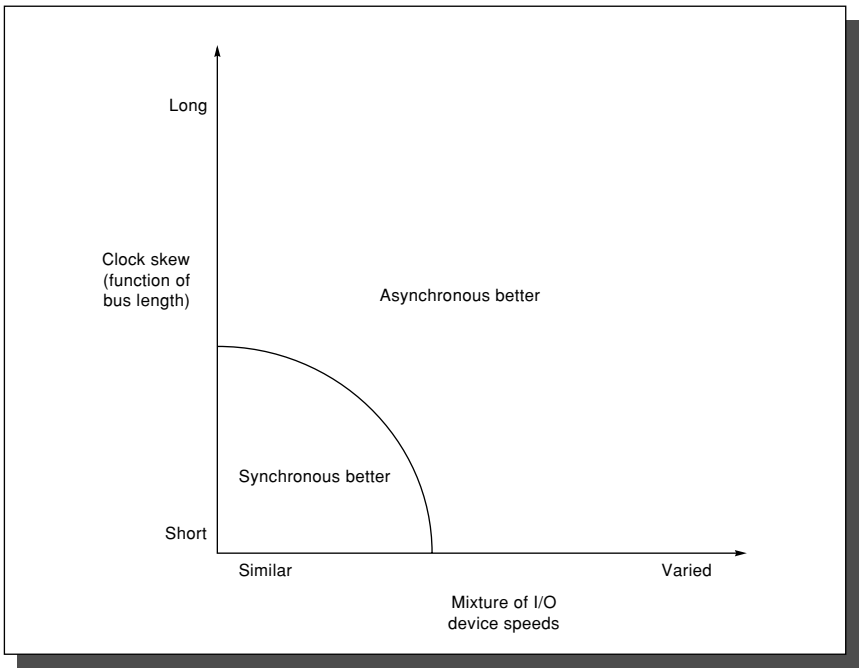


FIGURE 7.11 Preferred bus type as a function of length/clock skew and variation in I/O device speed. Synchronous is best when the distance is short and the I/O devices on the bus all transfer at similar speeds.

	IDE/Ultra ATA	SCSI	PCI	PCI-X
Data width (primary)	16 bits	8 or 16 bits (Wide)	32 or 64 bits	32 or 64 bits
Clock rate	up to 100 MHz	10 MHz (Fast), 20 MHz (Ultra), 40 MHz (Ultra2), 80 MHz (Ultra3 or Ultra160), 160 MHz (ultra4or Ultra320)	33 or 66 MHz	66, 100, 133 MHz
Number of bus masters	1	Multiple	Multiple	Multiple
Bandwidth, peak	200 MB/sec	320 MB/sec	533 MB/sec	1066 MB/sec
Clocking	Asynchronous	Asynchronous	Synchronous	Synchronous
Standard	—	ANSI X3.131	—	—

FIGURE 7.12 Summary of parallel I/O buses. Peripheral Component Interconnect (PCI) and PCI Extended (PCI-X) connect main memory to peripheral devices. IDE/ATA and SCSI compete as interfaces to storage devices. *IDE*, or *Integrated Drive Electronics*, is an early disk standard that connects two disks to a PC. It has been extended by *AT-bus Attachment* (*ATA*), to be both wider and faster. *Small Computer System Interconnect* (*SCSI*) connects up to 7 devices for 8-bit busses and up to 15 devices for 16-bit busses. They can even be different speeds, but they run at the rate of the slowest device. The peak bandwidth of a SCIS bus is the width (1 or 2 bytes) times the clock rate (10 to 160 MHz). Most SCSI buses today are 16-bits.

	I^2C	1-wire	RS232	SPI
Data width (primary)	1 bit	1 bit	2 bits	1 bit
Signal Wires	2	1	9 or 25	3
Clock rate	0.4 to 10 MHz	Asynchronous	0.040 MHz or asynchronous	asynchronous
Number of bus masters	Multiple	Multiple	Multiple	Multiple
Bandwidth, peak	0.4 to 3.4 Mbit/sec	0.014 Mbit/sec	0.192 Mbit/sec	1 Mbit/sec
Clocking	Asynchronous	Asynchronous	Asynchronous	Asynchronous
Standard	None	None	ElA, ITU-T V.21	None

FIGURE 7.13 Summary of serial I/O buses, often used in embedded computers. I^2C was invented by Phillips in the early 1980s. 1-wire was developed by Dallas Semiconductor. RS-232 was introduced in 1962. SPI was created by Motorola in the early 1980s.

	HP HyperPlane Crossbar	IBM SP	Sun Gigaplane-XB
Data width (primary)	64 bits	128 bits	128 bits
Clock rate	120 MHz	111 MHz	83.3 MHz
Number of bus masters	Multiple	Multiple	Multiple
Bandwidth per port, peak	960 MB/sec	1,700 MB/sec	1,300 MB/sec
Bandwidth total, peak	7,680 MB/sec	14,200 MB/sec	10,667 MB/sec
Clocking	Synchronous	Synchronous	Synchronous
Standard	None	None	None

FIGURE 7.14 Summary of CPU-memory interconnects found in 2000 servers. These servers use crossbars switches to connect nodes processors together instead of a shared bus interconnect. Each bus connects up to four processors and memory controllers, and then the crossbar connects the busses together. The number of slots in the crossbar is 16, 8, and 16, respectively.

Interfacing Storage Devices to the CPU

Having described I/O devices and looked at some of the issues of the connecting bus, we are ready to discuss the CPU end of the interface. The first question is where the physical connection of the I/O bus should be made. The two choices are connecting the bus to memory or to the cache. In this section, we examine the more usual case in which the I/O bus is connected to the main memory bus. Figure 7.15 shows a typical organization for desktops. In low-cost systems, the I/O bus *is* the memory bus; this means an I/O command on the bus could interfere with a CPU instruction fetch, for example.

Once the physical interface is chosen, the question becomes: How does the CPU address an I/O device that it needs to send or receive data? The most common practice is called *memory-mapped* I/O. In this scheme, portions of the machine's address space are assigned to I/O devices. Reads and writes to those

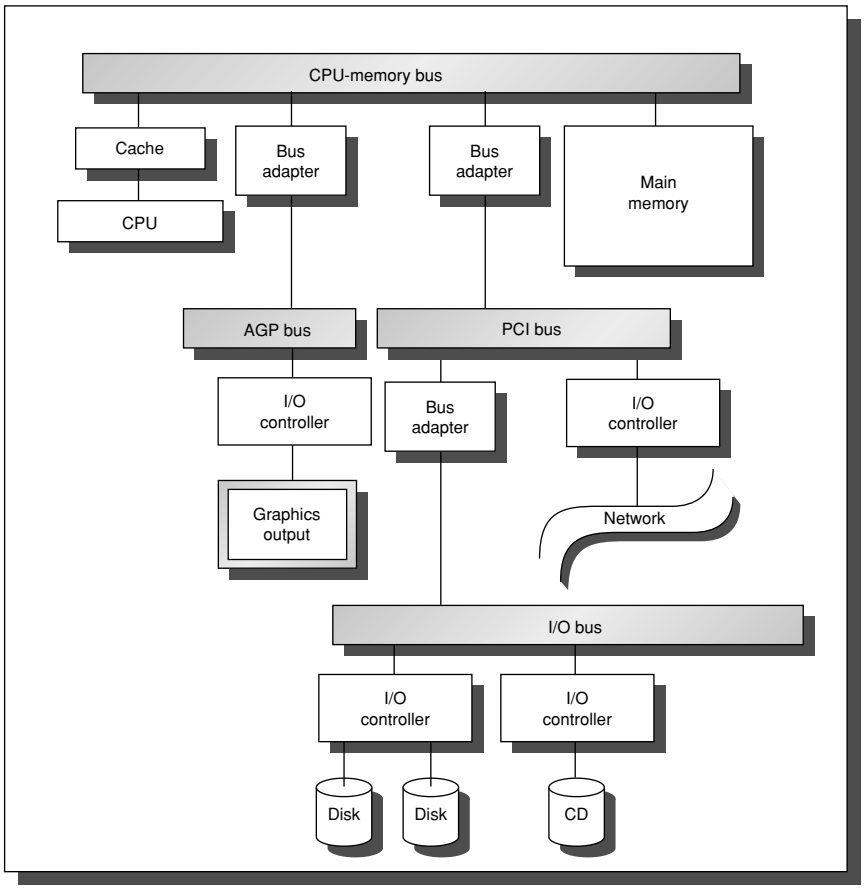


FIGURE 7.15 A typical interface of I/O devices and an I/O bus to the CPU-memory bus.

addresses may cause data to be transferred; some portion of the I/O space may also be set aside for device control, so commands to the device are just accesses to those memory-mapped addresses.

The alternative practice is to use dedicated I/O opcodes in the CPU. In this case, the CPU sends a signal that this address is for I/O devices. Examples of computers with I/O instructions are the Intel 80x86 and the IBM 370 computers. I/O opcodes have been waning in popularity.

No matter which addressing scheme is selected, each I/O device has registers to provide status and control information. Through either loads and stores in memory-mapped I/O or through special instructions, the CPU sets flags to determine the operation the I/O device will perform.

Any I/O event is rarely a single operation. For example, the DEC LP11 line printer has two I/O device registers: one for status information and one for data to be printed. The status register contains a *done bit*, set by the printer when it has printed a character, and an *error bit*, indicating that the printer is jammed or out of paper. Each byte of data to be printed is put into the data register; the CPU must then wait until the printer sets the done bit before it can place another character in the buffer.

This simple interface, in which the CPU periodically checks status bits to see if it is time for the next I/O operation, is called *polling*. As you might expect, CPUs are so much faster than I/O devices that polling may waste a lot of CPU time. A huge fraction of the CPU cycles must be dedicated to interrogating the I/O device rather than performing useful computation. This inefficiency was recognized long ago, leading to the invention of interrupts that notify the CPU when it is time to service the I/O device.

Interrupt-driven I/O, used by most systems for at least some devices, allows the CPU to work on some other process while waiting for the I/O device. For example, the LP11 has a mode that allows it to interrupt the CPU whenever the done bit or error bit is set. In general-purpose applications, interrupt-driven I/O is the key to multitasking operating systems and good response times.

The drawback to interrupts is the operating system overhead on each event. In real-time applications with hundreds of I/O events per second, this overhead can be intolerable. One hybrid solution for real-time systems is to use a clock to periodically interrupt the CPU, at which time the CPU polls all I/O devices.

Delegating I/O Responsibility from the CPU

We approached the task by starting with a simple scheme and then adding commands and features that we felt would enhance the power of the machine. Gradually the [display] processor became more complex. ... Finally the display processor came to resemble a full-fledged computer with some special graphics features. And then a strange thing happened. We felt compelled to add to the processor a second, subsidiary processor, which, itself, began to grow in complexity. It was then that we discovered the disturbing truth. Designing a display processor can become a never-ending cyclical process. In fact, we found the process so frustrating that we have come to call it the "wheel of reincarnation."

Ivan Sutherland, considered the father of computer graphics (1968)

Interrupt-driven I/O relieves the CPU from waiting for every I/O event, but many CPU cycles are still spent in transferring data. Transferring a disk block of 2048 words, for instance, would require at least 2048 loads from disk to CPU registers and 2048 stores from CPU registers to memory, as well as the overhead for the interrupt. Since I/O events so often involve block transfers, *direct memory access* (DMA) hardware is added to many computer systems to allow transfers of numbers of words without intervention by the CPU.

The DMA hardware is a specialized processor that transfers data between memory and an I/O device while the CPU goes on with other tasks. Thus, it is external to the CPU and must act as a master on the bus. The CPU first sets up the DMA registers, which contain a memory address and number of bytes to be transferred. More sophisticated DMA devices support *scatter/gather*, whereby a DMA device can write or read data from a list of separate addresses. Once the DMA transfer is complete, the DMA controller interrupts the CPU. There may be multiple DMA devices in a computer system; for example, DMA is frequently part of the controller for an I/O device.

Increasing the intelligence of the DMA device can further unburden the CPU. Devices called *I/O processors* (or *channel controllers*) operate either from fixed programs or from programs downloaded by the operating system. The operating system typically sets up a queue of *I/O control blocks* that contain information such as data location (source and destination) and data size. The I/O processor then takes items from the queue, doing everything requested and sending a single interrupt when the task specified in the I/O control blocks is complete. Whereas the LP11 line printer would cause 4800 interrupts to print a 60-line by 80-character page, an I/O processor could save 4799 of those interrupts.

I/O processors are similar to multiprocessors in that they facilitate several processes being executed simultaneously in the computer system. I/O processors are less general than CPUs, however, since they have dedicated tasks, and thus the parallelism they enable is much more limited. In addition, an I/O processor doesn't normally change information, as a CPU does, but just moves information from one place to another.

Embedded computers are characterized by a rich variety of DMA devices and I/O controllers. For example, Figure 7.16 shows the Au1000, a MIPS processor for embedded applications, which includes about 10 DMA channels and 20 I/O device controllers on chip.

Now that we have covered the basic types of storage devices and ways to connect them to the CPU, we are ready to look at ways to evaluate the performance of storage systems.

7.4 Reliability, Availability, and Dependability

Whereas people may be willing to live with a computer that occasionally crashes and forces all programs to be restarted, they insist that their information is never lost. The prime directive for storage is then to remember information, no matter what happens.

One persistent shortcoming with the general topic of making computers systems that can survive component faults has been confusion over terms. Conse-

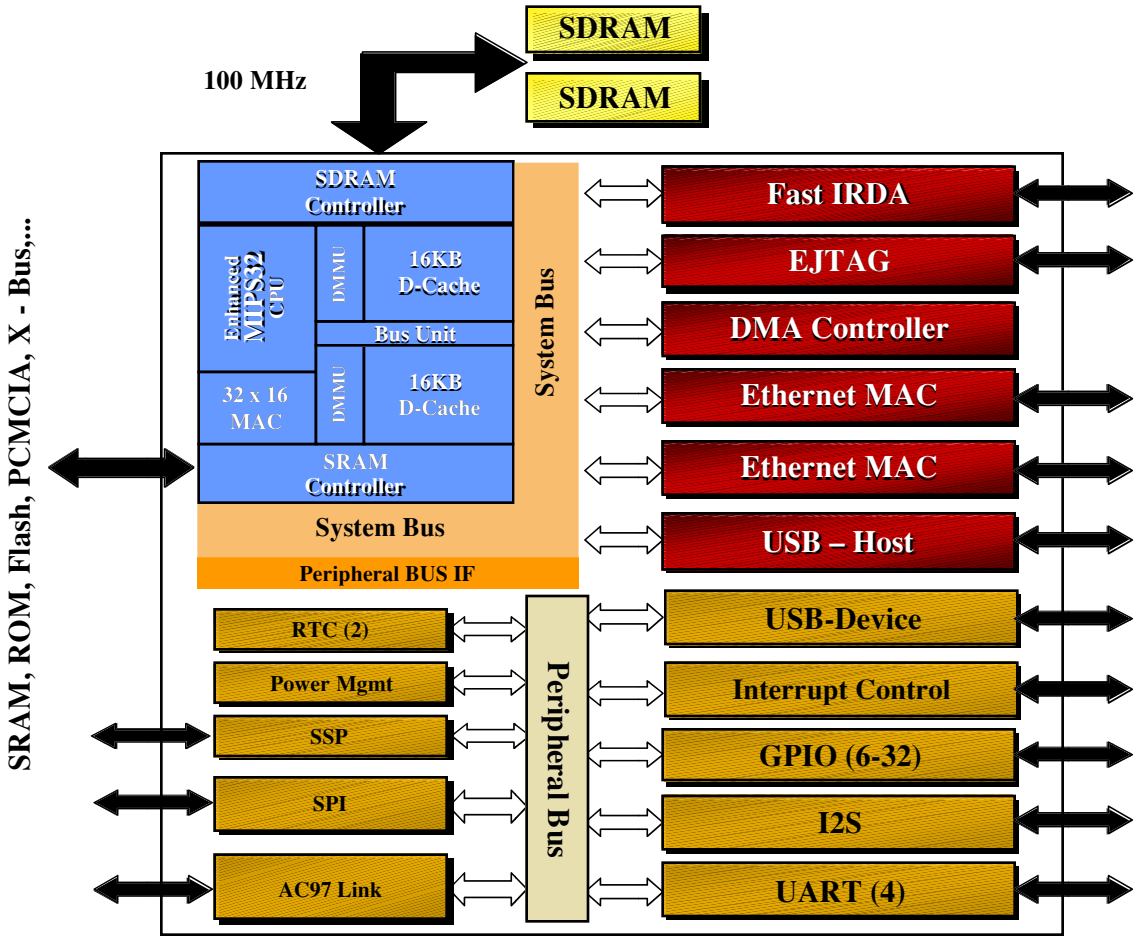


FIGURE 7.16 The Alchemy Semiconductor Au1000. Embedded devices typically have many DMAs and I/O interconnections, as illustrated in the Au1000. Eight DMA channels are included along with a separate IrDA DMA controller for networking. On chip controllers include an SDRAM memory controller, a static RAM controller, two Ethernet MAC layer controllers, USB host and device controllers, two interrupt controllers, two 32-bit GPIO buses, and several embedded bus controllers: four UARTs, a SPI, a SSP, a I2S, and a AC97. This MIPS32 core operates from 200 MHz, at 1.25V and 200 mW for the whole chip, to 500 MHz, at 1.8V and 900 mW. The on-chip system bus operates at 1/2 to 1/5 of the MIPS core clock rate.

quently, perfectly good words like reliability and availability have been abused over the years so that their precise meaning is unclear.

Here are some examples of the difficulties. Is a programming mistake a fault, error, or failure? Does it matter whether we are talking about when it was designed, or when the program is run? If the running program doesn't exercise the mistake, is it still a fault/error/failure? Try another one. Suppose an alpha particle

hits a DRAM memory cell. Is it a fault/error/failure if it doesn't change the value? Is it a fault/error/failure if the memory doesn't access the changed bit? Did a fault/error/failure still occur if the memory had error correction and delivered the corrected value to the CPU? A third example is a mistake by a human operator. Again, the same issues arise about data change, latency, and observability. You get the drift of the difficulties.

Clearly, we need precise definitions to discuss about such events intelligently.

Defining Failure

To avoid such imprecision, this subsection is based on the terminology used by Laprie [1985] and Gray and Siewiorek [1991], endorsed by IFIP working group 10.4 and the IEEE Computer Society Technical Committee on Fault Tolerance. We talk about a system as a single module, but the terminology applies to submodules recursively.

Laprie picked a new term—*dependability*— to have a clean slate to work with:

> Computer system *dependability* is the quality of delivered service such that reliance can justifiably be placed on this service. The *service* delivered by a system is its observed *actual behavior* as perceived by other system(s) interacting with this system's users. Each module also has an ideal *specified behavior*, where a *service specification* is an agreed description of the expected behavior. A system *failure* occurs when the actual behavior deviates from the specified behavior. The failure occurred because an *error*, a defect in that module. The cause of an error is a *fault*.
>
> When a fault occurs it creates a *latent error*, which becomes *effective* when it is activated; when the error actually affects the delivered service, a failure occurs. The time between the occurrence of an error and the resulting failure is the *error latency*. Thus, an error is the manifestation *in the system* of a fault, and a failure is the manifestation *on the service* of an error.

Let's go back to our motivating examples above. A programming mistake is a *fault*; the consequence is an *error* (or *latent error*) in the software; upon activation, the error becomes *effective*; when this effective error produces erroneous data which affect the delivered service, a *failure* occurs. An alpha particle hitting a DRAM can be considered a fault; if it changes the memory, it creates an error; the error will remain latent until the effected memory word is read; if the effective word error affects the delivered service, a failure occurs. (If ECC corrected the error, a failure would not occur.) A mistake by a human operator is a fault; the resulting altered data is an error; it is latent until activated; and so on as before.

To clarify, the relation between faults, errors, and failures is:

- A fault creates one or more latent errors.
- The properties of errors are a) a latent error becomes effective once activated;

b) an error may cycle between its latent and effective states; c) an effective error often propagates from one component to another, thereby creating new errors. Thus, an effective error is either a formerly latent error in that component or it has propagated from another error in that component or from elsewhere.

- A component failure occurs when the error affects the delivered service.
- These properties are recursive, and apply to any component in the system.

We can now return to see how Laprie defines reliability and availability. Users perceive a system alternating between two states of delivered service with respect to the service specification:

1. *Service accomplishment*, where the service is delivered as specified,
2. *Service interruption*, where the delivered service is different from the specified service.

Transitions between these two states are caused by failures (from state 1 to state 2) or *restorations* (2 to 1). Quantifying these transitions lead to the two main measures of dependability:

1. *Module reliability* is a measure of the continuous service accomplishment (or, equivalently, of the time to failure) from a reference initial instant. Hence, the Mean Time To Failure (MTTF) of disks in Figure 7.2 on page 490 is a reliability measure. The reciprocal of MTTF is a rate of failures. If a collection of modules have exponentially distributed lifetimes (see section 7.7), the overall failure rate of the collection is the sum of the failure rates of the modules. Service interruption is measured as Mean Time To Repair (MTTR).
2. *Module availability* is a measure of the service accomplishment with respect to the alternation between the two states of accomplishment and interruption. For non-redundant systems with repair, module availability is statistically quantified as:

$$\text{Module availability} = \frac{\text{MTTF}}{(\text{MTTF} + \text{MTTR})}$$

Note that reliability and availability are now quantifiable metrics, rather than synonyms for dependability. *Mean Time Between Failures* (*MTBF*) is simply the sum of MTTF + MTTR. Although MTBF is widely used, MTTF is often the more appropriate term.

EXAMPLE Assume a disk subsystem with the following components and MTTF:

- 10 disks, each rated at 1,000,000 hour MTTF;
- 1 SCSI controller, 500,000 hour MTTF
- 1 power supply, 200,000 hour MTTF
- 1 fan, 200,000 hour MTTF
- 1 SCSI cable, 1,000,000 hour MTTF

Using the simplifying assumption that the components lifetimes are exponentially distributed–which means that the age of the component is not important in probability of failure–and that failures are independent, compute the MTTF of the system as a whole.

ANSWER The sum of the failure rates is:

$$\text{Failure Rate}_{\text{system}} = 10 \times \frac{1}{1000000} + \frac{1}{500000} + \frac{1}{200000} + \frac{1}{200000} + \frac{1}{1000000} = \frac{10 + 2 + 5 + 5 + 1}{1000000\text{hours}} = \frac{23}{1000000\text{hours}}$$

The MTTF for the system is just the inverse of the failure rate

$$\text{MTTF}_{\text{system}} = \frac{1}{\text{Failure Rate}_{\text{system}}} = \frac{1000000\text{hours}}{23} = 43500 \text{ hours}$$

or just under 5 years.

Classifying faults and fault tolerance techniques may aid with understanding. Gray and Siewiorek classify faults into four categories according to their cause:

1. *Hardware faults*: devices that fail.
2. *Design faults*: faults in software (usually) and hardware design (occasionally).
3. *Operation faults*: mistakes by operations and maintenance personnel.
4. *Environmental faults*: fire, flood, earthquake, power failure, and sabotage.

Faults are also classified by their duration into transient, intermittent, and permanent [Nelson 1990]. *Transient faults* exist for a limited time and are not recurring. *Intermittent faults* cause a system to oscillate between faulty and fault free operation. *Permanent faults* do not correct themselves with passing of time.

Gray and Siewiorek divide improvements in module reliability into *valid construction* and *error correction*. Validation removes faults before the module is completed, ensuring that the module conforms to its specified behavior. Error correction occurs by having redundancy in designs to tolerate faults. *Latent error*

processing describes the practice of trying to detect and repair errors before they become effective, such as preventative maintenance. *Effective error processing* describes correction of the error after it becomes effective, either by *masking* the error or by *recovering* from the error. Error correction, such as that used in disk sectors, can mask errors. Error recovery is either *backward*–returning to a previous correct state, such as with checkpoint-restart–or *forward*–constructing a new correct state, such as by resending a disk block.

Taking a slightly different view, Laprie divides reliability improvements into four methods:

1. *Fault avoidance*: how to prevent, by *construction*, fault occurrence;
2. *Fault tolerance*: how to provide, by *redundancy*, service complying with the service specification in spite of faults having occurred or are occurring;
3. *Error removal*: how to minimize, by *verification*, the presence of latent errors;
4. *Error forecasting*: how to estimate, by *evaluation*, the presence, creation, and consequences of errors.

7.5 RAID: Redundant Arrays of Inexpensive Disks

An innovation that improves both dependability and performance of storage systems is *disk arrays*. One argument for arrays is that potential throughput can be increased by having many disk drives and, hence, many disk arms, rather than one large drive with one disk arm. For example, upcoming Figure 7.32 on page 544 shows how NFS throughput increases as the systems expand from 67 disks to 433 disks. Simply spreading data over multiple disks, called *striping*, automatically forces accesses to several disks. (Although arrays improve throughput, latency is not necessarily improved.) The drawback to arrays is that with more devices, dependability decreases: N devices generally have $1/N$ the reliability of a single device.

Although a disk array would have more faults than a smaller number of larger disks when each disk has the same reliability, dependability can be improved by adding redundant disks to the array to tolerate faults. That is, if a single disk fails, the lost information can be reconstructed from redundant information. The only danger is in having another disk fail between the time the first disk fails and the time it is replaced (termed *mean time to repair,* or MTTR). Since the *mean time to failure* (MTTF) of disks is tens of years, and the MTTR is measured in hours, redundancy can make the measured reliability of 100 disks much higher than that of a single disk. These systems have become known by the acronym *RAID*, standing originally for *redundant array of inexpensive disks*, although some have renamed it to *redundant array of independent disks* (see section 7.16).

The several approaches to redundancy have different overhead and performance. Figure 7.17 shows the standard RAID levels. It shows how eight disks of user data must be supplemented by redundant or check disks at each RAID level. It also shows the minimum number of disk failures that a system would survive.

RAID level		Minimum number of Disk faults survived	Example Data disks	Corresponding Check disks	Corporations producing RAID products at this level
0	Non-redundant striped	0	8	0	Widely used
1	Mirrored	1	8	8	EMC, Compaq (Tandem), IBM
2	Memory-style ECC	1	8	4	
3	Bit-interleaved parity	1	8	1	Storage Concepts
4	Block-interleaved parity	1	8	1	Network Appliance
5	Block-interleaved distributed parity	1	8	1	Widely used
6	P+Q redundancy	2	8	2	

FIGURE 7.17 RAID levels, their fault tolerance, and their overhead in redundant disks. The paper that introduced the term RAID [Patterson, Gibson, and Katz 1987] used a numerical classification that has become popular. In fact, the non-redundant disk array is often called RAID 0, indicating the data is striped across several disks but without redundancy. Note that mirroring (RAID 1) in this instance can survive up to 8 disk failures provided only one disk of each mirrored pair fails; worst case is both disks in a mirrored pair. RAID 6 has a regular, RAID 5 parity block across drives along with a second parity block on another drive. RAID 6 allows failure of any two drives, which is beyond the survival capability of a RAID 5. In 2001, there may be no commercial implementations of RAID 2 or RAID 6; the rest are found in a wide range of products. RAID 0+1, 1+0, 01, and 10 are discussed in the text below.

One problem is discovering when a disk faults. Fortunately, magnetic disks provide information about their correct operation. As mentioned in section 7.2, extra check information is recorded in each sector to discover errors within that sector. As long as we transfer at least one sector and check the error detection information when reading sectors, electronics associated with disks will with very high probability discover when a disk fails or loses information.

Another issue in the design of RAID systems is decreasing the mean time to repair. This reduction is typically done by adding *hot spares* to the system: extra disks that are not used in normal operation. When a failure occurs on an active disk in a RAID, an idle hot spare is first pressed into service. The data missing from the failed disk is then reconstructed onto the hot spare using the redundant data from the other RAID disks. If this process is performed automatically, MTTR is significantly reduced because waiting for the operator in the repair process is no longer the pacing item (see section 7.9).

A related issue is *hot swapping*. Systems with hot swapping allow components to be replaced shutting down the computer. Hence, a system with hot spares and hot swapping need never go off-line; the missing data is constructed immediately onto spares and the broken component is replaced to replenish the spare pool.

We cover here the most popular of these RAID levels; readers interested in more detail should see the paper by Chen et al. [1994].

No Redundancy (RAID 0)

This notation is refers to a disk array in which data is striped but there is no redundancy to tolerate disk failure. Striping across a set of disks makes the collection appear to software as a single large disk, which simplifies storage management. It also improves performance for large accesses, since many disks can operate at once. Video editing systems, for example, often stripe their data.

RAID 0 something of a misnomer as there is no redundancy, it is not in the original RAID taxonomy, and striping predates RAID. However, RAID levels are often left to the operator to set when creating a storage system, and RAID 0 is often listed as one of the options. Hence, the term RAID 0 has become widely used.

Mirroring (RAID 1)

This traditional scheme for tolerating disk failure, called *mirroring* or *shadowing*, uses twice as many disks as does RAID 0. Whenever data is written to one disk, that data is also written to a redundant disk, so that there are always two copies of the information. If a disk fails, the system just goes to the "mirror" to get the desired information. Mirroring is the most expensive RAID solution, since it requires the most disks.

One issue is how mirroring interacts with striping. Suppose you had, say, four disks worth of data to store and eight physical disks to use. Would you create four pairs of disks–each organized as RAID 1–and then stripe data across the four RAID 1 pairs? Alternatively, would you create two sets of four disks–each organized as RAID 0–and then mirror writes to both RAID 0 sets? The RAID terminology has evolved to call the former RAID 1+0 or RAID 10 ("striped mirrors") and the latter RAID 0+1 or RAID 01 ("mirrored stripes").

Bit-Interleaved Parity (RAID 3)

The cost of higher availability can be reduced to $1/N$, where N is the number of disks in a *protection group*. Rather than have a complete copy of the original data for each disk, we need only add enough redundant information to restore the lost information on a failure. Reads or writes go to all disks in the group, with one extra disk to hold the check information in case there is a failure. RAID 3 is popular in applications with large data sets, such as multimedia and some scientific codes.

Parity is one such scheme. Readers unfamiliar with parity can think of the redundant disk as having the sum of all the data in the other disks. When a disk fails, then you subtract all the data in the good disks from the parity disk; the remaining information must be the missing information. Parity is simply the sum modulo two. The assumption behind this technique is that failures are so rare that taking longer to recover from failure but reducing redundant storage is a good trade-off.

Just as direct-mapped associative placement in caches can be considered a special case of set-associative placement (see section 5.2), the mirroring can be considered the special case of one data disk and one parity disk ($N = 1$). Parity can be accomplished in this case by duplicating the data, so mirrored disks have the advantage of simplifying parity calculation. Duplicating data also means that the controller can improve read performance by reading from the disk of the pair that has the shortest seek distance. This optimization means the arms are no longer synchronized, however, and thus writes must now wait for the arm with the longer seek. Of course, the redundancy of $N = 1$ has the highest overhead for increasing disk availability.

Block-Interleaved Parity and Distributed Block-Interleaved Parity (RAID 4 and RAID 5)

Both these levels use the same ratio of data disks and check disks as RAID 3, but they access data differently. The parity is stored as blocks and associated with a set of data blocks.

In RAID 3, every access went to all disks. Some applications would prefer to do smaller accesses, allowing independent accesses to occur in parallel. That is the purpose of the next RAID levels. Since error-detection information in each sector is checked on reads to see if data is correct, such "small reads" to each disk can occur independently as long as the minimum access is one sector.

Writes are another matter. It would seem that each small write would demand that all other disks be accessed to read the rest of the information needed to recalculate the new parity, as in Figure 7.18. A "small write" would require reading the old data and old parity, adding the new information, and then writing the new parity to the parity disk and the new data to the data disk.

The key insight to reduce this overhead is that parity is simply a sum of information; by watching which bits change when we write the new information, we need only change the corresponding bits on the parity disk. Figure 7.18 shows the shortcut. We must read the old data from the disk being written, compare old data to the new data to see which bits change, read the old parity, change the corresponding bits, then write the new data and new parity. Thus, the small write involves four disk accesses to two disks instead of accessing all disks. This organization is RAID 4.

RAID 4 efficiently supports a mixture of large reads, large writes, small reads, and small writes. One drawback to the system is that the parity disk must be up-

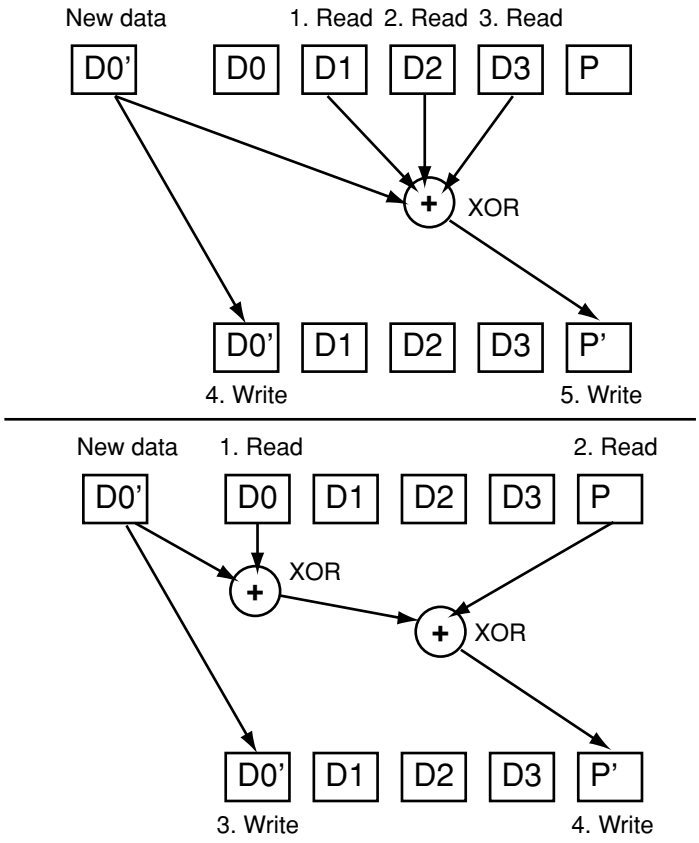


FIGURE 7.18 Small write update on RAID 3 vs. RAID 4/ RAID5. This optimization for small writes reduces the number of disk accesses as well as the number of disks occupied. This figure assumes we have four blocks of data and one block of parity. The straightforward RAID 3 parity calculation at the top of the figure reads blocks D1, D2, and D3 before adding block D0' to calculate the new parity P'. (In case you were wondering, the new data D0' comes directly from the CPU, so disks are not involved in reading it.) The RAID 4/ RAID 5shortcut at the bottom reads the old value D0 and compares it to the new value D0' to see which bits will change. You then read to old parity P and then change the corresponding bits to form P'. The logical function exclusive or does exactly what we want. This example replaces 3 disks reads (D1, D2, D3) and 2 disk writes (D0',P') involving all the disks for 2 disk reads (D0,P) and 2 disk writes (D0',P') which involve just 2 disks. Increasing the size of the parity group increases the savings of the shortcut.

dated on every write, so it is the bottleneck for back-to-back writes. To fix the parity-write bottleneck, the parity information can be spread throughout all the disks so that there is no single bottleneck for writes. The distributed parity organization is RAID 5.

Figure 7.19 shows how data are distributed in RAID 4 vs. RAID 5. As the organization on the right shows, in RAID 5 the parity associated with each row of data blocks is no longer restricted to a single disk. This organization allows multiple writes to occur simultaneously as long as the stripe units are not located in

the same disks. For example, a write to block 8 on the right must also access its parity block P2, thereby occupying the first and third disks. A second write to block 5 on the right, implying an update to its parity block P1, accesses the second and fourth disks and thus could occur at the same time as the write to block 8. Those same writes to the organization on the left would result in changes to blocks P1 and P2, both on the fifth disk, which would be a bottleneck.

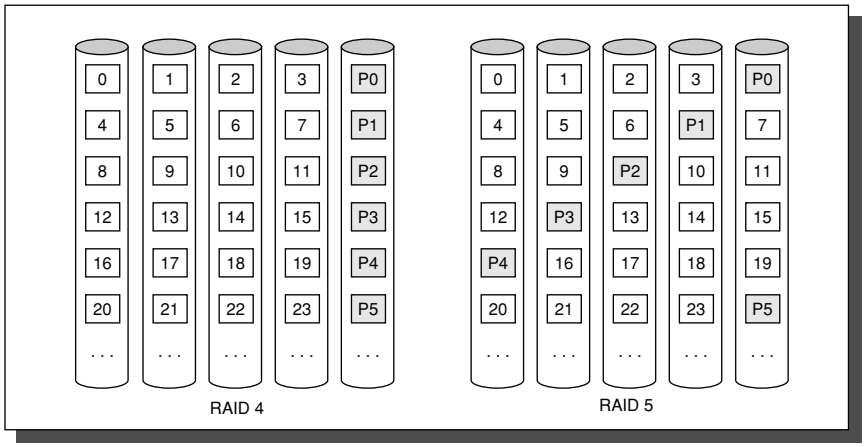


FIGURE 7.19 Block-interleaved parity (RAID 4) versus distributed block-interleaved parity (RAID 5). By distributing parity blocks to all disks, some small writes can be performed in parallel.

P+Q redundancy (RAID 6)

Parity based schemes protect against a single, self-identifying failures. When a single failure is not sufficient, parity can be generalized to have a second calculation over the data and another check disk of information. Yet another parity block is added to allow recovery from a second failure. Thus, the storage overhead is twice that of RAID 5. The small write shortcut of Figure 7.18 works as well, except now there are six disk accesses instead of four to update both P and Q information.

RAID Summary

The higher throughput, measured either as megabytes per second or as I/Os per second, as well the ability to recover from failures make RAID attractive. When combined with the advantages of smaller size and lower power of small-diameter drives, RAIDs now dominate large-scale storage systems.

Publications of real error rates are rare for two reasons. First, academics rarely have access to significant hardware resources to measure. Second, industrial re-

7.6 Errors and Failures in Real Systems

searchers are rarely allowed to publish failure information for fear that it would be used against their companies in the marketplace. Below are four exceptions.

Berkeley's Tertiary Disk

The Tertiary Disk project at the University of California created an art-image server for the Fine Arts Museums of San Francisco. This database consists of high quality images of over 70,000 art works. The database was stored on a cluster, which consisted of 20 PCs containing 368 disks connected by a switched Ethernet. It occupied in seven 7-foot high racks.

Component	Total in System	Total Failed	% Failed
SCSI Controller	44	1	2.3%
SCSI Cable	39	1	2.6%
SCSI Disk	368	7	1.9%
IDE Disk	24	6	25.0%
Disk Enclosure -Backplane	46	13	28.3%
Disk Enclosure - Power Supply	92	3	3.3%
Ethernet Controller	20	1	5.0%
Ethernet Switch	2	1	50.0%
Ethernet Cable	42	1	2.3%
CPU/Motherboard	20	0	0%

FIGURE 7.20 Failures of components in Tertiary Disk over eighteen months of operation. For each type of component, the table shows the total number in the system, the number that failed, and the percentage failure rate. Disk enclosures have two entries in the table because they had two types of problems, backplane integrity failure and power supply failure. Since each enclosure had two power supplies, a power supply failure did not affect availability. This cluster of 20 PCs, contained in seven 7-foot high, 19-inch wide rack, hosts 368 8.4 GB, 7200 RPM, 3.5-inch IBM disks. The PCs are P6-200MHz with 96 MB of DRAM each. They run FreeBSD 3.0 and the hosts are connected via switched 100 Mbit/second Ethernet. All SCSI disks are connected to two PCs via double-ended SCSI chains to support RAID-1.The primary application is called the Zoom Project, which in 1998 was the world's largest art image database, with 72,000 images. See Talagala et al [2000].

Figure 7.20 shows the failure rates of the various components of Tertiary Disk. In advance of building the system, the designers assumed that data disks would be the least reliable part of the system, as they are both mechanical and plentiful. Next would be the IDE disks, since there were fewer of them, then the power supplies, followed by integrated circuits. They assumed that passive devices like cables would scarcely ever fail.

Figure 7.20 shatters those assumptions. Since the designers followed the manufacturer's advice of making sure the disk enclosures had reduced vibration and good cooling, the data disks were very reliable. In contrast, the PC chassis containing the IDE disks did not afford the same environmental controls. (The IDE disks did not store data, but help the application and operating system to boot the PCs.) Figure 7.20 shows that the SCSI backplane, cables, and Ethernet cables were no more reliable than the data disks themselves!

As Tertiary Disk was a large system with many redundant components, it had the potential to survive this wide range of failures. Components were connected and mirrored images were placed no single failure could make any image unavailable. This strategy, which initially appeared to be overkill, proved to be vital.

This experience also demonstrated the difference between transient faults and hard faults. *Transient faults* are faults that come and go, at least temporarily fixing themselves. *Hard faults* stop the device from working properly, and will continue to misbehave until repaired. Virtually all the failures in Figure 7.20 appeared first as transient faults. It was up to the operator to decide if the behavior was so poor that they needed to be replaced or if they could continue. In fact, the word failure was not used; instead, the group borrowed terms normally used for dealing with problem employees, with the operator deciding whether a problem component should or should not be fired. Section 7.14 gives examples of transient and hard failures.

Tandem

The next example comes from industry. Gray [1990] collected data on faults for Tandem Computers, which was one of the pioneering companies in fault tolerant computing. Figure 7.21 graphs the faults that caused system failures between 1985 and 1989 in absolute faults per system and in percentage of faults encountered. The data shows a clear improvement in the reliability of hardware and maintenance. Disks in 1985 needed yearly service by Tandem, but they were replaced by disks that needed no scheduled maintenance. Shrinking number of chips and connectors per system plus software's ability to tolerate hardware faults reduced hardware's contribution to only 7% of failures by 1989. And when hardware was at fault, software embedded in the hardware device (firmware) was often the culprit. The data indicates that software in 1989 was the major source of reported outages (62%), followed by system operations (15%).

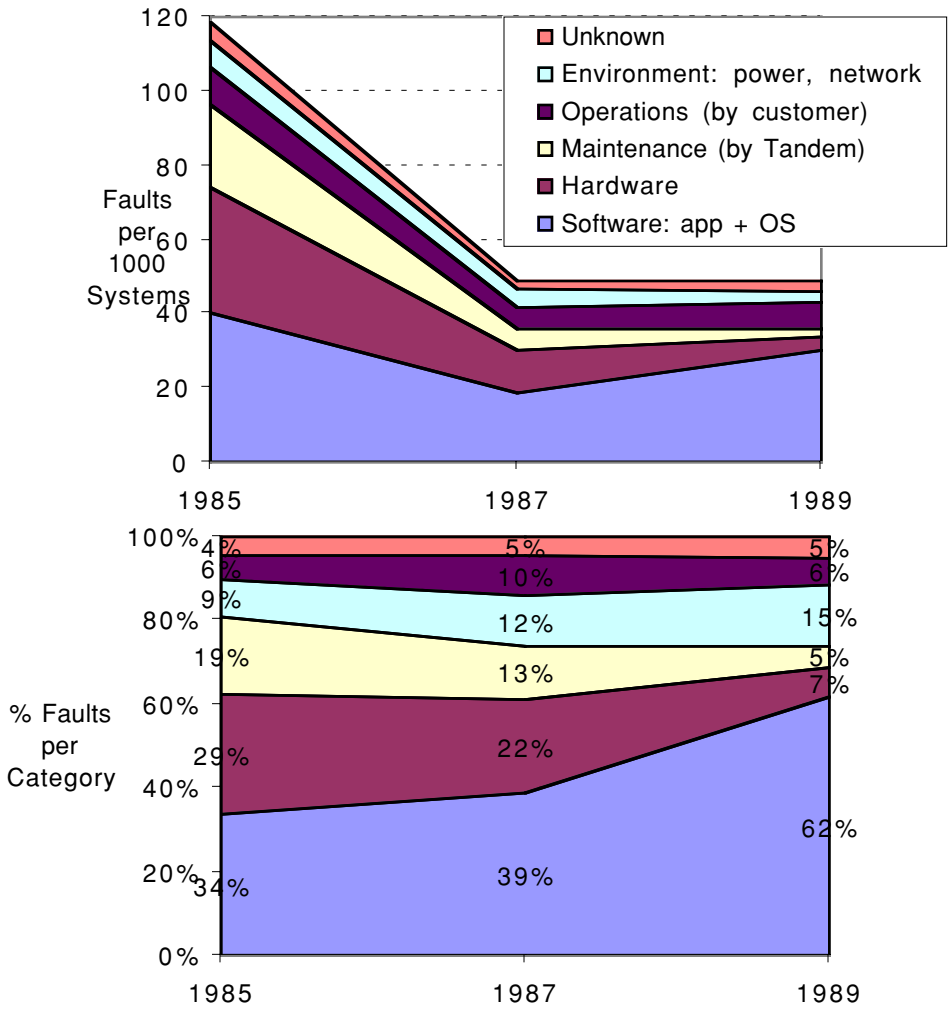


FIGURE 7.21 Faults in Tandem between 1985 and 1989. Gray [1990] collected these data for the fault tolerant Tandem computers based on reports of component failures by customers.

The problem with any such statistics are that these data only refer to what is reported; for example, environmental failures due to power outages were not reported to Tandem because they were seen as a local problem. Very difficult data to collect is operations faults, because it relies on the operators to report personal mistakes, which may affect the opinion of their managers, which in turn can affect job security and pay raises. Gray believes both environmental faults and operator faults are under-reported. His study concluded that achieving higher availability requires improvement in software quality and software fault tolerance, simpler operations, and tolerance of operational faults.

VAX

The next example is also from industry. Murphy and Gent [1995] measured faults in VAX systems. They classified faults as hardware, operating system, system management, or application/networking. Figure 7.22 shows their data for 1985 and 1993. They tried to improve the accuracy of data on operator faults by having the system automatically prompt the operator on each boot for the reason for that reboot. They also classified consecutive crashes to the same fault as operator fault. Although they believe is operator error is still under-reported, they did get more accurate information than did Gray who relied on a form that the operator filled out and then sent up the management chain. Note that the hardware/operating system went from causing 70% of the failures in 1985 to 28% in 1993. Murphy and Gent expected system management to be the primary dependability challenge in the future.

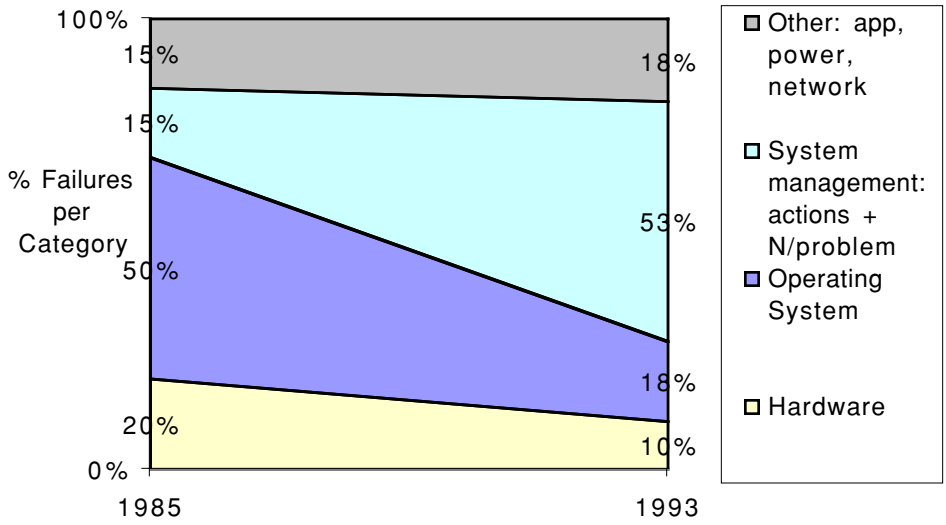


FIGURE 7.22 Causes of system failures on Digital VAX systems between 1985 and 1993 collected by Murphy and Gent [1995]. System management crashes include having several crashes for the same problem, suggesting that the problem was difficult for the operator to diagnose. It also included operator actions that directly resulted in crashes, such as giving parameters bad values, bad configurations, and bad application installation.

FCC

The final set of data comes from the government. The Federal Communications Commission (FCC) requires that all telephone companies submit explanations when they experience an outage that affects at least 30,000 people or lasts thirty minutes. These detailed disruption reports do not suffer from the self-reporting

problem of earlier figures, as investigators determine the cause of the outage rather than operators of the equipment. Kuhn [1997] studied the causes of outages between 1992 and 1994 and Enriquez [2001] did a follow-up study for the first half of 2001. In addition to reporting number of outages, the FCC data includes the number of customers affected and how long they were affected. Hence, we can look at the size and scope of failures, rather than assuming that all are equally important. Figure 7.23 plots the absolute and relative number of customer-outage minutes for those years, broken into four categories:

- Failures due to exceeding the network's capacity (overload).
- Failures due to people (human).
- Outages caused by faults in the telephone network software (software).
- Switch failure, cable failure, and power failure (hardware).

Although there was a significant improvement in failures due to overloading of the network over the years, failures due to humans increased, from about one third to two thirds of the customer-outage minutes.

These four examples and others suggest that the primary cause of failures in large systems today is faults by human operators. Hardware faults have declined due to a decreasing number of chips in systems, reduced power, and fewer connectors. Hardware dependability has improved through fault tolerance techniques such as RAID. At least some operating systems are considering reliability implications before new adding features, so in 2001 the failures largely occur elsewhere.

Although failures may be initiated due to faults by operators, it is a poor reflection on the state of the art of systems that the process of maintenance and upgrading are so error prone. Thus, the challenge for dependable systems of the future is either to tolerate faults by operators or to avoid faults by simplifying the tasks of system administration.

We have now covered the bedrock issue of dependability, giving definitions, case studies, and techniques to improve it. The next step in the storage tour is performance. We'll cover performance metrics, queuing theory, and benchmarks.

7.7 I/O Performance Measures

I/O performance has measures that have no counterparts in CPU design. One of these is diversity: Which I/O devices can connect to the computer system? Another is capacity: How many I/O devices can connect to a computer system?

In addition to these unique measures, the traditional measures of performance, namely response time and throughput, also apply to I/O. (I/O throughput is sometimes called *I/O bandwidth*, and response time is sometimes called *latency*.) The

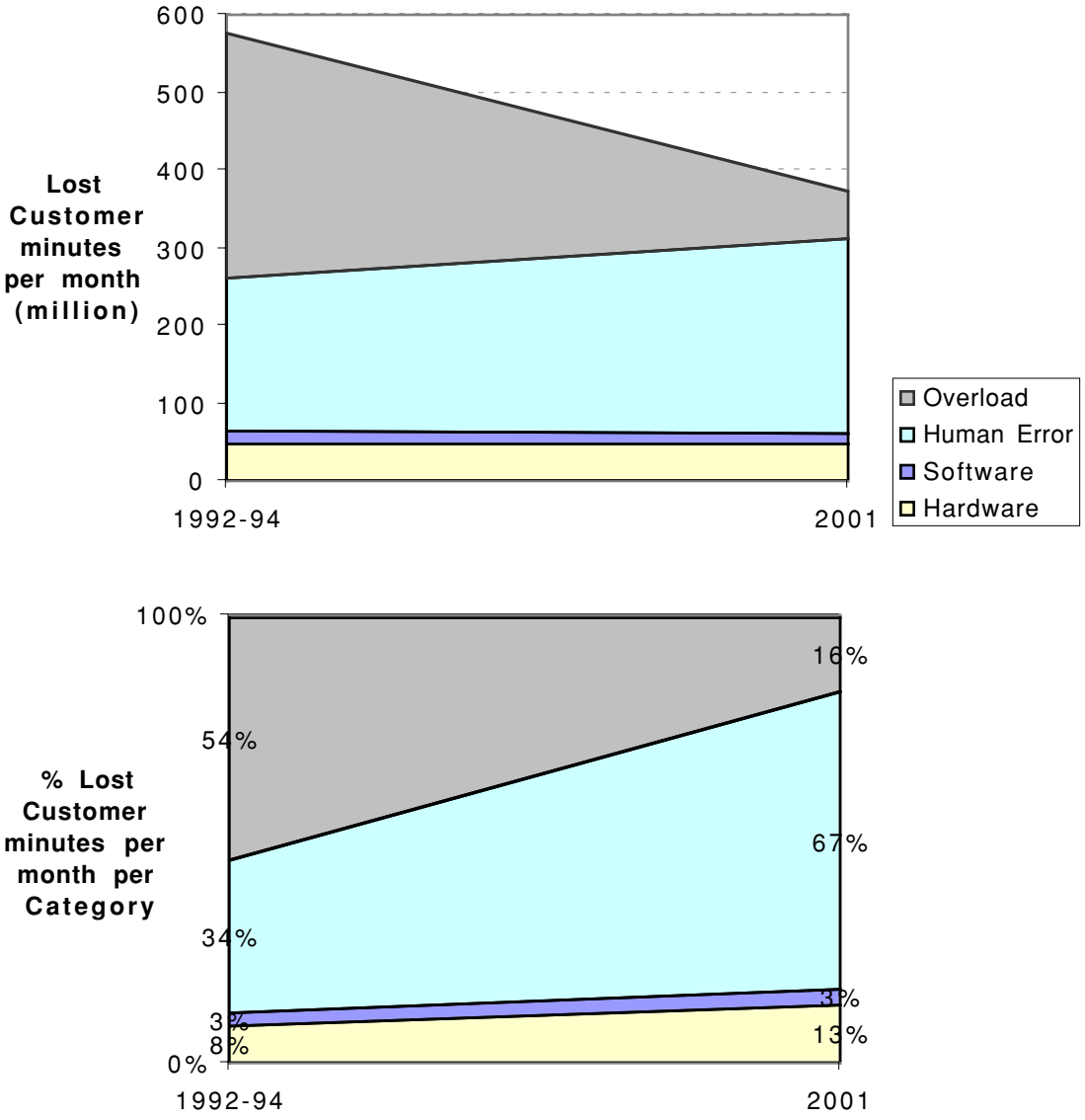


FIGURE 7.23 Failures in the Public Switched Telephone Network according to the FCC in 1992-94 and 2001. Note that in both absolute and relative terms that overload outages shrank and outages due to human error increased, with human error responsible for two-thirds of the outages for this graph in 2001. These charts leave out two categories collected by Kuhn [1997] and Enriquez [2001], vandalism and nature. Vandalism is less than 1% of customer minutes, and was not included because it was too small to plot. Nature is a very significant cause of outages in PSTN, as fires and floods can be extensive and their damage take a while to repair. Nature was not included because it has little relevance for indication of failures in computer systems. Customer-minutes multiplies the number of customers potentially affected by the length of the outage to indicate the size of the outage. Enriquez [2001] also reports blocked calls, which means calls that could not be made due to the outage. Blocked calls differentiate impact of outages during the middle of the day versus in 2001the middle of the night. Blocked-call data also suggests human error is the most important challenge for outages.

next two figures offer insight into how response time and throughput trade off against each other. Figure 7.24 shows the simple producer-server model. The producer creates tasks to be performed and places them in a buffer; the server takes tasks from the first-in-first-out buffer and performs them.

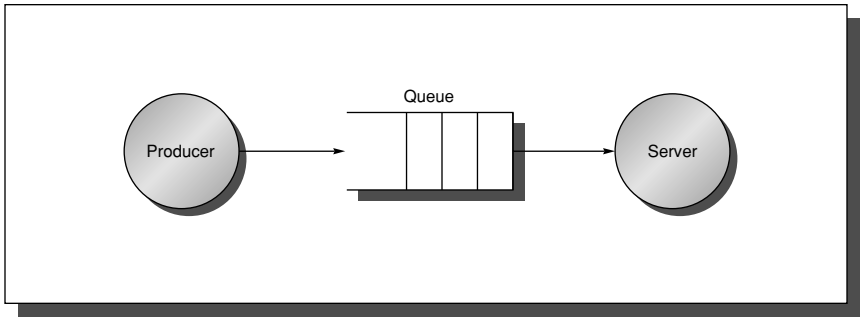


FIGURE 7.24 The traditional producer-server model of response time and throughput. Response time begins when a task is placed in the buffer and ends when it is completed by the server. Throughput is the number of tasks completed by the server in unit time.

Response time is defined as the time a task takes from the moment it is placed in the buffer until the server finishes the task. Throughput is simply the average number of tasks completed by the server over a time period. To get the highest possible throughput, the server should never be idle, and thus the buffer should never be empty. Response time, on the other hand, counts time spent in the buffer and is therefore minimized by the buffer being empty.

Another measure of I/O performance is the interference of I/O with CPU execution. Transferring data may interfere with the execution of another process. There is also overhead due to handling I/O interrupts. Our concern here is how much longer a process will take because of I/O for another process.

Throughput versus Response Time

Figure 7.25 shows throughput versus response time (or latency) for a typical I/O system. The knee of the curve is the area where a little more throughput results in much longer response time or, conversely, a little shorter response time results in much lower throughput.

How does the architect balance these conflicting demands? If the computer is interacting with human beings, Figure 7.26 suggests an answer. This figure presents the results of two studies of interactive environments: one keyboard oriented and one graphical. An interaction, or *transaction,* with a computer is divided into three parts:

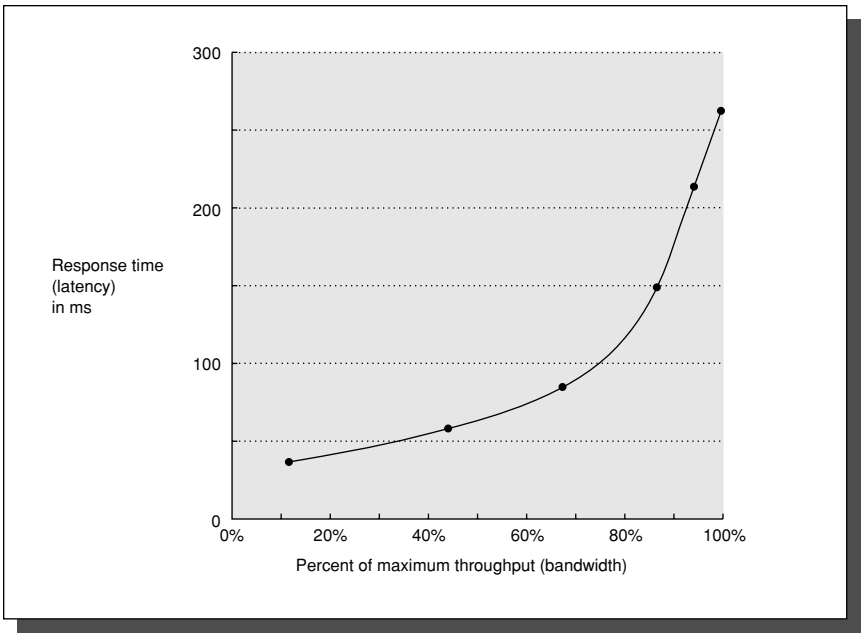


FIGURE 7.25 Throughput versus response time. Latency is normally reported as response time. Note that the minimum response time achieves only 11% of the throughput, while the response time for 100% throughput takes seven times the minimum response time. Note that the independent variable in this curve is implicit: To trace the curve, you typically vary load (concurrency). Chen et al. [1990] collected these data for an array of magnetic disks.

1. *Entry time*—The time for the user to enter the command. The graphics system in Figure 7.26 required 0.25 seconds on average to enter a command versus 4.0 seconds for the keyboard system.
2. *System response time*—The time between when the user enters the command and the complete response is displayed.
3. *Think time*—The time from the reception of the response until the user begins to enter the next command.

The sum of these three parts is called the *transaction time*. Several studies report that user productivity is inversely proportional to transaction time; *transactions per hour* are a measure of the work completed per hour by the user.

The results in Figure 7.26 show that reduction in response time actually decreases transaction time by more than just the response time reduction. Cutting system response time by 0.7 seconds saves 4.9 seconds (34%) from the conventional transaction and 2.0 seconds (70%) from the graphics transaction. This implausible result is explained by human nature: People need less time to think when given a faster response.

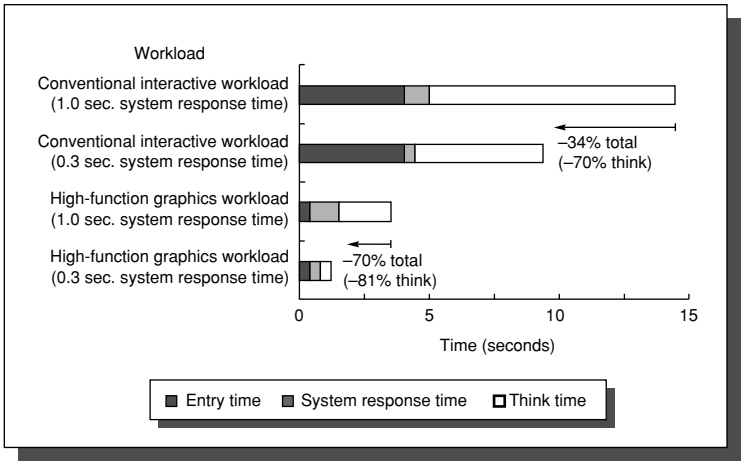


FIGURE 7.26 A user transaction with an interactive computer divided into entry time, system response time, and user think time for a conventional system and graphics system. The entry times are the same, independent of system response time. The entry time was 4 seconds for the conventional system and 0.25 seconds for the graphics system. (From Brady [1986].)

Whether these results are explained as a better match to the human attention span or getting people "on a roll," several studies report this behavior. In fact, as computer response times drop below one second, productivity seems to make a more than linear jump. Figure 7.27 compares transactions per hour (the inverse of transaction time) of a novice, an average engineer, and an expert performing physical design tasks on graphics displays. System response time magnified talent: a novice with subsecond system response time was as productive as an experienced professional with slower response, and the experienced engineer in turn could outperform the expert with a similar advantage in response time. In all cases the number of transactions per hour jumps more than linearly with subsecond response time.

Since humans may be able to get much more work done per day with better response time, it is possible to attach an economic benefit to lowering response time into the subsecond range [IBM 1982]. This assessment helps the architect decide how to tip the balance between response time and throughput.

Although these studies were on older machines, people's patience has not changed. Its is still a problem today as response times are often still much longer than a second, even if hardware is 1000 times faster. Examples of long delays starting an application on a desktop PC due to include many disk I/Os or network delays when clicking on WWW links.

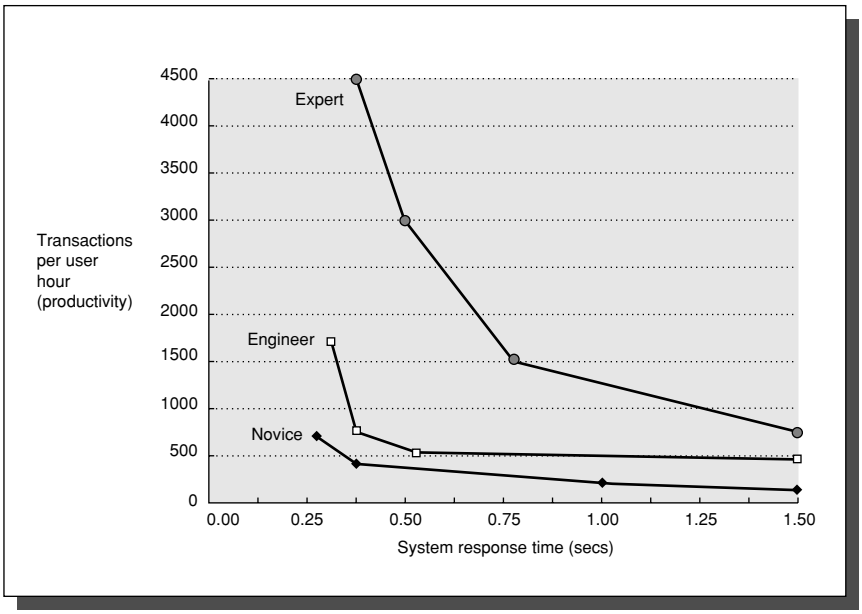


FIGURE 7.27 Transactions per hour versus computer response time for a novice, experienced engineer, and expert doing physical design on a graphics system. *Transactions per hour* are a measure of productivity. (From IBM [1982].)

Response Time vs. Throughput in Benchmarks

I/O benchmarks offer another perspective on the response time vs. throughput trade-off. Figure 7.28 shows the response time restrictions for three I/O benchmarks. The two reporting approaches report maximum throughput given either that 90% of response times must be less than a limit or that the average response time must be less than a limit.

I/O Benchmark	Response Time Restriction	Throughput Metric
TPC-C: Complex Query OLTP	≥ 90% of transaction must meet response time limit; 5 seconds for most types of transactions	new order transactions per minute
TPC-W: Transactional web benchmark	≥ 90%of web interactions must meet response time limit; 3 seconds for most types of web interactions	web interactions per second
SPECsfs97	Average response time ≤ 40 milliseconds	NFS operations per second

FIGURE 7.28 Response time restrictions for three I/O Benchmarks.

7.8 A Little Queuing Theory

In processor design we have simple back-of-the-envelope calculations of performance associated with the CPI formula in Chapter 1. The next step in accuracy is full-scale simulation of the system, which is considerably more work. In I/O systems we also have a best case analysis as a back-of-the-envelope calculation, and again full scale simulation is also much more accurate and much more work to calculate expected performance.

With I/O systems, however, we also have a mathematical tool to guide I/O design that is a little more work and much more accurate than best case analysis, but much less work than full scale simulation. Because of the probabilistic nature of I/O events and because of sharing of I/O resources, we can give a set of simple theorems that will help calculate response time and throughput of an entire I/O system. This helpful field is called *queuing theory*. Since there are many books are courses on the subject, this section serves only as a first introduction to the topic; interested readers should see section 7.16 to learn more.

Let's start with a black box approach to I/O systems, as in Figure 7.29. In our example, the CPU is making I/O requests that arrive at the I/O device, and the requests "depart" when the I/O device fulfills them.

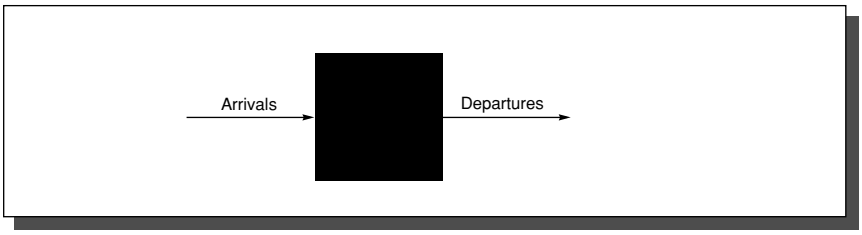


FIGURE 7.29 Treating the I/O system as a black box. This leads to a simple but important observation: If the system is in steady state, then the number of tasks entering the systems must equal the number of tasks leaving the system. This flow-balanced state is necessary but not sufficient for steady state. If the system has been observed or measured for a sufficiently long time and mean waiting times stabilize, then we say that the system has reached steady state.

We are usually interested in the long term, or steady state, of a system rather than in the initial start-up conditions. Suppose we weren't. Although there is a mathematics that helps (Markov chains), except for a few cases, the only way to solve the resulting equations. Since the purpose of this section is to show something a little harder than back-of-the-envelope calculations but less than simulation, we won't cover such analyses here. (Interested readers should follow the references at the end of this chapter.)

Hence, in this section we make the simplifying assumption that we are evaluating systems with multiple independent requests for I/O service that are in equilibrium: the input rate must be equal to the output rate. We also assume there is a steady supply of tasks, for in many real systems the task consumption rate is determined by system characteristics such as capacity. TPC-C is one example.

This leads us to *Little's Law*, which relates the average number of tasks in the system, the average arrival rate of new tasks, and the average time to perform a task:

$$\text{Mean number of tasks in system} = \text{Arrival rate} \times \text{Mean response time}$$

Little's Law applies to any system in equilibrium, as long as nothing inside the black box is creating new tasks or destroying them. Note that the arrival rate and the response time must use the same time unit; inconsistency in time units is a common cause of errors.

Let's try to derive Little's Law. Assume we observe a system for $\text{Time}_{\text{observe}}$ minutes. During that observation, we record how long it took each task to be serviced, and then sum those times. The number of tasks completed during $\text{Time}_{\text{observe}}$ is $\text{Number}_{\text{task}}$, and the sum of waiting times is $\text{Time}_{\text{accumulated}}$. Then

$$\text{Mean number of tasks in system} = \frac{\text{Time}_{\text{accumulated}}}{\text{Time}_{\text{observe}}}$$

$$\text{Mean response time} = \frac{\text{Time}_{\text{accumulated}}}{\text{Number}_{\text{tasks}}}$$

Algebra lets us split the first formula:

$$\frac{\text{Time}_{\text{accumulated}}}{\text{Time}_{\text{observe}}} = \frac{\text{Time}_{\text{accumulated}}}{\text{Number}_{\text{tasks}}} \times \frac{\text{Number}_{\text{tasks}}}{\text{Time}_{\text{observe}}}$$

Since the following definitions hold

$$\text{Mean number of tasks in system} = \frac{\text{Time}_{\text{accumulated}}}{\text{Time}_{\text{observe}}}$$

$$\text{Mean response time} = \frac{\text{Time}_{\text{accumulated}}}{\text{Number}_{\text{tasks}}}$$

$$\text{Arrival rate} = \frac{\text{Number}_{\text{tasks}}}{\text{Time}_{\text{observe}}}$$

if we substitute these three definitions in the formula above, and swap the resulting two terms on the right hand side, we get Little's Law.

$$\text{Mean number of tasks in system} = \text{Arrival rate} \times \text{Mean response time}$$

This simple equation is surprisingly powerful, as we shall see.

If we open the black box, we see Figure 7.30. The areas where the tasks accumulate, waiting to be serviced, is called the *queue,* or *waiting line*, and the device performing the requested service is called the *server*. Until we get to the last two pages of this section, we assume a single server.

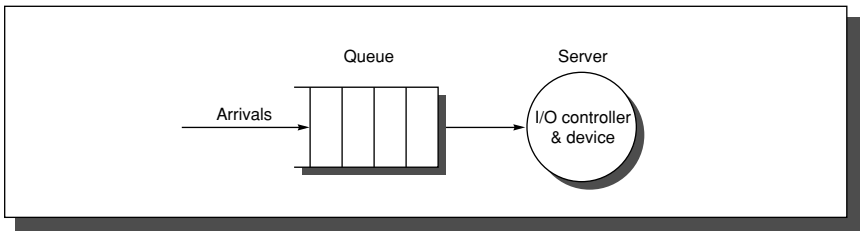


FIGURE 7.30 The single server model for this section. In this situation, an I/O request "departs" by being completed by the server.

Little's Law and a series of definitions lead to several useful equations:

$\text{Time}_{\text{server}}$—Average time to service a task; average service rate is $1/\text{Time}_{\text{server}}$, traditionally represented by the symbol μ in many queueing texts.

$\text{Time}_{\text{queue}}$—Average time per task in the queue.

$\text{Time}_{\text{system}}$—Average time/task in the system, or the response time, the sum of $\text{Time}_{\text{queue}}$ and $\text{Time}_{\text{server}}$.

Arrival rate—Average number of arriving tasks/second, traditionally represented by the symbol λ in many queueing texts.

$\text{Length}_{\text{server}}$—Average number of tasks in service.

$\text{Length}_{\text{queue}}$—Average length of queue.

$\text{Length}_{\text{system}}$—Average number of tasks in system, the sum of $\text{Length}_{\text{queue}}$ and $\text{Length}_{\text{server}}$.

One common misunderstanding can be made clearer by these definitions: whether the question is how long a task must wait in the queue before service starts ($\text{Time}_{\text{queue}}$) or how long a task takes until it is completed ($\text{Time}_{\text{system}}$). The latter term is what we mean by response time, and the relationship between the terms is $\text{Time}_{\text{system}} = \text{Time}_{\text{queue}} + \text{Time}_{\text{server}}$.

The mean number of tasks in service ($\text{Length}_{\text{server}}$) is simply $\text{Arrival rate} \times \text{Time}_{\text{server}}$, which is Little's Law. Server utilization is simply the mean number of tasks being serviced divided by by the service rate. For a single server, the service rate is $1/\text{Time}_{\text{server}}$. Server utilization (and, in this case, the mean number of tasks per server) is simply

$$\text{Server utilization} = \text{Arrival rate} \times \text{Time}_{\text{server}}$$

The value must be between 0 and 1, for otherwise there would be more tasks arriving than could be serviced, violating our assumption that the system is in equilibrium. Note that this formula is just a restatement of Little's Law. Utilization is also called *traffic intensity* and is represented by the symbol ρ in many texts.

EXAMPLE Suppose an I/O system with a single disk gets on average 50 I/O requests per second. Assume the average time for a disk to service an I/O request is10 ms. What is the utilization of the I/O system?

ANSWER Using the equation above, with 10 ms represented as 0.01 seconds:

$$\text{Server utilization} = \text{Arrival rate} \times \text{Time}_{\text{server}} = \frac{50}{\cancel{\text{sec}}} \times 0.01 \cancel{\text{sec}} = 0.50$$

Therefore, the I/O system utilization is 0.5. n

How the queue delivers tasks to the server is called the *queue discipline*. The simplest and most common discipline is *first-in-first-out* (FIFO). If we assume FIFO, we can relate time waiting in the queue to the mean number of tasks in the queue:

$$\text{Time}_{\text{queue}} = \text{Length}_{\text{queue}} \times \text{Time}_{\text{server}} + \text{Mean time to complete service of task when new task arrives if server is busy}$$

That is, the time in the queue is the number of tasks in the queue times the mean service time plus the time it takes the server to complete whatever task is being serviced when a new task arrives. (There is one more restriction about the arrival of tasks, which we reveal on page 534.)

The last component of the equation is not as simple as it first appears. A new task can arrive at any instant, so we have no basis to know how long the existing task has been in the server. Although such requests are random events, if we know something about the distribution of events we can predict performance.

To estimate the last component of the formula we need to know a little about distributions of *random variables*. A variable is random if it takes one of a specified set of values with a specified probability; that is, you cannot know exactly what its next value will be, but you may know the probability of all possible values.

Requests for service from an I/O system can be modeled by a random variable because the operating system is normally switching between several processes that generate independent I/O requests. We also model I/O service times by a random variable given the probabilistic nature of disks in terms of seek and rotational delays.

One way to characterize the distribution of values of a random variable with discrete values is a *histogram*, which divides the range between the minimum and maximum values into subranges called *buckets*. Histograms then plot the number in each bucket as columns. Histograms work well for distributions that are discrete values—for example, the number of I/O requests. For distributions that are not discrete values, such as time waiting for an I/O request, we have two choices. Either we need a curve to plot the values over the full range, so that we can accurately estimate the value, or we need a very fine time unit so that we get a very large number of buckets to accurately estimate time. For example, a histogram can be built of disk service times measured in intervals of ten microseconds although disk service times are truly continuous.

Hence, to be able to solve the last part of the equation above we need to characterize the distribution of this random variable. The mean time and some measure of the variance are sufficient for that characterization.

For the first term, we use the *arithmetic mean time* (see page 26 in Chapter 1 for a slightly different version of the formula). Let's first assume after measuring the number of occurrences, say n_i, of tasks one could compute frequency of occurrence of task *i*:

$$f_i = n_i \Big/ \left(\sum_{i=1}^{n} n_i \right)$$

Then arithmetic mean is:

$$\text{Arithmetic mean time} = f_1 \times T_1 + f_2 \times T_2 + \ldots + f_n \times T_n$$

where T_i is the time for task *i* and f_i is the frequency of occurrence of task *i*.

To characterize variability about the mean, many people use the standard deviation. Let's use the *variance* instead, which is simply the square of the standard deviation, as it will help us with characterizing the probability distribution. Given the arithmetic mean, the variance can be calculated as

$$\text{Variance} = (f_1 \times T_1^2 + f_2 \times T_2^2 + \ldots + f_n \times T_n^2) - \text{Arithmetic mean time}^2$$

It is important to remember the units when computing variance. Let's assume the distribution is of time. If time is on the order of 100 milliseconds, then squaring it yields 10,000 square milliseconds. This unit is certainly unusual. It would be more convenient if we had a unitless measure.

To avoid this unit problem, we use the *squared coefficient of variance*, traditionally called C^2:

$$C^2 = \frac{\text{Variance}}{\text{Arithmetic mean time}^2}$$

We can solve for C, the coefficient of variance, as

$$C = \frac{\sqrt{\text{Variance}}}{\text{Arithmetic mean time}} = \frac{\text{Standard deviation}}{\text{Arithmetic mean time}}$$

We are trying to characterize random events, but to be able to predict performance we need a distribution of random events where the mathematics is tractable. The most popular such distribution is the *exponential distribution*, which has a C value of 1.

Note that we are using a constant to characterize variability about the mean. The invariance of C over time reflects the property that the history of events has no impact on the probability of an event occurring now. This forgetful property is called *memoryless*, and this property is an important assumption used to predict behavior using these models. (Suppose this memoryless property did not exist; then we would have to worry about the exact arrival times of requests relative to each other, which would make the mathematics considerably less tractable!)

One of the most widely used exponential distributions is called a *Poisson distribution*, named after the mathematician Simeon Poisson. It is used to characterize random events in a given time interval, and has several desirable mathematical properties. The Poisson distribution is described by the following equation (called the probability mass function):

$$\text{probability}(k) = \frac{e^{-a} \times a^{k}}{k!}$$

where $a = \text{rate of events} \times \text{elapsed time}$. If interarrival times are exponentially distributed and we use Arrival rate from above for rate of events, the number of arrivals in a time interval t is a *Poisson process*, which has the Poisson distribution with $a = \text{Arrical rate} \times t$. As mentioned on page 533, the equation for $\text{Time}_{\text{server}}$ had another restriction on task arrival: it holds only for Poisson processes.

Finally, we can answer the question about the length of time a new task must wait for the server to complete a task, called the *average residual service time*, which again assumes Poisson arrivals:

$$\text{Average residual service time} = 1/2 \times \text{Weighted mean time} \times (1 + C^2)$$

Although we won't derive this formula, we can appeal to intuition. When the distribution is not random and all possible values are equal to the average, the standard deviation is 0 and so C is 0. The average residual service time is then just half the average service time, as we would expect. If the distribution is random and it is Poisson, then C is 1 and the average residual service time equals the weighted mean time.

EXAMPLE Using the definitions and formulas above, derive the average time waiting in the queue ($Time_{queue}$) in terms of the average service time ($Time_{server}$) and server utilization.

ANSWER All tasks in the queue ($Length_{queue}$) ahead of the new task must be completed before the task can be serviced; each takes on average $Time_{server}$. If a task is at the server, it takes average residual service time to complete. The chance the server is busy is *server utilization*, hence the expected time for service is Server utilization × Average residual service time. This leads to our initial formula:

$$Time_{queue} = Length_{queue} \times Time_{server} + Server\ utilization \times Average\ residual\ service\ time$$

Replacing average residual service time by its definition and $Length_{queue}$ by Arrival rate × $Time_{queue}$ yields

$$Time_{queue} = Server\ utilization \times (1/2 \times Time_{server} \times (1 + C^2)) + (Arrival\ rate \times Time_{queue}) \times Time_{server}$$

Since this section is concerned with exponential distributions, C^2 is 1. Thus

$$Time_{queue} = Server\ utilization \times Time_{server} + (Arrival\ rate \times Time_{queue}) \times Time_{server}$$

Rearranging the last term, let us replace Arrival rate × $Time_{server}$ by Server utilization:

$$\begin{aligned} Time_{queue} &= Server\ utilization \times Time_{server} + (Arrival\ rate \times Time_{server}) \times Time_{queue} \\ &= Server\ utilization \times Time_{server} + Server\ utilization \times Time_{queue} \end{aligned}$$

Rearranging terms and simplifying gives us the desired equation:

$$\begin{aligned} Time_{queue} &= Server\ utilization \times Time_{server} + Server\ utilization \times Time_{queue} \\ Time_{queue} - Server\ utilization \times Time_{queue} &= Server\ utilization \times Time_{server} \\ Time_{queue} \times (1 - Server\ utilization) &= Server\ utilization \times Time_{server} \\ Time_{queue} &= Time_{server} \times \frac{Server\ utilization}{(1 - Server\ utilization)} \end{aligned}$$

n

Little's Law can be applied to the components of the black box as well, since they must also be in equilibrium:

$$Length_{queue} = Arrival\ rate \times Time_{queue}$$

If we substitute for $\text{Time}_{\text{queue}}$ from above, we get

$$\text{Length}_{\text{queue}} = \text{Arrival rate} \times \text{Time}_{\text{server}} \times \frac{\text{Server utilization}}{\text{(1 – Server utilization)}}$$

Since $\text{Arrival rate} \times \text{Time}_{\text{server}} = \text{Server utilization}$, we can simplify further:

$$\text{Length}_{\text{queue}} = \text{Server utilization} \times \frac{\text{Server utilization}}{\text{(1 – Server utilization)}} = \frac{\text{Server utilization}^2}{\text{(1 – Server utilization)}}$$

EXAMPLE For the system in the example on page 533, which has a Server utilization of 0.5, what is the mean number of I/O requests in the queue?

ANSWER Using the equation above,

$$\text{Length}_{\text{queue}} = \frac{\text{Server utilization}^2}{\text{(1 – Server utilization)}} = \frac{0.5^2}{\text{(1 – 0.5)}} = \frac{0.25}{0.50} = 0.5$$

So there are 0.5 requests on average in the queue. n

As mentioned above, these equations and this section are based on an area of applied mathematics called queuing theory, which offers equations to predict behavior of such random variables. Real systems are too complex for queuing theory to provide exact analysis, and hence queuing theory works best when only approximate answers are needed.

Queuing theory makes a sharp distinction between past events, which can be characterized by measurements using simple arithmetic, and future events, which are predictions requiring more sophisticated mathematics. In computer systems, we commonly predict the future from the past; one example is least recently used block replacement (see Chapter 5). Hence, the distinction between measurements and predicted distributions is often blurred; we use measurements to verify the type of distribution and then rely on the distribution thereafter.

Let's review the assumptions about the queuing model:

- The system is in equilibrium.
- The times between two successive requests arriving, called the *interarrival times*, are exponentially distributed, which characterizes the arrival rate mentioned above.
- The number of sources of requests is unlimited (this is called an *infinite population model* in queuing theory; finite population models are used when systems are not in equilibrium).
- The server can start on the next job immediately after finishing with the prior one.
- There is no limit to the length of the queue, and it follows the first-in-first-out order discipline, so all tasks in line must be completed.

- There is one server

Such a queue is called *M*/*M*/*1*:

M = exponentially random request arrival ($C^2 = 1$), with M standing for A. A. Markov, the mathematician who defined and analyzed the memoryless processes mentioned above

M = exponentially random request arrival ($C^2 = 1$), with M again for Markov

1 = single server

The M/M/1 model is a simple and widely used model.

The assumption of exponential distribution is commonly used in queuing examples for three reasons, one good, one fair, and one bad. The good reason is that a superposition of many arbitrary distributions acts as an exponential distribution. Many times in computer systems, a particular behavior is the result of many components interacting, so an exponential distribution of interarrival times is the right model. The fair reason is that when variability is unclear, an exponential distribution with intermediate variability (C = 1) is a safer guess than low variability ($C \approx 0$) or high variability (large C). The bad reason is that the math is simpler if you assume exponential distributions.

Let's put queuing theory to work in a few examples.

EXAMPLE Suppose a processor sends 10 disk I/Os per second, these requests are exponentially distributed, and the average service time of an older disk is 20 ms. Answer the following questions:

1. On average, how utilized is the disk?
2. What is the average time spent in the queue?
3. What is the average response time for a disk request, including the queuing time and disk service time?

ANSWER Let's restate these facts:

Average number of arriving tasks/second is 40.

Average disk time to service a task is 20 ms (0.02 sec).

The server utilization is then

$$\text{Server utilization} = \text{Arrival rate} \times \text{Time}_{\text{server}} = 40 \times 0.02 = 0.8$$

Since the service times are exponentially distributed, we can use the simplified formula for the average time spent waiting in line:

$$\text{Time}_{\text{queue}} = \text{Time}_{\text{server}} \times \frac{\text{Server utilization}}{(1 - \text{Server utilization})} = 20\ \text{ms} \times \frac{0.8}{1 - 0.8} = 20 \times \frac{0.8}{0.2} = 20 \times 4 = 80\ \text{ms}$$

The average response time is

$$\text{Time}_{\text{queue}} + \text{Time}_{\text{server}} = 80 + 20 \text{ ms} = 100 \text{ ms}$$

Thus, on average we spend 80% of our time waiting in the queue!

n

EXAMPLE Suppose we get a new, faster disk. Recalculate the answers to the questions above, assuming the disk service time is 10 ms.

ANSWER The disk utilization is then

$$\text{Server utilization} = \text{Arrival rate} \times \text{Time}_{\text{server}} = 40 \times 0.01 = 0.4$$

The formula for the average time spent waiting in line:

$$\text{Time}_{\text{queue}} = \text{Time}_{\text{server}} \times \frac{\text{Server utilization}}{(1 - \text{Server utilization})} = 10 \text{ ms} \times \frac{0.4}{1 - 0.4} = 10 \times \frac{0.4}{0.6} = 10 \times \frac{2}{3} = 6.7 \text{ ms}$$

The average response time is 10 + 6.7 ms or 16.7 ms, 6.0 times faster than the old response time even though the new service time is only 2.0 times faster.

n

Thus far, we have been assuming a single server, such as a single disk. Many real systems have multiple disks and hence could use multiple servers. Such a system is called a *M*/*M*/*m* model in queueing theory.

Let's give the same formulas for the M/M/m queue, using $\text{N}_{\text{servers}}$ to represent the number of servers. The first two formulas are easy:

$$\text{Utilization} = \frac{\text{Arrival rate} \times \text{Time}_{\text{server}}}{\text{N}_{\text{servers}}}$$

$$\text{Length}_{\text{queue}} = \text{Arrival rate} \times \text{Time}_{\text{queue}}$$

The time waiting in the queue is

$$\text{Time}_{\text{queue}} = \text{Time}_{\text{server}} \times \frac{\text{P}_{\text{tasks} \geq \text{N}_{\text{servers}}}}{\text{N}_{\text{servers}} \times (1 - \text{Utilization})} \quad ,$$

This formula is related to the one for M/M/1, except we replace utilization of a single server with the probability of that a task will be queued as opposed to being immediately serviced, and divide the time in queue by the number of servers. Alas, calculating the probability of jobs being in the queue when there are $\text{N}_{\text{servers}}$ servers is much more complicated. First, the probability that there are no tasks in the system is:

$$\text{Prob}_{0\ \text{tasks}} = \left[1 + \frac{(\text{N}_{\text{servers}} \times \text{Utilization})^{\text{N}_{\text{servers}}}}{\text{N}_{\text{servers}}! \times (1 - \text{Utilization})} + \sum_{n=1}^{\text{N}_{\text{servers}}-1} \frac{(\text{N}_{\text{servers}} \times \text{Utilization})^{n}}{n!}\right]^{-1}$$

Then the probability there are as many or more tasks than we have servers is:

$$\text{Prob}_{\text{tasks} \geq \text{N}_{\text{servers}}} = \frac{\text{N}_{\text{servers}} \times \text{Utilization}^{\text{N}_{\text{servers}}}}{\text{N}_{\text{servers}}! \times (1 - \text{Utilization})} \times \text{Prob}_{0\ \text{tasks}}$$

Note that if $\text{N}_{\text{servers}}$ is 1, $\text{Prob}_{\text{task} \geq \text{Nservers}}$ simplifies back to Utilization, and we get the same formula as for M/M/1.

Let's try an example.

EXAMPLE Suppose instead of a new, faster disk, we add a second slow disk, and duplicate the data so that reads can be serviced by either disk. Let's assume that the requests are all reads. Recalculate the answers to the questions above, this time using a M/M/m queue.

ANSWER The average utilization of the two disks is then

$$\text{Server utilization} = \frac{\text{Arrival rate} \times \text{Time}_{\text{server}}}{\text{Number}_{\text{servers}}} = \frac{40 \times 0.02}{2} = 0.4$$

We first calculate the probability of no tasks in the queue:

$$\begin{aligned}\text{Prob}_{0\ \text{tasks}} &= \left[1 + \frac{(2 \times \text{Utilization})^{2}}{2! \times (1 - \text{Utilization})} + \sum_{n=1}^{1} \frac{(2 \times \text{Utilization})^{n}}{n!}\right]^{-1} \\ &= \left[1 + \frac{(2 \times 0.4)^{2}}{2 \times (1 - 0.4)} + (2 \times 0.4)\right]^{-1} = \left[1 + \frac{0.640}{1.2} + 0.800\right]^{-1} \\ &= [1 + 0.533 + 0.800]^{-1} = 2.333^{-1}\end{aligned}$$

We use this result to calculate the probability of tasks in the queue:

$$\begin{aligned}\text{Prob}_{\text{tasks} \geq \text{N}_{\text{servers}}} &= \frac{2 \times \text{Utilization}^{2}}{2! \times (1 - \text{Utilization})} \times \text{Prob}_{0\ \text{tasks}} \\ &= \frac{(2 \times 0.4)^{2}}{2 \times (1 - 0.4)} \times 2.333^{-1} = \frac{0.640}{1.2} \times 2.333^{-1} \\ &= 0.533 / 2.333 = 0.229\end{aligned}$$

Finally, the time waiting in the queue:

$$\begin{aligned}\text{Time}_{\text{queue}} &= \text{Time}_{\text{server}} \times \frac{P_{\text{tasks}} \geq N_{\text{servers}}}{N_{\text{servers}} \times (1 - \text{Utilization})} \\ &= 0.020 \times \frac{0.229}{2 \times (1 - 0.4)} = 0.020 \times \frac{0.229}{1.2} \\ &= 0.020 \times 0.190 = 0.0038\end{aligned}$$

The average response time is 20 + 3.8 ms or 23.8 ms. For this workload, two disks cut the queue waiting time by a factor of 21 over a single slow disk and a factor of 1.75 versus a single fast disk. The mean service time of a system with a single fast disk, however, is still 1.4 times faster than one with two disks since the disk service time is 2.0 times faster. n

Section 7.11 and the exercises have other examples using queuing theory to predict performance.

7.9 Benchmarks of Storage Performance and Availability

The prior subsection tries to predict the performance of storage subsystems. We also need to measure the performance of real systems to collect the values of parameters needed for prediction, to determine if the queuing theory assumptions hold, and to suggest what to do if the assumptions don't hold. Benchmarks help.

Transaction Processing Benchmarks

Transaction processing (TP, or OLTP for on-line transaction processing) is chiefly concerned with *I/O rate*: the number of disk accesses per second, as opposed to *data rate*, measured as bytes of data per second. TP generally involves changes to a large body of shared information from many terminals, with the TP system guaranteeing proper behavior on a failure. Suppose, for example, a bank's computer fails when a customer tries to withdraw money. The TP system would guarantee that the account is debited if the customer received the money *and* that the account is unchanged if the money was not received. Airline reservations systems as well as banks are traditional customers for TP.

As mentioned in Chapter 1, two dozen members of the TP community conspired to form a benchmark for the industry and, to avoid the wrath of their legal departments, published the report anonymously [1985]. This report led to the *Transaction Processing Council*, which in turn has led to seven benchmarks since its founding.

Benchmark	Data Size (GB)	Performance Metric	Date of First Results
A: Debit Credit (retired)	0.1 to 10	transactions per second	July, 1990
B: Batch Debit Credit (retired)	0.1 to 10	transactions per second	July, 1991
C: Complex Query OLTP	100 to 3000 (minimum 0.07 * tpm)	new order transactions per minute	September, 1992
D: Decision Support (retired)	100, 300, 1000	queries per hour	December, 1995
H: Ad hoc decision support	100, 300, 1000	queries per hour	October, 1999
R: Business reporting decision support	1000	queries per hour	August, 1999
W: Transactional web benchmark	≈ 50, 500	web interactions per second	July, 2000

FIGURE 7.31 Transaction Processing Council Benchmarks. The summary results include both the performance metric and the price-performance of that metric. TPC-A, TPC-B, and TPC-D were retired.

Figure 7.31 summarizes these benchmarks. Let's describe TPC-C to give the flavor of these benchmarks. TPC-C uses a database to simulate an order-entry environment of a wholesale supplier, including entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouses. It runs five concurrent transactions of varying complexity, and the database includes nine tables with a scalable range of records and customers. TPC-C is measured in transactions per minute (tpmC) and in price of system, including hardware, software, and three years of maintenance support.

These TPC benchmarks were either the first, and in some cases still the only ones, that have these unusual characteristics:

- *Price is included with the benchmark results*. The cost of hardware, software, and five-year maintenance agreements is included in a submission, which enables evaluations based on price-performance as well as high performance.
- *The data set generally must scale in size as the throughput increases*. The benchmarks are trying to model real systems, in which the demand on the system and the size of the data stored in it increase together. It makes no sense, for example, to have thousands of people per minute access hundreds of bank accounts.
- *The benchmark results are audited*. Before results can be submitted, they must be approved by a certified TPC auditor, who enforces the TPC rules that try to make sure that only fair results are submitted. Results can be challenged and disputes resolved by going before the TPC council.
- *Throughput is the performance metric but response times are limited*. For example, with TPC-C, 90% of the New-Order transaction response times must be less than 5 seconds.

- *An independent organization maintains the benchmarks*. Dues collected by TPC pay for an administrative structure including a Chief Operating Office. This organization settles disputes, conducts mail ballots on approval of changes to benchmarks, hold board meetings, and so on.

SPEC System-Level File Server (SFS) and Web Benchmarks

The SPEC benchmarking effort is best known for its characterization of processor performance, but has created benchmarks for other fields as well. In 1990 seven companies agreed on a synthetic benchmark, called SFS, to evaluate systems running the Sun Microsystems network file service NFS. This benchmark was upgraded to SFS 2.0 (also called SPEC SFS97) to include support for NSF version 3, using TCP in addition to UDP as the transport protocol, and making the mix of operations more realistic. Measurements on NFS systems to propose a reasonable synthetic mix of reads, writes, and file operations such as examining a file. SFS supplies default parameters for comparative performance. For example, half of all writes are done in 8-KB blocks and half are done in partial blocks of 1, 2, or 4 KB. For reads, the mix is 85% full blocks and 15% partial blocks.

Like TPC-C, SFS scales the amount of data stored according to the reported throughput: For every 100 NFS operations per second, the capacity must increase by 1 GB. It also limits the average response time, in this case to 40 ms. Figure 7.32 shows average response time versus throughput for four systems. Unfortunately, unlike the TPC benchmarks, SFS does not normalize for different price configurations. The fastest system in Figure 7.32 has 7 times the number of CPUs and disks as the slowest system, but SPEC leaves it to you to calculate price versus performance. As performance scaled to new heights, SPEC discovered bugs in the benchmark that impact the amount of work done during the measurement periods. Hence, it was retired in June 2001.

SPEC WEB is a benchmark for evaluating the performance of World Wide Web servers. The SPEC WEB99 workload simulates accesses to a web service provider, where the server supports home pages for several organizations. Each home page is a collection of files ranging in size from small icons to large documents and images, with some files being more popular than others. The workload defines four sizes of files and their frequency of activity:

- less than 1 KB, representing an small icon: 35% of activity
- 1 to 10 KB: 50% of activity
- 10 to 100 KB: 14% of activity
- 100 KB to 1 MB: representing a large document and image,1% of activity

For each home page, there are nine files in each of the four classes.

The workload simulates dynamic operations such as rotating advertisements on a web page, customized web page creation, and user registration. The workload is gradually increased until the server software is saturated with hits and the response time degrades significantly.

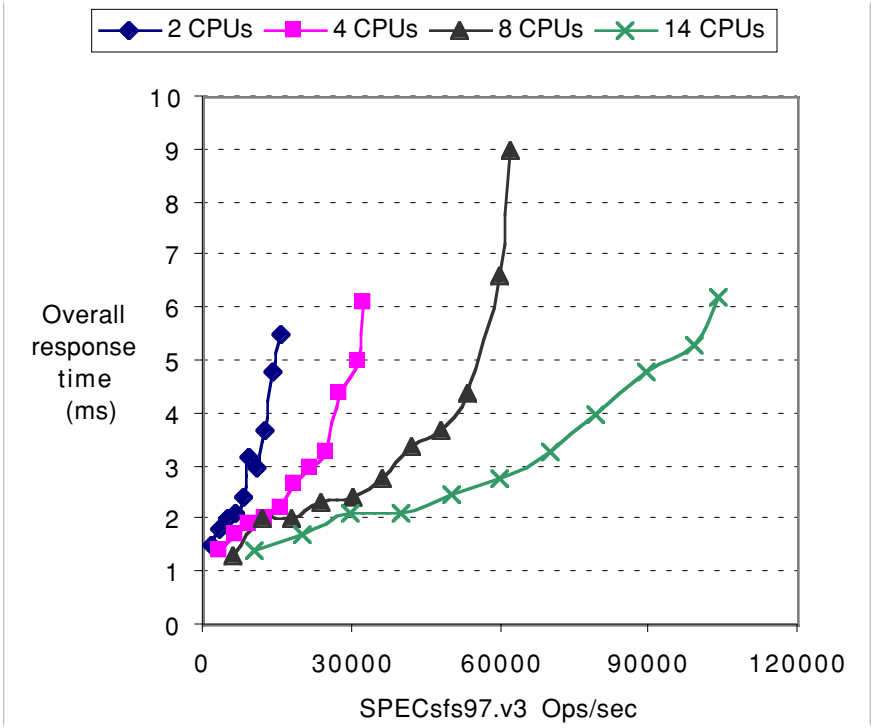


FIGURE 7.32 SPEC sfs97 performance for four EMC Celerra 507 NFS servers: 2, 4, 8, and 14 CPUs provided 15,723, 32,049, 61,809, and 104,607 ops per second. Each processor had its own file system running across about 30 disks. Reported in June 2000, these systems all used DART v2.1.15.200 operating system, 700 MHz Pentium III microprocessors, 0.5 GB of DRAM per processor, and Seagate Cheetah 36GB disks. The total number of disks per system was 67, 133, 265, and 433, respectively. These disks were connected using six Symmetrix Model 8430 disk controllers. The 40-ms average response time limit imposed by SPECsfs97 was not an issue for these machines. The benchmark was retired in June 2001 after bugs were uncovered that affect the comparability of results, which is a serious bug for a benchmark! For more information, see www.spec.org/osg/sfs97/sfs97_notice.html

Figure 7.33 shows results for Dell computers. The performance result represents the number of simultaneous connections the web server can support using the predefined workload. As the disk system is the same, it appears that the large memory is used for a file cache to reduce disk I/O. Although memory of this size may be common in practice, it lessens the role for SPEC WEB99 as a storage benchmark. Note that with a single processor the HTTP web server software and operating system make a significant difference in performance, which grows as the number of processors increase. A dual processor running TUX/Linux is faster than a quad processor running IIS/Windows 2000.

System Name	Result	CPUs	Result/ CPU	HTTP Version/OS	Pentium III	DRAM
PowerEdge 2400/667	732	1	732	IIS 5.0/Windows 2000	667 MHz EB	2 GB
PowerEdge 2400/667	1270	1	1270	TUX 1.0/Red Hat Linux 6.2	667 MHz EB	2 GB
PowerEdge 4400/800	1060	2	530	IIS 5.0/Windows 2000	800 MHz EB	4 GB
PowerEdge 4400/800	2200	2	1100	TUX 1.0/Red Hat Linux 6.2	800 MHz EB	4 GB
PowerEdge 6400/700	1598	4	400	IIS 5.0/Windows 2000	700 MHz Xeon	8 GB
PowerEdge 6400/700	4200	4	1050	TUX 1.0/Red Hat Linux 6.2	700 MHz Xeon	8 GB

FIGURE 7.33 SPEC WEB99 results in 2000 for Dell computers. Each machine uses five 9GB, 10,000 RPM disks except the fifth system, which had seven disk. The first four have 256 KB of L2 cache while the last two have 2 MB of L2 cache.

Examples of Benchmarks of Dependability and Availability

The TPC-C benchmark does in fact have a dependability requirement. The benchmarked system must be able to handle a single disk failure, which means in practice that all submitters are running some RAID organization in their storage system.

Relatively recent efforts have focused on the effectiveness on fault-tolerance in systems. Brown et al [2000] propose that availability be measured by examining the variations in system quality of service metrics over time as faults are injected into the system. For a web server the obvious metrics are performance, measured as requests satisfied per second and degree of fault-tolerance, measured as the number of faults that can be tolerated by the storage subsystem, network connection topology, and so forth.

The initial experiment injected a single fault—such as a disk sector write error—and recorded the system's behavior as reflected in the quality of service metrics. The example compared software RAID implementations provided by Linux, Solaris, and Windows 2000 Server. SPEC WEB99 was used to provide a workload and to measure performance. To inject faults, one of the SCSI disks in the software RAID volume was replaced with an emulated disk. It was just a PC running software with a special SCSI controller that makes the combination of PC, controller, and software appear to other devices on the SCSI bus as a disk drive. The disk emulator allowed the injection of faults. The faults injected included a variety of transient disk faults, such as correctable read errors, and permanent faults, such as disk media failures on writes.

Figure 7.34 shows the behavior of each system under different faults. The two top graphs show Linux (on the left) and Solaris (on the right). Both systems automatically reconstruct data onto a hot spare disk immediately when a disk failure is detected. As can be seen in the figure, Linux reconstructs slowly and Solaris reconstructs quickly. Windows is shown in the bottom; a single disk failed so the data is still available, but this system does not begin reconstructing on the hot spare until the operator gives permission. Linux and Solaris, in contrast, start reconstruction upon the fault injection.

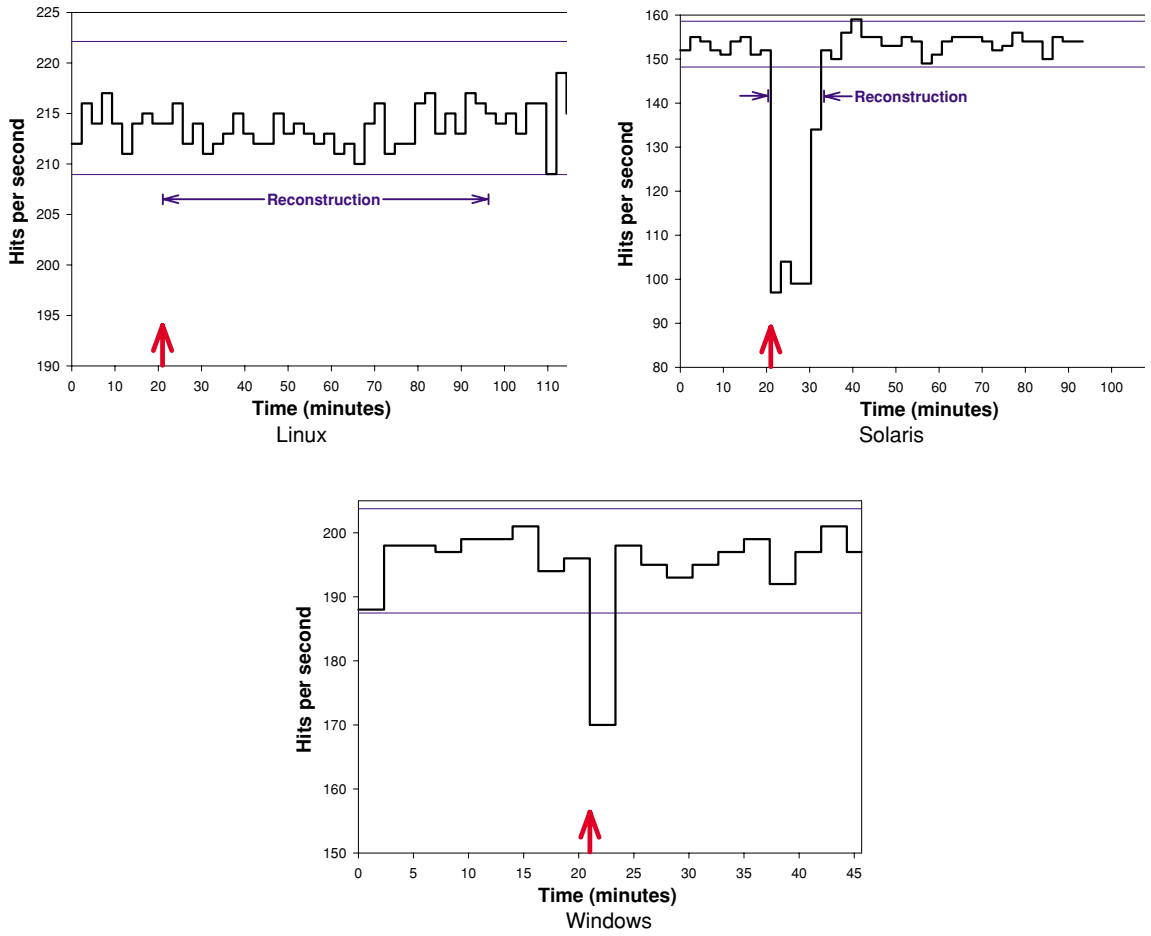


FIGURE 7.34 Availability benchmark for software RAID systems on the same computer running Redhat 6.0 Linux, Solaris 7, and Windows 2000 operating systems. Note the difference in philosophy on speed of reconstruction of Linux vs. Windows and Solaris. The Y-axis is behavior in hits per second running SPEC WEB99. The arrow indicates time of fault insertion. The lines at the top give the 99% confidence interval of performance before the fault is inserted. A 99% confidence interval means the if the variable is outside of this range, the probability is only 1% that this value would appear. **<<Artist: please add reconstruction arrows like in upper right graph to dip in lower graph; there are no Excel sheet for these graphs>>**

As RAID systems can lose data if a second disk fails before completing reconstruction, the longer the reconstruction (MTTR), the lower the availability (see section 6.7 below). Increased reconstruction speed implies decreased application performance, however, as reconstruction steals I/O resources from running applications. Thus, there is a policy choice between taking a performance hit during reconstruction, or lengthening the window of vulnerability and thus lowering the predicted MTTF.

Although none of the tested system documented their reconstruction policies outside of the source code, even a single fault injection was able to give insight into those policies. The experiments revealed that both Linux and Solaris initiate automatic reconstruction of the RAID volume onto a hot spare when an active disk is taken out of service due to a failure. Although Windows supports RAID reconstruction, the reconstruction must be initiated manually. Thus, without human intervention, a Windows system will not rebuild redundancy after a first failure, and will remain susceptible to a second failure indefinitely, which increases the window of vulnerability increases the window of vulnerability.

The fault-injection experiments also provided insight into other availability policies of Linux, Solaris, and Windows 2000 concerning automatic spare utilization, reconstruction rates, transient errors, and so on. Again, no system documented their policies.

In terms of managing transient faults, the fault-injection experiments revealed that Linux's software RAID implementation takes an opposite approach than do the RAID implementations in Solaris and Windows. The Linux implementation is paranoid—it would rather shut down a disk in a controlled manner at the first error, rather than wait to see if the error is transient. In contrast, Solaris and Windows are more forgiving—they ignore most transient faults with the expectation that they will not recur. Thus, these systems are substantially more robust to transients than the Linux system. Note that both Windows and Solaris do log the transient faults, ensuring that the errors are reported even if not acted upon. When faults were not transient, the systems behaved similarly.

Considering real failure data, none of the observed policies is particularly good, regardless of reconstruction behavior. Talagala [1999] reports that transient SCSI errors are frequent in a large system–such as the 368-disk Tertiary Disk farm–yet rarely do they indicate that a disk must be replaced. The logs covering 368 disks for 11 months indicate that 13 disks reported transient hardware errors but only 2 actually required replacement. In this situation, Linux's policy would have incorrectly wasted 11 disks and 11 spares, or 6% of the array. If there were not enough spares, data could have been lost despite no true disk failures. Equally poor would have been the response of Solaris or Windows, as these systems most likely would have ignored the stream of intermittent transient errors from the 2 truly defective disks, requiring administrator intervention to take them offline.

Future directions in availability benchmarking include characterizing a realistic fault-workload, injecting multiple faults, and applying the technique to other fault tolerant systems.

7.10 Crosscutting Issues

Thus far, we have ignored the role of the operating system in storage. In a manner analogous to the way compilers use an instruction set, operating systems determine what I/O techniques implemented by the hardware will actually be used.

For example, many I/O controllers used in early UNIX systems were 16-bit microprocessors. To avoid problems with 16-bit addresses in controllers, UNIX was changed to limit the maximum I/O transfer to 63 KB or less. Thus, a new I/O controller designed to efficiently transfer 1-MB files would never see more than 63 KB at a time under early UNIX, no matter how large the files.

The operating system enforces the protection between processes, which must include I/O activity as well as memory accesses. Since I/O is typically between a device and memory, the operating system must endure safety.

DMA and Virtual Memory

Given the use of virtual memory, there is the matter of whether DMA should transfer using virtual addresses or physical addresses. Here are a couple of problems with DMA using physically mapped I/O:

- Transferring a buffer that is larger than one page will cause problems, since the pages in the buffer will not usually be mapped to sequential pages in physical memory.
- Suppose DMA is ongoing between memory and a frame buffer, and the operating system removes some of the pages from memory (or relocates them). The DMA would then be transferring data to or from the wrong page of memory.

One answer is for the operating system to guarantee that those pages touched by DMA devices are in physical memory for the duration of the I/O, and the pages are said to be *pinned* into main memory. Note that the addresses from for a scatter/gather DMA transfer probably come from the page table.

To ensure protection often the operating system will copy user data into the kernel address space and then transfer between the kernel address space to the I/O device. Relentless copying of data is often the price paid for protection. If DMA supports scatter gather, the operating system may be able to create a list of addresses and transfer sizes to reduce some of the overhead of copying.

Another answer is *virtual DMA*. It allows the DMA to use virtual addresses that are mapped to physical addresses during the DMA. Thus, a buffer could be sequential in virtual memory, but the pages can be scattered in physical memory, and the virtual addresses provide the protection of other processes. The operating system would update the address tables of a DMA if a process is moved using virtual DMA. It Figure 7.35 shows address-translation registers added to the DMA device.

Asynchronous I/O and Operating Systems

As mentioned in section 7.2, disks typically spend much more time in mechanical delays than in transferring data. Thus, a natural path to higher I/O performance is parallelism, trying to get many disks to simultaneously be trying to get data for a program.

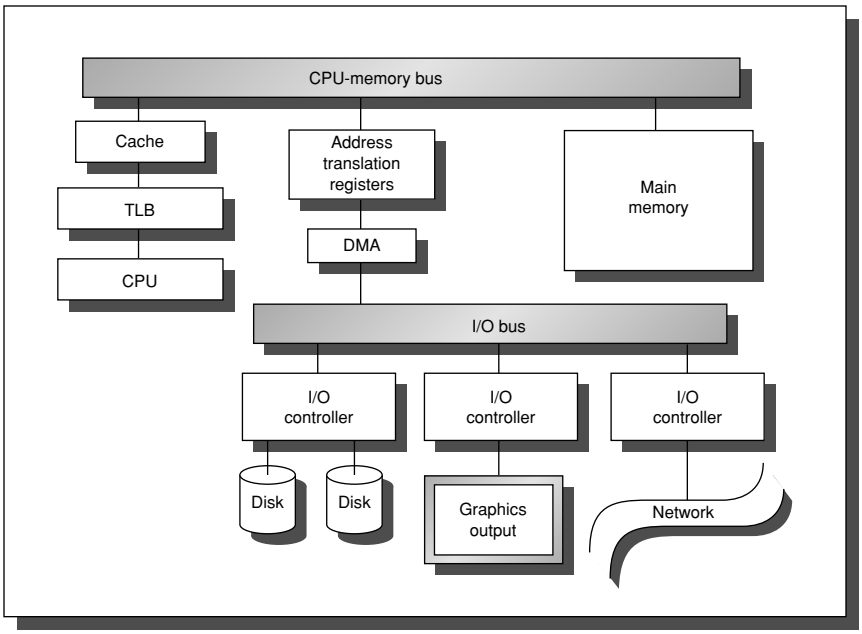


FIGURE 7.35 Virtual DMA requires a register for each page to be transferred in the DMA controller, showing the protection bits and the physical page corresponding to each virtual page.

The straightforward approach to I/O is to request data and then start using it. The operating system then switches to another process until the desired data arrives, and then the operating system switches back to the requesting process. Such a style is called *synchronous I/O*, in that the process waits until the data has been read from disk.

The alternative model is for the process to continue after making a request, and it is not blocked until it tries to read the requested data. Such *asynchronous I/O* allows the process to continue making requests so that many I/O requests can be operating simultaneously. Asynchronous I/O is shares the same philosophy as caches in out-of-order CPUs, trying to get the multiple events happening to get greater bandwidth.

Block Servers vs. Filers

The operating system typically provides the file abstraction on top of blocks stored on the disk. The terms logical units, logical volumes, and physical volumes are related terms used in Microsoft and UNIX systems to refer to subset collections of disk blocks. A *logical unit* is the element of storage exported from a disk array, usually constructed from a subset of the array's disks. A logical unit

appears to the server as single virtual "disk." In a RAID disk array, the logical unit is configured as a particular RAID layout, such as RAID 5. A *physical volume* is the device file used by the file system to access a logical unit. A *logical volume* provides a level of virtualization that enables the file system to split the physical volume across multiple pieces or to stripe data across multiple physical volumes. A logical unit is an abstraction of a disk array that presents a virtual disk to the operating system, while physical and logical volumes are abstractions used by the operating system to divide these virtual disks into smaller, independent file systems.

Having covered some of the terms for collections of blocks, the question arises as to where the file illusion should be maintained: in the server or at the other end of the storage area network?

The traditional answer is the server. It accesses storage as disk blocks and maintains the metadata. Most file systems use a file cache, so it is the job of the server to maintain consistency of file accesses. The disks may be *direct attached*–located inside the server box connected to an I/O bus–or attached over a storage area network, but the server transmits data blocks to the storage subsystem.

The alternative answer is the disk subsystem itself maintains the file abstraction, and the server uses a file system protocol to communicate with storage. Example protocols are Network File System (NFS) for Unix systems and Common Internet File System (CIFS) for Windows systems. Such devices are called *Network Attached Storage* (*NAS*) devices since it makes no sense for storage to be directly attached to the server. The name is something of a misnomer because a storage area network like FC-AL can also be used to connect to block servers. The term *filer* is often used for NAS devices that only provide file service and file storage. Network Appliances is one of the first companies to make filers.

Recently new products have been announced which sit between the compute server and the disk array controller. They provide snapshots of storage, caching, backup and so on. The goal is to make the storage system easier to manage.

The driving force behind placing storage on the network is make it easier for many computers to share information and for operators to maintain it.

Caches Cause Problems for Operating Systems—Stale Data

The prevalence of caches in computer systems has added to the responsibilities of the operating system. Caches imply the possibility of two copies of the data—one each for cache and main memory—while virtual memory can result in three copies—for cache, memory, and disk. These copies bring up the possibility of *stale data*: the CPU or I/O system could modify one copy without updating the other copies (see page 469). Either the operating system or the hardware must make sure that the CPU reads the most recently input data and that I/O outputs the correct data, in the presence of caches and virtual memory.

There are two parts to the stale-data problem:

1. The I/O system sees stale data on output because memory is not up-to-date.
2. The CPU sees stale data in the cache on input after the I/O system has updated memory.

The first dilemma is how to output correct data if there is a cache and I/O is connected to memory. A write-through cache solves this by ensuring that memory will have the same data as the cache. A write-back cache requires the operating system to flush output addresses to make sure they are not in the cache. This flush takes time, even if the data is not in the cache, since address checks are sequential. Alternatively, the hardware can check cache tags during output to see if they are in a write-back cache, and only interact with the cache if the output tries to read data that is in the cache.

The second problem is ensuring that the cache won't have stale data after input. The operating system can guarantee that the input data area can't possibly be in the cache. If it can't guarantee this, the operating system flushes input addresses to make sure they are not in the cache. Again, this takes time, whether or not the input addresses are in the cache. As before, extra hardware can be added to check tags during an input and invalidate the data if there is a conflict.

These problems are like cache coherency in a multiprocessor, discussed in Chapter 6. I/O can be thought of as a second dedicated processor in a multiprocessor.

Switches Replacing Buses

The cost of replacing passive buses with point-to-point links and switches (Chapter 8) is dropping as Moore's Law continues to reduce the cost of components. Combined with the higher I/O bandwidth demands from faster processors, faster disks, and faster local area networks, the decreasing cost advantage of buses means the days of buses in desktop and servers computers are numbered. In 2001, high end servers have already replaced processor-memory buses with switches—see Figure 7.14 on page 506—and switches are now available for high speed storage buses, such as fibre channel.

Not only do switched networks provide more aggregate bandwidth than do buses, the point-to-point links can be much longer. For example, the planned successor to the PCI I/O bus, called *Infiniband*, uses point-to-point links and switches. It delivers 2 to 24 gigabits/second of bandwidth per link and stretches the maximum length of the interconnect using copper wire from 0.5 meters of a PCI bus to 17 meters.

We'll return to discussion of switches in the next chapter.

Replication of Processors for Dependability

In this and prior chapters we have discussed providing extra resources to check and correct errors in main memory and in storage. As Moore's Law continues and

dependability increases in importance for servers, some manufacturers are placing multiple processors on a single chip for the primary purpose of improving the reliability of the processor.

The state-of-the-art in processor dependability is likely the IBM 390 mainframe. Naturally, all its caches and main memory are protected by ECC, but so are the register files. The G6 chips and modules include up to14 processors, some of which are used as built in spares. Each processor has redundant instruction fetch/decode, execution units, L1 cache, and register file to check for errors. At the completion of every instruction, the results produced by the two instruction-execution units are compared and, if equal, the results of the instruction are checkpointed for recovery in case the next instruction fails. Upon detecting an inconsistency, the processor will retry instructions several times to see if the error was transient. If an error is not transient, the hardware can swap in a spare processor in less than a second without disrupting the application.

7.11 Designing an I/O System in Five Easy Pieces

The art of I/O system design is to find a design that meets goals for cost, dependability, and variety of devices while avoiding bottlenecks to I/O performance. Avoiding bottlenecks means that components must be balanced between main memory and the I/O device, because performance—and hence effective cost/performance—can only be as good as the weakest link in the I/O chain. The architect must also plan for expansion so that customers can tailor the I/O to their applications. This expansibility, both in numbers and types of I/O devices, has its costs in longer I/O buses, larger power supplies to support I/O devices, and larger cabinets. Finally, storage must be dependable, adding new constraints on proposed designs.

In designing an I/O system, analyze performance, cost, capacity, and availability using varying I/O connection schemes and different numbers of I/O devices of each type. Here is one series of steps to follow in designing an I/O system. The answers for each step may be dictated by market requirements or simply by cost, performance, and availability goals.

1. List the different types of I/O devices to be connected to the machine, or list the standard buses that the machine will support.
2. List the physical requirements for each I/O device. Requirements include size, power, connectors, bus slots, expansion cabinets, and so on.
3. List the cost of each I/O device, including the portion of cost of any controller needed for this device.
4. List the reliability of each I/O device.
5. Record the CPU resource demands of each I/O device. This list should include
   - Clock cycles for instructions used to initiate an I/O, to support operation

of an I/O device (such as handling interrupts), and complete I/O

- CPU clock stalls due to waiting for I/O to finish using the memory, bus, or cache
- CPU clock cycles to recover from an I/O activity, such as a cache flush

6. List the memory and I/O bus resource demands of each I/O device. Even when the CPU is not using memory, the bandwidth of main memory and the I/O bus is limited.
7. The final step is assessing the performance and availability of the different ways to organize these I/O devices. Performance can only be properly evaluated with simulation, though it may be estimated using queuing theory. Reliability can be calculated assuming I/O devices fail independently and are that MTTFs are exponentially distributed. Availability can be computed from reliability by estimating MTTF for the devices, taking into account the time from failure to repair.

You then select the best organization, given your cost, performance, and availability goals.

Cost/performance goals affect the selection of the I/O scheme and physical design. Performance can be measured either as megabytes per second or I/Os per second, depending on the needs of the application. For high performance, the only limits should be speed of I/O devices, number of I/O devices, and speed of memory and CPU. For low cost, the only expenses should be those for the I/O devices themselves and for cabling to the CPU. Cost/performance design, of course, tries for the best of both worlds. Availability goals depend in part on the cost of unavailability to an organization.

To make these ideas clearer, the next dozen pages go through five examples. Each looks at constructing a disk array with about 2 terabytes of capacity for user data with two sizes of disks. To offer a gentle introduction to I/O design and evaluation, the examples evolve in realism.

To try to avoid getting lost in the details, let's start with an overview of the five examples:

1. *Naive cost-performance design and evaluation*: The first example calculates cost-performance of an I/O system for the two types of disks. It ignores dependability concerns, and makes the simplifying assumption of allowing 100% utilization of I/O resources. This example is also the longest.
2. *Availability of the first example*: The second example calculates the poor availability of this naive I/O design.
3. *Response times of the first example*: The third example uses queuing theory to calculate the impact on response time of trying to use 100% of an I/O resource.

4. *More realistic cost-performance design and evaluation*: Since the third example shows the folly of 100% utilization, the fourth example changes the design to obey common rules of thumb on utilization of I/O resources. It then evaluates cost-performance.
5. *More realistic design for availability and its evaluation*: Since the second example shows the poor availability when dependability is ignored, this final example uses a RAID 5 design. It then calculates availability and performance.

Figure 7.36 summarizes changes in the results in cost-performance, latency, and availability as examples become more realistic. Readers may want to first skim the examples, and then dive in when one catches their fancy.

	Simplistic Organization (Examples 1, 2, 3)	Performance Tuned Organization (Example 4)	Performance and Availability Tuned Organization (Examples 5)
	Small v. Large disks	**Small v. Large disks**	**Small v. Large disks**
Cost of 1.9 TB system	$47,200 v. $45,200	$49,200 v. $47,200	$57,750 v. $54,625
Performance (IOPS)	6,144 v. 3,072 IOPS	4,896 v. 2,448 IOPS	6,120 v. 3,060 IOPS
Cost-Performance	$8 v. $15 per IOPS	$10 v. $19 per IOPS	$9 v. $18 per IOPS
Disk Utilization	100%	80%	80%
Disk Access Latency	238 ms (@ 97%)	41 ms	41 ms
Availability: MTTF (hours)	9,524 v. 15,385	--	2,500,000 v. 5,200,000

FIGURE 7.36 Summary of cost, performance, and availability metrics of the five examples. on the next ten pages. Note that performance in the fifth example assumes all I/Os are reads.

First Example: Naive Design and Cost-Performance

Now let's take a long look at the cost/performance of two I/O organizations. This simple performance analysis assumes that resources can be used at 100% of their peak rate without degradation due to queueing. (The fourth example takes a more realistic view.)

EXAMPLE Assume the following performance and cost information:

- A 2500-MIPS CPU costing $20,000.
- A 16-byte-wide interleaved memory that can be accessed every 10 ns.
- 1000 MB/sec I/O bus with room for 20 Ultra3SCSI buses and controllers.

- Wide Ultra3SCSI buses that can transfer 160 MB/sec and support up to 15 disks per bus (these are also called SCSI *strings*).
- A $500 Ultra3SCSI controller that adds 0.3 ms of overhead to perform a disk I/O.
- An operating system that uses 50,000 CPU instructions for a disk I/O.
- A choice of a large disk containing 80 GB or a small disk containing 40 GB, each costing $10.00 per GB.
- A $1500 enclosure supplies power and cooling to either 8 80 GB disks or 12 40 GB disks.
- Both disks rotate at 15000 RPM, have an 5-ms average seek time, and can transfer 40 MB/sec.
- The storage capacity must be 1920 GB.
- The average I/O size is 32 KB.

Evaluate the cost per I/O per second (IOPS) of using small or large drives. Assume that every disk I/O requires an average seek and average rotational delay. Use the optimistic assumption that all devices can be used at 100% of capacity and that the workload is evenly divided among all disks.

ANSWER I/O performance is limited by the weakest link in the chain, so we evaluate the maximum performance of each link in the I/O chain for each organization to determine the maximum performance of that organization.

Let's start by calculating the maximum number of IOPS for the CPU, main memory, and I/O bus. The CPU I/O performance is determined by the speed of the CPU and the number of instructions to perform a disk I/O:

$$\text{Maximum IOPS for CPU} = \frac{2500 \text{ MIPS}}{\cancel{50{,}000 \text{ instructions per I/O}}} = 50{,}000 \text{ IOPS}$$

The maximum performance of the memory system is determined by the memory cycle time, the width of the memory, and the size of the I/O transfers:

$$\text{Maximum IOPS for main memory} = \frac{(1/10 \text{ ns}) \times 16}{\cancel{32 \text{ KB per I/O}}} \approx 50{,}000 \text{ IOPS}$$

The I/O bus maximum performance is limited by the bus bandwidth and the size of the I/O:

$$\text{Maximum IOPS for the I/O bus} = \frac{1000 \text{ MB/sec}}{\cancel{32 \text{ KB per I/O}}} \approx 31{,}250 \text{ IOPS}$$

Thus, no matter which disk is selected, the I/O bus limits the maximum performance to no more than 31,250 IOPS.

Now it's time to look at the performance of the next link in the I/O chain, the SCSI controllers. The time to transfer 32 KB over the SCSI bus is

$$\text{Ultra3SCSI bus transfer time} = \frac{32 \text{ KB}}{160 \text{ MB/sec}} = 0.2 \text{ ms}$$

Adding the 0.3-ms SCSI controller overhead means 0.5 ms per I/O, making the maximum rate per controller

$$\text{Maximum IOPS per Ultra3SCSI controller} = \frac{1}{0.5 \text{ ms}} = 2000 \text{ IOPS}$$

All organizations will use several controllers, so 2000 IOPS is not the limit for the whole system.

The final link in the chain is the disks themselves. The time for an average disk I/O is

$$\text{I/O time} = 5 \text{ ms} + \frac{0.5}{15000 \text{ RPM}} + \frac{32 \text{ KB}}{40 \text{ MB/sec}} = 5 + 2.0 + 0.8 = 7.8 \text{ ms}$$

Therefore, disk performance is

$$\text{Maximum IOPS (using average seeks) per disk} = \frac{1}{7.8 \text{ ms}} \approx 128 \text{ IOPS}$$

The number of disks in each organization depends on the size of each disk: 1920 GB can be either 24 80-GB disks or 48 40-GB disks. The maximum number of I/Os for all the disks is

$$\text{Maximum IOPS for 24 8-GB disks} = 24 \times 128 = 3072 \text{ IOPS}$$
$$\text{Maximum IOPS for 48 2-GB disks} = 48 \times 128 = 6144 \text{ IOPS}$$

Thus, provided there are enough SCSI strings, the disks become the new limit to maximum performance: 3072 IOPS for the 80-GB disks and 6144 for the 40-GB disks.

Although we have determined the performance of each link of the I/O chain, we still have to determine how many SCSI buses and controllers to use and how many disks to connect to each controller, as this may further limit maximum performance. The I/O bus is limited to 20 SCSI controllers, and the limit is 15 disks per SCSI string. The minimum number of controllers for the 80-GB disks is

$$\text{Minimum number of SCSI-2 strings for 24 80-GB disks} = \left\lceil \frac{24}{15} \right\rceil \text{ or 2 strings}$$

and for 40-GB disks

$$\text{Minimum number of SCSI-2 strings for 48 40-GB disks} = \left\lceil \frac{48}{15} \right\rceil \text{ or 4 strings}$$

Although the formulas suggest the ideal number of strings, they must be

matched with the requirements of the physical packaging. Three enclosures needed for 24 80-GB disks are a poor match to 2 strings, although 4 strings needed for 48 40-GB are a good match to the 4 enclosures. Thus, we increase the number of strings to 3 for the big disks.
We can calculate the maximum IOPS for each configuration:

$$\text{Maximum IOPS for 3 Ultra3SCSI strings} = 3 \times 2000 = 6000 \text{ IOPS}$$
$$\text{Maximum IOPS for 4 Ultra3SCSI strings} = 4 \times 2000 = 8000 \text{ IOPS}$$

The maximum performance of this number of controllers is higher than the disk I/O throughput, so there is no benefit of adding more strings and controllers.

Using the format

Min(CPU limit, memory limit, I/O bus limit, disk limit, string limit)

the maximum performance of each option is limited by the bottleneck (in boldface):

80-GB disks, 2 strings = Min(50,000, 50,000, 31,250, **3072**, 6000) = 3072 IOPS
40-GB disks, 4 strings = Min(50,000, 50,000, 31,250, **6144**, 8000) = 6144 IOPS

We can now calculate the cost for each organization:

$$\text{80-GB disks} = \$20{,}000 + 3 \times \$500 + 24 \times (80 \times \$10) + \$1500 \times \left\lceil \frac{24}{\cancel{8}} \right\rceil = \$45{,}200$$
$$\text{40-GB disks} = \$20{,}000 + 4 \times \$500 + 48 \times (40 \times \$10) + \$1500 \times \left\lceil \frac{48}{\cancel{12}} \right\rceil = \$47{,}200$$

Finally, the cost per IOPS is $15 for the large disks and $8 for the small disks. Calculating the maximum number of average I/Os per second, assuming 100% utilization of the critical resources, the small disks have about 1.9 times better cost/performance than the large disks in this example. n

Second Example: Calculating MTTF of First Example

We ignored dependability in the design above, so let's look at the resulting Mean Time To Fail.

EXAMPLE For the organizations in the last example, calculate the MTTF. Make the following assumptions, again assuming exponential lifetimes:

- CPU/Memory MTTF is 1,000,000 hours
- Disk MTTF is 1,000,000 hours;
- SCSI controller MTTF is 500,000 hours

- Power supply MTTF is 200,000 hours
- Fan MTTF is 200,000 hours
- SCSI cable MTTF is 1,000,000 hours
- Enclosure MTTF is 1,000,000 hours (not including MTTF of one fan and one power supply)

ANSWER Collecting these together, we compute these failure rates:

$$\text{Failure Rate}_{\text{big disks}} = \frac{1}{1000000} + \frac{24}{1000000} + \frac{2}{500000} + \frac{3}{200000} + \frac{3}{200000} + \frac{3}{1000000} + \frac{3}{1000000}$$

$$= \frac{1 + 24 + 4 + 15 + 15 + 3 + 3}{1000000 \text{ hours}} = \frac{65}{1000000 \text{ hours}}$$

$$\text{Failure Rate}_{\text{small disks}} = \frac{1}{1000000} + \frac{48}{1000000} + \frac{4}{500000} + \frac{4}{200000} + \frac{4}{200000} + \frac{4}{1000000} + \frac{4}{1000000}$$

$$= \frac{1 + 48 + 8 + 20 + 20 + 4 + 4}{1000000 \text{ hours}} = \frac{105}{1000000 \text{ hours}}$$

The MTTF for the system is just the inverse of the failure rate

$$\text{MTTF}_{\text{big disks}} = \frac{1}{\text{Failure Rate}_{\text{big disks}}} = \frac{1000000 \text{ hours}}{65} = 15385 \text{ hours}$$

$$\text{MTTF}_{\text{small disks}} = \frac{1}{\text{Failure Rate}_{\text{small disks}}} = \frac{1000000 \text{ hours}}{105} = 9524 \text{ hours}$$

The smaller, more numerous drives have almost twice the cost performance but about 60% of the reliability, and the collective reliability for both options is only about 1% of a single disk.

Third Example: Calculating Response time of First Example

The first example assumed that resources can be used 100%. It is instructive to see the impact on response time as we approach 100% utilization of a resource. Let's do this for just one disk to keep the calculations simple; the exercises do more disk.

EXAMPLE Recalculate performance in terms of response time. To simplify the calculation, ignore the SCSI strings and controller and just calculate for one disk. From the example above, the average disk service time is 7.8 ms. Assume Poisson arrivals with an exponential service time. Plot the mean response time for the following number of I/Os per second: 64, 72, 80, 88, 96, 104, 112, 120, and 124.

ANSWER To be able to calculate the average response time, we can use the equation for an M/M/1 queue given the assumptions above about arrival rates and services times. From page 538, the equations for time waiting in the queue is (evaluated for 64 I/O requests per second):

$$\text{Server utilization} = \text{Arrival rate} \times \text{Time}_{\text{server}} = 64 \times 0.0078 = 0.50$$

$$\text{Time}_{\text{queue}} = \text{Time}_{\text{server}} \times \frac{\text{Server utilization}}{(1 - \text{Server utilization})} = 7.8 \text{ ms} \times \frac{0.50}{1 - 0.50} = 7.8 \times \frac{0.5}{0.5} = 7.8 \text{ ms}$$

$$\text{Time}_{\text{system}} = \text{Time}_{\text{server}} + \text{Time}_{\text{queue}} = 7.8 + 7.8 = 15.6 \text{ ms}$$

Figure 7.37 shows the utilization and mean response time for other request rates, and Figure 7.38 plots the response times versus request rate.

Request rate	Utilization (%)	Mean response time (ms)
64	50%	15.6
72	56%	17.8
80	62%	20.7
88	69%	24.9
96	75%	31.1
104	81%	41.3
112	87%	61.7
120	94%	121.9
124	97%	237.8

FIGURE 7.37 Utilization and mean response time for one disk in the prior example, ignoring the impact of SCSI buses and controllers. The nominal service time is 7.8 ms. 100% utilization of disks is unrealistic.

n

Fourth Example: More Realistic Design and Cost-Performance

Figure 7.38 shows the severe increase in response time when trying to use 100% of a server. A variety of rules of thumb have been evolved to guide I/O designers to keep response time and contention low:

- No disk should be used more than 80% of the time.
- No disk arm should be seeking more than 60% of the time.
- No disk string should be utilized more than 40%.
- No I/O bus should be utilized more than 75%.

One reason the SCSI string bandwidth is set so low is that there is about a 20% SCSI command overhead on data transfers, further reducing available bandwidth.

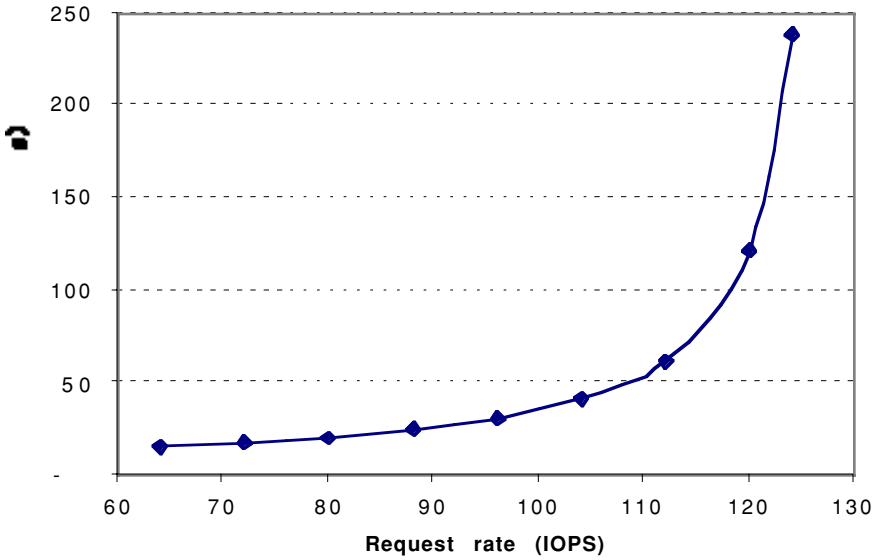


FIGURE 7.38 X-Y plot of response times in Figure 7.37.

EXAMPLE Recalculate performance in the example above using these rules of thumb, and show the utilization of each component before and after these assumptions.

ANSWER First let's see how much the resources are utilized using the assumptions above. The new limit on IOPS for disks used 80% of the time is 128×0.8 = 102 IOPS. Notice that the IOPS is in the relatively flat part of the response time graph in Figure 7.38, as we would hope. The utilization of seek time per disk is

$$\frac{\text{Time of average seek}}{\text{Time between I/Os}} = \frac{5}{\frac{1}{102 \text{ IOPS}}} = \frac{5}{9.8} = 51\%$$

This is below the rule of thumb of 60%.

The I/O bus can support 31,250 IOPS but the most that is used before was 6144 IOPS, which is just 6144/31250 or a 20% utilization. Thus, the I/O bus is far below the suggested limit.

The biggest impact is on the SCSI bus. A SCSI bus with 12 disks uses $12 \times 102 / 2000 = 61\%$. The revised limit per SCSI string is now 40%, which limits a SCSI bus to 800 IOPS.

With this data, we can recalculate IOPS for each organization:

80-GB disks, 3 strings = Min(50,000,50,000, 31,250, 2448, **2400**) = 2400 IOPS
40-GB disks, 4 strings = Min(50,000,50,000, 31,250, 4896, **3200**) = 3200 IOPS

Under these assumptions, the small disks have about 1.3 times the performance of the large disks.

Clearly, the string bandwidth is the bottleneck now. The number of disks per string that would not exceed the guideline is

$$\text{Number of disks per SCSI string at full bandwidth} = \left\lfloor \frac{800}{\cancel{102}} \right\rfloor = \lfloor 7.8 \rfloor = 7 \text{ disks}$$

and the ideal number of strings is

$$\text{Number of SCSI strings with 80-GB disks} = \left\lceil \frac{24}{\cancel{7}} \right\rceil = \lceil 3.6 \rceil = 4 \text{ strings}$$

$$\text{Number of SCSI strings with 40-GB disks} = \left\lceil \frac{48}{\cancel{7}} \right\rceil = \lceil 6.9 \rceil = 7 \text{ strings}$$

As mentioned before, the number of strings must match the packaging requirements. Three enclosures needed for 24 80-GB disks are a poor match to 4 strings, and 7 strings needed for 48 40-GB disks are a poor match to the 4 enclosures. Thus, we increase the number of enclosures to 4 for the big disks and increase the number of strings to 8 for small disks, so that each small-disk enclosure has two strings.

The IOPS for the suggested organization is:

80-GB disks, 4 strings = Min(50,000,50,000, 31,250, **2448**, 3200) = 2448 IOPS
40-GB disks, 8 strings = Min(50,000,50,000, 31,250, **4896**, 6400) = 4896 IOPS

We can now calculate the cost for each organization:

80-GB disks, 4 strings = \$20,000 + 4 × \$500 + 24 × (80 × \$10) + 4 x \$1500 = \$47,200
40-GB disks, 8 strings = \$20,000 + 8 × \$500 + 48 × (40 × \$10) + 4 x \$1500 = \$49,200

The respective cost per IOPS is \$19 versus \$10, or an advantage of about 1.9 for the small disks. Compared with the naive assumption that we could use 100% of resources, the cost per IOPS increased about 1.3 times.

Figure 7.39 shows the utilization of each resource before and after following these guidelines. Exercise 7.18 explores what happens when this SCSI limit is relaxed.

n

Fifth Example: Designing for Availability

Just as the fourth example made a more realistic design for performance, we can show a more realistic design for dependability. To tolerate faults we will add re-

	Rule of Thumb	100% Utilization		Following the Rule of Thumb			
Resource		**80-GB disks, 3 strings**	**40-GB disks, 4 strings**	**80-GB disks, 3 strings**	**40-GB disks, 4 strings**	**80-GB disks, 4 strings**	**40-GB disks, 8 strings**
CPU		6%	12%	5%	6%	5%	10%
Memory		6%	12%	5%	6%	5%	10%
I/O bus	75%	10%	20%	8%	10%	8%	16%
SCSI buses	40%	**51%**	**77%**	40%	40%	31%	31%
Disks	80%	**100%**	**100%**	78%	52%	80%	80%
Seek utilization	60%	**64%**	**64%**	50%	33%	51%	51%
IOPS		3072	6144	2400	3200	2448	4896

FIGURE 7.39 The percentage of utilization of each resource, before and after using the rules of thumb. Bold font shows resources in violation of the rules of thumb. Using the prior example, the utilization of three resources violated the rules of thumb: SCSI buses, disks, and seek utilization.

dundant hardware: extra disks, controllers, power supplies, fans, and controllers in a RAID-5 configuration.

To calculate reliability now, we need a formula to show what to expect when we can tolerate a failure and still provide service. To simplify the calculations we assume that the lifetimes of the components are exponentially distributed and the there is no dependency between the component failures. Instead of mean time to failure, we calculate *mean time until data loss* (*MTDL*), for a single failure will not, in general, result in lost service. For RAID, data is lost only if a second disk failure occurs in the group protected by parity before the first failed disk is repaired. Mean time until data loss is the mean time until a disk will fail divided by the chance that one of the remaining disks in the parity group will fail before the first failure is repaired. Thus, if the chance of a second failure before repair is large, then MTDL is small, and vice versa.

Assuming independent failures, since we have N disks, the mean time until one disk fails is $\text{MTTF}_{\text{disk}}/\text{N}$. The good approximation of the probability of the second failure is MTTR over the mean time until one of the remaining G -1 disks in the parity group will fail. Similar to before, the means time for G -1 disks is $(\text{MTTF}_{\text{disk}}/(\text{G}-1))$. Hence, a reasonable approximation for MTDL for a RAID is [Chen 1994]:

$$\text{MTDL} = \frac{\text{MTTF}_{\text{disk}}/\text{N}}{\dfrac{\text{MTTR}_{\text{disk}}}{(\text{MTTF}_{\text{disk}}/(\text{G}-1))}} = \frac{\text{MTTF}_{\text{disk}}^2/\text{N}}{(\text{G}-1)\times\text{MTTR}_{\text{disk}}} = \frac{\text{MTTF}_{\text{disk}}^2}{\text{N}\times(\text{G}-1)\times\text{MTTR}_{\text{disk}}}$$

where N is the number of disks in the system and G is the number of disks in a group protected by parity. Thus, MTDL increases with increased disk reliability, reduced parity group size, and reduced mean time to repair (MTTR).

The physical design of the disk array gives a strong suggestion to the parity group size. Figure 7.40 shows two ways of organizing a RAID. The problem with option 1 is if the string or string controller fails, then all the disks in the RAID group fail, and data is lost. Option 2, called *orthogonal RAID*, in contrast loses only one disk per RAID group even if a string controller fails. Note that if the string is located in a single enclosure, then orthogonal RAID also protects against power supply and fan failures.

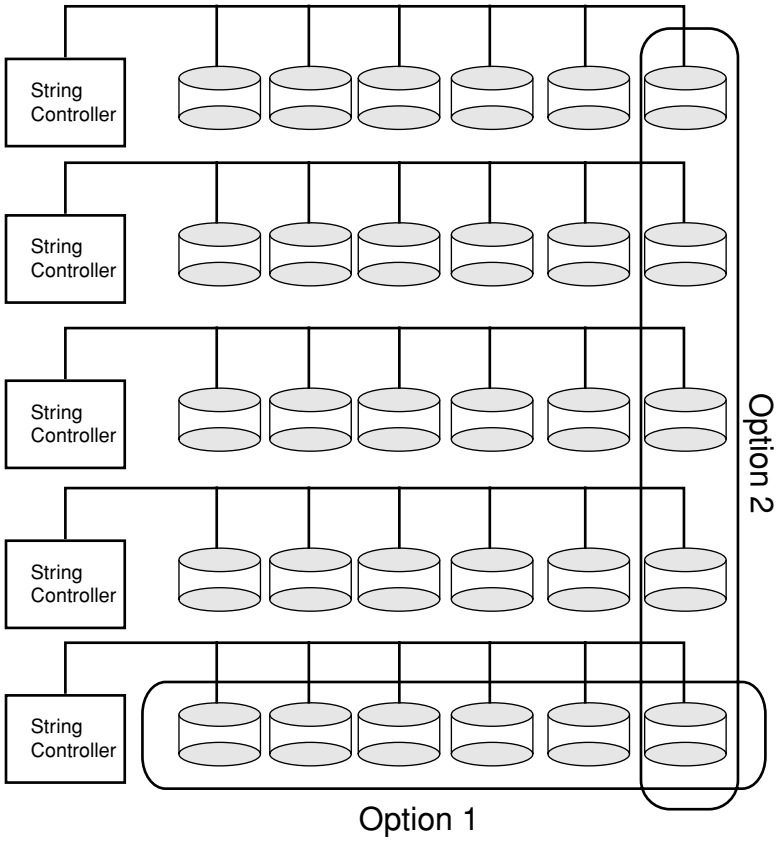


FIGURE 7.40 Two RAID organizations. Orthogonal RAID (Option 2) allows the RAID fault tolerant scheme to protect against string faults as well as disk faults.

EXAMPLE For the organizations in the fourth example and using the MTTF ratings of the components in the second example, create orthogonal RAID arrays and calculate the MTDL for the arrays.

ANSWER Both organizations use four enclosures, so we add a fifth enclosure in each to provide redundancy to tolerate faults. The redundant enclosure

contains 1 controller and 6 large disks or 2 controllers and 12 small disks. The failure rate of the enclosures can be calculated similar to a prior example:

$$\text{Enclosure Failure Rate}_{\text{big disks}} = \frac{6}{1000000} + \frac{1}{500000} + \frac{1}{200000} + \frac{1}{200000} + \frac{1}{1000000} + \frac{1}{1000000}$$
$$= \frac{6+2+5+5+1+1}{1000000\text{hours}} = \frac{20}{1000000\text{hours}}$$

$$\text{Enclosure Failure Rate}_{\text{small disks}} = \frac{12}{1000000} + \frac{2}{500000} + \frac{1}{200000} + \frac{1}{200000} + \frac{2}{1000000} + \frac{1}{1000000}$$
$$= \frac{12+4+5+5+2+1}{1000000\text{hours}} = \frac{29}{1000000\text{hours}}$$

The MTTF for each enclosure is just the inverse of the failure rate

$$\text{MTTF}_{\text{big disks}} = \frac{1}{\text{Failure Rate}_{\text{big disks}}} = \frac{1000000\text{hours}}{20} = 50000\text{hours}$$

$$\text{MTTF}_{\text{small disks}} = \frac{1}{\text{Failure Rate}_{\text{small disks}}} = \frac{1000000\text{hours}}{29} \approx 34500\text{hours}$$

As the array can continue to provide data despite the loss of a single component, we can modify the disk MTDL to calculate for enclosures:

$$\text{MTDL}_{\text{RAID}} = \frac{\text{MTTF}^2_{\text{enclosure}}}{N \times (G-1) \times \text{MTTR}_{\text{enclosure}}}$$

In this case, N = G = the number of enclosures. Even if we assume it takes 24 hours to replace an enclosure ($\text{MTTR}_{\text{enclosure}}$), the MTDL for each organization is:

$$\text{MTDL}_{\text{big disk RAID}} = \frac{50,000^2}{5 \times (5-1) \times 24} = \frac{2,500,000,000\text{hours}}{480} \approx 5,200,000\text{hours}$$

$$\text{MTDL}_{\text{small disk RAID}} = \frac{34,500^2}{5 \times (5-1) \times 24} = \frac{1,190,250,000\text{hours}}{480} \approx 2,500,000\ \text{hours}$$

We can now calculate the higher cost for RAID 5 organizations:

80-GB disks, 5 strings = \$20,000 + 5 × \$500 + 30 × (80 × \$10) + 5 x \$1500 = \$54,625
40-GB disks, 10 strings = \$20,000 + 10 × \$500 + 60 × (40 × \$10) + 5 x \$1500= \$57,750

If we evaluated the cost-reliability, for large disk costs \$11 per thousand hours of MTDL while the small disk system costs \$23 per thousand hours of MTDL.

The IOPS for the more dependable organization now depends on the mix of reads and writes in the I/O workload, since writes in RAID 5 system

are much slower than writes for RAID 0 systems. For simplicity, lets assume 100% reads. (The exercises look at other workloads.) Since RAID-5 allows reads to all disks, and there are more disks and strings in our dependable design, read performance improves as well as dependability.:

80-GB disks, 5 strings = Min(50,000,50,000, 31,250, **3060**, 4000) = 3060 IOPS
40-GB disks, 10 strings= Min(50,000,50,000, 31,250, **6120**, 8000) = 6120 IOPS

We can now calculate the cost per IOPS for RAID 5 organizations. Compared to the results from the first example, the respective cost per IOPS increased slightly from $15 to $18 and from $8 to $9, respectively. The exercises look at the impact on cost-performance as the I/O workload includes reads in a RAID 5 organization.

n

In both cases, given the reliability assumptions above, the mean time to data loss for redundant arrays containing several dozen disks is greater than the mean time to failure of a single disk. At least for a read-only workload, the cost-performance impact of dependability is small. Thus, a weakness was turned into a strength: the larger number of components allows redundancy so that some can fail without affecting the service.

7.12 Putting It All Together: EMC Symmetrix and Celerra

The EMC Symmetrix is one of the leading disk arrays that works with most computer systems, and the EMC Celerra is a relatively new filer for both UNIX NFS and Windows CIFS file systems. Both machines have significant features to improve dependability of storage. After reviewing the two architectures, we'll summarize the results of their performance and dependability benchmarks.

EMC Symmetrix 8000

The Symmetrix 8000 holds up to 384 disks, which are protected either via mirroring (RAID 1) or via a variation of RAID-5 that EMC calls RAID-S. The RAID-S group size is 4 or 8 drives. At 73 GB per drive, the total raw capacity is about 28 terabytes. Figure 7.41 shows its organization.

The internal architecture is built around four busses that run at 60 MHz and transfer 64 bits of data and 16 bits of error correcting code (ECC). With this scheme, any number of incorrect bits in any two nibbles can be detected while any number of incorrect bits in one nibble can be corrected. Each component is connected to two buses so that failure of a bus does not disconnect the component from the system. The components that connect to these four buses are:

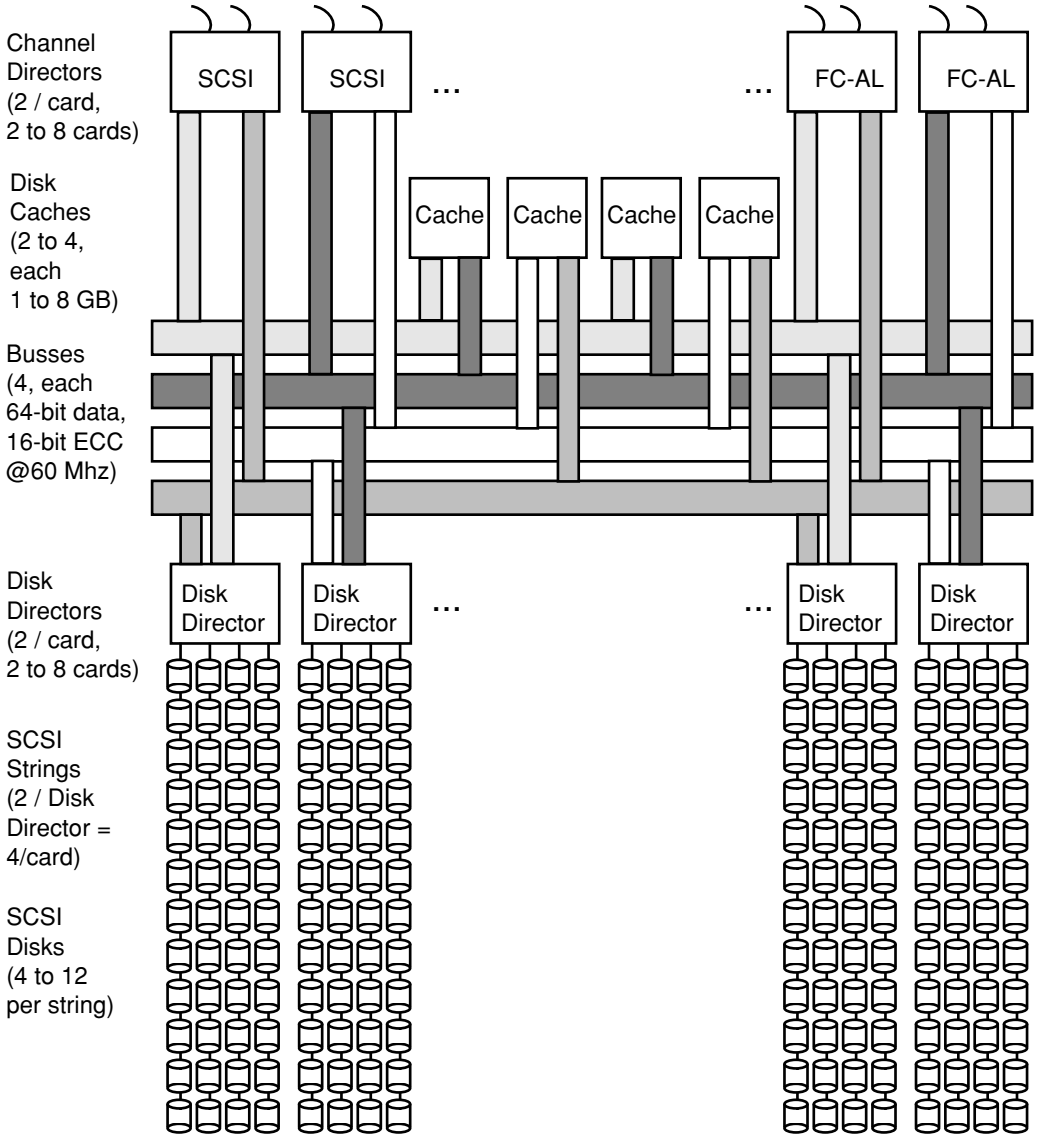


FIGURE 7.41 EMC Symmetrix 8000 organization. Every component is paired so that there is no single point of failure. If mirroring is used, then the disks are also paired. If RAID-S is used, there is one block of parity for every 4 to 8 blocks.

- *Channel Directors* connect the server host to the internal busses of the disk array, and work with SCSI, FC-AL, and ESCON (IBM mainframe I/O bus). They also run the algorithms to manage the caches. Up to sixteen channel directors are provided, packaged two directors per card.
- *Disk Cache Memory* acts as speed-matching buffer between the host servers and the disks; in addition, it exploits locality to reduce accesses to the disks. There are up to four slots for cache boards, and each contains 1 GB to 8 GB. Each system has at least two cache boards, producing systems that have from 2 GB to 32 GB of cache. EMC claims 90% to 95% read hit rates for the largest cache size. The caches will also buffer writes, allowing the system to report that the write is completed before it reaches the disk. The Channel Director monitors the amount of dirty data. It will not send the write complete signal if the cache is behind and the Channel Director needs to flush more data to disk to reduce the length of the write buffer queue. Symmetrix does not include batteries in the cache for nonvolatility, but instead provides batteries for the whole array to protect the whole system from short power failures.
- *Disk Directors* connect the internal busses to the disks. Each disk director has two Ultra1 SCSI strings, running at 40 MB/sec in 2001. Each string uses a redundant SCSI controller on a different director to watch the behavior of the primary controller and to take over in case it fails. Up to sixteen disk directors are provided, packaged two directors per card. With up to 12 SCSI disks on a string, we get $16 \times 2 \times 12 = 384$ drives.

Both directors contain the same embedded computers. They have two PowerPC 750s running at 333 MHz, each with 16 MB of DRAM and a 1MB L2 cache. The PowerPC buses contain 32 bits of data plus 4 bits for ECC, and run at 33 MHz. These computers also have several DMA devices, so requested data does not go through the computer memory, but directly between the disks and the cache or the cache and the host bus. The processors act independently, sharing only boot ROMs and an Ethernet port.

The storage system can exploit modifications of disks as requested by EMC, which disk manufacturers in turn make available to others. For example, some disks can understand a notion of request priority allowing the storage system to submit more requests to the drives knowing that the drives will maintain proper order in their internal queues.

The Symmetrix disk cache is controlled by a combination of LRU and prefetching algorithms, fetching between 2 and 12 blocks at a time. The cache memory is independent of the PowerPC processors. The cache is structured as a sequence of tracks of data each 32 KB long. Each 4-KB segment of a track has associated metadata that contains CRC checksums on the data and metadata used by other Symmetrix features. The Symmetrix provides “end-to-end” checking on transfers between disk to cache and between cache and the host server by ensuring that the both the DRAM ECC and associated CRC checksums match at the beginning and end of every data transfer.

As faults must be activated before they can become effective errors and then corrected, all cache locations are periodically read and rewritten using the ECC on memory to correct single bit errors and detect double bit errors. Cache scrubbing also keeps a record of errors for each block. If the channel director finds an uncorrectable error, then this section of the cache is "fenced" and removed from service. The data is first copied to another block of the cache. The service processor (see below) then contacts EMC to request repair of the failed component.

During idle time, disk scrubbing is performed analogously to cache scrubbing above, with the same benefit of turning latent errors into activated errors during relatively idle times. Correctable errors are logged, and uncorrectable errors cause the bad disk sector to be replaced, with the missing data coming from the redundant information. If too many sectors in a track must be skipped, the whole track is fenced. Such repairs to the cache and to the disks are transparent to the user.

Rather than have a XOR engine only in the disk directors, RAID-5 parity calculations are done inside the drives themselves and combined by the directors as needed. As mentioned above, small writes in RAID-5 involve four accesses over two disks. This optimization avoids having to read the rest of the data blocks of a group to calculate parity on a "small" write. Symmetrix supplies the new data and asks the disk to calculate which bits changed, and then passes this information to the disk containing parity for it to read the old parity and modify it.

Having the disk drive perform the XOR calculations provides two benefits. First, it avoids having a XOR engine become a bottleneck by spreading the function to each disk. Second, it allows the older data to be read and rewritten without an intervening seek; the same benefit applies when updating parity.

In addition to the dynamic nature of managing the cache, the Symmetrix can change how mirroring works to get better performance from the second disk. It monitors access patterns to the data and changes policy depending on the pattern. Data is organized into logical volumes, so there is a level of indirection between the logical data accesses and the layout of data on the physical disks. Depending on whether accesses are sequential or random, the mirror policy options include:

- *Interleaved*: the mirrors alternate which cylinders they serve, ranging from every other cylinder to large blocks of cylinders. This policy helps with sequential accesses by allowing one disk to seek to the next cylinder while the other disk is reading data.
- *Mixed*: One disk serves the first half of the data and one disk serves the second half. This policy helps with random accesses by allowing the two independent requests to be overlapped.

Policy options also let only a single disk serve all accesses, which helps error recovery.

The Symmetrix 8000 also has a service processor. It is just a laptop that talks to all directors over an internal Ethernet. To allow remote maintenance of the disk array, the service processor is also connected to a telephone line. All system er-

rors are logged to the service processor, which filters the log to determine whether it should contact EMC headquarters to see if repair is warranted. That is, it is predicting potential failures. If it suspects a failure, it contacts support personnel who review the data and decide if intervention is required. They can call back into the service processor to collect more data and to probe the Symmetrix to determine the root cause of the error. A customer service engineer is then dispatched to replace the failing component.

In addition to error logging and remote support, the service processor is used for code installation and upgrades, creating and modifying system configurations, running scripts, and other maintenance activities. To allow upgrades in the field, the service processor can systematically upgrade the EEPROMs of each director and then put the director into a busy state so that it performs no storage accesses until it reboots with the new software.

EMC Celerra 500

The Celerra contains no disk storage itself, but simply connects to clients on one side and to Symmetrix disk arrays on the other. Using the NAS terminology, its is called a *filer*. Its job is to translate the file requests from clients into commands for data from Symmetrix arrays, and to transfer files as requested.

The Celerra has 14 slots for *Data Movers*, which are simply PC motherboards that connect to the Symmetrix array via two SCSI buses. Each data mover contains a 700 MHz Pentium III processor, PCI bus, and 512 MB of DRAM. It also supports several varieties of network cards with varying number of networks: ATM, FDDI, two 1-Gigabit Ethernets per card, and eight 100-Mbit Ethernets per card. Each data mover acts as a fully autonomous file server, running EMC's real-time operating system called DART.

In addition to the Data Movers, there are two *Control Stations*, which act analogously to the service processor in the Symmetrix array. A pair of control stations provides protection in case one fails. The hardware used in control stations is the same as the hardware used in the Data Movers, but with a different function. They run Linux as their operating system.

Celerra has an extensive set of features to provide dependable file service:

- The Celerra has multiple fans, multiple power supplies, multiple batteries, and two power cords to the box. In every case, a single failure of one of these components does not affect behavior of the system.
- Each Data Mover can contact all the disks in the Symmetrix array over either SCSI bus, allowing the Symmetrix to continue despite a bus failure.
- Each Data Mover has two internal Ethernet cards, allowing communication with the Control Station to continue even if one card or network fails.
- Each Data Mover has at least two interfaces for clients, allowing redundant connections so that clients have at least two paths to each Data Mover.

- The software allows a Data Mover to act as a standby spare.
- There is space for a redundant Control Station, to take over in case the primary Control Station fails.

The Celerra relies on the Service Processor in the Symmetrix box to call home when attention is needed.

EMC Symmetrix and Celerra Performance and Availability

Figure 7.32 on page 544 shows the performance of the Celerra 507 with the Symmetrix 8700 running SPECsfs97, as the number of Data Movers scales from 2 to 14. The 100,000 NFS operations per second with 14 Data Movers set the record at the time it was submitted. Despite their focus on dependability–with a large number of features to detect and predict failures and to reduce mean time to repair–the Celerra/Symmetrix combination had leading performance on benchmark results.

The disk cache of the EMC Symmetrix disk array was subjected to initial availability and maintainability benchmarking (Lambright [2000]). A small number of experiments were performed with the goal of learning more about how to go about doing availability and maintainability benchmarks.

Faults were injected via software, going from narrowly focused faults to very broad faults. These were not intended to represent typical faults: they were intended to stress the system, and many are unlikely to occur in real systems.

As mentioned above, the EMC array has the ability to shrink the size of the cache in response to faults by fencing off a portion of the cache. It also has error correction that can prevent a fault from causing a failure. The system under test had 8 GB of cache and 96 disks each with 18 GB of capacity, and it was connected to an IBM mainframe over 12 channels. The workload was random I/O with 75% reads and 25% writes. Performance was evaluated using EMC benchmarks.

The first fault tests the behavior of the system when the CPUs in the front and back end get confused: the data structure representing which portions of the cache were available or fenced is not identical in each CPU. Thus, some CPUs assumed that the cache was bigger than what other CPUs assumed. Figure 7.42 shows the behavior when half of the CPUs are out-of-sync. A fault was injected at the 5-th minute and corrected at the 10-th minute. The I/O rate increases in the 12-th minute as the system catches up with delayed requests.

Performance dropped because some CPUs would try to access disabled memory, generating an error. As each error happened there was a short delay to report it; as the number of CPUs reporting errors increased, so did the delay.

The second fault experiment forced improper behavior of a cache software lock. The lock protects metadata related to the LRU replacement algorithm. The fault simulated a CPU in an infinite loop that repeatedly takes the cache lock without releasing it. Figure 7.43 shows the results: the flawed CPU takes the lock in the 6-th, 10-th, and 15-th minute, each time holding it for 20 seconds. Note

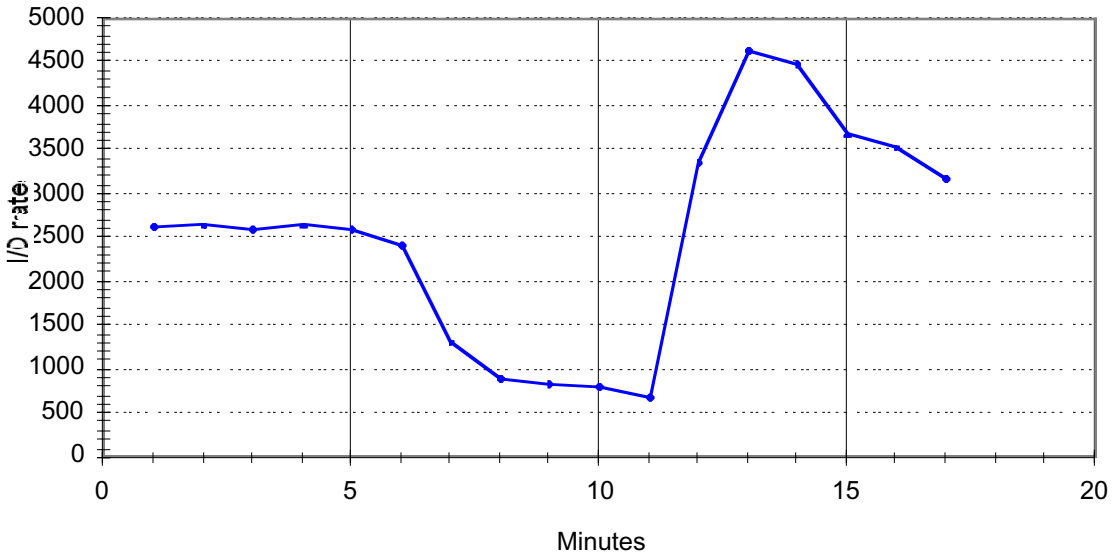


FIGURE 7.42 I/O rate as Symmetrix CPUs become inconsistent in their model of the size of the cache. Faults were inserted in the 5-th minute and corrected at in the 10-th minute.

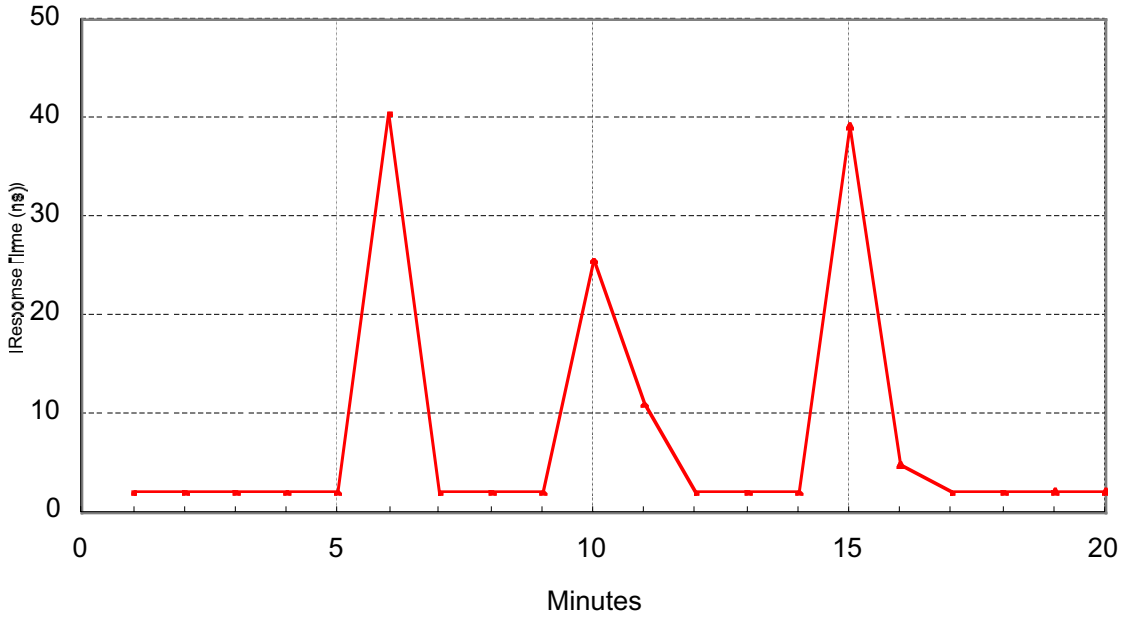


FIGURE 7.43 Host Response time as a rogue CPU hogs a lock for metadata. The lock was held for 20 seconds at minutes 6, 10, and 15.

that the Y-axis reports response time, so smaller is better. As expected, response time was impacted by this fault

Standard maintenance techniques fixed the first error, but the second error was much more difficult to diagnose. The benchmark experiments led to suggestions on improving EMC management utilities.

7.13 Another View: Sanyo DSC-110 Digital Camera

At the other end of the storage spectrum from giant servers are digital cameras. Digital cameras are basically embedded computers with removable, writable, nonvolatile storage and interesting I/O devices. Figure 7.44 shows our example.

FIGURE 7.44 The Sanyo VPC-SX500. Although newer cameras offer more pixels per picture, the principles are the same. This 1360 x 1024 pixel digital camera stores pictures either using Compact Flash memory, which ranges from 8 MB to 64 MB, or using a 340 MB IBM Microdrive. It is 4.3" wide x 2.5" high x 1.6" deep, and it weighs 7.4 ounces. In addition to taking still picture and converting it to JPEG format every 0.9 seconds, it can record a Quick Time video clip at VGA Size (640 x 480). Using the IBM Microdrive, it can record up to 7.5 minutes at 15 frames per second with sound (10,000 images) or 50 minutes for 160 x 120 pixel video with sound. Without video, it can record up to 12 hours of 8-bit 8 KHz audio. The Flash memory storage capacity is 5X to 40X shorter, so its video and audio capacity are also 5X to 40X smaller. One technological advantage is the use of a custom system on a chip to reduce size and power, so the camera only needs two AA batteries to operate versus four in other digital cameras.

When powered on, the microprocessor first runs diagnostics on all components and writes any error messages to the liquid crystal display (LCD) on the back of the camera. This camera uses a 1.8-inch low temperature polysilicon TFT

color LCD. When a photographer takes a picture, he first holds the shutter halfway so that the microprocessor can take a light reading. The microprocessor then keeps the shutter open to get the necessary light, which is captured by a charged-couple device (CCD) as red, green, and blue pixels. For the camera in Figure 7.44, the CCD is a 1/2 inch, 1360 x 1024 pixel, progressive scan chip. The pixels are scanned out row-by-row and then passed through routines for white balance, color, and aliasing correction, and then stored in a 4-MB frame buffer. The next step is to compress the image into a standard format, such as JPEG, and store it in the removable Flash memory. The photographer picks the compression, in this camera called either fine or normal, with a compression ratio of 10x to 20x. An 8 MB Flash memory can store at least 19 fine-quality compressed images or 31 normal-quality compressed images. The microprocessor then updates the LCD display to show that there is room for one less picture.

Although the above paragraph covers the basics of a digital camera, there are many more features that are included: showing the recorded images on the color LCD display; sleep mode to save battery life; monitoring battery energy; buffering to allow recording a rapid sequence of uncompressed images; and, in this camera, video recording using MPEG format and audio recording using WAV format.

The VPC-SX500 camera allows the photographer to use a 340 MB IBM Microdrive instead of CompactFlash memory. Figure 7.45 compares CompactFlash and the IBM Microdrive.

Characteristics	**Sandisk Type I CompactFlash SDCFB-64-144**	**Sandisk Type II CompactFlash SDCF2B-300-530**	**IBM 340 MB Microdrive DSCM-10340**
Formatted data capacity (MB)	64	300	340
Bytes per sector	512	512	512
Data transfer rate in MB/second	4 (burst)	4 (burst)	2.6 to 4.2
Link speed to buffer in MB/second	6	6	13
Power standby/operating in Watts	0.15 / 0.66	0.15 / 0.66	0.07/ 0.83
Size: height x width x depth in inches	1.43 x 1.68 x 0.13	1.43 x 1.68 x 0.20	1.43 x 1.68 x 0.20
Weight in grams (454 grams/pound)	11.4	13.5	16
Write cycles before sector wear out	300,000	300,000	not applicable
Load/Unload cycles (on/off)	not applicable	not applicable	300,000
Nonrecoverable read errors per bits read	<1 per 10^{14}	<1 per 10^{14}	< 1 per 10^{13}
Shock tolerance: operating, not operating	2000 G, 2000 G	2000 G, 2000G	175 G, 1500G
Mean Time Between Failures (hours)	>1,000,000	>1,000,000	(see caption)
Best Price (in August 2001)	$41	$595	$165

FIGURE 7.45 Characteristics of three storage alternatives for digital cameras. IBM matches the Type II form factor in the Microdrive, while the CompactFlash card uses that space to include many more Flash chips. IBM does not quote MTTF for the 1.0-inch drives, but the service life is five years or 8800 powered on hours, whichever is first.

The CompactFlash standard package was proposed by Sandisk Corporation in 1994 for the PCMCIA-ATA cards of portable PCs. Because it follows the ATA interface, it simulates a disk interface including seek commands, logical tracks, and so on. It includes a built-in controller to support many types of Flash memory and to help with chip yield for Flash memories byte mapping out bad blocks.

The electronic brain of this camera is an embedded computer with several special functions embedded on the chip (Okada [1999]). Figure 7.46 shows the block diagram of a similar chip to the one in the camera. Such chips have been *Systems On a Chip* (*SOC*), because they essentially integrate into a single chip all the parts that were found on a small printed circuit board of the past. SOC generally reduce size and lower power compared to less integrated solutions; Sanyo claims SOC enables the camera to operate on half the number of batteries and to offer a smaller form factor than competitors' cameras. For higher performance, it has two busses. The16-bit bus is for the many slower I/O devices: Smart Media interface, program and data memory, DMA. The 32-bit bus is for the SDRAM, the signal processor (which is connected to the CCD), the Motion JPEG encoder, and the NTSC/PAL encoder (which is connected to the LCD). Unlike desktop microprocessors, not the large variety of I/O buses that this chip must integrate. The 32-bit RISC MPU is a proprietary design and runs at 28.8 MHz, the same clock rate as the busses. This 700 milliWatt chip contains 1.8M transistors in a 10.5 x 10.5 mm die implemented using a 0.35 micron process.

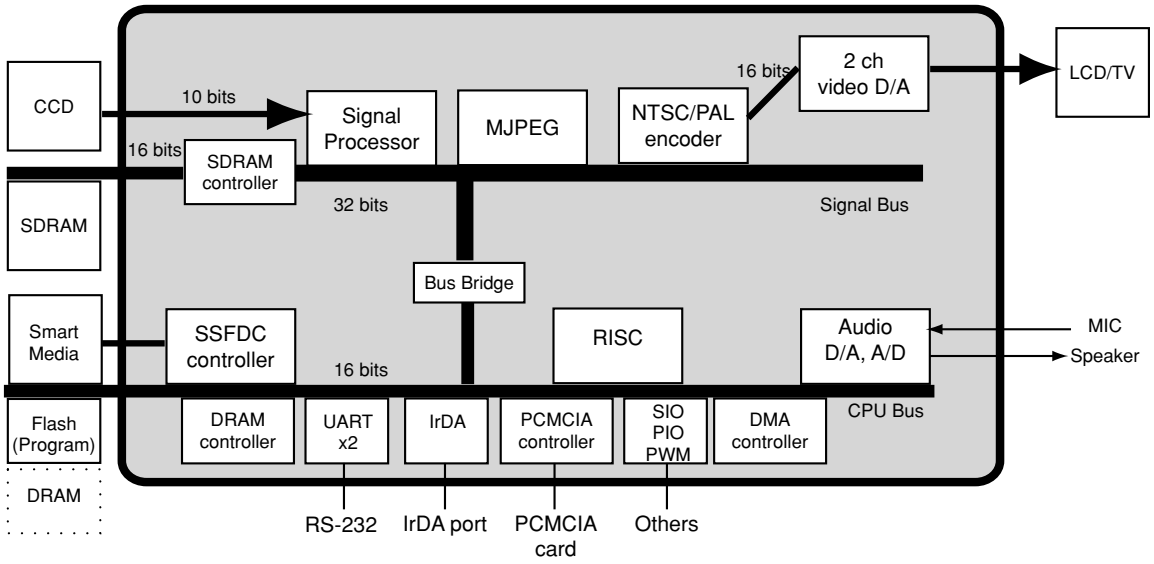


FIGURE 7.46 The system on a chip (SOC) found in Sanyo digital cameras. This block diagram, found in Okada [1999], is for the predecessor of the SOC in the camera in Figure 7.45. The successor SOC, called Super Advanced IC, uses three buses instead of two, operates at 60 MHz, consumes 800 mW, and fits 3.1M transistors in a 10.2 x 10.2 mm using a 0.35 micron process. Note that this embedded system has twice as many transistors as the state-of-the-art, high performance microprocessor in 1990! The SOC in the figure is limited to processing 1024 x 768 pixels, but its successor supports 1360 x 1024 pixels.

7.14 Fallacies and Pitfalls

Fallacy: The rated Mean Time To Failure of disks is 1,200,000 hours or almost 140 years, so disks practically never fail.

The current marketing practices of disk manufacturers can mislead users. How is such a MTTF calculated? Early in the process manufacturers will put thousands of disks in a room, run them for a few months, and count the number that fail. They compute MTTF as the total number of hours that the disks were cumulatively up divided by the number that failed.

One problem is that this number far exceeds the lifetime of a disk, which is commonly assumed to be five years or 43,800 hours. For this large MTTF to make some sense, disk manufacturers argue that the model corresponds to a user who buys a disk, and then keeps replacing the disk every five years--the planned lifetime of the disk. The claim is that if many customers (and their great-grandchildren) did this for the next century, on average they would replace a disk 27 times before a failure, or about 140 years.

A more useful measure would be percentage of disks that fail. Assume 1000 disks with a 1,000,000-hour MTTF and the disks are used 24 hours a day. If you replaced failed disks with a new one having the same reliability characteristics, the number that would fail over 5 years (43,800 hours) is:

$$\text{Failed disks} = \frac{1000 \text{ drives} \times 43800 \text{ hours/drive}}{1000000 \text{ hours/failure}} = 44$$

Stated alternatively, 4.4% would fail over the 5-year period. If they were powered on less per day, then fewer would fail, provided the number of load/unload cycles are not exceeded (see Figure 7.2 on page 490).

Fallacy: Components fail fast.

A good deal of the fault tolerant literature is based on the simplifying assumption that a component operates perfectly until a latent error becomes effective, and then a failure occurs which stops the component.

The Tertiary Disk project had the opposite experience. Many components started acting strangely long before they failed, and it was generally up to the system operator to determine whether to declare a component as failed. The component would generally be willing to continue to act in violation of the service agreement (see section 7.4) until an operator "terminated" that component.

Figure 7.47 shows the history of four drives that were terminated, and the number of hours they started acting strangely before they were replaced.

Messages in system log for failed disk	Number of log messages	Duration (hours)
Hardware Failure (Peripheral device write fault [for] Field Replaceable Unit)	1763	186
Not Ready (Diagnostic failure: ASCQ = Component ID [of] Field Replaceable Unit)	1460	90
Recovered Error (Failure Prediction Threshold Exceeded [for] Field Replaceable Unit)	1313	5
Recovered Error (Failure Prediction Threshold Exceeded [for] Field Replaceable Unit)	431	17

FIGURE 7.47 Record in system log for 4 of the 368 disks in Tertiary Disk that were replaced over 18 months. See Talagala [1999].These messages, matching the SCSI specification, were placed into the system log by device drivers. Messages started occurring as much as a week before one drive was replaced by the operator. The third and fourth message indicates that the drive's failure prediction mechanism's detected and predicted imminent failure, yet it was still hours before the drives were replaced by the operator.

Fallacy: Computers systems achieve 99.999% availability ("Five 9's), as advertised.

Marketing departments of companies making servers have started bragging about the availability of their computer hardware; in terms of Figure 7.48, they claim availability of 99.999%, nicknamed *five nines*. Even the marketing departments of operating system companies have tried to give this impression.

Five minutes of unavailability per year is certainly impressive, but given the failure data collected in surveys, it's hard to believe. For example, Hewlett Packard claims with the HP-9000 server hardware and HP-UX operating system can deliver 99.999% availability guarantee "in certain pre-defined, pre-tested customer environments" (see Hewlett Packard [1998]). This guarantee does not include failures due to operator faults, application faults, or environmental faults, which are likely the dominant fault categories today. Nor does it include scheduled down time. Its also unclear what is the financial penalty to a company if a system does not match its guarantee.

Microsoft has also promulgated a five 9's marketing campaign. In January 2001, www.microsoft.com was unavailable for 22 hours. For its web site to achieve 99.999% availability, it will require a clean slate for the next 250 years.

In contrast to marketing suggestions, well-managed servers in 2001 typically achieve 99% to 99.9% availability.

Pitfall: Where a function is implemented affects its reliability.

In theory, it is fine to move the RAID function into software. In practice, it is very difficult to make it work reliably.

The software culture is generally based on eventual correctness via a series of releases and patches. It is also difficult to isolate from other layers of software. For example, proper software behavior is often based on having the proper version and patch release of the operating system. Thus, many customers have lost data due software bugs or incompatibilities in environment in software RAID systems.

Unavailability (Minutes per year)	Availability (in percent)	Availability Class ("number of nines")
50,000	90%	1
5,000	99%	2
500	99.9%	3
50	99.99%	4
5	99.999%	5
0.5	99.9999%	6
0.05	99.99999%	7

FIGURE 7.48 Minutes unavailable per year to achieve availability class. (from Gray and Siewiorek [1991].) Note that five nines means unavailable five minutes per year.

Obviously, hardware systems are not immune to bugs, but the hardware cultures tends to have greater emphasis on testing correctness in the initial release. In addition, the hardware is more likely to be independent of the version of the operating system.

Fallacy: Semiconductor memory will soon replace magnetic disks in desktop and server computer systems.

When the first edition of this book was written, disks were growing in capacity at 29% per year and DRAMs at 60% per year. One exercise even asked when DRAMs would match the cost per bit of magnetic disks.

At about the same time, these same questions were being asked inside of disk manufacturing companies such as IBM. Therefore, disk manufacturers pushed the rate of technology improvement to match the rate of DRAMs—60% per year—with magneto-resistive heads being the first advance to accelerate disk technology. Figure 7.49 shows the relative areal density of DRAM to disk, with the gap closing in the late 1980s and widening ever since. In 2001, the gap is larger than it was in 1975. Instead of DRAMs wiping out disks, disks are wiping out tapes!

Fallacy: Since head-disk assemblies of disks are the same technology independent of the disk interface, the disk interface matters little in price.

As the high-tech portion of the disk are the heads, arms, platters, motors, and so on, it stands to reason that the I/O interface should matter little in the price of a disk. Perhaps you should pay $25 extra per drive for the more complicated SCSI interface versus the PC IDE interface. Figure 7.50 shows this reason does not stand.

There are two explanations for a factor of 2.5 difference in price per megabyte between SCSI and IDE disks. First, the PC market is much more competitive than the server market; PCs normally use IDE drives and servers normally use

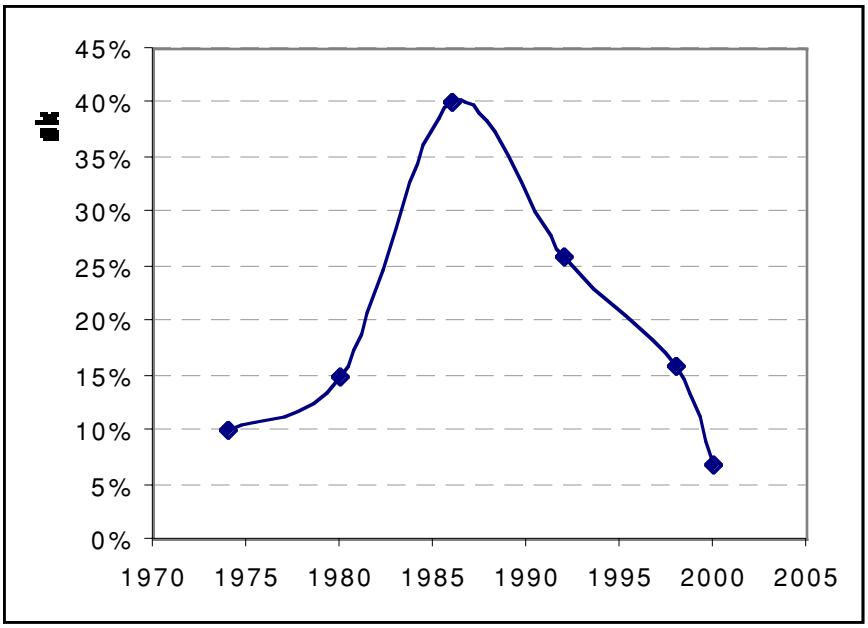


FIGURE 7.49 Areal density of DRAMs vs. Maximal areal density of magnetic disks in products, as a percentage, over time. Source: New York Times, 2/23/98, page C3, "Makers of disk drives crowd even more data into even smaller spaces. Year 2000 data added to the N.Y. Times information.

SCSI drives. Second, SCSI drives tend to be higher performance in rotation speed and seek times. To try to account for the performance differences, the second ratio line Figure 7.50 is limited to comparisons of disks with similar capacity and performance but different interfaces, yet the ratio in 2000 was still about 2.0.

A third argument for the price difference is called the *manufacturing learning curve*. The rational is that every doubling in manufacturing volume reduces costs by a significant percentage. As about 10 times as many IDE/ATA drives are sold per year as SCSI drives, if manufacturing costs dropped 20% for every doubling in volume, the learning curve effect would explain a cost factor of 1.8.

Fallacy: The time of an average seek of a disk in a computer system is the time for a seek of one-third the number of cylinders.

This fallacy comes from confusing the way manufacturers market disks with the expected performance, and from the false assumption that seek times are linear in distance. The one-third-distance rule of thumb comes from calculating the distance of a seek from one random location to another random location, not including the current cylinder and assuming there are a large number of cylinders. In the past, manufacturers listed the seek of this distance to offer a consistent basis

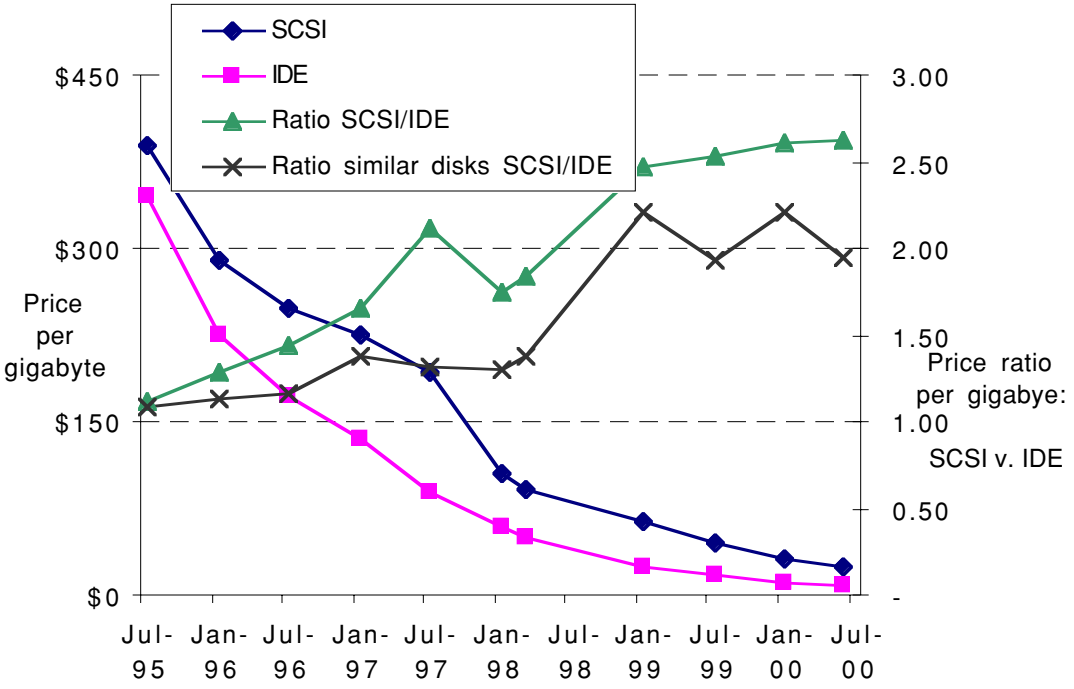


FIGURE 7.50 Price per gigabyte of 3.5-inch disks between 1995 and 2000 for IDE/ATA and SCSI drives. The data comes from the same sources as Figure 7.3 on page 493. The downward-heading lines plot price per gigabyte, and the upward-heading lines plot ratio of SCSI price to IDE price. The first upward-line is simply the ratio of the average price per gigabyte of SCSI versus IDE. The second such line is limited to comparisons of disks with the same capacity and the same RPM; it is the geometric mean of the ratios of the prices of the similar disks for each month. Note that the ratio of SCSI prices to IDE/ATA prices got larger over time, presumably because of the increasing volume of IDE versus SCSI drives and increasing competition for IDE disk suppliers.

for comparison. (As mentioned on page 489, today they calculate the "average" by timing all seeks and dividing by the number.) Assuming (incorrectly) that seek time is linear in distance, and using the manufacturer's reported minimum and "average" seek times, a common technique to predict seek time is

$$\text{Time}_{\text{seek}} = \text{Time}_{\text{minimum}} + \frac{\text{Distance}}{\text{Distance}_{\text{average}}} \times (\text{Time}_{\text{average}} - \text{Time}_{\text{minimum}})$$

The fallacy concerning seek time is twofold. First, seek time is *not* linear with distance; the arm must accelerate to overcome inertia, reach its maximum traveling speed, decelerate as it reaches the requested position, and then wait to allow the arm to stop vibrating (*settle time*). Moreover, sometimes the arm must pause to control vibrations. For disks with more than 200 cylinders, Chen and Lee [1995] modeled the seek distance as

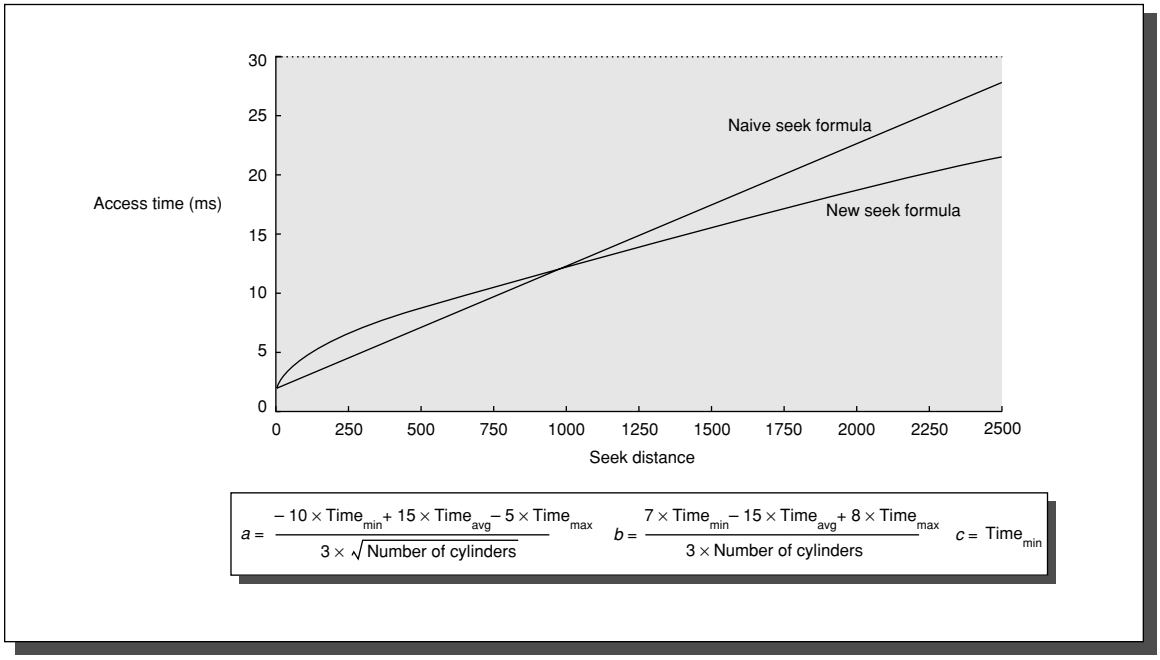


FIGURE 7.51 Seek time versus seek distance for sophisticated model versus naive model. Chen and Lee [1995] found the equations shown above for parameters a, b, and c worked well for several disks.

$$\text{Seek time}(\text{Distance}) = a \times \sqrt{\text{Distance} - 1} + b \times (\text{Distance} - 1) + c$$

where a, b, and c are selected for a particular disk so that this formula will match the quoted times for Distance = 1, Distance = max, and Distance = 1/3 max. Figure 7.51 above plots this equation versus the fallacy equation. Unlike the first equation, the square root of the distance reflects acceleration and deceleration.

The second problem is that the average in the product specification would only be true if there were no locality to disk activity. Fortunately, there is both temporal and spatial locality (see page 377 in Chapter 5): disk blocks get used more than once, and disk blocks near the current cylinder are more likely to be used than those farther away. For example, Figure 7.52 shows sample measurements of seek distances for two workloads: a UNIX timesharing workload and a business-processing workload. Notice the high percentage of disk accesses to the same cylinder, labeled distance 0 in the graphs, in both workloads.

Thus, this fallacy couldn't be more misleading. (The Exercises debunk this fallacy in more detail.)

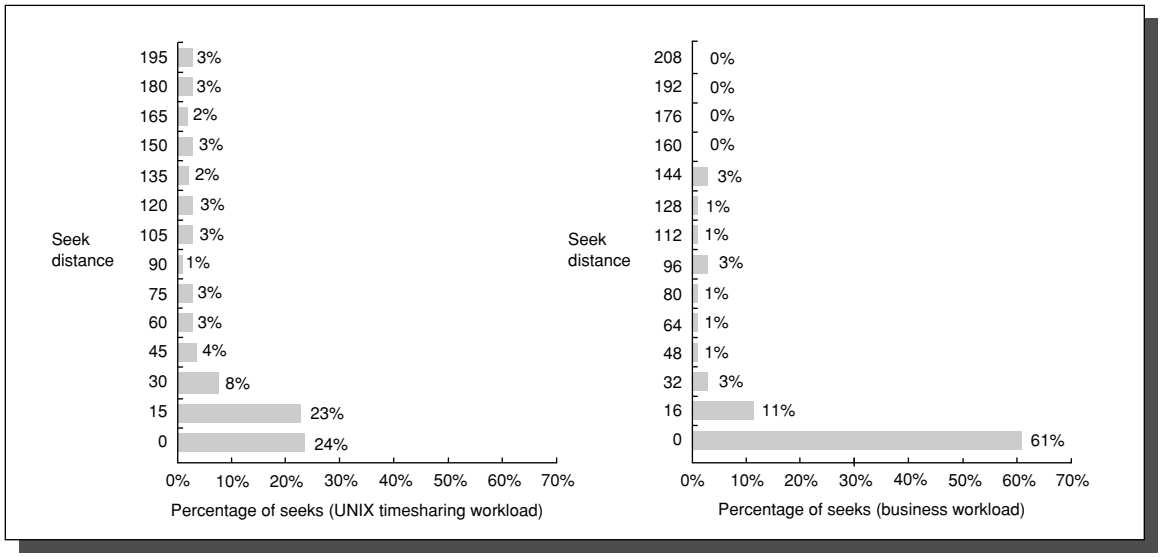


FIGURE 7.52 Sample measurements of seek distances for two systems. The measurements on the left were taken on a UNIX timesharing system. The measurements on the right were taken from a business-processing application in which the disk seek activity was scheduled to improve throughput. Seek distance of 0 means the access was made to the same cylinder. The rest of the numbers show the collective percentage for distances between numbers on the y axis. For example, 11% for the bar labeled 16 in the business graph means that the percentage of seeks between 1 and 16 cylinders was 11%. The UNIX measurements stopped at 200 of the 1000 cylinders, but this captured 85% of the accesses. The business measurements tracked all 816 cylinders of the disks. The only seek distances with 1% or greater of the seeks that are not in the graph are 224 with 4% and 304, 336, 512, and 624 each having 1%. This total is 94%, with the difference being small but nonzero distances in other categories. Measurements courtesy of Dave Anderson of Seagate.

7.15 Concluding Remarks

Storage is one of those technologies that we tend to take for granted. And yet, if we look at the true status of things today, storage is king. One can even argue that servers, which have become commodities, are now becoming peripheral to storage devices. Driving that point home are some estimates from IBM, which expects storage sales to surpass server sales in the next two years.

Michael Vizard, Editor in Chief, *Infoworld*, August 11, 2001

As their value is becoming increasingly evident, storage systems have become the target of innovation and investment.

The challenge for storage systems today is dependability and maintainability. Not only do users want to be sure their data is never lost (reliability), applications today increasingly demand that the data is always available to access (availability). Despite improvements in hardware and software reliability and fault-tolerance, the awkwardness of maintaining such systems is a problem both for cost and for availability. Challenges in storage dependability and maintainability today dominate the challenges in performance.

Disk capacity is now the fastest improving computer technology, doubling every year. Hence, despite the challenges of dependability and maintainability, new storage applications arrive, such as digital cameras and digital libraries.

Today we are just a few keystrokes away from much of humankind's knowledge. Just this application has changed your life: How often do you search the world wide web versus go to the library?

Getting those requests to digital repositories and getting the answer back is the challenge of networks, the topic of the next chapter. In addition to explaining the Internet, the next chapter also gives the anatomy of a WWW search engine, showing how a network of thousands of desktop computers can provide a valuable and reliable service.

7.16 Historical Perspective and References

Mass storage is a term used there to imply a unit capacity in excess of one million alphanumeric characters…

Hoagland [1963]

The variety of storage I/O and issues leads to a varied history for the rest of the story. (Smotherman [1989] explores the history of I/O in more depth.) This section discusses magnetic storage, RAID, and I/O buses and controllers. Jain [1991] and Lazowska et al [1984] offer books for those interested in learning more about queuing theory.

Magnetic Storage

Magnetic recording was invented to record sound, and by 1941, magnetic tape was able to compete with other storage devices. It was the success of the ENIAC in 1947 that led to the push to use tapes to record digital information. Reels of magnetic tapes dominated removable storage through the 1970s. In the 1980s, the IBM 3480 cartridge became the de facto standard, at least for mainframes. It can transfer at 3 MB/sec by reading 18 tracks in parallel. The capacity is just 200 MB for this 1/2-inch tape. The 9840 cartridge, used by StorageTek in the Powderhorn, transfers at 10 MB/sec and stores 20,000 MB. This device records the tracks in a zigzag fashion rather than just longitudinally, so that the head reverses direction

to follow the track. This technique is called *serpentine recording*. Another 1/2-inch tape is Digital Linear Tape, with DLT7000 storing 35,000 MB and transferring at 5 MB/sec. Its competitor is helical scan, which rotates the head to get the increased recording density. In 2001, the 8-mm helical-scan tapes contain 20000 MB and transfer at about 3 MB/second. Whatever their density and cost, the serial nature of tapes creates an appetite for storage devices with random access.

In 1953, Reynold B. Johnson of IBM picked a staff of 15 scientists with the goal of building a radically faster random access storage system than tape. The goal was to have the storage equivalent of 50,000 standard IBM punch cards and to fetch the data in a single second. Johnson's disk drive design was simple but untried: the magnetic read/write sensors would have to float a few thousandths of an inch above the continuously rotating disk. Twenty-four months later the team emerged with the functional prototype. It weighed one ton, and occupied about 300 cubic feet of space. The RAMAC-350 (Random Access Method of Accounting Control) used 50 platters that were 24 inches in diameter, rotated at 1200 RPM, with a total capacity of 5 MB and an access time of 1 second.

Starting with the RAMAC, IBM maintained its leadership in the disk industry, with its storage headquarters in San Jose, California where Johnson's team did its work. Many of the future leaders of competing disk manufacturers started their careers at IBM, and many disk companies are located near San Jose.

Although RAMAC contained the first disk, a major breakthrough in magnetic recording was found in later disks with air-bearing read-write heads, where the head would ride on a cushion of air created by the fast-moving disk surface. This cushion meant the head could both follow imperfections in the surface and yet be very close to the surface. Subsequent advances have come largely from improved quality of components and higher precision. In 2001, heads fly 2 to 3 microinches above the surface, whereas in the RAMAC drive was 1000 microinches away.

Moving-head disks quickly became the dominant high-speed magnetic storage, though their high cost meant that magnetic tape continued to be used extensively until the 1970s. The next important development for hard disks was the removable hard disk drive developed by IBM in 1962; this made it possible to share the expensive drive electronics and helped disks overtake tapes as the preferred storage medium. The IBM 1311 disk in 1962 had an areal density of 50,000 bits per square inch and a cost of about $800 per megabyte.

IBM also invented the floppy disk drive in 1970, originally to hold microcode for the IBM 370 series. Floppy disks became popular with the PC about 10 years later.

The second major disk breakthrough was the so-called Winchester disk design in about 1973. Winchester disks benefited from two related properties. First, integrated circuits lowered the costs of not only CPUs, but also of disk controllers and the electronics to control disk arms. Reductions in the cost of the disk electronics made it unnecessary to share the electronics, and thus made nonremovable disks economical. Since the disk was fixed and could be in a sealed enclosure, both the environmental and control problems were greatly reduced. Sealing the system al-

lowed the heads to fly closer to the surface, which in turn enables increases in areal density. The first sealed disk that IBM shipped had two spindles, each with a 30-MB disk; the moniker "30-30" for the disk led to the name Winchester. (America's most popular sporting rifle, the Winchester 94 was nicknamed the "30-30" after the caliber of its cartridge.) Winchester disks grew rapidly in popularity in the 1980s, completely replacing removable disks by the middle of that decade. Before this time, the cost of the electronics to control the disk meant that the media had to be removable.

In 2001, IBM sold disks with 25 billion bits per square inch at a price of about $0.01 per megabyte. (See Hospodor and Hoagland [1993] for more on magnetic storage trends.) The disk industry today is responsible for 90% of the mass storage market.

As mentioned in the section 7.14, as DRAMs started to close the areal density gap and appeared to be catching up with disk storage, internal meetings at IBM called into question the future of disk drives. Disk designers concluded that disks must improve at 60% per year to forestall the DRAM threat, in contrast to the historical 29% per year. The essential enabler was magneto-resistive heads, with giant magneto-resistive heads enabling the current densities.

Because of this competition, the gap in time between when a density record is achieved in the lab and when a disk is shipped with that density has closed considerably. In 2001, the lab record is 60 Gbits/square inch, but drives are shipping with a third of that density. It is also unclear to disk engineers whether evolutionary change will achieve 1000 Gbits/square inch.

The personal computer created a market for small form-factor disk drives, since the 14-inch disk drives used in mainframes were bigger than the PC. In 2001, the 3.5-inch drive is the market leader, although the smaller 2.5-inch drive needed for laptop computers is significant in sales volume. Personal video recorders–which record television on disk instead of tape–may become a significant consumer of disk drives. Existing form factors and speed are sufficient, with the focus on low noise and high capacity for PVRs. Hence, a market for large, slow, quiet disks may develop. It remains to be seen whether hand-held devices or video cameras, requiring even smaller disks, will become as significant in sales volume as PCs or laptops. For example, 1.8-inch drives were developed in the early 1990s for palmtop computers, but that market chose Flash instead, and hence 1.8-inch drives disappeared.

RAID

The small form factor hard disks for PCs in the 1980s led a group at Berkeley to propose Redundant Arrays of Inexpensive Disks, or RAID. This group had worked on the Reduced Instruction Set Computers effort, and so expected much faster CPUs to become available. Their questions were what could be done with the small disks that accompanied their PCs, and what could be done in the area of I/O to keep up with much faster processors. They argued to replace one main-

frame drive with 50 small drives, as you could get much greater performance with that many independent arms. The many small drives even offered savings in power consumption and floor space.

The downside of many disks was much lower MTTF. Hence, on their own they reasoned out the advantages of redundant disks and rotating parity to addresses how to get greater performance with many small drives yet have reliability as high as that of a single mainframe disk.

The problem they experienced when explaining their ideas was that some researchers had heard of disk arrays with some form of redundancy, and they didn't understand the Berkeley proposal. Hence, the first RAID paper (Patterson, Gibson, Katz [1987]) is not only a case for arrays of small form factor disk drives, but something of a tutorial and classification of existing work on disk arrays. Mirroring (RAID 1) had long been used in fault tolerant computers such as those sold by Tandem; Thinking Machines had a arrays with 32 data disks and 7 check disks using ECC for correction (RAID 2) in 1987, and Honeywell Bull had a RAID 2 product even earlier; and disk arrays with a single parity disk had been used in scientific computers in the same time frame (RAID 3). Their paper then described single parity disk with support for sector accesses (RAID 4) and rotated parity (RAID 5). Chen et al. [1994] survey the original RAID ideas, commercial products, and more recent developments.

Unknown to the Berkeley group, engineers at IBM working on the AS/400 computer also came up with rotated parity to give greater reliability for a collection of large disks. IBM filed a patent on RAID 5 before the Berkeley group wrote their paper. Patents for RAID 1, RAID 2, RAID 3 from several companies predate the IBM RAID 5 patent, which has led to plenty of courtroom action.

The Berkeley paper written was before the World Wide Web, but it captured the imagination of many engineers, as copies were faxed around the world. One engineer at what is now Seagate received seven copies of the paper from friends and customers.

EMC had been a supplier of DRAM boards for IBM computers, but around 1988 new policies from IBM made it nearly impossible for EMC to continue to sell IBM memory boards. Apparently, the Berkeley paper also crossed the desks of EMC executives, and so they decided to go after the market dominated by IBM disk storage products instead. As the paper advocated, their model was to use many small drives to compete with mainframe drives, and EMC announced a RAID product in 1990. It relied on mirroring (RAID 1) for reliability; RAID-5 products came much later for EMC. Over the next year, Micropolis offered a RAID-3 product, Compaq offered a RAID-4 product, and Data General, IBM, and NCR offered RAID-5 products.

The RAID ideas soon spread to the rest of workstation and server industry. An article explaining RAID in Byte magazine (see Anderson 1990) lead to RAID products being offered on desktop PCs, which was something of a surprise to the Berkeley group. They had focused on performance with good availability, but higher availability was attractive to the PC market.

Another surprise was the cost of the disk arrays. With redundant power supplies and fans, the ability to "hot swap" a disk drive, the RAID hardware controller itself, the redundant disks, and so on, the first disk arrays cost many times the cost of the disks. Perhaps as a result, the Inexpensive in RAID morphed into Independent. Many marketing departments and technical writers today know of RAID only as Redundant Arrays of Independent Disks.

The EMC transformation was successful; in 2000 EMC was the leading supplier of storage systems. RAID was a $27B industry in 2000, and more than 80% of the nonPC drives sales were found in RAIDs.

In recognition of their role, in 1999 Garth Gibson, Randy Katz, and David Patterson received the IEEE Reynold B. Johnson Information Storage Award "for the development of Redundant Arrays of Inexpensive Disks (RAID)."

I/O Buses and Controllers

The ubiquitous microprocessor has inspired not only the personal computers of the 1970s, but also the trend in the late 1980s and 1990s of moving controller functions into I/O devices. I/O devices continued this trend by moving controllers into the devices themselves. These devices are called *intelligent devices*, and some bus standards (e.g., SCSI) have been created specifically for them. Intelligent devices can relax the timing constraints by handling many low-level tasks themselves and queuing the results. For example, many SCSI-compatible disk drives include a track buffer on the disk itself, supporting read ahead and connect/disconnect. Thus, on a SCSI string some disks can be seeking and others loading their track buffer while one is transferring data from its buffer over the SCSI bus. The controller in the original RAMAC, built from vacuum tubes, only needed to move the head over the desired track, wait for the data to pass under the head, and transfer data with calculated parity.

SCSI, which stands for *small computer systems interface*, is an example of one company inventing a bus and generously encouraging other companies to build devices that would plug into it. Shugart created this bus, originally called SASI. It was later standardized by the IEEE.

There have been several candidates to be the successor to SCSI, with the current leading contender being Fibre Channel Arbitrated Loop (FC-AL). The SCSI committee continues to increase the clock rate of the bus, giving this standard a new life, and SCSI is lasting much longer than some of its proposed successors.

Perhaps the first multivendor bus was the PDP-11 Unibus in 1970 from DEC. Alas, this open-door policy on buses is in contrast to companies with proprietary buses using patented interfaces, thereby preventing competition from plug-compatible vendors. Making a bus proprietary also raises costs and lowers the number of available of I/O devices that plug into proprietary buses, since such devices must have an interface designed just for that bus. The PCI bus pushed by Intel represented a return to open, standard I/O buses inside computers. Its immediate successor is PCI-X, with Infiniband under development in 2000, with both standardized by multi-company trade associations.

The machines of the RAMAC era gave us I/O interrupts as well as storage devices. The first machine to extend interrupts from detecting arithmetic abnormalities to detecting asynchronous I/O events is credited as the NBS DYSEAC in 1954 [Leiner and Alexander 1954]. The following year, the first machine with DMA was operational, the IBM SAGE. Just as today's DMA has, the SAGE had address counters that performed block transfers in parallel with CPU operations.

The early IBM 360s pioneered many of the ideas that we use in I/O systems today. The 360 was the first commercial machine to make heavy use of DMA, and it introduced the notion of I/O programs that could be interpreted by the device. Chaining of I/O programs was an important feature. The concept of channels introduced in the 360 corresponds to the I/O bus of today.

Myer and Sutherland [1968] wrote a classic paper on the trade-off of complexity and performance in I/O controllers. Borrowing the religious concept of the "Wheel of Reincarnation," they eventually noticed they were caught in a loop of continuously increasing the power of an I/O processor until it needed its own simpler coprocessor. The quote on page 508 captures their cautionary tale.

The IBM mainframe I/O channels, with their I/O processors, can be thought of as an inspiration for Infiniband, with their processors on their Host Channel Adaptor cards. How Infiniband will compete with FC-AL as an I/O interconnect will be interesting to watch. Infiniband is one of the Storage Area Networks discussed in the next chapter.

References

ANDERSON, M.H. [1990] "STRENGTH (AND SAFETY) IN NUMBERS (RAID, DISK STORAGE TECHNOLOGY)," *BYTE*, VOL.15, (NO.13), DEC. P.337-9.

ANON, ET AL. [1985]. "A measure of transaction processing power," Tandem Tech. Rep. TR 85.2. Also appeared in *Datamation*, April 1, 1985.

BASHE, C. J., W. BUCHHOLZ, G. V. HAWKINS, J. L. INGRAM, AND N. ROCHESTER [1981]. "The architecture of IBM's early computers," *IBM J. Research and Development* 25:5 (September), 363–375.

BASHE, C. J., L. R. JOHNSON, J. H. PALMER, AND E. W. PUGH [1986]. *IBM's Early Computers*, MIT Press, Cambridge, Mass.

BRADY, J. T. [1986]. "A theory of productivity in the creative process," *IEEE CG&A* (May), 25–34.

Brown, A. and D.A. Patterson [2000]. "Towards Maintainability, Availability, and Growth Benchmarks: A Case Study of Software RAID Systems." *Proceedings of the 2000 USENIX Annual Technical Conference*, San Diego, CA,(June).

BUCHER, I. V. AND A. H. HAYES [1980]. "I/O performance measurement on Cray-1 and CDC 7000 computers," *Proc. Computer Performance Evaluation Users Group, 16th Meeting*, NBS 500-65, 245–254.

CHEN, P. M., G. A. GIBSON, R. H. KATZ, AND D. A. PATTERSON [1990]. "An evaluation of redundant arrays of inexpensive disks using an Amdahl 5890," *Proc. 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (May), Boulder, Colo.

CHEN, P. M., E. K. LEE, G. A. GIBSON, R. H. KATZ, AND D. A. PATTERSON [1994]. "RAID: High-performance, reliable secondary storage," *ACM Computing Surveys* 26:2 (June), 145–88.

CHEN, P. M. AND E. K. LEE [1995]. "Striping in a RAID level 5 disk array," *Proc. 1995 ACM SIG-*

METRICS Conference on Measurement and Modeling of Computer Systems (May), 136–145.

DOHERTY, W. J. AND R. P. KELISKY [1979]. "Managing VM/CMS systems for user effectiveness," *IBM Systems J.* 18:1, 143–166.

Enriquez, P. [2001] "What Happened to my Dial Tone? A study of FCC service disruption reports, " poster, *Richard Tapia Symposium on the Celebration of Diversity in Computing*, October 18-20, Houston, Texas.

FRIESENBORG, S. E. AND R. J. WICKS [1985]. "DASD expectations: The 3380, 3380-23, and MVS/XA," Tech. Bulletin GG22-9363-02 (July 10), Washington Systems Center.

Gibson, G. A. [1992] *Redundant disk arrays: reliable, parallel secondary storage,* ACM Distinguished Dissertation Series, MIT Press, Cambridge, Mass.

GOLDSTEIN, S. [1987]. "Storage performance—An eight year outlook," Tech. Rep. TR 03.308-1 (October), Santa Teresa Laboratory, IBM, San Jose, Calif.

GRAY, J. (ED.) [1993]. *The Benchmark Handbook for Database and Transaction Processing Systems,* 2nd ed. Morgan Kaufmann Publishers, San Francisco.

GRAY, J. AND A. REUTER [1993]. *Transaction Processing: Concepts and Techniques,* Morgan Kaufmann Publishers, San Francisco.

GRAY, J. AND D.P. SIEWIOREK, [1991] "High-availability computer systems." *Computer*, 24:9, (Sept), 39-48.

GRAY, J. [1990]. "A census of Tandem system availability between 1985 and 1990." *IEEE Transactions on Reliability*, vol.39, (no.4), (Oct.) 409-18.

HENLY, M. AND B. MCNUTT [1989]. "DASD I/O characteristics: A comparison of MVS to VM," Tech. Rep. TR 02.1550 (May), IBM, General Products Division, San Jose, Calif.

HOAGLAND, A. S. [1963]. *Digital Magnetic Recording*, Wiley, New York.

HEWLETT PACKARD [1998] . HP's "5NINES:5MINUTES" Vision Extends Leadership and Re-Defines High Availability in Mission-Critical Environments, (Feb 10), see http://www.future.enterprisecomputing.hp.com/ia64/news/5nines_vision_pr.html

HOSPODOR, A. D. AND A. S. HOAGLAND [1993]. "The changing nature of disk controllers." *Proc. IEEE* 81:4 (April), 586–94.

IBM [1982]. *The Economic Value of Rapid Response Time*, GE20-0752-0, White Plains, N.Y., 11–82.

IMPRIMIS [1989]. *Imprimis Product Specification, 97209 Sabre Disk Drive IPI-2 Interface 1.2 GB,* Document No. 64402302 (May).

JAIN, R. [1991]. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*, Wiley, New York.

KATZ, R. H., D. A. PATTERSON, AND G. A. GIBSON [1990]. "Disk system architectures for high performance computing," *Proc. IEEE* 78:2 (February).

KIM, M. Y. [1986]. "Synchronized disk interleaving," *IEEE Trans. on Computers* C-35:11 (November).

Kuhn, D. R. [1997]. "Sources of Failure in the Public Switched Telephone Network." I*EEE Computer* 30:4 (April).

Lambright, D [2000]. "Experiences in Measuring the Reliability of a Cache-Based Storage System," *Proceedings of First Workshop on Industrial Experiences with Systems Software* (WIESS 2000), collocated with the 4th Symposium on Operating Systems Design and Implementation (OSDI), San Diego, California. (October 22).

LAPRIE, J.-C. [1985] "Dependable computing and fault tolerance: concepts and terminology." *Fifteenth Annual International Symposium on Fault-Tolerant Computing FTCS 15*. Digest of Papers. Ann Arbor, MI, USA, (19-21 June) 2-11.

Lazowska, E.D., J. Zahorjan, G. S. Graham, and K. C. Sevcik [1984]. *Quantitative system performance : computer system analysis using queueing network models*, Prentice-Hall, Englewood Cliffs, N.J. (Although out of print, it is available online at www.cs.washington.edu/homes/lazowska/qsp/)

LEINER, A. L. [1954]. "System specifications for the DYSEAC," *J. ACM* 1:2 (April), 57–81.

LEINER, A. L. AND S. N. ALEXANDER [1954]. "System organization of the DYSEAC," *IRE Trans. of Electronic Computers* EC-3:1 (March), 1–10.

MABERLY, N. C. [1966]. *Mastering Speed Reading,* New American Library, New York.

MAJOR, J. B. [1989]. "Are queuing models within the grasp of the unwashed?," *Proc. Int'l Conference on Management and Performance Evaluation of Computer Systems,* Reno, Nev. (December 11-15), 831–839.

Mueller, M.; Alves, L.C.; Fischer, W.; Fair, M.L.; Modi, I. [1999] "RAS strategy for IBM S/390 G5 and G6," *IBM Journal of Research and Developmen*t, 43:5-6 (Sept.-Nov), 875-88.

Myer, T. H. and I. E. Sutherland [1968]. "On the Design of Display Processors," *Communications of the ACM*, 11:6 (June), 410-414.

National Storage Industry Consortium [1998], *Tape Roadmap*, (June), see www.nsic.org.

Nelson, V.P. [1990]"Fault-tolerant computing: fundamental concepts." *Computer*, vol.23, (no.7), (July). p.19-25.

Okada, S.; Okada, S.; Matsuda, Y.; Yamada, T.; Kobayashi, [1999] "A. System on a chip for digital still camera," *IEEE Transactions on Consumer Electronics*, 45:.3, (Aug.) 584-90.

PATTERSON, D. A., G. A. GIBSON, AND R. H. KATZ [1987]. "A case for redundant arrays of inexpensive disks (RAID)," Tech. Rep. UCB/CSD 87/391, Univ. of Calif. Also appeared in *ACM SIGMOD Conf. Proc.*, Chicago, June 1–3, 1988, 109–116.

PAVAN, P., R. BEZ, P.,OLIVO, E. ZANONI [1997] "Flash memory cells-an overview." *Proceedings of the IEEE*, vol.85, (no.8),(Aug.)p.1248-71.

ROBINSON, B. AND L. BLOUNT [1986]. "The VM/HPO 3880-23 performance results," IBM Tech. Bulletin GG66-0247-00 (April), Washington Systems Center, Gaithersburg, Md.

SALEM, K. AND H. GARCIA-MOLINA [1986]. "Disk striping," *IEEE 1986 Int'l Conf. on Data Engineering*.

SCRANTON, R. A., D. A. THOMPSON, AND D. W. HUNTER [1983]. "The access time myth," Tech. Rep. RC 10197 (45223) (September 21), IBM, Yorktown Heights, N.Y.

SEAGATE [2000] *Seagate Cheetah 73 Family: ST173404LW/LWV/LC/LCV Product Manua*l, Volume 1, see http://www.seagate.com/support/disc/manuals/scsi/29478b.pdf.

SMOTHERMAN, M. [1989]. "A sequencing-based taxonomy of I/O systems and review of historical machines," *Computer Architecture News* 17:5 (September), 5–15. Reprinted in *Computer Architecture Readings*, Morgan Kauffman, 1999, 451-461.

Talagala, N., S. Asami, D. Patterson, R. Futernick, and D. Hart, [2000]" The Art of Massive Storage: A Case Study of a Web Image Archive", *Computer*, (November).

Talagala, N. and D. Patterson. [1999] "An Analysis of Error Behavior in a Large Storage System". Technical Report UCB//CSD-99-1042, Computer Science Division, University of California at Berkeley. (February).

THADHANI, A. J. [1981]. "Interactive user productivity," *IBM Systems J*. 20:4, 407–423.

E X E R C I S E S

- One of my students, Nisha Talagala, finished her PhD and as a self-indentifying benchmark to automatically classify disks. I think this would be a GREAT exercise, similar to Exercises 5.2-5.3 in the cache chapter. The idea is that students could run it and learn about there own disks. They don't need to modify data, but they may need to be super user on their PC/workstation. Interest project might be to port it to NT or windows. Like the cache example, there could be slides analyzing a figure in the exercises as well as running the program on their own disks. See http://www.cs.berkeley.edu/~nisha/bench.html. Perhaps not one of the first exercises, but should be included later. There is related work by Schindler and Ganger in Sigmetrics 2000.
- The analysis of our I/O system design performance did not include queuing theory for the full array, just one disk. Do the same analysis for close to 100% utilization using M/M/m queue. Then do it for the same analysis when the system follows the rules of thumb.

The new sections on reliability calculations and RAID examples suggests a new set of exercises. Here are a few:

- A simple example is calculating performance and cost performance of the fifth example when the workload is not 100% reads. Examples include 100% writes, and what is the highest percentage of writes that keeps cost-performance is within, say, 1.2 times that of the nonredundant solution in example four?
- One issue that was ignored by the example is the performance of the RAID system when a drive has failed. Assume that the non faulting workload would keep the system 50% utilized. First assume 100% reads. What is the % of the nonfaulting performance available assuming a single disk has failed? What is it if an enclosure fails? Another exercise can redo the example assuming 80% reads and 20% writes.
- One way to improve reliability is to reduce MTTR. If we must wait for a human to notice the failure, then its hard to make much an improvement. Having standby spares in place can significantly reduce MTTR. Redo the calculations assuming that you have a spare enclosure of disks that can be put to work on a failiure.How long does it take to recover on a failure with a standby spare? How does this affect MTDL? How does it affect cost per I/O? How does parity group size affect MTTR?
- Redo the example with RAID, this time adding 1 redundant power supply and 1 redundant fan per enclosure. How much does this improve the MTTF of the enclosure? How much does it improve MTDL of the RAID?
- The example assumed the RPM, seek time, MTTF, bandwidth, and cost per GB was the same for the large disk drive and the small disk drive. Go to a web site

and find the best cost-performance 3.5 inch drive and the best cost-performance 2.5 inch drive. Assume in a single enclosure you can pack 8 3.5 inch "half height" drives (1.7 inches high), 12 3.5 inch "low profile" (1.0 inches high), or 36 2.5 inch drives. Assume that all have a SCSI interface so that you can connect up to 15 drives on a string. Design the RAID organization and calculate the cost-performance and reliability as in the example on page 563. Use parameters from that example if you cannot find more recent information form web sites.

- Good idea to talk about the reliability terminology: maybe some examples, and ask what they are: fault, error, failure? When would things fail?
- A discussion topic is the so called "superparamagnetic limit" of disks. What are the issues, do people believe its a real limit, what would be the impact if it were? see http://www.research.ibm.com/journal/rd/443/thompson.html.
- Another discussion topic is what services people rely on magnetic tapes: backup, media distribution, ... How would systems have (or user's expectations) have to change to do backup without tapes? To build systems that didn't need backup?
- Another discussion is the technology direction of disks: what is happening to the relative rates of seek time, transfer rate, RPM, and capacity. Perhaps can include disks for 1st and 2nd edition of book to give historical perspective. Based on these disks, calculate the trends. Be sure to include time it takes to read a full disk sequentially over the years, and the time it takes to do random 32KB seeks over years. What are the impacts of these trends? What opportunities will arise? What problems of these trends for systems designers?
- This one would be a research topic exercises. A more sophisticated analysis of RAID failures relies on Markov models of faults; see Gibson [1992]. Learn about Markov models and redo the simplified failure analysis of the disk array.

7.1 [10] <7.14> Using the formulas in the fallacy starting on page 578, including the caption of Figure 7.51 (page 580), calculate the seek time for moving the arm over one-third of the cylinders of the disk in Figure 7.2 (page 490).

7.2 [25] <7.14> Using the formulas in the fallacy starting on page 578, including the caption of Figure 7.51 (page 580), write a short program to calculate the "average" seek time by estimating the time for all possible seeks using these formulas and then dividing by the number of seeks. How close is the answer to Exercise 7.1 to this answer?

7.3 [20] <7.14> Using the formulas in the fallacy starting on page 578, including the caption of Figure 7.51 (page 580) and the statistics in Figure 7.52 (page 581), calculate the average seek distance on the disk in Figure 7.2 (page 490). Use the midpoint of a range as the seek distance. For example, use 98 as the seek distance for the entry representing 91–105 in Figure 7.52. For the business workload, just ignore the missing 5% of the seeks. For the UNIX workload, assume the missing 15% of the seeks have an average distance of 300 cylinders. If you were misled by the fallacy, you might calculate the average distance as

884/3. What is the measured distance for each workload?

7.4 [20] <7.14> Figure 7.2 (page 490) gives the manufacturer's average seek time. Using the formulas in the fallacy starting on page 578, including the equations in Figure 7.51 (page 580), and the statistics in Figure 7.52 (page 581), what is the average seek time for each workload on the disk in Figure 7.2 using the measurements? Make the same assumptions as in Exercise 7.3.

- The following example needs to be updated: faster computer, bigger disks, cheaper per MB disks

7.5 [20/15/15/15/15/15] <7.7> The I/O bus and memory system of a computer are capable of sustaining 1000 MB/sec without interfering with the performance of an 800-MIPS CPU (costing $50,000). Here are the assumptions about the software:

- Each transaction requires 2 disk reads plus 2 disk writes.
- The operating system uses 15,000 instructions for each disk read or write.
- The database software executes 40,000 instructions to process a transaction.
- The transfer size is 100 bytes.

You have a choice of two different types of disks:

- A small disk that stores 500 MB and costs $100.
- A big disk that stores 1250 MB and costs $250.

Either disk in the system can support on average 30 disk reads or writes per second.

Answer parts (a)–(f) using the TPS benchmark in section 7.7. Assume that the requests are spread evenly to all the disks, that there is no waiting time due to busy disks, and that the account file must be large enough to handle 1000 TPS according to the benchmark ground rules.

a. [20] <7.7> How many TPS transactions per second are possible with each disk organization, assuming that each uses the minimum number of disks to hold the account file?

b. [15] <7.7> What is the system cost per transaction per second of each alternative for TPS?

c. [15] <7.7> How fast does a CPU need to be to make the 1000 MB/sec I/O bus a bottleneck for TPS? (Assume that you can continue to add disks.)

d. [15] <7.7> As manager of MTP (Mega TP), you are deciding whether to spend your development money building a faster CPU or improving the performance of the software. The database group says they can reduce a transaction to 1 disk read and 1 disk write and cut the database instructions per transaction to 30,000. The hardware group can build a faster CPU that sells for the same amount as the slower CPU with the same development budget. (Assume you can add as many disks as needed to get higher performance.) How much faster does the CPU have to be to match the performance gain of the software improvement?

e. [15] <7.7> The MTP I/O group was listening at the door during the software presentation. They argue that advancing technology will allow CPUs to get faster without significant investment, but that the cost of the system will be dominated by disks if they don't develop new small, faster disks. Assume the next CPU is 100% faster at the same cost and that the new disks have the same capacity as the old ones. Given the new CPU and the old software, what will be the cost of a system with enough old small disks so that they do not limit the TPS of the system?

f. [15] <7.7> Start with the same assumptions as in part (e). Now assume that you have as many new disks as you had old small disks in the original design. How fast must the new disks be (I/Os per second) to achieve the same TPS rate with the new CPU as the system in part (e)? What will the system cost?

n Next one needs to be updated to newer disk parameters

7.6 [20] <7.7> Assume that we have the following two magnetic-disk configurations: a single disk and an array of four disks. Each disk has 20 surfaces, 885 tracks per surface, and 16 sectors/track. Each sector holds 1K bytes, and it revolves at 7200 RPM. Use the seek-time formula in the fallacy starting on page 578, including the equations in Figure 7.51 (page 580). The time to switch between surfaces is the same as to move the arm one track. In the disk array all the spindles are synchronized—sector 0 in every disk rotates under the head at the exact same time—and the arms on all four disks are always over the same track. The data is "striped" across all four disks, so four consecutive sectors on a single-disk system will be spread one sector per disk in the array. The delay of the disk controller is 2 ms per transaction, either for a single disk or for the array. Assume the performance of the I/O system is limited only by the disks and that there is a path to each disk in the array. Calculate the performance in both I/Os per second and megabytes per second of these two disk organizations, assuming the request pattern is random reads of 4 KB of sequential sectors. Assume the 4 KB are aligned under the same arm on each disk in the array.

7.7 [20]<7.7> Start with the same assumptions as in Exercise 7.5 (e). Now calculate the performance in both I/Os per second and megabytes per second of these two disk organizations assuming the request pattern is reads of 4 KB of sequential sectors where the average seek distance is 10 tracks. Assume the 4 KB are aligned under the same arm on each disk in the array.

7.8 [20] <7.7> Start with the same assumptions as in Exercise 7.5 (e). Now calculate the performance in both I/Os per second and megabytes per second of these two disk organizations assuming the request pattern is random reads of 1 MB of sequential sectors. (If it matters, assume the disk controller allows the sectors to arrive in any order.)

7.9 [20] <7.2> Assume that we have one disk defined as in Exercise 7.5 (e). Assume that we read the next sector after any read and that *all* read requests are one sector in length. We store the extra sectors that were read ahead in a disk cache. Assume that the probability of receiving a request for the sector we read ahead at some time in the future (before it must be discarded because the disk-cache buffer fills) is 0.1. Assume that we must still pay the controller overhead on a disk-cache read hit, and the transfer time for the disk cache is 250 ns per word. Is the read-ahead strategy faster? (*Hint*: Solve the problem in the steady state by assuming that the disk cache contains the appropriate information and a request has just missed.)

- I'd try updating this, possibly using a second level cache

7.10 [20/10/20/20] <7.7–7.10> Assume the following information about a MIPS machine:

- Loads 2 cycles.
- Stores 2 cycles.
- All other instructions are 1 cycle.

Use the summary instruction mix information on MIPS for gcc from Chapter 2.

Here are the cache statistics for a write-through cache:

- Each cache block is four words, and the whole block is read on any miss.
- Cache miss takes 23 cycles.
- Write through takes 16 cycles to complete, and there is no write buffer.

Here are the cache statistics for a write-back cache:

- Each cache block is four words, and the whole block is read on any miss.
- Cache miss takes 23 cycles for a clean block and 31 cycles for a dirty block.
- Assume that on a miss, 30% of the time the block is dirty.

Assume that the bus

- Is only busy during transfers
- Transfers on average 1 word / clock cycle
- Must read or write a single word at a time (it is not faster to access two at once)

a. [20] <7.7–7.10> Assume that DMA I/O can take place simultaneously with CPU cache hits. Also assume that the operating system can guarantee that there will be no stale-data problem in the cache due to I/O. The sector size is 1 KB. Assume the cache miss rate is 5%. On the average, what percentage of the bus is used for each cache write policy? (This measured is called the *traffic ratio* in cache studies.)

b. [10] <7.7–7.10> Start with the same assumptions as in part (a). If the bus can be loaded up to 80% of capacity without suffering severe performance penalties, how much memory bandwidth is available for I/O for each cache write policy? The cache miss rate is still 5%.

c. [20] <7.7–7.10> Start with the same assumptions as in part (a). Assume that a disk sector read takes 1000 clock cycles to initiate a read, 100,000 clock cycles to find the data on the disk, and 1000 clock cycles for the DMA to transfer the data to memory. How many disk reads can occur per million instructions executed for each write policy? How does this change if the cache miss rate is cut in half?

d. [20] <7.7–7.10> Start with the same assumptions as in part (c). Now you can have any number of disks. Assuming ideal scheduling of disk accesses, what is the maximum number of sector reads that can occur per million instructions executed?

7.11 [50] < 7.7> Take your favorite computer and write a program that achieves maximum

bandwidth to and from disks. What is the percentage of the bandwidth that you achieve compared with what the I/O device manufacturer claims?

7.12 [20] <7.2,7.4> Search the World Wide Web to find descriptions of recent magnetic disks of different diameters. Be sure to include at least the information in Figure 7.2 on page 490.

7.13 [20] <7.14> Using data collected in Exercise 7.12, plot the two projections of seek time as used in Figure 7.51 (page 580). What seek distance has the largest percentage of difference between these two predictions? If you have the real seek distance data from Exercise 7.12, add that data to the plot and see on average how close each projection is to the real seek times.

n Multiply both targets by factors of at least 10X

7.14 [15] <7.2,7.4> Using the answer to Exercise 7.13, which disk would be a good building block to build a 100-GB storage subsystem using mirroring (RAID 1)? Why?

7.15 [15] <7.2,7.4> Using the answer to Exercise 7.13, which disk would be a good building block to build a 1000-GB storage subsystem using distributed parity (RAID 5)? Why?

n We need some queueing questions, but we need to be careful that they match the limited amount of queuing theory that they know; we had to drop stuff since I had some of it wrong. These next two figures point to the wrong example; may work with simple one there, which just calculates for a single disk.

7.16 [15] <7.7> Starting with the Example on page 538, calculate the average length of the queue and the average length of the system.

7.17 [15] <7.7> Redo the Example that starts on page 538, but this time assume the distribution of disk service times has a squared coefficient of variance of 2.0 (C = 2.0), versus 1.0 in the Example. How does this change affect the answers?

7.18 [20] <7.11> The I/O utilization rules of thumb on page 559 are just guidelines and are subject to debate. Redo the Example starting on page 560, but increase the limit of SCSI utilization to 50%, 60%, ..., until it is never the bottleneck. How does this change affect the answers? What is the new bottleneck? (*Hint*: Use a spreadsheet program to find answers.)

n Do next one for STC Powderhorn in Figure 7.6 on page 498

7.19 [15]<7.2> Tape libraries were invented as archival storage, and hence have relatively few readers per tape. Calculate how long it would take to read all the data for a system with 6000 tapes, 10 readers that read at 9 MB/sec, and 30 seconds per tape to put the old tape away and load a new tape.

n Replace Byte with PC Magazine, and record both January and July. Mention which figures: Figure 7.5 on page 495 and Figure 7.4 on page 494

7.20 [25]<7.2>Extend the figures, showing price per system and price per megabyte of disks by collecting data from advertisements in the January issues of *Byte* magazine after 1995. How fast are prices changing now

8 Interconnection Networks and Clusters

"The Medium is the Message" because it is the medium that shapes and controls the search and form of human associations and actions.

Marshall McLuhan
Understanding Media (1964)

The marvels—of film, radio, and television—are marvels of one-way communication, which is not communication at all.

Milton Mayer
On the Remote Possibility of Communication (1967)

8.1	Introduction	563
8.2	A Simple Network	570
8.3	Interconnection Network Media	580
8.4	Connecting More Than Two Computers	583
8.5	Network Topology	592
8.6	Practical Issues for Commercial Interconnection Networks	600
8.7	Examples of Interconnection Networks	604
8.8	Internetworking	610
8.9	Crosscutting Issues for Interconnection Networks	615
8.10	Clusters	619
8.11	Designing a Cluster	624
8.12	Putting It All Together: The Goggle Cluster of PCs	638
8.13	Another View: Inside a Cell Phone	645
8.14	Fallacies and Pitfalls	650
8.15	Concluding Remarks	653
8.16	Historical Perspective and References	654
	Exercises	660

8.1 Introduction

Thus far we have covered the components of a single computer, which has been the traditional focus of computer architecture. In this chapter we see how to connect computers together, forming a community of computers. Figure 8.1 shows the generic components of this community: computer nodes, hardware and software interfaces, links to the interconnection network, and the interconnection network. Interconnection networks are also called *networks* or *communication subnets*, and nodes are sometimes called *end systems* or *hosts*. The connection of two or more interconnection networks is called *internetworking*, which relies on communication standards to convert information from one kind of network to another. The Internet is the most famous example of internetworking.

There are two reasons that computer architects should devote attention to networking. In addition to providing external connectivity, Moore's Law shrunk networks so much that they connect the components *within* a single computer. Using a network to connect autonomous systems within a computer has long been found in mainframes, but today this such designs can be found in PCs too. Switches are replacing buses as the normal communication technique: between

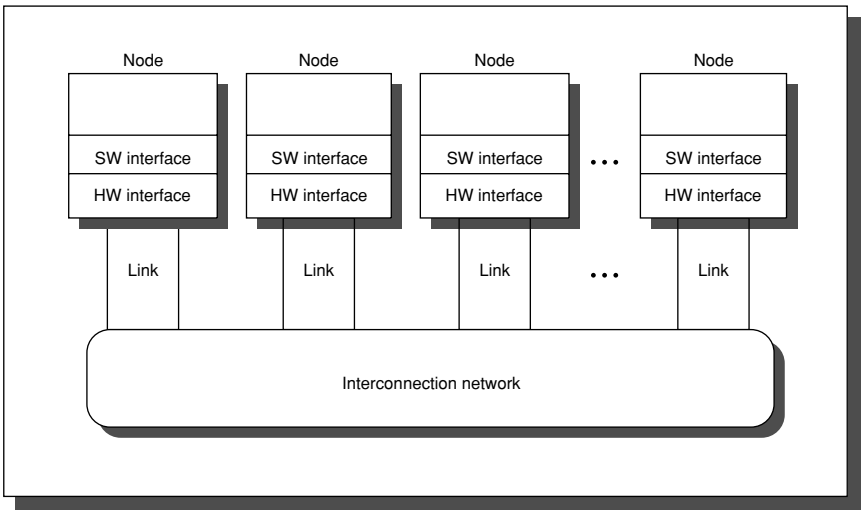


FIGURE 8.1 Drawing of the generic interconnection network.

computers, between I/O devices, between boards, between chips, and even between modules inside chips. As a result, computer architects must understand networking terminology, problems and solutions in order to design and evaluate modern computers.

The second reason architects should study networking is that today almost all computers are--or will be--networked to other devices. Thus, understanding networking is critical; any device without a network is somehow flawed. Just as a modern computer without a memory hierarchy "broken"–hence a chapter just for it–a modern computer without a network is "broken" too. Hence this chapter.

This topic is vast, with portions of Figure 8.1 the subject of whole books and college courses. Networking is also a buzzword-rich environment, where many simply ideas are obscured behind acronyms and unusual definitions. To help you breakthrough the buzzword barrier, Figure 8.2 is a glossary of about 80 networking terms. The goal of this chapter is to provide computer architects a gentle, qualitative introduction to networking. It defines terms, helps you understand the architectural implications of interconnection network technology, provides introductory explanations of the key ideas, and give references to more detailed descriptions.

Most of this chapter is on networking, but the final quarter of this chapter focuses on clusters. A *cluster* is the coordinated use of interconnected computers in a machine room. In contrast to the qualitative network introduction, these sections give a more quantitative description of clusters, including many examples. It ends with a guided tour of the Google clusters.

Term	*Definition*
adaptive routing	Router picks best path based upon measure of delay on outgoing links
ATM	Asynchronous Transfer Mode is a WAN designed for real-time traffic such as digital voice
attenuation	Loss of signal strength as signal passes through the medium over a long distance
backpressure flow control	When the receiver cannot accept another message, separate wires between adjacent senders and receivers tell the sender to stop immediately. It causes links between two end points to freeze until the receiver makes room for the next message.
bandwidth	Maximum rate the network can propagate information once the message enters the it
base station	A network architecture that uses boxes connected via land lines to communicate to wireless handsets
bisection bandwidth	Sum of the bandwidth of lines that cross that imaginary dividing line between two roughly equal parts of the network, each with half the nodes
bit error rate	BER, the error rate of a network, typically in errors per million bits transferred
blade	A removable computer component that fits vertically into a box in a standard VME rack
blocking	Contention that prevents a message from making progress along a link of a switch
bridge	OSI layer 2 networking device that connects multiple LANs, which can operate in parallel; in contrast, a router connects networks with incompatible addresses at OSI layer 3
category 5 wire	"Cat 5" twisted-pair, copper wire used for 10, 100, and 1000 Mbits/sec LANs
carrier sensing	"Listening" to the medium to be sure it is unused before trying to send a message
channel	In wireless networks, it is a pair of frequency bands that allow 2-way communication
checksum	A field of a message for a error correction code
circuit switching	A circuit is established from source to destination, reserving bandwidth along a path until the circuit is broken
cluster	Coordinated use of interconnected computers in a machine room
coaxial cable	A single stiff copper wire is surrounded by insulating material and a shield; historically faster and longer distance than twisted pair copper wire
collision	Two nodes (or more) on a shared medium try to send at the same time
collision detection	"Listening" to shared medium after sending to see if a message collided with another

FIGURE 8.2 Networking terms in this chapter and their definitions

Term	*Definition*
collocation site	A warehouse for remote hosting of servers with expansible networking, space, cooling, and security
communication subnets	Another name for interconnection network
credit-based flow control	To reduce overhead for flow control, a sender is given a credit to send up to N packets, and only checks for network delays when the credit is spent
cut-through routing	The switch examines the header, decides where to send the message, and then start transmitting it immediately without waiting for the rest of the message. When the head of the message blocks, the message stays strung out over the network.
destination-based routing	The message contains a destination address, and the switch picks a path to deliver the message, often by table lookup
deterministic routing	Router always picks the same path for the message
end systems	Another name for interconnection network node as opposed to the intermediate switches
end-to-end argument	Intermediate functions (error checking, performance optimization, and so on) may be incomplete as compared to performing the function end-to end
Ethernet	The most popular LAN, it has scaled from its original 3 Mbits/second rate using shared media in 1975 to switched media at 1000 Mbits/second in 2001; it shows no signs of stopping
fat tree	a network topology with extra links at each level enhancing a simple tree, so bandwidth between each level is normally constant (see Figure 8.14 on page 595)
FC-AL	Fibre Channel Arbitrated Loop; a SAN for storage devices
frequency-division multiplexing	Divide the bandwidth of the transmission line into a fixed number of frequencies, and assign each frequency to a conversation.
full duplex	Two-way communication on a network segment
header	The first part of a message that contains no user information, but contents helps that network, such as providing the destination address
host	Another name for interconnection network node
hub	An OSI layer 1 networking device that connects multiples LANs to act as one
Infiniband	An emerging standard SAN for both storage and systems in a machine room

FIGURE 8.2 Networking terms in this chapter and their definitions

Term	*Definition*
interference	In wireless networks, reduction of signal due to frequency reuse; frequency is reused to try to increase the number of simultaneous conversations over a large area
internetworking	Connection of two or more interconnection networks
IP	Internet Protocol is an OSI layer 3 protocol, at the network layer
iSCSI	SCSI over IP networks, it is a competitor to SANs using IP and Ethernet switches
LAN	Local Area Network, for machines in a building or campus, such as Ethernet
message	The smallest piece of electronic "mail" sent over a network
multimode fiber	An inexpensive optical fiber that reduces bandwidth and distance for cost
multipath fading	In wireless networks, interference between multiple versions of signal that arrive at different times, determined by time between fastest signal and slowest signal relative to signal bandwidth
multistage switch	a switch containing many smaller switches that perform a portion of routing
OSI layer	Open System Interconnect models the network as seven layers (see 8.25 on page 612)
overhead	In this chapter, networking overhead is sender overhead + receiver overhead + time of flight
packet switching	In contrast to circuit switching, information is broken into packets (usually fixed or maximum size), each with it's own destination address, and they are routed independently
payload	The middle part of the message that contains user information
peer-to-peer protocol	Communication between two nodes occurs logically at the same level of the protocol
peer-to-peer wireless	Instead of communicating to base stations, peer-to-peer wireless networks communicate between handsets
protocol	The sequence of steps that network software follows to communicate
rack unit	An R.U. is 1.7 inches, the height of a single slot in a standard 19-inch VME rack; there are 44 R.U. in standard 6-foot rack
receiver overhead	The time for the processor to pull the message from the interconnection network
router	OSI layer 3 networking device that connects multiples LANs with incompatible addresses
SAN	Originally System Area Network but more recently Storage Area Network, it connects computers and/or storage devices in a machine room. FC-AL or Infiniband are SANs.

FIGURE 8.2 Networking terms in this chapter and their definitions

Term	*Definition*
sender overhead	The time for the processor to inject the message into the network; the processor is busy for the entire time
shadow fading	In wireless networks, when the received signal is blocked by objects; buildings outdoors or walls indoors
signal-to-noise ratio	SNR, the ratio of the strength of the signal carrying information to the background noise
simplex	One-way communication on a network segment
single-mode fiber	"Single-wavelength" fiber is narrower and more expensive than multimode fiber but it offers greater bandwidth and distance
source-based routing	The message specifies the path to the destination at each switch
store-and-forward	Each switch waits for the full message to arrive before it is sent on to the next switch
TCP	Transmission Control Protocol, it is an OSI layer 4 protocol (transport layer)
throughput	In networking, measured speed of the medium or network bandwidth delivered to an application; i.e., does not give credit for headers and trailers
time of flight	The time for the first bit of the message to arrive at the receiver
trailer	The last part of a message that has no user information but helps the network, such as error correction code
transmission time	The time for the message to pass through the network (not including time of flight)
transport latency	Time that the message spends in the interconnection network (including time of flight)
twisted pairs	Two wires twisted together to reduce electrical interference
virtual circuit	A logical circuit is established between source and destination for a message to follow
WAN	Wide Area Network, a network across a continent, such as ATM
wavelength division multiplexing	WDM sends different streams simultaneously on the same fiber using different wavelengths of light and then demultiplexes the different wavelengths at the receiver
window	In TCP, the number of TCP datagrams that can be sent without waiting for approval
wireless network	A network that communicates without physical connections, such as radio
wormhole routing	The switch examines the header, decides where to send the message, and then starts transmitting it immediately without waiting for the rest of the message. The tail continues when the head blocks, potentially compressing the strung-out message into a single switch

FIGURE 8.2 Networking terms in this chapter and their definitions

Let's start with the generic types of interconnections. Depending on the number of nodes and their proximity, these interconnections are given different names:

- *Wide area network (WAN)*—Also called *long haul network*, the WAN connects computers distributed throughout the world. WANs include thousands of computers, and the maximum distance is thousands of kilometers. ATM is a current example of a WAN.
- *Local area network (LAN)*—This device connects hundreds of computers, and the distance is up to a few kilometers. Unlike a WAN, a LAN connects computers distributed throughout a building or on a campus. The most popular and enduring LAN is Ethernet.
- *Storage or System area network (SAN)*—This interconnection network is for a machine room, so the maximum distance of a link is typically less than 100 meters, and it can connect hundreds of nodes. Today SAN usually means *Storage* area network as it connects computers to storage devices, such as disk arrays. Originally SAN meant a *System* area network to connect computers together, such as PCs in a cluster. A recent SAN trying to network both storage and system is Infiniband.

Figure 8.3 shows the rough relationship of these systems in terms of number autonomous systems connected, including a bus for comparison. Note the area of overlap between buses, SANs, and LANs, which lead to product competition.

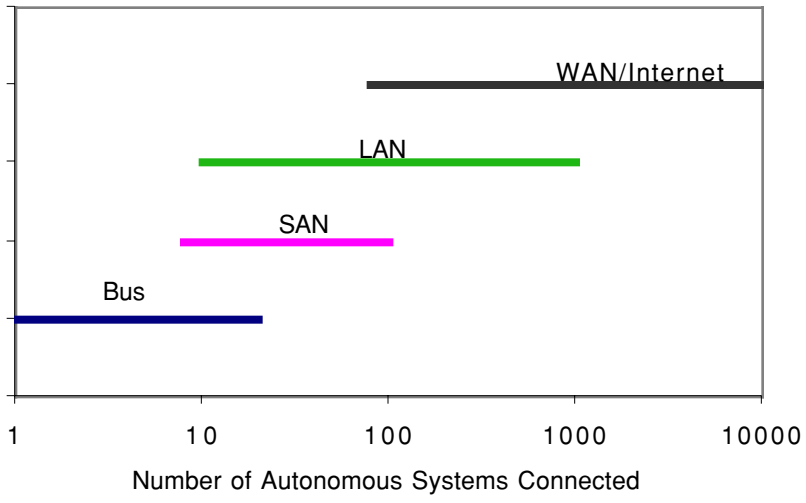


FIGURE 8.3 Relationship of four types of interconnects in terms of number of autonomous systems connected: bus, system or storage area network, local area network, and wide area network/Internet. Note that there are overlapping ranges where buses, SANs, and LANs compete. Some supercomputers have a switch-based custom network to interconnect up to thousands of computers; such interconnects are basically custom SANs.

These three types of interconnection networks have been designed and sustained by several different cultures—Internet, telecommunications, workgroup/enterprise, storage, and high performance computing—each using its own dialects and its own favorite approaches to the goal of interconnecting autonomous computers.

This chapter gives a common framework for evaluating all interconnection networks, using a single set of terms to describe the basic alternatives. Figure 8.22 in section 8.7 gives several other examples of each of these interconnection networks. As we shall see, some components are common to all types and some are quite different.

We begin the chapter in section 8.2 by exploring the design and performance of a simple network to introduce the ideas. We then consider the following problems: which media to use as the interconnect (8.3), how to connect many computers together (8.4 and 8.5), and what are the practical issues for commercial networks (8.6). We follow with examples illustrating the trade-offs for each type of network (8.7), explore internetworking (8.8), and cross cutting issues for networks (8.9). With this gentle introduction to networks in sections 8.2 to 8.9, readers interested in more depth should try the suggested reading in section 8.16. Sections 8.10 to 8.12 switch to clusters, and give a more quantitative description with designs and examples. Section 8.13 gives a view of networks from the embedded perspective, using a cell phone and wireless networks as the example. We conclude in sections 8.14 to 8.16 with the traditional ending of the chapters.

As we shall see, networking shares more characteristics with storage than with processors and memory. Like storage, the operating system controls what features of the network are used. Again like storage, performance includes both latency and bandwidth, and queueing theory is a valuable tool. Like RAID, networking assumes failures occur, and thus dependability in the presence of errors is the norm.

8.2 A Simple Network

There is an old network saying: Bandwidth problems can be cured with money. Latency problems are harder because the speed of light is fixed—you can't bribe God.

Anonymous

To explain the complexities and concepts of networks, this section describes a simple network of two computers. We then describe the software steps for these two machines to communicate. The remainder of the section gives a detailed and then a simple performance model, including several examples to see the implications of key network parameters.

Suppose we want to connect two computers together. Figure 8.4 shows a simple model with a unidirectional wire from machine A to machine B and vice versa. At the end of each wire is a first-in-first-out (FIFO) queue to hold the data. In this simple example, each machine wants to read a word from the other's memory. A *message* is the information sent between machines over an interconnection network.

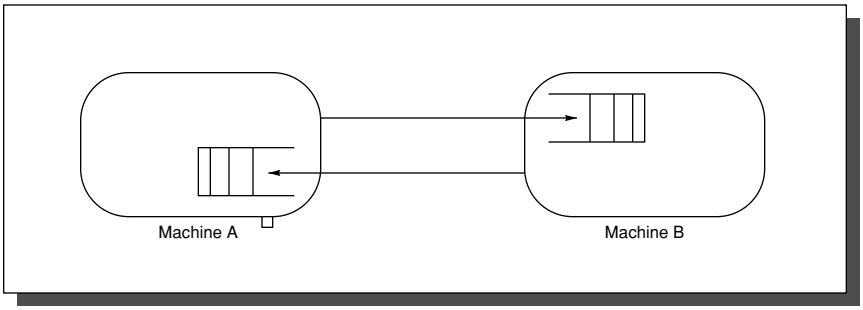


FIGURE 8.4 A simple network connecting two machines.

For one machine to get data from the other, it must first send a request containing the address of the data it desires from the other node. When a request arrives, the machine must send a reply with the data. Hence, each message must have at least 1 bit in addition to the data to determine whether the message is a new request or a reply to an earlier request. The network must distinguish between information needed to deliver the message, typically called the *header* or the *trailer* depending on where it is relative to the data, and the *payload,* which contains the data. Figure 8.5 shows the format of messages in our simple network. This example shows a single-word payload, but messages in some interconnection networks can include hundreds of words.

Interconnection networks involve normally software. Even this simple example invokes software to translate requests and replies into messages with the appropriate headers. An application program must usually cooperate with the operating system to send a message to another machine, since the network will be shared with all the processes running on the two machines, and the operating system cannot allow messages for one process to be received by another. Thus, the messaging software must have some way to distinguish between processes; this distinction may be included in an expanded header. Although hardware support can reduce the amount of work, most is done by software.

In addition to protection, network software is often responsible for ensuring reliable delivery of messages. The twin responsibilities are ensuring that the message is neither garbled nor lost in transit.

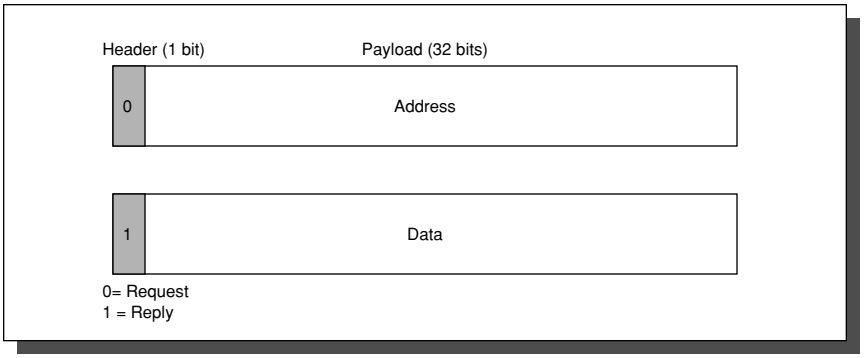


FIGURE 8.5 Message format for our simple network. Messages must have extra information beyond the data.

Adding a *checksum* field (or some other error detection code) to the message format meets the first responsibility. This redundant information is calculated when the message is first sent and checked upon receipt. The receiver then sends an acknowledgment if the message passes the test.

One way to meet the second responsibility is to have a timer record the time each message is sent and to presume the message is lost if the timer expires before an acknowledgment arrives. The message is then re-sent.

The software steps to send a message are as follows:

1. The application copies data to be sent into an operating system buffer.
2. The operating system calculates the checksum, includes it in the header or trailer of the message, and then starts the timer.
3. The operating system sends the data to the network interface hardware and tells the hardware to send the message.

Message reception is in just the reverse order:

3. The system copies the data from the network interface hardware into the operating system buffer.
2. The system calculates the checksum over the data. If the checksum matches the sender's checksum, the receiver sends an acknowledgment back to the sender. If not, it deletes the message, assuming that the sender will resend the message when the associated timer expires.
1. If the data pass the test, the system copies the data to the user's address space and signals the application to continue.

The sender must still react to the acknowledgment:

- When the sender gets the acknowledgment, it releases the copy of the message from the system buffer.
- If the sender gets the time-out instead of an acknowledgment, it resends the data and restarts the timer.

Here we assume that the operating system keeps the message in its buffer to support retransmission in case of failure. Figure 8.6 shows how the message format looks now.

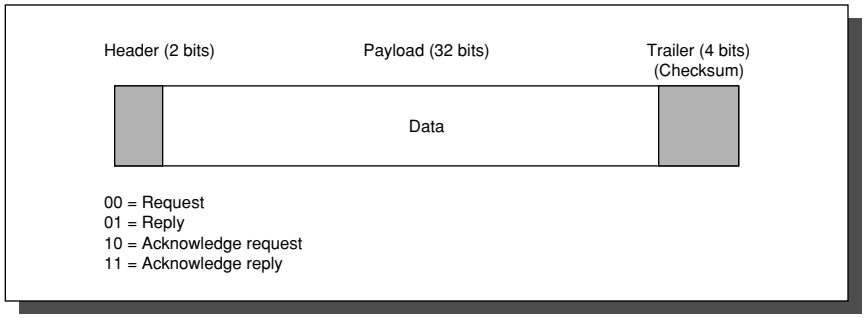


FIGURE 8.6 Message format for our simple network. Note that the checksum is in the trailer.

The sequence of steps that software follows to communicate is called a *protocol* and generally has the symmetric but reversed steps between sending and receiving.

Note that this example protocol above is for sending a *single* message. When an application does not require a response before sending the next message, the sender can overlap the time to send with the transmission delays and the time to receive.

A protocol must handle many more issues than reliability. For example, if two machines are from different manufacturers, they might order bytes differently within a word (see section 2.3 of Chapter 2). The software must reverse the order of bytes in each word as part of the delivery system. It must also guard against the possibility of duplicate messages if a delayed message were to become unstuck. It is often necessary to deliver the messages to the application in the order they are sent, and so sequence numbers may be added to the header to enable assembly. Finally, it must work when the receiver's FIFO becomes full, suggesting feedback to control the flow of messages from the sender (see section 8.4).

Now that we have covered the steps in sending and receiving a message, we can discuss performance.

Figure 8.7 shows the many performance parameters of interconnection networks. This figure is critical to understanding network performance, so study it

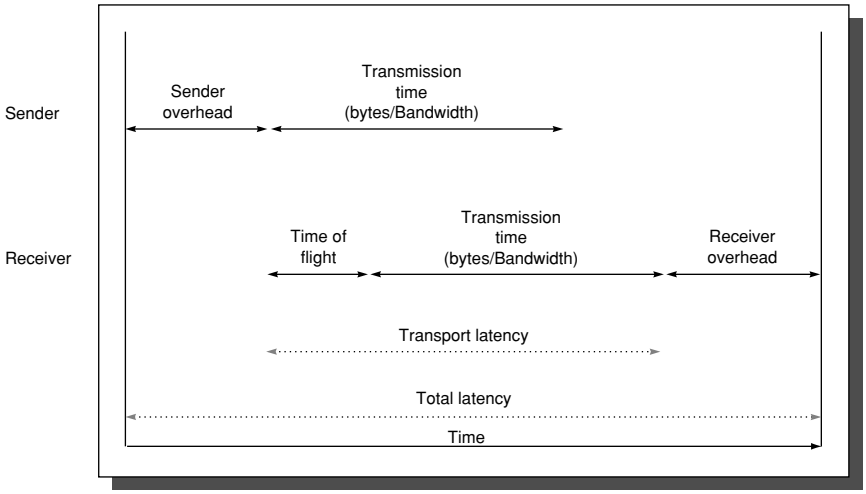


FIGURE 8.7 Performance parameters of interconnection networks. Depending on whether it is an SAN, LAN, or WAN, the relative lengths of the time of flight and transmission may be quite different from those shown here. (Based on a presentation by Greg Papadopolous of Sun Microsystems.)

well! Note that the parameters in Figure 8.7 apply to the interconnect in *many* levels of the system: inside a chip, between chips on a board, between computers in a cluster, and so on. The units change, but the principles remain the same, as does the bandwidth that results.

These terms are often used loosely, leading to confusion, so we define them here precisely:

- *Bandwidth*—We use this most widely used term to refer to the maximum rate at which the network can propagate information once the message enters the network. Unlike disks, bandwidth includes the headers and trailers as well as the payload, and the units are traditionally bits/second rather than bytes/second. The term bandwidth is also used to mean the measured speed of the medium or network bandwidth delivered to an application. *Throughput* is sometimes used for this latter term.

- *Time of flight*—The time for the first bit of the message to arrive at the receiver, including the delays due to repeaters or other hardware in the network. Time of flight can be milliseconds for a WAN or nanoseconds for an SAN.

- *Transmission time*—The time for the message to pass through the network, not including time of flight. One way to measure it is the difference in time between when the first bit of the message arrives at the receiver and when the last bit of the message arrives at the receiver. Note that by definition transmission time is equal to the size of the message divided by the bandwidth. This measure assumes there are no other messages to contend for the network.

- *Transport latency*—The sum of time of flight and transmission time. Transport latency is the time that the message spends in the interconnection network. Stated alternatively, it is the time between when the first bit of the message is injected into the network and when the last bit of the message arrives at the receiver. It does not include the overhead of injecting the message into the network nor pulling it out when it arrives.
- *Sender overhead*—The time for the processor to inject the message into the network, including both hardware and software components. Note that the processor is busy for the entire time, hence the use of the term *overhead*. Once the processor is free, any subsequent delays are considered part of the transport latency. For pedagogic reasons, we assume overhead is not dependent on message size. (Typically, only very large messages have larger overhead.)
- *Receiver overhead*—The time for the processor to pull the message from the interconnection network, including both hardware and software components. In general, the receiver overhead is larger than the sender overhead: for example, the receiver may pay the cost of an interrupt.

The total latency of a message can be expressed algebraically:

$$\text{Total latency} = \text{Sender overhead} + \text{Time of flight} + \frac{\text{Message size}}{\text{Bandwidth}} + \text{Receiver overhead}$$

Let's look at how the time of flight and overhead parameters change in importance as we go from SAN to LAN to WAN.

EXAMPLE

Assume a network with a bandwidth of 1000 Mbits/second has a sending overhead of 80 microseconds and a receiving overhead of 100 microseconds. Assume two machines. One wants to send a 10000-byte message to the other (including the header), and the message format allows 10000 bytes in a single message. Let's compare SAN, LAN, and WAN by changing the distance between the machines. Calculate the total latency to send the message from one machine to another in a SAN assuming they are 10 meters apart. Next, perform the same calculation but assume the machines are now 500 meters apart, as in a LAN. Finally, assume they are 1000 *kilometers* apart, as in a WAN.

ANSWER

The speed of light is 299,792.5 kilometers per second in a vacuum, and signals propagate at about 63% to 66% of the speed of light in a conductor. Since this is an estimate, in this chapter we'll round the speed of light to 300,000 kilometers per second, and assume we can achieve two-thirds of that in a conductor. Hence, we can estimate time of flight. Let's plug the parameters for the short distance of a SAN into the formula above:

$$\text{Total latency} = \text{Sender overhead} + \text{Time of flight} + \frac{\text{Message size}}{\text{Bandwidth}} + \text{Receiver overhead}$$

$$= 80\mu\text{secs} + \frac{0.01\text{km}}{2/3 \times 300,000\ \text{km/sec}} + \frac{10000\ \text{bytes}}{1000\ \text{Mbits/sec}} + 100\ \mu\text{secs}$$

Converting all terms into microseconds (μsecs) leads to

$$\text{Total latency} = 80\mu\text{secs} + \frac{0.01 \times 10^6}{2/3 \times 300,000}\ \mu\text{secs} + \frac{10000 \times 8}{1000}\ \mu\text{secs} + 100\ \mu\text{secs}$$

$$= 80\ \mu\text{secs} + 0.05\ \mu\text{secs} + 80\ \mu\text{secs} + 100\ \mu\text{sec} = 260 + 0.05\ \mu\text{secs}$$

$$= 260\mu\text{secs}$$

Substituting an example LAN distance into the third equation yields

$$\text{Total latency} = 80\mu\text{secs} + \frac{0.5\text{km}}{2/3 \times 300,000\ \text{km/sec}} + \frac{10000\ \text{bytes}}{1000\ \text{Mbits/sec}} + 100\ \mu\text{secs}$$

$$= 80\ \mu\text{secs} + 2.50\ \mu\text{secs} + 80\ \mu\text{secs} + 100\ \mu\text{sec} = 260 + 2.5\ \mu\text{secs}$$

$$= 262\mu\text{secs}$$

Substituting the WAN distance into the equation yields

$$\text{Total latency} = 80\mu\text{secs} + \frac{1000\ \text{km}}{2/3 \times 300,000\ \text{km/sec}} + \frac{10000\ \text{bytes}}{1000\ \text{Mbits/sec}} + 100\ \mu\text{secs}$$

$$= 80\ \mu\text{secs} + 5000\ \mu\text{secs} + 80\ \mu\text{secs} + 100\ \mu\text{sec} = 260 + 5000\ \mu\text{secs}$$

$$= 5260\mu\text{secs}$$

The increased fraction of the latency required by time of flight for long distances, as well as the greater likelihood of errors over long distances, are why wide area networks use more sophisticated and time-consuming protocols. Complexity increases from protocols used on a bus versus a LAN versus the Internet as we go from ten to hundreds to thousands of nodes.

Note that messages in LANs and WANs go through switches which add to the latency, which we neglected above. Generally, switch latency is small compared to overhead in LANs or time of flight in SANs.

As mentioned above, when an application does not require a response before sending the next message, the sender can overlap the sending overhead with the transport latency and receiver overhead. Increased latency affects the structure of programs that try to hide this latency, requiring quite different solutions if the latency is 1, 100, or 10,000 microseconds.

n

Note that the example above shows that time of flight for SANs is so short relative to overhead that it can be ignored, yet in WANs, time of flight is so long that sender and receiver overheads can be ignored. Thus, we can simplify the performance equation by combining sender overhead, receiver overhead, and time of flight into a single term called *Overhead*:

$$\text{Total latency} \approx \text{Overhead} + \frac{\text{Message size}}{\text{Bandwidth}}$$

We can use this formula to calculate the effective bandwidth delivered by the network as message size varies:

$$\text{Effective bandwidth} = \frac{\text{Message size}}{\text{Total latency}}$$

Let's use this simpler equation to explore the impact of overhead and message size on effective bandwidth.

EXAMPLE

Plot the effective bandwidth versus message size for overheads of 25 and 250 microseconds and for network bandwidths of 100, 1000, and 10000 Mbits/second. Vary message size from 16 bytes to 4 megabytes. For what message sizes is the effective bandwidth virtually the same as the raw network bandwidth? If overhead is 250 microseconds, for what message sizes is the effective bandwidth always less than 100 Mbits/second?

ANSWER

Figure 8.8 plots effective bandwidth versus message size using the simplified equation above. The notation "oX,bwY" means an overhead of X microseconds and a network bandwidth of Y Mbits/second. To amortize the cost of high overhead, message sizes must be four megabytes for effective bandwidth to be about the same as network bandwidth. Assuming the high overhead, message sizes about 3K bytes or less will not break the 100 Mbits/second barrier no matter what the actual network bandwidth.

Thus, we must lower overhead as well as increase network bandwidth unless messages are very large.n

Hence, message size is important in getting full benefit of fast networks. What is the natural size of messages? Figure 8.9 above shows the size of Network File System (NFS) messages for 239 machines at Berkeley collected over a period of one week. One plot is cumulative in messages sent, and the other is cumulative in data bytes sent. The maximum NFS message size is just over 8 KB, yet 95% of the messages are less than 192 bytes long..Figure 8.10 below shows the similar results for Internet traffic, where the maximum transfer unit was 1500 bytes.

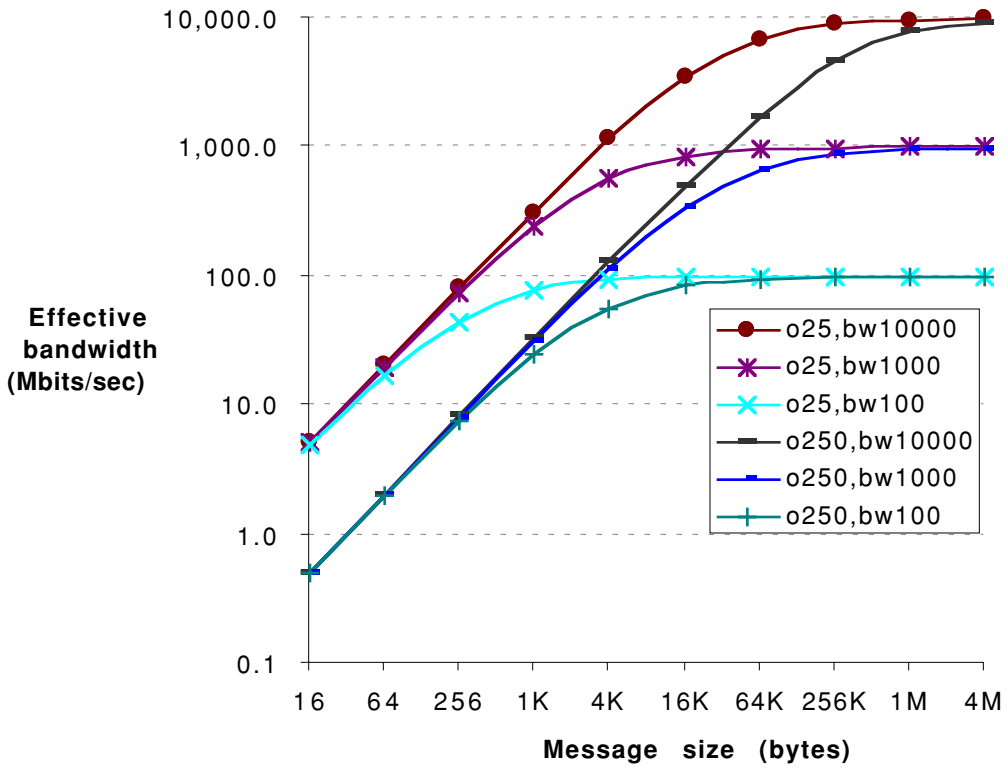


FIGURE 8.8 Bandwidth delivered versus message size for overheads of 25 and 250 microseconds and for network bandwidths of 100, 1000, and 10000 Mbits/second. Note that with 250 microseconds of overhead and a network bandwidth of 1000 Mbits/second, only the 4-MB message size gets an effective bandwidth of 1000 Mbits/second. In fact, message sizes must be greater than 256 B for the effective bandwidth to exceed 10 Mbits/second. The notation "oX,bwY" means an overhead of X microseconds and a network bandwidth of Y Mbits/second. **<<Artist: label lines, drop legend.>>**

Again, 60% of the messages are less than 192 bytes long, and 1500-byte messages represented 50% of the bytes transferred. Many applications send far more small messages than large messages, since requests and acknowledgements are more frequent than data

Summarizing this section, even this simple network has brought up the issues of protection, reliability, heterogeneity, software protocols, and a more sophisticated performance model. The next four sections address other key questions:

- Which media are available to connect computers together?
- What issues arise if you want to connect more than two computers?
- What practical issues arise for commercial networks?

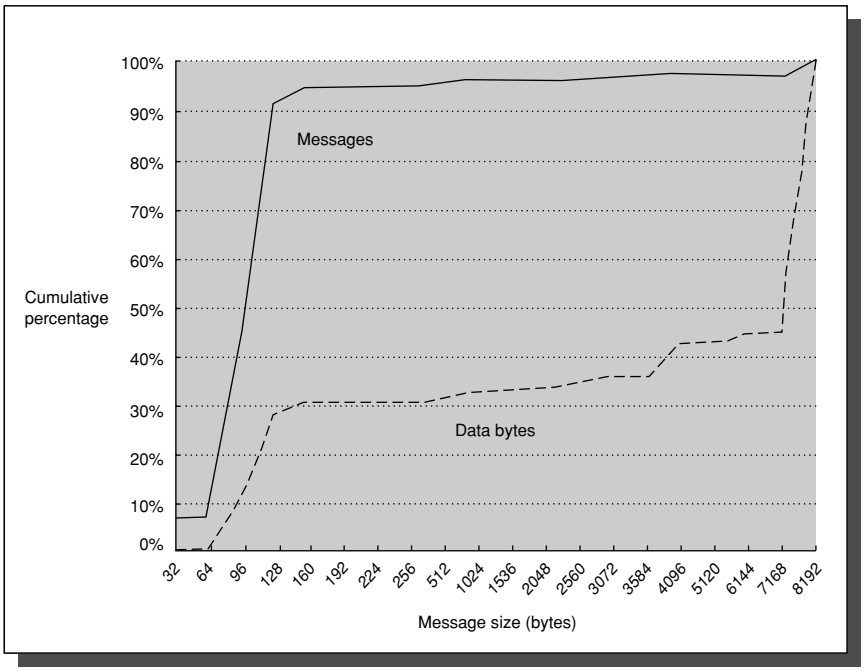


FIGURE 8.9 Cumulative percentage of messages and data transferred as message size varies for NFS traffic. Each x-axis entry includes all bytes up to the next one; e.g., 32 represents 32 bytes to 63 bytes. More than half the bytes are sent in 8-KB messages, but 95% of the messages are less than 192 bytes. Figure 8.50 (page 651) shows details of this measurement. Collected at the University of California at Berkeley.

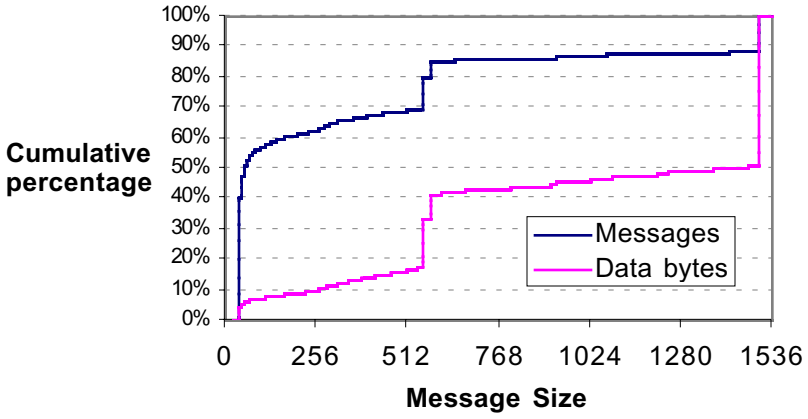


FIGURE 8.10 Cumulative percentage of messages and data transferred as message size varies for Internet traffic. About 40% of the messages were 40 bytes long, and 50% of the data transfer was in messages 1500 bytes long. The maximum transfer unit of most switches was 1500 bytes. Collect by Vern Paxton on MCI Internet traffic in 1998.

8.3 Interconnection Network Media

Just as there is a memory hierarchy, there is a hierarchy of media to interconnect computers that varies in cost, performance, and reliability. Network media have another figure of merit, the maximum distance between nodes. This section covers three popular examples, and Figure 8.11 illustrates them.

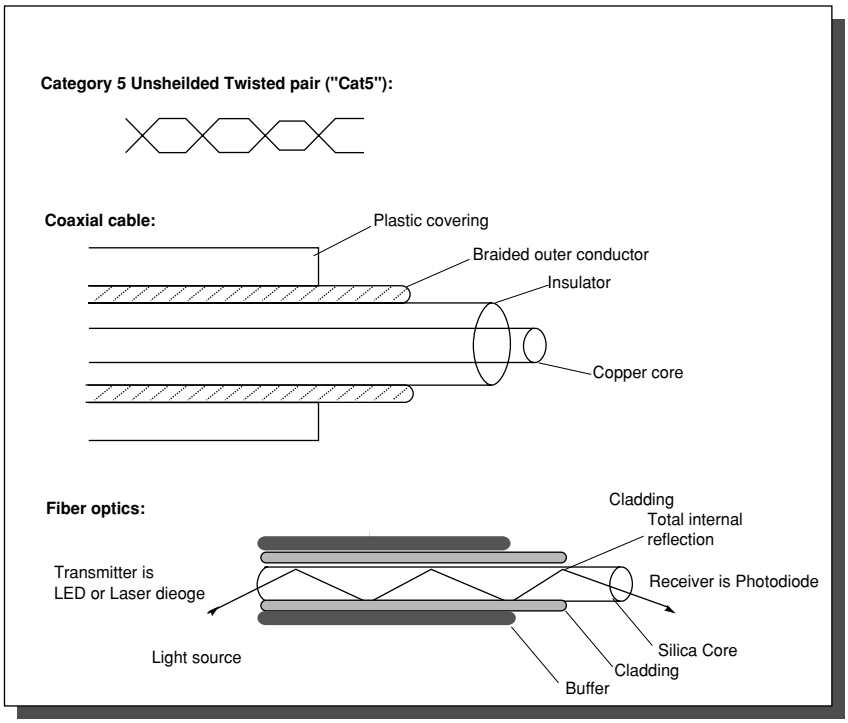


FIGURE 8.11 Three network media. (From a presentation by David Culler of U.C. Berkeley.)

The first medium is *twisted pairs* of copper wires. These are two insulated wires, each about 1 mm thick. They are twisted together to reduce electrical interference, since two parallel lines form an antenna but a twisted pair does not. As they can transfer a few megabits per second over several kilometers without amplification, twisted pair were the mainstay of the telephone system. Telephone companies bundled together (and sheathed) many pairs coming into a building. Twisted pairs can also offer tens of megabits per second of bandwidth over shorter distances, making them plausible for LANs.

The desire to go at higher speeds with the less expensive copper led to improvements in the quality of unshielded twisted-pair copper cabling systems. The original telephone-line quality was called Level 1. Level 3 was good enough for 10 Mbits/second Ethernet. The desire for even greater bandwidth lead to the Level 5 or Category 5, which is sufficient for 100 Mbits/second Ethernet. By limiting the length to 100 meters, "Cat5" wiring can be used for 1000 Mbits/second Ethernet links today. It uses the RJ-45 connector, which is similar to the connector found on telephone lines.

Coaxial cable was deployed by cable television companies to deliver a higher rate over a few kilometers. To offer high bandwidth and good noise immunity, insulating material surrounds a single stiff copper wire, and then cylindrical conductor surrounds the insulator, often woven as a braided mesh. A 50-ohm baseband coaxial cable delivers 10 megabits per second over a kilometer.

Connecting to this heavily insulated media is more challenging. The original technique was a T junction: the cable is cut in two and a connector is inserted that reconnects the cable and adds a third wire to a computer. A less invasive solution is a vampire tap: a hole of precise depth and width is first drilled into the cable, terminating in the copper core. A connector is then screwed in without having to cut the cable.

To keep up with the demands of bandwidth and distance, it became clear that the telephone company would need to find new media. The solution could be more expensive provided that it offered much higher bandwidth and that supplies were plentiful. The answer was to replace copper with glass and electrons with photons. *Fiber optics* transmits digital data as pulses of light.

A fiber optic network has three components:

1. the transmission medium, a fiber optic cable;
2. the light source, an LED or laser diode;
3. the light detector, a photodiode.

First, cladding surrounds the glass fiber core to confine the light. A buffer then surrounds the cladding to protect the core and cladding. Note that unlike twisted pairs or coax, fibers are one-way, or *simplex,* media. A two-way, or *full duplex,* connection between two nodes requires two fibers.

Since light bends or refracts at interfaces, it can slowly spread as it travels down the cable unless the diameter of the cable is limited to one wavelength of light; then it transfers in a straight line. Thus, fiber optic cables are of two forms:

1. *Multimode fiber*—It uses inexpensive LEDs as a light source. It is typically much larger than the wavelength of light: typically 62.5 microns in diameter vs. the 1.3-micron wavelength of infrared light. Since it is wider it has more dispersion problems, where some wave frequencies have different propagation velocities. The LEDs and dispersion limit it to up to a few hundred meters at 1000 Mbits/second or a few kilometers at 100 Mbits /second. It is older and less expensive than single mode fiber.

2. *Single-mode fiber*—This "single-wavelength" fiber (typically 8 to 9 microns in diameter) requires more expensive laser diodes for light sources and currently transmits gigabits per second for hundreds of kilometers, making it the medium of choice for telephone companies. The loss of signal strength as it passes through a medium, called *attenuation*, limits the length of the fiber.

Although single-mode fiber is a better transmitter, it is much more difficult to attach connectors to single-mode; it is less reliable and more expensive, and the cable itself has restrictions on the degree it can be bent. The cost, bandwidth, and distance of single-mode fiber is affected by the power of the light source, the sensitivity of the light detector, and the attenuation rate per kilometer of the fiber cable. Typically, glass fiber has better characteristics than the less expensive plastic fiber, and so is more widely used.

Connecting fiber optics to a computer is more challenging than connecting cable. The vampire tap solution of cable fails because it loses light. There are two forms of T-boxes:

1. Taps are fused onto the optical fiber. Each tap is passive, so a failure cuts off just a single computer.
2. In an active repeater, light is converted to electrical signals, sent to the computer, converted back to light, and then sent down the cable. If an active repeater fails, it blocks the network.

These taps and repeaters also reduce optical signal strength, reducing the useful distance of a single piece of fiber.

In both cases, fiber optics has the additional cost of optical-to-electrical and electrical-to-optical conversion as part of the computer interface. Hence, the network interface cards for fiber optics are considerably more expensive than for Cat5 copper wire. In 2001, most switches for fiber involve such a conversion to allow switching, although expensive all optical switches are beginning to be available.

To achieve even more bandwidth from a fiber, *wavelength division multiplexing (WDM)* sends different streams simultaneously on the same fiber using different wavelengths of light, and then demultiplexes the different wavelengths at the receiver. In 2001, WDM can deliver a combined 40 Gbits/second using about 8 wavelengths, with plans to go to 80 wavelengths and deliver 400 Gbits/second.

The product of the bandwidth and maximum distance forms a single figure of merit: gigabit-kilometers per second. According to Desurvire [1992], since 1975 optical fibers have increased transmission capacity by tenfold every four years by this measure.

Let's compare media in an example.

EXAMPLE Suppose you have 25 magnetic tapes, each containing 40 GB. Assume that you have enough tape readers to keep any network busy. How long will it take to transmit the data over a distance of one kilometer? Assume the choices are Category 5 twisted pair wires at 100 Mbits/second, multimode fiber at 1000 Mbits/second, and single mode fiber at 2500 Mbits/second. How do they compare to delivering the tapes by car?

ANSWER The amount of data is 1000 GB. The time for each medium is given below:

$$\text{Twisted pair} = \frac{1000 \times 1024 \times 8 \text{ Mb}}{100 \text{ Mb/sec}} = 81{,}920 \text{ secs} = 22.8 \text{ hours}$$

$$\text{Multimode fiber} = \frac{1000 \times 1024 \times 8 \text{ Mb}}{1000 \text{ Mb/sec}} = 8192 \text{ secs} = 2.3 \text{ hours}$$

$$\text{Single-mode fiber} = \frac{1000 \times 1024 \times 8 \text{ Mb}}{2500 \text{ Mb/sec}} = 3277 \text{ secs} = 0.9 \text{ hours}$$

$$\begin{aligned}\text{Car} &= \text{Time to load car} + \text{Transport time} + \text{Time to unload car}\\ &= 300 \text{ secs} + \frac{1 \text{ km}}{30 \text{ kph}} + 300 \text{ secs} = 300 \text{ secs} + 120 \text{ secs} + 300 \text{ secs}\\ &= 720 \text{ secs} = 0.3 \text{ hours}\end{aligned}$$

A car filled with high-density tapes is a high-bandwidth medium! n

8.4 Connecting More Than Two Computers

Computer power increases by the square of the number of nodes on the network.

Robert Metcalf (“Metcalf’s Law”)

Thus far, we have discussed two computers communicating over private lines, but what makes interconnection networks interesting is the ability to connect hundreds of computers together. And what makes them more interesting also makes them more challenging to build.

Shared versus Switched Media

Certainly the simplest way to connect multiple computers is to have them share a single interconnection medium, just as I/O devices share a single I/O bus. The most popular LAN, Ethernet, originally was simply a bus shared by a hundred of computers.

Given that the medium is shared, there must be a mechanism to coordinate and arbitrate the use of the shared medium so that only one message is sent at a time.

If the network is small, it may be possible to have an additional central arbiter to give permission to send a message. (Of course, this leaves open the question of how the nodes talk to the arbiter.)

Centralized arbitration is impractical for networks with a large number of nodes spread out over a kilometer, so we must distribute arbitration. A first step towards arbitration is looking before you leap. A node first checks the network to avoid trying to send a message while another message is already on the network. If the interconnection is idle, the node tries to send. Looking first is not a guarantee of success, of course, as some other node may decide to send at the same instant. When two nodes send at the same time, it is called a *collision*. Let's assume that the network interface can detect any resulting collisions by listening to hear if the data were garbled by other data appearing on the line. Listening to avoid and detect collisions is called *carrier sensing and collision detection*. This is the second step of arbitration.

The problem is not solved. If every node on the network waited exactly the same amount of time, listened to be sure there was no traffic, and then tried to send again, we could still have synchronized nodes that would repeatedly bump heads. To avoid repeated head-on collisions, each node whose message was garbled waits (or "backs off") a *random* time before resending. Note that randomization breaks the synchronization. Subsequent collisions result in exponentially increasing time between attempts to retransmit, so as not to tax the network.

Although this approach is not guaranteed to be fair—some subsequent node may transmit while those that collided are waiting—it does control congestion on the shared medium. If the network does not have high demand from many nodes, this simple approach works well. Under high utilization, performance degrades since the medium is shared.

Another approach to arbitration is to pass a token between the nodes, with the token giving the node the right to use the network. If the shared media is connected in a ring, then the token can rotate through all the nodes on the ring.

Shared media have some of the same advantages and disadvantages as buses: they are inexpensive, but they have limited bandwidth. And like buses, they must have a arbitration scheme to solve conflicting demands.

The alternative to sharing the media is to have a dedicated line to a switch that in turn provides a dedicated line to all destinations. Figure 8.12 shows the potential bandwidth improvement of switches: *Aggregate bandwidth* is many times that of a single shared medium.

Switches allow communication directly from source to destination, without intermediate nodes to interfere with these signals. Such *point-to-point* communication is faster than a line shared between many nodes because there is no arbitration and the interface is simpler electrically. Of course, it does pay the added latency of going through the switch, trading off arbitration overhead for switching overhead.

Given the obvious advantages, why weren't switches always used? Earlier computers were much slower and so could share media. In addition, applications

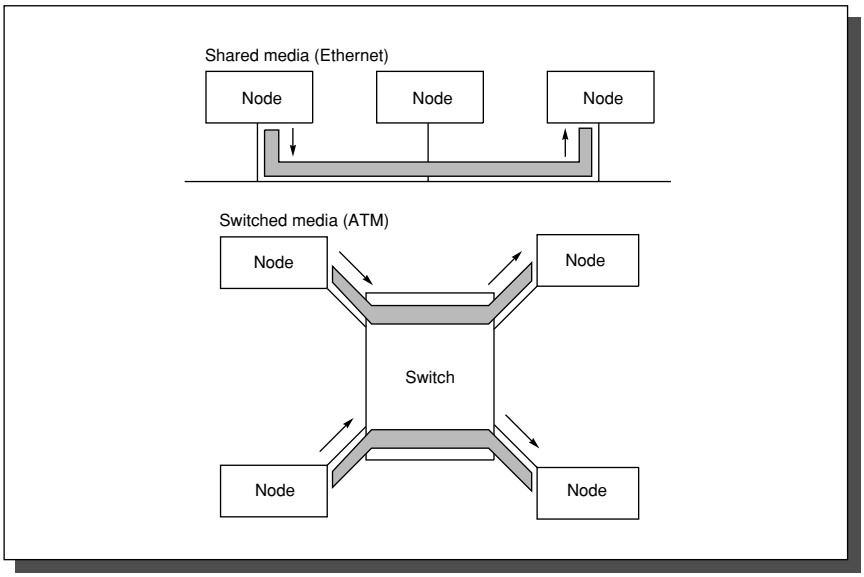


FIGURE 8.12 Shared medium versus switch. Ethernet was originally a shared medium, and but Ethernet switches are now available. All nodes on the shared media must share the 100 Mb/sec interconnection, but switches can support multiple 100 Mb/sec transfers simultaneously. Low cost Ethernet switches are sometimes implemented with an internal bus with higher bandwidth, but high-speed switches have a cross-bar interconnect.

such as the World Wide Web rely on the network much more than older applications. Finally, earlier switches would take several large boards, and be as a large as a computer. In 2001, a single chip contains a full 64-by-64 switch, or at least a large slice of it. Moore's Law is making switches more attractive, and so technology trends favor switches today.

Every node of a shared line will see every message, even if it is just to check to see whether or not the message is for that node, so this style of communication is sometimes called *broadcast* to contrast it with point-to-point. The shared medium makes it easy to broadcast a message to every node, and even to broadcast to subsets of nodes, called *multicasting*.

Switches allow multiple pairs of nodes to communicate simultaneously, giving these interconnections much higher *aggregate* bandwidth than the speed of a shared link to a node. Switches also allow the interconnection network to scale to a very large number of nodes. Switches are also called *data switching exchanges*, *multistage interconnection networks*, or even *interface message processors* (*IMPs*). Depending on the distance of the node to the switch and desired bandwidth, the network medium is either copper wire or optical fiber.

EXAMPLE Compare 16 nodes connected three ways: a single 100 Mb/sec shared media; a switch connected via Cat5, each segment running at 100 Mb/sec; and a switch connected via optical fibers, each running at 1000 Mb/sec. The shared media is 500 meters long, and the average length of each segment to a switch is 50 meters. Both switches can support the full bandwidth. Assume each switch adds 5 microseconds to the latency. Calculate the aggregate bandwidth and transport latency. Assume the average message size is 125 bytes, and ignore the overhead of sending or receiving a message and contention for the network.

ANSWER The aggregate bandwidth of each example is the simplest calculation: 100 Mb/sec for the shared media; 16×100, or 1600 Mb/sec for the switched twisted pairs; and 16×1000, or 16000 Mb/sec for the switched optical fibers.

The transport time is

$$\text{Transport time} = \text{Time of flight} + \frac{\text{Message size}}{\text{Bandwidth}}$$

For coax we just plug in the distance, bandwidth, and message size:

$$\begin{aligned}\text{Transport time}_{\text{shared}} &= \frac{500/1000 \times 10^6}{2/3 \times 300,000}\ \mu\text{secs} + \frac{125 \times 8}{100}\ \mu\text{secs} \\ &= 2.5\ \mu\text{secs} + 10\ \mu\text{secs} \\ &= 12.5\ \mu\text{secs}\end{aligned}$$

For the switches, the distance is twice the average segment, since there is one segment from the sender to the switch and one from the switch to the receiver. We must also add the latency for the switch.

$$\begin{aligned}\text{Transport time}_{\text{swtich}} &= 2 \times \left(\frac{50/1000 \times 10^6}{2/3 \times 300,000}\right)\ \mu\text{secs} + 5\ \mu\text{secs} + \frac{125 \times 8}{100}\ \mu\text{secs} \\ &= 0.5\ \mu\text{secs} + 5\ \mu\text{secs} + 10\ \mu\text{secs} \\ &= 15.5\ \mu\text{secs}\end{aligned}$$

$$\begin{aligned}\text{Transport time}_{\text{fiber}} &= 2 \times \left(\frac{50/1000 \times 10^6}{2/3 \times 300,000}\right)\ \mu\text{secs} + 5\ \mu\text{secs} + \frac{125 \times 8}{1000}\ \mu\text{secs} \\ &= 0.5\ \mu\text{secs} + 5\ \mu\text{secs} + 1\ \mu\text{secs} \\ &= 6.5\ \mu\text{secs}\end{aligned}$$

Although the bandwidth of the switch is many times the shared media, the latency for unloaded networks is comparable. n

Switches allow communication to harvest the same rapid advance from silicon as have processors and main memory. Whereas the switches from telecommunications companies were once the size of mainframe computers, today we see single-chip switches. Just as single-chip processors led to processors replacing logic in a surprising number of places, single-chip switches are increasingly replacing buses and shared media networks.

Connection-Oriented versus Connectionless Communication

Before computers arrived on the scene, the telecommunications industry allowed communication around the world. An operator set up a *connection* between a caller and a callee, and once the connection is established, a conversation can continue for hours. To share transmission lines over long distances, the telecommunications industry used switches to multiplex several conversations on the same lines. Since audio transmissions have relatively low bandwidth, the solution was to divide the bandwidth of the transmission line into a fixed number of frequencies, with each frequency assigned to a conversation. This technique is called *frequency-division multiplexing*.

Although a good match for voice, frequency-division multiplexing is inefficient for sending data. The problem is that the frequency channel is dedicated to the conversation whether or not there is anything being said. Hence, the long distance lines are "busy" based on the *number* of conversations, and not on the *amount* of information being sent at a particular time. An alternative style of communication is called *connectionless*, where each package is routed to the destination by looking at its address. The postal system is a good example of connectionless communication.

Closely related to the idea of connection versus connectionless communication are the terms *circuit switching* and *packet switching*. Circuit switching is the traditional way to offer a connection-based service. A circuit is established from source to destination to carry the conversation, reserving bandwidth until the circuit is broken. The alternative to circuit-switched transmission is to divide the information into *packets,* or *frames*, with each packet including the destination of the packet plus a portion of the information. Queuing theory in section 6.4 tells us that packets cannot use all of the bandwidth, but in general, this *packet-switched* approach allows more use of the bandwidth of the medium and is the traditional way to support connectionless communication.

EXAMPLE Let's compare a single 1000 Mbits/sec packet switched network with ten 100 Mbits/sec packet-switched networks. Assume that the mean size of a packet is 250 bytes, the arrival rate is 250,000 packets per second, and the interarrival times are exponentially distributed. What is the mean response time for each alternative? What is the intuitive reason behind the difference?

ANSWER From section 6.4 in the prior chapter, we can use an M/M/1 queue to calculate the mean response time for the single fast network:

$$\text{Service rate} = \frac{\text{Bandwidth}}{\text{Message size}} = \frac{1000 \times 10^6}{250 \times 8} = \frac{1000 \times 10^6}{2000} = 500{,}000 \text{ packets per second}$$

$$\text{Time}_{\text{server}} = \frac{1}{500{,}000} = 2 \text{ µsecs}$$

$$\text{Utilization} = \frac{\text{Arrival rate}}{\text{Service rate}} = \frac{250{,}000}{500{,}000} = 0.5$$

$$\text{Time}_{\text{queue}} = \text{Time}_{\text{server}} \times \frac{\text{Server utilization}}{(1 - \text{Server utilization})} = 2 \text{ µsecs} \times \frac{0.5}{1 - 0.5} = 2 \times \frac{0.5}{0.5} = 2 \text{ µsecs}$$

$$\text{Mean response time} = \text{Time}_{\text{queue}} + \text{Time}_{\text{server}} = 2 + 2 = 4 \text{ µsecs}$$

The 10 slow networks can be modeled by an M/M/m queue, and the appropriate formulas are found in section 6.7:

$$\text{Service rate} = \frac{100 \times 10^6}{250 \times 8} = \frac{100 \times 10^6}{2000} = 50{,}000 \text{ packets per second}$$

$$\text{Time}_{\text{server}} = \frac{1}{50{,}000} = 0.00002 \text{ secs} = 20 \text{ µsecs}$$

$$\text{Utilization} = \frac{\text{Arrival rate}}{m \times \text{Service rate}} = \frac{250{,}000}{10 \times 50000} = \frac{250{,}000}{500{,}000} = 0.5$$

$$\text{Time}_{\text{queue}} = \text{Time}_{\text{server}} \times \frac{\text{Server utilization}}{m \times (1 - \text{Server utilization})} = 20 \text{ µsecs} \times \frac{0.5}{10 \times (1 - 0.5)} = 2 \times \frac{0.5}{0.5} = 2 \text{ µsecs}$$

$$\text{Mean response time} = \text{Time}_{\text{queue}} + \text{Time}_{\text{server}} = 2 + 20 = 22 \text{ µsecs}$$

The intuition is clear from the results: the service time is much less for the faster network even though the queuing times are the same. This intuition is the argument for "statistical multiplexing" using packets; queuing times are not worse for a single faster network, and the latency for a single packet is much less. Stated alternatively, you get better latency when you use an unloaded fast network, and data traffic is bursty so it works. n

Although connections traditionally align with circuit switching, providing the user with the appearance of a logical connection on top of a packet-switched network is certainly possible. TCP/IP, as we shall see in section 8.8, is a connection-oriented service that operates over packet-switched networks.

Routing: Delivering Messages

Given that the path between nodes may be difficult to navigate depending upon the topology, the system must be able to route the message to the desired node. Shared media has a simple solution: The message is broadcast to *all* nodes that share the media, and each node looks at an address within the message to see whether the message is for that node. This routing also made it easy to broadcast one message to all nodes by reserving one address for everyone; broadcast is much harder to support in switch-based networks.

Switched media use three solutions for routing. In *source-based routing,* the message specifies the path to the destination. Since the network merely follows directions, it can be simpler. A second alternative is the *virtual circuit*, whereby a circuit is established between source and destination, and the message simply names the circuit to follow. ATM uses virtual circuits. The third approach is a *destination-based routing*, where the message merely contains a destination address, and the switch must pick a path to deliver the message. IP uses destination routing. Hence, ATM switches are simpler conceptually; once a virtual circuit is established, packet switching is very fast. On the other hand, IP routers must decide how to route every packet it receives by doing a routing table lookup on every packet.

Destination-based routing may be *deterministic* and always follow the same path, or it may be *adaptive*, allowing the network to pick different routes to avoid failures or congestion. Closely related to adaptive routing is *randomized routing,* whereby the network will randomly pick between several equally good paths to spread the traffic throughout the network, thereby avoiding hot spots.

Switches in WANs route messages using a *store-and-forward* policy; each switch waits for the full message to arrive in the switch before it is sent on to the next switch. Generally store-and-forward can retry a message within the network in case of failure. The alternative to store-and-forward, available in some SANs, is for the switch to examine the header, decide where to send the message, and then start transmitting it immediately without waiting for the rest of the message. It requires retransmission from the source on a failure within the network.

This alternative is called either *cut-through* routing or *wormhole* routing. In wormhole routing, when the head of the message is blocked, the message stays strung out over the network, potentially blocking other messages. Cut-through routing lets the tail continue when the head is blocked, compressing the strung-out message into a single switch. Clearly, cut-through routing requires a buffer large enough to hold the largest packet, while wormhole routing needs only to buffer the piece of the packet sent between switches.

The advantage of both cut-through and wormhole routing over store-and-forward is that latency reduces from a function of the number of intermediate switches *multiplied* by the size of the packet to the time for the first part of the packet to negotiate the switches *plus* the transmission time.

EXAMPLE The CM-5 supercomputer used wormhole routing, with each switch buffer being just 4 bits per port. Compare efficiency of store-and-forward versus wormhole routing for a 128-node machine using a CM-5 interconnection sending a 16-byte payload. Assume each switch takes 0.25 microseconds and that the transfer rate is 20 MBytes/sec.

ANSWER The CM-5 interconnection for 128 nodes is hierarchy (see Figure 8.14 on page 595), and a message goes through seven intermediate switches. Each CM-5 packet has four bytes of header information, so the length of this packet is 20 bytes. The time to transfer 20 bytes over one CM-5 link is

$$\frac{20}{20 \text{ MB/sec}} = 1 \text{ µsec}$$

Then the time for store and forward is

$$(\text{Switches} \times \text{Switch delay}) + ((\text{Switches} + 1) \times \text{Transfer time}) = (7 \times 0.25) + (8 \times 1) = 9.75 \text{ µsecs}$$

while wormhole routing is

$$(\text{Switches} \times \text{Switch delay}) + \text{Transfer time} = (7 \times 0.25) + 1 = 2.75 \text{ µsecs}$$

For this example, wormhole routing improves latency by more than a factor of three. n

A final routing issue is the order in which packets arrive. Some networks require that packets arrive in the order sent. The alternative removes this restriction, requiring software to reassemble the packets in proper order.

Congestion Control

One advantage of a circuit-switched network is that once a circuit is established, it ensures there is sufficient bandwidth to deliver all the information sent along that circuit. Moreover, switches along a path can be requested to give specific quality of service guarantees. Thus, interconnection bandwidth is reserved as circuits are established rather than consumed as data are sent, and if the network is full, no more circuits can be established. You may have encountered this blockage when trying to place a long distance phone call on a popular holiday or to a television show, as the telephone system tells you that "all circuits are busy" and asks you to please call back at a later time.

Packet-switched networks generally do not reserve interconnect bandwidth in advance, so the interconnection network can become clogged with too many packets. Just as with rush hour traffic, a traffic jam of packets increases packet latency. Packets take longer to arrive, and in extreme cases fewer packets per second are delivered by the interconnect, just as is the case for the poor rush-hour commuters. There is even the computer equivalent of gridlock: *deadlock* is

achieved when packets in the interconnect can make no forward progress no matter what sequence of events happens. Chapter 6 addresses how to avoid this ultimate congestion in the context of a multiprocessor.

Higher bandwidth and longer distance networks exacerbate these problems, as this example illustrates.

EXAMPLE

Assume a 155 Mbits/sec network stretching from San Francisco to New York City. How many bytes will be in flight? What is the number if the network is upgraded to 1000 Mbits/sec?

ANSWER

Use the prior assumptions and speed of light. The distance between San Francisco and New York City is 4120 km. Calculating time of flight:

$$\text{Time of flight} = \frac{4120 \text{ km}}{2/3 \times 300,000 \text{ km/sec}} = 0.0206 \text{ secs}$$

Let's assume the network delivers 50% of the peak bandwidth. The number of bytes in transit on a 155 Mbits/sec network is

$$\begin{aligned}\text{Bytes in transit} &= \text{Delivered bandwidth} \times \text{Time of Flight} \\ &= \frac{0.5 \times 155 \text{ Mbits/sec}}{8} \times 0.0206 \text{ secs} = 9.7 \text{ MB/sec} \times 0.0206 \text{ secs} \\ &= 0.200 \text{MB}\end{aligned}$$

At 1000 Mbits/sec the number is

$$\begin{aligned}\text{Bytes in transit} &= \frac{0.5 \times 1000 \text{ Mbits/sec}}{8} \times 0.0206 \text{ secs} = 62.5 \text{ MB/sec} \times 0.0206 \text{ secs} \\ &= 1.718 \text{MB}\end{aligned}$$

More than a megabyte of messages will be a challenge to control and to store in the network. n

The solution to congestion is to prevent new packets from entering the network until traffic reduces, just as metering lights guarding on-ramps control the rate of cars entering a freeway. There are three basic schemes used for congestion control in computer interconnection networks, each with its own weaknesses: packet discarding, flow control, and choke packets.

The simplest, and most callous, is *packet discarding*. If a packet arrives at a switch and there is no room in the buffer, the packet is discarded. This scheme relies on higher-level software that handles errors in transmission to resend lost packets. Internetworking protocols such as UDP discard packets.

The second scheme is to rely on *flow control* between pairs of receivers and senders. The idea is to use feedback to tell the sender when it is allowed to send the next packet. One version of feedback is via separate wires between adjacent senders and receivers that tell the sender to stop immediately when the receiver cannot accept another message. This *backpressure* feedback is rapidly sent back to the original sender over dedicated lines, causing all links between the two end points to be frozen until the receiver can make room for the next message. Backpressure flow control is common in supercomputer networks, SANs and even some gigabit Ethernet switches which send fake collision signal to control flow.

A more sophisticated variation of feedback is for the ultimate destination to give the original sender a credit to send n packets before getting permission to send more. These are generically called *credit-based flow control*. A *window* is one version of credit-based flow control. The window's size determines the minimum frequency of communication from receiver to sender. The goal of the window is to send enough packets to overlap the latency of the interconnection with the overhead to send and receive a packet. The TCP protocol uses a window.

This brings us to a point of confusion on terminology in many papers and textbooks. Note that flow control describes just two nodes of the interconnection and not the total interconnection network between all end systems. *Congestion control* refers to schemes that reduce traffic when the collective traffic of all nodes is too large for the network to handle. Hence, flow control helps congestion control, but it is not a universal solution.

Choke packets are basis of the third scheme. The observation is that you only want to limit traffic when the network is congested. The idea is for each switch to see how busy it is, entering a warning state when it passes a threshold. Each packet received by the switch in a warning state are sent back to the source via a choke packet that includes the intended destination. The source is expected to reduce traffic to that destination by a fixed percentage. Since it likely will have already sent many packets along that path, it waits for all the packets in transit to be returned before taking choke packets seriously.

8.5 Network Topology

The number of topologies described in publications would be difficult to count, but the number that have been used commercially is just a handful, with designers of parallel supercomputers being the most visible and imaginative. They have used regular topologies to simplify packaging and scalability. The topologies of SANS, LANs and WANs are more haphazard, having more to do with the challenges of long distance or simply the connection of equipment purchased over several years. Topology matters less today than it did in the past. You don't want to rewrite your application for each new topology, but you would like the system to take advantage of locality that naturally occurs in programs.

Centralized Switch

Figure 8.13 illustrates two of the popular switch organizations, with the path from node P_0 to node P_6 shown in gray in each topology. A fully connected, or *crossbar,* interconnection allows any node to communicate with any other node in one pass through the interconnection. Routing depends on the style of addressing. In source-based routing, the message includes a sequence of out-bound arcs to reach a destination. Once an outgoing arc is picked, that portion of the routing

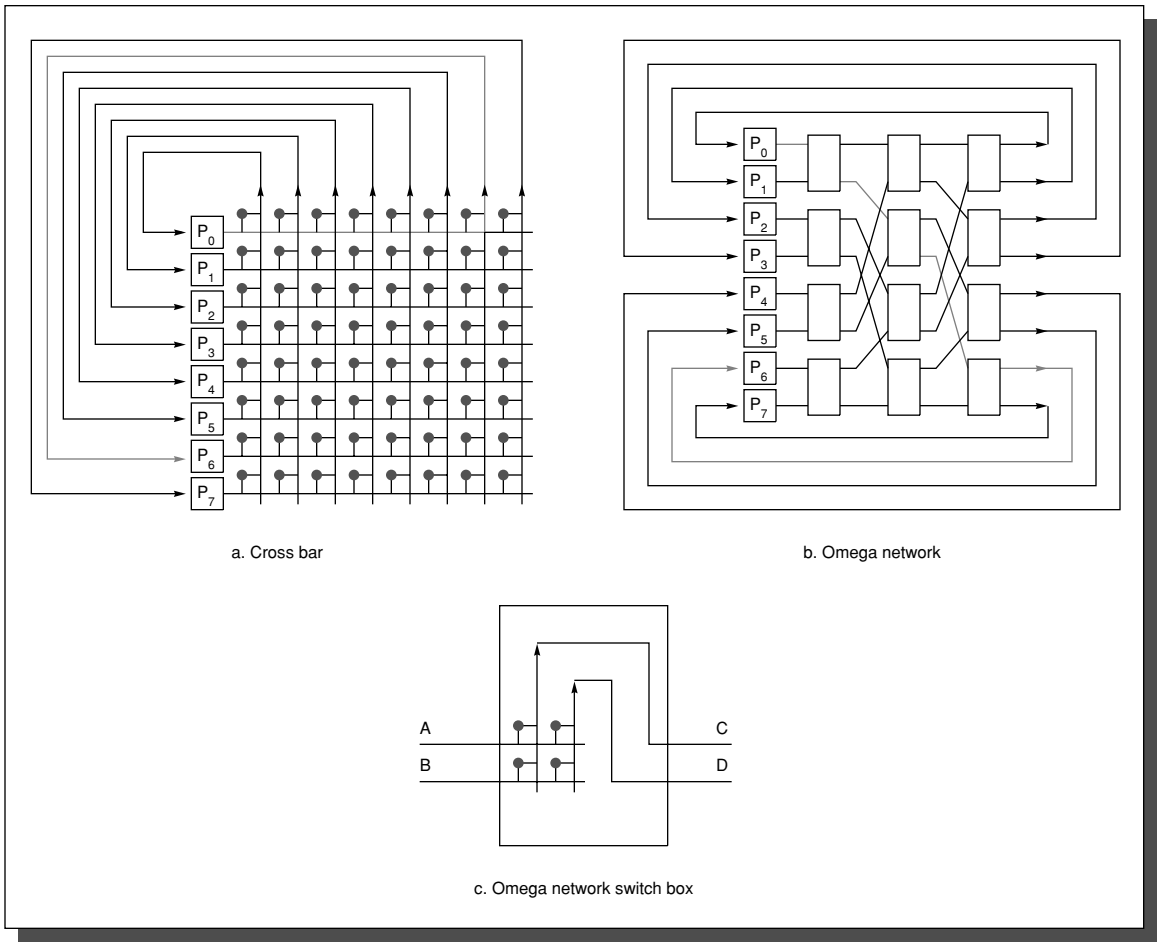


FIGURE 8.13 Popular switch topologies for eight nodes. The links are unidirectional; data come in at the left and exit out the right link. The switch box in (c) can pass A to C and B to D or B to C and A to D. The crossbar uses n^2 switches, where n is the number of processors, while the Omega network uses $n/2 \log_2 n$ of the large switch boxes, each of which is logically composed of four of the smaller switches. In this case the crossbar uses 64 switches versus 12 switch boxes or 48 switches in the Omega network. The crossbar, however, can simultaneously route any permutation of traffic pattern between processors. The Omega network cannot.

sequence may be dropped from the packet. In destination-based routing, a table decides which port to take for a given address. Some networks will run programs in the switches ("spanning tree protocols") to generate the routing table on the fly once the network is connected. The Internet does something similar for routing.

An *Omega* interconnection uses less hardware than the crossbar interconnection ($n/2 \log_2 n$ vs. n^2 switches), but contention is more likely to occur between messages. The amount of contention depends on the pattern of communication. This form of contention is called *blocking*. For example, in the Omega interconnection in Figure 8.13 a message from P_1 to P_7 blocks while waiting for a message from P_0 to P_6. Of course, if two nodes try to send to the same destination—both P_0 and P_1 send to P_6—there will be contention for that link, even in the crossbar.Routing in an Omega net can uses the same techniques as in a full-crossbar.

A tree is the basis of another switch, with bandwidth added higher in the tree to match the requirements of common communications patterns. Figure 8.14 shows this topology, called a *fat tree*. Interconnections are normally drawn as graphs, with each arc of the graph representing a link of the communication interconnection, with nodes shown as black squares and switches shown as shaded circles.

To double the number of nodes in a fat tree, we just add another level to the top of the tree. Notices that this also increases the bandwidth at the top of the tree, which is an advantage of a fat tree.

This figure shows that there are multiple paths between any two nodes in a fat tree. For example, between node 0 and node 8 there are four paths. Such redundancy can help with fault tolerance. In addition, if messages are randomly assigned to different paths, then this should spread the load throughout the switch and result in fewer congestion problems.

Thus far, the switch is separate from the processor and memory, and assumed to be located in a central location. Looking inside this switch, we see many smaller switches. The term *multistage switch* is sometimes used to refer to centralized units to reflect the multiple steps that a message may travel before it reaches a computer.

Distributed Switch

Instead of centralizing these small switching elements, an alternative is to place one small switch at every computer, yielding a distributed switching function.

Given a distributed switch, the question is how to connect the switches together. Figure 8.15 shows that a low-cost alternative to full interconnection is a network that connects a sequence of nodes together. This topology is called a *ring*. Since some nodes are not directly connected, some messages will have to hop along intermediate nodes until they arrive at the final destination. Unlike shared lines, a ring is capable of many simultaneous transfers: the first node can send to the second at the same time as the third node can send to the fourth, for example. Rings

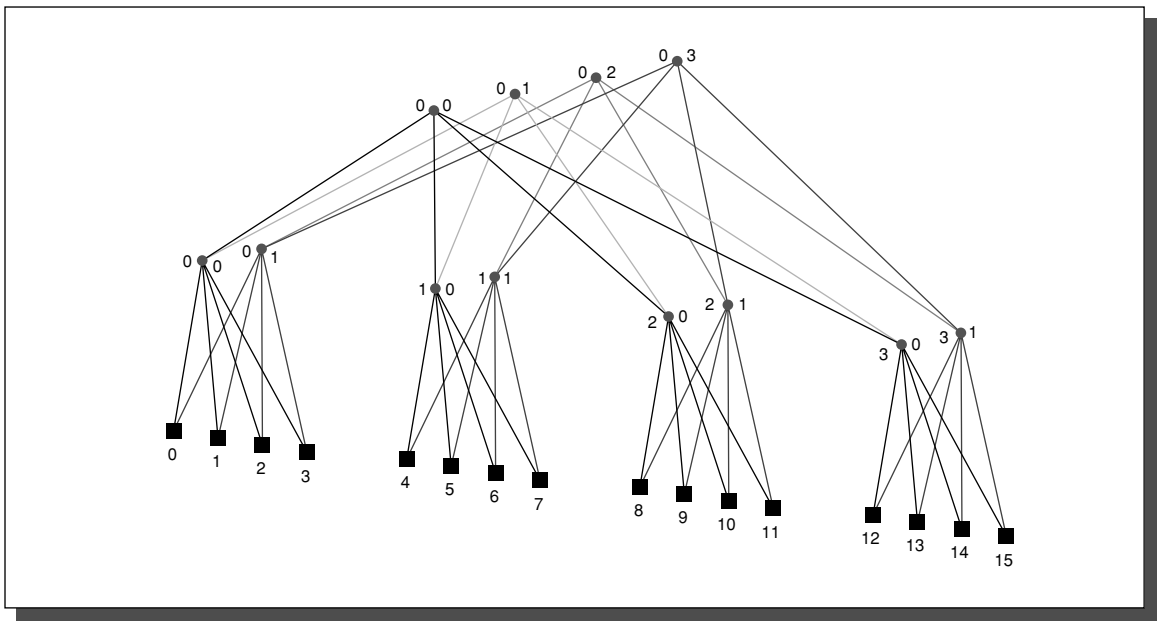


FIGURE 8.14 A fat-tree topology for 16 nodes. The shaded circles are switches, and the squares at the bottom are processor-memory nodes. A simple 4-ary tree would only have the links at the front of the figure; that is, the tree with the root labeled 0,0. This three-dimensional view suggests the increase in bandwidth via extra links at each level over a simple tree, so bandwidth between each level of a fat tree is normally constant rather than reduced by a factor of four as in a 4-ary tree. Multiple paths and random routing give it the ability to route common patterns well, which ensures no single pattern from a broad class of communication patterns will do badly. In the CM-5 fat-tree implementation, the switches have four downward connections and two or four upward connections; in this figure the switches have two upward connections.

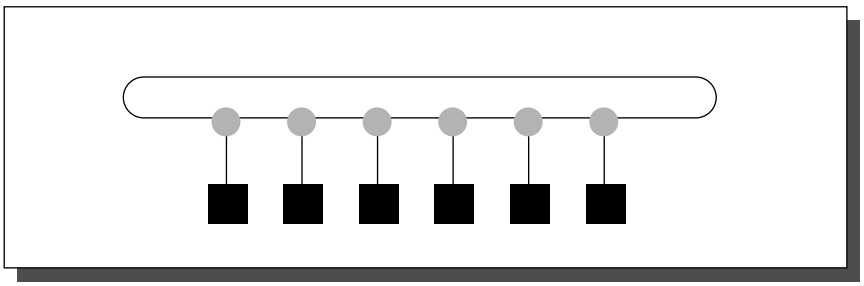


FIGURE 8.15 A ring network topology.

are not quite as good as this sounds because the average message must travel through $n/2$ switches, where n is the number of nodes. To first order, a ring is like a pipelined bus: on the plus side are point-to-point links, and on the minus side are "bus repeater" delays.

One variation of rings used in local area networks is the *token ring*. To simplify arbitration, a single slot, or *token,* goes around the ring to determine which node is allowed to send a message. A node can send only when it gets the token. (A token is simply a special bit pattern.) In this section we evaluate the ring as a topology with more bandwidth than a bus, neglecting its advantages in arbitration.

A straightforward but expensive alternative to a ring is to have a dedicated communication link between every element of a distributed switch. The tremendous improvement in performance of fully connected switches is offset by the enormous increase in cost, typically going up with the square of the number of nodes. This cost inspires designers to invent new topologies that are between the cost of rings and the performance of fully connected networks. The evaluation of success depends in large part on the nature of the communication in the interconnection network. Real machines frequently add extra links to these simple topologies to improve performance and reliability. Figure 8.16 illustrates three popular topologies for high performance computers with distributed switches.

One popular measure for interconnections, in addition to the ones covered in section 8.2, is the *bisection bandwidth*. This measure is calculated by dividing the interconnect into two roughly equal parts, each with half the nodes. You then sum the bandwidth of the lines that cross that imaginary dividing line. For fully connected interconnections the bisection bandwidth is proportional to $(n/2)^2$, where n is the number of nodes. For a bus, bisection bandwidth is just the speed of one link.

Since some interconnections are not symmetric, the question arises as to where to draw the imaginary line when bisecting the interconnect. Bisection bandwidth is a worst-case metric, so the answer is to choose the division that makes interconnection performance worst. Stated alternatively, calculate bisection bandwidths for all pairs of equal-sized parts, and pick the smallest. Figure 8.17 summarizes these different topologies using bisection bandwidth and the number of links for 64 nodes.

EXAMPLE A common communication pattern in scientific programs is to consider the nodes as elements of a two-dimensional array and then have communication to the nearest neighbor in a given direction. (This pattern is sometimes called NEWS communication, standing for north, east, west, and south, the directions on the compass.) Map an eight-by-eight array onto the 64 nodes in each topology, and assume every link of every interconnect is the same speed. How long does it take each node to send one message to its northern neighbor and one to its eastern neighbor? Ignore nodes that have no northern or eastern neighbors.

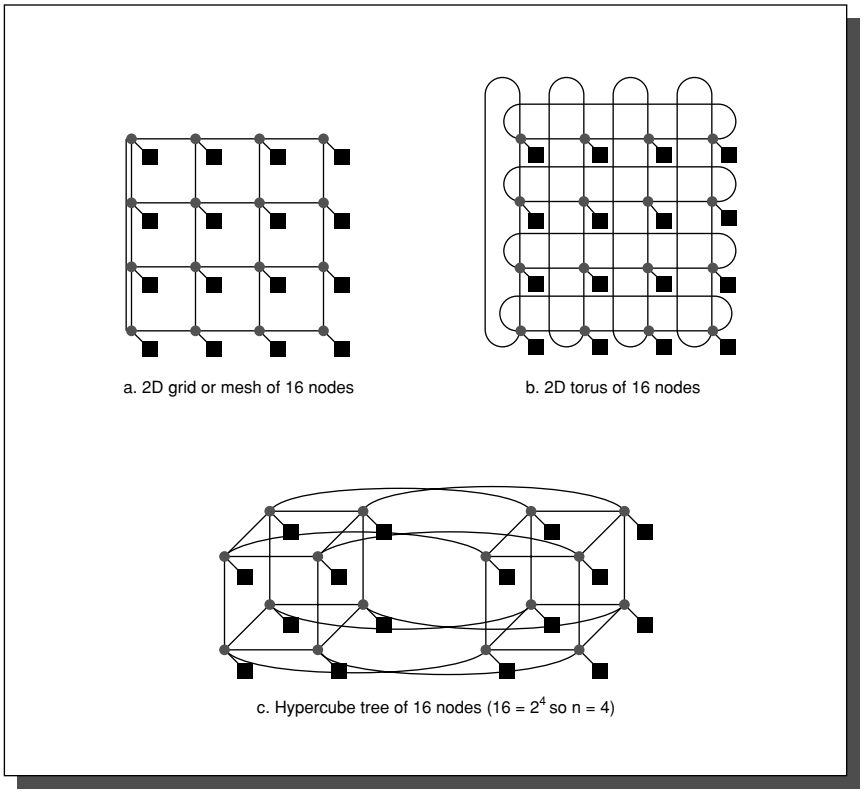


FIGURE 8.16 Network topologies that have appeared in commercial supercomputers. The shaded circles represent switches, and the black squares represent nodes. Even though a switch has many links, generally only one goes to the node. Frequently these basic topologies are supplemented with extra arcs to improve performance and reliability. For example, connecting the switches in the left and right columns of the 2D grid using the unused ports on each switch forms a 2D torus. The Boolean hypercube topology is an *n*-dimensional interconnect for 2^n nodes, requiring *n* ports per switch (plus one for the processor), and thus *n* nearest neighbor nodes.

ANSWER In this case, we want to send $2 \times (64 - 8)$, or 112, messages. Here are the cases, again in increasing order of difficulty of explanation:

- *Bus*—The placement of the eight-by-eight array makes no difference for the bus, since all nodes are equally distant. The 112 transfers are done sequentially, taking 112 time units.
- *Fully connected*—Again the nodes are equally distant; all transfers are done in parallel, taking one time unit.
- *Ring*—Here the nodes are differing distances. Assume the first row

Evaluation category	Bus	Ring	2D torus	6-cube	Fully connected
Performance					
Bisection bandwidth	1	2	16	32	1024
Cost					
Ports per switch	NA	3	5	7	64
Total number of lines	1	128	192	256	2080

FIGURE 8.17 Relative cost and performance of several interconnects for 64 nodes. The bus is the standard reference at unit cost, and of course there can be more than one data line along each link between nodes. Note that any network topology that scales the bisection bandwidth linearly must scale the number of interconnection lines faster than linearly. Figure 8.13a on page 593 is an example of a fully connected network.

of the array is placed on nodes 0 to 7, the second row on nodes 8 to 15, and so on. It takes just one time unit to send to the eastern neighbor, for this is a send from node n to node $n + 1$. In this scheme the northern neighbor is exactly eight nodes away, so it takes eight time units for each node to send to its northern neighbor. The ring total is nine time units.

- *2D torus*—There are eight rows and eight columns in our grid of 64 nodes, which is a perfect match to the NEWS communication. It takes just two time units to send to the northern and eastern neighbors.
- *6-cube*—It is possible to place the array so that it will take just two time units for this communication pattern, as in the case of the torus.

This simple analysis of interconnection networks in this section ignores several important practical considerations in the construction of an interconnection network. First, these three-dimensional drawings must be mapped onto chips, boards, and cabinets that are essentially two-dimensional media, often tree-like. For example, due to the fixed height of cabinets, an n-node Intel Paragon used an $n/16 \times 16$ rectangular grid rather than the ideal of $\sqrt{n} \times \sqrt{n}$. Another consideration is the internal speed of the switch: if it is fixed, then more links per switch means lower bandwidth per link, potentially affecting the desirability of different topologies. Yet another consideration is that the latency through a switch depends on the complexity of the routing pattern, which in turn depends on the topology. Finally, the bandwidth from the processor is often the limiting factor: if there is only one port in and out of the processor, then it can only send or receive one message per time unit regardless of the technology.

Topologies that appear elegant when sketched on the blackboard may look awkward when constructed from chips, cables, boards, and boxes. The bottom

line is that quality of implementation matters more than topology. To put these topologies in perspective, Figure 8.18 lists those used in commercial high performance computers.

Institution	Name	Number of nodes	Basic topology	Data bits/link	Network clock rate	Peak BW/link (MB/sec)	Bisection (MB/sec)	Year
Thinking Machines	CM-2	1024 to 4096	12-cube	1	7 MHz	1	1024	1987
Intel	Delta	540	2D grid	16	40 MHz	40	640	1991
Thinking Machines	CM-5	32 to 2048	Multistage fat tree	4	40 MHz	20	10,240	1991
Intel	Paragon	4 to 2048	2D grid	16	100 MHz	175	6400	1992
IBM	SP-2	2 to 512	Multistage fat tree	8	40 MHz	40	20,480	1993
Cray Research	T3E	16 to 2048	3D torus	16	300? MHz	600	122,000	1997
Intel	ASCI Red	4536 (x 2 CPUS)	2D Grid			800	51,600	1996
IBM	ASCI Blue Pacific	1336 (x 4 CPUS)				150		
SGI	ASCI Blue Mountain	1464 (x 2 CPUS)	Fat Hyper-cube			800	200 x nodes	1998
IBM	ASCI Blue Horizon	144 (x 8 CPUs)	Multistage Omega			115		1999
IBM	SP	1 to 512 (x 2 to 16 CPUs)	Multistage Omega			500		2000
IBM	ASCI White	484 (x 16 CPUs)	Multistage Omega			500		2001

FIGURE 8.18 Characteristics of interconnections of some commercial supercomputers. The bisection bandwidth is for the largest machine. The 2D grid of the Intel Delta is 16 rows by 35 columns and the ASCI Red is 38 rows by 32 columns. The fat-tree topology of the CM-5 is restricted in the lower two levels, hence the lower bandwidth in the bisection. Note that the Cray T3D has two processors per node and the Intel Paragon has from two to four processors per node.

Once again the issues discussed in this section apply at many levels, from inside a chip to a country-sized WAN. The redundancy of a topology matter so that the network can survive despite failures. This is true within a switch as well, so that a single chip failure need not lead to switch failure. It also must be true for a WAN, so that a single backhoe cannot take down the network of a country. The switch then depends on the implementation technology and the demands of the application: it is a multistage network whose topology can be anything from a bus to Omega network.

8.6 Practical Issues for Commercial Interconnection Networks

There are practical issues in addition to the technical issues described so far that are important considerations for some interconnection networks: connectivity, standardization, and fault tolerance.

Connectivity

The number of machines that communication affects the complexity of the network and its protocols. The protocols must target the largest size of the network, and handle the types of anomalous events that occur. Hundreds of machines communicating are a much easier than millions.

Connecting the Network to the Computer

Where the network attaches to the computer affects both the network interface hardware and software. Questions include whether to use the memory bus or the I/O bus, whether to use polling or interrupts, and how to avoid invoking the operating system. The network interface is the often the network bottleneck.

Computers have a hierarchy of buses with different cost/performance. For example, a personal computer in 2001 has a memory bus, a PCI bus for fast I/O devices, and an USB bus for slow I/O devices. I/O buses follow open standards and have less stringent electrical requirements. Memory buses, on the other hand, provide higher bandwidth and lower latency than I/O buses. Where to connect the network to the machine depends on the performance goals and whether you hope to buy a standard network interface card or are willing to design or buy one that only works with the memory bus on your model of computer. A few SAN plugs into the memory bus, but most SANs and all LANs and WANs plug into the I/O bus.

The location of the network connection significantly affects the software interface to the network as well as the hardware. As mentioned in section 6.6, one key is whether the interface is coherent with the processor's caches: the sender may have to flush the cache before each send, and the receiver may have to flush its cache before each receive to prevent the stale data problem. Such flushes increase send and receive overhead. A memory bus is more likely to be cache-coherent than an I/O bus and therefore more likely to avoid these extra cache flushes.

A related question of where to connect to the computer is how to connect to the software: Do you use programmed I/O or direct memory access (DMA) to send a message? (See section 6.6.) In general, DMA is the best way to send large

messages. Whether to use DMA to send small messages depends on the efficiency of the interface to the DMA. The DMA interface is usually memory-mapped, and so each interaction is typically at the speed of main memory rather than of a cache access. If DMA setup takes many accesses, each running at uncached memory speeds, then the sender overhead may be so high that it is faster to simply send the data directly to the interface.

Standardization: Cross-Company Interoperability

Standards are useful in many places in computer design, but with interconnection networks they are often critical. Advantages of successful standards include low cost and stability. The customer has many vendors to choose from, which both keeps price close to cost due to competition. It makes the viability of the interconnection independent of the stability of a single company. Components designed for a standard interconnection may also have a larger market, and this higher volume can lower the vendor's costs, further benefiting the customer. Finally, a standard allows many companies to build products with interfaces to the standard, so the customer does not have to wait for a single company to develop interfaces to all the products the customer might be interested in.

One drawback of standards is the time it takes for committees to agree on the definition of standards, which is a problem when technology is changing quickly. Another problem is *when* to standardize: on one hand, designers would like to have a standard before anything is built; on the other, it would be better if something is built before standardization to avoid legislating useless features or omitting important ones. When done too early, it is often done entirely by committee, which is like asking all of the chefs in France to prepare a single dish of food; masterpieces are rarely served. Standards can also suppress innovation at that level, since the standard fixes interfaces.

LANs and WANs use standards and interoperate effectively. WANs involve many types of companies and must connect to many brands of computers, so it is difficult to imagine a proprietary WAN ever being successful. The ubiquitous nature of the Ethernet shows the popularity of standards for LANs as well as WANs, and it seems unlikely that many customers would tie the viability of their LAN to the stability of a single company.

Alas, some SANs are standardized yet switches from different companies do not interoperate, and some interoperate as well as LANs and WANs.

Message Failure Tolerance

Although some hardware designers try to build fault free networks, in practice it is only a question of the rate of faults, not whether you can prevent them. Thus, the communication system must have mechanisms for retransmission of a message in case of failure. Often it is handled in higher layers of the software protocol at the end points, requiring retransmission at the source. Given the long time of flight for WANs, often they can retransmit from hop to hop rather relying only on retransmission from the source.

Node Failure Tolerance

The second practical issue refers to whether or not the interconnection relies on all the nodes being operational in order for the interconnection to work properly. Since software failures are generally much more frequent than hardware failures, the question is whether a software crash on a single node can prevent the rest of the nodes from communicating.

Clearly, WANs would be useless if they demanded that thousands of computers spread across a continent be continuously available, and so they all tolerate the failures of individual nodes. LANs connect dozens to hundreds of computers together, and again it would be impractical to require that no computer ever fail. All successful LANs normally survive node failures.

Although most SANs have the ability to work around failed nodes and switches, it is not clear that all communication layer software supports this feature. Typically, low latency schemes sacrifice fault tolerance.

EXAMPLE Figure 8.19 shows the number of failures of 58 desktop computers on a local area network for a period of just over one year. Suppose that one local area network is based on a network that requires all machines to be operational for the interconnection network to send data; if a node crashes, it cannot accept messages, so the interconnection becomes choked with data waiting to be delivered. An alternative is the traditional local area network, which can operate in the presence of node failures; the interconnection simply discards messages for a node that decides not to accept them. Assuming that you need to have both your workstation and the connecting LAN to get your work done, how much greater are your chances of being prevented from getting your work done using the failure-intolerant LAN versus traditional LANs? Assume the down time for a crash is less than 30 minutes. Calculate using the one-hour intervals from this figure.

ANSWER Assuming the numbers for Figure 8.19, the percentage of hours that you can't get your work done using the failure-intolerant network is

$$\frac{\text{Intervals with failures}}{\text{Total intervals}} = \frac{\text{Total intervals} - \text{Intervals no failures}}{\text{Total intervals}}$$

$$= \frac{8974 - 8605}{8974} = \frac{369}{8974} = 4.1\%$$

The percentage of hours that you can't get your work done using the traditional network is just the time your workstation has crashed. If these failures are equally distributed among workstations, the percentage is

$$\frac{\text{Failures/Machines}}{\text{Total intervals}} = \frac{654/58}{8974} = \frac{11.28}{8974} = 0.13\%$$

Failed machines per time interval	One-hour intervals with number of failed machines in first column	Total failures per one-hour interval	One-day intervals with number of failed machines in first column	Total failures per one-day interval
0	8605	0	184	0
1	264	264	105	105
2	50	100	35	70
3	25	75	11	33
4	10	40	6	24
5	7	35	9	45
6	3	18	6	36
7	1	7	4	28
8	1	8	4	32
9	2	18	2	18
10	2	20		
11	1	11	2	22
12			1	12
17	1	17		
20	1	20		
21	1	21	1	21
31			1	31
38			1	38
58			1	58
Total	**8974**	**654**	**373**	**573**

FIGURE 8.19 Measurement of reboots of 58 DECstation 5000s running Ultrix over a 373-day period. These reboots are distributed into time intervals of one hour and one day. The first column sorts the intervals according to the number of machines that failed in that interval. The next two columns concern one-hour intervals, and the last two columns concern one-day intervals. The second and fourth columns show the number of intervals for each number of failed machines. The third and fifth columns are just the product of the number of failed machines and the number of intervals. For example, there were 50 occurrences of one-hour intervals with two failed machines, for a total of 100 failed machines, and there were 35 days with two failed machines, for a total of 70 failures. As we would expect, the number of failures per interval changes with the size of the interval. For example, the day with 31 failures might include one hour with 11 failures and one hour with 20 failures. The last row shows the total number of each column: the number of failures doesn't agree because multiple reboots of the same machine in the same interval do not result in separate entries. (Randy Wang of U.C. Berkeley collected these data.)

Hence, you are more than 30 times more likely to be prevented from getting your work done with the failure-intolerant LAN than with the traditional LAN, according to the failure statistics in Figure 8.19. Stated alternatively, the person responsible for maintaining the LAN would receive a thirtyfold increase in phone calls from irate users! n

One practical issue ties to node failure tolerance: If the interconnection can survive a failure, can it also continue operation while a new node is added to the interconnection? If not, each addition of a new node disables the interconnection network. Disabling is impractical for both WANs and LANs.

Finally, we have been discussing the ability of the network to operate in the presence of failed nodes. Clearly as important to the happiness of the network administrator is the reliability of the network media and switches themselves, for their failure is certain to frustrate much of the user community.

8.7 Examples of Interconnection Networks

To further understand these issues, we look at ten design decisions on the topics we covered so far using examples from LAN, SAN, and WAN:

- What is the target bandwidth?
- What is the message format?
- Which media are used?
- Is the network shared or switched?
- Is it connection-oriented or connectionless?
- Does it use store-and-forward or cut-through routing?
- Is routing use source-based, destination-based, or virtual-circuit based?
- What is used for congestion control?
- What topologies are supported?
- Does it follow a standard?

Ethernet: *The* Local Area Network

The first example is the Ethernet. It has been extraordinarily successful, with the 10 Mbits/second standard proposed in 1978 used practically everywhere. In 2001, the 100 Mbits/second standard proposed in 1994 is closing in popularity. Many classes of computers include Ethernet as a standard interface. This packet-switched network is connectionless, and it routes using the destination address. Figure 8.20 shows the packet formats for Ethernet, as well as the other two examples. Ethernet is codified as IEEE standard 802.3.

Designed originally for co-axial cable, today Ethernets are primarily Cat5 copper wire, with optical fiber reserved for longer distances and higher bandwidths. There is even a wireless version, which is testimony to its ubiquity.

Computers became thousands of times faster than they were in 1978 and the shared interconnection was no faster for almost 20 years. Hence, past engineers

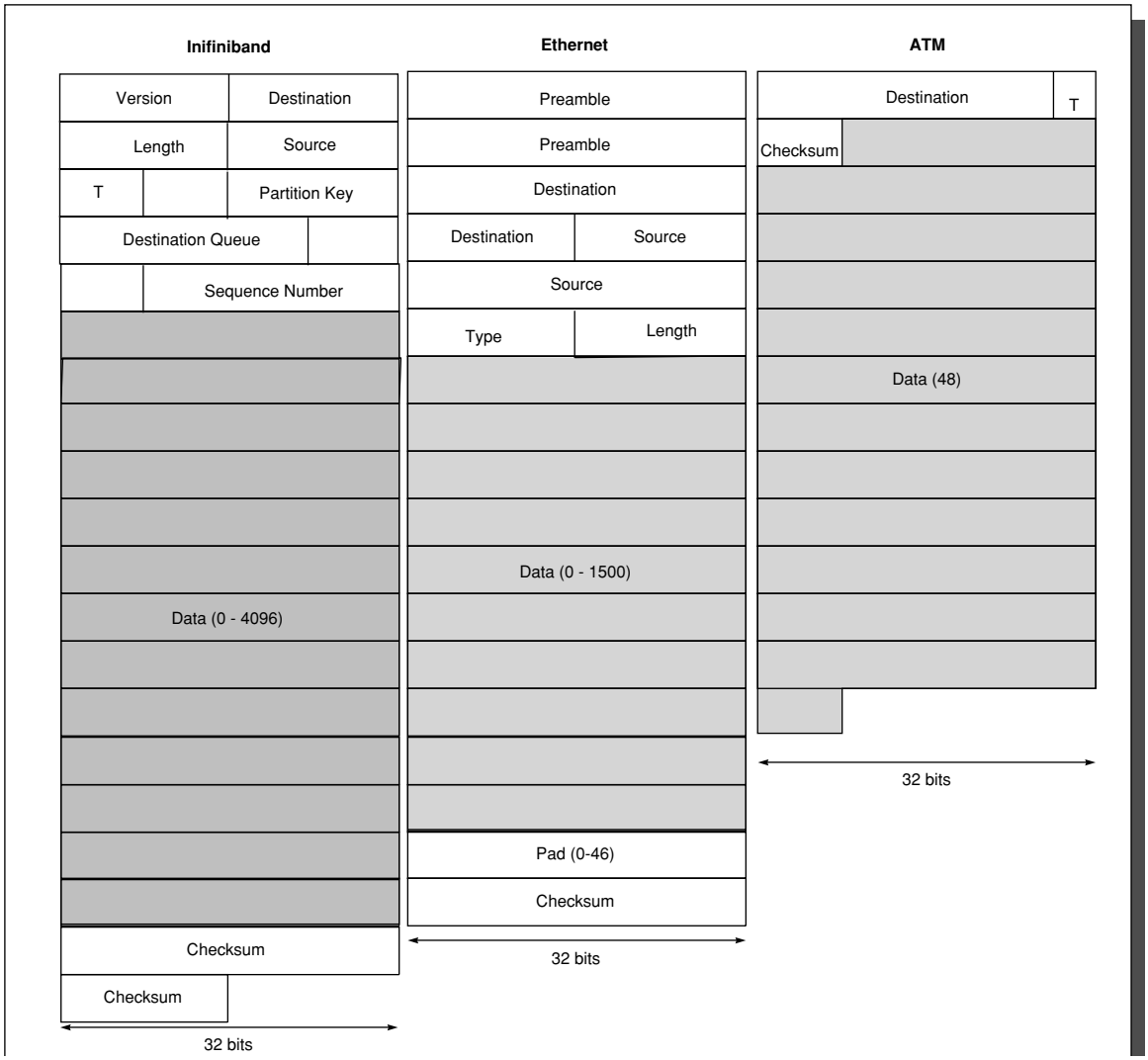


FIGURE 8.20 Packet format for Infiniband, Ethernet, and ATM. ATM calls their messages "cells" instead of packets, so the proper name is ATM cell format. The width of each drawing is 32 bits. All three formats have destination addressing fields, encoded differently for each situation. All three also have a checksum field to catch transmission errors, although the ATM checksum field is calculated only over the header; ATM relies on higher-level protocols to catch errors in the data. Both Infiniband and Ethernet have a length field, since the packets hold a variable amount of data, with the former counted in 32-bit words and the latter in bytes.Infiniband and ATM headers have a type field (T) that gives the type of packet. The remaining Ethernet fields are a preamble to allow the receiver to recover the clock from the self-clocking code used on the Ethernet, the source address, and a pad field to make sure the smallest packet is 64 bytes (including the header). Infiniband includes a version field for protocol version, a sequence number to allow in-order delivery, a field to select the destination queue, and a partition key field.Infiniband has many more small fields not shown and many other packet formats; above is a simplified view. ATM's short packet, fixed is a good match to real-time demand of digital voice

invented temporary solutions until a faster Ethernet was available. One solution was to use multiple Ethernets to connect machines, and to connect these smaller Ethernets with devices that can take traffic from one Ethernet and pass it on to another as needed. These devices allow individual Ethernets to operate in parallel, thereby increasing the aggregate interconnection bandwidth of a collection of computers. In effect these devices provide similar functionality to the switches described above for point-to-point networks.

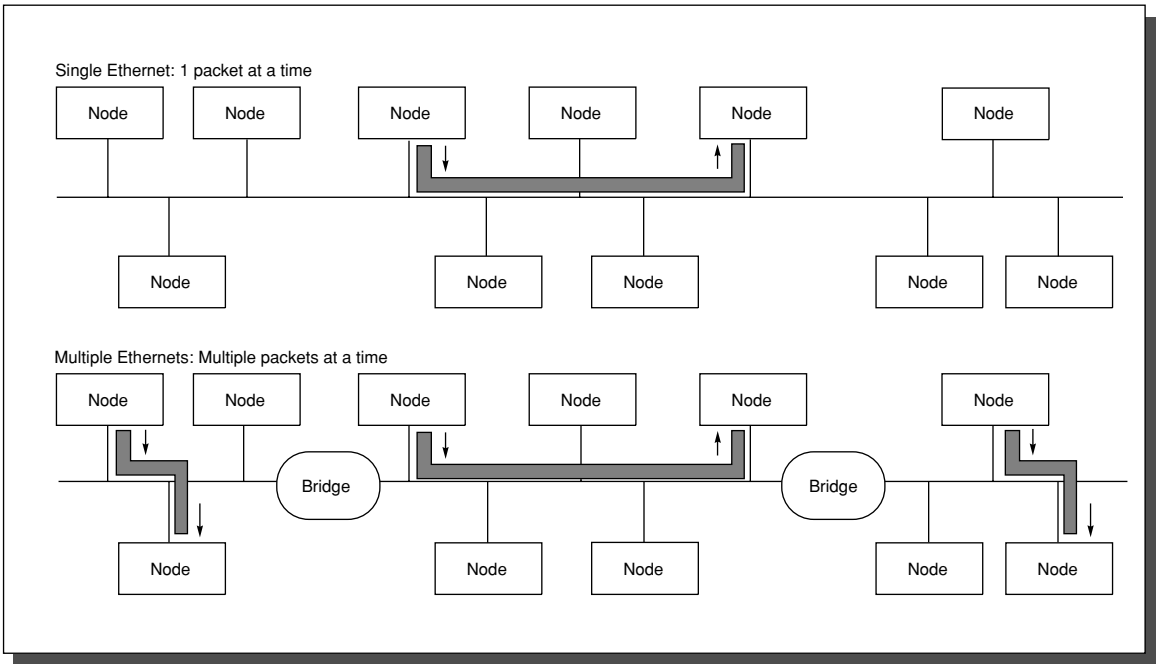


FIGURE 8.21 The potential increased bandwidth of using many Ethernets and bridges.

Figure 8.21 shows the potential parallelism. Depending on how they pass traffic and what kinds of interconnections they can put together, these devices have different names:

- *Bridges*—These devices connect LANs together, passing traffic from one side to another depending on the addresses in the packet. Bridges operate at the Ethernet protocol level and are usually simpler and cheaper than routers, discussed next. Using the notation of the OSI model described in the next section (see Figure 8.25 on page 612), bridges operate at layer 2, the data link layer.
- *Routers* or *gateways*—These devices connect LANs to WANs or WANs to WANs, and resolve incompatible addressing. Generally slower than bridges, they operate at OSI layer 3, the network layer. Routers divide the interconnect

into separate smaller subnets, which simplifies manageability and improves security.

The final network devices are *hubs,* but they merely extend multiple segments into a single LAN. Thus, hubs do not help with performance, as only one message can transmit at a time. Hubs operate at OCI layer 1, the physical layer.

Since these devices were not planned as part of the Ethernet standard, their ad hoc nature has added to the difficulty and cost of maintaining LANs.

In 2001, Ethernet link speed is available at 10, 100, and 1000 Mbits/second, with 10000 Mbits per second likely available in 2002 to 2003. Although 10 and 100 Mbits/sec can share the media with multiple devices, 1000 Mbits/second and above relies on point-to-point links and switches. Ethernet switches normally use cut-through routing.

Due to its age, Ethernet has no real flow control. It originally used carrier sensing with exponential back-off (see page 584) to arbitrate for the shared media. Some switches try to use that interface to retrofit their version of flow control, but flow control is not part of an Ethernet standard.

Storage Area Network: Infiniband

A SAN that tries to optimize based on shorter distances is Infiniband. This new standard has clock rates of 2.5 GHz and can transmit data at a peak speed of 2000 Mbits/second per link. These point-to-point links can be bundled together in groups of 4 to 12 to give 4 to 12 times the bandwidth per link. Like Ethernet, it is a packet switched, connectionless network. It also relies only on switches, as does gigabit Ethernet, and also uses cut-through routing and destination-based addressing. The distances are much shorter than Ethernet, with category 5 wire limited to 17 meters and optical fiber limited to 100 meters. It uses backpressure for flow control (see page 592). When going to storage, it relies on the SCSI command set. Although it is not a traditional standard, a trade organization of cooperating companies is responsible for Infiniband.

Given the similarities, why does one need a separate standard for a storage area network versus a local area network? The storage community believes a SAN has different emphasis from a LAN. First, protocol overhead is much lower for a SAN. A gigabit per second LAN can fully occupy a 0.8 to 1.0 GHz CPU when running TCP/IP (see page 653). The Infiband protocol, on the other hand, places a very light load on the host computer. The reason is a controller on the Infiniband network interface card that offloads the processing from the host comptuer. Second, protection is much more important in the LAN than the SAN. The SAN is for data only, and is behind the server. From a SAN perspective, the server is like a firewall for the SAN, and hence the SAN is not required to provide protection. Third, storage designers think that graceful behavior under congestion is critical for SANs. The lack of flow control in Ethernet can lead to a lack of grace under pressure. TCP/IP copes with congestion by dropping packets, but storage applications do not appreciate dropped packets.

Not surprisingly, the LAN advocates have a response. First, Ethernet switches are less costly than SAN switches due to greater competition in the marketplace. Second, since Internet Protocol (IP) networks are naturally large, they enable replication of data to geographically diverse sites of the Internet. This geographical advantage both protects against disasters and offers an alternative to tape backup. Thus far, SANs have been relatively small, both in number of nodes and physical distance. Finally, although TCP/IP does have overhead, to try to preserve server utilization, TCP/IP off-loading engines are appearing in the marketplace.

Some LAN advocates are embracing a standard called *iSCSI*, which exports native SCSI commands over IP networks. The operating system intercepts SCSI commands, and repackages and sends them in a TCP/IP message. At the receiving end, it unpacks messages into SCSI commands and issues them locally. iSCSI allows a company to send SCSI commands and data over its internal WAN or, if transmitted over the Internet, to locations with Internet access.

Wide Area Network: ATM

Asynchronous Transfer Mode (*ATM*) is latest of the ongoing standards set by the telecommunications industry. Although it flirted as competition to Ethernet as a LAN in the 1990s, today ATM has retreated to its WAN stronghold.

The telecommunications standard has scalable bandwidth built in. It starts at 155 Mbits/second, and scales by factors of four to 620 Mbits/second, 2480 Mbits per second, and so on. Since it is a WAN, ATM's media is fiber, both single mode and multimode. Although it is a switched media, unlike the other examples, it relies on connections for communication. ATM uses virtual channels for routing to multiplex different connections on a single network segment, thereby avoiding the inefficiencies of conventional connection-based networking. The WAN focus also leads to store-and-forward routing. Unlike the other protocols, Figure 8.20 shows ATM has a small, fixed sized packet. (For those curious to the selection of a 48-byte payload, see Section 8.16.) It uses a credit-based flow control scheme (see page 592).

The reason for connections and small packets is quality of service. Since the telecommunications industry is concern about voice traffic, predictability matters as well as bandwidth. Establishing a connection has less variability than connectionless networking, and it simplifies store and forward routing. The small, fixed packet also makes it simpler to have fast routers and switches. Towards that goal, ATM even offers its own protocol stack to compete with TCP/IP. Surprisingly, even though the switches are simple, the ATM suite of protocols is large and complex. The dream was a seamless infrastructure from LAN to WAN, avoiding the hodge-podge of routers common today. That dream has faded from inspiration to nostalgia.

	LAN			SAN			WAN
	10-Mb Ethernet	**100-Mb Ethernet**	**1000-Mb Ethernet**	**FC-AL**	**Infiniband**	**Myrinet**	**ATM**
Length (meters)	500/ 2500	200	100	30/1000	17/100	10/550/ 10000	
Number data lines	1	1	4/1	2	1, 4, or 12	?	1
Clock rate (MHz)	10	100	1000	1000	2500	1000	155/622…
Switch?	Optional	Optional	Yes	Optional	Yes	Yes	Yes
Nodes	≤254	≤254	≤254	≤127	≤≈1000	≤≈1000	≈10000
Media	Copper	Copper	Copper/ fiber	Copper/ fiber	Copper/fiber	Copper/ m.m./ s.m.fiber	Copper/fiber
Peak link BW (Mbits/sec)	10	100	1000	800	2000, 8000, or 24000	1300 to 2000	155/622/...
Topology	Line or Star	Line or Star	Star	Ring or Star	Star	Star	Star
Connec-tionless?	Yes	Yes	Yes	Yes	Yes	Yes	No
Routing	Dest. based	Dest. based	Dest. based	Destina-tion based	Destina-tion based	Dest. based	Virtual circuit
Store & forward?	No	No	No	No	No	No	Yes
Congestion control	Carrier sense	Carrier sense	Carrier sense	Credit-based	Back-pressure	Back-pressure	Credit based
Standard	IEEE 802.3	IEEE 802.3	IEEE 802.3ab-1999	ANSI Task Group X3T11	Infiniband Trade Association	ANSI/ VITA 26-1998	ATM Forum

FIGURE 8.22 Several examples of SAN, LAN, and WAN interconnection networks. FC-AL is a network for disks.

Summary

Figure 8.22 summarizes answers to the ten questions from the start of this section. It covers three example networks covered here, plus a few other. This section shows how similar technology gets different spins for different concerns of LAN, SAN, and WAN. Nevertheless, the inherent similarity leads to marketplace competition. ATM tried (and failed) to usurp the LAN championship from Ethernet, and in 2001 Ethernet/iSCSI is trying to compete with Fibre Channel Arbitrated Loop (FC-AL) and Infiniband for the SAN markets.

.

8.8 Internetworking

Undoubtedly one of the most important innovations in the communications community has been internetworking. It allows computers on independent and incompatible networks to communicate reliably and efficiently. Figure 8.23 illustrates the need to cross networks. It shows the networks and machines involved in transferring a file from Stanford University to the University of California at Berkeley, a distance of about 75 km.

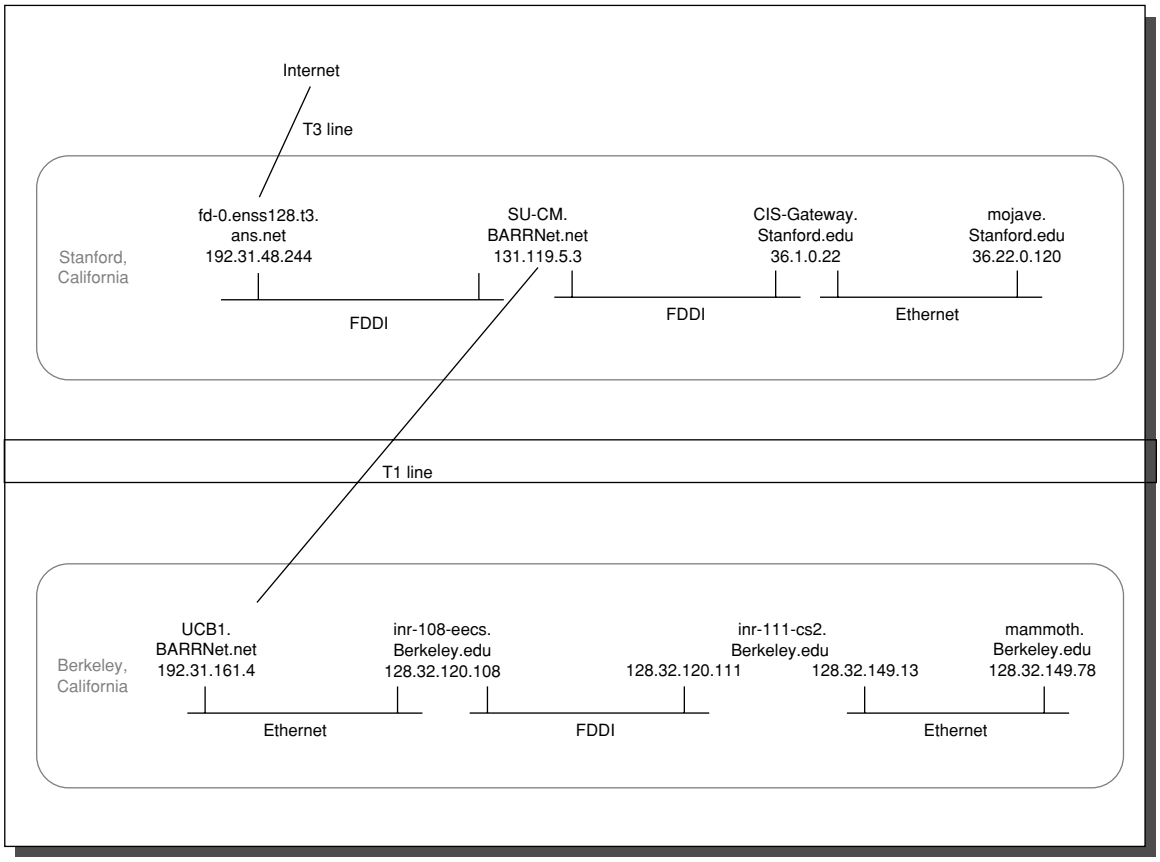


FIGURE 8.23 The connection established between mojave.stanford.edu and mammoth.berkeley.edu. (1995) FDDI is a 100 Mbits/sec LAN, while a T1 line is a 1.5 Mbits/sec telecommunications line and a T3 is a 45 Mbits/sec telecommunications line. BARRNet stands for Bay Area Research Network. Note that inr-111-cs2.Berkeley.edu is a router with two Internet addresses, one for each port.

The low cost of internetworking is remarkable. For example, it is vastly less expensive to send electronic mail than to make a coast-to-coast telephone call and leave a message on an answering machine. This dramatic cost improvement is achieved using the same long-haul communication lines as the telephone call, which makes the improvement even more impressive.

The enabling technologies for internetworking are software standards that allow reliable communication without demanding reliable networks. The underlying principle of these successful standards is that they were composed as a hierarchy of layers, each layer taking responsibility for a portion of the overall communication task. Each computer, network, and switch implements its layer of the standards, relying on the other components to faithfully fulfill their responsibilities. These layered software standards are called *protocol families* or *protocol suites*. They enable applications to work with any interconnection without extra work by the application programmer. Figure 8.24 suggests the hierarchical model of communication.

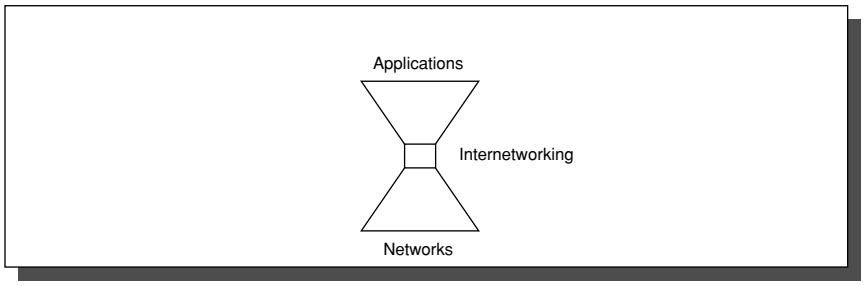


FIGURE 8.24 The role of internetworking. The width indicates the relative number of items at each level.

The most popular internetworking standard is *TCP/IP*, which stands for *transmission control protocol/internet protocol*. This protocol family is the basis of the humbly named *Internet*, which connects tens of millions of computers around the world. This popularity means TCP/IP is used even when communicating locally across compatible networks; for example, the network file system NFS uses IP even though it is very likely to be communicating across a homogenous LAN such as Ethernet.

We use TCP/IP as our protocol family example; other protocol families follow similar lines. Section 8.16 gives the history of TCP/IP.

The goal of a family of protocols is to simplify the standard by dividing responsibilities hierarchically among layers, with each layer offering services needed by the layer above. The application program is at the top, and at the bottom is the physical communication medium, which sends the bits. Just as abstract data types simplify the programmer's task by shielding the programmer from details of the implementation of the data type, this layered strategy makes the standard easier to understand.

There were many efforts at network protocols, which led to confusion in terms. Hence, Open Systems Interconnect (OSI) developed a model that popularized describing networks as a series of layers. Figure 8.25 shows the model. Although all protocols do not exactly follow this layering, the nomenclature for the different layers is widely used. Thus, you can hear discussions about a simple layer 3 switch versus a layer 7 smart switch.

Layer number	Layer name	Main Function	Example Protocol	Network component
7	Application	Used for applications specifically written to run over the network	FTP, DNS, NFS, http	Gateway, smart switch
6	Presentation	Translates from application to network format, and vice-versa		Gateway
5	Session	Establishes, maintains and ends sessions across the network	Named pipes, RPC	Gateway
4	Transport	Additional connection below the session layer	TCP	Gateway
3	Network	Translates logical network address and names to their physical address (e.g., computer name to MAC address)	IP	Router, ATM switch
2	Data Link	Turns packets into raw bits and at the receiving end turns bits into packets	Ethernet	Bridge, Network Interface Card
1	Physical	Transmits raw bit stream over physical cable	IEEE 802	Hub

FIGURE 8.25 The OSI model layers. Based on /www.geocities.com/SiliconValley/Monitor/3131/ne/osimodel.html

The key to protocol families is that communication occurs *logically at the same level* of the protocol in both sender and receiver, but *services of the lower level implement it*. This style of communication is called *peer-to-peer*. As an analogy, imagine that General A needs to send a message to General B on the battlefield. General A writes the message, puts it in an envelope addressed to General B, and gives it to a colonel with orders to deliver it. This colonel puts it in an envelope and writes the name of the corresponding colonel who reports to General B, and gives it to a major with instructions for delivery. The major does the same thing and gives it to a captain, who gives it to a lieutenant, who gives it to a sergeant. The sergeant takes the envelope from the lieutenant, puts it into an envelope with the name of a sergeant who is in General B's division, and finds a private with orders to take the large envelope. The private borrows a motorcycle and

delivers the envelope to the other sergeant. Once it arrives, it is passed up the chain of command, with each person removing an outer envelope with his name on it and passing on the inner envelope to his superior. As far as General B can tell, the note is from another general. Neither general knows who was involved in transmitting the envelope, nor how it was transported from one division to the other.

Protocol families follow this analogy more closely than you might think, as Figure 8.26 shows. The original message includes a header and possibly a trailer sent by the lower-level protocol. The next-lower protocol in turn adds its own header to the message, possibly breaking it up into smaller messages if it is too large for this layer. Reusing our analogy, a long message from the general is divided and placed in several envelopes if it could not fit in one. This division of the message and appending of headers and trailers continues until the message descends to the physical transmission medium. The message is then sent to the destination. Each level of the protocol family on the receiving end will check the message at its level and peel off its headers and trailers, passing it on to the next higher level and putting the pieces back together. This nesting of protocol layers for a specific message is called a *protocol stack*, reflecting the last-in-first-out nature of the addition and removal of headers and trailers.

As in our analogy, the danger in this layered approach is the considerable latency added to message delivery. Clearly, one way to reduce latency is to reduce the number of layers. But keep in mind that protocol families *define* a standard, but do not force how the to *implement* the standard. Just as there are many ways to implement an instruction set architecture, there are many ways to implement a protocol family.

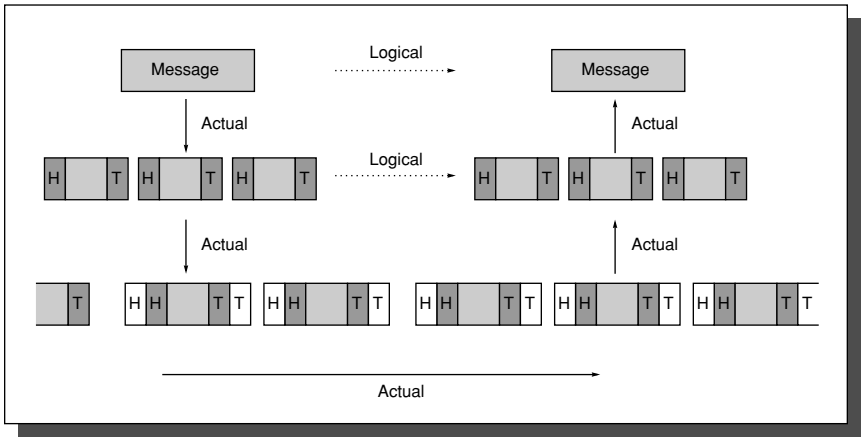


FIGURE 8.26 A generic protocol stack with two layers. Note that communication is peer-to-peer, with headers and trailers for the peer added at each sending layer and removed by each receiving layer. Each layer offers services to the one above to shield it from unnecessary details.

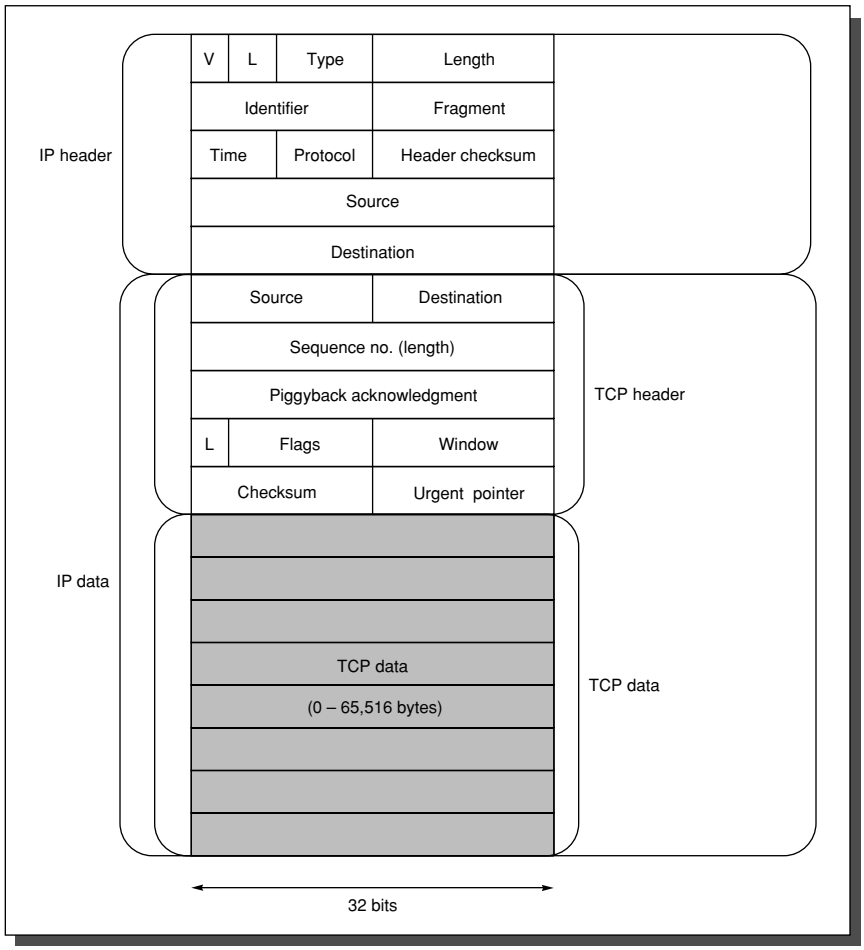


FIGURE 8.27 The headers for IP and TCP. This drawing is 32 bits wide. The standard headers for both are 20 bytes, but both allow the headers to optionally lengthen for rarely transmitted information. Both headers have a length of header field (L) to accommodate the optional fields, as well as source and destination fields. The length field of the whole datagram is in a separate length field in IP, while TCP combines the length of the datagram with the sequence number of the datagram by giving the sequence number in bytes. TCP uses the checksum field to be sure that the datagram is not corrupted, and the sequence number field to be sure the datagrams are assembled into the proper order when they arrive. IP provides checksum error detection only for the header, since TCP has protected the rest of the packet. One optimization is that TCP can send a sequence of datagrams before waiting for permission to send more. The number of datagrams that can be sent without waiting for approval is called the *window*, and the window field tells how many bytes may be sent beyond the byte being acknowledged by this datagram. TCP will adjust the size of the window depending on the success of the IP layer in sending datagrams; the more reliable and faster it is, the larger TCP makes the window. Since the window slides forward as the data arrives and is acknowledged, this technique is called a *sliding window protocol*. The *piggyback acknowledgment field* of TCP is another optimization. Since some applications send data back and forth over the same connection, it seems wasteful to send a datagram containing only an acknowledgment. This piggyback field allows a datagram carrying data to also carry the acknowledgment for a previous transmission, "piggybacking" on top of a data transmission. The *urgent pointer field* of TCP gives the address within the datagram of an important byte, such as a break character. This pointer allows the application software to skip over data so that the user doesn't have to wait for all prior data to be processed before seeing a character

that tells the software to stop. The *identifier field* and *fragment field* of IP allow intermediary machines to break the original datagram into many smaller datagrams. A unique identifier is associated with the original datagram and placed in every fragment, with the fragment field saying which piece is which. The *time-to-live field* allows a datagram to be killed off after going through a maximum number of intermediate switches no matter where it is in the network. Knowing the maximum number of hops that it will take for a datagram to arrive—if it ever arrives—simplifies the protocol software. The *protocol field* identifies which possible upper layer protocol sent the IP datagram; in our case, it is TCP. The V (for *version*) and *type fields* allow different versions of the IP protocol software for the network. Explicit version numbering is included so that software can be upgraded gracefully machine by machine, without shutting down the entire network

Our protocol stack example is TCP/IP. Let's assume that the bottom protocol layer is Ethernet. The next level up is the Internet Protocol or IP layer; the official term for an IP packet is *datagram*. The IP layer routes the datagram to the destination machine, which may involve many intermediate machines or switches. IP makes a best effort to deliver the packets, but does not guarantee delivery, content, or order of datagrams. The TCP layer above IP makes the guarantee of reliable, in-order delivery and prevents corruption of datagrams.

Following the example in Figure 8.26, assume an application program wants to send a message to a machine via an Ethernet. It starts with TCP. The largest number of bytes that can be sent at once is 64 KB. Since the data may be much larger than 64 KB, TCP must divide it into smaller segments and reassemble them in proper order upon arrival. TCP adds a 20-byte header (Figure 8.27) to every datagram, and passes them down to IP. The IP layer above the physical layer adds a 20-byte header, also shown in Figure 8.27. The data sent down from the IP level to the Ethernet is sent in packets with the format shown in Figure 8.20 on page 605. Note that the TCP packet appears inside the data portion of the IP datagram, just as Figure 8.26 suggests.

8.9 Crosscutting Issues for Interconnection Networks

This section describes four topics discussed in other chapters that are fundamental to interconnections.

Density-Optimized Processors versus SPEC-optimized Processors

Given that people all over the world are accessing WWW sites, it doesn't really matter where your servers are located. Hence, many servers are kept at *collocation sites*, which charge by network bandwidth reserved and used, and by space occupied and power consumed.

Desktop microprocessors in the past have been designed to be as fast as possible at whatever heat could be dissipated, with little regard to the size of the package and surrounding chips. One microprocessor in 2001 burns 135 watts! Floor space efficiency was also largely ignored. As a result of these priorities, power is a major cost for collocation sites, and density of processors is limited by the power consumed and dissipated.

With portable computers making different demands on power consumption and cooling for processors and disks, the opportunity exists for using this technology to create considerably denser computation. In such a case performance per watt or performance per cubic foot could replace performance per microprocessor as the important figure of merit.

The key is that many applications already work with large clusters (see section 8.10), so its possible that replacing 64 power hungry processors with, say, 256 efficient processors could be cheaper to run yet be software compatible.

Smart Switches vs. Smart Interface Cards

Figure 8.28 shows a trade-off is where intelligence is located in the network. Generally the question is whether to have smarter network interfaces or smarter switches. Making one side smarter generally makes the other side easier and less expensive.

By having an inexpensive interface it was possible for Ethernet to become standard as part of most desktop and server computers. Lower cost switches were made available for people with small configurations, not needing sophisticated routing tables and spanning tree protocols of larger Ethernet switches.

Infiniband is trying a hybrid approach by offering lower cost interface cards for less demanding devices, such as disks, in the hopes that it will be included with some I/O devices. As Inifiband is planned as the successor to PCI bus, computers may come with an Host Channel Adapter built in.

Protection and User Access to the Network

A challenge is to ensure safe communication across a network without invoking the operating system in the common case. The Cray Research T3D supercomputer offers an interesting case study. It supports a global address space, so loads and stores can access memory across the network. Protection is ensured because each access is checked by the TLB.

To support transfer of larger objects, a block transfer engine (BLT) was added to the hardware. Protection of access requires invoking the operating system before using the BLT, to check the range of accesses to be sure there will be no protection violations.

Figure 8.29 compares the bandwidth delivered as the size of the object varies for reads and writes. For very large reads, 512 KB, the BLT does achieve the highest performance: 140 MBytes/sec. But simple loads get higher performance for 8 KB or less. For the write case, both achieve a peak of 90 MBytes/sec, presumably because of the limitations of the memory bus. But for writes, BLT can only match the performance of simple stores for transfers of 2 MB; anything smaller and it's faster to send stores. Clearly, a BLT that avoided invoking the operating system in the common case would be more useful.

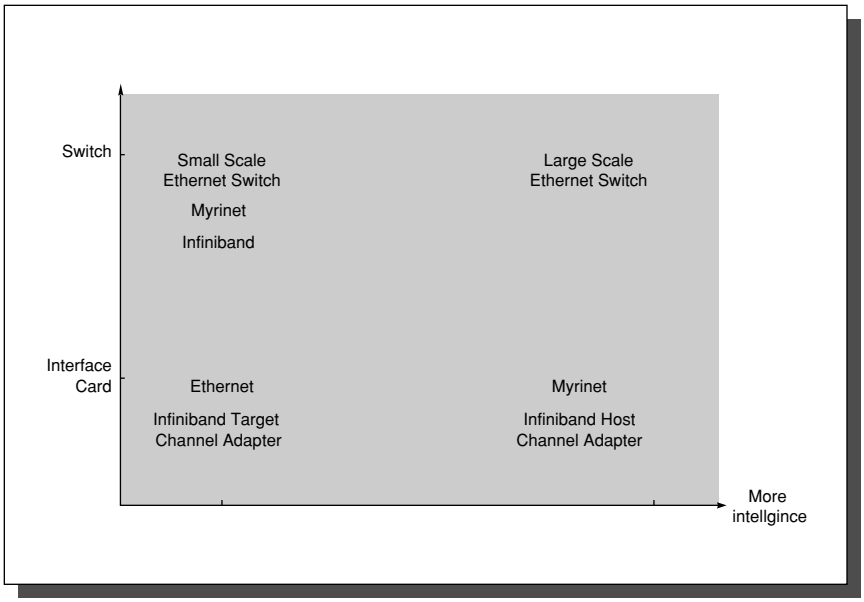


FIGURE 8.28 Intelligence in a network: Switch vs. Interface card. Note that Ethernet switches comes in two styles, depending on the size of the network, and that Infiniband network interfaces come in two styles, depending on whether they are attached to a computer or to a storage device. Myrinet is a proprietary System Area Network

Efficient Interface to Memory Hierarchy versus Interconnection Network

Traditional evaluations of processor performance, such as SPECint and SPECfp, encourage integration of the memory hierarchy with the processor, as the efficiency of the memory hierarchy translates directly into processor performance. Hence, microprocessors have first-level caches on chips along with buffers for writes, and usually have second-level caches on-chip or immediately next to the chip.

Benchmarks such as SPECint and SPECfp do not reward good interfaces to interconnection networks, and hence many machines make the access time to the network delayed by the full memory hierarchy. Writes must lumber their way through full write buffers, and reads must go through the cycles of first- and second-level cache misses before reaching the interconnection. This hierarchy results in newer systems having higher latencies to interconnections than older machines.

Let's compare three machines from the past. A 40-MHz SPARCstation-2, a 50-MHz SPARCstation-20 without an external cache, and a 50-MHz SPARCstation-20 with an external cache. According to SPECint95, this list is in order of in-

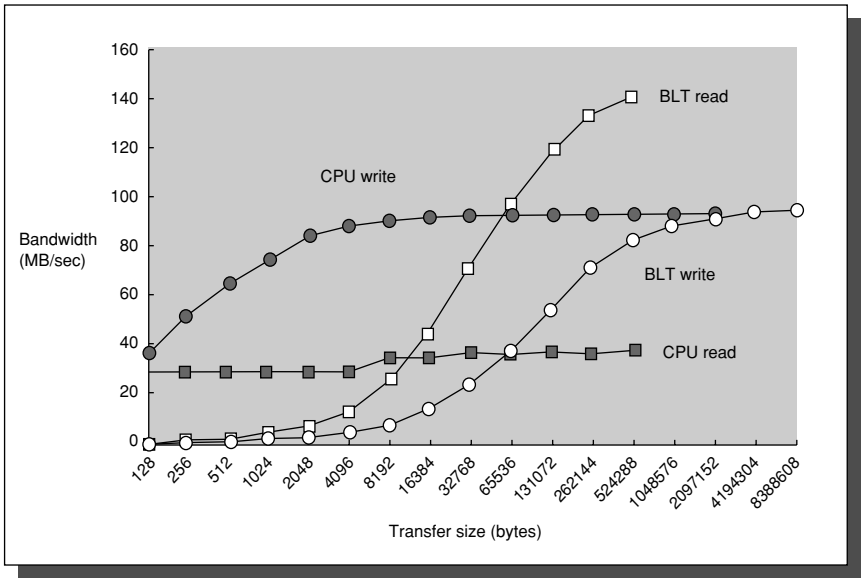


FIGURE 8.29 Bandwidth versus transfer size for simple memory access instructions versus a block transfer device on the Cray Research T3D. (Arpaci et al. [1995].)

creasing performance. The time to access the I/O bus (S-bus), however, increases in this sequence: 200 ns, 500 ns, and 1000 ns. The SPARCstation-2 is fastest because it has a single bus for memory and I/O, and there is only one level to the cache. The SPARCstation-20 memory access must first go over the memory bus (M-bus) and then to the I/O bus, adding 300 ns. Machines with a second-level cache pay an extra penalty of 500 ns before accessing the I/O bus.

On the other hand, recent computers have dramatically improved memory bandwidth, which is helpful to network bandwidth.

Compute-Optimized Processors versus Receiver Overhead

The overhead to receive a message likely involves an interrupt, which bears the cost of flushing and then restarting the processor pipeline. As mentioned earlier, to read the network status and to receive the data from the network interface likely operates at cache miss speeds. As microprocessors become more superscalar and go to faster clock rates, the number of missed instruction issue opportunities per message reception is likely to rise quickly over time.

8.10 Clusters

...do-it-yourself Beowulf clusters built from commodity hardware and software...has mobilized a community around a standard architecture and tools. Beowulf's economics and sociology are poised to kill off the other .... architectural lines–and will likely affect traditional supercomputer centers as well.

Gordon Bell and Jim Gray [2001]

Instead of relying on custom machines and custom networks to build massively parallel machines, the introduction of switches as part of LAN technology meant that high network bandwidth and scaling was available from off-the-shelf components. When combined with using desktop computers and disks as the computing and storage devices, a much less expensive computing infrastructure could be created that could tackle very large problems. And by their component nature, *clusters* are much easier to scale and more easily isolate failures.

There are many mainframe applications—such as databases, file servers, Web servers, simulations, and multiprogramming/batch processing—amenable to running on more loosely coupled machines than the cache-coherent NUMA machines of Chapter 6. These applications often need to be highly available, requiring some form of fault tolerance and repairability. Such applications—plus the similarity of the multiprocessor nodes to desktop computers and the emergence of high-bandwidth, switch-based local area networks—lead to clusters of off-the-shelf, whole computers for large-scale processing.

Performance Challenges of Clusters

One drawback is that clusters are usually connected using the I/O bus of the computer, whereas multiprocessors are usually connected on the memory bus of the computer. The memory bus has higher bandwidth and much lower latency, allowing multiprocessors to drive the network link at higher speed and to have fewer conflicts with I/O traffic on I/O-intensive applications. This connection point also means that clusters generally use software-based communication while multiprocessors use hardware for communication. However, it makes connections nonstandard and hence more expensive.

A second weakness is the division of memory: a cluster of N machines has N independent memories and N copies of the operating system, but a shared address multiprocessor allows a single program to use almost all the memory in the computer. Thus, a sequential program in a cluster has 1/Nth the memory available compared to a sequential program in a shared memory multiprocessor. Interest-

ingly, the drop in DRAM prices has made memory costs so low that this multiprocessor advantage is much less important in 2001 than it was in 1995. The primary issue in 2001 is whether the maximum memory per cluster node is sufficient for the application.

Dependability and Scalability Advantage of Clusters

The weakness of separate memories for program size turns out to be a strength in system availability and expansibility. Since a cluster consists of independent computers are connected through a local area network, it is much easier to replace a machine without bringing down the system in a cluster than in an shared memory multiprocessor. Fundamentally, the shared address means that it is difficult to isolate a processor and replace a processor without significant work by the operating system and hardware designer. Since the cluster software is a layer that runs on top of local operating systems running on each computer, it is much easier to disconnect and replace a broken machine.

Given that clusters are constructed from whole computers and independent, scalable networks, this isolation also makes it easier to expand the system without bringing down the application that runs on top of the cluster. High availability and rapid, incremental extensibility make clusters attractive to service providers for the World Wide Web.

Pros and Cons of Cost of Clusters

One drawback of clusters has been that the cost of ownership. Administering a cluster of N machines is close to the cost of administering N independent machines, while the cost of administering a shared address space multiprocessor with N processors is close to the cost of administering a single, big machine.

Another difference between the two tends to be the price for equivalent computing power for large-scale machines. Since large-scale multiprocessors have small volumes, the extra development costs of large machines must be amortized over few systems, resulting in higher cost to the customer. As we shall see, even prices for components common to small machines are increased, possibly to recover development. In addition, the manufacturer learning curve (see 573 in the prior chapter) brings down the price of components used in the high volume PC market. Since the same switches sold in high volume for small systems can be composed to construct large networks for large clusters, local area network switches have the same economy-of-scale advantages as small computers.

Originally, the partitioning of memory into separate modules in each node was a significant disadvantage to clusters, as division means memory is used less efficiently than on a shared address computer. The incredible drop in price of memory has mitigated this weakness, dramatically changed the trade-offs in favor of clusters.

Shooting for the Best of Both Worlds

As is often the case with two competing solutions, each side tries to borrow ideas from the other to become more attractive.

On one side of the battle, to combat the high-availability weakness of multiprocessors, hardware designers and operating system developers are trying to offer the ability to run multiple operating systems on portions of the full machine. The goal is that a node can fail or be upgraded without bringing down the whole machine. For example, the Sun Fire 6800 server has these features (see section 5.15).

On the other side of the battle, since both system administration and memory size limits are approximately linear in the number of independent machines, some are reducing the cluster problems by constructing clusters from small-scale shared memory multiprocessors.

A more radical approach is to keep storage outside of the cluster, possibly over a SAN, so that all computers inside can be treated as clones of one another. As the nodes may cost on the order of a few thousand dollars, it can be cheaper to simply discard a flaky node than spend the labor costs to try hard to repair it. The tasks of the failed node are then handed off to another clone. Clusters are also benefiting from faster SANs and from network interface cards that offer lower-overhead communication.

Popularity of Clusters

Low cost, scaling and fault isolation proved a perfect match to the companies providing services over the Internet since the mid 1990s. Internet applications such as search engines and email servers are amenable to more loosely coupled computers, as the parallelism consists of millions of independent tasks. Hence, companies like Amazon, AOL, Google, Hotmail, Inktomi, WebTV, and Yahoo rely on clusters of PCs or workstations to provide services used by millions of people every day. We delve into Google in section 8.11.

Clusters are growing in popularity in the scientific computing market as well. Figure 8.30 shows the mix of architecture styles between 1993 and 2000 for the top 500 fastest scientific computers. One attraction is that individual scientists can afford to construct clusters themselves, allowing them to dedicate their cluster to their problem. Shared supercomputers are placed on monthly allocation of CPU time, so its plausible for a scientist to get more work done from a private cluster than from a shared supercomputer. It is also relatively easy for the scientist to scale his computing over time as he gets more money for computing.

Clusters are also growing in popularity in the database community. Figure 8.31 plots the cost-performance and the cost-performance per processor of the different architecture styles running the TPC-C benchmark. Note in the top graph

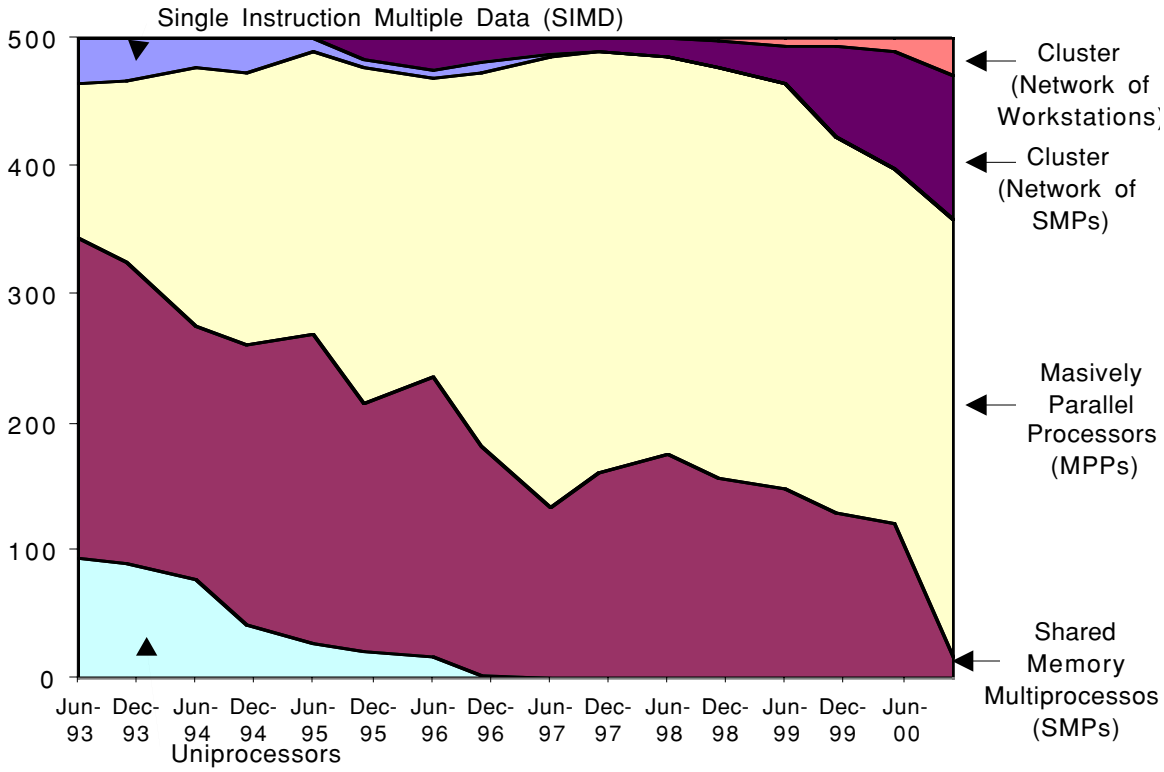


FIGURE 8.30 Plot of Top 500 supercomputer sites between 1993 and 2000. Note that clusters of various kinds grew from 2% to almost 30% in the last three years, while uniprocessors and SMPs have almost disappeared. In fact, most of the MPPs in the list look are similar to clusters. In 2001, the top 500 collectively has a performance of about 100 Teraflops [Bell 2001]. Performance is measured as speed of running Linpack, which solves a dense system of linear equations. This list at `www.top500.org` is updated twice a year.

that not only are clusters fastest, they achieve good cost performance. For example, five SMPs with just 6 to 8 processors have worse cost-performance than the 280-processor cluster! Only small SMPs with two to four processors have much better cost performance than clusters. This combination of high performance and cost-effectiveness is rare. Figure 8.32 shows similar results for TPC-H.

The bottom half of Figure 8.31 shows the scalability of clusters for TPC-C. They scale by about a factor of eight in price or processors while maintaining respectable cost performance.

Now that we have covered the pros and cons of clusters and showed their successes in several fields, the next step is to design some clusters.

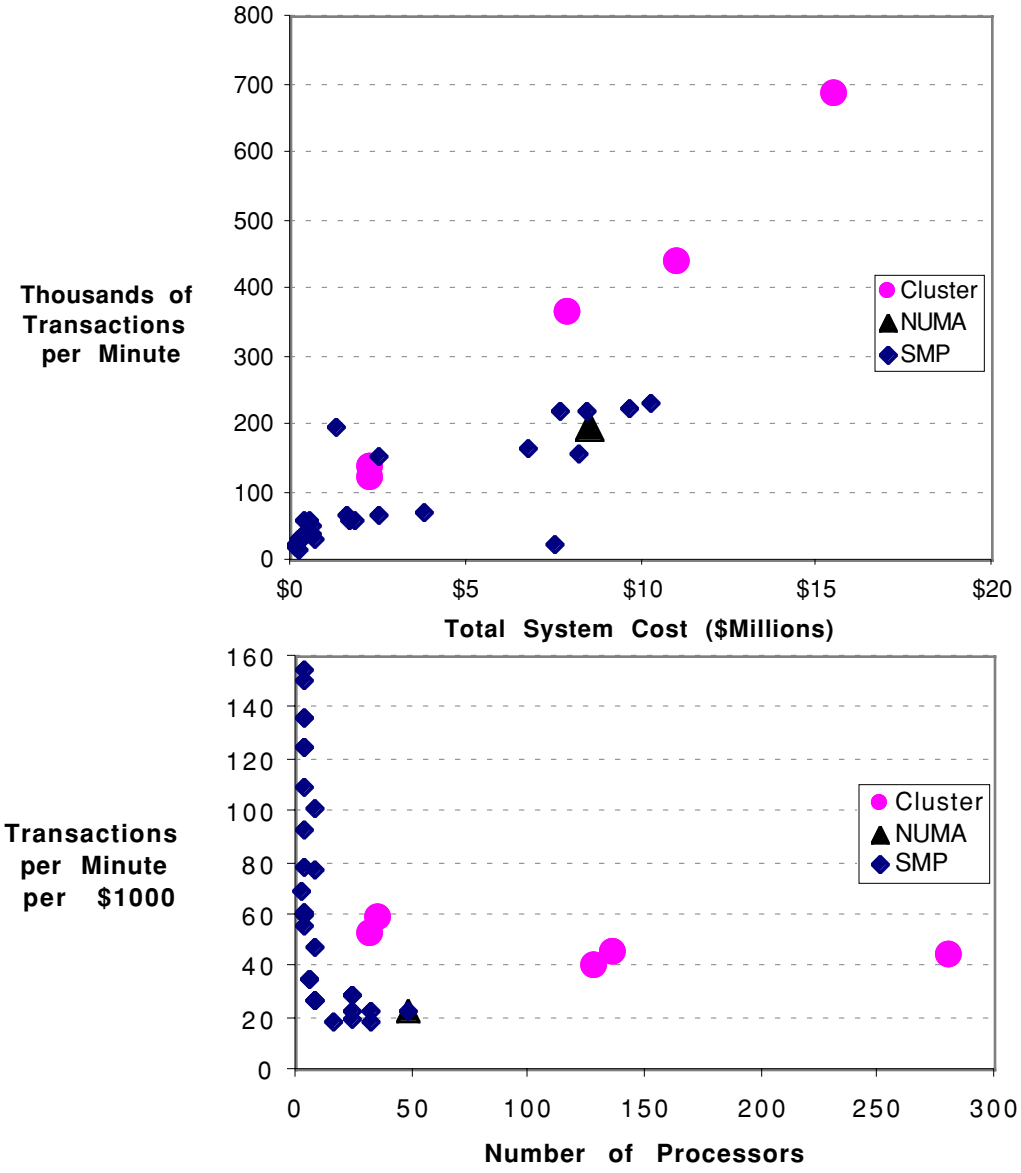


FIGURE 8.31 Performance, Cost, and Cost-Performance per Processor for TPC-C. Not only do clusters have the highest tpmC rating, they have better cost performance ($/tpmC) for any SMP with a total cost over $1M. The bottom graph shows that clusters get high performance by scaling. They can sustain 40 to 50 transactions per minute per $1000 of cost from 32 to 280 processors. Figure 8.40 on page 636 describes the leftmost cluster, and Figure 8.41 on page 637 shows the cost model of TPC-C in more detail. These plots are for all computers that have run version 5 of the TPC-C benchmark as of August 2001.

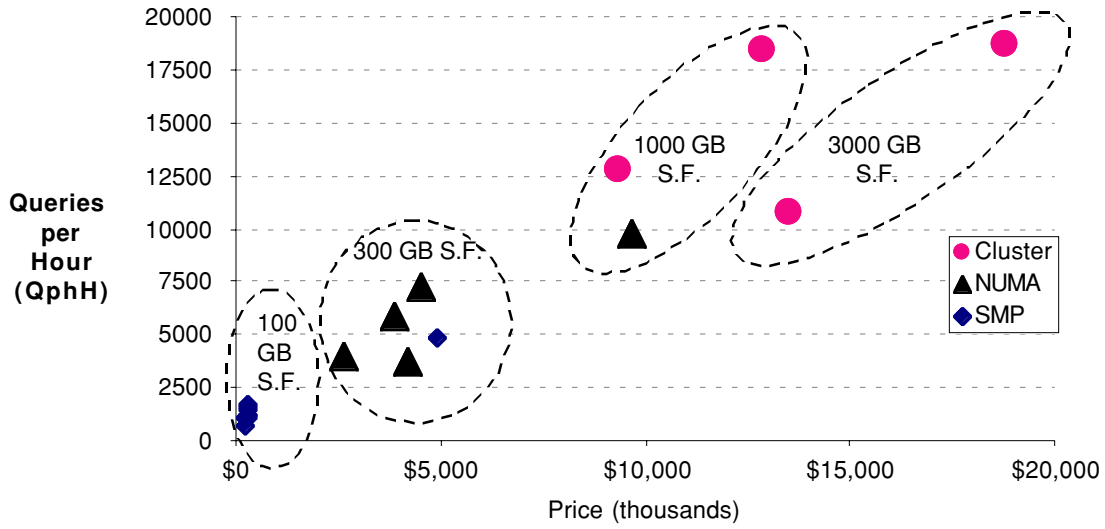


FIGURE 8.32 Performance vs. Cost for TPC-H in August 2001. Clusters are used for the largest computers, NUMA the smaller computers, and SMP the smallest. In violation of TPC-H rules, this figure plots results for different TPC-H scale factors (SF): 100 GB, 300 GB, 1000 GB, and 3000 GB. The ovals separate them.

8.11 Designing a Cluster

To take the discussion of clusters from the abstract to the concrete, this section goes through four examples of cluster design. Like section 7.11 in the prior chapter, the examples evolve in realism. The examples of the last chapter which examined performance and availability apply to clusters as well. Instead, we show cost trade-offs, a topic rarely found in computer architecture.

In each case we are a designing a system with about 32 processors, 32 GB of DRAM, and 32 or 64 disks. Figure 8.33 lists the components we use to construct the cluster, including their prices.

Before starting the examples, Figure 8.33 confirms some of the philosophical points of the prior section. Note that difference in cost and speed processor is in the smaller systems versus the larger multiprocessor. In addition, the price per DRAM DIMM goes up with the size of the computers.

Regarding the processors, the server chip includes a much larger L2 cache, increasing from 0.25 MB to 1 MB. Due to its much larger die size, the price of 1-MB-cache chip is more than double the 0.25-MB-cache. The purpose of the larger L2 cache is to reduce memory bandwidth to allow eight processors to share a memory system. Not only are these large caches chips much more expensive, its

IBM model name	xSeries 300	xSeries 330	xSeries 370
Maximum number processors per box	1	2	8
Pentium III Processor Clock Rate (MHz)	1000	1000	700
L2 Cache (KB)	256	256	1,024
Price of base computer with 1 Processor	$1,759	$1,939	$14,614
Price per extra Processor	n.a.	$799	$1,779
Price per 256 MB SDRAM DIMM	$159	$269	$369
Price per 512 MB SDRAM DIMM	$549	$749	$1,069
Price per 1024 MB SDRAM DIMM	n.a.	$1,689	$2,369
IBM 36.4 GB 10K RPM Ultra160 SCSI	$579	$639	$639
IBM 73.4 GB 10K RPM Ultra160 SCSI	n.a.	$1,299	$1,299
PCI slots: 32bit,33 MHz / 64bit,33 MHz / 64bit,66 MHz	1 / 0 / 0	0 / 2 / 0	0 / 8 / 4
Rack space (VME Rack Units)	1	1	8
Power Supply	200 W	200 W	3 x 750 W
Emulex cLAN-1000 Host Adapter (1 Gbit)	$795	$795	$795
Emulex cLAN5000 8-port switch	$6,280	$6,280	$6,280
Emulex cLAN5000 Rack space (R.U.)	1	1	1
Emulex cLAN5300 30-port switch	$15,995	$15,995	$15,995
Emulex cLAN5300 Rack space (R.U.)	2	2	2
Emulex cLAN-1000 10-meter cable	$135	$135	$135
Extra PCI Ultra160 SCSI Adapter	$299	$299	$299
EXP300 Storage Enclosure (up to 14 disks)	$3,179	$3,179	$3,179
EXP300 Rack space (VME Rack Units)	3	3	3
Ultra2 SCSI 4-meter cable	$105	$105	$105
Standard 19-in Rack (44 VME Rack Units)	$1795	$1795	$1795

FIGURE 8.33 Prices of options for three rack-mounted servers from IBM and 1-Gbit Ethernet switches from Emulex in August 2001. Note the higher price for processors and DRAM DIMMs with larger computers. The base price of these computers includes 256 MB of DRAM (512 MB for 8-way server), two slots for disks, an UltraSCSI 160 adapter, two 100 Mbit Ethernets, a CD-ROM drive, a floppy drive, six to eight fans, and SVGA graphics. The power supply for the Emulex switches is 200 watts and is 500 watts for the EXP300. n the xSeries 370 you must add an accelerator costing $1249 to go over 4 CPUs.

has also been hard for Intel to achieve the similar clock rates to the small-cache chips: 700 MHz vs. 1000 MHz in August 2001.

The higher price of the DRAM is harder too explain based on cost. For example, all include ECC. The uniprocessor uses 133 MHz SDRAM and the 2-way and 8-way both use registered DIMM modules (RDIMM) SDRAM. There might a slightly higher cost for the buffered DRAM between the uniprocessor and 2-way boxes, but it is hard to explain increasing price 1.5 times for the 8-way SMP vs. the 2-way SMP. In fact, the 8-way SDRAM operates at just 100 MHz. Presumably, customers willing to pay a premium for processors for an 8-way SMP are also willing to pay more for memory.

Reasons for higher price matters little to the designer of a cluster. The task is to minimize cost for a given performance target. To motivate this section, here is an overview of the four examples:

1. *Cost of Cluster Hardware Alternatives with Local Disk*: The first example compares the cost of building from a uniprocessor, a 2-way SMP, and an 8-way SMP. In this example, the disks are directly attached to the computers in the cluster.
2. *Cost of Cluster Hardware Alternatives with Disks over SAN*: The second example moves the disk storage behind a RAID controller on a Storage Area Network.
3. *Cost of Cluster Options that is more realistic*: The third example includes the cost of software, the cost of space, some maintenance costs, and operator costs.
4. *Cost and Performance of a Cluster for Transaction Processing*: This final example describes a similar cluster tailored by IBM to run the TPC-C benchmark. (It is one of the cluster results in Figure 8.31.)This example has more memory and many more disks to achieve a high TPC-C result, and at the time of this writing, it the 13th fastest TPC-C system. In fact, the machine with the fastest TPC-C is just a replicated version of this cluster with a bigger LAN switch. This section highlights the differences between this database-oriented cluster and the prior examples.

First Example: Cost of Cluster Hardware Alternatives with Local Disk

This first example looks only at hardware cost of the three alternatives using the IBM pricing information. We'll look at the cost of software and space later.

EXAMPLE Using the information in Figure 8.33, compare the cost of the hardware for three clusters built from the three options in the figure. In addition, calculate the rack space. The goal for this example is to construct a cluster with 32 processors, 32 GB of memory protected by ECC, and more than 2 TB of disk. Connect the clusters with 1 gigabit, switched Ethernet.

ANSWER Figure 8.34 shows the logical organization of the three clusters.

Let's start with the 1-processor option (IBM xSeries model 300). First, we need 32 processors and thus 32 computers. The maximum memory for this computer is 1.5 GB, allowing 1GB x 32 = 32 GB. Each computer can hold two disks and the largest disk available in the model 300 is 36.4 GB, yielding 32 x 2 x 36.4 GB or 2330 GB. Using the built-in slots for storage is the least expensive solution, so we'll take this option. Each computer needs its own Gbit Host Adapter, but 32 cables are more than a 30-port switch can handle. Thus, we use two Emulex cLAN5300 switches. We connect the two switches together with four cables, leaving plenty for of ports for the 32 computers.

A standard VME rack is 19 inches wide and about 6 feet tall, with a typical depth of 30 inches. This size is so popular that it has its own units: 1 *VME rack unit* (RU) is about 1.75 inches high, so a rack can hold objects up to 44 RU. The 32 uniprocessor computers each use 1 rack unit of space, plus 2 rack units for each switch, for a total of 36 rack units. This fits snugly in one standard rack.

For the 2-processor case (model 330), everything is halved. The 32 processors need only 16 computers. The maximum memory is 4 GB, but we need just 2 GB per computer to hit our target of 32 GB total. This model allows 73.4 GB disks, so we need only 16 x 2 x 73.4 GB to reach 2.3 TB, and these disks fit in the slots in the computers. A single 30-port switch has more ports than we need. The total space demand is 18 rack units (16 x 1 + 1 x 2), or less than half a standard rack.

The 8-processor case (model 370) needs only 4 computers to hold 32 processors. The maximum memory is 32 GB, but we need just 8 GB per computer to reach our target. Since there are only 4 computers, the 8-port switch is fine. The shortingcoming is in disks. At 2 disks per computer, these 4 computers can hold at most 8 disks, and the maximum capacity per disk is still 73.4 GB. The solution is to add a storage expansion box (EXP300) to each computer, which can hold up to 14 disks. This solution requires adding an external UltraSCSI controller to each computer as well. The rack space is 8 RU for the computer, 3 RU for the disk enclosure, and 1 RU for the switch. Alas, the total is 4 x (8 + 3) + 1 or 45 rack units, which just misses the maximum of a standard rack. Hence, this option occupies two racks.

Figure 8.35 shows the total cost of each option. This example shows some issues for clusters:

* *Expansibility incurs high prices*. For example, just 4 of the base 8-way SMPs--each with just one processor and 0.5 GB of DRAM--costs more than 32 uniprocessor computers, each with 1 processor and 0.25 GB of DRAM. The only hope of cost competitiveness is to occupy all the options of a large SMP.

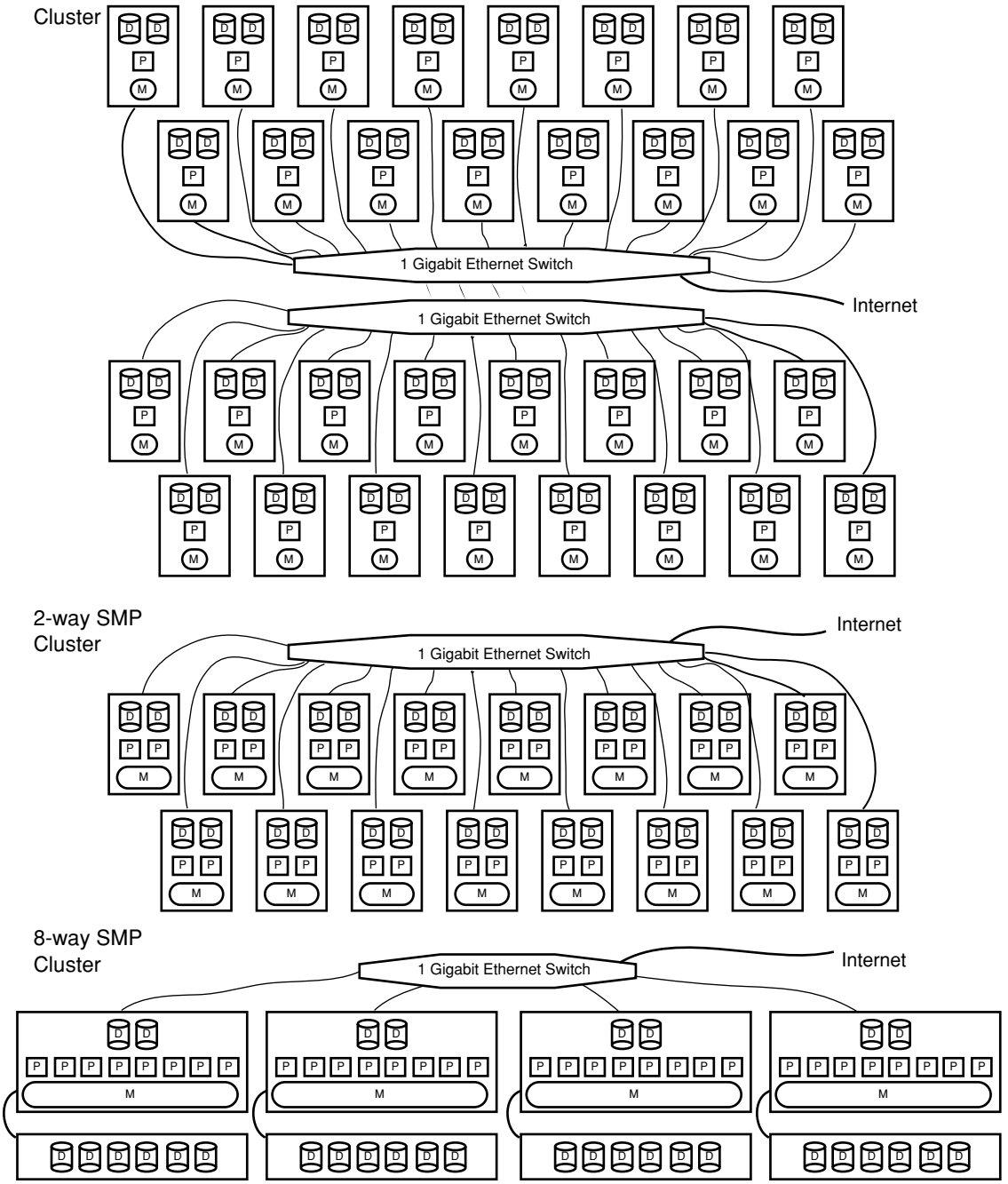


FIGURE 8.34 Three cluster organizations based on uniprocessors (top), 2-way SMPs (middle), and 8-way SMPs (bottom). P stands for processor, M for memory (1, 2, and 8 GB), and D for disk (36.4, 73.4, 73.4 GB).

* *Network vs. local bus trade-off.* Figure 8.35 shows how the larger SMPs need less to spend less on networking, as the memory buses carry more of the communication workload.

The uniprocessor cluster costs 1.1 times the 2-way SMP option, and the 8-way SMP cluster cost 1.6 times the 2-way SMP. The 2-way SMP wins the cost competition because the components are relatively cost-effective and it needs fewer systems and network components. n

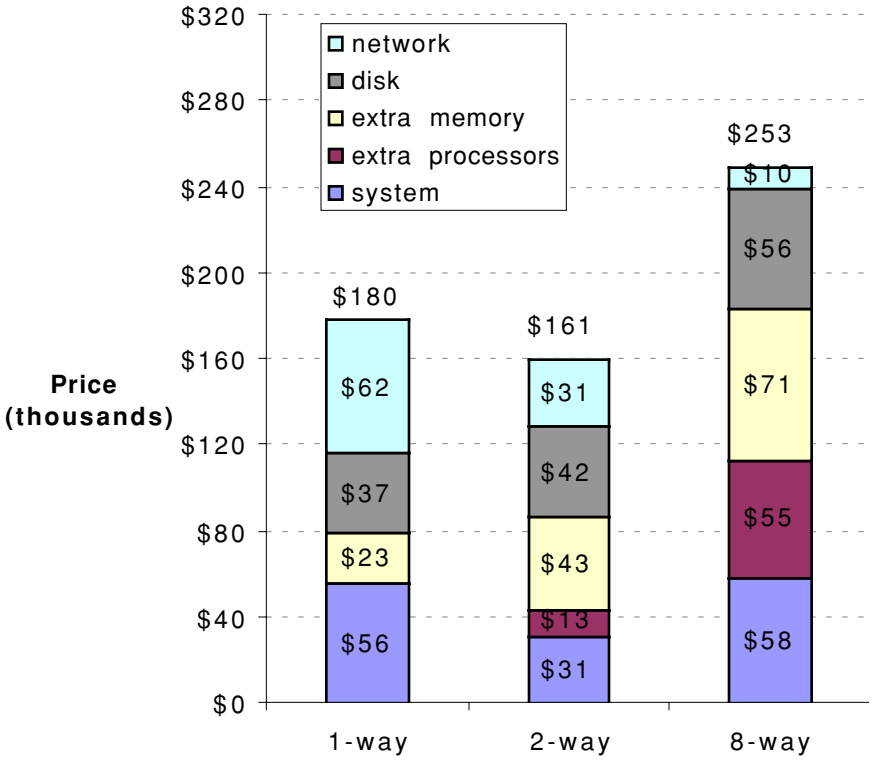


FIGURE 8.35 Price of three clusters with a total of 32 processors, 32 GB memory, and 2.3 TB disk. Note the reduction in network costs as the size of the SMP increases, since the memory buses supply more of the interprocessor communication. Rack prices are included in the total price, but are too small to show in the bars. They account for $1725 in the first two cases and $3450 in the third case.

Second Example: Using a SAN for disks.

The previous example uses disks local to the computer. Although this can reduce costs and space, the problem for the operator is that 1) there is no protection

against a single disk failure, and 2) there is state in each computer that must be managed separately. Hence, the system is down on a disk failures until the operator arrives, and there is no separate visibility or access to storage.

This second example centralizes the disks behind a RAID controller in each case using FC-AL as the Storage Area Network. To keep comparisons fair, we continue use of IBM components. Figure 8.36 lists the costs of the components in this option. Note that this IBM RAID controller requires FC-AL disks.

IBM FC-AL High Availability RAID storage server	$15,999
IBM 73.4 GB 10K RPM FC-AL disk	$1,699
IBM EXP500 FC-AL storage enclosure (up to 10 disks)	$3,815
FC-AL 10-meter cables	$100
IBM PCI FC-AL Host Bus adaptor	$1,485
IBM FC-AL RAID server rack space (VME rack units)	3
IBM EXP500 FC-AL rack space (VME rack units)	3

FIGURE 8.36 Components for Storage Area Network cluster.

EXAMPLE Using the information in Figure 8.36, calculate the cost of the hardware for three clusters above but now use the SAN and RAID controller.

ANSWER The change from the clusters in the first example is that we remove all internal SCSI disks and replace them with FC-AL disks behind the RAID storage server. To connect to the RAID box, we add a FC-AL host bus adapter per computer to the uniprocessor and 2-way SMP clusters and replace the SCSI host bus adapter in the 8-way SMP cluster.

FC-AL can be connected in a loop with up to 127 devices, so there is no problem in connecting the computers to the RAID box. The RAID box has a separate FC-AL loop for the disks. It has room for 10 FC-AL disks, so we need three EXP500 enclosures for the remaining 22 FC-AL disks. (The FC-AL disks are half-height, which are taller than the low profile SCSI disks, so we can fit only 10 FC-AL disks per enclosure.) We just need to add cables for each segment of the loop.

Since the RAID box needs 3 rack units as do each of the 3 enclosures, we need 12 additional rack units of space. This adds a second rack to the uniprocessor cluster, but there is sufficient space in the racks of the other clusters. If we use RAID-5 and have a parity group size of 8 disks, we still have 28 disks of data or 28 x 73.4 or 2.05 TB of user data, which is sufficient for our goals.

Figure 8.37 shows the hardware costs of this solution. Since there

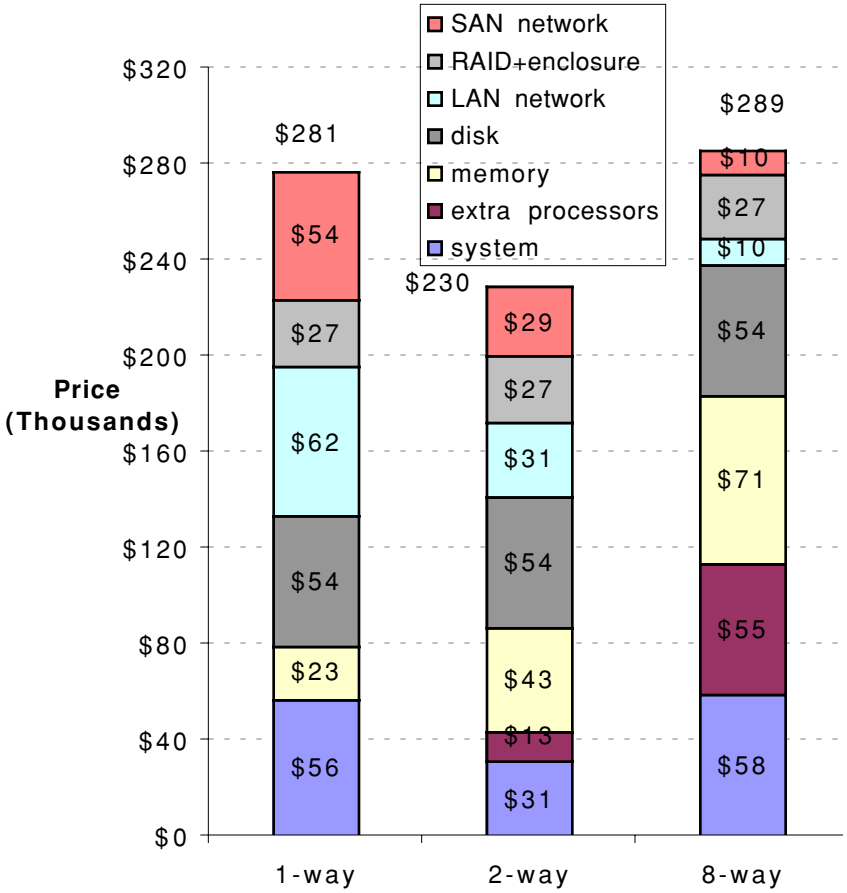


FIGURE 8.37 Prices for hardware for three clusters using SAN for storage. As in Figure 8.35, the cost of the SAN network also shrinks as the servers increase in number of processors per computer. They share the FC-AL host bus adapters and also have fewer cables. Rack prices are too small to see in the columns, but they account for $3450, $1725, and $3450, respectively.

must be one FC-AL host bus adapter per computer, they cost enough to bring the prices of the uniprocessor and 8-way SMP clusters to parity. The 2-way SMP is still substantially cheaper. Notice that again the cost of both the LAN network and the SAN network decrease as the number of computers in the cluster decrease.

The SAN adds about $40,000 to $100,000 to the price of the hardware for the clusters. We'll see in the next example whether we justify such costs.

Third Example: Accounting for Other Costs

The first and second examples only calculated the cost of the hardware (which is what you might expect from book on computer architecture). There are two other obvious costs not included: software and the cost of a maintenance agreement for the hardware. Figure 8.38 lists the costs covered in this example.

Software: Windows 2000 1-4 CPUs + IBM Director	$799
Software: Windows 2000 1-8 CPUs + IBM Director	$3,295
Software: SQL Server Database (per processor!)	$16,541
3-year HW maintenance: LAN switches + HBA	$45,000
3-year HW maintenance: IBM xSeries computers	7.5%
Rack space rental (monthly per rack)	$800 to $1200
Extra 20 amp circuit per rack (monthly)	$200 to $400
Bandwidth charges per megabit (monthly)	$500 to $2000
Operator costs (yearly)	$100,000
DLT tapes (40 GB raw, 80 GB compressed)	$70

FIGURE 8.38 Components for Storage Area Network cluster in 2001. Notice the higher cost of the operating system in the larger server. (Redhat Linix 7.1, however, is $49 for all three.)

Notice that Microsoft quadruples the price when the operating system runs on a computer with 5 to 8 processors versus a computer with 1 to 4 processors. Moreover, the database cost is primarily a *linear* function of the number of processors. Once again, software pricing appears to be based on value to the customer versus cost of development.

Another significant cost is the cost of the operators to keep the machine running, upgrade software, perform backup and restore, and so on. In 2001, the cost (including overhead) is about $100,000 per year for an operator.

In addition to labor costs, backup uses up tapes to act as the long-term storage for system. A typical backup policy is daily incremental dumps and weekly full dumps. A common practice is to save four weekly tapes and then one full dump per month for the last six months. The total is 10 full dumps, plus a week of incremental dumps.

There are other costs, however. One is the cost of the space to house the server. Thus, collocation sites have been created to provide virtual machine rooms for companies. They provide scalable space, power, cooling, and network bandwidth plus provide physical security. They make money by charging rent for space, for network bandwidth, and for optional services from on-site administrators.

Collocation rates are negotiated and much cheaper per unit as space requirements increase. A rough guideline in 2001 is that rack space, which includes one 20-amp circuit, costs $800 to $1200 per month. It drops by 20% if you use more than 75 to 100 racks. Each additional 20 amp circuit per rack costs another $200 to $400 per month. Although we are not calculating these costs in this case, they also charge for network bandwidth: $1500 to $2000 per Mbits/sec per month, if your continuous use is just 1-10 Mbits/second, drops to $500 to $750 per Mbits/sec per month, if your continuous use measures 1 - 2 Gbits/second.

Pacific Gas and Electric in Silicon Valley limits a single building to have no more than 12 megawatts of power and the typical size of a building is no more than 100,000 square feet. Thus, a guideline is that collocation sites are designed assuming no more than 100 watts per square foot. If you include the space for people to get access to a rack to repair and replace components, a rack needs about 10 square feet. Thus, collocation sites expect at most 1000 watts per rack.

EXAMPLE Using the information in Figure 8.38, calculate the total cost of ownership for three years: purchase prices, operator costs, and maintenance costs.

ANSWER Figure 8.39 shows the total cost of ownership for the six clusters.

To keep things simple, we assume each system with local disks needs a full -time operator, but the clusters that access their disks over an SAN with RAID need only a half-time operator. Thus, operator cost is 3 x $100,000 = $300,000 or 3 x $50,000 = $150,000.

For backup, let's assume we need enough tapes to store 2 TB for a full dump. We need four sets for the weekly dumps plus six more sets so that we can have a six-month archive. Tape units normally compress their data to get a factor of two in density, so we'll assume compression successfully turns 40 GB drives into 80 GB drives. The cost of these tapes is:

$$10 \times \frac{2000\text{GB}}{\cancel{80\text{GB/tape}}} \times \$\,70 \;=\; 10 \times 25 \times \$\,70 \;=\; \$17,500$$

The daily backups depend on the amount of data changed. If 2 tapes per day are sufficient (up to 8% changes per day), we need to spend another

$$7 \times 2 \times \$\,70 \;=\; 14 \times \$\,70 \;=\; \$\,980$$

The figure lists maintenance costs for the computers and the LAN. The disks come with a 3-year warranty, so there is no extra maintenance cost for them.

The cost per rack of rental space for three years is 3 x 12 x $1000 or $36,000.

Figure 8.39 shows the 2-way SMP using SAN is the winner. Note that hardware costs are only a half to a third of the cost of ownership. Over

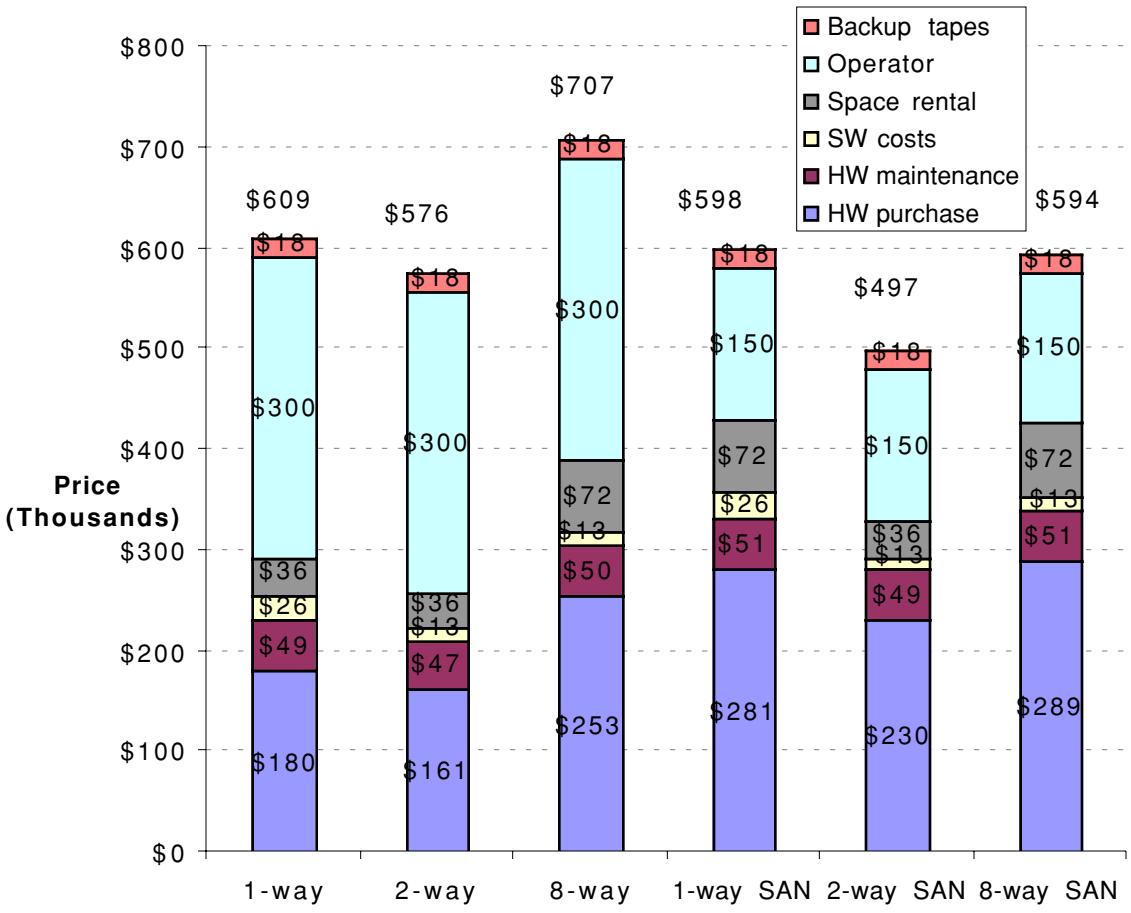


FIGURE 8.39 Total cost of ownership for three years for clusters in Figures 8.35 and 8.37. Operator costs are as significant as purchase price, and hence the assumption that SAN halves operator costs is very significant.

three years the operator costs can be more than the cost of purchase of the hardware, so reducing those costs significantly reduces total cost of ownership.

Our results depend on some critical assumptions, but surveys of the total cost of ownership for items with storage go up to factors five to ten over purchase price. n

Fourth Example: Cost and Performance of a Cluster for Transaction Processing

The August 2001 TPC-C report includes a cluster built from similar building blocks to the examples above. This cluster also has 32 processors, uses the same IBM computers as building blocks, and it uses the same switch to connect computers together. Figure 8.40 shows its organization. It achieves 121,319 queries for hour for $2.2M.

Here are the key differences:

- *Disk size*: since TPC-C cares more about I/Os per second (IOPS) than disk capacity, this clusters uses many small fast disks. The use of small disks gives many more IOPS for the same capacity. These disks also rotate at 15000 RPM vs. 10000 RPM, delivering more IOPS per disk. The 9.1-GB disk costs $405 and the 18.2-GB disk costs $549, or an increase in dollars per GB of factor of 1.7 to 2.5. The totals are 560 9.1-GB disks and 160 18.2-GB disks, yielding a total capacity of 8 TB. (Presumably the reason for the mix of sizes is get sufficient capacity and IOPS to run the benchmark.) These 720 disks need $\lceil 720/14 \rceil$ or 52 enclosures, which is 13 enclosures per computer. In contrast, earlier 8-way clusters achieved 2 TB with 32 disks, as we cared more about cost per GB than IOPS.
- *RAID*: Since the TPC-C benchmark does not factor in human costs for running a system, there is little incentive to use a SAN. TPC-C does require a RAID protection of disks, however. IBM used a RAID product that plugs into a PCI card and provides four SCSI strings. To get higher availability and performance, each enclosure attaches to two SCSI buses. Thus, there are 52 x 2 or 104 SCSI cables attached to the 28 RAID controllers which support up to 28 x 4 or 106 strings.
- *Memory*: Conventional wisdom for TPC-C is to pack as much DRAM as possible into the servers. Hence, each of the four 8-way SMPs is stuffed with the maximum of 32 GB, yielding a total of 128 GB.
- *Processor*: This benchmark uses 900 MHz Pentium III with a 2MB L2 cache. The price is $6599 as compared to prior 8-way clusters for $1799 for the 700 MHz Pentium III with a 1 MB L2 cache.
- *PCI slots*: This cluster uses 7 of the 12 available PCI bus slots for the RAID controllers compared to 1 PCI bus slot for an external SCSI or FC-AL controller in the prior 8-way clusters. This greater utilization follows the guideline of trying to use all resources of a large SMP.
- *Tape Reader, Monitor, Uninterruptable Power Supply*: To make the system easier to come up and to keep running for the benchmark, IBM includes one DLT tape reader, four monitors, and four UPSs.

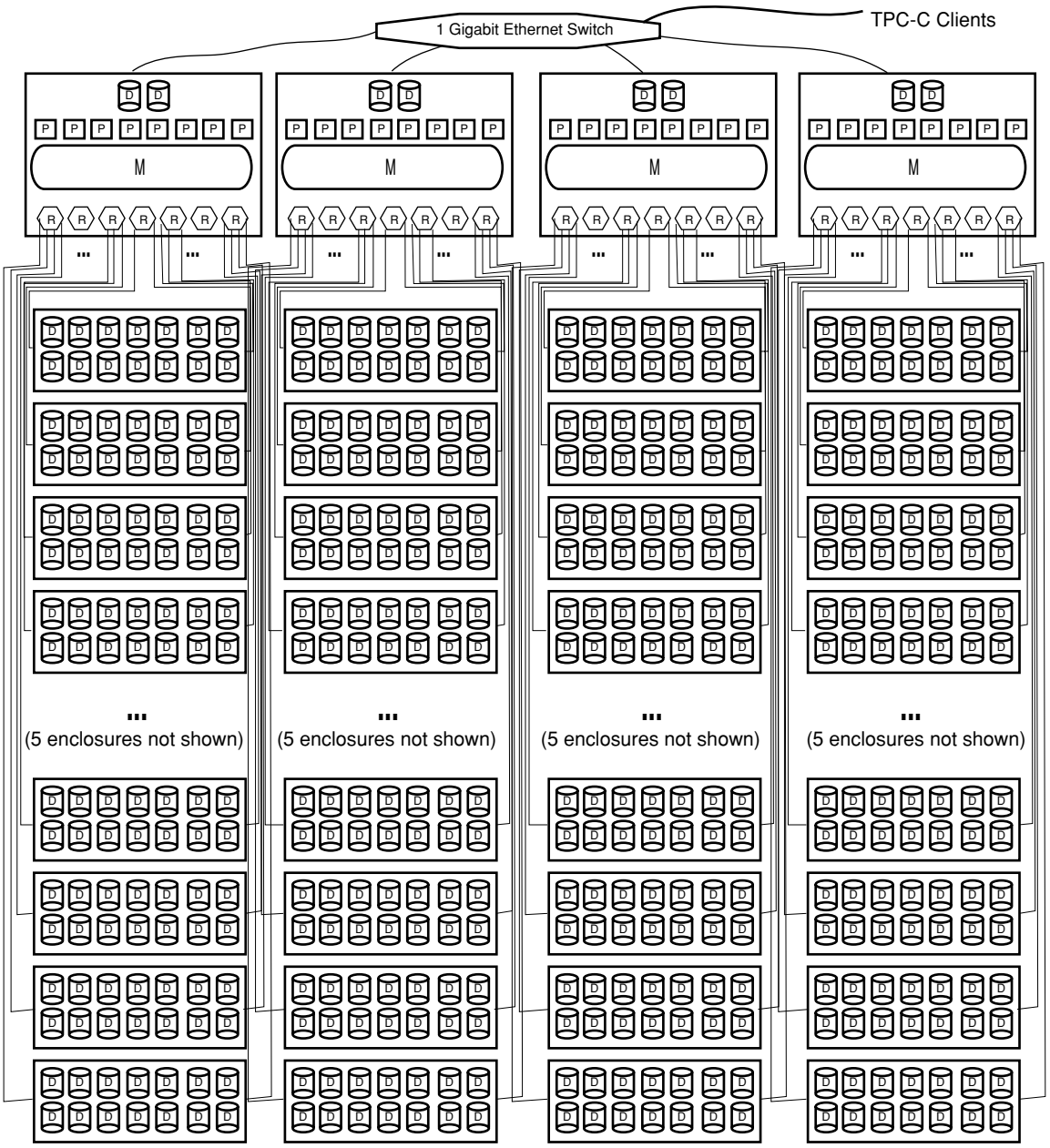


FIGURE 8.40 IBM Cluster for TPC-C. This cluster has 32 Pentium III processors, each running at 900 MHz with a 2MB L2 cache. The total of DRAM memory is 128 GB. Seven PCI slots in each computer contain RAID controllers (R for RAID), and each has four Ultra160 SCSI strings. These strings connect to 13 storage enclosures per computer, giving 52 total. Each enclosure has 14 SCSI disks, either 9.1 GB or 18.2 GB. The total is 560 9.1 GB disk and 140 18.2 GB disks. There are also two 9.1 GB disks inside each computer that are used for paging and rebooting.

- *Maintenance and spares*: TPC-C allows use of spares to reduce maintenance costs, which is a minimum of two spares or 10% of the items. Hence, there are two spare Ethernet switches, host adapters, and cables for TPC**-C.**

Figure 8.41 compares the 8-way cluster from before to this TPC-C cluster. Note that almost half of the cost is in software, installation, and maintenance for the TPC-C cluster. At the time of this writing, the computer with the fastest TPC-C result basically scales this cluster from 4 to 35 xSeries 370 servers and uses bigger Ethernet switches..

	8-way SAN Cluster		TPC-C Cluster	
4 Systems (700 MHz/1MB v. 900 MHz/2MB)	$58	17%	$76	3%
28 Extra processors (700 MHz/1MB v. 900 MHz/2MB)	$55	16%	$190	8%
Extra memory (8 GB v. 32 GB)	$71	20%	$306	14%
Disk drives (2TB/73.4GB v. 8TB/9.1,18.2 GB)	$54	15%	$316	14%
Disk enclosures (3 v. 52)	$11	3%	$165	7%
RAID controller (1 v. 28)	$16	4%	$69	3%
LAN network (1 switch/4 HBAs v. 3 switches/6 HBAs)	$10	3%	$24	1%
SAN network (4 NICs, cables v. 0)	$10	3%	n.a.	0%
Software (Windows v. Windows + SQL server + installation)	$13	4%	$951	42%
Maintenance + hardware setup costs	$51	14%	$115	5%
Racks, UPS, backup (2 racks vs. 7 racks + 4 UPS +1 tape unit)	$3	1%	$40	2%
Total	$352	100%	$2,252	100%

FIGURE 8.41 Comparing 8-way SAN cluster and TPC-C cluster in price (in $1000) and percentage. The higher cost of the system and extra processors is due to using the faster chips with the larger caches. Memory costs are higher due to more total memory and using the more expensive 1 GB DIMMs. The increased disk costs and disk enclosure costs are due to higher capacity and using smaller drives. Software costs increase due to adding SQL server database plus IBM charges for software installation of this cluster. Similarly, although hardware maintenance costs are close, IBM charged to setup seven racks of hardware, whereas we assumed the customer assembled two racks of hardware "for free." Finally, SAN costs are higher due to TPC-C policy of buying spares to lower maintenance costs.

Summary of Examples

With completion of the cluster tour, you've seen a variety of cluster designs, including one representative of the state-of-the-art cost-performance cluster in 2001. Note that we concentrated on cost in constructing these clusters, but only book length prevents us from evaluating the performance and availability bottlenecks in these designs. Given the similarity to performance analysis of storage systems in the last chapter, we leave that to the reader in the exercises.

Having completed the tour of cluster examples, a few things standout. First, the cost of purchase is less than half the cost of ownership. Thus, inventions that only help with hardware costs can solve only a part of the problem. For example, despite the higher costs of SAN, they may lower cost of ownership sufficiently to justify the investment. Second, the smaller computers are generally cheaper and faster for a given function compared to the larger computers. In this case, for the larger cache required to allow several processors to share a bus means a much larger die, which increases cost and limits clock rate. Third, space and power matter for both ends of the computing spectrum: clusters at the high end and embedded computers at the low end.

8.12 Putting It All Together: The Goggle Cluster of PCs

Figure 8.42 shows the rapid growth of the World Wide Web and the corresponding demand for searching it. The number of pages indexed grew by a factor of 1000 between 1994 and 1997, but people were still only interested in the top 10 answers, which was a problem for search engines. In 1997, only one quarter of the search engines would find themselves in their top 10 queries.

Date	WWW pages indexed (Million)	Queries per day (Million)	Search Engine
April 1994	**0.11**	**0.0015**	**World Wide Web Worm**
November 1997	**100**	**20**	**Alta Vista**
December 2000	**1327**	**70**	**Google**

FIGURE 8.42 Growth in pages indexed and search queries performed by several search engines. [Brin and Page, 1998] Searches have been growing about 20% per month at Google, or about 8.9 times per year. Most of the 1.3 billion pages are fully indexed and cached at Google. Google also indexes pages based only on the URLs in cached and indexed pages, so about 40% of the 1.3 billion are just URLs without cached copies of the page at Google.

Searched the web for Hennessy Patterson. Results 1 - 10 of about 13,300. Search took 0.23 seconds.

Computer Architecture: A Quantitative Approach

... on currently predominant and emerging commercial systems, the Hennessy and Patterson have prepared entirely new chapters covering additional advanced topics: ...
www.mkp.com/books_catalog/1-55860-329-8.asp - 13k - Cached - Similar pages

FIGURE 8.43 First entry in result of a search for "Hennessy Patterson". Note that the search took less than 1/4 second, and that it includes a capsule summary of the contents from the WWW page at Morgan Kauffman, and that it offers you to either follow the actual URL (www.mkp...) or just read the cached copy of the page (Cached) stored in the Google cluster.

Google was designed first to be a search engine that could scale at that growth rate. In addition to keeping up with the demand, Google improved the relevance of the top queries produced so that user would likely get what the desired result . For example, Figure 8.43 shows the first Google result for the query "Hennessy Patterson," which from your authors' perspective is the right answer. Techniques to improve search relevance include ranking pages by popularity, examining the text at the anchor sites of the URLs, and proximity of keyword text within a page.

Search engines also have a major reliability requirement, as people are using it at all times of the day and from all over the world. Google must essentially be continuously available.

Since a search engine is normally interacting with a person, its latency must not exceed its users' patience. Google's goal is that no search takes more than 0.5 seconds, including network delays.

As the figures above show, bandwidth is also vital. In 2000, Google served an *average* of almost 1000 queries per second as well as searched and indexed more than a billion pages.

In addition, a search engine must crawl the WWW regularly to have up-to-date information to search. Google crawls the entire WWW and updates its index every 4 weeks, so that every WWW page is visited once a month. Google also keeps a local copy of the text of most pages so that it can provide the snippet text as well as offer a cached copy of the page, as shown in Figure 8.43.

Description of the Google Infrastructure

To keep up with such demand, in December 2000 Google uses more than 6000 processors and 12000 disks, giving Google a total of about one petabyte of disk storage. At the time, the Google site was likely the single system with the largest storage capacity in the private sector.

Rather than achieving availability by using RAID storage, Google relies on redundant sites each with thousands of disks and processors: two sites are in Silicon Valley and one in Virginia. The search index, which is a small number of terabytes, plus the repository of cached pages, which is on the order of the same size, are replicated across the three sites. Thus, if a single site fails, there are still two more that can retain the service. In addition, the index and repository are replicated within a site to help share the workload as well as to continue to provide service within a site even if components fail.

Each site is connected to the Internet via OC48 (2488 Mbits/sec) links of the collocation site. To provide against failure of the collocation link, there is a separate OC12 link connecting the two Silicon Valley sites so that in an emergency both sites can use the Internet link at one site. The external link is unlikely to fail at both sites since different network providers supply the OC48 lines. (The Virginia site now has a sister site to provide so as to provide the same benefits.)

Figure 8.44 shows the floor plan of a typical site. The OC48 link connects to two Foundry BigIron 8000 switches via a large Cisco 12000 switch. Note that

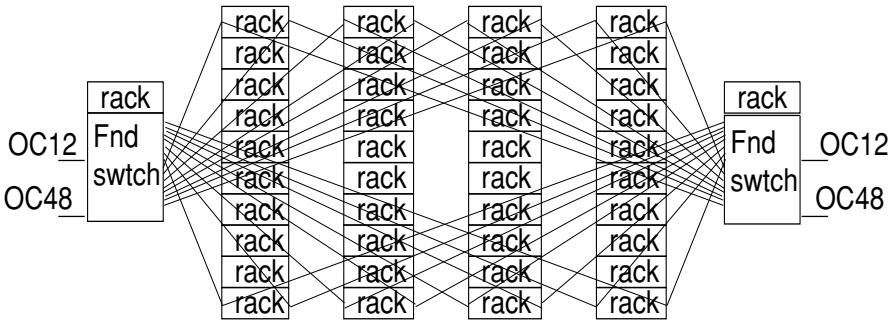


FIGURE 8.44 Floor plan of a Google cluster, from a God's eye view. There are 40 racks, each connected via 4 copper Gbit Ethernet links to 2 redundant Foundry 128 by 128 switches ('Fnd swtch"). Figure 8.45 shows a rack contains 80 PCs, so this facility has about 3200 PCs. (For clarity, the links are only shown for the top and bottom rack in each row.) These racks are on a raised floor so that the cables can be hidden and protected. Each Foundry switch in turn is connected to the collocation site network via an OC48 (2.4 Gbit) to the Internet. There are two Foundry switches so that the cluster is still connected even if one switch fails. There is also a separate OC12 (622 Mbit) link to a separate nearby collocation site in case the OC48 network of one collocation site fails; it can still serve traffic over the OC12 to the other sites network. Each Foundry switch can handle 128 1-Gbit Ethernet lines and each rack has 2 1-Gbit Ethernet lines per switch, so the maximum number of racks for the site is 64. The two racks near the Foundry switches contain a few PCs to act as front ends and help with tasks such as html service, load balancing, monitoring, and UPS to keep the switch and fronts up in case of a short power failure. It would seem that a facility that has redundant diesel engines to provide independent power for the whole site would make UPS redundant. A survey of data center users suggests power failures still happen yearly.

this link is also connected to the rest of the servers in the site. These two switches are redundant so that a switch failure does not disconnect the site. There is also an OC12 link from the Foundry switches to the sister site for emergencies. Each switch can connect to 128 1-Gbit/sec Ethernet lines. Racks of PCs, each with 4 1-Gbit/sec Ethernet interfaces, are connected to the 2 Foundry switches. Thus, a single site can support $2 \times 128/4$ or 64 racks of PCs.

Figure 8.45 shows Google's rack of PCs. Google uses PCs that are only 1 VME rack unit. To connect these PCs to the Foundry switches, it uses an HP Ethernet switch. It is 4 RU high, leaving room in the rack for 40 PCs. This switch has modular network interfaces, which are organized as removable *blades*. Each blade can contain 8 100- Mbits/s Ethernet interfaces or a single 1-Gbit Ethernet interface. Thus, 5 blades are used to connect 100 Mbits/s Cat5 cables to each of the 40 PCs in the rack, and 2 blades are used to connect 1-Gbit/sec copper cables to the two Foundry switches.

As Figure 8.45 shows, to pack even more PCs in a rack Google uses the same configuration in the *front and back* of the rack, yielding 80 PCs and 2 switches per rack. There is about a 3-inch gap in the middle between the columns of PCs for the hot air to exit, which is drawn out of the "chimney" via exhaust fans at the top of the rack.

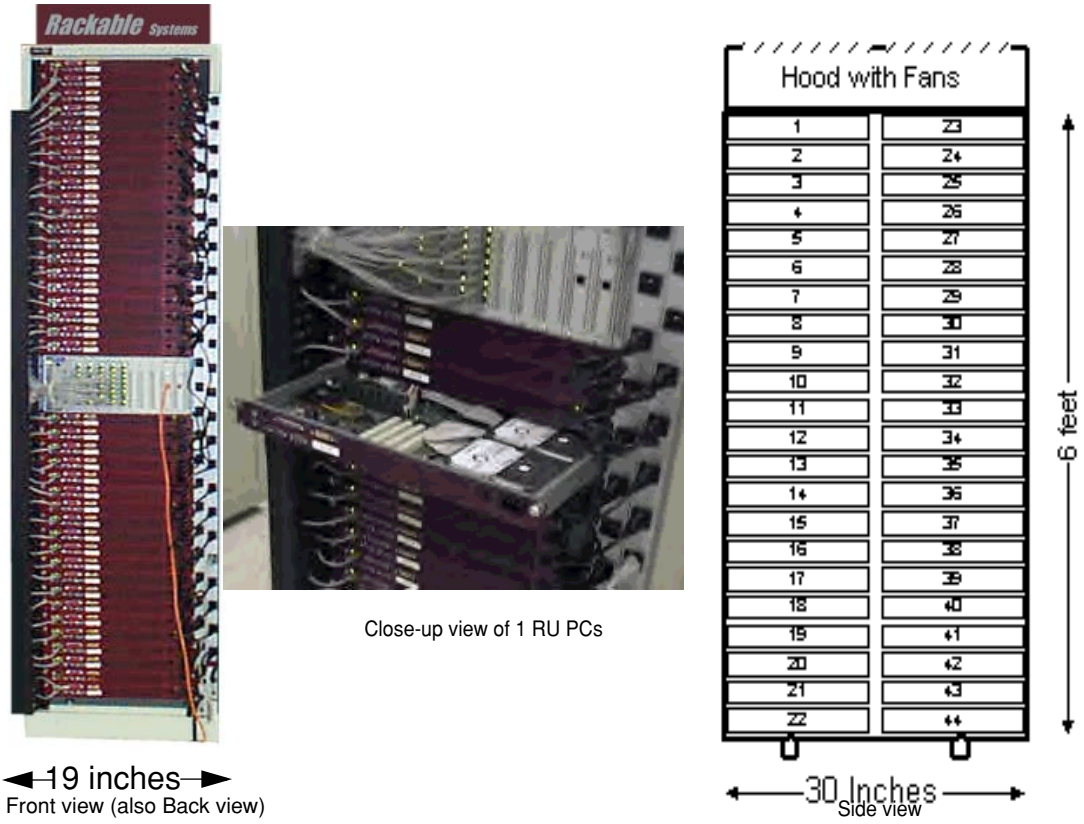


FIGURE 8.45 Front view, side view, and close-up of a rack of PCs used by Google. The photograph on the left shows the HP Procurve 4000M Ethernet switch in the middle, with 20 PCs above and 20 PCs below. Each PC connects via a Cat5 cable on the left side to the switch in the middle, running 100 Mbit Ethernet. Each "blade" of the switch can hold 8 100 Mbit Ethernet interfaces or 1 1Gbit interface. There are also two 1 Gbit Ethernet links leaving the switch on the right. Thus, each PC has only 2 cables: 1 Ethernet and 1 power cord. The far right of the photo shows a power strip, with each of the 40 PCs and the switch connected to it. Each PC is 1 VME rack unit (RU) high. The switch in the middle is 4 RU high. The photo on the middle is a close up of rack, showing contents of a 1 RUPC. This unit contains 2 Maxtor DiamondMax 5400 RPM IDE drives on the right of the box, 256 MB of 100 MHz SDRAM, a PC motherboard, a single power supply, and an Intel microprocessor. Each PC runs versions 2.2.16 or 2.2.17 Linix kernels on a slightly modified RedHat release. Between March 2000 and November 2000, over the period the Google site was populated, the microprocessor varied in performance from a 533 MHz Celeron to an 800 MHz Pentium III. The goal was selecting good cost performance, which was often close to $200 per chip. Disk capacity varied from 40 to 80 GB. You can see the Ethernet cables on the left, power cords on the right, and table Ethernet cables connected to the switch at the top of the figure. In December 2000 the unassembled parts costs are about $500 for the two drives, $200 for the microprocessor, $100 for the motherboard, and $100 for the DRAM. Including the enclosure, power supply, fans, cabling and so on, an assembled PC might cost $1300 to $1700. The drawing on the right shows that PCs are kept in two columns, front and back, so that a single rack holds 80 PCs and 2 switches. The typical power per PC is about 55 watts and about 70 watts per switch, so a rack uses about 4500 watts. Heat is exhausted into a 3-inch vent between the two columns, and the hot air is drawn out the top using fans. (The drawing shows uses 22 PCs per side each 2 RU high instead of the Google configuration of 40 1 RU PCs plus a switch per side.) (Photos and figure from Rackable Systems: http://www.rackable.com/advantage.htm).

The PC itself is a fairly standard: 2 Maxtor ATA/IDE drives, 256 MB of SDRAM, a modest Intel microprocessor, a PC motherboard, one power supply and a few fans. Each PC runs the Linix operating system. To get the best value per dollar, every 2-3 months Google increases the capacity of the drives or the speed of the processor. Thus, the 40 rack site shown above was populated between March and November 2000 has microprocessors that are from a 533 MHz Celeron to an 800 MHz Pentium III, disks that vary in capacity between 40 and 80 GB and in speed at 5400 to 7200 RPM, and memory bus speed is either 100 or 133 MHz.

Performance

Each collocation site connects to the Internet via OC48 (2488 Mbits/sec) links, which is shared by Google and the other Internet service providers. If a typical response to a query is, say, 4000 bytes, then the average bandwidth demand is

$$\frac{70,000,000 \text{ queries/day} \times 4000 \text{ B/query} \times 8 \text{ bits/B}}{24 \times 60 \times 60 \text{ seconds/day}} = \frac{2,240,000 \text{ Mbits}}{86{,}400 \text{ seconds}} \approx 26 \text{Mbit/s}$$

which is just 1% of the link speed of each site. Even if we multiply by a factor of 4 to account for peak versus average demand and requests as well as responses, Google needs little of that bandwidth.

Crawling the web and updating the sites needs much more bandwidth than serving the queries. Let's estimate some parameters to put things into perspective. Assume that it takes 7 days to crawl a billion pages:

$$\frac{1,000,000,000 \text{ pages} \times 4000 \text{ B/page} \times 8 \text{ bits/B}}{24 \times 60 \times 60 \text{ seconds/day} \times 7 \text{days}} = \frac{32,000,000 \text{ Mbits}}{604{,}800 \text{ seconds}} \approx 59 \text{Mbit/s}$$

This data is collected at a single site, but the final multi-terabyte index and repository must then be replicated at the other two sites. If we assume we have 7 days to replicate the data and that we are shipping, say, 5 terabytes from one site to two sites, then the average bandwidth demand is

$$2 \times \frac{5,000,000 \text{ MB} \times 8 \text{ bits/B}}{24 \times 60 \times 60 \text{ seconds/day} \times 7 \text{days}} = \frac{80{,}000{,}000 \text{ Mbits}}{604{,}800 \text{ seconds}} \approx 132 \text{Mbit/s}$$

Hence, the machine to person bandwidth is relatively trivial, with the real bandwidth demand being machine to machine. Moreover, Google's search rate is growing 20% per month, and the number of pages indexed has more than doubled every year since 1997, so bandwidth must be available for growth.

Time of flight for messages across the United States takes about 0.1 seconds, so it's important for Europe to be served from the Virginia site and for California to be served by Silicon Valley sites. To try to achieve the goal of 1/2 second latency, Google software normally guesses where the search is from in order to reduce time of flight delays.

Cost

Given that the basic building block of the Google cluster is a PC, the capital cost of a site is typically a function of the cost of a PC. Rather than buy the latest microprocessor, Google looks for the best cost-performance. Thus, in March 2000 an 800 MHz Pentium III cost about $800, while a 533 MHz Celeron cost under $200, and the difference in performance couldn't justify the extra $600 per machine. (When you purchase PCs by the thousands, every $100 per PC is important.) By November the price of the 800 MHz Pentium III dropped to $200, so it was a better investment. When accounting for this careful buying plus the enclosures and power supplies, your authors estimate the PC cost was $1300 to $1700.

The switches cost about $1500 for the HP Ethernet switch and about $100,000 each for the Foundry switches. If the racks themselves cost about $1000 to $2000 each, the total capital cost of a 40-rack site is about $4.5M to $6.0M. Including 3200 microprocessors and 0.8 terabytes of DRAM, the disk storage costs about $10,000 to $15,000 per terabyte. To put this into perspective, the leading performer for the TPC-C database benchmark in August 2001 is a scaled up version of the cluster from the last example. The hardware alone costs about $10.8M, which includes 280 microprocessors, 0.5 terabytes of DRAM, and 116 terabytes SCSI disks organized as RAID I. Ignoring the RAID I overhead, disk storage costs about $93,000 per terabyte, about a factor of 8 higher than Google despite having 1/8 the number of processors and about 5/8 the DRAM.

The Google rack with 80 PCs, with each PC operating at about 55 Watts, uses 4500 Watts in 10 square feet. It is considerably higher than the 1000 Watts per rack expected by the collocation sites. Each Google rack also uses 60 amps. As mentioned above, reducing power per PC is a major opportunity for the future of such clusters, especially as the cost per kilowatt hour is increasing and the cost per Mbits/second is decreasing.

Reliability

Not surprisingly, the biggest failure in the Google PC is software. On an average day, about 20 machines will be rebooted, and that normally solves the problem. To reduce the number of cables per PC as well as cost, Google has no ability to remotely reboot a machine. The software stops giving work to a machine when it observes unusual behavior, and the operator calls the collocation site and tells them to location of the machine that needs to be rebooted, and a person at the site finds the label and pushes the switch on the front panel. Occasionally the person hits the wrong switch either by mistake or due to mislabeling on the outside of the box.

The next reliability problem is the hardware, which has about 1/10th the failures of software. Typically, about 2% to 3% of the PCs have need to be replaced per year, with failures due to disks and DRAM accounting for 95% of these failures. The remaining 5% are due to problems with the motherboard, power supply, and connectors, and so on. The microprocessors themselves never seem to fail.

The DRAM failures are perhaps a third of the failures. Google sees errors both to bits changing inside DRAM and when bits transfer over the 100 to 133 MHz bus. There was no ECC protection available on PC desktop motherboard chip sets in 2000, so it was not used. The DRAM is determined to be the problem when Linix cannot be installed with a proper check sum until the DRAM is replaced. As PC motherboard chip sets become available, Google plans to start using ECC both to correct some failures but, more importantly, to make it easier to see when DRAMs fail. The extra cost of the ECC is trivial given the wide fluctuation in DRAM prices: careful purchasing procedures are more important than whether or not the DIMM has ECC.

Disks are the remaining PC failures. In addition to the standard failures that result is message to error log in the console, in almost equal numbers these disks will occasionally result in a *performance failure*, with no error message to the log. Instead of delivering normal read bandwidths at 28 Mbytes/second, disks will suddenly drop to 4 MB/second or even 0.8 MB/second. As the disks are under warranty for 5 years, Google sends the disks back to the manufacture for either operational or performance failures to get replacements. Thus, there has been no exploration of the reason for the disk anomalies.

When a PC has problems, it is reconfigured out of the system, and about once a week a person removes the broken PCs. They are usually repaired and then reinserted into the rack.

In regards to the switches, over a 2-year period perhaps 200 of the HP Ethernet switches were deployed, and 2 to 3 have failed. None of the six Foundry switches has failed in the field, although some have had problems on delivery. These switches have a blade-based design with 16 blades per switch, and 2 to 3 of the blades have failed.

The final issue is collocation reliability. The experience of many Internet service providers is that once a year there will be a power outage that affects either the whole site or a major fraction of a site. On average, there is also a network outage so that the whole site is disconnected from the Internet. These outages can last for hours.

There also that collocation site reliability follows a “bathtub” curve: high failures in the beginning, which quickly fall to low rates in the middle, and then rises to high rates at the end. When they are new, the sites are empty and so continuously filled with new equipment. With more people and new equipment being installed, there is a higher outage rate. Once the site is full of equipment, there are fewer people around and less change, so the site has a low failure rate. Once the equipment becomes outdated and it starts being replaced, the activity in the site increases and so does the failure rate. Thus, the failure rate of site depends in part on its age, just as the classic bathtub reliability curves would predict. It is also a function of the people, and if there is a turnover in people, the fault rate can change.

Google accommodates collocation unreliability by having multiple sites with different network providers, plus leased lines between pairs of site for emergen-

cies. Power failures, network outages, and so do not affect the availability of the Google service.

8.13 Another View: Inside a Cell Phone

In 1999, there were 76 million cellular subscribers in the United States, a 25% growth from the year before. That growth rate is almost 35% per year worldwide, as developing countries find it much cheaper to install cellular towers than copper-wire-based infrastructure. Thus, in many countries, the number of cell phones in use exceeds the number of wired phones in use.

Not surprisingly, the cellular handset market is growing at 35% per year, with about 280 million cellular phone handsets sold in 1999. To put that in perspective, in the same year sales of personal computers were 120 million. These numbers mean that tremendous engineering resources are available to improve cell phones, and cell phones are probably leaders in engineering innovation per cubic inch. [Grice, 2000].

Before unveiling the anatomy of a cell phone, let's try a short introduction to wireless technology.

Background on Wireless Networks

Networks can be created out of thin air as well as out of copper and glass, creating *wireless networks*. Much of this section is based on a report from the National Research Council [1997].

A radio wave is an electromagnetic wave propagated by an antenna. Radio waves are modulated, which means that the sound signal is superimposed on the stronger radio wave that carries the sound signal, and hence is called the *carrier signal*. Radio waves have a particular wavelength or frequency: they are measured either the length of the complete wave or as the number of waves per second. Long waves have low frequencies and short waves have high frequencies. FM radio stations transmit on the band of 88 MHz to 108 MHz using frequency modulations (FM) to record the sound signal.

By tuning into different frequencies, a radio receiver can pick up a specific signal. In addition to AM and FM radio, other frequencies are reserved for citizen band radio, television, pagers, air traffic control radar, Global Positioning System, and so on. In the United States, the Federal Communications Commission decides who gets to use frequencies and for what purpose

The *bit error rate* (*BER*) of a wireless link is determined by the received signal power, noise due to interference caused by the receiver hardware, interference from other sources, and characteristics of the channel. Noise is typically proportional to the radio frequency bandwidth, and a key measure is the *signal-to-noise ratio* (*SNR*) required to achieve a given BER. Figure 8.46 lists more challenges for wireless communication.

Challenge	Description	Impact
Path loss	Received power divided by transmitted power; the radio must overcome signal-to-noise ratio (SNR) of noise from interference. Path loss is exponential in distance, and depends on interference if its above 100 meters	1 Watt transmit power, 1 GHz transmit frequency, 1 Mbits/sec data rate at 10^{-7} BER, distance between radios can be 728 meters in free space vs. 4 meters in a dense jungle
Shadow fading	Received signal blocked by objects, buildings outdoors or walls indoors; increase power to improve received SNR. It depends on the number of objects and their dielectric properties	If transmitter is moving, need to change transmit power to ensure received SNR in region
Multipath fading	Interference between multiple versions of signal that arrive at different times, determined by time between fastest signal and slowest signal relative to signal bandwidth.	900 MHz transmit frequency signal power changes every 30 cm
Interference	Frequency reuse, adjacent channel, narrow band interference	Requires filters, spread spectrum

FIGURE 8.46 Challenges for wireless communication.

Typically, wireless communication is selected because the communicating devices are mobile or because wiring is inconvenient, which means the wireless network must rearrange itself dynamically. Such rearrangement makes routing more challenging. A second challenge is that wireless signals are not protected and hence are subject to mutual interference, especially as devices move. Power is the another challenge for wireless communication, both because the devices tend to be battery powered and because antennas radiate power to communicate and little of it reaches the receiver. As a result, raw bit error rates are typically a thousand to a million times higher than copper wire.

There are two primary architectures for wireless networks: *base-station* architectures and *peer-to-peer* architectures. Base stations are connected by land lines for longer distance communication, and the mobile units communicate only with a single local base station. Peer-to-peer architectures allow mobile units to communicate with each other, and messages hop from one unit to the next until delivered to the desired unit. Although peer-to-peer is more reconfigurable, base stations tend to be more reliable since there is only one hop between the device and the station. *Cellular telephony*, the most popular example of wireless networks, relies on radio with base stations.

Cellular systems exploit exponential path loss to reuse the same frequency at spatially separated locations, thereby greatly increasing the number of customers served. Cellular systems will divide a city into nonoverlaping hexagonal cells which use different frequencies if nearby, reusing a frequency only when cells are far enough apart so that mutual interference is acceptable.

At the intersection of three hexagonal cells is a base station with transmitters and antennas that is connected to a switching office which coordinates handoffs when a mobile device leaves one cell and goes into another, as well as to accept and place calls over land lines. Depending on topography, population and so on, the radius of a typical cell is two to ten miles.

The Cell Phone

Figure 8.47 shows the components of a radio, which is the heart of a cell phone. Radio signals are first received by the antenna, then amplified, passed through a mixer, then filtered, demodulated, and finally decoded. The antenna acts as the interface between the medium through which radio waves travel and electronics of the transmitter or receiver. Antennas can be designed to work best in particular directions, giving both transmission and reception directional properties. Modulation encodes information in the amplitude, phase, or frequency of the signal to increase its robustness under impaired conditions. Radio transmitters go through the same steps, just in the opposite order.

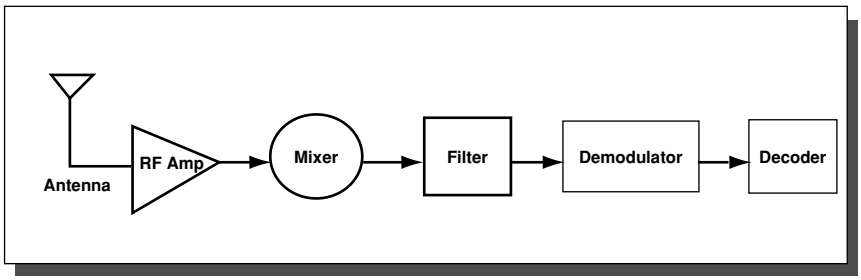


FIGURE 8.47 A radio receiver consists of an antenna, radio frequency amplifier, mixer, filters, demodulator, and decoder. A mixer accepts two signal input and forms an output signal at the sum and difference frequencies. Filters select a narrower band of frequencies to pass on to the next stage. Modulation encodes information to make it more robust. Decoding turns signals into information. Depending on the application, all electrical components can be either analog or digital. For example, a car radio is all analog components, but PC modem is all digital except for the amplifier. Today analog silicon chips are used for the RF amplifier and first mixer in cellular phones.

Originally, all components were analog, but over time most were replaced by digital components, requiring the radio signal to be converted from analog to digital. The desire for flexibility in the number of radio bands led to software routines replacing some of these functions in programmable chips, such as digital signal processors. Because such processors are typically found in mobile devices, emphasis is placed on performance per joule to extend battery life, performance per square millimeter of silicon to reduce size and cost, and bytes per task to reduce memory size.

Figure 8.48 shows the generic block diagram of the electronics of a cell phone handset, with the DSP performing the signal processing and the microcontroller handling the rest of the tasks. Cell phone handsets are basically mobile computers acting as a radio. The include standard I/O devices—keyboard and LCD display—plus a microphone, speaker, and antenna for wireless networking. Battery efficiency affects sales, both in standby power when waiting for a call and in minutes of speaking.

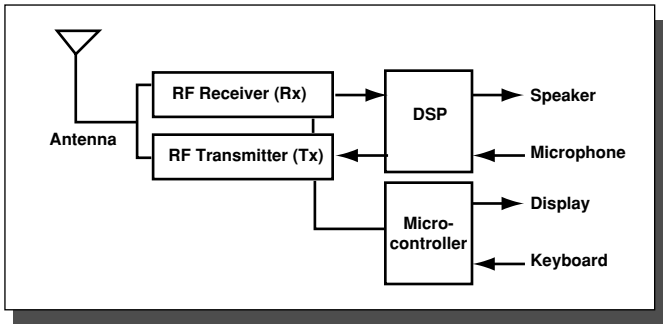


FIGURE 8.48 Block diagram of a cell phone. The DSP performs the signal processing steps of Figure 8.47, and the microcontroller controls the user interface, battery management, and call setup. (Based on Figure 1.3 of Groe and Larson[2000])

When a cell phone is turned on, the first task is to find a cell. It scans the full bandwidth to find the strongest signal, which it keeps doing every seven seconds or if the signals strength drops, as its designed to work from moving vehicles. It then picks an unused radio channel. The local switching office registers the cell phone and records its phone number and electronic serial number, and assigns it voice channel for the phone conversation. To be sure the cell phone got the right channel, the base station sends a special tone on it, which the cell phone sends back to acknowledge it. The cell phone times out after five seconds of it doesn't hear supervisory tone, and starts the process all over again. The original base station makes a handoff request to the incoming base station as the signal strength drops offs.

To achieve a two way conversation over radio, frequency bands are set aside for each direction, forming a frequency pair or *channel*. The original cellular base stations transmitted at 869.04 to 893.97 (called the *forward path*) and cell phones transmitted at 824.04 MHz to 848.97 MHz (called the *reverse path*), with the frequency gap to keep them from interfering with each other. Cells might have had between 4 and 80 channels. Channels were divided into setup channels for call setup, and voice channels that handle the data or voice traffic.

The communication is done digitally, just like a modem, at 9,600 bits/second. Since wireless is a lossy medium, especially from a moving vehicle, the handset send each message is five times. To preserve battery life, the original cell phones typically transmit at two signal strengths--0.6 watts and 3.0 watts--depending on the distance to cell. This relatively low power not only allows smaller batteries and thus smaller cell phones, it aids frequency reuse, which is key to cellular telephony.

Figure 8.49 shows a circuit board from an Ericsson digital phone, with the components identified. Note that the board contains two processors. A Z-80 mi-

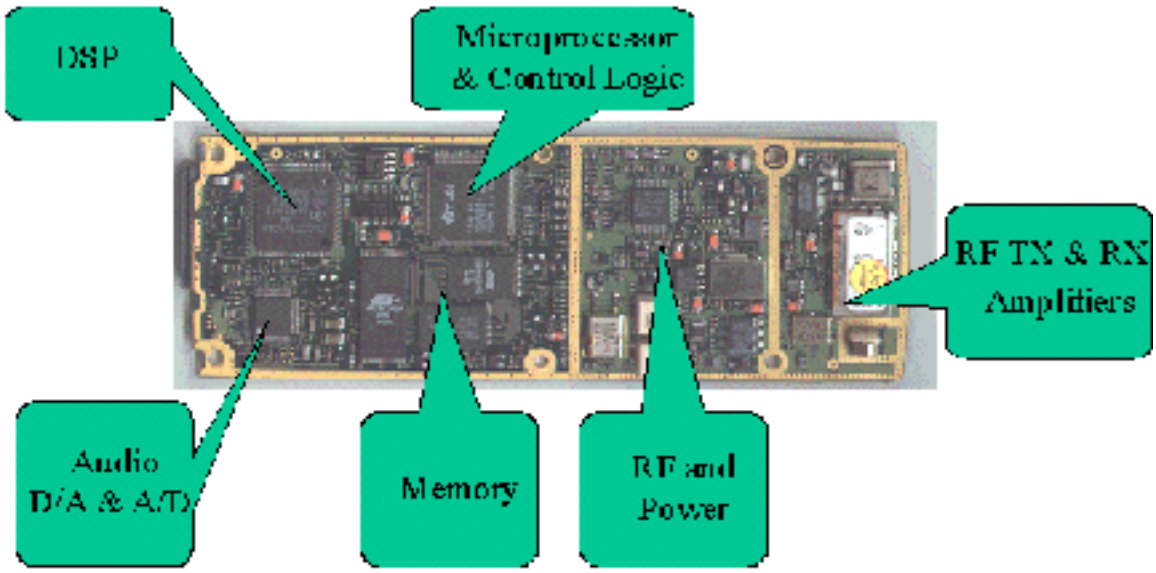


FIGURE 8.49 Circuit card from an Ericsson cell phone. (From Brain [2000]) <<Redo with more subtle labels>>

crocontroller is responsible for controlling the functions of the board, I/O with the keyboard and display, and coordinating with the base station. The DSP handles all signal compression and decompression. In addition there are dedicated chips for Analog-to-Digital and Digital-to-Analog conversion, amplifiers, power management, and RF interfaces.

In 2001, a cell phone has about 10 integrated circuits, including parts made in exotic technologies like gallium arsinide and silicon germanium as well as to standard CMOS. The economics and desire for flexibility will likely shrink this to a few chips, but it appears that a separate microcontroller and DSP will be found inside those chips, with code implementing many of the functions.

Cell Phone Standards and Evolution

Improved communication speeds for cellular phone were developed, with multiple standards. *Code division multiple access* (*CDMA*), as one popular example, uses a wider radio frequency band for a path than the original cellular phones, called *AMPS* for *Advanced Mobile Phone Service*, a mostly analog system. The wider frequency makes it more difficult to block, and is called *spread spectrum*. Other standards are *time division multiple access* (*TDMA*) and *global system for mobile communication* (*GSM*). These second generation standards—CDMA, GSM, and TDMA—are mostly digital.

The big difference for CDMA is that all callers share the same channel, which operates at a much higher rate, and then distinguishes the different calls by encoding each one uniquely. Each CDMA phone call starts at 9600 bits/second, it is then encoded and transmitted as equal sized messages at 1.25 megabits/second. Rather than send each signal five times as in AMPS, each bit is stretched so that it takes eleven times the minimum frequency, thereby accommodating interference and yet successful transmission. The base station receives the messages its separates them into the separate 9600 bits/second streams for each call.

To enhance privacy, CDMA uses pseudo-random sequences from a set of 64 predefined codes. To synchronize the handset and base station so as to pick a common pseudo-random seed, CDMA relies on a clock from the Global Positioning System, which continuously transmits an accurate time signal. By carefully selecting the codes, the shared traffic sounds like random noise to the listener. Hence, as more users share a channel there is more noise, and the signal to noise ratio gradually degrades. Thus, the capacity of the CDMA system is a matter of taste, depending upon sensitivity of the listener to background noise.

In addition, CDMA uses speech compression and varies the rate of data transferred depending how much activity is going on in the call. Both these techniques preserve bandwidth, which allows for more calls per cell. CDMA must regulate power carefully so that signals near the cell tower do not overwhelm those from far away, with the goal of all signals reach the tower at about the same level. The side benefit is that CDMA handsets emit less power, which both helps battery life and increases capacity when users are close to the tower.

Thus, compared to AMPS, CDMA improves capacity of a system by up to an order of magnitude, has better call quality, has better battery life, and enhances users' privacy. After considerable commercial turmoil, there is a new third generation standard called International Mobile Telephony 2000 (IMT-2000) which is based primarily on two competing versions of CDMA and one TDMA. This standard may lead to cell phones which work anywhere in the world.

8.14 Fallacies and Pitfalls

Myths and hazards are widespread with interconnection networks. This section has just a few warnings, so proceed carefully.

Pitfall: Using bandwidth as the only measure of network performance.

Many network companies apparently believe that given sophisticated protocols like TCP/IP that maximize delivered bandwidth, there is only one figure of merit for networks. This may be true for some applications, such as video, where there is little interaction between the sender and the receiver. Many applications, however, are of a request-response nature, and so for every large message there must be one or more small messages. One example is NFS.

		Overhead (secs)			Transmission (secs)		Total time (secs)	
Size	No. messages	ATM	Ethernet	No. data bytes	ATM	Ethernet	ATM	Ethernet
32	771,060	532	389	33,817,052	4	48	536	436
64	56,923	39	29	4,101,088	0	5	40	34
96	4,082,014	2817	2057	428,346,316	46	475	2863	2532
128	5,574,092	3846	2809	779,600,736	83	822	3929	3631
160	328,439	227	166	54,860,484	6	56	232	222
192	16,313	11	8	3,316,416	0	3	12	12
224	4820	3	2	1,135,380	0	1	3	4
256	24,766	17	12	9,150,720	1	9	18	21
512	32,159	22	16	25,494,920	3	23	25	40
1024	69,834	48	35	70,578,564	8	72	56	108
1536	8842	6	4	15,762,180	2	14	8	19
2048	9170	6	5	20,621,760	2	19	8	23
2560	20,206	14	10	56,319,740	6	51	20	61
3072	13,549	9	7	43,184,992	4	39	14	46
3584	4200	3	2	16,152,228	2	14	5	17
4096	67,808	47	34	285,606,596	29	255	76	290
5120	6143	4	3	35,434,680	4	32	8	35
6144	5858	4	3	37,934,684	4	34	8	37
7168	4140	3	2	31,769,300	3	28	6	30
8192	287,577	198	145	2,390,688,480	245	2132	444	2277
Total	11,387,913	7858	5740	4,352,876,316	452	4132	8310	9872

FIGURE 8.50 Total time on 10 Mbit Ethernet and a 155 Mbit ATM, calculating the total overhead and transmission time separately. Note that the size of the headers needs to be added to the data bytes to calculate transmission time. The higher overhead of the software driver for ATM offset the higher bandwidth of the network. These measurements were performed in 1994 using SPARCstation 10s, the Fore Systems SBA-200 ATM interface card and he Fore Systems ASX-200 switch. (NFS measurements taken by Mike Dahlin of U.C. Berkeley.)

Figure 8.50 compares a shared 10 Mbits/second Ethernet LAN to a switched 155 Mbits/second ATM LAN for NFS traffic. Ethernet drivers were better tuned than the ATM drivers, such that 10 Mbits/s Ethernet was faster than 155 Mbits/s ATM for payloads of 512 bytes or less. Figure 8.50 shows the overhead time, transmission time, and total time to send all the NFS messages over Ethernet and ATM. The peak link speed of ATM is 15 times faster and the measured link speed for 8-KB messages is almost 9 times faster. Yet the higher overheads offset the benefits so that ATM would transmit NFS traffic only 1.2 times faster.

Pitfall: Ignoring software overhead when determining performance.

Low software overhead requires cooperation with the operating system as well as with the communication libraries. Figure 8.50 gives one example.

Another example comes from supercomputers. The CM-5 supercomputer had a software overhead of 20 µsecs to send a message and a hardware overhead of 0.5 microseconds. The Intel Paragon reduced the hardware overhead to just 0.2 microseconds, but the initial release of software has a software overhead of 250 microseconds. Later releases reduced this overhead to 25 microseconds, which still dominates the hardware overhead.

This pitfall is simply Amdahl's Law applied to networks: Faster network hardware is superfluous if there is not a corresponding decrease in software overhead.

Pitfall: Trying to provide features only within the network vs. end-to-end.

The concern is providing features at a lower level that only partially satisfy the communication demand that can only be accomplished at the highest level. Saltzer, Reed, and Clark [1984] give the *end-to-end argument* as

> *The function in question can completely and correctly be specified only with the knowledge and help of the application standing at the endpoints of the communication system. Therefore, providing that questioned function as a feature of the communication system itself is not possible. [page 278]*

Their example of the pitfall was a network at MIT that used several gateways, each of which added a checksum from one gateway to the next. The programmers of the application assumed the checksum guaranteed accuracy, incorrectly believing that the message was protected while stored in the memory of each gateway. One gateway developed a transient failure that swapped one pair of bytes per million bytes transferred. Over time the source code of one operating system was repeatedly passed through the gateway, thereby corrupting the code. The only solution was to correct the infected source files by comparing to paper listings and repairing the code by hand! Had the checksums been calculated and checked by the application running on the end systems, safety would have been assured.

There is a useful role for intermediate checks, however, provided that end-to-end checking is available. End-to-end checking may show that something is broken between two nodes, but it doesn't point to where the problem is. Intermediate checks can discover the broken component.

A second issue regards performance using intermediate checks. Although it is sufficient to retransmit the whole in case of failures from the end point, it can be much faster to retransmit a portion of the message at an intermediate point rather than wait for time-out and a full message retransmit at the end point. Balakrishnan et al [1997] found that, for wireless networks, such an intermediate retransmission for TCP/IP communication results in 10-30% higher throughput.

Pitfall: Relying on TCP/IP for all networks, regardless of latency, bandwidth, or software requirements.

The network designers on the first workstations decided it would be elegant to use a single protocol stack no matter where the destination of the message: across a room or across an ocean, the TCP/IP overhead must be paid. This might have been a wise decision especially given the unreliability of early Ethernet hardware, but it sets a high software overhead barrier for commercial systems. Such an obstacle lowers the enthusiasm for low-latency network interface hardware and low-latency interconnection networks if the software is just going to waste hundreds of microseconds when the message must travel only dozens of meters. It also can use significant processor resources. One rough rule of thumb is that each Mbits/second of TCP/IP bandwidth needs about 1 MHz of processor speed, and so a 1000 Mbits/second link could saturate a processor with a 800 to 1000 MHz clock.

The flip side is that from a software perspective, TCP/IP is the most desirable target since it is the most connected and hence largest number of opportunities. The downside of using software optimized to a particular LAN or SAN is that it is limited. For example, communication from a Java program depends on TCP/IP, so optimization for another protocol would require creation of glue software to interface Java to it.

TCP/IP advocates point out that the protocol itself is theoretically not as burdensome as the current implementations, but progress has been modest in commercial systems. There are also TCP/IP off-loading engines entering the market, with the hope of preserving the universal software model while to reducing processor utilization and message latency. If processors to continue to improve much faster than network speeds, or if multiple processors become ubiquitous, software TCP/IP may become less significant on processor utilization and message latency.

8.15 Concluding Remarks

Networking is one of the most exciting fields in computer science and engineering today. The purpose of this chapter to lower the cost of entry into this field by providing definitions and the basic issues so that readers can more easily go into more depth.

The Internet and World Wide Web pervade our society and will likely revolutionize how we access information. Although we couldn't have the Internet without the telecommunication media, it is protocol suites such as TCP/IP that make electronic communication practical. More than most areas of computer science and engineering, these protocols embrace failures as the norm; the network must operate reliably in the presence of failures. Interconnection network hardware and software blend telecommunications with data communications, calling into

question whether they should remain as separate academic disciplines or be combined into a single field.

The silicon revolution has made its way to the switch: just as the "killer micro" changed computing, whatever turns out to be the "killer network" will transform communication. We are seeing the same dramatic change in cost/performance in switches as the mainframe-minicomputer-microprocessor change did to processors. In 2001, companies that make switches are acquiring companies that make embedded microprocessors, just to have better microprocessors for their switches.

Inexpensive switches mean that network bandwidth can scale with the number of nodes, even to the level of the traditional I/O bus. Both I/O designers and memory system designers must consider how to best select and deploy switches. Thus, networking issues apply to all levels of computers systems today: communication within chips, between chips on a board, between boards, and between computers in a machine room, on a campus, or in a country.

The availability and scalability of networks are transforming the machine room. Disks are being connected over SAN to servers versus being directly attached, and clusters of smaller computers connected by a LAN are replacing large servers. The cost-performance, scalability, and fault isolation of clusters have made them attractive to diverse communities: database, scientific computing, and Internet service providers. It's hard to think what else these communities have in common. The challenges for clusters today are the cost of administration.

After decades of low network performance on shared media, networking is in "catch up" mode, and should improve faster than microprocessors. We are not near any performance plateaus, so we expect rapid advance SANs, LANs, and WANs.

This greater network performance is key to the information and communication centric vision of the future of our field. The dramatic improvement in cost/performance of communications has enabled millions of people around the world to find others with common interests. As the quotes at the beginning of this chapter suggest, the authors believe this revolution in two-way communication will change the form of human associations and actions.

8.16 Historical Perspective and References

This chapter has taken the unusual perspective that computers inside the machine room on a LAN or SAN and computers on an intercontinental WAN share many of the same concerns. Although this observation may be true, their histories are very different. We highlight readings on each topic, but good general texts on networking have been written by Davie, Peterson, and Clark [1999] and by Kurose and Ross [2001].

Wide Area Networks

The earliest of the data interconnection networks are WANs. The forerunner of the Internet is the ARPANET, which in 1969 connected computer science departments across the U.S. that had research grants funded by the Advanced Research Project Agency (ARPA), a U.S. government agency. It was originally envisioned as using reliable communications at lower levels. It was the practical experience with failures of underlying technology that led to the failure-tolerant TCP/IP, which is the basis for the Internet today. Vint Cerf and Robert Kahn are credited with developing the TCP/IP protocols in the mid 1970s, winning the ACM Software Award in recognition of that achievement. Kahn [1972] is an early reference on the ideas of ARPANET. For those interested in learning more about TPC/IP, Stevens [1994] has written classic books on the topic.

In 1975, there were roughly 100 networks in the ARPANET and only 200 in 1983; in 1995 the Internet encompasses 50,000 networks worldwide, about half of which are in the United States. In 2000, that number is hard to calculate, but the number of IP hosts grew by a factor of 20 in five years. The key networks that made the Internet possible, such as ARPANET and NSFNET, have been replaced by fully commercial systems, and yet the Internet still thrives.

The exciting application of the Internet is the World Wide Web, developed by Tim Berners-Lee, a programmer at the European Center for Particle Research (CERN) in 1989 for information access. In 1992, a young programmer at the University of Illinois, Marc Andreessen, developed a graphical interface for Web called Mosaic. It became immensely popular. He later became a founder of Netscape, which popularized commercial browsers. In May 1995, at the time of the second edition of this book, there were over 30,000 web pages, and the number was doubling every two months. In November 2000, during the writing of the third edition of this book, there were almost 100 million Internet hosts and more than 1.3 billion WWW pages.

Alles [1995] offers a good survey on ATM. ATM is just the latest of the ongoing standards set by the telecommunications industry, and it is undoubtedly the future for this community. Communication forces standardization by competitive companies, sometimes leading to anomalies. For example, the telecommunication companies in North America wanted to use 64-byte packets to match their existing equipment, while the Europeans wanted 32-byte packets to match their existing equipment. The new standard compromise was 48 bytes to ensure that neither group had an advantage in the marketplace!

Finally, WANs today rely on fiber. Fiber has made such advances that it's original assumption of packet switching is no longer true: WAN bandwidth is not precious. Today WAN fibers are often underutilized. Goralski [1997] discusses advances in fiber optics.

Local Area Networks

ARPA's success with wide area networks led directly to the most popular local area networks. Many researchers at Xerox Palo Alto Research Center had been funded by ARPA while working at universities, and so they all knew the value of networking. In 1974, this group invented the Alto, the forerunner of today's desktop computers [Thacker et al. 1982], *and* the Ethernet [Metcalfe and Boggs 1976], today's LAN. This group--David Boggs, Butler Lampson, Ed McCreight, Bob Sprowl, and Chuck Thacker--became luminaries in computer science and engineering, collecting a treasure chest of awards between them.

This first Ethernet provided a 3 Mbits/sec interconnection, which seemed like an unlimited amount of communication bandwidth with computers of that era. It relied on the interconnect technology developed for the cable television industry. Special microcode support gave a round-trip time of 50 microseconds for the Alto over Ethernet, which is still a respectable latency. It was Boggs' experience as a ham radio operator that led to a design that did not need a central arbiter, but instead listened before use and then varied back-off times in case of conflicts.

The announcement by Digital Equipment Corporation, Intel, and Xerox of a standard for 10 Mbits/sec Ethernet was critical to the commercial success of Ethernet. This announcement short-circuited a lengthy IEEE standards effort, which eventually did publish IEEE 802.3 as an standard for Ethernet.

There have been several unsuccessful candidates in trying to replace the Ethernet. The FDDI committee, unfortunately, took a very long time to agree on the standard and the resulting interfaces were expensive. It was also a shared medium when switches are becoming affordable. ATM also missed the opportunity due in part to the long time to standardize the LAN version of ATM.

Due to failures of the past, LAN modernization efforts have been centered on extending Ethernet to lower cost media, to switched interconnect, to higher link speeds, and to new domains such as wireless communication. Spurgeon [2001] has a nice on-line summary of Ethernet technology, including some of its history.

Massively Parallel Processors

One of the places of innovation in interconnect networks was in massively parallel processors (MPPs). An early MPP was the Cosmic Cube [Seitz 1985], which used Ethernet interface chips to connect 8086 computers in a hypercube. SAN interconnections have improved considerably since then, with messages routed automatically through intermediate switches to their final destinations at high bandwidths and with low latency. Considerable research has gone into the benefits over different topologies in both construction and program behavior. Whether due to faddishness or changes in technology is hard to say, but topologies certainly become very popular and then disappear. The hypercube, widely popular in the 1980s, almost disappeared from MPPs of the 1990s. Cut-through routing, however, has been preserved and is covered by Dally and Seitz [1986].

Chapter 6 records the poor current state of such machines. Government programs such as the Accelerated Strategic Computing Initiative (ASCI) still result in a handful of one-of-a-kind MPPs costing $50 to $100 million, yet these are basically clusters of SMPs.

Clusters

Clusters were probably "invented" in the 1960s by customers who could not fit all there work in one computer, or who needed a backup machine in case of failure of the primary machine [Pfister, 1998]. Tandem introduced a 16-node cluster in 1975. Digital followed with VAXclusters, introduced in 1984. They were originally independent computers that shared I/O devices, requiring a distributed operating system to coordinate activity. Soon they had communication links between computers, in part so that the computers could be geographically distributed to increase availability in case of a disaster at a single site. Users log onto the cluster and are unaware of which machine they are running on. DEC (now Compaq) sold more than 25,000 clusters by 1993. Other early companies were Tandem (now Compaq) and IBM (still IBM), and today virtually every company has cluster products. Most of these products are aimed at availability, with performance scaling as a secondary benefit. Yet in 2000 clusters generally dominate the list of top performers of the TPC-C database benchmark

Scientific computing on clusters emerged as a competitor to MPPs. In 1993, the Beowulf Project started with the goal of fulfilling NASA's desire for a 1 GFLOPS computer for under $50,000. In 1994, a 16-node cluster build from off the shelf PCs using 80486s achieved that goal [Bell 2001.] This emphasis led to a variety of software interfaces to make it easier to submit, coordinate, and debug large programs or a large number of independent programs. In 2001, the fastest (and largest) supercomputers are typically clusters, at least by some popular measures.

Efforts were made to reduce latency of communication in clusters as well as to increase bandwidth, and several research projects worked on that problem. (One commercial result of the low latency research was the VI interface standard, which has been embraced by Infiniband, discussed below.) Low latency then proved useful in other applications. For example, in 1997 a cluster of 100 UltraSPARC desktop computers at UC Berkeley, connected by 160-MB/sec per link Myrinet switches, was used to set world records in database sort—sorting 8.6 GB of data originally on disk in one minute—and in cracking an encrypted message—taking just 3.5 hours to decipher a 40-bit DES key. This research project, called Network of Workstations [Anderson et al, 1995], also developed the Inktomi search engine, which led to a startup company with the same name.

For those interested in learning more, Pfister [1998] has written an entertaining book on clusters. In even greater details, Sterling [2001] has written a do-it-yourself-book on how to build a Beowulf cluster.

System or Storage Area Networks (SANs)

At the second edition of this book, a new class of networks was emerging: *system area networks*. These networks are designed for a single room or single floor and thus the length is ten to hundreds of meters, and were for use in clusters. Close distance means the wires can be wider and faster at lower cost, network hardware can ensure in order delivery, and cascading switches consume less handshaking time. There is also less reason to go to the cost of optical fiber, since the distance advantage of fiber is less important for SANs. The limited size of the networks also makes source-based routing plausible, further simplifying the network. Both Tandem Computers and Myricom sold SANs.

In the intervening years the acronym SAN has been co-opted to also mean *storage* area networks, whereby networking technology is used to connect storage devices to compute servers. Today most people mean storage when they say SAN. The most widely used example in 2001 is Fibre Channel Arbitrated Loop (FC-AL). Not only are disk arrays attached to servers via FC-AL links, there are even some disks with FC-AL links. There are also companies selling FC-AL switches so that storage area networks can enjoy the benefits of greater bandwidth and interconnectivity of switching.

In October 2000 the Infiniband Trade Association announced version 1.0 specification of Infiniband. Led by Intel, HP, IBM, Sun, and other companies, it was proposed as a successor to the PCI bus that brings point-to-point links and switches with its own set of protocols. It's characteristics are desirable potentially both for system area networks to connect clusters and for storage area networks to connect disk arrays to servers. To learn more, the Inifiband standard [2001] is available on the WWW.

The chief competition for Infiniband is the rapidly improving Ethernet technology. The Internet Engineering Task Force is proposing a standard called iSCSI to send SCSI command over IP networks (Satran[2001]). Given the likely cost advantages of the higher volume Ethernet switches and interface cards, in 2001, it is unclear who will win.

Will Infiniband take over the machine room, leaving the WAN as the only link that is not Infiniband? Or will Ethernet will dominate the machine room, even taking over some of the role of storage area networks, leaving Infiniband to simply be an I/O bus replacement? Or will there be a three-level solution: Infiniband in the machine room, Ethernet in the building and on the campus, and then WAN for country? Will TCP/IP off-loading engines become available that can reduce processor utilization and provide low latency yet still provide the software interfaces and generality of TCP/IP? Or will software TCP/IP and faster multiprocessors be sufficient?

In 2001, it is very hard to tell which will win. A wonderful characteristic of computer architecture is that such issues will not remain endless academic debates, unresolved as people rehash the same arguments repeatedly. Instead, the battle is fought in the marketplace, with well-funded and talented groups giving

their best efforts at shaping the future. Moreover, constant changes to technology reward those who are either astute or lucky. The best combination of technology and follow-through has often determined commercial success.

Let the games begin! Time will tell us who wins and who loses, and we will likely know the score by the next edition of this text.

References

ALLES, A. [1995]. "ATM Internetworking," (May), www.cisco.com/warp/public/614/12.html.

ANDERSON, T. E., D. E. CULLER, D. PATTERSON [1995]. "A CASE FOR NOW (NETWORKS OF WORKSTATIONS)," *IEEE MICRO* 15:1 (FEBRUARY), 54–64.

ARPACI, R. H., D. E. CULLER, A. KRISHNAMURTHY, S. G. STEINBERG, AND K. YELICK [1995]. "Empirical evaluation of the CRAY-T3D: A compiler perspective," *Proc. 23rd Int'l Symposium on Computer Architecture* (June), Italy.

Balakrishnan, H.; Padmanabhan, V.N.; Seshan, S.; Katz, R.H. [1997] A comparison of mechanisms for improving TCP performance over wireless links. *IEEE/ACM Transactions on Networking*, vol.5, (no.6), Dec., 756-69.

Brain, M. [2000] "Iside a Digital Cell Phone, "
http://www.howstuffworks.com/inside-cell-phone.htm.

BREWER, E. A. AND B. C. KUSZMAUL [1994]. "How to get good performance from the CM-5 data network." *Proc. Eighth Int'l Parallel Processing Symposium* (April), Cancun, Mexico.

Brin, S.; Page, L. [1998]. "The anatomy of a large-scale hypertextual Web search engine. " *Proc. 7th International World Wide Web Conferenc*e, Brisbane, Qld., Australia, (14-18 April) , 107-17.

COMER, D. [1993]. *Internetworking with TCP/IP,* 2nd ed., Prentice Hall, Englewood Cliffs, N.J.

DALLY, W. J. AND C. I. SEITZ [1986]. "The torus routing chip," *Distributed Computing* 1:4, 187–96.

Davie, B. S., L. L. Peterson, and D. Clark [1999] *Computer Networks: A Systems Approach*, second edition, Morgan Kaufmann Publishers, San Francisco.

DESURVIRE, E. [1992]. "Lightwave communications: The fifth generation," *Scientific American* (International Edition) 266:1 (January), 96–103.

Grice, C. and M. Kanellos [2000] "Cell phone industry at crossroads: Go high or low? , " CNET News, August 31,
http://technews.netscape.com/news/0-1004-201-2518386-0.html?tag=st.ne.1002.tgif.sf.

Goralski, W.[1997]. SONET : a guide to Synchronous Optical Network, New York : McGraw-Hill.

Groe, John B. amd Lawrence E. Larson.[2000] *CDMA mobile radio design* , Boston : Artech House.

InfiniBand Trade Association [2001]. *InfiniBand™ Architecture Specifications Release 1.0.a*, www.infinibandta.org.

KAHN, R. E. [1972]. "Resource-sharing computer communication networks," *Proc. IEEE* 60:11 (November), 1397–1407.

Kurose, J. F. and K. W. Ross [2001]. *Computer networking : a top-down approach featuring the Internet*, Addison-Wesley, Boston

METCALFE, R. M. [1993]. "Computer/network interface design: Lessons from Arpanet and Ethernet. *IEEE J. on Selected Areas in Communications* 11:2 (February), 173–80.

METCALFE, R. M. AND D. R. BOGGS [1976]. "Ethernet: Distributed packet switching for local computer networks," *Comm. ACM* 19:7 (July), 395–404.

National Research Council [1997]. *The evolution of untethered communications*, Computer Science and Telecommunications Board, Washington, D.C. : National Academy Press.

PARTRIDGE, C. [1994]. *Gigabit Networking*. Addison-Wesley, Reading, Mass.

Pfister, Gregory F. [1998]*In search of clusters* , 2nd ed. Upper Saddle River, NJ : Prentice Hall PTR.

SALTZER, J. H., D. P. REED, D. D. CLARK [1984]. "End-to-end arguments in system design," *ACM Trans. on Computer Systems* 2:4 (November), 277–88.

SEITZ, C. L. [1985]. "The Cosmic Cube (concurrent computing),"*Communications of the ACM* 28:1 (January), 22–33.

Sterling, T. [2001]. *Beowulf PC Cluster Computing with Windows and Beowulf PC Cluster Computing with Linux*, MIT Press, Cambridge, MA.

Spurgeon, C. [2001] "Charles Spurgeon's Ethernet Web Site,"wwwhost.ots.utexas.edu/ethernet/ethernet-home.html.

Satran, J et. al.[2001] "iSCSI," IPS working group of IETF, Internet draft http://www.ietf.org/internet-drafts/draft-ietf-ips-iscsi-07.txt

Stevens, W. R. [1994-1996]. *TCP/IP illustrated*, (three volumes) Addison-Wesley Pub. Co., Reading, Mass..

TANENBAUM, A. S. [1988]. *Computer Networks,* 2nd ed., Prentice Hall, Englewood Cliffs, N.J.

THACKER, C. P., E. M. MCCREIGHT, B. W. LAMPSON, R. F. SPROULL, AND D. R. BOGGS [1982]. "Alto: A personal computer," in *Computer Structures: Principles and Examples*, D. P. Siewiorek, C. G. Bell, and A. Newell, eds., McGraw-Hill, New York, 549–572.

WALRAND, J. [1991]. *Communication Networks: A First Course*, Aksen Associates: Irwin, Homewood, Ill.

E X E R C I S E S

- Using the examples from section 8.11, use the techniques from Chapter 7 to calculate the reliability of the cluster. The results of failrues on Tertiary Disk give one set of failure information. What is the MTTF? Where are the single points of failure? How could the designs be changed to improve MTTF?
- Along similar lines, calculate the performance bottlenecks? How does it change if we use rules of thumb on utilization for Chapter 7 vs. assuming 100% utilization?
- The SAN versions just use FC-AL loops versus adding a FC-AL switch. What would have to change in the disk system to make a FC-AL switch valuable? (RAID is the bottleneck with only a single FC-AL loop between the box and the server.)
- Undoubtedly the top 10 of TPC-C has changed. Find a cluster from Dell or Compaq, and go to their web sites to determine the prices of the varying cluster strategies as we did in the examples. Note that the execute overview lists all the components and their prices at the time of the benchmark. They can serve as good placeholders until or unless you can find the current real prices online. They also supply maintenance costs.
- Add a discussion question on use of Ethernet vs.Infiniband in the machine room. What are the technical advantages of each? What are the economic ad-

vantages of each? Why would people maintaining the system prefer one to the other?

- In all exercises, should go to faster Ethernet (and ATM).
- We could use 5 to 10 more exercises.
- Some simple ones: go to the TPC web site and look at which architectures--clusters vs. some form of multiprocessors--dominate each benchmark in performance and in cost performance. Make a discussion question as to why this might vary between benchmarks. How has it changed since the data in the figure? Have the trends contrinued, or not?
- Do a similar study for the Linpack benchmarks (List of Top 500 supercomputers). See if there is older versions of the list so you can see how machine types and brand names change over time. How has it changed since the data in the figure? Have the trends contrinued, or not?
- If you have access to an SMP and a cluster, write a program to measure latency of communication and bandwidth of communication between processors.

8.1 [15] <8.2> Assume the overhead to send a zero-length data packet on an Ethernet is 500 microseconds and that an unloaded network can transmit at 90% of the peak 10 Mbits/sec rating. Plot the delivered bandwidth as the data transfer size varies from 32 bytes to 1500.

- Change Ethernet speed in the next one. Figure still there?

8.2 [15] <8.2> One reason that ATM has a fixed transfer size is that when a short message is behind a long message, a node may need to wait for an entire transfer to complete. For applications that are time-sensitive, such as when transmitting voice or video, the large transfer size may result in transmission delays that are too long for the application. On an unloaded interconnection, what is the worst-case delay if a node must wait for one full-size Ethernet packet versus an ATM transfer? See Figure 8.20 (page 605) to find the packet sizes. For this question assume you can transmit at 100% of the 155 Mbits/sec of the ATM network and 100% of the 10 Mbits/sec Ethernet.

- Update next one to larger tapes, speeds. Match assumptions in revised example?

8.3 [20/10] <8.3>Is electronic communication always fastest for longer distances than the Example on page 583? Calculate the time to send 100 GB using 10 8-mm tapes and an overnight delivery service versus sending 100 GB by FTP over the Internet. Make the following four assumptions:

- The tapes are picked up at 4 P.M. Pacific time and delivered 4200 km away at 10 A.M. Eastern time (7 A.M. Pacific time).
- On one route the slowest link is a T1 line, which transfers at 1.5 Mbits/sec.
- On another route the slowest link is a 10 Mbits/sec Ethernet.

- n You can use 50% of the slowest link between the two sites.

a. [20] <8.3> Will all the bytes sent by either Internet route arrive before the overnight delivery person arrives?

b. [10] <8.3> What is the bandwidth of overnight delivery? Calculate the average bandwidth of overnight delivery service for a 100-GB package.

- n Perhaps a next exercise can add bandwidth of networking links on campus and over the internet. Mary Baker at Stanford has created a new set of software that is much more efficient at finding bandwidth. Latency can be figured out from ping and traceroute (I recall). I can imagine several exercises along these lines.

8.4 [20/20/20/20] <8.8> If you have access to a UNIX system, use `ping` to explore the Internet. First read the manual page. Then use `ping` without option flags to be sure you can reach the following sites. It should say that `X is alive`. Depending on your system, you may be able to see the path by setting the flags to verbose mode (`-v`) and trace route mode (`-R`) to see the path between your machine and the example machine. Alternatively, you may need to use the program `traceroute` to see the path. If so, try its manual page. You may want to use the UNIX command `script` to make a record of your session.

a. [20] <8.8> Trace the route to another machine on the same local area network.

b. [20] <8.8> Trace the route to another machine on your campus that is *not* on the same local area network.

c. [20] <8.8> Trace the route to another machine *off campus*. For example, if you have a friend you send email to, try tracing that route. See if you can discover what types of networks are used along that route.

d. [20] <8.8> One of the more interesting sites is the McMurdo NASA government station in Antarctica. Trace the route to mcmvax.mcmurdo.gov.

- n Change next to Ethernet example?

8.5 [12/15/15] <8.4> Assume 64 nodes and 16×16 ATM switches in the following. (This exercise was suggested by Mark Hill.)

a. [12] <8.4> Design a switch topology that has the minimum number of switches.

b. [15] <8.4> Design a switch topology that has the minimum latency through the switches. Assume unit delay in the switches and zero delay for wires.

c. [15] <8.4> Design a switch topology that balances the bandwidth required for all links. Assume a uniform traffic pattern.

- n I think this example was dropped?

8.6 [20] <8.4> Redo the cut-through routing calculation for CM-5 on page 590 of different sizes: 64, 256, and 1024 nodes.

- n I dropped the all-to-all example. Perhaps put in as exercise? Reword and see if this exercise still makes sense

8.7 [15] <8.4> Calculate the time to perform a broadcast (from-one-to-all) on each of the

topologies in Figure 8.17 on page 598, making the same assumptions as the two Examples on pages 584–588.

n I dropped the all-to-all example. Reword and see if this exercise still makes sense

8.8 [20] <8.4> The two Examples on pages 584–588 assumed unlimited bandwidth between the node and the network interface. Redo the calculations in Figure 8.17 on page 598, this time assuming a node can only issue one message in a time unit.

8.9 [15] <8.4> Compare the interconnection latency of a crossbar, Omega network, and fat tree with eight nodes. Use Figure 8.13 on page 593 and add a fat tree similar to Figure 8.14 on page 595 as a third option. Assume that each switch costs a unit time delay. Assume the fat tree randomly picks a path, so give the best case and worst case for each example. How long will it take to send a message from node P0 to P6? How long will it take P1 and P7 to also communicate?

n figure in next exercise was dropped. Replacing it with 8.50 on page 651 requires changing the question. Perhaps use the data in the figure to first calculate what is the delivered Mbits/sec (accounting for overheads) for each network for each size of NSF payload. Then ask what is n1/2 for each network given those overheads.

n <<Figure below is gone, so pick another example?>>

8.10 [15] <8.4> One interesting measure of the latency and bandwidth of an interconnection is to calculate the size of a message needed to achieve one-half of the peak bandwidth. This halfway point is sometimes referred to as $n_{1/2}$, taken from the vector processing. Using Figure 7.36 on page 621, estimate $n_{1/2}$ for TCP/IP message using ATM and the Ethernet.

8.11 [15] <8.8> Use FTP to transfer a file from a remote site and then between local sites on the same LAN. What is the difference in bandwidth for each transfer? Try the transfer at different times of day or days of the week. Is the WAN or LAN the bottleneck?

8.12 [15] <8.4> Draw the topology of a 6-cube similar to the drawing of the 4-cube in Figure 8.16 on page 597.

8.13 [12/12/12/15/15/18] <8.7> Use M/M/1 queuing model to answer this exercise. Measurements of a network bridge show that packets arrive at 200 packets per second and that the gateway forwards them in about 2 ms.

a. [12] <8.7> What is the utilization of the gateway?

b. [12] <8.7> What is the mean number of packets in the gateway?

c. [12] <8.7> What is the mean time spent in the gateway?

d. [15] <8.7> Plot the response time versus utilization as you vary the arrival rate.

e. [15] <8.7> For an M/M/1 queue, the probability of finding n or more tasks in the system is Utilization^n. What is the chance of an overflow of the FIFO if it can hold 10 messages?

f. [18] <8.7> How big must the gateway be to have packet loss due to FIFO overflow to

be less than one packet per million?

8.14 [20] <8.7> The imbalance between the time of sending and receiving can cause problems in network performance. Sending too fast can cause the network to back up and increase the latency of messages, since the receivers will not be able to pull out the message fast enough. A technique called *bandwidth matching* proposes a simple solution: Slow down the sender so that it matches the performance of the receiver [Brewer 1994]. If two machines exchange an equal number of messages using a protocol like UDP, one will get ahead of the other, causing it to send all its messages first. After the receiver puts all these messages away, it will then send its messages. Estimate the performance for this case versus a bandwidth-matched case. Assume the send overhead is 200 microseconds, the receive overhead is 300 microseconds, time of flight is 5 microseconds, and latency is 10 microseconds, and that the two machines want to exchange 100 messages.

8.15 [40] <8.7> Compare the performance of UDP with and without bandwidth matching by slowing down the UDP send code to match the receive code as advised by bandwidth matching [Brewer 1994]. Devise an experiment to see how much performance changes as a result. How should you change the send rate when two nodes send to the same destination? What if one sender sends to two destinations?

C.1	Introduction	C-2
C.2	Addressing Modes and Instruction Formats	C-4
C.3	Instructions: The MIPS Core Subset	C-5
C.4	Instructions: Multimedia Extensions of the Desktop/Server RISCs	C-16
C.5	Instructions: Digital Signal-Processing Extensions of the Embedded RISCs	C-18
C.6	Instructions: Common Extensions to MIPS Core	C-19
C.7	Instructions Unique to MIPS64	C-24
C.8	Instructions Unique to Alpha	C-26
C.9	Instructions Unique to SPARC v.9	C-27
C.10	Instructions Unique to PowerPC	C-31
C.11	Instructions Unique to PA-RISC 2.0	C-32
C.12	Instructions Unique to ARM	C-35
C.13	Instructions Unique to Thumb	C-36
C.14	Instructions Unique to SuperH	C-37
C.15	Instructions Unique to M32R	C-38
C.16	Instructions Unique to MIPS16	C-38
C.17	Concluding Remarks	C-40
C.18	Acknowledgments	C-41
	References	C-41

C

A Survey of RISC Architectures for Desktop, Server, and Embedded Computers

RISC: any computer announced after 1985.

Steven Przybylski
A Designer of the Stanford MIPS

© 2003 Elsevier Science (USA). All rights reserved.

C.1 Introduction

We cover two groups of reduced instruction set computer (RISC) architectures in this appendix. The first group is the desktop and server RISCs:

- Digital Alpha
- Hewlett-Packard PA-RISC
- IBM and Motorola PowerPC
- Silicon Graphics MIPS
- Sun Microsystems SPARC

The second group is the embedded RISCs:

- Advanced RISC Machines ARM
- Advanced RISC Machines Thumb
- Hitachi SuperH
- Mitsubishi M32R
- Silicon Graphics MIPS16

There has never been another class of computers so similar. This similarity allows the presentation of 10 architectures in about 50 pages. Characteristics of the desktop and server RISCs are found in Figure C.1 and the embedded RISCs in Figure C.2.

Notice that the embedded RISCs tend to have 8 to 16 general-purpose registers while the desktop/server RISCs have 32, and that the length of instructions is 16 to 32 bits in embedded RISCs but always 32 bits in desktop/server RISCs.

Although shown as separate embedded instruction set architectures, Thumb and MIPS16 are really optional modes of ARM and MIPS invoked by call instructions. When in this mode they execute a subset of the native architecture using 16-bit-long instructions. These 16-bit instruction sets are not intended to be full architectures, but they are enough to encode most procedures. Both machines expect procedures to be homogeneous, with all instructions in either 16-bit mode or 32-bit mode. Programs will consist of procedures in 16-bit mode for density or in 32-bit mode for performance.

One complication of this description is that some of the older RISCs have been extended over the years. We decided to describe the latest version of the architectures: Alpha version 3, MIPS64, PA-RISC 2.0, and SPARC version 9 for the desktop/server; ARM version 4, Thumb version 1, Hitachi SuperH SH-3, M32R version 1, and MIPS16 version 1 for the embedded ones.

The remaining sections proceed as follows. After discussing the addressing modes and instruction formats of our RISC architectures, we present the survey of the instructions in five steps:

	Alpha	MIPS I	PA-RISC 1.1	PowerPC	SPARC v.8
Date announced	1992	1986	1986	1993	1987
Instruction size (bits)	32	32	32	32	32
Address space (size, model)	64 bits, flat	32 bits, flat	48 bits, segmented	32 bits, flat	32 bits, flat
Data alignment	Aligned	Aligned	Aligned	Unaligned	Aligned
Data addressing modes	1	1	5	4	2
Protection	Page	Page	Page	Page	Page
Minimum page size	8 KB	4 KB	4 KB	4 KB	8 KB
I/O	Memory mapped	Memory mapped	Memory mapped	Memory mapped	Memory mapped
Integer registers (number, model, size)	31 GPR × 64 bits	31 GPR × 32 bits	31 GPR × 32 bits	32 GPR × 32 bits	31 GPR × 32 bits
Separate floating-point registers	31 × 32 or 31 × 64 bits	16 × 32 or 16 × 64 bits	56 × 32 or 28 × 64 bits	32 × 32 or 32 × 64 bits	32 × 32 or 32 × 64 bits
Floating-point format	IEEE 754 single, double	IEEE 754 single, double	IEEE 754 single, double	IEEE 754 single, double	IEEE 754 single, double

Figure C.1 Summary of the first version of five recent architectures for desktops and servers. Except for the number of data address modes and some instruction set details, the integer instruction sets of these architectures are very similar. Contrast this with Figure C.34. Later versions of these architectures all support a flat, 64-bit address space.

	ARM	Thumb	SuperH	M32R	MIPS16
Date announced	1985	1995	1992	1997	1996
Instruction size (bits)	32	16	16	16/32	16/32
Address space (size, model)	32 bits, flat	32 bits, flat	32 bits, flat	32 bits, flat	32/64 bits, flat
Data alignment	Aligned	Aligned	Aligned	Aligned	Aligned
Data addressing modes	6	6	4	3	2
Integer registers (number, model, size)	15 GPR x 32 bits	8 GPR + SP, LR x 32 bits	16 GPR x 32 bits	16 GPR x 32 bits	8 GPR + SP, RA x 32/64 bits
I/O	Memory mapped	Memory mapped	Memory mapped	Memory mapped	Memory mapped

Figure C.2 Summary of five recent architectures for embedded applications. Except for number of data address modes and some instruction set details, the integer instruction sets of these architectures are similar. Contrast this with Figure C.34.

- Instructions found in the MIPS core, which is defined in Chapter 2 of the main text
- Multimedia extensions of the desktop/server RISCs

- Digital signal-processing extensions of the embedded RISCs
- Instructions not found in the MIPS core but found in two or more architectures
- The unique instructions and characteristics of each of the 10 architectures

We give the evolution of the instruction sets in the final section and conclude with a speculation about future directions for RISCs.

C.2 Addressing Modes and Instruction Formats

Figure C.3 shows the data addressing modes supported by the desktop architectures. Since all have one register that always has the value 0 when used in address modes, the absolute address mode with limited range can be synthesized using `zero` as the base in displacement addressing. (This register can be changed by ALU operations in PowerPC; it is always 0 in the other machines.) Similarly, register indirect addressing is synthesized by using displacement addressing with an offset of 0. Simplified addressing modes is one distinguishing feature of RISC architectures.

Figure C.4 shows the data addressing modes supported by the embedded architectures. Unlike the desktop RISCs, these embedded machines do not reserve a register to contain 0. Although most have two to three simple addressing modes, ARM and SuperH have several, including fairly complex calculations. ARM has an addressing mode that can shift one register by any amount, add it to the other registers to form the address, and then update one register with this new address.

References to code are normally PC-relative, although jump register indirect is supported for returning from procedures, for case statements, and for pointer function calls. One variation is that PC-relative branch addresses are shifted left 2 bits before being added to the PC for the desktop RISCs, thereby increasing the branch distance. This works because the length of all instructions for the desktop RISCs is 32 bits and instructions must be aligned on 32-bit words in memory. Embedded architectures with 16-bit-long instructions usually shift the PC-relative address by 1 for similar reasons.

Addressing mode	Alpha	MIPS64	PA-RISC 2.0	PowerPC	SPARC v.9
Register + offset (displacement or based)	X	X	X	X	X
Register + register (indexed)	—	X (FP)	X (Loads)	X	X
Register + scaled register (scaled)	—	—	X	—	—
Register + offset and update register	—	—	X	X	—
Register + register and update register	—	—	X	X	—

Figure C.3 Summary of data addressing modes supported by the desktop architectures. PA-RISC also has short address versions of the offset addressing modes. MIPS64 has indexed addressing for floating-point loads and stores. (These addressing modes are described in Figure 2.6 on page 98.)

Addressing mode	ARM v.4	Thumb	SuperH	M32R	MIPS16
Register + offset (displacement or based)	X	X	X	X	X
Register + register (indexed)	X	X	X	—	—
Register + scaled register (scaled)	X	—	—	—	—
Register + offset and update register	X	—	—	—	—
Register + register and update register	X	—	—	—	—
Register indirect	—	—	X	X	—
Autoincrement, autodecrement	X	X	X	X	—
PC-relative data	X	X (loads)	X	—	X (loads)

Figure C.4 Summary of data addressing modes supported by the embedded architectures. SuperH and M32R have separate register indirect and register + offset addressing modes rather than just putting 0 in the offset of the latter mode. This increases the use of 16-bit instructions in the M32R, and it gives a wider set of address modes to different data transfer instructions in SuperH. To get greater addressing range, ARM and Thumb shift the offset left 1 or 2 bits if the data size is half word or word. (These addressing modes are described in Figure 2.6 on page 98.)

Figure C.5 shows the format of the desktop RISC instructions, which includes the size of the address in the instructions. Each instruction set architecture uses these four primary instruction formats. Figure C.6 shows the six formats for the embedded RISC machines. The desire to have smaller code size via 16-bit instructions leads to more instruction formats.

Figures C.7 and C.8 show the variations in extending constant fields to the full width of the registers. In this subtle point, the RISCs are similar but not identical.

C.3 Instructions: The MIPS Core Subset

The similarities of each architecture allow simultaneous descriptions, starting with the operations equivalent to the MIPS core.

MIPS Core Instructions

Almost every instruction found in the MIPS core is found in the other architectures, as Figures C.9 through C.13 show. (For reference, definitions of the MIPS instructions are found in Section 2.12 and on the back inside cover of the book.) Instructions are listed under four categories: data transfer (Figure C.9); arithmetic, logical (Figure C.10); control (Figure C.11); and floating point (Figure C.12). A fifth category (Figure C.13) shows conventions for register usage and pseudoinstructions on each architecture. If a MIPS core instruction requires a short sequence of instructions in other architectures, these instructions are separated by semicolons in Figures C.9 through C.13. (To avoid confusion, the destination register will always be the leftmost operand in this appendix, independent

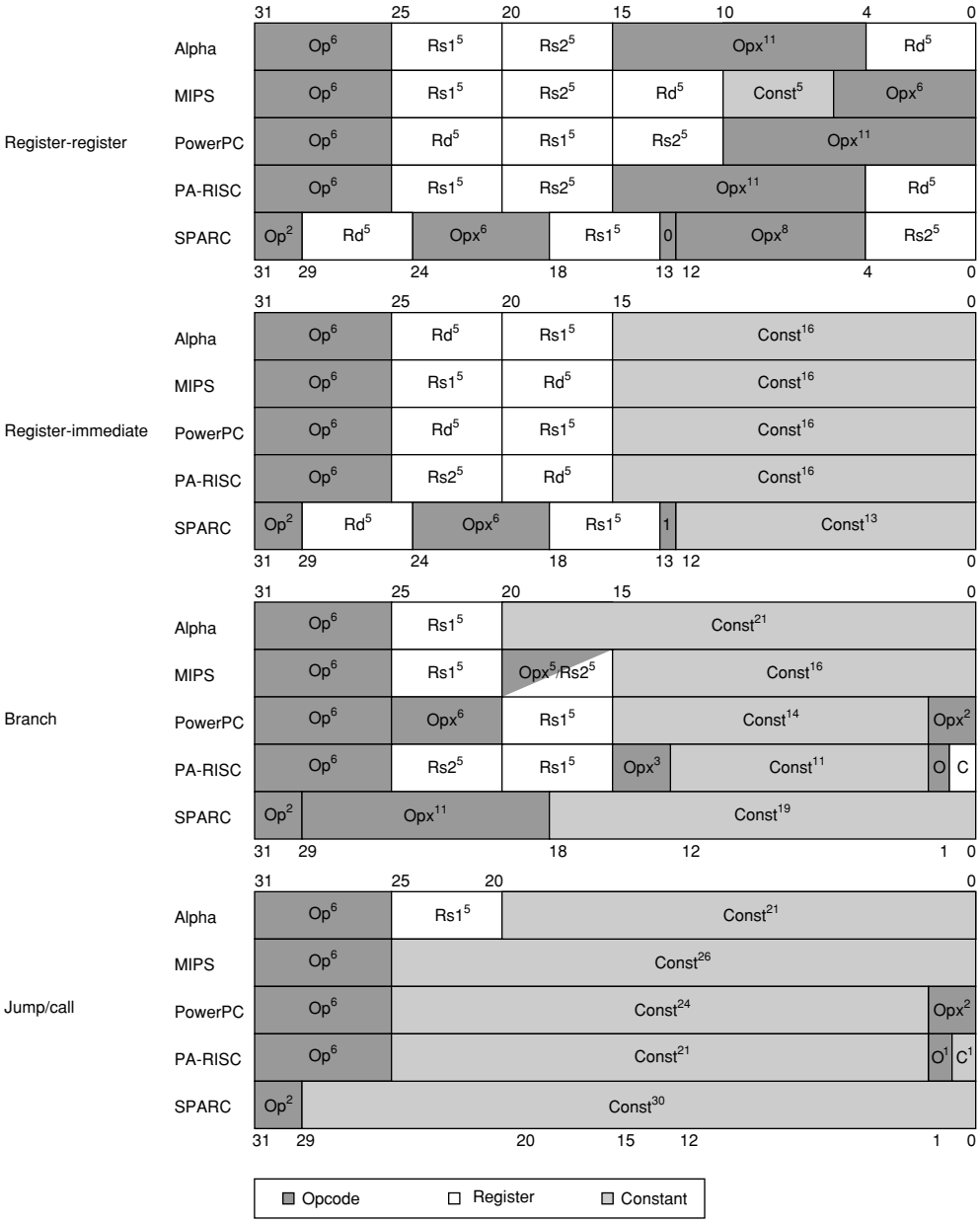


Figure C.5 Instruction formats for desktop/server RISC architectures. These four formats are found in all five architectures. (The superscript notation in this figure means the width of a field in bits.) Although the register fields are located in similar pieces of the instruction, be aware that the destination and two source fields are scrambled. Op = the main opcode, Opx = an opcode extension, Rd = the destination register, Rs1 = source register 1, Rs2 = source register 2, and Const = a constant (used as an immediate or as an address). Unlike the other RISCs, Alpha has a format for immediates in arithmetic and logical operations that is different from the data transfer format shown here. It provides an 8-bit immediate in bits 20 to 13 of the RR format, with bits 12 to 5 remaining as an opcode extension.

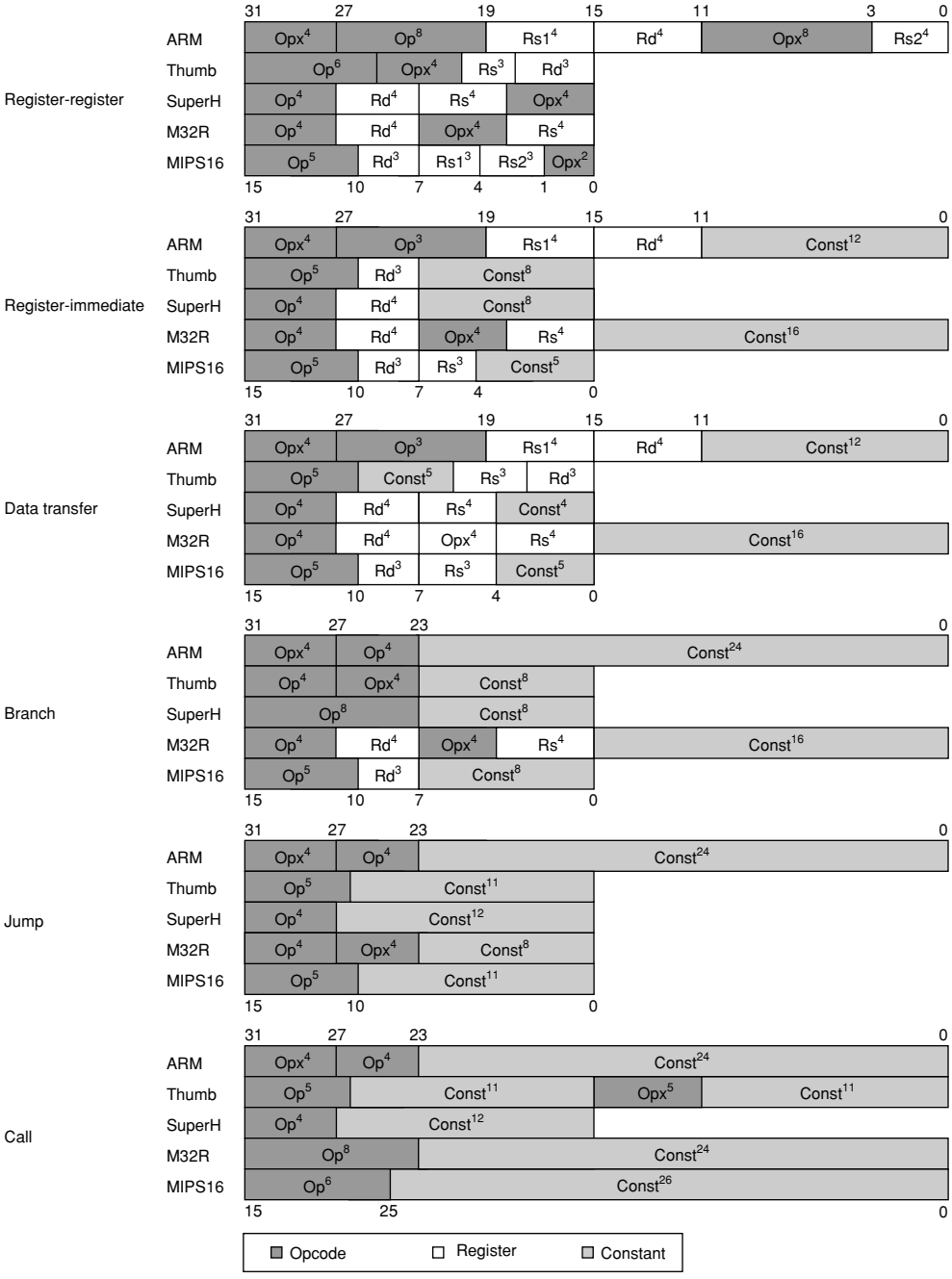


Figure C.6 Instruction formats for embedded RISC architectures. These six formats are found in all five architectures. The notation is the same as Figure C.5. Note the similarities in branch, jump, and call formats, and the diversity in register-register, register-immediate, and data transfer formats. The differences result from whether the architecture has 8 or 16 registers, whether it is a 2- or 3-operand format, and whether the instruction length is 16 or 32 bits.

Format: instruction category	Alpha	MIPS64	PA-RISC 2.0	PowerPC	SPARC v.9
Branch: all	Sign	Sign	Sign	Sign	Sign
Jump/call: all	Sign	—	Sign	Sign	Sign
Register-immediate: data transfer	Sign	Sign	Sign	Sign	Sign
Register-immediate: arithmetic	Zero	Sign	Sign	Sign	Sign
Register-immediate: logical	Zero	Zero	—	Zero	Sign

Figure C.7 Summary of constant extension for desktop RISCs. The constants in the jump and call instructions of MIPS are not sign-extended since they only replace the lower 28 bits of the PC, leaving the upper 4 bits unchanged. PA-RISC has no logical immediate instructions.

Format: instruction category	ARM v.4	Thumb	SuperH	M32R	MIPS16
Branch: all	Sign	Sign	Sign	Sign	Sign
Jump/call: all	Sign	Sign/Zero	Sign	Sign	—
Register-immediate: data transfer	Zero	Zero	Zero	Sign	Zero
Register-immediate: arithmetic	Zero	Zero	Sign	Sign	Zero/Sign
Register-immediate: logical	Zero	—	Zero	Zero	—

Figure C.8 Summary of constant extension for embedded RISCs. The 16-bit-length instructions have much shorter immediates than those of the desktop RISCs, typically only 5 to 8 bits. Most embedded RISCs, however, have a way to get a long address for procedure calls from two sequential half words. The constants in the jump and call instructions of MIPS are not sign-extended since they only replace the lower 28 bits of the PC, leaving the upper 4 bits unchanged. The 8-bit immediates in ARM can be rotated right an even number of bits between 2 and 30, yielding a large range of immediate values. For example, all powers of 2 are immediates in ARM.

of the notation normally used with each architecture.) Figures C.14 through C.17 show the equivalent listing for embedded RISCs. Note that floating point is generally not defined for the embedded RISCs.

Every architecture must have a scheme for compare and conditional branch, but despite all the similarities, each of these architectures has found a different way to perform the operation.

Compare and Conditional Branch

SPARC uses the traditional four condition code bits stored in the program status word: *negative, zero, carry,* and *overflow.* They can be set on any arithmetic or logical instruction; unlike earlier architectures, this setting is optional on each instruction. An explicit option leads to fewer problems in pipelined implementation. Although condition codes can be set as a side effect of an operation, explicit compares are synthesized with a subtract using `r0` as the destination. SPARC conditional branches test condition codes to determine all possible unsigned and signed relations. Floating point uses separate condition codes to encode the IEEE 754

Data transfer (instruction formats)	R-I	R-I	R-I, R-R	R-I, R-R	R-I, R-R
Instruction name	**Alpha**	**MIPS64**	**PA-RISC 2.0**	**PowerPC**	**SPARC v.9**
Load byte signed	LDBU; SEXTB	LB	LDB; EXTRW,S 31,8	LBZ; EXTSB	LDSB
Load byte unsigned	LDBU	LBU	LDB, LDBX, LDBS	LBZ	LDUB
Load half word signed	LDWU; SEXTW	LH	LDH; EXTRW,S 31,16	LHA	LDSH
Load half word unsigned	LDWU	LHU	LDH, LDHX, LDHS	LHZ	LDUH
Load word	LDLS	LW	LDW, LDWX, LDWS	LW	LD
Load SP float	LDS*	LWC1	FLDWX, FLDWS	LFS	LDF
Load DP float	LDT	LDC1	FLDDX, FLDDS	LFD	LDDF
Store byte	STB	SB	STB, STBX, STBS	STB	STB
Store half word	STW	SH	STH, STHX, STHS	STH	STH
Store word	STL	SW	STW, STWX, STWS	STW	ST
Store SP float	STS	SWC1	FSTWX, FSTWS	STFS	STF
Store DP float	STT	SDC1	FSTDX, FSTDS	STFD	STDF
Read, write special registers	MF_, MT_	MF, MT_	MFCTL, MTCTL	MFSPR, MF_, MTSPR, MT_	RD, WR, RDPR, WRPR, LDXFSR, STXFSR
Move integer to FP register	ITOFS	MFC1/ DMFC1	STW; FLDWX	STW; LDFS	ST; LDF
Move FP to integer register	FTTOIS	MTC1/ DMTC1	FSTWX; LDW	STFS; LW	STF; LD

Figure C.9 Desktop RISC data transfer instructions equivalent to MIPS core. A sequence of instructions to synthesize a MIPS instruction is shown separated by semicolons. If there are several choices of instructions equivalent to MIPS core, they are separated by commas. For this figure, half word is 16 bits and word is 32 bits. Note that in Alpha, LDS converts single-precision floating point to double precision and loads the entire 64-bit register.

conditions, requiring a floating-point compare instruction. Version 9 expanded SPARC branches in four ways: a separate set of condition codes for 64-bit operations; a branch that tests the contents of a register and branches if the value is =, not=, <, <=, >=, or <= 0 (see MIPS below); three more sets of floating-point condition codes; and branch instructions that encode static branch prediction.

PowerPC also uses four condition codes: *less than, greater than, equal,* and *summary overflow,* but it has eight copies of them. This redundancy allows the PowerPC instructions to use different condition codes without conflict, essentially giving PowerPC eight extra 4-bit registers. Any of these eight condition codes can be the target of a compare instruction, and any can be the source of a conditional branch. The integer instructions have an option bit that behaves as if the integer op is followed by a compare to zero that sets the first condition "register." PowerPC also lets the second "register" be optionally set by floating-point instructions. PowerPC provides logical operations among these eight 4-bit condition code registers (CRAND, CROR, CRXOR, CRNAND, CRNOR, CREQV), allowing more complex conditions to be tested by a single branch.

Arithmetic/logical (instruction formats)	R-R, R-I	R-R, R-I	R-R, R-I	R-R, R-I	R-R, R-I
Instruction name	**Alpha**	**MIPS64**	**PA-RISC 2.0**	**PowerPC**	**SPARC v.9**
Add	ADDL	ADDU, ADDU	ADDL, LDO, ADDI, UADDCM	ADD, ADDI	ADD
Add (trap if overflow)	ADDLV	ADD, ADDI	ADDO, ADDIO	ADDO; MCRXR; BC	ADDcc; TVS
Sub	SUBL	SUBU	SUB, SUBI	SUBF	SUB
Sub (trap if overflow)	SUBLV	SUB	SUBTO, SUBIO	SUBF/oe	SUBcc; TVS
Multiply	MULL	MULT, MULTU	SHiADD;...; (i=1,2,3)	MULLW, MULLI	MULX
Multiply (trap if overflow)	MULLV	—	SHiADDO;...;	—	—
Divide	—	DIV, DIVU	DS;...; DS	DIVW	DIVX
Divide (trap if overflow)	—	—	—	—	—
And	AND	AND, ANDI	AND	AND, ANDI	AND
Or	BIS	OR, ORI	OR	OR, ORI	OR
Xor	XOR	XOR, XORI	XOR	XOR, XORI	XOR
Load high part register	LDAH	LUI	LDIL	ADDIS	SETHI (B fmt.)
Shift left logical	SLL	SLLV, SLL	DEPW, Z 31-i,32-i	RLWINM	SLL
Shift right logical	SRL	SRLV, SRL	EXTRW, U 31, 32-i	RLWINM 32-i	SRL
Shift right arithmetic	SRA	SRAV, SRA	EXTRW, S 31, 32-i	SRAW	SRA
Compare	CMPEQ, CMPLT, CMPLE	SLT/U, SLTI/U	COMB	CMP(I)CLR	SUBcc r0,...

Figure C.10 Desktop RISC arithmetic/logical instructions equivalent to MIPS core. Dashes mean the operation is not available in that architecture, or not synthesized in a few instructions. Such a sequence of instructions is shown separated by semicolons. If there are several choices of instructions equivalent to MIPS core, they are separated by commas. Note that in the "Arithmetic/logical" category all machines but SPARC use separate instruction mnemonics to indicate an immediate operand; SPARC offers immediate versions of these instructions but uses a single mnemonic. (Of course these are separate opcodes!)

MIPS uses the contents of registers to evaluate conditional branches. Any two registers can be compared for equality (BEQ) or inequality (BNE), and then the branch is taken if the condition holds. The set-on-less-than instructions (SLT, SLTI, SLTU, SLTIU) compare two operands and then set the destination register to 1 if less and to 0 otherwise. These instructions are enough to synthesize the full set of relations. Because of the popularity of comparisons to 0, MIPS includes special compare-and-branch instructions for all such comparisons: greater than or equal to zero (BGEZ), greater than zero (BGTZ), less than or equal to zero (BLEZ), and less than zero (BLTZ). Of course, equal and not equal to zero can be synthesized using r0 with BEQ and BNE. Like SPARC, MIPS I uses a condition code for floating point with separate floating-point compare and branch instructions;

Control (instruction formats)	B, J/C	B, J/C	B, J/C	B, J/C	B, J/C
Instruction name	**Alpha**	**MIPS64**	**PA-RISC 2.0**	**PowerPC**	**SPARC v.9**
Branch on integer compare	B_ (<, >, <=, >=, =, not=)	BEQ, BNE, B_Z (<, >, <=, >=)	COMB, COMIB	BC	BR_Z, BPcc (<, >, <=, >=, =, not=)
Branch on floating-point compare	FB_(<, >, <=, >=, =, not=)	BC1T, BC1F	FSTWX f0; LDW t; BB t	BC	FBPfcc (<, >, <=, >=, =,...)
Jump, jump register	BR, JMP	J, JR	BL r0, BLR r0	B, BCLR, BCCTR	BA, JMPL r0,...
Call, call register	BSR	JAL, JALR	BL, BLE	BL, BLA, BCLRL, BCCTRL	CALL, JMPL
Trap	CALL_PAL GENTRAP	BREAK	BREAK	TW, TWI	Ticc, SIR
Return from interrupt	CALL_PAL REI	JR; ERET	RFI, RFIR	RFI	DONE, RETRY, RETURN

Figure C.11 Desktop RISC control instructions equivalent to MIPS core. If there are several choices of instructions equivalent to MIPS core, they are separated by commas.

Floating point (instruction formats)	R-R	R-R	R-R	R-R	R-R
Instruction name	**Alpha**	**MIPS64**	**PA-RISC 2.0**	**PowerPC**	**SPARC v.9**
Add single, double	ADDS, ADDT	ADD.S, ADD.D	FADD FADD/dbl	FADDS, FADD	FADDS, FADDD
Subtract single, double	SUBS, SUBT	SUB.S, SUB.D	FSUB FSUB/dbl	FSUBS, FSUB	FSUBS, FSUBD
Multiply single, double	MULS, MULT	MUL.S, MUL.D	FMPY FMPY/dbl	FMULS, FMUL	FMULS, FMULD
Divide single, double	DIVS, DIVT	DIV.S, DIV.D	FDIV, FDIV/dbl	FDIVS, FDIV	FDIVS, FDIVD
Compare	CMPT_ (=, <, <=, UN)	C_.S, C_.D (<, >, <=, >=, =,...)	FCMP, FCMP/dbl (<, =, >)	FCMP	FCMPS, FCMPD
Move R-R	ADDT Fd, F31, Fs	MOV.S, MOV.D	FCPY	FMV	FMOVS/D/Q
Convert (single, double, integer) to (single, double, integer)	CVTST, CVTTS, CVTTQ, CVTQS, CVTQT	CVT.S.D, CVT.D.S, CVT.S.W, CVT.D.W, CVT.W.S, CVT.W.D	FCNVFF,s,d FCNVFF,d,s FCNVXF,s,s FCNVXF,d,d FCNVFX,s,s FCNVFX,d,s	—, FRSP, —, FCTIW, —, —	FSTOD, FDTOS, FSTOI, FDTOI, FITOS, FITOD

Figure C.12 Desktop RISC floating-point instructions equivalent to MIPS core. Dashes mean the operation is not available in that architecture, or not synthesized in a few instructions. If there are several choices of instructions equivalent to MIPS core, they are separated by commas.

Conventions	Alpha	MIPS64	PA-RISC 2.0	PowerPC	SPARC v.9
Register with value 0	r31 (source)	r0	r0	r0 (addressing)	r0
Return address register	(any)	r31	r2, r31	link (special)	r31
No-op	LDQ_U r31,...	SLL r0, r0, r0	OR r0, r0, r0	ORI r0, r0, #0	SETHI r0, 0
Move R-R integer	BIS..., r31,...	ADD..., r0,...	OR..., r0,...	OR rx, ry, ry	OR..., r0,...
Operand order	OP Rs1, Rs2, Rd	OP Rd, Rs1, Rs2	OP Rs1, Rs2, Rd	OP Rd, Rs1, Rs2	OP Rs1, Rs2, Rd

Figure C.13 Conventions of desktop RISC architectures equivalent to MIPS core.

Instruction name	ARM v.4	Thumb	SuperH	M32R	MIPS16
Data transfer (instruction formats)	**DT**	**DT**	**DT**	**DT**	**DT**
Load byte signed	LDRSB	LDRSB	MOV.B	LDB	LB
Load byte unsigned	LDRB	LDRB	MOV.B; EXTU.B	LDUB	LBU
Load half word signed	LDRSH	LDRSH	MOV.W	LDH	LH
Load half word unsigned	LDRH	LDRH	MOV.W; EXTU.W	LDUH	LHU
Load word	LDR	LDR	MOV.L	LD	LW
Store byte	STRB	STRB	MOV.B	STB	SB
Store half word	STRH	STRH	MOV.W	STH	SH
Store word	STR	STR	MOV.L	ST	SW
Read, write special registers	MRS, MSR	—[1]	LDC, STC	MVFC, MVTC	MOVE

Figure C.14 Embedded RISC data transfer instructions equivalent to MIPS core. A sequence of instructions to synthesize a MIPS instruction is shown separated by semicolons. Note that floating point is generally not defined for the embedded RISCs. Thumb and MIPS16 are just 16-bit instruction subsets of the ARM and MIPS architectures, so machines can switch modes and execute the full instruction set. We use —[1] to show sequences that are available in 32-bit mode but not 16-bit mode in Thumb or MIPS16.

MIPS IV expanded this to eight floating-point condition codes, with the floating-point comparisons and branch instructions specifying the condition to set or test.

Alpha compares (CMPEQ, CMPLT, CMPLE, CMPULT, CMPULE) test two registers and set a third to 1 if the condition is true and to 0 otherwise. Floating-point compares (CMTEQ, CMTLT, CMTLE, CMTUN) set the result to 2.0 if the condition holds and to 0 otherwise. The branch instructions compare one register to 0 (BEQ, BGE, BGT, BLE, BLT, BNE) or its least-significant bit to 0 (BLBC, BLBS) and then branch if the condition holds.

PA-RISC has many branch options, which we'll see in Section C.8. The most straightforward is a compare and branch instruction (COMB), which compares two registers, branches depending on the standard relations, and then tests the least-significant bit of the result of the comparison.

Arithmetic/logical (instruction formats)	R-R, R-I	R-R, R-I	R-R, R-I	R-R, R-I	R-R, R-I
Instruction name	**ARM v.4**	**Thumb**	**SuperH**	**M32R**	**MIPS16**
Add	ADD	ADD	ADD	ADD, ADDI, ADD3	ADDU, ADDIU
Add (trap if overflow)	ADDS; SWIVS	ADD; BVC .+4; SWI	ADDV	ADDV, ADDV3	—[1]
Subtract	SUB	SUB	SUB	SUB	SUBU
Subtract (trap if overflow)	SUBS; SWIVS	SUB; BVC .+1; SWI	SUBV	SUBV	—[1]
Multiply	MUL	MUL	MUL	MUL	MULT, MULTU
Multiply (trap if overflow)					—
Divide	—	—	DIV1, DIVoS, DIVoU	DIV, DIVU	DIV, DIVU
Divide (trap if overflow)	—	—			—
And	AND	AND	AND	AND, AND3	AND
Or	ORR	ORR	OR	OR, OR3	OR
Xor	EOR	EOR	XOR	XOR, XOR3	XOR
Load high part register	—	—		SETH	—[1]
Shift left logical	LSL[3]	LSL[2]	SHLL, SHLLn	SLL, SLLI, SLL3	SLLV, SLL
Shift right logical	LSR[3]	LSR[2]	SHRL, SHRLn	SRL, SRLI, SRL3	SRLV, SRL
Shift right arithmetic	ASR[3]	ASR[2]	SHRA, SHAD	SRA, SRAI, SRA3	SRAV, SRA
Compare	CMP,CMN, TST,TEQ	CMP, CMN, TST	CMP/cond, TST	CMP/I, CMPU/I	CMP/I[2], SLT/I, SLT/IU

Figure C.15 Embedded RISC arithmetic/logical instructions equivalent to MIPS core. Dashes mean the operation is not available in that architecture, or not synthesized in a few instructions. Such a sequence of instructions is shown separated by semicolons. If there are several choices of instructions equivalent to MIPS core, they are separated by commas. Thumb and MIPS16 are just 16-bit instruction subsets of the ARM and MIPS architectures, so machines can switch modes and execute the full instruction set. We use —[1] to show sequences that are available in 32-bit mode but not 16-bit mode in Thumb or MIPS16. The superscript 2 shows new instructions found only in 16-bit mode of Thumb or MIPS16, such as CMP/I[2]. ARM includes shifts as part of every data operation instruction, so the shifts with superscript 3 are just a variation of a move instruction, such as LSR[3].

ARM is similar to SPARC, in that it provides fourtraditional condition codes that are optionally set. CMP subtracts one operand from the other and the difference sets the condition codes. Compare negative (CMN) *adds* one operand to the other, and the sum sets the condition codes. TST performs logical AND on the two operands to set all condition codes but overflow, while TEQ uses exclusive OR to

Control (instruction formats)	B, J, C	B, J, C	B, J, C	B, J, C	B, J, C
Instruction name	**ARM v.4**	**Thumb**	**SuperH**	**M32R**	**MIPS16**
Branch on integer compare	B/cond	B/cond	BF, BT	BEQ, BNE, BC,BNC, B__Z	BEQZ[2], BNEZ[2], BTEQZ[2], BTNEZ[2]
Jump, jump register	MOV pc,ri	MOV pc,ri	BRA, JMP	BRA, JMP	B[2], JR
Call, call register	BL	BL	BSR, JSR	BL, JL	JAL, JALR, JALX[2]
Trap	SWI	SWI	TRAPA	TRAP	BREAK
Return from interrupt	MOVS pc, r14	—[1]	RTS	RTE	—[1]

Figure C.16 Embedded RISC control instructions equivalent to MIPS core. Thumb and MIPS16 are just 16-bit instruction subsets of the ARM and MIPS architectures, so machines can switch modes and execute the full instruction set. We use —[1] to show sequences that are available in 32-bit mode but not 16-bit mode in Thumb or MIPS16. The superscript 2 shows new instructions found only in 16-bit mode of Thumb or MIPS16, such as BTEQZ[2].

Conventions	ARM v.4	Thumb	SuperH	M32R	MIPS16
Return address reg.	R14	R14	PR (special)	R14	RA (special)
No-op	MOV r0,r0	MOV r0,r0	NOP	NOP	SLL r0, r0
Operands, order	OP Rd, Rs1, Rs2	OP Rd, Rs1	OP Rs1, Rd	OP Rd, Rs1	OP Rd, Rs1, Rs2

Figure C.17 Conventions of embedded RISC instructions equivalent to MIPS core.

set the first three condition codes. Like SPARC, the conditional version of the ARM branch instruction tests condition codes to determine all possible unsigned and signed relations. As we shall see in Section C.9, one unusual feature of ARM is that every instruction has the option of executing conditionally depending on the condition codes. (This bears similarities to the annulling option of PA-RISC, seen in Section C.8.)

Not surprisingly, Thumb follows ARM. Differences are that setting condition codes are not optional, the TEQ instruction is dropped, and there is no conditional execution of instructions.

The Hitachi SuperH uses a single T-bit condition that is set by compare instructions. Two branch instructions decide to branch if either the T bit is 1 (BT) or the T bit is 0 (BF). The two flavors of branches allow fewer comparison instructions.

Mitsubishi M32R also offers a single condition code bit (C) used for signed and unsigned comparisons (CMP, CMPI, CMPU, CMPUI) to see if one register is less than the other or not, similar to the MIPS set-on-less-than instructions. Two branch instructions test to see if the C bit is 1 or 0: BC and BNC. The M32R also includes instructions to branch on equality or inequality of registers (BEQ and

BNE) and all relations of a register to 0 (BGEZ, BGTZ, BLEZ, BLTZ, BEQZ, BNEZ). Unlike BC and BNC, these last instructions are all 32 bits wide.

MIPS16 keeps set-on-less-than instructions (SLT, SLTI, SLTU, SLTIU), but instead of putting the result in one of the eight registers, it is placed in a special register named T. MIPS16 is always implemented in machines that also have the full 32-bit MIPS instructions and registers; hence, register T is really register 24 in the full MIPS architecture. The MIPS16 branch instructions test to see if a register is or is not equal to zero (BEQZ and BNEZ). There are also instructions that branch if register T is or is not equal to zero (BTEQZ and BTNEZ). To test if two registers are equal, MIPS added compare instructions (CMP, CMPI) that compute the exclusive OR of two registers and place the result in register T. Compare was added since MIPS16 left out instructions to compare and branch if registers are equal or not (BEQ and BNE).

Figures C.18 and C.19 summarize the schemes used for conditional branches.

	Alpha	MIPS64	PA-RISC 2.0	PowerPC	SPARC v.9
Number of condition code bits (integer and FP)	0	8 FP	8 FP	8 × 4 both	2 × 4 integer, 4 × 2 FP
Basic compare instructions (integer and FP)	1 integer, 1 FP	1 integer, 1 FP	4 integer, 2 FP	4 integer, 2 FP	1 FP
Basic branch instructions (integer and FP)	1	2 integer, 1 FP	7 integer	1 both	3 integer, 1 FP
Compare register with register/const and branch	—	=, not=	=, not=, <, <=, >, >=, even, odd	—	—
Compare register to zero and branch	=, not=, <, <=, >, >=, even, odd	=, not=, <, <=, >, >=	=, not=, <, <=, >, >=, even, odd	—	=, not=, <, <=, >, >=

Figure C.18 Summary of five desktop RISC approaches to conditional branches. Floating-point branch on PA-RISC is accomplished by copying the FP status register into an integer register and then using the branch on bit instruction to test the FP comparison bit. Integer compare on SPARC is synthesized with an arithmetic instruction that sets the condition codes using r0 as the destination.

	ARM v.4	Thumb	SuperH	M32R	MIPS16
Number of condition code bits	4	4	1	1	1
Basic compare instructions	4	3	2	2	2
Basic branch instructions	1	1	2	3	2
Compare register with register/const and branch	—	—	=, >, >=	=, not=	—
Compare register to zero and branch	—	—	=, >, >=	=, not=, <, <=, >, >=	=, not=

Figure C.19 Summary of five embedded RISC approaches to conditional branches.

C.4 Instructions: Multimedia Extensions of the Desktop/ Server RISCs

Since every desktop microprocessor by definition has its own graphical displays, as transistor budgets increased it was inevitable that support would be added for graphics operations. Many graphics systems use 8 bits to represent each of the three primary colors plus 8 bits for a location of a pixel.

The addition of speakers and microphones for teleconferencing and video games suggested support of sound as well. Audio samples need more than 8 bits of precision, but 16 bits are sufficient.

Every microprocessor has special support so that bytes and half words take up less space when stored in memory, but due to the infrequency of arithmetic operations on these data sizes in typical integer programs, there is little support beyond data transfers. The architects of the Intel i860, which was justified as a graphical accelerator within the company, recognized that many graphics and audio applications would perform the same operation on vectors of these data. Although a vector unit was beyond the transistor budget of the i860 in 1989, by partitioning the carry chains within a 64-bit ALU (see Section H.8), it could perform simultaneous operations on short vectors of eight 8-bit operands, four 16-bit operands, or two 32-bit operands. The cost of such partitioned ALUs was small. Applications that lend themselves to such support include MPEG (video), games like DOOM (3D graphics), Adobe Photoshop (digital photography), and teleconferencing (audio and image processing).

Like a virus, over time such multimedia support has spread to nearly every desktop microprocessor. HP was the first successful desktop RISC to include such support. As we shall see, this virus spread unevenly. The PowerPC is the only holdout, and rumors are that it is "running a fever."

These extensions have been called subword parallelism, vector, or SIMD (single instruction, multiple data) (see Chapter 2). Since Intel marketing uses SIMD to describe the MMX extension of the 80x86, that has become the popular name. Figure C.20 summarizes the support by architecture.

From Figure C.20 you can see that in general MIPS MDMX works on 8 bytes or 4 half words per instruction, HP PA-RISC MAX2 works on 4 half words, SPARC VIS works on 4 half words or 2 words, and Alpha doesn't do much. The Alpha MAX operations are just byte versions of compare, min, max, and absolute difference, leaving it up to software to isolate fields and perform parallel adds, subtracts, and multiplies on bytes and half words. MIPS also added operations to work on two 32-bit floating-point operands per cycle, but they are considered part of MIPS V and not simply multimedia extensions (see Section C.7).

One feature not generally found in general-purpose microprocessors is saturating operations. Saturation means that when a calculation overflows, the result is set to the largest positive number or most negative number, rather than a modulo calculation as in two's complement arithmetic. Commonly found in digital signal processors (see the next section), these saturating operations are helpful in routines for filtering.

Instruction category	Alpha MAX	MIPS MDMX	PA-RISC MAX2	PowerPC	SPARC VIS
Add/subtract		8B, 4H	4H		4H, 2W
Saturating add/sub		8B, 4H	4H		
Multiply		8B, 4H			4B/H
Compare	8B (>=)	8B, 4H (=,<,<=)			4H, 2W (=, not=, >, <=)
Shift right/left		8B, 4H	4H		
Shift right arithmetic		4H	4H		
Multiply and add		8B, 4H			
Shift and add (saturating)			4H		
And/or/xor	8B, 4H, 2W	8B, 4H, 2W	8B, 4H, 2W		8B, 4H, 2W
Absolute difference	8B				8B
Max/min	8B, 4W	8B, 4H			
Pack (2*n* bits --> *n* bits)	2W->2B, 4H->4B	2*2W->4H, 2*4H->8B	2*4H->8B		2W->2H, 2W->2B, 4H->4B
Unpack/merge	2B->2W, 4B->4H	2*4B->8B, 2*2H->4H			4B->4H, 2*4B->8B
Permute/shuffle		8B, 4H	4H		
Register sets	Integer	Fl. Pt. + 192b Acc.	Integer		Fl. Pt.

Figure C.20 Summary of multimedia support for desktop RISCs. B stands for byte (8 bits), H for half word (16 bits), and W for word (32 bits). Thus 8B means an operation on 8 bytes in a single instruction. Pack and unpack use the notation 2*2W to mean 2 operands each with 2 words. Note that MDMX has vector/scalar operations, where the scalar is specified as an element of one of the vector registers. This table is a simplification of the full multimedia architectures, leaving out many details. For example, MIPS MDMX includes instructions to multiplex between two operands, HP MAX2 includes an instruction to calculate averages, and SPARC VIS includes instructions to set registers to constants. Also, this table does not include the memory alignment operation of MDMX, MAX, and VIS.

These machines largely used existing register sets to hold operands: integer registers for Alpha and HP PA-RISC and floating-point registers for MIPS and Sun. Hence data transfers are accomplished with standard load and store instructions. MIPS also added a 192-bit (3*64) wide register to act as an accumulator for some operations. By having 3 times the native data width, it can be partitioned to accumulate either 8 bytes with 24 bits per field or 4 half words with 48 bits per field. This wide accumulator can be used for add, subtract, and multiply/add instructions. MIPS claims performance advantages of 2 to 4 times for the accumulator.

Perhaps the surprising conclusion of this table is the lack of consistency. The only operations found on all four are the logical operations (`AND`, `OR`, `XOR`), which do not need a partitioned ALU. If we leave out the frugal Alpha, then the only other common operations are parallel adds and subtracts on 4 half words.

Each manufacturer states that these are instructions intended to be used in hand-optimized subroutine libraries, an intention likely to be followed, as a compiler that works well with all desktop RISCs' multimedia extensions would be challenging.

C.5 Instructions: Digital Signal-Processing Extensions of the Embedded RISCs

One feature found in every digital signal processor (DSP) architecture is support for integer multiply-accumulate. The multiplies tend to be on shorter words than regular integers, such as 16-bits, and the accumulator tends to be on longer words, such as 64 bits. The reason for multiply-accumulate is to efficiently implement digital filters, common in DSP applications. Since Thumb and MIPS16 are subset architectures, they do not provide such support. Instead, programmers should use the DSP or multimedia extensions found in the 32-bit mode instructions of ARM and MIPS64.

Figure C.21 shows the size of the multiply, the size of the accumulator, and the operations and instruction names for the embedded RISCs. Machines with accumulator sizes greater than 32 and less than 64 bits will force the upper bits to remain as the sign bits, thereby "saturating" the add to set to maximum and minimum fixed-point values if the operations overflow.

	ARM v.4	Thumb	SuperH	M32R	MIPS16
Size of multiply	32B × 32B	—	32B × 32B, 16B × 16B	32B × 16B, 16B × 16B	—
Size of accumulator	32B/64B	—	32B/42B, 48B/64B	56B	—
Accumulator name	Any GPR or pairs of GPRs	—	`MACH, MACL`	`ACC`	—
Operations	32B/64B product + 64B accumulate signed/unsigned	—	32B product + 42B/32B accumulate (operands in memory); 64B product + 64B/48B accumulate (operands in memory); clear MAC	32B/48B product + 64B accumulate, round, move	—
Corresponding instruction names	MLA, SMLAL, UMLAL	—	`MAC, MACS, MAC.L, MAC.LS, CLRMAC`	`MACHI/MACLO, MACWHI/ MACWLO, RAC, RACH, MVFACHI/MVFACLO, MVTACHI/MVTACLO`	—

Figure C.21 Summary of five embedded RISC approaches to multiply-accumulate.

C.6 Instructions: Common Extensions to MIPS Core

Figures C.22 through C.28 list instructions not found in Figures C.9 through C.17 in the same four categories. Instructions are put in these lists if they appear in more than one of the standard architectures. The instructions are defined using the hardware description language defined in Figure C.29.

Although most of the categories are self-explanatory, a few bear comment:

- The "atomic swap" row means a primitive that can exchange a register with memory without interruption. This is useful for operating system semaphores in a uniprocessor as well as for multiprocessor synchronization (see Section 6.7).
- The 64-bit data transfer and operation rows show how MIPS, PowerPC, and SPARC define 64-bit addressing and integer operations. SPARC simply defines all register and addressing operations to be 64 bits, adding only special instructions for 64-bit shifts, data transfers, and branches. MIPS includes

Name	Definition	Alpha	MIPS64	PA-RISC 2.0	PowerPC	SPARC v.9
Atomic swap R/M (for locks and semaphores)	Temp<---Rd; Rd<---Mem[x]; Mem[x]<---Temp	LDL/Q_L; STL/Q_C	LL; SC	— (see C.8)	LWARX; STWCX	CASA, CASX
Load 64-bit integer	Rd<---$_{64}$ Mem[x]	LDQ	LD	LDD	LD	LDX
Store 64-bit integer	Mem[x]<---$_{64}$ Rd	STQ	SD	STD	STD	STX
Load 32-bit integer unsigned	$Rd_{32..63}$<---$_{32}$ Mem[x]; $Rd_{0..31}$<---$_{32}$ 0	LDL; EXTLL	LWU	LDW	LWZ	LDUW
Load 32-bit integer signed	$Rd_{32..63}$<---$_{32}$ Mem[x]; $Rd_{0..31}$<---$_{32}$ $Mem[x]_0^{32}$	LDL	LW	LDW; EXTRD,S 63, 8	LWA	LDSW
Prefetch	Cache[x]<---*hint*	FETCH, FETCH_M*	PREF, PREFX	LDD, r0 LDW, r0	DCBT, DCBTST	PRE-FETCH
Load coprocessor	Coprocessor<--- Mem[x]	—	LWCi	CLDWX, CLDWS	—	—
Store coprocessor	Mem[x]<--- Coprocessor	—	SWCi	CSTWX, CSTWS	—	—
Endian	(Big/Little Endian?)	Either	Either	Either	Either	Either
Cache flush	(Flush cache block at this address)	ECB	CPOop	FDC, FIC	DCBF	FLUSH
Shared-memory synchronization	(All prior data transfers complete before next data transfer may start)	WMB	SYNC	SYNC	SYNC	MEMBAR

Figure C.22 Data transfer instructions not found in MIPS core but found in two or more of the five desktop architectures. The load linked/store conditional pair of instructions gives Alpha and MIPS atomic operations for semaphores, allowing data to be read from memory, modified, and stored without fear of interrupts or other machines accessing the data in a multiprocessor (see Chapter 6). Prefetching in the Alpha to external caches is accomplished with FETCH and FETCH_M; on-chip cache prefetches use LD_Q A, R31, and LD_Y A. F31 is used in the Alpha 21164 (see Bhandarkar [1995], p. 190).

Name	Definition	Alpha	MIPS64	PA-RISC 2.0	PowerPC	SPARC v.9
64-bit integer arithmetic ops	Rd<---$_{64}$Rs1 op$_{64}$ Rs2	ADD, SUB, MUL	DADD, DSUB DMULT, DDIV	ADD, SUB, SHLADD, DS	ADD, SUBF, MULLD, DIVD	ADD, SUB, MULX, S/UDIVX
64-bit integer logical ops	Rd<---$_{64}$Rs1 op$_{64}$ Rs2	AND, OR, XOR	AND, OR, XOR	AND, OR, XOR	AND, OR, XOR	AND, OR, XOR
64-bit shifts	Rd<---$_{64}$Rs1 op$_{64}$ Rs2	SLL, SRA, SRL	DSLL/V, DSRA/V, DSRL/V	DEPD,Z EXTRD,S EXTRD,U	SLD, SRAD, SRLD	SLLX, SRAX, SRLX
Conditional move	if (cond) Rd<---Rs	CMOV_	MOVN/Z	SUBc, n; ADD	—	MOVcc, MOVr
Support for multiword integer add	CarryOut, Rd <--- Rs1 + Rs2 + OldCarryOut	—	ADU; SLTU; ADDU, DADU; SLTU; DADDU	ADDC	ADDC, ADDE	ADDcc
Support for multiword integer sub	CarryOut, Rd <--- Rs1 Rs2 + OldCarryOut	—	SUBU; SLTU; SUBU, DSUBU; SLTU; DSUBU	SUBB	SUBFC, SUBFE	SUBcc
And not	Rd <--- Rs1 & ~(Rs2)	BIC	—	ANDCM	ANDC	ANDN
Or not	Rd <--- Rs1 | ~(Rs2)	ORNOT	—	—	ORC	ORN
Add high immediate	Rd$_{0..15}$<---Rs1$_{0..15}$ + (Const<<16);	—	—	ADDIL (R-I)	ADDIS (R-I)	—
Coprocessor operations	(Defined by coprocessor)	—	COPi	COPR,i	—	IMPDEPi

Figure C.23 Arithmetic/logical instructions not found in MIPS core but found in two or more of the five desktop architectures.

Name	Definition	Alpha	MIPS64	PA-RISC 2.0	PowerPC	SPARC v.9
Optimized delayed branches	(Branch not always delayed)	—	BEQL, BNEL, B_ZL (<, >, <=, >=)	COMBT, n, COMBF, n	—	BPcc, A, FPBcc, A
Conditional trap	if (COND) {R31<---PC; PC <---0..0#i}	—	T_,,T_I (=, not=, <, >, <=, >=)	SUBc, n; BREAK	TW, TD, TWI, TDI	Tcc
No. control registers	Misc. regs (virtual memory, interrupts, . . .)	6	equiv. 12	32	33	29

Figure C.24 Control instructions not found in MIPS core but found in two or more of the five desktop architectures.

the same extensions, plus it adds separate 64-bit signed arithmetic instructions. PowerPC adds 64-bit right shift, load, store, divide, and compare and has a separate mode determining whether instructions are interpreted as 32- or 64-bit operations; 64-bit operations will not work in a machine that only supports 32-bit mode. PA-RISC is expanded to 64-bit addressing and operations in version 2.0.

Name	Definition	Alpha	MIPS64	PA-RISC 2.0	PowerPC	SPARC v.9
Multiply and add	Fd <--- (Fs1 × Fs2) + Fs3	—	MADD.S/D	FMPYFADD sgl/dbl	FMADD/S	
Multiply and sub	Fd <--- (Fs1 × Fs2) – Fs3	—	MSUB.S/D		FMSUB/S	
Neg mult and add	Fd <--- -((Fs1 × Fs2) + Fs3)	—	NMADD.S/D	FMPYFNEG sgl/dbl	FNMADD/S	
Neg mult and sub	Fd <--- -((Fs1 × Fs2) – Fs3)	—	NMSUB.S/D		FNMSUB/S	
Square root	Fd <--- SQRT(Fs)	SQRT_	SQRT.S/D	FSQRT sgl/dbl	FSQRT/S	FSQRTS/D
Conditional move	if (cond) Fd<---Fs	FCMOV_	MOVF/T, MOVF/T.S/D	FTESTFCPY	—	FMOVcc
Negate	Fd <--- Fs ^ x80000000	CPYSN	NEG.S/D	FNEG sgl/dbl	FNEG	FNEGS/D/Q
Absolute value	Fd <--- Fs & x7FFFFFFF	—	ABS.S/D	FABS/dbl	FABS	FABSS/D/Q

Figure C.25 Floating-point instructions not found in MIPS core but found in two or more of the five desktop architectures.

Name	Definition	ARM v.4	Thumb	SuperH	M32R	MIPS16
Atomic swap R/M (for semaphores)	Temp<---Rd; Rd<---Mem[x]; Mem[x]<---Temp	SWP, SWPB	—[1]	(see TAS)	LOCK; UNLOCK	—[1]
Memory management unit	Paged address translation	Via coprocessor instructions	—[1]	LDTLB		—[1]
Endian	(Big/Little Endian?)	Either	Either	Either	Big	Either

Figure C.26 Data transfer instructions not found in MIPS core but found in two or more of the five embedded architectures. We use —[1] to show sequences that are available in 32-bit mode but not 16-bit mode in Thumb or MIPS16.

- The "prefetch" instruction supplies an address and hint to the implementation about the data. Hints include whether the data is likely to be read or written soon, likely to be read or written only once, or likely to be read or written many times. Prefetch does not cause exceptions. MIPS has a version that adds two registers to get the address for floating-point programs, unlike non-floating-point MIPS programs. (See Section 5.6 to learn more about prefetching.)
- In the "Endian" row, "Big/Little" means there is a bit in the program status register that allows the processor to act either as Big Endian or Little Endian (see Section 2.3). This can be accomplished by simply complementing some of the least-significant bits of the address in data transfer instructions.

Name	Definition	ARM v.4	Thumb	SuperH	M32R	MIPS16
Load immediate	Rd<---Imm	MOV	MOV	MOV, MOVA	LDI, LD24	LI
Support for multiword integer add	CarryOut, Rd <--- Rd + Rs1 + OldCarryOut	ADCS	ADC	ADDC	ADDX	—[1]
Support for multiword integer sub	CarryOut, Rd <--- Rd – Rs1 + OldCarryOut	SBCS	SBC	SUBC	SUBX	—[1]
Negate	Rd <--- 0 – Rs1		NEG[2]	NEG	NEG	NEG
Not	Rd <--- ~(Rs1)	MVN	MVN	NOT	NOT	NOT
Move	Rd <--- Rs1	MOV	MOV	MOV	MV	MOVE
Rotate right	Rd <--- Rs i, >> $Rd_{0...i-1}$ <--- $Rs_{31-i...31}$	ROR	ROR	ROTR		
And not	Rd <--- Rs1 & ~(Rs2)	BIC	BIC			

Figure C.27 Arithmetic/logical instructions not found in MIPS core but found in two or more of the five embedded architectures. We use —[1] to show sequences that are available in 32-bit mode but not in 16-bit mode in Thumb or MIPS16. The superscript 2 shows new instructions found only in 16-bit mode of Thumb or MIPS16, such as NEG[2].

Name	Definition	ARM v.4	Thumb	SuperH	M32R	MIPS16
No. control registers	Misc. registers	21	29	9	5	36

Figure C.28 Control information in the five embedded architectures.

- The "shared-memory synchronization" helps with cache-coherent multiprocessors: All loads and stores executed before the instruction must complete before loads and stores after it can start. (See Chapter 6.)
- The "coprocessor operations" row lists several categories that allow for the processor to be extended with special-purpose hardware.

One difference that needs a longer explanation is the optimized branches. Figure C.30 shows the options. The Alpha and PowerPC offer branches that take effect immediately, like branches on earlier architectures. To accelerate branches, these machines use branch prediction (see Section 3.4). All the rest of the desktop RISCs offer delayed branches (see Appendix A). The embedded RISCs generally do not support delayed branch, with the exception of SuperH, which has it as an option.

The other three desktop RISCs provide a version of delayed branch that makes it easier to fill the delay slot. The SPARC "annulling" branch executes the instruction in the delay slot only if the branch is taken; otherwise the instruction is annulled. This means the instruction at the target of the branch can safely be copied into the delay slot since it will only be executed if the branch is taken. The restrictions are that the target is not another branch and that the target is known at compile time. (SPARC also offers a nondelayed jump because an unconditional branch with the annul bit set does *not* execute the following instruction.) Later

Notation	Meaning	Example	Meaning
<--	Data transfer. Length of transfer is given by the destination's length; the length is specified when not clear.	`Regs[R1]<--Regs[R2];`	Transfer contents of R2 to R1. Registers have a fixed length, so transfers shorter than the register size must indicate which bits are used.
M	Array of memory accessed in bytes. The starting address for a transfer is indicated as the index to the memory array.	`Regs[R1]<--M[x];`	Place contents of memory location x into R1. If a transfer starts at `M[i]` and requires 4 bytes, the transferred bytes are `M[i]`, `M[i+1]`, `M[i+2]`, and `M[i+3]`.
<--n	Transfer an *n*-bit field, used whenever length of transfer is not clear.	$M[y] \leftarrow\!\!-_{16} M[x]$;	Transfer 16 bits starting at memory location x to memory location y. The length of the two sides should match.
X_n	Subscript selects a bit.	$Regs[R1]_0 \leftarrow\!\!- 0$;	Change sign bit of R1 to 0. (Bits are numbered from MSB starting at 0.)
$X_{m..n}$	Subscript selects a field.	$Regs[R3]_{24..31} \leftarrow\!\!- M[x]$;	Moves contents of memory location x into low-order byte of R3.
X^n	Superscript replicates a bit field.	$Regs[R3]_{0..23} \leftarrow\!\!- 0^{24}$;	Sets high-order three bytes of R3 to 0.
##	Concatenates two fields.	$Regs[R3] \leftarrow\!\!- 0^{24} \# \# M[x]$; $F2 \# \# F3 \leftarrow\!\!-_{64} M[x]$;	Moves contents of location x into low byte of R3; clears upper three bytes. Moves 64 bits from memory starting at location x; 1st 32 bits go into F2, 2nd 32 into F3.
*, &	Dereference a pointer; get the address of a variable.	`p*<--&x;`	Assign to object pointed to by p the address of the variable x.
<<, >>	C logical shifts (left, right).	`Regs[R1] << 5`	Shift R1 left 5 bits.
==, !=, >, <, >=, <=	C relational operators; equal, not equal, greater, less, greater or equal, less or equal.	`(Regs[R1]== Regs[R2]) & (Regs[R3]!=Regs[R4])`	True if contents of R1 equal the contents of R2 and contents of R3 do not equal the contents of R4.
&, |, ^, !	C bitwise logical operations: and, or, exclusive or, and complement.	`(Regs[R1] & (Regs[R2]| Regs[R3]))`	Bitwise AND of R1 and bitwise OR of R2 and R3.

Figure C.29 Hardware description notation (and some standard C operators).

versions of the MIPS architecture have added a branch likely instruction that also annuls the following instruction if the branch is not taken. PA-RISC allows almost any instruction to annul the next instruction, including branches. Its "nullifying" branch option will execute the next instruction depending on the direction of the branch and whether it is taken (i.e., if a forward branch is *not* taken or a backward branch is taken). Presumably this choice was made to optimize loops, allowing the instructions following the exit branch and the looping branch to execute in the common case.

Now that we have covered the similarities, we will focus on the unique features of each architecture. We first cover the desktop/server RISCs, ordering them by length of description of the unique features from shortest to longest, and then the embedded RISCs.

	(Plain) branch	Delayed branch	Annulling delayed branch	
Found in architectures	Alpha, PowerPC, ARM, Thumb, SuperH, M32R, MIPS 16	MIPS64, PA-RISC, SPARC, SuperH	MIPS64, SPARC	PA-RISC
Execute following instruction	Only if branch *not* taken	Always	Only if branch taken	If forward branch *not* taken or backward branch taken

Figure C.30 When the instruction following the branch is executed for three types of branches.

C.7 Instructions Unique to MIPS64

MIPS has gone through five generations of instruction sets, and this evolution has generally added features found in other architectures. Here are the salient unique features of MIPS, the first several of which were found in the original instruction set.

Nonaligned Data Transfers

MIPS has special instructions to handle misaligned words in memory. A rare event in most programs, it is included for supporting 16-bit minicomputer applications and for doing `memcpy` and `strcpy` faster. Although most RISCs trap if you try to load a word or store a word to a misaligned address, on all architectures misaligned words can be accessed without traps by using four load byte instructions and then assembling the result using shifts and logical ors. The MIPS load and store word left and right instructions (`LWL`, `LWR`, `SWL`, `SWR`) allow this to be done in just two instructions: `LWL` loads the left portion of the register and `LWR` loads the right portion of the register. `SWL` and `SWR` do the corresponding stores. Figure C.31 shows how they work. There are also 64-bit versions of these instructions.

Remaining Instructions

Below is a list of the remaining unique details of the MIPS64 architecture:

- *NOR*—This logical instruction calculates ~(Rs1 | Rs2).
- *Constant shift amount*—Nonvariable shifts use the 5-bit constant field shown in the register-register format in Figure C.5.
- *SYSCALL*—This special trap instruction is used to invoke the operating system.
- *Move to/from control registers*—`CTCi` and `CFCi` move between the integer registers and control registers.

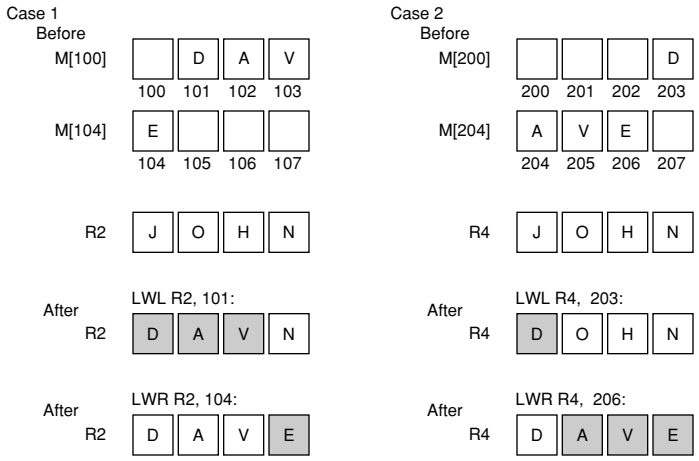


Figure C.31 MIPS instructions for unaligned word reads. This figure assumes operation in Big Endian mode. Case 1 first loads the 3 bytes 101, 102, and 103 into the left of R2, leaving the least-significant byte undisturbed. The following LWR simply loads byte 104 into the least-significant byte of R2, leaving the other bytes of the register unchanged using LWL. Case 2 first loads byte 203 into the most-significant byte of R4, and the following LWR loads the other 3 bytes of R4 from memory bytes 204, 205, and 206. LWL reads the word with the first byte from memory, shifts to the left to discard the unneeded byte(s), and changes only those bytes in Rd. The byte(s) transferred are from the first byte to the lowest-order byte of the word. The following LWR addresses the last byte, right-shifts to discard the unneeded byte(s), and finally changes only those bytes of Rd. The byte(s) transferred are from the last byte up to the highest-order byte of the word. Store word left (SWL) is simply the inverse of LWL, and store word right (SWR) is the inverse of LWR. Changing to Little Endian mode flips which bytes are selected and discarded. (If big-little, left-right, load-store seem confusing, don't worry; they work!)

- *Jump/call not PC-relative*—The 26-bit address of jumps and calls is not added to the PC. It is shifted left 2 bits and replaces the lower 28 bits of the PC. This would only make a difference if the program were located near a 256-MB boundary.
- *TLB instructions*—Translation lookaside buffer (TLB) misses were handled in software in MIPS I, so the instruction set also had instructions for manipulating the registers of the TLB (see Chapter 5 for more on TLBs). These registers are considered part of the "system coprocessor." Since MIPS I the instructions differ among versions of the architecture; they are more part of the implementations than part of the instruction set architecture.
- *Reciprocal and reciprocal square root*—These instructions, which do not follow IEEE 754 guidelines of proper rounding, are included apparently for applications that value speed of divide and square root more than they value accuracy.

- *Conditional procedure call instructions*—BGEZAL saves the return address and branches if the content of Rs1 is greater than or equal to zero, and BLTZAL does the same for less than zero. The purpose of these instructions is to get a PC-relative call. (There are "likely" versions of these instructions as well.)
- *Parallel single-precision floating-point operations*—As well as extending the architecture with parallel integer operations in MDMX, MIPS64 also supports two parallel 32-bit floating-point operations on 64-bit registers in a single instruction. "Paired single" operations include add (ADD.PS), subtract (SUB.PS), compare (C.__.PS), convert (CVT.PS.S, CVT.S.PL, CVT.S.PU), negate (NEG.PS), absolute value (ABS.PS), move (MOV.PS, MOVF.PS, MOVT.PS), multiply (MUL.PS), multiply-add (MADD.PS), and multiply-subtract (MSUB.PS).

There is no specific provision in the MIPS architecture for floating-point execution to proceed in parallel with integer execution, but the MIPS implementations of floating point allow this to happen by checking to see if arithmetic interrupts are possible early in the cycle (see Appendix H). Normally, exception detection would force serialization of execution of integer and floating-point operations.

C.8 Instructions Unique to Alpha

The Alpha was intended to be an architecture that was easy to build high-performance implementations. Toward that goal, the architects originally made two controversial decisions: imprecise floating-point exceptions and no byte or half-word data transfers.

To simplify pipelined execution, Alpha does not require that an exception act as if no instructions past a certain point are executed and that all before that point have been executed. It supplies the TRAPB instruction, which stalls until all prior arithmetic instructions are guaranteed to complete without incurring arithmetic exceptions. In the most conservative mode, placing one TRAPB per exception-causing instruction slows execution by roughly five times but provides precise exceptions (see Darcy and Gay [1996]).

Code that does not include TRAPB does not the obey IEEE 754 floating-point standard. The reason is that parts of the standard (NaNs, infinities, and denormal) are implemented in software on Alpha, as it is on many other microprocessors. To implement these operations in software, however, programs must find the offending instruction and operand values, which cannot be done with imprecise interrupts!

When the architecture was developed, it was believed by the architects that byte loads and stores would slow down data transfers. Byte loads require an extra shifter in the data transfer path, and byte stores require that the memory system perform a read-modify-write for memory systems with error correction codes since the new ECC value must be recalculated. This omission meant that byte stores require the sequence load word, replace desired byte, and then store word.

(Inconsistently, floating-point loads go through considerable byte swapping to convert the obtuse VAX floating-point formats into a canonical form.)

To reduce the number of instructions to get the desired data, Alpha includes an elaborate set of byte manipulation instructions: extract field and zero rest of a register (`EXTxx`), insert field (`INSxx`), mask rest of a register (`MSKxx`), zero fields of a register (`ZAP`), and compare multiple bytes (`CMPGE`).

Apparently the implementors were not as bothered by load and store byte as were the original architects. Beginning with the shrink of the second version of the Alpha chip (21164A), the architecture *does* include loads and stores for bytes and half words.

Remaining Instructions

Below is a list of the remaining unique instructions of the Alpha architecture:

- *PAL code*—To provide the operations that the VAX performed in microcode, Alpha provides a mode that runs with all privileges enabled, interrupts disabled, and virtual memory mapping turned off for instructions. PAL (privileged architecture library) code is used for TLB management, atomic memory operations, and some operating system primitives. PAL code is called via the `CALL_PAL` instruction.
- *No divide*—Integer divide is not supported in hardware.
- *"Unaligned" load-store*—`LDQ_U` and `STQ_U` load and store 64-bit data using addresses that ignore the least-significant three bits. Extract instructions then select the desired unaligned word using the lower address bits. These instructions are similar to `LWL/R,SWL/R` in MIPS.
- *Floating-point single precision represented as double precision*—Single-precision data are kept as conventional 32-bit formats in memory but are converted to 64-bit double-precision format in registers.
- *Floating-point register F31 is fixed at zero*—To simplify comparisons to zero.
- *VAX floating-point formats*—To maintain compatibility with the VAX architecture, in addition to the IEEE 754 single- and double-precision formats called S and T, Alpha supports the VAX single- and double-precision formats called F and G, but not VAX format D. (D had too narrow an exponent field to be useful for double precision and was replaced by G in VAX code.)
- *Bit count instructions*—Version 3 of the architecture added instructions to count the number of leading zeros (`CTLZ`), count the number of trailing zeros (`CTTZ`), and count the number of ones in a word (`CTPOP`). Originally found on Cray computers, these instructions help with decryption.

C.9 Instructions Unique to SPARC v.9

Several features are unique to SPARC.

Register Windows

The primary unique feature of SPARC is register windows, an optimization for reducing register traffic on procedure calls. Several banks of registers are used, with a new one allocated on each procedure call. Although this could limit the depth of procedure calls, the limitation is avoided by operating the banks as a circular buffer, providing unlimited depth. The knee of the cost-performance curve seems to be six to eight banks.

SPARC can have between 2 and 32 windows, typically using eight registers each for the globals, locals, incoming parameters, and outgoing parameters. (Given that each window has 16 unique registers, an implementation of SPARC can have as few as 40 physical registers and as many as 520, although most have 128 to 136, so far.) Rather than tie window changes with call and return instructions, SPARC has the separate instructions `SAVE` and `RESTORE`. `SAVE` is used to "save" the caller's window by pointing to the next window of registers in addition to performing an add instruction. The trick is that the source registers are from the caller's window of the addition operation, while the destination register is in the callee's window. SPARC compilers typically use this instruction for changing the stack pointer to allocate local variables in a new stack frame. `RESTORE` is the inverse of `SAVE`, bringing back the caller's window while acting as an add instruction, with the source registers from the callee's window and the destination register in the caller's window. This automatically deallocates the stack frame. Compilers can also make use of it for generating the callee's final return value.

The danger of register windows is that the larger number of registers could slow down the clock rate. This was not the case for early implementations. The SPARC architecture (with register windows) and the MIPS R2000 architecture (without) have been built in several technologies since 1987. For several generations the SPARC clock rate has not been slower than the MIPS clock rate for implementations in similar technologies, probably because cache access times dominate register access times in these implementations. The current-generation machines took different implementation strategies—in order vs. out of order—and it's unlikely that the number of registers by themselves determined the clock rate in either machine. Recently, other architectures have included register windows: Tensilica and IA-64.

Another data transfer feature is alternate space option for loads and stores. This simply allows the memory system to identify memory accesses to input/output devices, or to control registers for devices such as the cache and memory management unit.

Fast Traps

Version 9 SPARC includes support to make traps fast. It expands the single level of traps to at least four levels, allowing the window overflow and underflow trap handlers to be interrupted. The extra levels mean the handler does not need to check for page faults or misaligned stack pointers explicitly in the code, thereby

making the handler faster. Two new instructions were added to return from this multilevel handler: `RETRY` (which retries the interrupted instruction) and `DONE` (which does not). To support user-level traps, the instruction `RETURN` will return from the trap in nonprivileged mode.

Support for LISP and Smalltalk

The primary remaining arithmetic feature is tagged addition and subtraction. The designers of SPARC spent some time thinking about languages like LISP and Smalltalk, and this influenced some of the features of SPARC already discussed: register windows, conditional trap instructions, calls with 32-bit instruction addresses, and multiword arithmetic (see Taylor et al. [1986] and Ungar et al. [1984]). A small amount of support is offered for tagged data types with operations for addition, subtraction, and hence comparison. The two least-significant bits indicate whether the operand is an integer (coded as 00), so `TADDcc` and `TSUBcc` set the overflow bit if either operand is not tagged as an integer or if the result is too large. A subsequent conditional branch or trap instruction can decide what to do. (If the operands are not integers, software recovers the operands, checks the types of the operands, and invokes the correct operation based on those types.) It turns out that the misaligned memory access trap can also be put to use for tagged data, since loading from a pointer with the wrong tag can be an invalid access. Figure C.32 shows both types of tag support.

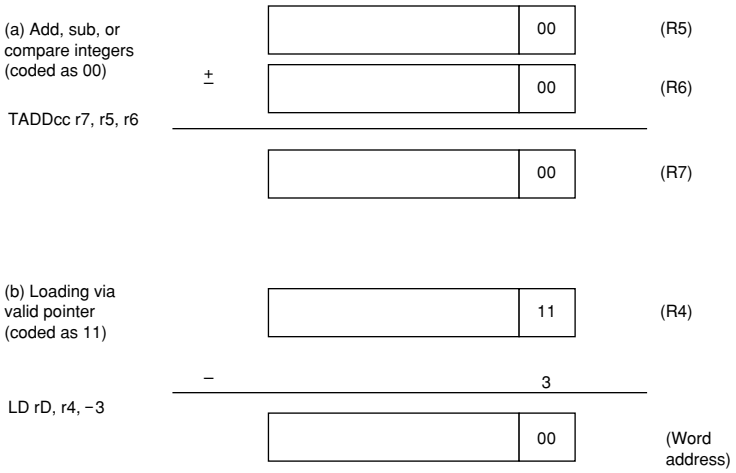


Figure C.32 SPARC uses the two least-significant bits to encode different data types for the tagged arithmetic instructions. (a) Integer arithmetic, which takes a single cycle as long as the operands and the result are integers. (b) The misaligned trap can be used to catch invalid memory accesses, such as trying to use an integer as a pointer. For languages with paired data like LISP, an offset of −3 can be used to access the even word of a pair (CAR) and +1 can be used for the odd word of a pair (CDR).

Overlapped Integer and Floating-Point Operations

SPARC allows floating-point instructions to overlap execution with integer instructions. To recover from an interrupt during such a situation, SPARC has a queue of pending floating-point instructions and their addresses. `RDPR` allows the processor to empty the queue. The second floating-point feature is the inclusion of floating-point square root instructions `FSQRTS`, `FSQRTD`, and `FSQRTQ`.

Remaining Instructions

The remaining unique features of SPARC are as follows:

- `JMPL` uses `Rd` to specify the return address register, so specifying `r31` makes it similar to `JALR` in MIPS and specifying `r0` makes it like `JR`.
- `LDSTUB` loads the value of the byte into `Rd` and then stores `FF16` into the addressed byte. This version 8 instruction can be used to implement a semaphore (see Chapter 6).
- *CASA* (*CASXA*) atomically compares a value in a processor register to a 32-bit (64-bit) value in memory; if and only if they are equal, it swaps the value in memory with the value in a second processor register. This version 9 instruction can be used to construct wait-free synchronization algorithms that do not require the use of locks.
- `XNOR` calculates the exclusive OR with the complement of the second operand.
- `BPcc`, `BPr`, and `FBPcc` include a branch-prediction bit so that the compiler can give hints to the machine about whether a branch is likely to be taken or not.
- `ILLTRAP` causes an illegal instruction trap. Muchnick [1988] explains how this is used for proper execution of aggregate returning procedures in C.
- `POPC` counts the number of bits set to one in an operand, also found in the third version of the Alpha architecture.
- *Nonfaulting loads* allow compilers to move load instructions ahead of conditional control structures that control their use. Hence, nonfaulting loads will be executed speculatively.
- *Quadruple-precision floating-point arithmetic and data transfer* allow the floating-point registers to act as eight 128-bit registers for floating-point operations and data transfers.
- *Multiple-precision floating-point results for multiply* mean that two single-precision operands can result in a double-precision product and two double-precision operands can result in a quadruple-precision product. These instructions can be useful in complex arithmetic and some models of floating-point calculations.

C.10 Instructions Unique to PowerPC

PowerPC is the result of several generations of IBM commercial RISC machines—IBM RT/PC, IBM Power1, and IBM Power2—plus the Motorola 88x00.

Branch Registers: Link and Counter

Rather than dedicate one of the 32 general-purpose registers to save the return address on procedure call, PowerPC puts the address into a special register called the *link register.* Since many procedures will return without calling another procedure, link doesn't always have to be saved away. Making the return address a special register makes the return jump faster since the hardware need not go through the register read pipeline stage for return jumps.

In a similar vein, PowerPC has a *count register* to be used in for loops where the program iterates for a fixed number of times. By using a special register the branch hardware can determine quickly whether a branch based on the count register is likely to branch, since the value of the register is known early in the execution cycle. Tests of the value of the count register in a branch instruction will automatically decrement the count register.

Given that the count register and link register are already located with the hardware that controls branches, and that one of the problems in branch prediction is getting the target address early in the pipeline (see Appendix A), the PowerPC architects decided to make a second use of these registers. Either register can hold a target address of a conditional branch. Thus PowerPC supplements its basic conditional branch with two instructions that get the target address from these registers (`BCLR`, `BCCTR`).

Remaining Instructions

Unlike most other RISC machines, register 0 is not hardwired to the value 0. It cannot be used as a base register—that is, it generates a 0 in this case—but in base + index addressing it can be used as the index. The other unique features of the PowerPC are as follows:

- *Load multiple* and *store multiple* save or restore up to 32 registers in a single instruction.
- `LSW` and `STSW` permit fetching and storing of fixed- and variable-length strings that have arbitrary alignment.
- *Rotate with mask* instructions support bit field extraction and insertion. One version rotates the data and then performs logical `AND` with a mask of ones, thereby extracting a field. The other version rotates the data but only places the bits into the destination register where there is a corresponding 1 bit in the mask, thereby inserting a field.

- *Algebraic right shift* sets the carry bit (`CA`) if the operand is negative and any 1 bits are shifted out. Thus a signed divide by any constant power of 2 that rounds toward 0 can be accomplished with a `SRAWI` followed by `ADDZE`, which adds `CA` to the register.
- *CBTLZ* will count leading zeros.
- *SUBFIC* computes (immediate – RA), which can be used to develop a one's or two's complement.
- *Logical shifted immediate* instructions shift the 16-bit immediate to the left 16 bits before performing AND, OR, or XOR.

C.11 Instructions Unique to PA-RISC 2.0

PA-RISC was expanded slightly in 1990 with version 1.1 and changed significantly in 2.0 with 64-bit extensions in 1996. PA-RISC perhaps has the most unusual features of any desktop RISC machine. For example, it has the most addressing modes, instruction formats, and, as we shall see, several instructions that are really the combination of two simpler instructions.

Nullification

As shown in Figure C.30, several RISC machines can choose to not execute the instruction following a delayed branch in order to improve utilization of the branch slot. This is called *nullification* in PA-RISC, and it has been generalized to apply to any arithmetic/logical instruction as well as to all branches. Thus an add instruction can add two operands, store the sum, and cause the following instruction to be skipped if the sum is zero. Like conditional move instructions, nullification allows PA-RISC to avoid branches in cases where there is just one instruction in the then part of an if statement.

A Cornucopia of Conditional Branches

Given nullification, PA-RISC did not need to have separate conditional branch instructions. The inventors could have recommended that nullifying instructions precede unconditional branches, thereby simplifying the instruction set. Instead, PA-RISC has the largest number of conditional branches of any RISC machine. Figure C.33 shows the conditional branches of PA-RISC. As you can see, several are really combinations of two instructions.

Synthesized Multiply and Divide

PA-RISC provides several primitives so that multiply and divide can be synthesized in software. Instructions that shift one operand 1, 2, or 3 bits and then add,

Name	Instruction	Notation	
COMB	Compare and branch	`if (cond(Rs1,Rs2))`	`{PC <--- PC + offset12}`
COMIB	Compare imm. and branch	`if (cond(imm5,Rs2))`	`{PC <--- PC + offset12}`
MOVB	Move and branch	`Rs2 <--- Rs1, if (cond(Rs1,0))`	`{PC <--- PC + offset12}`
MOVIB	Move immediate and branch	`Rs2 <--- imm5, if (cond(imm5,0))`	`{PC <--- PC + offset12}`
ADDB	Add and branch	`Rs2 <--- Rs1 + Rs2, if (cond(Rs1 + Rs2,0))`	`{PC <--- PC + offset12}`
ADDIB	Add imm. and branch	`Rs2 <--- imm5 + Rs2, if (cond(imm5 + Rs2,0))`	`{PC <--- PC + offset12}`
BB	Branch on bit	`if (cond(Rsp,0)`	`{PC <--- PC + offset12}`
BVB	Branch on variable bit	`if (cond(Rssar,0)`	`{PC <--- PC + offset12}`

Figure C.33 The PA-RISC conditional branch instructions. The 12-bit offset is called `offset12` in this table, and the 5-bit immediate is called `imm5`. The 16 conditions are =, <, <=, odd, signed overflow, unsigned no overflow, zero or no overflow unsigned, never, and their respective complements. The `BB` instruction selects one of the 32 bits of the register and branches depending if its value is 0 or 1. The `BVB` selects the bit to branch using the shift amount register, a special-purpose register. The subscript notation specifies a bit field.

trapping or not on overflow, are useful in multiplies. (Alpha also includes instructions that multiply the second operand of adds and subtracts by 4 or by 8: `S4ADD`, `S8ADD`, `S4SUB`, and `S8SUB`.) Divide step performs the critical step of nonrestoring divide, adding or subtracting depending on the sign of the prior result. Magenheimer et al. [1988] measured the size of operands in multiplies and divides to show how well the multiply step would work. Using these data for C programs, Muchnick [1988] found that by making special cases the average multiply by a constant takes 6 clock cycles and multiply of variables takes 24 clock cycles. PA-RISC has 10 instructions for these operations.

The original SPARC architecture used similar optimizations, but with increasing numbers of transistors the instruction set was expanded to include full multiply and divide operations. PA-RISC gives some support along these lines by putting a full 32-bit integer multiply in the floating-point unit; however, the integer data must first be moved to floating-point registers.

Decimal Operations

COBOL programs will compute on decimal values, stored as 4 bits per digit, rather than converting back and forth between binary and decimal. PA-RISC has instructions that will convert the sum from a normal 32-bit add into proper decimal digits. It also provides logical and arithmetic operations that set the condition codes to test for carries of digits, bytes, or half words. These operations also test whether bytes or half words are zero. These operations would be useful in arithmetic on 8-bit ASCII characters. Five PA-RISC instructions provide decimal support.

Remaining Instructions

Here are some remaining PA-RISC instructions:

- *Branch vectored* shifts an index register left 3 bits, adds it to a base register, and then branches to the calculated address. It is used for case statements.
- *Extract* and *deposit* instructions allow arbitrary bit fields to be selected from or inserted into registers. Variations include whether the extracted field is sign-extended, whether the bit field is specified directly in the instruction or indirectly in another register, and whether the rest of the register is set to zero or left unchanged. PA-RISC has 12 such instructions.
- To simplify use of 32-bit address constants, PA-RISC includes `ADDIL`, which adds a left-adjusted 21-bit constant to a register and places the result in register 1. The following data transfer instruction uses offset addressing to add the lower 11 bits of the address to register 1. This pair of instructions allows PA-RISC to add a 32-bit constant to a base register, at the cost of changing register 1.
- PA-RISC has nine debug instructions that can set breakpoints on instruction or data addresses and return the trapped addresses.
- *Load* and *clear* instructions provide a semaphore or lock that reads a value from memory and then writes zero.
- *Store bytes short* optimizes unaligned data moves, moving either the leftmost or the rightmost bytes in a word to the effective address, depending on the instruction options and condition code bits.
- Loads and stores work well with caches by having options that give hints about whether to load data into the cache if it's not already in the cache. For example, load with a destination of register 0 is defined to be software-controlled cache prefetch.
- PA-RISC 2.0 extended cache hints to stores to indicate block copies, recommending that the processor not load data into the cache if it's not already in the cache. It also can suggest that on loads and stores, there is spatial locality to prepare the cache for subsequent sequential accesses.
- PA-RISC 2.0 also provides an optional branch-target stack to predict indirect jumps used on subroutine returns. Software can suggest which addresses get placed on and removed from the branch-target stack, but hardware controls whether or not these are valid.
- *Multiply*/*add* and *multiply*/*subtract* are floating-point operations that can launch two independent floating-point operations in a single instruction in addition to the fused multiply/add and fused multiply/negate/add introduced in version 2.0 of PA-RISC.

C.12 Instructions Unique to ARM

It's hard to pick the most unusual feature of ARM, but perhaps it is conditional execution of instructions. Every instruction starts with a 4-bit field that determines whether it will act as a nop or as a real instruction, depending on the condition codes. Hence conditional branches are properly considered as conditionally executing the unconditional branch instruction. Conditional execution allows avoiding a branch to jump over a single instruction. It takes less code space and time to simply conditionally execute one instruction.

The 12-bit immediate field has a novel interpretation. The 8 least-significant bits are zero-extended to a 32-bit value, then rotated right the number of bits specified in the first 4 bits of the field multiplied by 2. Whether this split actually catches more immediates than a simple 12-bit field would be an interesting study. One advantage is that this scheme can represent all powers of 2 in a 32-bit word.

Operand shifting is not limited to immediates. The second register of all arithmetic and logical processing operations has the option of being shifted before being operated on. The shift options are shift left logical, shift right logical, shift right arithmetic, and rotate right. Once again, it would be interesting to see how often operations like rotate-and-add, shift-right-and-test, and so on occur in ARM programs.

Remaining Instructions

Below is a list of the remaining unique instructions of the ARM architecture:

- *Block loads and stores*—Under control of a 16-bit mask within the instructions, any of the 16 registers can be loaded or stored into memory in a single instruction. These instructions can save and restore registers on procedure entry and return. These instructions can also be used for block memory copy—offering up to four times the bandwidth of a single register load-store—and today block copies are the most important use.
- *Reverse subtract*—RSB allows the first register to be subtracted from the immediate or shifted register. RSC does the same thing, but includes the carry when calculating the difference.
- *Long multiplies*—Similar to MIPS, Hi and Lo registers get the 64-bit signed product (SMULL) or the 64-bit unsigned product (UMULL).
- *No divide*—Like the Alpha, integer divide is not supported in hardware.
- *Conditional trap*—A common extension to the MIPS core found in desktop RISCs (Figures C.22 through C.25), it comes for free in the conditional execution of all ARM instructions, including SWI.
- *Coprocessor interface*—Like many of the desktop RISCs, ARM defines a full set of coprocessor instructions: data transfer, moves between general-purpose and coprocessor registers, and coprocessor operations.

- *Floating-point architecture*—Using the coprocessor interface, a floating-point architecture has been defined for ARM. It was implemented as the FPA10 coprocessor.
- *Branch and exchange instruction sets*—The BX instruction is the transition between ARM and Thumb, using the lower 31 bits of the register to set the PC and the most-significant bit to determine if the mode is ARM (1) or Thumb (0).

C.13 Instructions Unique to Thumb

In the ARM version 4 model, frequently executed procedures will use ARM instructions to get maximum performance, with the less frequently executed ones using Thumb to reduce the overall code size of the program. Since typically only a few procedures dominate execution time, the hope is that this hybrid gets the best of both worlds.

Although Thumb instructions are translated by the hardware into conventional ARM instructions for execution, there are several restrictions. First, conditional execution is dropped from almost all instructions. Second, only the first 8 registers are easily available in all instructions, with the stack pointer, link register, and program counter used implicitly in some instructions. Third, Thumb uses a two-operand format to save space. Fourth, the unique shifted immediates and shifted second operands have disappeared and are replaced by separate shift instructions. Fifth, the addressing modes are simplified. Finally, putting all instructions into 16 bits forces many more instruction formats.

In many ways the simplified Thumb architecture is more conventional than ARM. Here are additional changes made from ARM in going to Thumb:

- *Drop of immediate logical instructions*—Logical immediates are gone.
- *Condition codes implicit*—Rather than have condition codes set optionally, they are defined by the opcode. All ALU instructions and none of the data transfers set the condition codes.
- *Hi/Lo register access*—The 16 ARM registers are halved into Lo registers and Hi registers, with the 8 Hi registers including the stack pointer (SP), link register, and PC. The Lo registers are available in all ALU operations. Variations of `ADD`, `BX`, `CMP`, and `MOV` also work with all combinations of Lo and Hi registers. SP and PC registers are also available in variations of data transfers and add immediates. Any other operations on the Hi registers require one `MOV` to put the value into a Lo register, perform the operation there, and then transfer the data back to the Hi register.
- *Branch/call distance*—Since instructions are 16 bits wide, the 8-bit conditional branch address is shifted by 1 instead of by 2. Branch with link is specified in two instructions, concatenating 11 bits from each instruction and shifting them left to form a 23-bit address to load into PC.

- *Distance for data transfer offsets*—The offset is now 5 bits for the general-purpose registers and 8 bits for SP and PC.

C.14 Instructions Unique to SuperH

Register 0 plays a special role in SuperH address modes. It can be added to another register to form an address in indirect indexed addressing and PC-relative addressing. R0 is used to load constants to give a larger addressing range than can easily be fit into the 16-bit instructions of the SuperH. R0 is also the only register that can be an operand for immediate versions of AND, CMP, OR, and XOR.

Below is a list of the remaining unique details of the SuperH architecture:

- *Decrement and test*—`DT` decrements a register and sets the T bit to 1 if the result is 0.
- *Optional delayed branch*—Although the other embedded RISC machines generally do not use delayed branches (see Appendix A), SuperH offers optional delayed branch execution for `BT` and `BF`.
- *Many multiplies*—Depending if the operation is signed or unsigned, if the operands are 16 bits or 32 bits, or if the product is 32 bits or 64 bits, the proper multiply instruction is `MULS`, `MULU`, `DMULS`, `DMULU`, or `MUL`. The product is found in the MACL and MACH registers.
- *Zero and sign extension*—Byte or halfwords are either zero-extended (`EXTU`) or sign-extended (`EXTS`) within a 32-bit register.
- *One-bit shift amounts*—Perhaps in an attempt to make them fit within the 16-bit instructions, shift instructions only shift a single bit at a time.
- *Dynamic shift amount*—These variable shifts test the sign of the amount in a register to determine whether they shift left (positive) or shift right (negative). Both logical (`SHLD`) and arithmetic (`SHAD`) instructions are supported. These instructions help offset the 1-bit constant shift amounts of standard shifts.
- *Rotate*—SuperH offers rotations by 1 bit left (`ROTL`) and right (`ROTR`), which set the T bit with the value rotated, and also have variations that include the T bit in the rotations (`ROTCL` and `ROTCR`).
- *SWAP*—This instruction either swaps the high and low bytes of a 32-bit word or the two bytes of the rightmost 16 bits.
- *Extract word (`XTRCT`)*—The middle 32 bits from a pair of 32-bit registers are placed in another register.
- *Negate with carry*—Like `SUBC` (Figure C.27), except the first operand is 0.
- *Cache prefetch*—Like many of the desktop RISCs (Figures C.22 through C.25), SuperH has an instruction (`PREF`) to prefetch data into the cache.
- *Test-and-set*—SuperH uses the older test-and-set (`TAS`) instruction to perform atomic locks or semaphores (see Chapter 9). `TAS` first loads a byte from

memory. It then sets the T bit to 1 if the byte is 0 or to 0 if the byte is not 0. Finally, it sets the most-significant bit of the byte to 1 and writes the result back to memory.

C.15 Instructions Unique to M32R

The most unusual feature of the M32R is a slight VLIW approach to the pairs of 16-bit instructions. A bit is reserved in the first instruction of the pair to say whether this instruction can be executed in parallel with the next instruction—that is, the two instructions are independent—or if these two must be executed sequentially. (An earlier machine that offered a similar option was the Intel i860.) This feature is included for future implementations of the architecture.

One surprise is that all branch displacements are shifted left 2 bits before being added to the PC and the lower 2 bits of the PC are set to 0. Since some instructions are only 16 bits long, this shift means that a branch cannot go to any instruction in the program: It can only branch to instructions on word boundaries. A similar restriction is placed on the return address for the branch-and-link and jump-and-link instructions: they can only return to a word boundary. Thus for a slightly larger branch distance, software must ensure that all branch addresses and all return addresses are aligned to a word boundary. The M32R code space is probably slightly larger, and it probably executes more NOP instructions than it would if the branch address were only shifted left 1 bit.

However, the VLIW feature above means that a NOP can execute in parallel with another 16-bit instruction, so that the padding doesn't take more clock cycles. The code size expansion depends on the ability of the compiler to schedule code and to pair successive 16-bit instructions; Mitsubishi claims that code size overall is only 7% larger than that for the Motorola 680x0 architecture.

The last remaining novel feature is that the result of the divide operation is the remainder instead of the quotient.

C.16 Instructions Unique to MIPS16

MIPS16 is not really a separate instruction set but a 16-bit extension of the full 32-bit MIPS architecture. It is compatible with any of the 32-bit address MIPS architectures (MIPS I, MIPS II) or 64-bit architectures (MIPS III, IV, V). The ISA mode bit determines the width of instructions: 0 means 32-bit-wide instructions and 1 means 16-bit-wide instructions. The new `JALX` instruction toggles the ISA mode bit to switch to the other ISA. `JR` and `JALR` have been redefined to set the ISA mode bit from the most-significant bit of the register containing the branch address, and this bit is not considered part of the address. All jump and link instructions save the current mode bit as the most-significant bit of the return address.

Hence MIPS supports whole procedures containing either 16-bit or 32-bit instructions, but it does not support mixing the two lengths together in a single procedure. The one exception is the JAL and JALX: These two instructions need 32 bits even in the 16-bit mode, presumably to get a large enough address to branch to far procedures.

In picking this subset, MIPS decided to include opcodes for some three-operand instructions and to keep 16 opcodes for 64-bit operations. The combination of this many opcodes and operands in 16 bits led the architects to provide only 8 easy-to-use registers—just like Thumb—whereas the other embedded RISCs offer about 16 registers. Since the hardware must include the full 32 registers of the 32-bit ISA mode, MIPS16 includes move instructions to copy values between the 8 MIPS16 registers and the remaining 24 registers of the full MIPS architecture. To reduce pressure on the 8 visible registers, the stack pointer is considered a separate register. MIPS16 includes a variety of separate opcodes to do data transfers using sp as a base register and to increment sp: LWSP, LDSP, SWSP, SDSP, ADJSP, DADJSP, ADDIUSPD, and DADDIUSP.

To fit within the 16-bit limit, immediate fields have generally been shortened to 5 to 8 bits. MIPS16 provides a way to extend its shorter immediates into the full width of immediates in the 32-bit mode. Borrowing a trick from the Intel 8086, the EXTEND instruction is really a 16-bit prefix that can be prepended to any MIPS16 instruction with an address or immediate field. The prefix supplies enough bits to turn the 5-bit fields of data transfers and 5- to 8-bit fields of arithmetic immediates into 16-bit constants. Alas, there are two exceptions. ADDIU and DADDIU start with 4-bit immediate fields, but since EXTEND can only supply 11 more bits, the wider immediate is limited to 15 bits. EXTEND also extends the 3-bit shift fields into 5-bit fields for shifts. (In case you were wondering, the EXTEND prefix does *not* need to start on a 32-bit boundary.)

To further address the supply of constants, MIPS16 added a new addressing mode! PC-relative addressing for load word (LWPC) and load double (LDPC) shifts an 8-bit immediate field by 2 or 3 bits, respectively, adding it to the PC with the lower 2 or 3 bits cleared. The constant word or double word is then loaded into a register. Thus 32-bit or 64-bit constants can be included with MIPS16 code, despite the loss of LIU to set the upper register bits. Given the new addressing mode, there is also an instruction (ADDIUPC) to calculate a PC-relative address and place it in a register.

MIPS16 differs from the other embedded RISCs in that it can subset a 64-bit address architecture. As a result it has 16-bit instruction-length versions of 64-bit data operations: data transfer (LD, SD, LWU), arithmetic operations (DADDU/IU, DSUBU, DMULT/U, DDIV/U), and shifts (DSLL/V, DSRA/V, DSRL/V).

Since MIPS plays such a prominent role in this book, we show all the additional changes made from the MIPS core instructions in going to MIPS16:

- *Drop of signed arithmetic instructions*—Arithmetic instructions that can trap were dropped to save opcode space: ADD, ADDI, SUB, DADD, DADDI, DSUB.
- *Drop of immediate logical instructions*—Logical immediates are gone too: ANDI, ORI, XORI.

- *Branch instructions pared down*—Comparing two registers and then branching did not fit, nor did all the other comparisons of a register to zero. Hence these instructions didn't make it either: `BEQ`, `BNE`, `BGEZ`, `BGTZ`, `BLEZ`, and `BLTZ`. As mentioned in Section C.3, to help compensate MIPS16 includes compare instructions to test if two registers are equal. Since compare and set-on-less-than set the new T register, branches were added to test the T register.
- *Branch distance*—Since instructions are 16 bits wide, the branch address is shifted by one instead of by two.
- *Delayed branches disappear*—The branches take effect before the next instruction. Jumps still have a one-slot delay.
- *Extension and distance for data transfer offsets*—The 5-bit and 8-bit fields are zero-extended instead of sign-extended in 32-bit mode. To get greater range, the immediate fields are shifted left 1, 2, or 3 bits depending on whether the data is half word, word, or double word. If the `EXTEND` prefix is prepended to these instructions, they use the conventional signed 16-bit immediate of the 32-bit mode.
- *Extension of arithmetic immediates*—The 5-bit and 8-bit fields are zero-extended for set-on-less-than and compare instructions, for forming a PC-relative address, and for adding to SP and placing the result in a register (`ADDIUSP`, `DADDIUSP`). Once again, if the `EXTEND` prefix is prepended to these instructions, they use the conventional signed 16-bit immediate of the 32-bit mode. They are still sign-extended for general adds and for adding to SP and placing the result back in SP (`ADJSP`, `DADJSP`). Alas, code density and orthogonality are strange bedfellows in MIPS16!
- *Redefining shift amount of 0*—MIPS16 defines the value 0 in the 3-bit shift field to mean a shift of 8 bits.
- *New instructions added due to loss of register 0 as zero*—Load immediate, negate, and not were added, since these operations could no longer be synthesized from other instructions using r0 as a source.

C.17 Concluding Remarks

This appendix covers the addressing modes, instruction formats, and all instructions found in 10 recent RISC architectures. Although the later sections concentrate on the differences, it would not be possible to cover 10 architectures in these few pages if there were not so many similarities. In fact, we would guess that more than 90% of the instructions executed for any of these architectures would be found in Figures C.9 through C.17. To contrast this homogeneity, Figure C.34 gives a summary for four architectures from the 1970s in a format similar to that shown in Figure C.1. (Imagine trying to write a single chapter in this style for those architectures!) In the history of computing, there has never been such widespread agreement on computer architecture.

	IBM 360/370	Intel 8086	Motorola 68000	DEC VAX
Date announced	1964/1970	1978	1980	1977
Instruction size(s) (bits)	16, 32, 48	8, 16, 24, 32, 40, 48	16, 32, 48, 64, 80	8, 16, 24, 32,..., 432
Addressing (size, model)	24 bits, flat/ 31 bits, flat	4 + 16 bits, segmented	24 bits, flat	32 bits, flat
Data aligned?	Yes 360/ No 370	No	16-bit aligned	No
Data addressing modes	2/3	5	9	= 14
Protection	Page	None	Optional	Page
Page size	2 KB & 4 KB	—	0.25 to 32 KB	0.5 KB
I/O	Opcode	Opcode	Memory mapped	Memory mapped
Integer registers (size, model, number)	16 GPR × 32 bits	8 dedicated data × 16 bits	8 data and 8 address × 32 bits	15 GPR × 32 bits
Separate floating-point registers	4 × 64 bits	Optional: 8 × 80 bits	Optional: 8 × 80 bits	0
Floating-point format	IBM (floating hexadecimal)	IEEE 754 single, double, extended	IEEE 754 single, double, extended	DEC

Figure C.34 Summary of four 1970s architectures. Unlike the architectures in Figure C.1, there is little agreement between these architectures in any category. (See Appendix D for more details on the 80x86 and Appendix E for a description of the VAX.)

This style of architecture cannot remain static, however. Like people, instruction sets tend to get bigger as they get older. Figure C.35 shows the genealogy of these instruction sets, and Figure C.36 shows which features were added to or deleted from generations of desktop RISCs over time.

As you can see, all the desktop RISC machines have evolved to 64-bit address architectures, and they have done so fairly painlessly. The only remaining major desktop 32-bit address architecture is the Intel 80x86.

Its 64-bit address successor is IA-64. If IA-64 proves successful, then microprocessor architectures of the 1970s will finally step into history rather than shape the cost and performance of modern desktop computing.

C.18 Acknowledgments

We would like to thank the following people for comments on drafts of this appendix: Professor Steven B. Furber, University of Manchester; Dr. Dileep Bhandarkar, Intel Corporation; Dr. Earl Killian, Silicon Graphics/MIPS; and Dr. Hiokazu Takata, Mitsubishi Electric Corporation.

References

Bhandarkar, D. P. [1995]. *Alpha Architecture and Implementations,* Digital Press, Newton, Mass.

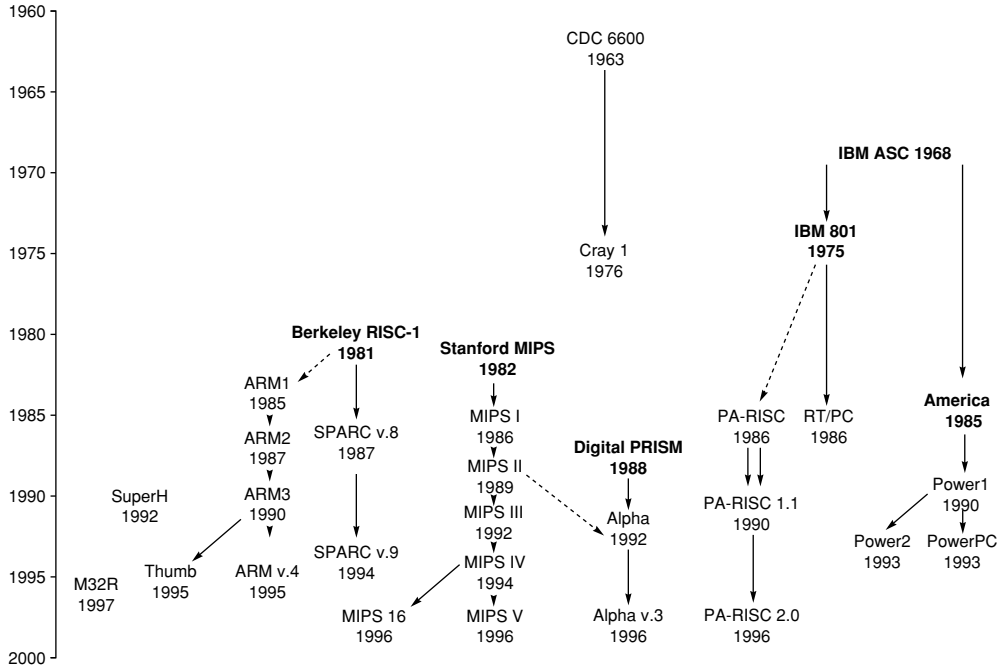


Figure C.35 The lineage of RISC instruction sets. Commercial machines are shown in plain text and research machines in bold. The CDC-6600 and Cray-1 were load-store machines with register 0 fixed at 0, and separate integer and floating-point registers. Instructions could not cross word boundaries. An early IBM research machine led to the 801 and America research projects, with the 801 leading to the unsuccessful RT/PC and America leading to the successful Power architecture. Some people who worked on the 801 later joined Hewlett-Packard to work on the PA-RISC. The two university projects were the basis of MIPS and SPARC machines. According to Furber [1996], the Berkeley RISC project was the inspiration of the ARM architecture. While ARM1, ARM2, and ARM3 were names of both architectures and chips, ARM version 4 is the name of the architecture used in ARM7, ARM8, and StrongARM chips. (There are no ARM v.4 and ARM5 chips, but ARM6 and early ARM7 chips use the ARM3 architecture.) DEC built a RISC microprocessor in 1988 but did not introduce it. Instead, DEC shipped workstations using MIPS microprocessors for three years before they brought out their own RISC instruction set, Alpha 21064, which is very similar to MIPS III and PRISM. The Alpha architecture has had small extensions, but they have not been formalized with version numbers; we used version 3 because that is the version of the reference manual. The Alpha 21164A chip added byte and halfword loads and stores, and the Alpha 21264 includes the MAX multimedia and bit count instructions. Internally, Digital names chips after the fabrication technology: EV4 (21064), EV45 (21064A), EV5 (21164), EV56 (21164A), and EV6 (21264). "EV" stands for "extended VAX."

Darcy, J. D., and D. Gay [1996]. "FLECKmarks: Measuring floating point performance using a full IEEE compliant arithmetic benchmark," CS 252 class project, U.C. Berkeley (see *HTTP.CS.Berkeley.EDU/~darcy/Projects/cs252/*).

Digital Semiconductor [1996]. *Alpha Architecture Handbook, Version 3*, Digital Press, Maynard, Mass., Order number EC-QD2KB-TE (October).

Furber, S. B. [1996]. *ARM System Architecture*, Addison-Wesley, Harlow, England (see *www.cs.man.ac.uk/amulet/publications/books/ARMsysArch*).

	PA-RISC			SPARC		MIPS					Power		
Feature	**1.0**	**1.1**	**2.0**	**v. 8**	**v. 9**	**I**	**II**	**III**	**IV**	**V**	**1**	**2**	**PC**
Interlocked loads	X	"	"	X	"		+	"	"		X	"	"
Load-store FP double	X	"	"	X	"		+	"	"		X	"	"
Semaphore	X	"	"	X	"		+	"	"		X	"	"
Square root	X	"	"	X	"		+	"	"			+	"
Single-precision FP ops	X	"	"	X	"	X	"	"	"				+
Memory synchronize	X	"	"	X	"		+	"	"		X	"	"
Coprocessor	X	"	"	X	—	X	"	"	"				
Base + index addressing	X	"	"	X	"				+		X	"	"
Equiv. 32 64-bit FP registers		"	"		+			+	"		X	"	"
Annulling delayed branch	X	"	"	X	"		+	"	"				
Branch register contents	X	"	"		+	X	"	"	"				
Big/Little Endian		+	"		+	X	"	"	"				+
Branch-prediction bit					+		+	"	"		X	"	"
Conditional move					+				+		X	"	—
Prefetch data into cache			+		+				+		X	"	"
64-bit addressing/ int. ops			+		+			+	"				+
32-bit multiply, divide		+	"		+	X	"	"	"		X	"	"
Load-store FP quad					+							+	—
Fused FP mul/add			+						+		X	"	"
String instructions	X	"	"								X	"	—
Multimedia support		X	"	X						X			

Figure C.36 Features added to desktop RISC machines. X means in the original machine, + means added later, " means continued from prior machine, and — means removed from architecture. Alpha is not included, but it added byte and word loads and stores, and bit count and multimedia extensions, in version 3. MIPS V added the MDMX instructions and paired single floating-point operations.

Hewlett-Packard [1994]. *PA-RISC 2.0 Architecture Reference Manual,* 3rd ed.

Hitachi [1997]. *SuperH RISC Engine SH7700 Series Programming Manual* (see *www.halsp.hitachi.com/tech_prod/* and search for title).

IBM [1994]. *The PowerPC Architecture,* Morgan Kaufmann, San Francisco.

Kane, G. [1996]. *PA-RISC 2.0 Architecture*, Prentice Hall PTR, Upper Saddle River, N.J.

Kane, G., and J. Heinrich [1992]. *MIPS RISC Architecture*, Prentice Hall, Englewood Cliffs, N.J.

Kissell, K. D. [1997]. *MIPS16: High-Density for the Embedded Market* (see *www.sgi.com/MIPS/arch/MIPS16/MIPS16.whitepaper.pdf*).

Magenheimer, D. J., L. Peters, K. W. Pettis, and D. Zuras [1988]. "Integer multiplication and division on the HP precision architecture," *IEEE Trans. on Computers* 37:8, 980–990.

MIPS [1997]. *MIPS16 Application Specific Extension Product Description*, (see *www.sgi.com/MIPS/arch/MIPS16/mips16.pdf*)

Mitsubishi [1996]. *Mitsubishi 32-Bit Single Chip Microcomputer M32R Family Software Manual* (September).

Muchnick, S. S. [1988]. "Optimizing compilers for SPARC," *Sun Technology* 1:3 (Summer), 64–77.

Silicon Graphics [1996]. *MIPS V Instruction Set* (see *http://www.sgi.com/MIPS/arch/ISA5/#MIPSV_indx*).

Sites, R. L., and R. Witek (eds.) [1995]. *Alpha Architecture Reference Manual, Second Edition,* Digital Press, Newton, Mass.

Sun Microsystems [1989]. *The SPARC Architectural Manual,* Version 8, Part No. 800-1399-09, August 25.

Taylor, G., P. Hilfinger, J. Larus, D. Patterson, and B. Zorn [1986]. "Evaluation of the SPUR LISP architecture," *Proc. 13th Symposium on Computer Architecture* (June), Tokyo.

Ungar, D., R. Blau, P. Foley, D. Samples, and D. Patterson [1984]. "Architecture of SOAR: Smalltalk on a RISC," *Proc. 11th Symposium on Computer Architecture* (June), Ann Arbor, Mich., 188–197.

Weaver, D. L., and T. Germond [1994]. *The SPARC Architectural Manual,* Version 9, Prentice Hall, Englewood Cliffs, N.J.

Weiss, S., and J. E. Smith [1994]. *Power and PowerPC,* Morgan Kaufmann, San Francisco.

D.1	Introduction	D-2
D.2	80x86 Registers and Data Addressing Modes	D-3
D.3	80x86 Integer Operations	D-6
D.4	80x86 Floating-Point Operations	D-10
D.5	80x86 Instruction Encoding	D-12
D.6	Putting It All Together: Measurements of Instruction Set Usage	D-14
D.7	Concluding Remarks	D-20
D.8	Historical Perspective and References	D-21

D

An Alternative to RISC: The Intel 80x86

The x86 isn't all that complex—it just doesn't make a lot of sense.

Mike Johnson
Leader of 80x86 Design at AMD,
Microprocessor Report (1994)

© 2003 Elsevier Science (USA). All rights reserved.

D.1 Introduction

MIPS was the vision of a single architect. The pieces of this architecture fit nicely together and the whole architecture can be described succinctly. Such is not the case of the 80x86: It is the product of several independent groups who evolved the architecture over 20 years, adding new features to the original instruction set as you might add clothing to a packed bag. Here are important 80x86 milestones:

- 1978—The Intel 8086 architecture was announced as an assembly language–compatible extension of the then-successful Intel 8080, an 8-bit microprocessor. The 8086 is a 16-bit architecture, with all internal registers 16 bits wide. Whereas the 8080 was a straightforward accumulator machine, the 8086 extended the architecture with additional registers. Because nearly every register has a dedicated use, the 8086 falls somewhere between an accumulator machine and a general-purpose register machine, and can fairly be called an *extended accumulator* machine.
- 1980—The Intel 8087 floating-point coprocessor is announced. This architecture extends the 8086 with about 60 floating-point instructions. Its architects rejected extended accumulators to go with a hybrid of stacks and registers, essentially an *extended stack* architecture: A complete stack instruction set is supplemented by a limited set of register-memory instructions.
- 1982—The 80286 extended the 8086 architecture by increasing the address space to 24 bits, by creating an elaborate memory mapping and protection model, and by adding a few instructions to round out the instruction set and to manipulate the protection model. Because it was important to run 8086 programs without change, the 80286 offered a *real addressing mode* to make the machine look just like an 8086.
- 1985—The 80386 extended the 80286 architecture to 32 bits. In addition to a 32-bit architecture with 32-bit registers and a 32-bit address space, the 80386 added new addressing modes and additional operations. The added instructions make the 80386 nearly a general-purpose register machine. The 80386 also added paging support in addition to segmented addressing (see Chapter 5). Like the 80286, the 80386 has a mode to execute 8086 programs without change.

This history illustrates the impact of the "golden handcuffs" of compatibility on the 80x86, as the existing software base at each step was too important to jeopardize with significant architectural changes. Fortunately, the subsequent 80486 in 1989, Pentium in 1992, and P6 in 1995 were aimed at higher performance, with only four instructions added to the user-visible instruction set: three to help with multiprocessing plus a conditional move instruction.

Since 1997 Intel has added hundreds of instructions to support multimedia by operating on many narrower data types within a single clock (see Chapter 2). These SIMD or vector instructions are primarily used in handcoded libraries or drivers and rarely generated by compilers. The first extension, called MMX,

appeared in 1997. It consists of 57 instructions that pack and unpack multiple bytes, 16-bit words, or 32-bit double words into 64-bit registers and performs shift, logical, and integer arithmetic on the narrow data items in parallel. It supports both saturating and nonsaturating arithmetic. MMX uses the registers comprising the floating-point stack and hence there is no new state for operating systems to save.

In 1999 Intel added another 70 instructions, labeled SSE as part of Pentium III. The primary changes were to add eight separate registers, double their width to 128 bits, and add a single-precision floating-point data type. Hence four 32-bit floating-point operations can be performed in parallel. To improve memory performance, SSE included cache prefetch instructions plus streaming store instructions that bypass the caches and write directly to memory.

In 2001 Intel added yet another 144 instructions, this time labeled SSE2. The new data type is double-precision arithmetic, which allows pairs of 64-bit floating-point operations in parallel. Almost all of these 144 instructions are versions of existing MMX and SSE instructions that operate on 64 bits of data in parallel. Not only does this change enable multimedia operations, it gives the compiler a different target for floating-point operations than the unique stack architecture. Compilers can choose to use the eight SSE registers as floating-point registers as found in the RISC machines. This change has boosted performance on the Pentium 4, the first microprocessor to include SSE2 instructions. At the time of announcement, a 1.5 GHz Pentium 4 was 1.24 times faster than a 1 GHz Pentium III for SPECint2000(base), but it was 1.88 times faster for SPECfp2000(base).

Whatever the artistic failures of the 80x86, keep in mind that there are more instances of this architectural family than of any other server or desktop processor in the world, perhaps 500 million in 2001. Nevertheless, its checkered ancestry has led to an architecture that is difficult to explain and impossible to love.

We start our explanation with the registers and addressing modes, move on to the integer operations, then cover the floating-point operations, and conclude with an examination of instruction encoding.

D.2 80x86 Registers and Data Addressing Modes

The evolution of the instruction set can be seen in the registers of the 80x86 (Figure D.1). Original registers are shown in black type, with the extensions of the 80386 shown in a lighter shade, a coloring scheme followed in subsequent figures. The 80386 basically extended all 16-bit registers (except the segment registers) to 32 bits, prefixing an "E" to their name to indicate the 32-bit version. The arithmetic, logical, and data transfer instructions are two-operand instructions that allow the combinations shown in Figure D.2.

To explain the addressing modes we need to keep in mind whether we are talking about the 16-bit mode used by both the 8086 and 80286 or the 32-bit mode available on the 80386 and its successors. The seven data memory addressing modes supported are

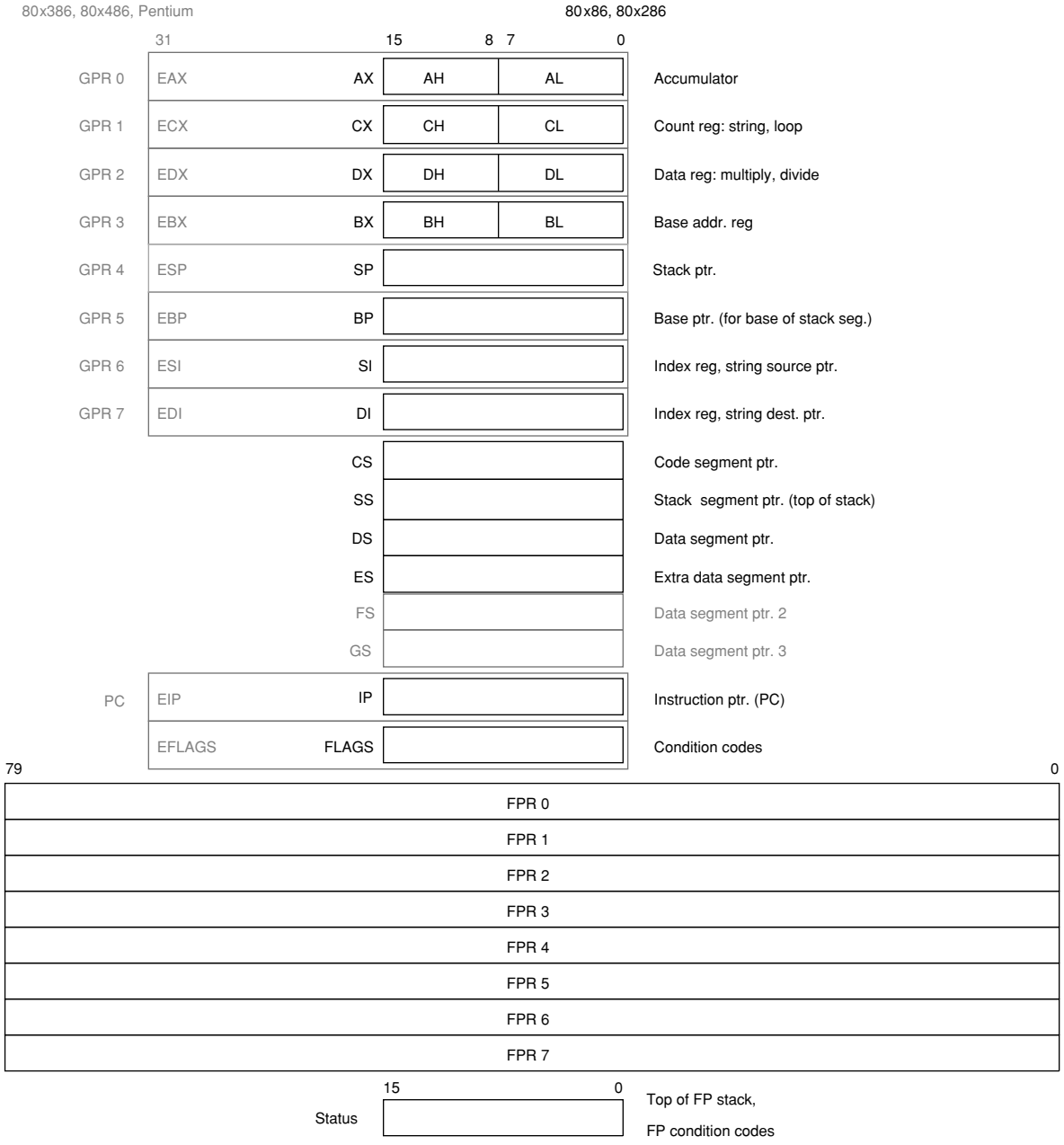


Figure D.1 The 80x86 has evolved over time, and so has its register set. The original set is shown in black, and the extended set in gray. The 8086 divided the first four registers in half so that they could be used either as one 16-bit register or as two 8-bit registers. Starting with the 80386, the top eight registers were extended to 32 bits and could also be used as general-purpose registers. The floating-point registers on the bottom are 80 bits wide, and although they look like regular registers they are not. They implement a stack, with the top of stack pointed to by the status register. One operand must be the top of stack, and the other can be any of the other seven registers below the top of stack.

Source/destination operand type	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

Figure D.2 Instruction types for the arithmetic, logical, and data transfer instructions. The 80x86 allows the combinations shown. The only restriction is the absence of a memory-memory mode. Immediates may be 8, 16, or 32 bits in length; a register is any one of the 14 major registers in Figure D.1 (not IP or FLAGS).

- absolute
- register indirect
- based
- indexed
- based indexed with displacement
- based with scaled indexed
- based with scaled indexed and displacement

Displacements can be 8 or 32 bits in 32-bit mode, and 8 or 16 bits in 16-bit mode. If we count the size of the address as a separate addressing mode, the total is 11 addressing modes.

Although a memory operand can use any addressing mode, there are restrictions on what registers can be used in a mode. Section D.5 on 80x86 instruction encodings gives the full set of restrictions on registers, but the following description of addressing modes gives the basic register options:

- *Absolute*—With 16-bit or 32-bit displacement, depending on the mode.
- *Register indirect*—`BX`, `SI`, `DI` in 16-bit mode and `EAX`, `ECX`, `EDX`, `EBX`, `ESI`, and `EDI` in 32-bit mode.
- *Based mode with 8-bit or 16-bit/32-bit displacement*—`BP`, `BX`, `SI`, `DI` in 16-bit mode and `EAX`, `ECX`, `EDX`, `EBX`, `ESI`, and `EDI` in 32-bit mode. The displacement is either 8 bits or the size of the address mode: 16 or 32 bits. (Intel gives two different names to this single addressing mode, *based* and *indexed*, but they are essentially identical and we combine them. This book uses indexed addressing to mean something different and is explained next.)
- *Indexed*—Address is sum of two registers. The allowable combinations are `BX+SI`, `BX+DI`, `BP+SI`, and `BP+DI`. This mode is called *based indexed* on the 8086. (The 32-bit mode uses a different addressing mode to get the same effect.)

- *Based indexed with 8- or 16-bit displacement*—The address is the sum of displacement and contents of two registers. The same restrictions on registers apply as in indexed mode.
- *Base plus scaled indexed*—This addressing mode and the next were added in the 80386, and are only available in 32-bit mode. The address calculation is

$$\text{Base register} + 2^{\text{Scale}} \times \text{Index register}$$

  where *Scale* has the value 0, 1, 2, or 3, *Index register* can be any of the eight 32-bit general registers except ESP, and *Base register* can be any of the eight 32-bit general registers.
- *Base plus scaled index with 8- or 32-bit displacement*—The address is the sum of the displacement and the address calculated by the scaled mode immediately above. The same restrictions on registers apply.

The 80x86 uses Little Endian addressing.

Ideally, we would refer discussion of 80x86 logical and physical addresses to Chapter 5, but the segmented address space prevents us from hiding that information. Figure D.3 shows the memory mapping options on the generations of 80x86 machines; Chapter 5 describes the segmented protection scheme in greater detail.

The assembly language programmer clearly must specify which segment register should be used with an address, no matter which address mode is used. To save space in the instructions, segment registers are selected automatically depending on which address register is used. The rules are simple: References to instructions (IP) use the code segment register (CS), references to the stack (BP or SP) use the stack segment register (SS), and the default segment register for the other registers is the data segment register (DS). The next section explains how they can be overridden.

D.3 80x86 Integer Operations

The 8086 provides support for both 8-bit (*byte*) and 16-bit (called *word*) data types. The data type distinctions apply to register operations as well as memory accesses. The 80386 adds 32-bit addresses and data, called double words. Almost every operation works on both 8-bit data and one longer data size. That size is determined by the mode and is either 16 or 32 bits.

Clearly some programs want to operate on data of all three sizes, so the 80x86 architects provide a convenient way to specify each version without expanding code size significantly. They decided that most programs would be dominated by either 16- or 32-bit data, and so it made sense to be able to set a default large size. This default size is set by a bit in the code segment register. To override the default size, an 8-bit *prefix* is attached to the instruction to tell the machine to use the other large size for this instruction.

The prefix solution was borrowed from the 8086, which allows multiple prefixes to modify instruction behavior. The three original prefixes override the

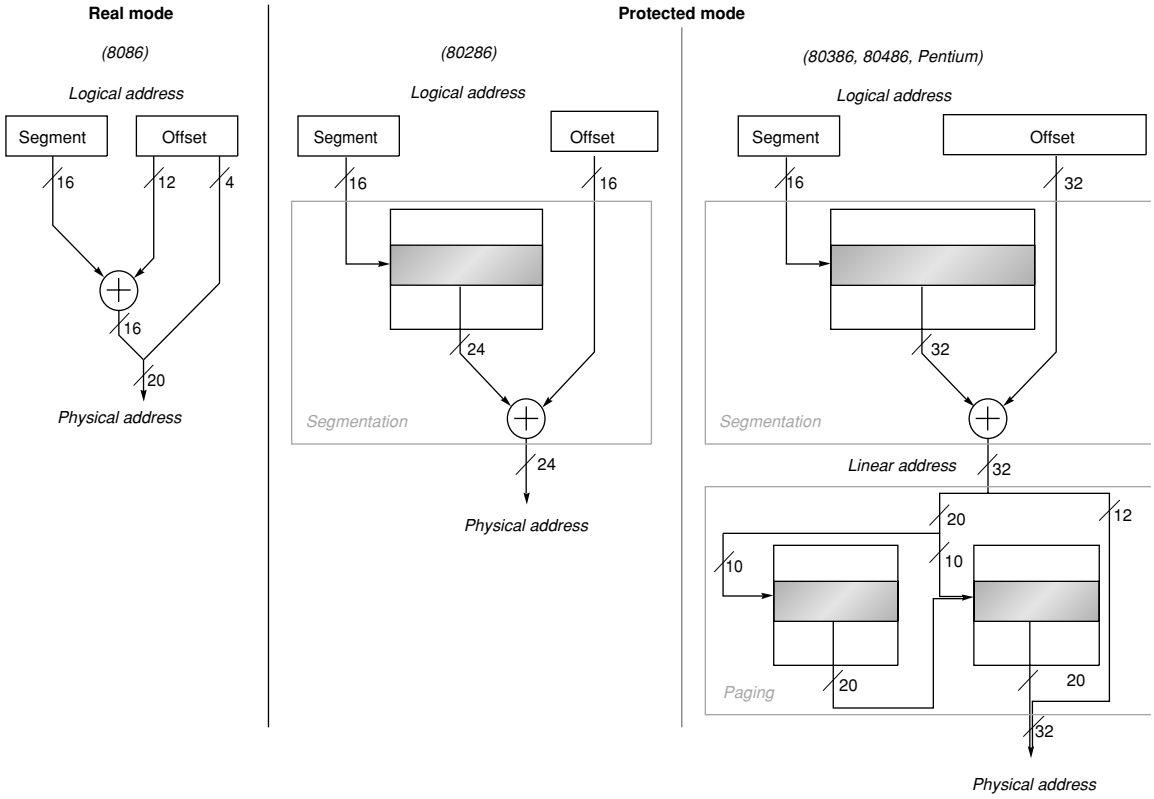


Figure D.3 The original segmented scheme of the 8086 is shown on the left. All 80x86 processors support this style of addressing, called *real mode*. It simply takes the contents of a segment register, shifts it left 4 bits, and adds it to the 16-bit offset, forming a 20-bit physical address. The 80286 (center) used the contents of the segment register to select a segment descriptor, which includes a 24-bit base address among other items. It is added to the 16-bit offset to form the 24-bit physical address. The 80386 and successors (right) expand this base address in the segment descriptor to 32 bits and also add an optional paging layer below segmentation. A 32-bit linear address is first formed from the segment and offset, and then this address is divided into two 10-bit fields and a 12-bit page offset. The first 10-bit field selects the entry in the first-level page table, and then this entry is used in combination with the second 10-bit field to access the second-level page table to select the upper 20 bits of the physical address. Prepending this 20-bit address to the final 12-bit field gives the 32-bit physical address. Paging can be turned off, redefining the 32-bit linear address as the physical address. Note that a "flat" 80x86 address space comes simply by loading the same value in all the segment registers; that is, it doesn't matter which segment register is selected.

default segment register, lock the bus so as to perform a semaphore (see Chapter 6), or repeat the following instruction until CX counts down to zero. This last prefix was intended to be paired with a byte move instruction to move a variable number of bytes. The 80386 also added a prefix to override the default address size.

The 80x86 integer operations can be divided into four major classes:

1. Data movement instructions, including move, push, and pop.

2. Arithmetic and logic instructions, including logical operations, test, shifts, and integer and decimal arithmetic operations.
3. Control flow, including conditional branches and unconditional jumps, calls, and returns.
4. String instructions, including string move and string compare.

Figure D.4 shows some typical 80x86 instructions and their functions.

The data transfer, arithmetic, and logic instructions are unremarkable, except that the arithmetic and logic instruction operations allow the destination to be either a register or a memory location.

Control flow instructions must be able to address destinations in another segment. This is handled by having two types of control flow instructions: "near" for intrasegment (within a segment) and "far" for intersegment (between segments) transfers. In far jumps, which must be unconditional, two 16-bit quantities follow the opcode in 16-bit mode. One of these is used as the instruction pointer, while the other is loaded into CS and becomes the new code segment. In 32-bit mode the first field is expanded to 32 bits to match the 32-bit program counter (EIP).

Calls and returns work similarly—a far call pushes the return instruction pointer and return segment on the stack and loads both the instruction pointer and the code segment. A far return pops both the instruction pointer and the code segment from the stack. Programmers or compiler writers must be sure to always use the same type of call *and* return for a procedure—a near return does not work with a far call, and vice versa.

Instruction	Function
JE name	if equal(CC) {IP←name}; IP–128 ≤ name < IP+128
JMP name	IP←name
CALLF name, seg	SP←SP–2; M[SS:SP]←IP+5; SP←SP–2; M[SS:SP]←CS; IP←name; CS←seg;
MOVW BX,[DI+45]	BX←$_{16}$M[DS:DI+45]
PUSH SI	SP←SP–2; M[SS:SP]←SI
POP DI	DI←M[SS:SP]; SP←SP+2
ADD AX,#6765	AX←AX+6765
SHL BX,1	BX←BX$_{1..15}$ ## 0
TEST DX,#42	Set CC flags with DX & 42
MOVSB	M[ES:DI]←$_{8}$M[DS:SI]; DI←DI+1; SI←SI+1

Figure D.4 Some typical 80x86 instructions and their functions. A list of frequent operations appears in Figure D.5. We use the abbreviation SR:X to indicate the formation of an address with segment register SR and offset X. This effective address corresponding to SR:X is (SR<<4)+X. The CALLF saves the IP of the next instruction and the current CS on the stack. The hardware description language is described on the back inside cover of this book.

String instructions are part of the 8080 ancestry of the 80x86 and are not commonly executed in most programs.

Figure D.5 lists some of the integer 80x86 instructions. Many of the instructions are available in both byte and word formats.

Instruction	Meaning
Control	**Conditional and unconditional branches**
JNZ, JZ	Jump if condition to IP + 8-bit offset; JNE (for JNZ), JE (for JZ) are alternative names
JMP, JMPF	Unconditional jump—8- or 16-bit offset intrasegment (near), and intersegment (far) versions
CALL, CALLF	Subroutine call—16-bit offset; return address pushed; near and far versions
RET, RETF	Pops return address from stack and jumps to it; near and far versions
LOOP	Loop branch—decrement CX; jump to IP + 8-bit displacement if CX ≠ 0
Data transfer	**Move data between registers or between register and memory**
MOV	Move between two registers or between register and memory
PUSH	Push source operand on stack
POP	Pop operand from stack top to a register
LES	Load ES and one of the GPRs from memory
Arithmetic/logical	**Arithmetic and logical operations using the data registers and memory**
ADD	Add source to destination; register-memory format
SUB	Subtract source from destination; register-memory format
CMP	Compare source and destination; register-memory format
SHL	Shift left
SHR	Shift logical right
RCR	Rotate right with carry as fill
CBW	Convert byte in AL to word in AX
TEST	Logical AND of source and destination sets flags
INC	Increment destination; register-memory format
DEC	Decrement destination; register-memory format
OR	Logical OR; register-memory format
XOR	Exclusive OR; register-memory format
String instructions	**Move between string operands; length given by a repeat prefix**
MOVS	Copies from string source to destination; may be repeated
LODS	Loads a byte or word of a string into the A register

Figure D.5 Some typical operations on the 80x86. Many operations use register-memory format, where either the source or the destination may be memory and the other may be a register or immediate operand.

D.4 80x86 Floating-Point Operations

Intel provided a stack architecture with its floating-point instructions: loads push numbers onto the stack, operations find operands in the top two elements of the stacks, and stores can pop elements off the stack, just as the stack example in Figure 2.2 on page 93 suggests.

Intel supplemented this stack architecture with instructions and addressing modes that allow the architecture to have some of the benefits of a register-memory model. In addition to finding operands in the top two elements of the stack, one operand can be in memory or in one of the seven registers below the top of the stack.

This hybrid is still a restricted register-memory model, however, in that loads always move data to the top of the stack while incrementing the top of stack pointer and stores can only move the top of stack to memory. Intel uses the notation `ST` to indicate the top of stack, and `ST(i)` to represent the *i*th register below the top of stack.

One novel feature of this architecture is that the operands are wider in the register stack than they are stored in memory, and all operations are performed at this wide internal precision. Numbers are automatically converted to the internal 80-bit format on a load and converted back to the appropriate size on a store. Memory data can be 32-bit (single-precision) or 64-bit (double-precision) floating-point numbers, called *real* by Intel. The register-memory version of these instructions will then convert the memory operand to this Intel 80-bit format before performing the operation. The data transfer instructions also will automatically convert 16- and 32-bit integers to reals, and vice versa, for integer loads and stores.

The 80x86 floating-point operations can be divided into four major classes:

1. Data movement instructions, including load, load constant, and store.
2. Arithmetic instructions, including add, subtract, multiply, divide, square root, and absolute value.
3. Comparison, including instructions to send the result to the integer CPU so that it can branch.
4. Transcendental instructions, including sine, cosine, log, and exponentiation.

Figure D.6 shows some of the 60 floating-point operations. We use the curly brackets `{}` to show optional variations of the basic operations: `{I}` means there is an integer version of the instruction, `{P}` means this variation will pop one operand off the stack after the operation, and `{R}` means reverse the sense of the operands in this operation.

Not all combinations are provided. Hence

```
F{I}SUB{R}{P}
```

represents these instructions found in the 80x86:

Data transfer	Arithmetic	Compare	Transcendental
F{I}LD mem/ST(i)	F{I}ADD{P} mem/ST(i)	F{I}COM{P}{P}	FPATAN
F{I}ST{P} mem/ST(i)	F{I}SUB{R}{P} mem/ST(i)	F{I}UCOM{P}{P}	F2XM1
FLDPI	F{I}MUL{P} mem/ST(i)	FSTSW AX/mem	FCOS
FLD1	F{I}DIV{R}{P} mem/ST(i)		FPTAN
FLDZ	FSQRT		FPREM
	FABS		FSIN
	FRNDINT		FYL2X

Figure D.6 The floating-point instructions of the 80x86. The first column shows the data transfer instructions, which move data to memory or to one of the registers below the top of the stack. The last three operations push constants on the stack: pi, 1.0, and 0.0. The second column contains the arithmetic operations described above. Note that the last three operate only on the top of stack. The third column is the compare instructions. Since there are no special floating-point branch instructions, the result of the compare must be transferred to the integer CPU via the FSTSW instruction, either into the AX register or into memory, followed by an SAHF instruction to set the condition codes. The floating-point comparison can then be tested using integer branch instructions. The final column gives the higher-level floating-point operations.

```
FSUB
FISUB
FSUBR
FISUBR
FSUBP
FSUBRP
```

There are no pop or reverse pop versions of the integer subtract instructions.

Note that we get even more combinations when including the operand modes for these operations. The floating-point add has these options, ignoring the integer and pop versions of the instruction:

FADD		Both operands in stack, result replaces top of stack.
FADD	ST(i)	One source operand is *i*th register below the top of stack, and the result replaces the top of stack.
FADD	ST(i),ST	One source operand is the top of stack, and the result replaces *i*th register below the top of stack.
FADD	mem32	One source operand is a 32-bit location in memory, and the result replaces the top of stack.
FADD	mem64	One source operand is a 64-bit location in memory, and the result replaces the top of stack.

As mentioned above SSE2 presents yet another model of IEEE floating-point registers.

D.5 80x86 Instruction Encoding

Saving the worst for last, the encoding of instructions in the 8086 is complex, with many different instruction formats. Instructions may vary from one byte, when there are no operands, to up to six bytes, when the instruction contains a 16-bit immediate and uses 16-bit displacement addressing. Prefix instructions increase 8086 instruction length beyond the obvious sizes.

The 80386 additions expand the instruction size even further, as Figure D.7 shows. Both the displacement and immediate fields can be 32 bits long, two more prefixes are possible, the opcode can be 16 bits long, and the scaled index mode specifier adds another 8 bits. The maximum possible 80386 instruction is 17 bytes long.

Figure D.8 shows the instruction format for several of the example instructions in Figure D.4. The opcode byte usually contains a bit saying whether the operand is a byte wide or the larger size, 16 bits or 32 bits depending on the mode. For some instructions the opcode may include the addressing mode and the register; this is true in many instructions that have the form `register ←register op immediate`. Other instructions use a "postbyte" or extra opcode

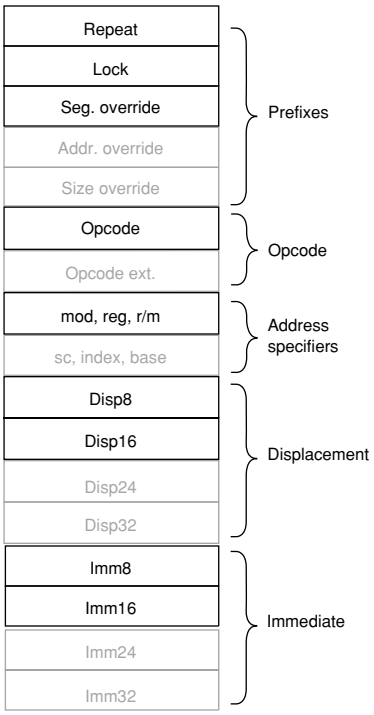


Figure D.7 The instruction format of the 8086 (black type) and the extensions for the 80386 (shaded type). Every field is optional except the opcode.

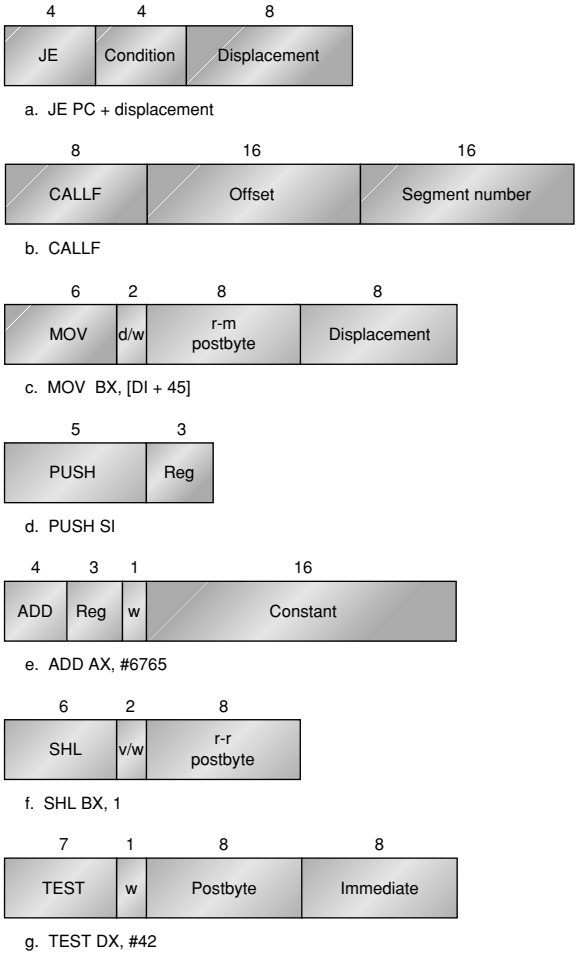


Figure D.8 Typical 8086 instruction formats. The encoding of the postbyte is shown in Figure D.9. Many instructions contain the 1-bit field w, which says whether the operation is a byte or a word. Fields of the form v/w or d/w are a d-field or v-field followed by the w-field. The d-field in `MOV` is used in instructions that may move to or from memory and shows the direction of the move. The field v in the `SHL` instruction indicates a variable-length shift; variable-length shifts use a register to hold the shift count. The `ADD` instruction shows a typical optimized short encoding usable only when the first operand is `AX`. Overall instructions may vary from one to six bytes in length.

byte, labeled "mod, reg, r/m" in Figure D.7, which contains the addressing mode information. This postbyte is used for many of the instructions that address memory. The based with scaled index uses a second postbyte, labeled "sc, index, base" in Figure D.7.

The floating-point instructions are encoded in the escape opcode of the 8086 and the postbyte address specifier. The memory operations reserve 2 bits to decide whether the operand is a 32- or 64-bit real or a 16- or 32-bit integer. Those same 2 bits are used in versions that do not access memory to decide whether the stack should be popped after the operation and whether the top of stack or a lower register should get the result.

Alas, you cannot separate the restrictions on registers from the encoding of the addressing modes in the 80x86. Hence Figures D.9 and D.10 show the encoding of the two postbyte address specifiers for both 16- and 32-bit mode.

D.6 Putting It All Together: Measurements of Instruction Set Usage

In this section we present detailed measurements for the 80x86, and then compare the measurements to MIPS for the same programs. To facilitate comparisons among dynamic instruction set measurements, we use a subset of the SPEC92 programs. The 80x86 results were taken in 1994 using the Sun Solaris FORTRAN and C compilers V2.0 and executed in 32-bit mode. These compilers were comparable in quality to the compilers used for MIPS.

		w = 1			mod = 0		mod = 1		mod = 2		
reg	w = 0	16b	32b	r/m	16b	32b	16b	32b	16b	32b	mod = 3
0	AL	AX	EAX	0	addr=BX+SI	=EAX	*same*	*same*	*same*	*same*	*same*
1	CL	CX	ECX	1	addr=BX+DI	=ECX	*addr as*	*addr as*	*addr as*	*addr as*	*as*
2	DL	DX	EDX	2	addr=BP+SI	=EDX	*mod=0*	*mod=0*	*mod=0*	*mod=0*	*reg*
3	BL	BX	EBX	3	addr=BP+SI	=EBX	*+ disp8*	*+ disp8*	*+ disp16*	*+ disp32*	*field*
4	AH	SP	ESP	4	addr=SI	=(sib)	SI+disp16	(sib)+disp8	SI+disp8	(sib)+disp32	"
5	CH	BP	EBP	5	addr=DI	=disp32	DI+disp8	EBP+disp8	DI+disp16	EBP+disp32	"
6	DH	SI	ESI	6	addr=disp16	=ESI	BP+disp8	ESI+disp8	BP+disp16	ESI+disp32	"
7	BH	DI	EDI	7	addr=BX	=EDI	BX+disp8	EDI+disp8	BX+disp16	EDI+disp32	"

Figure D.9 The encoding of the first address specifier of the 80x86, "*mod, reg, r/m.*" The first four columns show the encoding of the 3-bit reg field, which depends on the w bit from the opcode and whether the machine is in 16- or 32-bit mode. The remaining columns explain the mod and r/m fields. The meaning of the 3-bit r/m field depends on the value in the 2-bit mod field and the address size. Basically, the registers used in the address calculation are listed in the sixth and seventh columns, under mod = 0, with mod = 1 adding an 8-bit displacement and mod = 2 adding a 16- or 32-bit displacement, depending on the address mode. The exceptions are r/m = 6 when mod = 1 or mod = 2 in 16-bit mode selects BP plus the displacement; r/m = 5 when mod =1 or mod = 2 in 32-bit mode selects EBP plus displacement; and r/m = 4 in 32-bit mode when mod ≠3 (sib) means use the scaled index mode shown in Figure D.10. When mod = 3, the r/m field indicates a register, using the same encoding as the reg field combined with the w bit.

Index		Base
0	EAX	EAX
1	ECX	ECX
2	EDX	EDX
3	EBX	EBX
4	no index	ESP
5	EBP	if mod = 0, disp32 if mod ≠ 0, EBP
6	ESI	ESI
7	EDI	EDI

Figure D.10 Based plus scaled index mode address specifier found in the 80386. This mode is indicated by the *(sib)* notation in Figure D.9. Note that this mode expands the list of registers to be used in other modes: register indirect using ESP comes from Scale = 0, Index = 4, and Base = 4, and base displacement with EBP comes from Scale = 0, Index = 5, and mod = 0. The two-bit scale field is used in this formula of the effective address: Base register + $2^{\text{Scale}} \times$ Index register.

Remember that these measurements depend on the benchmarks chosen and the compiler technology used. Although we feel that the measurements in this section are reasonably indicative of the usage of these architectures, other programs may behave differently from any of the benchmarks here, and different compilers may yield different results. In doing a real instruction set study, the architect would want to have a much larger set of benchmarks, spanning as wide an application range as possible, and consider the operating system and its usage of the instruction set. Single-user benchmarks like those measured here do not necessarily behave in the same fashion as the operating system.

We start with an evaluation of the features of the 80x86 in isolation, and later compare instruction counts with those of DLX.

Measurements of 80x86 Operand Addressing

We start with addressing modes. Figure D.11 shows the distribution of the operand types in the 80x86. These measurements cover the "second" operand of the operation; for example,

```
mov EAX, [45]
```

counts as a single memory operand. If the types of the first operand were counted, the percentage of register usage would increase by about a factor of 1.5.

The 80x86 memory operands are divided into their respective addressing modes in Figure D.12. Probably the biggest surprise is the popularity of the addressing modes added by the 80386, the last four rows of the figure. They

	Integer average	FP average
Register	45%	22%
Immediate	16%	6%
Memory	39%	72%

Figure D.11 Operand type distribution for the average of five SPECint92 programs (compress, eqntott, espresso, gcc, li) and the average of five SPECfp92 programs (doduc, ear, hydro2d, mdljdp2, su2cor).

Addressing mode	Integer average	FP average
Register indirect	13%	3%
Base + 8-bit disp.	31%	15%
Base + 32-bit disp.	9%	25%
Indexed	0%	0%
Based indexed + 8-bit disp.	0%	0%
Based indexed + 32-bit disp.	0%	1%
Base + scaled indexed	22%	7%
Base + scaled indexed + 8-bit disp.	0%	8%
Base + scaled indexed + 32-bit disp.	4%	4%
32-bit direct	20%	37%

Figure D.12 Operand addressing mode distribution by program. This chart does not include addressing modes used by branches or control instructions.

account for about half of all the memory accesses. Another surprise is the popularity of direct addressing. On most other machines, the equivalent of the direct addressing mode is rare. Perhaps the segmented address space of the 80x86 makes direct addressing more useful, since the address is relative to a base address from the segment register.

These addressing modes largely determine the size of the Intel instructions. Figure D.13 shows the distribution of instruction sizes. The average number of bytes per instruction for integer programs is 2.8, with a standard deviation of 1.5, and 4.1 with a standard deviation of 1.9 for floating-point programs. The difference in length arises partly from the differences in the addressing modes: Integer programs rely more on the shorter register indirect and 8-bit displacement addressing modes, while floating-point programs more frequently use the 80386 addressing modes with the longer 32-bit displacements.

Given that the floating-point instructions have aspects of both stacks and registers, how are they used? Figure D.14 shows that, at least for the compilers used

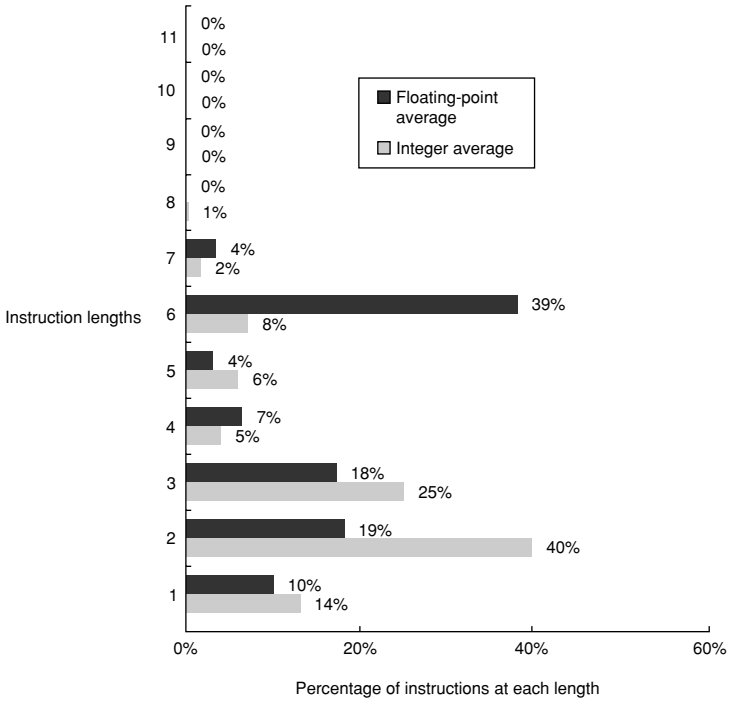


Figure D.13 Averages of the histograms of 80x86 instruction lengths for five SPECint92 programs and for five SPECfp92 programs, all running in 32-bit mode.

Option	doduc	ear	hydro2d	mdljdp2	su2cor	FP average
Stack (2nd operand ST (1))	1.1%	0.0%	0.0%	0.2%	0.6%	0.4%
Register (2nd operand ST(i), i > 1)	17.3%	63.4%	14.2%	7.1%	30.7%	26.5%
Memory	81.6%	36.6%	85.8%	92.7%	68.7%	73.1%

Figure D.14 The percentage of instructions for the floating-point operations (add, sub, mul, div) that use each of the three options for specifying a floating-point operand on the 80x86. The three options are 1) the strict stack model of implicit operands on the stack, 2) register version naming an explicit operand that is not one of the top two elements of the stack, and 3) memory operand.

in this measurement, the stack model of execution is rarely followed. (See Section D.8 for a historical explanation of this observation.)

Finally, Figures D.15 and D.16 show the instruction mixes for 10 SPEC92 programs.

Instruction	compress	eqntott	espresso	gcc (cc1)	li	Int. average
load	20.8%	18.5%	21.9%	24.9%	23.3%	22%
store	13.8%	3.2%	8.3%	16.6%	18.7%	12%
add	10.3%	8.8%	8.15%	7.6%	6.1%	8%
sub	7.0%	10.6%	3.5%	2.9%	3.6%	5%
mul				0.1%		0%
div						0%
compare	8.2%	27.7%	15.3%	13.5%	7.7%	16%
mov reg-reg	7.9%	0.6%	5.0%	4.2%	7.8%	4%
load imm	0.5%	0.2%	0.6%	0.4%		0%
cond. branch	15.5%	28.6%	18.9%	17.4%	15.4%	20%
uncond. branch	1.2%	0.2%	0.9%	2.2%	2.2%	1%
call	0.5%	0.4%	0.7%	1.5%	3.2%	1%
return, jmp indirect	0.5%	0.4%	0.7%	1.5%	3.2%	1%
shift	3.8%		2.5%	1.7%		1%
and	8.4%	1.0%	8.7%	4.5%	8.4%	6%
or	0.6%		2.7%	0.4%	0.4%	1%
other (xor, not, . . .)	0.9%		2.2%	0.1%		1%
load FP						0%
store FP						0%
add FP						0%
sub FP						0%
mul FP						0%
div FP						0%
compare FP						0%
mov reg-reg FP						0%
other (abs, sqrt, . . .)						0%

Figure D.15 80x86 instruction mix for five SPECint92 programs.

Comparative Operation Measurements

Figures D.17 and D.18 show the number of instructions executed for each of the 10 programs on the 80x86 and the ratio of instruction execution compared with that for DLX: Numbers less than 1.0 mean the 80x86 executes fewer instructions than DLX. The instruction count is surprisingly close to DLX for many integer programs, as you would expect a load-store instruction set architecture like DLX to execute more instructions than a register-memory architecture like the 80x86. The floating-point programs always have higher counts for the 80x86, presumably due to the lack of floating-point registers and the use of a stack architecture.

Instruction	doduc	ear	hydro2d	mdljdp2	su2cor	FP average
load	8.9%	6.5%	18.0%	27.6%	27.6%	20%
store	12.4%	3.1%	11.5%	7.8%	7.8%	8%
add	5.4%	6.6%	14.6%	8.8%	8.8%	10%
sub	1.0%	2.4%	3.3%	2.4%	2.4%	3%
mul						0%
div						0%
compare	1.8%	5.1%	0.8%	1.0%	1.0%	2%
mov reg-reg	3.2%	0.1%	1.8%	2.3%	2.3%	2%
load imm	0.4%	1.5%				0%
cond. branch	5.4%	8.2%	5.1%	2.7%	2.7%	5%
uncond branch	0.8%	0.4%	1.3%	0.3%	0.3%	1%
call	0.5%	1.6%		0.1%	0.1%	0%
return, jmp indirect	0.5%	1.6%		0.1%	0.1%	0%
shift	1.1%		4.5%	2.5%	2.5%	2%
and	0.8%	0.8%	0.7%	1.3%	1.3%	1%
or	0.1%			0.1%	0.1%	0%
other (xor, not, . . .)						0%
load FP	14.1%	22.5%	9.1%	12.6%	12.6%	14%
store FP	8.6%	11.4%	4.1%	6.6%	6.6%	7%
add FP	5.8%	6.1%	1.4%	6.6%	6.6%	5%
sub FP	2.2%	2.7%	3.1%	2.9%	2.9%	3%
mul FP	8.9%	8.0%	4.1%	12.0%	12.0%	9%
div FP	2.1%		0.8%	0.2%	0.2%	0%
compare FP	9.4%	6.9%	10.8%	0.5%	0.5%	5%
mov reg-reg FP	2.5%	0.8%	0.3%	0.8%	0.8%	1%
other (abs, sqrt, . . .)	3.9%	3.8%	4.1%	0.8%	0.8%	2%

Figure D.16 80x86 instruction mix for five SPECfp92 programs.

Another question is the total amount of data traffic for the 80x86 versus DLX, since the 80x86 can specify memory operands as part of operations while DLX can only access via loads and stores. Figures D.17 and D.18 also show the data reads, data writes, and data read-modify-writes for these 10 programs. The total accesses ratio to DLX of each memory access type is shown in the bottom rows, with the read-modify-write counting as one read and one write. The 80x86 performs about two to four times as many data accesses as DLX for floating-point programs, and 1.25 times as many for integer programs. Finally, Figure D.19 shows the percentage of instructions in each category for 80x86 and DLX.

	compress	eqntott	espresso	gcc (cc1)	li	Int. avg.
Instructions executed on 80x86 (millions)	2226	1203	2216	3770	5020	
Instructions executed ratio to DLX	0.61	1.74	0.85	0.96	0.98	1.03
Data reads on 80x86 (millions)	589	229	622	1079	1459	
Data writes on 80x86 (millions)	311	39	191	661	981	
Data read-modify-writes on 80x86 (millions)	26	1	129	48	48	
Total data reads on 80x86 (millions)	615	230	751	1127	1507	
Data read ratio to DLX	0.85	1.09	1.38	1.25	0.94	1.10
Total data writes on 80x86 (millions)	338	40	319	709	1029	
Data write ratio to DLX	1.67	9.26	2.39	1.25	1.20	3.15
Total data accesses on 80x86 (millions)	953	269	1070	1836	2536	
Data access ratio to DLX	1.03	1.25	1.58	1.25	1.03	1.23

Figure D.17 Instructions executed and data accesses on 80x86 and ratios compared to DLX for five SPECint92 programs.

	doduc	ear	hydro2d	mdljdp2	su2cor	FP average
Instructions executed on 80x86 (millions)	1223	15,220	13,342	6197	6197	
Instructions executed ratio to DLX	1.19	1.19	2.53	2.09	1.62	1.73
Data reads on 80x86 (millions)	515	6007	5501	3696	3643	
Data writes on 80x86 (millions)	260	2205	2085	892	892	
Data read-modify-writes on 80x86 (millions)	1	0	189	124	124	
Total data reads on 80x86 (millions)	517	6007	5690	3820	3767	
Data read ratio to DLX	2.04	2.36	4.48	4.77	3.91	3.51
Total data writes on 80x86 (millions)	261	2205	2274	1015	1015	
Data write ratio to DLX	3.68	33.25	38.74	16.74	9.35	20.35
Total data accesses on 80x86 (millions)	778	8212	7965	4835	4782	
Data access ratio to DLX	2.40	3.14	5.99	5.73	4.47	4.35

Figure D.18 Instructions executed and data accesses for five SPECfp92 programs on 80x86 and ratio to DLX.

D.7 Concluding Remarks

Beauty is in the eye of the beholder.

Old Adage

As we have seen, "orthogonal" is not a term found in the Intel architectural dictionary. To fully understand which registers and which addressing modes are

	Integer average		FP average	
Category	x86	DLX	x86	DLX
Total data transfer	34%	36%	28%	2%
Total integer arithmetic	34%	31%	16%	12%
Total control	24%	20%	6%	10%
Total logical	8%	13%	3%	2%
Total FP data transfer	0%	0%	22%	33%
Total FP arithmetic	0%	0%	25%	41%

Figure D.19 Percentage of instructions executed by category for 80x86 and DLX for the averages of five SPECint92 and SPECfp92 programs of Figures D.17 and D.18.

available, you need to see the encoding of all addressing modes and sometimes the encoding of the instructions.

Some argue that the inelegance of the 80x86 instruction set is unavoidable, the price that must be paid for rampant success by any architecture. We reject that notion. Obviously no successful architecture can jettison features that were added in previous implementations, and over time some features may be seen as undesirable. The awkwardness of the 80x86 began at its core with the 8086 instruction set and was exacerbated by the architecturally inconsistent expansions of the 8087, 80286, and 80386.

A counterexample is the IBM 360/370 architecture, which is much older than the 80x86. It dominates the mainframe market just as the 80x86 dominates the PC market. Due undoubtedly to a better base and more compatible enhancements, this instruction set makes much more sense than the 80x86 more than 30 years after its first implementation.

For better or worse, Intel had a 16-bit microprocessor years before its competitors' more elegant architectures, and this head start led to the selection of the 8086 as the CPU for the IBM PC. What it lacks in style is made up in quantity, making the 80x86 beautiful from the right perspective.

The saving grace of the 80x86 is that its architectural components are not too difficult to implement, as Intel has demonstrated by rapidly improving performance of integer programs since 1978. High floating-point performance is a larger challenge in this architecture.

D.8 Historical Perspective and References

The complexity of the x86 is not an impassable barrier. . . . The biggest weakness in the x86 instruction set is the lack of registers coupled with an extremely painful addressing scheme.

Mike Johnson, Leader of 80x86 Design at AMD
Microprocessor Report (1994)

There are numerous descriptions of the 80x86 architecture that have been published—Wakerly's [1989] is both concise and easy to understand. Crawford and Gelsinger [1988] is a thorough description of the 80386.

The ancestors of the 80x86 were the first microprocessors, produced late in the first half of the 1970s. The Intel 4004 and 8008 were extremely simple 4- and 8-bit accumulator-style machines. Morse et al. [1980] describe the evolution of the 8086 from the 8080 in the late 1970s as an attempt to provide a 16-bit machine with better throughput. At that time almost all programming for microprocessors was done in assembly language—both memory and compilers were in short supply. Intel wanted to keep its base of 8080 users, so the 8086 was designed to be "compatible" with the 8080. The 8086 was *never* object-code compatible with the 8080, but the machines were close enough that translation of assembly language programs could be done automatically.

In early 1980, IBM selected a version of the 8086 with an 8-bit external bus, called the 8088, for use in the IBM PC. They chose the 8-bit version to reduce the cost of the machine. This choice, together with the tremendous success of the IBM PC, has made the 8086 architecture ubiquitous. The success of the IBM PC was due in part because IBM opened the architecture of the PC and enabled the PC-clone industry to flourish. As discussed in the introduction of this appendix, the 80286, 80386, 80486, Pentium, and P6 have extended the architecture and provided a series of performance enhancements.

Although the 68000 was chosen for the popular Macintosh, the Macintosh was never as pervasive as the PC, partly because Apple did not allow clones until recently, and the 68000 did not acquire the same software leverage that the 8086 enjoys. The Motorola 68000 may have been more significant *technically* than the 8086, but the impact of the selection by IBM and IBM's open architecture strategy dominated the technical advantages of the 68000 in the market.

Kahan's history [1990] of the stack architecture selection for the 8086 is entertaining. The floating-point architecture of the companion 8087 had to be retrofitted into the 8086 opcode space, making it inconvenient to offer two operands per instruction as found in the rest of the 8086. Hence the decision for one operand per instruction using a stack: "The designer's task was to make a Virtue of this Necessity." Rather than the classical stack architecture, which has no provision for avoiding common subexpressions from being pushed and popped from memory into the top of the stack found in registers, Intel tried to combine a flat register file with a stack. The reasoning was the restriction of the top of stack as one operand was not so bad since it only required the execution of an `FXCH` instruction (which swapped registers) to get the same result as a two-operand instruction, and `FXCH` was much faster than the floating-point operations of the 8087.

Since floating-point expressions are not that complex, Kahan reasoned that eight registers meant that the stack would rarely overflow. Hence he urged that the 8087 use this hybrid scheme with the provision that stack overflow or stack underflow would interrupt the 8086 so that interrupt software could give the illusion to the compiler writer of an unlimited stack for floating-point data. The Intel 8087 was implemented in Israel, and 7500 miles and 10 time zones made communication difficult from California. According to Palmer and Morse [1984]:

> Unfortunately, nobody tried to write a software stack manager until after the 8087 was built, and by then it was too late; what was too complicated to perform in hardware turned out to be even worse in software. One thing found lacking is the ability to conveniently determine if an invalid-operation exception is indeed due to a stack overflow. . . . Also lacking is the ability to restart the instruction that caused the stack overflow . . . [p. 93]

The result is that the stack exceptions are too slow to handle in software. As Kahan [1990] says:

> Consequently, almost all higher-level languages' compilers emit inefficient code for the 80x87 family, degrading the chip's performance by typically 50% with spurious stores and loads necessary simply to preclude stack over/underflow. . . .
>
> I still regret that the 8087's stack implementation was not quite so neat as my original intention. . . . If the original design had been realized, compilers today would use the 80x87 and its descendents more efficiently, and Intel's competitors could more easily market faster but compatible 80x87 imitations.

The P6 renames the floating-point registers (see Chapter 3), effectively providing up to 40 floating-point registers at any given instant. The main effect of the stack organization is to force design teams to use transistors for dereferencing the stack before doing the renaming.

Hewlett-Packard and Intel have announced a new, common instruction set architecture. It is also upward compatible with the 80x86, and thus the 80x86 instruction set will be available in some form in computers of this century. Instruction set anthropologists will peel off layer by layer from such machines until they uncover artifacts from the first microprocessor. Given such a find, how will they judge 20th-century computer architecture?

References

Crawford, J., and P. Gelsinger [1988]. *Programming the 80386,* Sybex Books, Alameda, Calif.

Kahan, J. [1990]. "On the advantage of the 8087's stack," unpublished course notes, Computer Science Division, University of California at Berkeley.

Morse, S., B. Ravenal, S. Mazor, and W. Pohlman [1980]. "Intel microprocessors—8080 to 8086," *Computer* 13:10 (October).

Palmer, J., and S. Morse [1984]. *The 8087 Primer,* J. Wiley, New York, p. 93.

Wakerly, J. [1989]. *Microcomputer Architecture and Programming,* J. Wiley, New York.

E.1	Introduction	E-2
E.2	VAX Operands and Addressing Modes	E-2
E.3	Encoding VAX Instructions	E-5
E.4	VAX Operations	E-6
E.5	An Example to Put It All Together: swap	E-10
E.6	A Longer Example: sort	E-13
E.7	Fallacies and Pitfalls	E-18
E.8	Concluding Remarks	E-19
E.9	Historical Perspective and Further Reading	E-20
	Exercises	E-21

E

Another Alternative to RISC: The VAX Architecture

In principle, there is no great challenge in designing a large virtual address minicomputer system.…The real challenge lies in two areas: compatibility—very tangible and important; and simplicity—intangible but nonetheless important.

William Strecker
"VAX-11/780—A Virtual Address Extension to the PDP-11 Family," AFIPS Proc., *National Computer Conference, 1978.*

Entities should not be multiplied unnecessarily.

William of Occam
Quodlibeta Septem, 1320
(This quote is known as "Occam's Razor.")

© 2003 Elsevier Science (USA). All rights reserved.

E.1 Introduction

To enhance your understanding of instruction set architectures, we chose the VAX as the representative *Complex Instruction Set Computers* (*CISC*) because it is so different from MIPS and yet still easy to understand. By seeing two such divergent styles, we are confident that you will be able to learn other instruction sets on your own.

At the time the VAX was designed, the prevailing philosophy was to create instruction sets that were close to programming languages in order to simplify compilers. For example, because programming languages had loops, instruction sets should have loop instructions. As VAX architect William Strecker said ("VAX-11/780—A Virtual Address Extension to the PDP-11 Family," *AFIPS Proc.,* National Computer Conference, 1978):

> A major goal of the VAX-11 instruction set was to provide for effective compiler generated code. Four decisions helped to realize this goal: . . . 1) A very regular and consistent treatment of operators. . . . 2) An avoidance of instructions unlikely to be generated by a compiler. . . . 3) Inclusions of several forms of common operators. . . . 4) Replacement of common instruction sequences with single instructions. Examples include procedure calling, multiway branching, loop control, and array subscript calculation.

Recall that DRAMs of the mid-1970s contained less than 1/1000th the capacity of today's DRAMs, so code space was also critical. Hence, another prevailing philosophy was to minimize code size, which is de-emphasized in fixed-length instruction sets like MIPS. For example, MIPS address fields always use 16 bits, even when the address is very small. In contrast, the VAX allows instructions to be a variable number of bytes, so there is little wasted space in address fields.

Books the size of the one you are reading have been written just about the VAX, so this VAX extension cannot be exhaustive. Hence, the following sections describe only a few of its addressing modes and instructions. To show the VAX instructions in action, later sections show VAX assembly code for two C procedures. The general style will be to contrast these instructions with the MIPS code that you are already familiar with.

The differing goals for VAX and MIPS have led to very different architectures. The VAX goals, simple compilers and code density, led to the powerful addressing modes, powerful instructions, and efficient instruction encoding. The MIPS goals were high performance via pipelining, ease of hardware implementation, and compatibility with highly optimizing compilers. The MIPS goals led to simple instructions, simple addressing modes, fixed-length instruction formats, and a large number of registers.

E.2 VAX Operands and Addressing Modes

The VAX is a 32-bit architecture, with 32-bit-wide addresses and 32-bit-wide registers. Yet the VAX supports many other data sizes and types, as Figure E.1

shows. Unfortunately, VAX uses the name "word" to refer to 16-bit quantities; in this text a word means 32 bits. Figure E.1 shows the conversion between the MIPS data type names and the VAX names. Be careful when reading about VAX instructions, as they refer to the names of the VAX data types.

The VAX provides 16 32-bit registers. The VAX assembler uses the notation `r0`, `r1`, . . . , `r15` to refer to these registers, and we will stick to that notation. Alas, 4 of these 16 registers are effectively claimed by the instruction set architecture. For example, `r14` is the stack pointer (`sp`) and `r15` is the program counter (`pc`). Hence, `r15` cannot be used as a general-purpose register, and using `r14` is very difficult because it interferes with instructions that manipulate the stack. The other dedicated registers are `r12`, used as the argument pointer (`ap`), and `r13`, used as the frame pointer (`fp`); their purpose will become clear later. (Like MIPS, the VAX assembler accepts either the register number or the register name.)

VAX addressing modes include those discussed in Chapter 2, which has all the MIPS addressing modes: *register*, *displacement*, *immediate*, and *PC-relative*. Moreover, all these modes can be used for jump addresses or for data addresses.

But that's not all the addressing modes. To reduce code size, the VAX has three lengths of addresses for displacement addressing: 8-bit, 16-bit, and 32-bit addresses called, respectively, *byte displacement*, *word displacement*, and *long displacement* addressing. Thus, an address can be not only as small as possible, but also as large as necessary; large addresses need not be split, so there is no equivalent to the MIPS `lui` instruction (see page 134).

Those are still not all the VAX addressing modes. Several have a *deferred* option, meaning that the object addressed is only the *address* of the real object, requiring another memory access to get the operand. This addressing mode is called *indirect addressing* in other machines. Thus, *register deferred, autoincrement deferred,* and *byte*/*word*/*long displacement deferred* are other addressing modes to choose from. For example, using the notation of the VAX assembler, `r1` means the operand is register 1 and `(r1)` means the operand is the location in memory pointed to by `r1`.

Bits	Data type	MIPS name	VAX name
8	Integer	Byte	Byte
16	Integer	Half word	Word
32	Integer	Word	Long word
32	Floating point	Single precision	F_floating
64	Integer	Double word	Quad word
64	Floating point	Double precision	D_floating or G_floating
8n	Character string	Character	Character

Figure E.1 VAX data types, their lengths, and names. The first letter of the VAX type (b, w, l, f, q, d, g, c) is often used to complete an instruction name. Examples of move instructions include `movb`, `movw`, `movl`, `movf`, `movq`, `movd`, `movg`, and `movc3`. Each move instruction transfers an operand of the data type indicated by the letter following `mov`.

There is yet another addressing mode. *Indexed addressing* automatically converts the value in an index operand to the proper byte address to add to the rest of the address. For a 32-bit word, we needed to multiply the index of a 4-byte quantity by 4 before adding it to a base address. Indexed addressing, called *scaled addressing* on some computers, automatically multiplies the index of a 4-byte quantity by 4 as part of the address calculation.

To cope with such a plethora of addressing options, the VAX architecture separates the specification of the addressing mode from the specification of the operation. Hence, the opcode supplies the operation and the number of operands, and each operand has its own addressing mode specifier. Figure E.2 shows the name, assembler notation, example, meaning, and length of the address specifier.

The VAX style of addressing means that an operation doesn't know where its operands come from; a VAX `add` instruction can have three operands in registers, three operands in memory, or any combination of registers and memory operands.

Example How long is the following instruction?

```
addl3 r1,737(r2),(r3)[r4]
```

The name `addl3` means a 32-bit add instruction with three operands. Assume the length of the VAX opcode is 1 byte.

Answer The first operand specifier—`r1`— indicates register addressing and is 1 byte long. The second operand specifier—`737(r2)`—indicates displacement addressing and has two parts: The first part is a byte that specifies the word displacement addressing mode and base register (`r2`); the second part is the 2-byte long displacement (`737`). The third operand specifier—`(r3)[r4]`—also has two parts: The first byte specifies register deferred addressing mode (`(r3)`), and the second byte specifies the Index register and the use of indexed addressing (`[r4]`).

Thus, the total length of the instruction is 1 + (1) + (1 + 2) + (1 + 1) = 7 bytes.

In this example instruction, we show the VAX destination operand on the left and the source operands on the right, just as we show MIPS code. The VAX assembler actually expects operands in the opposite order, but we felt it would be less confusing to keep the destination on the left for both machines. Obviously, left or right orientation is arbitrary; the only requirement is consistency.

Elaboration Because the PC is one of the 16 registers that can be selected in a VAX addressing mode, 4 of the 22 VAX addressing modes are synthesized from other addressing modes. Using the PC as the chosen register in each case, *immediate* addressing is really autoincrement, *PC-relative* is displacement, *absolute* is autoincrement deferred, and *relative deferred* is displacement deferred.

Addressing mode name	Syntax	Example	Meaning	Length of address specifier in bytes
Literal	#value	#–1	–1	1 (6-bit signed value)
Immediate	#value	#100	100	1 + length of the immediate
Register	rn	r3	r3	1
Register deferred	(rn)	(r3)	Memory[r3]	1
Byte/word/long displacement	Displacement (rn)	100(r3)	Memory[r3 + 100]	1 + length of the displacement
Byte/word/long displacement deferred	@displacement (rn)	@100(r3)	Memory[Memory [r3 + 100]]	1 + length of the displacement
Indexed (scaled)	Base mode [rx]	(r3)[r4]	Memory[r3 + r4 × *d*] (where *d* is data size in bytes)	1 + length of base addressing mode
Autoincrement	(rn)+	(r3)+	Memory[r3]; r3 = r3 + *d*	1
Autodecrement	– (rn)	–(r3)	r3 = r3 – *d;* Memory[r3]	1
Autoincrement deferred	@(rn)+	@(r3)+	Memory[Memory[r3]]; r3 = r3 + *d*	1

Figure E.2 Definition and length of the VAX operand specifiers. The length of each addressing mode is 1 byte plus the length of any displacement or immediate field needed by the mode. Literal mode uses a special 2-bit tag and the remaining 6 bits encode the constant value. If the constant is too big, it must use the immediate addressing mode. Note the length of an immediate operand is dictated by the length of the data type indicated in the opcode, not the value of the immediate. The symbol *d* in the last four modes represents the length of the data in bytes; *d* is 4 for 32-bit add.

E.3 Encoding VAX Instructions

Given the independence of the operations and addressing modes, the encoding of instructions is quite different from MIPS.

VAX instructions begin with a single byte opcode containing the operation and the number of operands. The operands follow the opcode. Each operand begins with a single byte, called the *address specifier*, that describes the addressing mode for that operand. For a simple addressing mode, such as register addressing, this byte specifies the register number as well as the mode (see the rightmost column in Figure E.2). In other cases, this initial byte can be followed by many more bytes to specify the rest of the address information.

As a specific example, let's show the encoding of the add instruction from the example on page E-4:

```
addl3 r1,737(r2),(r3)[r4]
```

Assume that this instruction starts at location 201.

Figure E.3 shows the encoding. Note that the operands are stored in memory in opposite order to the assembly code above. The execution of VAX instructions

Byte address	Contents at each byte	Machine code
201	opcode containing `addl3`	$c1_{hex}$
202	index mode specifier for `[r4]`	44_{hex}
203	register indirect mode specifier for `(r3)`	63_{hex}
204	word displacement mode specifier using `r2` as base	$c2_{hex}$
205	the 16-bit constant 737	$e1_{hex}$
206		02_{hex}
207	register mode specifier for `r1`	51_{hex}

Figure E.3 The encoding of the VAX instruction addl3 r1,737(r2),(r3)[r4], assuming it starts at address 201. To satisfy your curiosity, the right column shows the actual VAX encoding in hexadecimal notation. Note that the 16-bit constant 737_{ten} takes two bytes.

begins with fetching the source operands, so it makes sense for them to come first. Order is not important in fixed-length instructions like MIPS, since the source and destination operands are easily found within a 32-bit word.

The first byte, at location 201, is the opcode. The next byte, at location 202, is a specifier for the index mode using register `r4`. Like many of the other specifiers, the left 4 bits of the specifier give the mode and the right 4 bits give the register used in that mode. Since `addl3` is a 4-byte operation, `r4` will be multiplied by 4 and added to whatever address is specified next. In this case it is register deferred addressing using register `r3`. Thus bytes 202 and 203 combined define the third operand in the assembly code.

The following byte, at address 204, is a specifier for word displacement addressing using register `r2` as the base register. This specifier tells the VAX that the following two bytes, locations 205 and 206, contain a 16-bit address to be added to `r2`.

The final byte of the instruction gives the destination operand, and this specifier selects register addressing using register `r1`.

Such variability in addressing means that a single VAX operation can have many different lengths; for example, an integer add varies from 3 bytes to 19 bytes. VAX implementations must decode the first operand before they can find the second, and so implementors are strongly tempted to take one clock cycle to decode each operand; thus this sophisticated instruction set architecture can result in higher clock cycles per instruction, even when using simple addresses.

E.4 VAX Operations

In keeping with its philosophy, the VAX has a large number of operations as well as a large number of addressing modes. We review a few here to give the flavor of the machine.

Given the power of the addressing modes, the VAX *move* instruction performs several operations found in other machines. It transfers data between any two addressable locations and subsumes load, store, register-register moves, and memory-memory moves as special cases. The first letter of the VAX data type (b, w, l, f, q, d, g, c in Figure E.1) is appended to the acronym `mov` to determine the size of the data. One special move, called *move address*, moves the 32-bit *address* of the operand rather than the data. It uses the acronym `mova`.

The arithmetic operations of MIPS are also found in the VAX, with two major differences. First, the type of the data is attached to the name. Thus `addb`, `addw`, and `addl` operate on 8-bit, 16-bit, and 32-bit data in memory or registers, respectively; MIPS has a single add instruction that operates only on the full 32-bit register. The second difference is that to reduce code size, the add instruction specifies the number of unique operands; MIPS always specifies three even if one operand is redundant. For example, the MIPS instruction

```
add         $1, $1, $2
```

takes 32 bits like all MIPS instructions, but the VAX instruction

```
addl2       r1, r2
```

uses `r1` for both the destination and a source, taking just 24 bits: 8 bits for the opcode and 8 bits each for the two register specifiers.

Number of Operations

Now we can show how VAX instruction names are formed:

$$\text{(operation)(datatype)}\binom{2}{3}$$

The operation `add` works with data types byte, word, long, float, and double and comes in versions for either 2 or 3 unique operands, so the following instructions are all found in the VAX:

```
addb2    addw2    addl2    addf2    addd2
addb3    addw3    addl3    addf3    addd3
```

Accounting for all addressing modes (but ignoring register numbers and immediate values) and limiting to just byte, word, and long, there are more than 30,000 versions of integer add in the VAX; MIPS has just 4!

Another reason for the large number of VAX instructions is the instructions that either replace sequences of instructions or take fewer bytes to represent a single instruction. Here are four such examples (* means the data type):

VAX operation	Example	Meaning
clr*	clrl r3	r3 = 0
inc*	incl r3	r3 = r3 + 1
dec*	decl r3	r3 = r3 – 1
push*	pushl r3	sp = sp – 4; Memory[sp] = r3;

The *push* instruction in the last row is exactly the same as using the move instruction with autodecrement addressing on the stack pointer:

```
movl –(sp), r3
```

Brevity is the advantage of `pushl`: It is one byte shorter since `sp` is implied.

Branches, Jumps, and Procedure Calls

The VAX branch instructions are related to the arithmetic instructions because the branch instructions rely on *condition codes*. Condition codes are set as a side effect of an operation, and they indicate whether the result is positive, negative, zero, or if an overflow occurred. Most instructions set the VAX condition codes according to their result; instructions without results, such as branches, do not. The VAX condition codes are N (Negative), Z (Zero), V (oVerflow), and C (Carry). There is also a *compare* instruction `cmp*` just to set the condition codes for a subsequent branch.

The VAX branch instructions include all conditions. Popular branch instructions include `beql`(=), `bneq`(≠), `blss`(<), `bleq`(≤), `bgtr`(>), and `bgeq`(≥), which do just what you would expect. There are also unconditional branches whose name is determined by the size of the PC-relative offset. Thus `brb` (*branch byte*) has an 8-bit displacement and `brw` (*branch word*) has a 16-bit displacement.

The final major category we cover here is the procedure *call and return* instructions. Unlike the MIPS architecture, these elaborate instructions can take dozens of clock cycles to execute. The next two sections show how they work, but we need to explain the purpose of the pointers associated with the stack manipulated by `calls` and `ret`. The *stack pointer,* `sp`, is just like the stack pointer in MIPS; it points to the top of the stack.The *argument pointer,* `ap`, points to the base of the list of arguments or parameters in memory that are passed to the procedure. The *frame pointer,* `fp`, points to the base of the local variables of the procedure that are kept in memory (the *stack frame*). The VAX call and return instructions manipulate these pointers to maintain the stack in proper condition across procedure calls and to provide convenient base registers to use when accessing memory operands. As we shall see, call and return also save and restore the general-purpose registers as well as the program counter. Figure E.4 gives a further sampling of the VAX instruction set.

Instruction type	Example	Instruction meaning
Data transfers	Move data between byte, half-word, word, or double-word operands; * is data type	
	`mov*`	Move between two operands
	`movzb*`	Move a byte to a half word or word, extending it with zeros
	`mova*`	Move the 32-bit address of an operand; data type is last
	`push*`	Push operand onto stack
Arithmetic/logical	Operations on integer or logical bytes, half words (16 bits), words (32 bits); * is data type	
	`add*_`	Add with 2 or 3 operands
	`cmp*`	Compare and set condition codes
	`tst*`	Compare to zero and set condition codes
	`ash*`	Arithmetic shift
	`clr*`	Clear
	`cvtb*`	Sign-extend byte to size of data type
Control	Conditional and unconditional branches	
	`beql, bneq`	Branch equal, branch not equal
	`bleq, bgeq`	Branch less than or equal, branch greater than or equal
	`brb, brw`	Unconditional branch with an 8-bit or 16-bit address
	`jmp`	Jump using any addressing mode to specify target
	`aobleq`	Add one to operand; branch if result ≤ second operand
	`case_`	Jump based on case selector
Procedure	Call/return from procedure	
	`calls`	Call procedure with arguments on stack (see Section E.6)
	`callg`	Call procedure with FORTRAN-style parameter list
	`jsb`	Jump to subroutine, saving return address (like MIPS `jal`)
	`ret`	Return from procedure call
Floating point	Floating-point operations on D, F, G, and H formats	
	`addd_`	Add double-precision D-format floating numbers
	`subd_`	Subtract double-precision D-format floating numbers
	`mulf_`	Multiply single-precision F-format floating point
	`polyf`	Evaluate a polynomial using table of coefficients in F format
Other	Special operations	
	`crc`	Calculate cyclic redundancy check
	`insque`	Insert a queue entry into a queue

Figure E.4 Classes of VAX instructions with examples. The asterisk stands for multiple data types: b, w, l, d, f, g, h, and q. The underline, as in `addd_`, means there are 2-operand (`addd2`) and 3-operand (`addd3`) forms of this instruction.

E.5 An Example to Put It All Together: swap

To see programming in VAX assembly language, we translate two C procedures `swap` and `sort`. The C code for `swap` is reproduced in Figure E.5. The next section covers `sort`.

We describe the `swap` procedure in three general steps of assembly language programming:

1. Allocate registers to program variables
2. Produce code for the body of the procedure
3. Preserve registers across the procedure invocation

The VAX code for these procedures is based on code produced by the VMS C compiler using optimization.

Register Allocation for swap

In contrast to MIPS, VAX parameters are normally allocated to memory, so this step of assembly language programming is more properly called "variable allocation." The standard VAX convention on parameter passing is to use the stack. The two parameters, `v[]` and `k`, can be accessed using register `ap`, the argument pointer: the address `4(ap)` corresponds to `v[]` and `8(ap)` corresponds to `k`. Remember that with byte addressing the address of sequential 4-byte words differs by 4. The only other variable is `temp`, which we associate with register `r3`.

Code for the Body of the Procedure swap

The remaining lines of C code in `swap` are

```
temp = v[k];
v[k] = v[k + 1];
v[k + 1] = temp;
```

```
swap(int v[], int k)
{
   int temp;
   temp = v[k];
   v[k] = v[k + 1];
   v[k + 1] = temp;
}
```

Figure E.5 A C procedure that swaps two locations in memory. This procedure will be used in the sorting example in the next section.

Since this program uses v[] and k several times, to make the programs run faster the VAX compiler first moves both parameters into registers:

```
movl        r2, 4(ap)            ;r2 = v[]
movl        r1, 8(ap)            ;r1 = k
```

Note that we follow the VAX convention of using a semicolon to start a comment; the MIPS comment symbol # represents a constant operand in VAX assembly language.

The VAX has indexed addressing, so we can use index k without converting it to a byte address. The VAX code is then straightforward:

```
movl        r3, (r2)[r1]         ;r3 (temp) = v[k]
addl3       r0, #1,8(ap)         ;r0 = k + 1
movl        (r2)[r1],(r2)[r0]    ;v[k] = v[r0] (v[k + 1])
movl        (r2)[r0],r3          ;v[k + 1] = r3 (temp)
```

Unlike the MIPS code, which is basically two loads and two stores, the key VAX code is one memory-to-register move, one memory-to-memory move, and one register-to-memory move. Note that the addl3 instruction shows the flexibility of the VAX addressing modes: It adds the constant 1 to a memory operand and places the result in a register.

Now we have allocated storage and written the code to perform the operations of the procedure. The only missing item is the code that preserves registers across the routine that calls swap.

Preserving Registers across Procedure Invocation of swap

The VAX has a pair of instructions that preserve registers calls and ret. This example shows how they work.

The VAX C compiler uses a form of callee convention. Examining the code above, we see that the values in registers r0, r1, r2, and r3 must be saved so that they can later be restored. The calls instruction expects a 16-bit mask at the beginning of the procedure to determine which registers are saved: if bit *i* is set in the mask, then register *i* is saved on the stack by the calls instruction. In addition, calls saves this mask on the stack to allow the return instruction (ret) to restore the proper registers. Thus the calls executed by the caller does the saving, but the callee sets the call mask to indicate what should be saved.

One of the operands for calls gives the number of parameters being passed, so that calls can adjust the pointers associated with the stack: the argument pointer (ap), frame pointer (fp), and stack pointer (sp). Of course, calls also saves the program counter so that the procedure can return!

Thus, to preserve these four registers for swap, we just add the mask at the beginning of the procedure, letting the calls instruction in the caller do all the work:

```
.word ^m<r0,r1,r2,r3>  ;set bits in mask for 0,1,2,3
```

This directive tells the assembler to place a 16-bit constant with the proper bits set to save registers r0 though r3.

The return instruction undoes the work of calls. When finished, ret sets the stack pointer from the current frame pointer to pop everything calls placed on the stack. Along the way, it restores the register values saved by calls, including those marked by the mask and old values of the fp, ap, and pc.

To complete the procedure swap, we just add one instruction:

```
ret      ;restore registers and return
```

The Full Procedure swap

We are now ready for the whole routine. Figure E.6 identifies each block of code with its purpose in the procedure, with the MIPS code on the left and the VAX code on the right. This example shows the advantage of the scaled indexed addressing and the sophisticated call and return instructions of the VAX in reducing the number of lines of code. The 17 lines of MIPS assembly code became 8 lines of VAX assembly code. It also shows that passing parameters in memory results in extra memory accesses.

Keep in mind that the number of instructions executed is not the same as performance; the fallacy on page E-18 makes this point.

MIPS versus VAX

		Saving register			
swap:	addi	$29,$29, –12	swap:	.word	^m<r0,r1,r2,r3>
	sw	$2, 0($29)			
	sw	$15, 4($29)			
	sw	$16, 8($29)			
		Procedure body			
	muli	$2, $5,4		movl	r2, 4(a)
	add	$2, $4,$2		movl	r1, 8(a)
	lw	$15, 0($2)		movl	r3, (r2)[r1]
	lw	$16, 4($2)		addl3	r0, #1,8(ap)
	sw	$16, 0($2)		movl	(r2)[r1],(r2)[r0]
	sw	$15, 4($2)		movl	(r2)[r0],r3
		Restoring registers			
	lw	$2, 0($29)			
	lw	$15, 4($29)			
	lw	$16, 8($29)			
	addi	$29,$29, 12			
		Procedure return			
	jr	$31		ret	

Figure E.6 MIPS versus VAX assembly code of the procedure swap in Figure E.5 on page E-10.

Elaboration VAX software follows a convention of treating registers `r0` and `r1` as temporaries that are not saved across a procedure call, so the VMS C compiler does include registers `r0` and `r1` in the register saving mask. Also, the C compiler should have used `r1` instead of `8(ap)` in the `addl3` instruction; such examples inspire computer architects to try to write compilers!

E.6 A Longer Example: sort

We show the longer example of the `sort` procedure. Figure E.7 shows the C version of the program. Once again we present this procedure in several steps, concluding with a side-by-side comparison to MIPS code.

Register Allocation for sort

The two parameters of the procedure `sort`, `v` and `n`, are found in the stack in locations `4(ap)` and `8(ap)`, respectively. The two local variables are assigned to registers: `i` to `r6` and `j` to `r4`. Because the two parameters are referenced frequently in the code, the VMS C compiler copies the *address* of these parameters into registers upon entering the procedure:

```
moval   r7,8(ap)    ;move address of n into r7
moval   r5,4(ap)    ;move address of v into r5
```

It would seem that moving the *value* of the operand to a register would be more useful than its address, but once again we bow to the decision of the VMS C compiler. Apparently the compiler cannot be sure that `v` and `n` don't overlap in memory.

Code for the Body of the sort Procedure

The procedure body consists of two nested *for* loops and a call to `swap`, which includes parameters. Let's unwrap the code from the outside to the middle.

```
sort (int v[], int n)
{
    int i, j;
    for (i = 0; i < n; i = i + 1) {
       for (j = i – 1; j >= 0 && v[j] > v[j + 1]; j = j – 1)
           { swap(v,j);
       }
    }
}
```

Figure E.7 A C procedure that performs a bubble sort on the array v.

The Outer Loop

The first translation step is the first for loop:

```
for (i = 0; i < n; i = i + 1) {
```

Recall that the C for statement has three parts: initialization, loop test, and iteration increment. It takes just one instruction to initialize `i` to 0, the first part of the for statement:

```
clrl        r6              ;i = 0
```

It also takes just one instruction to increment `i`, the last part of the for:

```
incl        r6              ;i = i + 1
```

The loop should be exited if `i < n` is *false,* or said another way, exit the loop if `i ≥ n`. This test takes two instructions:

```
for1tst:   cmpl   r6,(r7)  ;compare r6 and memory[r7] (i:n)
           bgeq   exit1    ;go to exit1 if r6 ≥ mem[r7] (i ≥ n)
```

Note that `cmpl` sets the condition codes for use by the conditional branch instruction `bgeq`.

The bottom of the loop just jumps back to the loop test:

```
            brb  for1tst   ;branch to test of outer loop
exit1:
```

The skeleton code of the first for loop is then

```
           clrl   r6       ;i = 0
for1tst:   cmpl   r6,(r7)  ;compare r6 and memory[r7] (i:n)
           bgeq exit1      ;go to exit1 if r6 ≥ mem[r7] (i ≥ n)
                 ...
                 (body of first for loop)
                 ...
           incl   r6       ;i = i + 1
           brb    for1tst  ;branch to test of outer loop
exit1:
```

The Inner Loop

The second for loop is

```
 for (j = i – 1; j >= 0 && v[j] > v[j + 1]; j = j – 1) {
```

The initialization portion of this loop is again one instruction:

```
subl3       r4,r6,#1 ;j = i – 1
```

The decrement of j is also one instruction:

```
decl        r4      ;j = j – 1
```

The loop test has two parts. We exit the loop if either condition fails, so the first test must exit the loop if it fails (j < 0):

```
for2tst:blss    exit2     ;go to exit2 if r4 < 0 (j < 0)
```

Notice that there is no explicit comparison. The lack of comparison is a benefit of condition codes, with the conditions being set as a side effect of the prior instruction. This branch skips over the second condition test.

The second test exits if v[j] > v[j + 1] is false, or exits if v[j] ≤ v[j + 1]. First we load v and put j + 1 into registers:

```
movl          r3,(r5)    ;r3 = Memory[r5] (r3 = v)
addl3         r2,r4,#1   ;r2 = r4 + 1 (r2 = j + 1)
```

Register indirect addressing is used to get the operand pointed to by r5.

Once again the index addressing mode means we can use indices without converting to the byte address, so the two instructions for v[j] ≤≤ v[j + 1] are

```
cmpl   (r3)[r4],(r3)[r2]  ;v[r4] : v[r2] (v[j]:v[j + 1])
bleq    exit2             ;go to exit2 if v[j] ≤ v[j + 1]
```

The bottom of the loop jumps back to the full loop test:

```
brb         for2tst       # jump to test of inner loop
```

Combining the pieces, the second for loop looks like this:

```
            subl3  r4,r6, #1      ;j = i – 1
for2tst:    blss   exit2          ;go to exit2 if r4 < 0 (j < 0)
            movl   r3,(r5)        ;r3 = Memory[r5] (r3 = v)
            addl3  r2,r4,#1       ;r2 = r4 + 1 (r2 = j + 1)
            cmpl   (r3)[r4],(r3)[r2];v[r4] : v[r2]
            bleq    exit2         ;go to exit2 if v[j]≤v[j+1]
                   ...
                   (body of second for loop)
                   ...
            decl   r4             ;j = j – 1
            brb    for2tst        ;jump to test of inner loop
exit2:
```

Notice that the instruction `blss` (at the top of the loop) is testing the condition codes based on the new value of `r4` (`j`), set either by the `subl3` before entering the loop or by the `decl` at the bottom of the loop.

The Procedure Call

The next step is the body of the second for loop:

```
swap(v,j);
```

Calling `swap` is easy enough:

```
calls       #2,swap
```

The constant 2 indicates the number of parameters pushed on the stack.

Passing Parameters

The C compiler passes variables on the stack, so we pass the parameters to `swap` with these two instructions:

```
pushl       (r5)    ;first swap parameter is v
pushl        r4     ;second swap parameter is j
```

Register indirect addressing is used to get the operand of the first instruction.

Preserving Registers across Procedure Invocation of sort

The only remaining code is the saving and restoring of registers using the callee save convention. This procedure uses registers `r2` through `r7`, so we add a mask with those bits set:

```
.word ^m<r2,r3,r4,r5,r6,r7>; set mask for registers 2-7
```

Since `ret` will undo all the operations, we just tack it on the end of the procedure.

The Full Procedure sort

Now we put all the pieces together in Figure E.8. To make the code easier to follow, once again we identify each block of code with its purpose in the procedure and list the MIPS and VAX code side by side. In this example, 11 lines of the `sort` procedure in C become the 44 lines in the MIPS assembly language and 20 lines in VAX assembly language. The biggest VAX advantages are in register saving and restoring and indexed addressing.

MIPS versus VAX					
Saving registers					
	sort:	addi	$29,$29, −36	sort:	.word ^m<r2,r3,r4,r5,r6,r7>
		sw	$15, 0($29)		
		sw	$16, 4($29)		
		sw	$17, 8($29)		
		sw	$18,12($29)		
		sw	$19,16($29)		
		sw	$20,20($29)		
		sw	$24,24($29)		
		sw	$25,28($29)		
		sw	$31,32($29)		
Procedure body					
Move parameters		move	$18, $4	moval	r7,8(ap)
		move	$20, $5	moval	r5,4(ap)
Outer loop		add	$19, $0, $0	clrl	r6
	for1tst:	slt	$8, $19, $20	for1tst: cmpl	r6,(r7)
		beq	$8, $0, exit1	bgeq	exit1
Inner loop		addi	$17, $19, −1	subl3	r4,r6,#1
	for2tst:	slti	$8, $17, 0	for2tst:	
		bne	$8, $0, exit2	blss	exit2
		muli	$15, $17, 4	movl	r3,(r5)
		add	$16, $18, $15		
		lw	$24, 0($16)		
		lw	$25, 4($16)	addl3	r2,r4,#1
		slt	$8, $25, $24	cmpl	(r3)[r4],(r3)[r2]
		beq	$8, $0, exit2	bleq	exit2
Pass parameters and call		move	$4, $18	pushl	(r5)
		move	$5, $17	pushl	r4
		jal	swap	calls	#2,swap
Inner loop		addi	$17, $17, −1	decl	r4
		j	for2tst	brb	for2tst
Outer loop	exit2:	addi	$19, $19, 1	exit2: incl	r6
		j	for1tst	brb	for1tst
Restoring registers					
	exit1:	lw	$15, 0($29)		
		lw	$16, 4($29)		
		lw	$17, 8($29)		
		lw	$18,12($29)		
		lw	$19,16($29)		
		lw	$20,20($29)		
		lw	$24,24($29)		
		lw	$25,28($29)		
		lw	$31,32($29)		
		addi	$29,$29, 36		
Procedure return					
		jr	$31	exit1: ret	

Figure E.8 MIPS32 versus VAX assembly version of procedure sort in Figure E.7 on page E-13.

E.7 Fallacies and Pitfalls

The ability to simplify means to eliminate the unnecessary so that the necessary may speak.

Hans Hoffman
Search for the Real, 1967

Fallacy *It is possible to design a flawless architecture.*

All architecture design involves trade-offs made in the context of a set of hardware and software technologies. Over time those technologies are likely to change, and decisions that may have been correct at one time later look like mistakes. For example, in 1975 the VAX designers overemphasized the importance of code size efficiency and underestimated how important ease of decoding and pipelining would be ten years later. And almost all architectures eventually succumb to the lack of sufficient address space. Avoiding these problems in the long run, however, would probably mean compromising the efficiency of the architecture in the short run.

Fallacy *An architecture with flaws cannot be successful.*

The IBM 360 is often criticized in the literature—the branches are not PC-relative, and the address is too small in displacement addressing. Yet, the machine has been an enormous success because it correctly handled several new problems. First, the architecture has a large amount of address space. Second, it is byte addressed and handles bytes well. Third, it is a general-purpose register machine. Finally, it is simple enough to be efficiently implemented across a wide performance and cost range.

The Intel 8086 provides an even more dramatic example. The 8086 architecture is the only widespread architecture in existence today that is not truly a general-purpose register machine. Furthermore, the segmented address space of the 8086 causes major problems for both programmers and compiler writers. Finally, it is hard to implement. It has generally provided only half the performance of the RISC architectures for the last eight years, despite significant investment by Intel. Nevertheless, the 8086 architecture—because of its selection as the microprocessor in the IBM PC—has been enormously successful.

Fallacy *The architecture that executes fewer instructions is faster.*

Designers of VAX machines performed a quantitative comparison of VAX and MIPS for implementations with comparable organizations, the VAX 8700 and the MIPS M2000. Figure E.9 shows the ratio of the number of instructions executed and the ratio of performance measured in clock cycles. MIPS executes about twice as many instructions as the VAX while the MIPS M2000 has almost three times the performance of the VAX 8700.

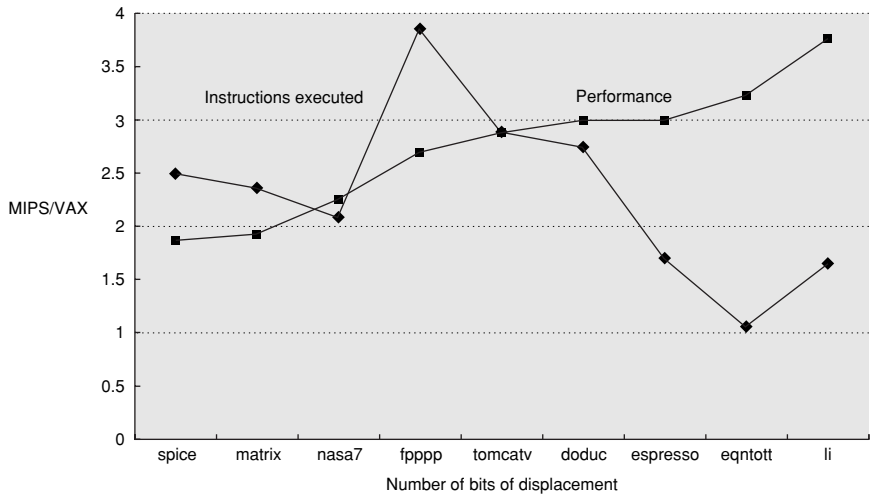


Figure E.9 Ratio of MIPS M2000 to VAX 8700 in instructions executed and performance in clock cycles using SPEC89 programs. On average, MIPS executes a little over twice as many instructions as the VAX, but the CPI for the VAX is almost six times the MIPS CPI, yielding almost a threefold performance advantage. (Based on data from "Performance from Architecture: Comparing a RISC and CISC with Similar Hardware Organization," by D. Bhandarkar and D. Clark in *Proc. Symp. Architectural Support for Programming Languages and Operating Systems IV,* 1991.)

E.8 Concluding Remarks

The Virtual Address eXtension of the PDP-11 architecture… provides a virtual address of about 4.3 gigabytes which, even given the rapid improvement of memory technology, should be adequate far into the future.

William Strecker
"VAX-11/780—A Virtual Address Extension to the PDP-11 Family," *AFIPS Proc.,* National Computer Conference, 1978

We have seen that instruction sets can vary quite dramatically, both in how they access operands and in the operations that can be performed by a single instruction. Figure E.10 compares instruction usage for both architectures for two programs; even very different architectures behave similarly in their use of instruction classes.

A product of its time, the VAX emphasis on code density and complex operations and addressing modes conflicts with the current emphasis on easy decoding, simple operations and addressing modes, and pipelined performance.

With more than 600,000 sold, the VAX architecture has had a very successful run. In 1991 DEC made the transition from VAX to Alpha, a 64-bit address architecture very similar to MIPS.

Program	Machine	Branch	Arithmetic/ logical	Data transfer	Floating point	Totals
gcc	VAX	30%	40%	19%		89%
	MIPS	24%	35%	27%		86%
spice	VAX	18%	23%	15%	23%	79%
	MIPS	4%	29%	35%	15%	83%

Figure E.10 The frequency of instruction distribution for two programs on VAX and MIPS.

Orthogonality is key to the VAX architecture; the opcode is independent of the addressing modes that are independent of the data types and even the number of unique operands.Thus a few hundred operations expand to hundreds of thousands of instructions when accounting for the data types, operand counts, and addressing modes.

E.9 Historical Perspective and Further Reading

VAX: the most successful minicomputer design in industry history… the VAX was probably the hacker's favorite machine.…Especially noted for its large, assembler-programmer-friendly instruction set—an asset that became a liability after the RISC revolution.

Eric Raymond
The New Hacker's Dictionary, 1991

In the mid-1970s, DEC realized that the PDP-11 was running out of address space. The 16-bit space had been extended in several creative ways, but the small address space was a problem that could only be postponed, not overcome.

In 1977, DEC introduced the VAX. Strecker described the architecture and called the VAX "a Virtual Address eXtension of the PDP-11." One of DEC's primary goals was to keep the installed base of PDP-11 customers. Thus, the customers were to think of the VAX as a 32-bit successor to the PDP-11. A 32-bit PDP-11 was possible—there were three designs—but Strecker reports that they were "overly compromised in terms of efficiency, functionality, programming ease." The chosen solution was to design a new architecture and include a PDP-11 compatibility mode that would run PDP-11 programs without change. This mode also allowed PDP-11 compilers to run and to continue to be used. The VAX-11/780 resembled the PDP-11 in many ways. These are among the most important:

1. Data types and formats are mostly equivalent to those on the PDP-11. The F and D floating formats came from the PDP-11. G and H formats were added

later. The use of the term "word" to describe a 16-bit quantity was carried from the PDP-11 to the VAX.

2. The assembly language was made similar to the PDP-11s.
3. The same buses were supported (Unibus and Massbus).
4. The operating system, VMS, was "an evolution" of the RSX-11M/IAS OS (as opposed to the DECsystem 10/20 OS, which was a more advanced system), and the file system was basically the same.

The VAX-11/780 was the first machine announced in the VAX series. It is one of the most successful and heavily studied machines ever built. It relied heavily on microprogramming, taking advantage of the increasing capacity of fast semiconductor memory to implement the complex instructions and addressing modes. The VAX is so tied to microcode that we predict it will be impossible to build the full VAX instruction set without microcode.

To offer a single-chip VAX in 1984, DEC reduced the instructions interpreted by microcode by trapping some instructions and performing them in software. DEC engineers found that 20% of VAX instructions are responsible for 60% of the microcode, yet are only executed 0.2% of the time. The final result was a chip offering 90% of the performance with a reduction in silicon area by more than a factor of 5.

The cornerstone of DEC's strategy was a single architecture, VAX, running a single operating system, VMS. This strategy worked well for over ten years. Today, DEC has transitioned to the Alpha RISC architecture. Like the transition from the PDP-11 to the VAX, Alpha offers the same operating system, file system, and data types and formats of the VAX. Instead of providing a VAX compatibility mode, the Alpha approach is to "compile" the VAX machine code into the Alpha machine code.

To Probe Further

Levy, H., and R. Eckhouse [1989]. *Computer Programming and Architecture: The VAX,* Digital Press, Boston.
This book concentrates on the VAX, but includes descriptions of other machines.

Exercises

E.1 [3] <E.4> The following VAX instruction decrements the location pointed to be register `r5`:

```
decl (r5)
```

What is the single MIPS instruction, or if it cannot be represented in a single instruction, the shortest sequence of MIPS instructions, that performs the same operation? What are the lengths of the instructions on each machine?

E.2 [5] <E.4> This exercise is the same as Exercise E.1, except this VAX instruction clears a location using autoincrement deferred addressing:

```
clrl @(r5)+
```

E.3 [5] <E.5> This exercise is the same as Exercise E.1, except this VAX instruction adds 1 to register r5, placing the sum back in register r5, compares the sum to register r6, and then branches to L1 if r5 < r6:

```
aoblss r6, r5,L1 # r5 = r5 + 1; if (r5 < r6) goto L1.
```

E.4 [5] <E.2> Show the single VAX instruction, or minimal sequence of instructions, for this C statement:

```
a = b + 100;
```

Assume a corresponds to register r3 and b corresponds to register r4.

E.5 [10] <E.2> Show the single VAX instruction, or minimal sequence of instructions, for this C statement:

```
x[i + 1] = x[i] + c;
```

Assume c corresponds to register r3, i to register r4, and x is an array of 32-bit words beginning at memory location $4{,}000{,}000_{ten}$.

F.1	Introduction	F-2
F.2	System/360 Instruction Set	F-3
F.3	360 Detailed Measurements	F-6
F.4	Historical Perspective and References	F-8

F

The IBM 360/370 Architecture for Mainframe Computers

We are not at all humble in this announcement. This is the most important product announcement that this corporation has ever made in its history. It's not a computer in any previous sense. It's not a product, but a line of products … that spans in performance from the very low part of the computer line to the very high.

IBM spokesman
at announcement of System/360 (1964)

© 2003 Elsevier Science (USA). All rights reserved.

F.1 Introduction

The term "computer architecture" was coined by IBM in 1964 for use with the IBM 360. Amdahl, Blaauw, and Brooks [1994] used the term to refer to the programmer-visible portion of the instruction set. They believed that a family of machines of the same architecture should be able to run the same software. Although this idea may seem obvious to us today, it was quite novel at the time. IBM, even though it was the leading company in the industry, had five different architectures before the 360. Thus, the notion of a company standardizing on a single architecture was a radical one. The 360 designers hoped that six different divisions of IBM could be brought together by defining a common architecture. Their definition of architecture was

> . . . the structure of a computer that a machine language programmer must understand to write a correct (timing independent) program for that machine.

The term "machine language programmer" meant that compatibility would hold, even in assembly language, while "timing independent" allowed different implementations.

The IBM 360 was introduced in 1964 with six models and a 25:1 performance ratio. Amdahl, Blaauw, and Brooks [1994] discuss the architecture of the IBM 360 and the concept of permitting multiple object-code–compatible implementations. The notion of an instruction set architecture as we understand it today was the most important aspect of the 360. The architecture also introduced several important innovations, now in wide use:

1. 32-bit architecture
2. Byte-addressable memory with 8-bit bytes
3. 8-, 16-, 32-, and 64-bit data sizes

In 1971, IBM shipped the first System/370 (models 155 and 165), which included a number of significant extensions of the 360, as discussed by Case and Padegs [1978], who also discuss the early history of System/360. The most important addition was virtual memory, though virtual memory 370s did not ship until 1972 when a virtual memory operating system was ready. By 1978, the high-end 370 was several hundred times faster than the low-end 360s shipped ten years earlier. In 1984, the 24-bit addressing model built into the IBM 360 needed to be abandoned, and the 370-XA (eXtended Architecture) was introduced. While old 24-bit programs could be supported without change, several instructions could not function in the same manner when extended to a 32-bit addressing model (31-bit addresses supported) because they would not produce 31-bit addresses. Converting the operating system, which was written mostly in assembly language, was no doubt the biggest task.

Several studies of the IBM 360 and instruction measurement have been made. Shustek's thesis [1978] is the best known and most complete study of the 360/370 architecture. He made several observations about instruction set complexity that were not fully appreciated until some years later. Another important study of the

360 is the Toronto study by Alexander and Wortman [1975] done on an IBM 360 using 19 XPL programs.

F.2 System/360 Instruction Set

The 360 instruction set is shown in the following tables, organized by instruction type and format. System/370 contains 15 additional user instructions.

Integer/Logical and Floating-Point R-R Instructions

The * indicates the instruction is floating point, and may be either D (double precision) or E (single precision).

Instruction	Description
ALR	Add logical register
AR	Add register
A*R	FP addition
CLR	Compare logical register
CR	Compare register
C*R	FP compare
DR	Divide register
D*R	FP divide
H*R	FP halve
LCR	Load complement register
LC*R	Load complement
LNR	Load negative register
LN*R	Load negative
LPR	Load positive register
LP*R	Load positive
LR	Load register
L*R	Load FP register
LTR	Load and test register
LT*R	Load and test FP register
MR	Multiply register
M*R	FP multiply
NR	And register
OR	Or register
SLR	Subtract logical register
SR	Subtract register
S*R	FP subtraction
XR	Exclusive or register

Branches and Status Setting R-R Instructions

These are R-R format instructions that either branch or set some system status; several of them are privileged and legal only in supervisor mode.

Instruction	Description
BALR	Branch and link
BCTR	Branch on count
BCR	Branch/condition
ISK	Insert key
SPM	Set program mask
SSK	Set storage key
SVC	Supervisor call

Branches/Logical and Floating-Point Instructions—RX Format

These are all RX format instructions. The symbol "+" means either a word operation (and then stands for nothing) or H (meaning half word); for example, A+ stands for the two opcodes A and AH. The symbol "*" is D or E standing for double- or single-precision floating point.

Instruction	Description
A+	Add
A*	FP add
AL	Add logical
C+	Compare
C*	FP compare
CL	Compare logical
D	Divide
D*	FP divide
L+	Load
L*	Load FP register
M+	Multiply
M*	FP multiply
N	And
O	Or
S+	Subtract
S*	FP subtract
SL	Subtract logical
ST+	Store
ST*	Store FP register
X	Exclusive or

Branches and Special Loads and Stores—RX format

Instruction	Description
BAL	Branch and link
BC	Branch condition
BCT	Branch on count
CVB	Convert-binary
CVD	Convert-decimal
EX	Execute
IC	Insert character
LA	Load address
STC	Store character

RS and SI Format Instructions

These are the RS and SI format instructions. The symbol "*" may be A (arithmetic) or L (logical).

Instruction	Description
BXH	Branch/high
BXLE	Branch/low-equal
CLI	Compare logical immediate
HIO	Halt I/O
LPSW	Load PSW
LM	Load multiple
MVI	Move immediate
NI	And immediate
OI	Or immediate
RDD	Read direct
SIO	Start I/O
SL*	Shift left A/L
SLD*	Shift left double A/L
SR*	Shift right A/L
SRD*	Shift right double A/L
SSM	Set system mask
STM	Store multiple
TCH	Test channel
TIO	Test I/O
TM	Test under mask
TS	Test-and-set
WRD	Write direct
XI	Exclusive or immediate

SS Format Instructions

These are add decimal or string instructions.

Instruction	Description
AP	Add packed
CLC	Compare logical chars
CP	Compare packed
DP	Divide packed
ED	Edit
EDMK	Edit and mark
MP	Multiply packed
MVC	Move character
MVN	Move numeric
MVO	Move with offset
MVZ	Move zone
NC	And characters
OC	Or characters
PACK	Pack (Character → decimal)
SP	Subtract packed
TR	Translate
TRT	Translate and test
UNPK	Unpack
XC	Exclusive or characters
ZAP	Zero and add packed

F.3 360 Detailed Measurements

Figure F.1 shows the frequency of instruction usage for four IBM 360 programs.

Instruction	PLIC	FORTGO	PLIGO	COBOLGO	Average
Control	**32%**	**13%**	**5%**	**16%**	**16%**
BC, BCR	28%	13%	5%	14%	15%
BAL, BALR	3%			2%	1%
Arithmetic/logical	**29%**	**35%**	**29%**	**9%**	**26%**
A, AR	3%	17%	21%		10%
SR	3%	7%			3%
SLL		6%	3%		2%
LA	8%	1%	1%		2%
CLI	7%				2%
NI				7%	2%
C	5%	4%	4%	0%	3%
TM	3%	1%		3%	2%
MH			2%		1%
Data transfer	**17%**	**40%**	**56%**	**20%**	**33%**
L, LR	7%	23%	28%	19%	19%
MVI	2%		16%	1%	5%
ST	3%		7%		3%
LD		7%	2%		2%
STD		7%	2%		2%
LPDR		3%			1%
LH	3%				1%
IC	2%				1%
LTR		1%			0%
Floating point		**7%**			**2%**
AD		3%			1%
MDR		3%			1%
Decimal, string	**4%**			**40%**	**11%**
MVC	4%			7%	3%
AP				11%	3%
ZAP				9%	2%
CVD				5%	1%
MP				3%	1%
CLC				3%	1%
CP				2%	1%
ED				1%	0%
Total	**82%**	**95%**	**90%**	**85%**	**88%**

Figure F.1 Distribution of instruction execution frequencies for the four 360 programs. All instructions with a frequency of execution greater than 1.5% are included. Immediate instructions, which operate on only a single byte, are included in the section that characterized their operation, rather than with the long character-string versions of the same operation. By comparison, the average frequencies for the major instruction classes of the VAX are 23% (control), 28% (arithmetic), 29% (data transfer), 7% (floating point), and 9% (decimal). Once again, a 1% entry in the average column can occur because of entries in the constituent columns. These programs are a compiler for the programming language PL-I and run time systems for the programming languages FORTRAN, PL/I, and Cobol.

F.4 Historical Perspective and References

The IBM 360 was the first computer to sell in large quantities with both byte addressing using 8-bit bytes and general-purpose registers. The 360 also had register-memory and limited memory-memory instructions. This architecture blazed the path for binary compatibility, which others have followed.

The architects of the IBM 360 were aware of the importance of address size and planned for the architecture to extend to 32 bits of address. Only 24 bits were used in the IBM 360, however, because the low-end 360 models would have been even slower with the larger addresses in 1964. Unfortunately, the architects didn't reveal their plans to the software people, and programmers who stored extra information in the upper 8 "unused" address bits foiled the expansion effort. Virtually every computer since then will check to make sure the unused bits stay unused, and will trap if the bits have the wrong value.

IBM officially extended the address to 32 bits in 1970 with the IBMs/370 architecture. Only recently did IBM expand this architecture to a flat, 64-bit address, with the IBMs/390.

References

Alexander, W. G., and D. B. Wortman [1975]. "Static and dynamic characteristics of XPL programs," *IEEE Computer* 8(11) (November), 41–46.

Amdahl, G. M., G. A. Blaauw, and F. P. Brooks, Jr. [1964]. "Architecture of the IBM System/360," *IBM J. Research and Development* 8:2 (April), 87–101.

Case, R. P., and A. Padegs [1978]. "The architecture of the IBM System/370," *Communications of the ACM,* 21:1, 73–96.

Shustek, L. J. [1978]. *Analysis and Performance of Computer Instruction Sets*. Ph.D. dissertation, Stanford University (January).

G.1	Why Vector Processors?	G-2
G.2	Basic Vector Architecture	G-4
G.3	Two Real-World Issues: Vector Length and Stride	G-16
G.4	Enhancing Vector Performance	G-23
G.5	Effectiveness of Compiler Vectorization	G-32
G.6	Putting It All Together: Performance of Vector Processors	G-34
G.7	Fallacies and Pitfalls	G-40
G.8	Concluding Remarks	G-42
G.9	Historical Perspective and References	G-43
	Exercises	G-49

G

Vector Processors

Revised by Krste Asanovic
Department of Electrical Engineering and Computer Science, MIT

I'm certainly not inventing vector processors. There are three kinds that I know of existing today. They are represented by the Illiac-IV, the (CDC) Star processor, and the TI (ASC) processor. Those three were all pioneering processors. . . . One of the problems of being a pioneer is you always make mistakes and I never, never want to be a pioneer. It's always best to come second when you can look at the mistakes the pioneers made.

Seymour Cray
Public lecture at Lawrence Livermore Laboratories on the introduction of the Cray-1 (1976)

© 2003 Elsevier Science (USA). All rights reserved.

G.1 Why Vector Processors?

In Chapters 3 and 4 we saw how we could significantly increase the performance of a processor by issuing multiple instructions per clock cycle and by more deeply pipelining the execution units to allow greater exploitation of instruction-level parallelism. (This appendix assumes that you have read Chapters 3 and 4 completely; in addition, the discussion on vector memory systems assumes that you have read Chapter 5.) Unfortunately, we also saw that there are serious difficulties in exploiting ever larger degrees of ILP.

As we increase both the width of instruction issue and the depth of the machine pipelines, we also increase the number of independent instructions required to keep the processor busy with useful work. This means an increase in the number of partially executed instructions that can be in flight at one time. For a dynamically-scheduled machine, hardware structures, such as instruction windows, reorder buffers, and rename register files, must grow to have sufficient capacity to hold all in-flight instructions, and worse, the number of ports on each element of these structures must grow with the issue width. The logic to track dependencies between all in-flight instructions grows quadratically in the number of instructions. Even a statically scheduled VLIW machine, which shifts more of the scheduling burden to the compiler, requires more registers, more ports per register, and more hazard interlock logic (assuming a design where hardware manages interlocks after issue time) to support more in-flight instructions, which similarly cause quadratic increases in circuit size and complexity. This rapid increase in circuit complexity makes it difficult to build machines that can control large numbers of in-flight instructions, and hence limits practical issue widths and pipeline depths.

Vector processors were successfully commercialized long before instruction-level parallel machines and take an alternative approach to controlling multiple functional units with deep pipelines. Vector processors provide high-level operations that work on *vectors*—linear arrays of numbers. A typical vector operation might add two 64-element, floating-point vectors to obtain a single 64-element vector result. The vector instruction is equivalent to an entire loop, with each iteration computing one of the 64 elements of the result, updating the indices, and branching back to the beginning.

Vector instructions have several important properties that solve most of the problems mentioned above:

- A single vector instruction specifies a great deal of work—it is equivalent to executing an entire loop. Each instruction represents tens or hundreds of operations, and so the instruction fetch and decode bandwidth needed to keep multiple deeply pipelined functional units busy is dramatically reduced.
- By using a vector instruction, the compiler or programmer indicates that the computation of each result in the vector is independent of the computation of other results in the same vector and so hardware does not have to check for data hazards within a vector instruction. The elements in the vector can be

computed using an array of parallel functional units, or a single very deeply pipelined functional unit, or any intermediate configuration of parallel and pipelined functional units.

- Hardware need only check for data hazards between two vector instructions once per vector operand, not once for every element within the vectors. That means the dependency checking logic required between two vector instructions is approximately the same as that required between two scalar instructions, but now many more elemental operations can be in flight for the same complexity of control logic.
- Vector instructions that access memory have a known access pattern. If the vector's elements are all adjacent, then fetching the vector from a set of heavily interleaved memory banks works very well (as we saw in Section 5.8). The high latency of initiating a main memory access versus accessing a cache is amortized, because a single access is initiated for the entire vector rather than to a single word. Thus, the cost of the latency to main memory is seen only once for the entire vector, rather than once for each word of the vector.
- Because an entire loop is replaced by a vector instruction whose behavior is predetermined, control hazards that would normally arise from the loop branch are nonexistent.

For these reasons, vector operations can be made faster than a sequence of scalar operations on the same number of data items, and designers are motivated to include vector units if the application domain can use them frequently.

As mentioned above, vector processors pipeline and parallelize the operations on the individual elements of a vector. The operations include not only the arithmetic operations (multiplication, addition, and so on), but also memory accesses and effective address calculations. In addition, most high-end vector processors allow multiple vector instructions to be in progress at the same time, creating further parallelism among the operations on different vectors.

Vector processors are particularly useful for large scientific and engineering applications, including car crash simulations and weather forecasting, for which a typical job might take dozens of hours of supercomputer time running over multigigabyte data sets. Multimedia applications can also benefit from vector processing, as they contain abundant data parallelism and process large data streams. A high-speed pipelined processor will usually use a cache to avoid forcing memory reference instructions to have very long latency. Unfortunately, big, long-running, scientific programs often have very large active data sets that are sometimes accessed with low locality, yielding poor performance from the memory hierarchy. This problem could be overcome by not caching these structures if it were possible to determine the memory access patterns and pipeline the memory accesses efficiently. Novel cache architectures and compiler assistance through blocking and prefetching are decreasing these memory hierarchy problems, but they continue to be serious in some applications.

G.2 Basic Vector Architecture

A vector processor typically consists of an ordinary pipelined scalar unit plus a vector unit. All functional units within the vector unit have a latency of several clock cycles. This allows a shorter clock cycle time and is compatible with long-running vector operations that can be deeply pipelined without generating hazards. Most vector processors allow the vectors to be dealt with as floating-point numbers, as integers, or as logical data. Here we will focus on floating point. The scalar unit is basically no different from the type of advanced pipelined CPU discussed in Chapters 3 and 4, and commercial vector machines have included both out-of-order scalar units (NEC SX/5) and VLIW scalar units (Fujitsu VPP5000).

There are two primary types of architectures for vector processors: *vector-register processors* and *memory-memory vector processors*. In a vector-register processor, all vector operations—except load and store—are among the vector registers. These architectures are the vector counterpart of a load-store architecture. All major vector computers shipped since the late 1980s use a vector-register architecture, including the Cray Research processors (Cray-1, Cray-2, X-MP, Y-MP, C90, T90, and SV1), the Japanese supercomputers (NEC SX/2 through SX/5, Fujitsu VP200 through VPP5000, and the Hitachi S820 and S-8300), and the mini-supercomputers (Convex C-1 through C-4). In a memory-memory vector processor, all vector operations are memory to memory. The first vector computers were of this type, as were CDC's vector computers. From this point on we will focus on vector-register architectures only; we will briefly return to memory-memory vector architectures at the end of the appendix (Section G.9) to discuss why they have not been as successful as vector-register architectures.

We begin with a vector-register processor consisting of the primary components shown in Figure G.1. This processor, which is loosely based on the Cray-1, is the foundation for discussion throughout most of this appendix. We will call it VMIPS; its scalar portion is MIPS, and its vector portion is the logical vector extension of MIPS. The rest of this section examines how the basic architecture of VMIPS relates to other processors.

The primary components of the instruction set architecture of VMIPS are the following:

- *Vector registers*—Each vector register is a fixed-length bank holding a single vector. VMIPS has eight vector registers, and each vector register holds 64 elements. Each vector register must have at least two read ports and one write port in VMIPS. This will allow a high degree of overlap among vector operations to different vector registers. (We do not consider the problem of a shortage of vector-register ports. In real machines this would result in a structural hazard.) The read and write ports, which total at least 16 read ports and 8 write ports, are connected to the functional unit inputs or outputs by a pair of crossbars. (The description of the vector-register file design has been simplified here. Real machines make use of the regular access pattern within a vector instruction to reduce the costs of the vector-register file circuitry [Asanovic 1998]. For example, the Cray-1 manages to implement the register file with only a single port per register.)

- *Vector functional units*—Each unit is fully pipelined and can start a new operation on every clock cycle. A control unit is needed to detect hazards, both from conflicts for the functional units (structural hazards) and from conflicts for register accesses (data hazards). VMIPS has five functional units, as shown in Figure G.1. For simplicity, we will focus exclusively on the floating-point functional units. Depending on the vector processor, scalar operations either use the vector functional units or use a dedicated set. We assume the functional units are shared, but again, for simplicity, we ignore potential conflicts.
- *Vector load-store unit*—This is a vector memory unit that loads or stores a vector to or from memory. The VMIPS vector loads and stores are fully pipelined, so that words can be moved between the vector registers and memory

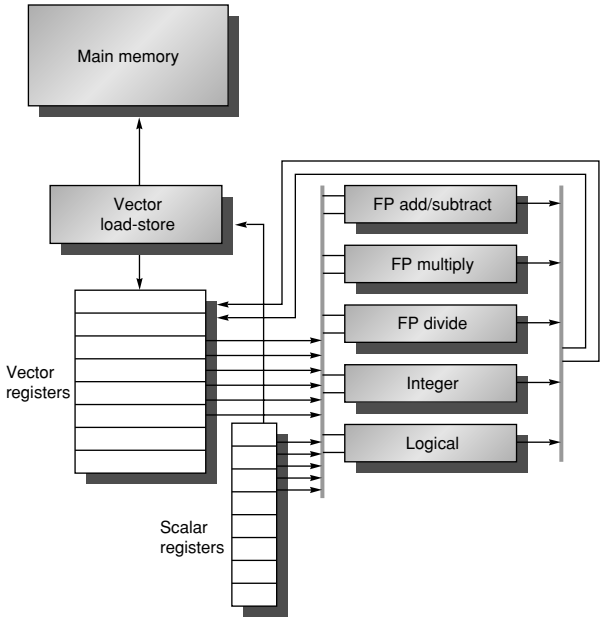


Figure G.1 The basic structure of a vector-register architecture, VMIPS. This processor has a scalar architecture just like MIPS. There are also eight 64-element vector registers, and all the functional units are vector functional units. Special vector instructions are defined both for arithmetic and for memory accesses. We show vector units for logical and integer operations. These are included so that VMIPS looks like a standard vector processor, which usually includes these units. However, we will not be discussing these units except in the exercises. The vector and scalar registers have a significant number of read and write ports to allow multiple simultaneous vector operations. These ports are connected to the inputs and outputs of the vector functional units by a set of crossbars (shown in thick gray lines). In Section G.4 we add chaining, which will require additional interconnect capability.

with a bandwidth of 1 word per clock cycle, after an initial latency. This unit would also normally handle scalar loads and stores.

- *A set of scalar registers*—Scalar registers can also provide data as input to the vector functional units, as well as compute addresses to pass to the vector load-store unit. These are the normal 32 general-purpose registers and 32 floating-point registers of MIPS. Scalar values are read out of the scalar register file, then latched at one input of the vector functional units.

Figure G.2 shows the characteristics of some typical vector processors, including the size and count of the registers, the number and types of functional units, and the number of load-store units. The last column in Figure G.2 shows the number of *lanes* in the machine, which is the number of parallel pipelines used to execute operations within each vector instruction. Lanes are described later in Section G.4; here we assume VMIPS has only a single pipeline per vector functional unit (one lane).

In VMIPS, vector operations use the same names as MIPS operations, but with the letter "V" appended. Thus, `ADDV.D` is an add of two double-precision vectors. The vector instructions take as their input either a pair of vector registers (`ADDV.D`) or a vector register and a scalar register, designated by appending "VS" (`ADDVS.D`). In the latter case, the value in the scalar register is used as the input for all operations—the operation `ADDVS.D` will add the contents of a scalar register to each element in a vector register. The scalar value is copied over to the vector functional unit at issue time. Most vector operations have a vector destination register, although a few (population count) produce a scalar value, which is stored to a scalar register. The names `LV` and `SV` denote vector load and vector store, and they load or store an entire vector of double-precision data. One operand is the vector register to be loaded or stored; the other operand, which is a MIPS general-purpose register, is the starting address of the vector in memory. Figure G.3 lists the VMIPS vector instructions. In addition to the vector registers, we need two additional special-purpose registers: the vector-length and vector-mask registers. We will discuss these registers and their purpose in Sections G.3 and G.4, respectively.

How Vector Processors Work: An Example

A vector processor is best understood by looking at a vector loop on VMIPS. Let's take a typical vector problem, which will be used throughout this appendix:

```
Y = a × X + Y
```

X and Y are vectors, initially resident in memory, and a is a scalar. This is the so-called *SAXPY* or *DAXPY* loop that forms the inner loop of the Linpack benchmark. (SAXPY stands for single-precision a × X plus Y; DAXPY for double-precision a × X plus Y.) Linpack is a collection of linear algebra routines, and the routines for performing Gaussian elimination constitute what is known as the

Processor (year)	Clock rate (MHz)	Vector registers	Elements per register (64-bit elements)	Vector arithmetic units	Vector load-store units	Lanes
Cray-1 (1976)	80	8	64	6: FP add, FP multiply, FP reciprocal, integer add, logical, shift	1	1
Cray X-MP (1983) Cray Y-MP (1988)	118 166	8	64	8: FP add, FP multiply, FP reciprocal, integer add, 2 logical, shift, population count/parity	2 loads 1 store	1
Cray-2 (1985)	244	8	64	5: FP add, FP multiply, FP reciprocal/ sqrt, integer add/shift/population count, logical	1	1
Fujitsu VP100/ VP200 (1982)	133	8–256	32–1024	3: FP or integer add/logical, multiply, divide	2	1 (VP100) 2 (VP200)
Hitachi S810/ S820 (1983)	71	32	256	4: FP multiply-add, FP multiply/ divide-add unit, 2 integer add/logical	3 loads 1 store	1 (S810) 2 (S820)
Convex C-1 (1985)	10	8	128	2: FP or integer multiply/divide, add/ logical	1	1 (64 bit) 2 (32 bit)
NEC SX/2 (1985)	167	8 + 32	256	4: FP multiply/divide, FP add, integer add/logical, shift	1	4
Cray C90 (1991) Cray T90 (1995)	240 460	8	128	8: FP add, FP multiply, FP reciprocal, integer add, 2 logical, shift, population count/parity	2 loads 1 store	2
NEC SX/5 (1998)	312	8 + 64	512	4: FP or integer add/shift, multiply, divide, logical	1	16
Fujitsu VPP5000 (1999)	300	8–256	128–4096	3: FP or integer multiply, add/logical, divide	1 load 1 store	16
Cray SV1 (1998) SV1ex (2001)	300 500	8	64	8: FP add, FP multiply, FP reciprocal, integer add, 2 logical, shift, population count/parity	1 load-store 1 load	2 8 (MSP)
VMIPS (2001)	500	8	64	5: FP multiply, FP divide, FP add, integer add/shift, logical	1 load-store	1

Figure G.2 Characteristics of several vector-register architectures. If the machine is a multiprocessor, the entries correspond to the characteristics of one processor. Several of the machines have different clock rates in the vector and scalar units; the clock rates shown are for the vector units. The Fujitsu machines' vector registers are configurable: The size and count of the 8K 64-bit entries may be varied inversely to one another (e.g., on the VP200, from eight registers each 1K elements long to 256 registers each 32 elements long). The NEC machines have eight foreground vector registers connected to the arithmetic units plus 32–64 background vector registers connected between the memory system and the foreground vector registers. The reciprocal unit on the Cray processors is used to do division (and square root on the Cray-2). Add pipelines perform add and subtract. The multiply/divide-add unit on the Hitachi S810/820 performs an FP multiply or divide followed by an add or subtract (while the multiply-add unit performs a multiply followed by an add or subtract). Note that most processors use the vector FP multiply and divide units for vector integer multiply and divide, and several of the processors use the same units for FP scalar and FP vector operations. Each vector load-store unit represents the ability to do an independent, overlapped transfer to or from the vector registers. The number of lanes is the number of parallel pipelines in each of the functional units as described in Section G.4. For example, the NEC SX/5 can complete 16 multiplies per cycle in the multiply functional unit. The Convex C-1 can split its single 64-bit lane into two 32-bit lanes to increase performance for applications that require only reduced precision. The Cray SV1 can group four CPUs with two lanes each to act in unison as a single larger CPU with eight lanes, which Cray calls a Multi-Streaming Processor (MSP).

Instruction	Operands	Function
ADDV.D	V1,V2,V3	Add elements of V2 and V3, then put each result in V1.
ADDVS.D	V1,V2,F0	Add F0 to each element of V2, then put each result in V1.
SUBV.D	V1,V2,V3	Subtract elements of V3 from V2, then put each result in V1.
SUBVS.D	V1,V2,F0	Subtract F0 from elements of V2, then put each result in V1.
SUBSV.D	V1,F0,V2	Subtract elements of V2 from F0, then put each result in V1.
MULV.D	V1,V2,V3	Multiply elements of V2 and V3, then put each result in V1.
MULVS.D	V1,V2,F0	Multiply each element of V2 by F0, then put each result in V1.
DIVV.D	V1,V2,V3	Divide elements of V2 by V3, then put each result in V1.
DIVVS.D	V1,V2,F0	Divide elements of V2 by F0, then put each result in V1.
DIVSV.D	V1,F0,V2	Divide F0 by elements of V2, then put each result in V1.
LV	V1,R1	Load vector register V1 from memory starting at address R1.
SV	R1,V1	Store vector register V1 into memory starting at address R1.
LVWS	V1,(R1,R2)	Load V1 from address at R1 with stride in R2, i.e., R1+i × R2.
SVWS	(R1,R2),V1	Store V1 from address at R1 with stride in R2, i.e., R1+i × R2.
LVI	V1,(R1+V2)	Load V1 with vector whose elements are at R1+V2(i), i.e., V2 is an index.
SVI	(R1+V2),V1	Store V1 to vector whose elements are at R1+V2(i), i.e., V2 is an index.
CVI	V1,R1	Create an index vector by storing the values 0, 1 × R1, 2 × R1,...,63 × R1 into V1.
S--V.D S--VS.D	V1,V2 V1,F0	Compare the elements (EQ, NE, GT, LT, GE, LE) in V1 and V2. If condition is true, put a 1 in the corresponding bit vector; otherwise put 0. Put resulting bit vector in vector-mask register (VM). The instruction S--VS.D performs the same compare but using a scalar value as one operand.
POP	R1,VM	Count the 1s in the vector-mask register and store count in R1.
CVM		Set the vector-mask register to all 1s.
MTC1	VLR,R1	Move contents of R1 to the vector-length register.
MFC1	R1,VLR	Move the contents of the vector-length register to R1.
MVTM	VM,F0	Move contents of F0 to the vector-mask register.
MVFM	F0,VM	Move contents of vector-mask register to F0.

Figure G.3 The VMIPS vector instructions. Only the double-precision FP operations are shown. In addition to the vector registers, there are two special registers, VLR (discussed in Section G.3) and VM (discussed in Section G.4). These special registers are assumed to live in the MIPS coprocessor 1 space along with the FPU registers. The operations with stride are explained in Section G.3, and the use of the index creation and indexed load-store operations are explained in Section G.4.

Linpack benchmark. The DAXPY routine, which implements the preceding loop, represents a small fraction of the source code of the Linpack benchmark, but it accounts for most of the execution time for the benchmark.

For now, let us assume that the number of elements, or length, of a vector register (64) matches the length of the vector operation we are interested in. (This restriction will be lifted shortly.)

Example Show the code for MIPS and VMIPS for the DAXPY loop. Assume that the starting addresses of X and Y are in Rx and Ry, respectively.

Answer Here is the MIPS code.

```
        L.D     F0,a        ;load scalar a
        DADDIU  R4,Rx,#512  ;last address to load
Loop:   L.D     F2,0(Rx)    ;load X(i)
        MUL.D   F2,F2,F0    ;a × X(i)
        L.D     F4,0(Ry)    ;load Y(i)
        ADD.D   F4,F4,F2    ;a × X(i) + Y(i)
        S.D     0(Ry),F4    ;store into Y(i)
        DADDIU  Rx,Rx,#8    ;increment index to X
        DADDIU  Ry,Ry,#8    ;increment index to Y
        DSUBU   R20,R4,Rx   ;compute bound
        BNEZ    R20,Loop    ;check if done
```

Here is the VMIPS code for DAXPY.

```
        L.D     F0,a        ;load scalar a
        LV      V1,Rx       ;load vector X
        MULVS.D V2,V1,F0    ;vector-scalar multiply
        LV      V3,Ry       ;load vector Y
        ADDV.D  V4,V2,V3    ;add
        SV      Ry,V4       ;store the result
```

There are some interesting comparisons between the two code segments in this example. The most dramatic is that the vector processor greatly reduces the dynamic instruction bandwidth, executing only six instructions versus almost 600 for MIPS. This reduction occurs both because the vector operations work on 64 elements and because the overhead instructions that constitute nearly half the loop on MIPS are not present in the VMIPS code.

Another important difference is the frequency of pipeline interlocks. In the straightforward MIPS code every `ADD.D` must wait for a `MUL.D`, and every `S.D` must wait for the `ADD.D`. On the vector processor, each vector instruction will only stall for the first element in each vector, and then subsequent elements will flow smoothly down the pipeline. Thus, pipeline stalls are required only once per vector operation, rather than once per vector element. In this example, the pipeline stall frequency on MIPS will be about 64 times higher than it is on VMIPS. The pipeline stalls can be eliminated on MIPS by using software pipelining or loop unrolling (as we saw in Chapter 4). However, the large difference in instruction bandwidth cannot be reduced.

Vector Execution Time

The execution time of a sequence of vector operations primarily depends on three factors: the length of the operand vectors, structural hazards among the operations, and the data dependences. Given the vector length and the *initiation rate,* which is the rate at which a vector unit consumes new operands and produces new results, we can compute the time for a single vector instruction. All modern supercomputers have vector functional units with multiple parallel pipelines (or *lanes*) that can produce two or more results per clock cycle, but may also have some functional units that are not fully pipelined. For simplicity, our VMIPS implementation has one lane with an initiation rate of one element per clock cycle for individual operations. Thus, the execution time for a single vector instruction is approximately the vector length.

To simplify the discussion of vector execution and its timing, we will use the notion of a *convoy,* which is the set of vector instructions that could potentially begin execution together in one clock period. (Although the concept of a convoy is used in vector compilers, no standard terminology exists. Hence, we created the term *convoy.*) The instructions in a convoy *must not* contain any structural or data hazards (though we will relax this later); if such hazards were present, the instructions in the potential convoy would need to be serialized and initiated in different convoys. Placing vector instructions into a convoy is analogous to placing scalar operations into a VLIW instruction. To keep the analysis simple, we assume that a convoy of instructions must complete execution before any other instructions (scalar or vector) can begin execution. We will relax this in Section G.4 by using a less restrictive, but more complex, method for issuing instructions.

Accompanying the notion of a convoy is a timing metric, called a *chime,* that can be used for estimating the performance of a vector sequence consisting of convoys. A chime is the unit of time taken to execute one convoy. A chime is an approximate measure of execution time for a vector sequence; a chime measurement is independent of vector length. Thus, a vector sequence that consists of m convoys executes in m chimes, and for a vector length of n, this is approximately $m \times n$ clock cycles. A chime approximation ignores some processor-specific overheads, many of which are dependent on vector length. Hence, measuring time in chimes is a better approximation for long vectors. We will use the chime measurement, rather than clock cycles per result, to explicitly indicate that certain overheads are being ignored.

If we know the number of convoys in a vector sequence, we know the execution time in chimes. One source of overhead ignored in measuring chimes is any limitation on initiating multiple vector instructions in a clock cycle. If only one vector instruction can be initiated in a clock cycle (the reality in most vector processors), the chime count will underestimate the actual execution time of a convoy. Because the vector length is typically much greater than the number of instructions in the convoy, we will simply assume that the convoy executes in one chime.

Example Show how the following code sequence lays out in convoys, assuming a single copy of each vector functional unit:

```
LV        V1,Rx       ;load vector X
MULVS.D   V2,V1,F0    ;vector-scalar multiply
LV        V3,Ry       ;load vector Y
ADDV.D    V4,V2,V3    ;add
SV        Ry,V4       ;store the result
```

How many chimes will this vector sequence take? How many cycles per FLOP (floating-point operation) are needed ignoring vector instruction issue overhead?

Answer The first convoy is occupied by the first `LV` instruction. The `MULVS.D` is dependent on the first `LV`, so it cannot be in the same convoy. The second `LV` instruction can be in the same convoy as the `MULVS.D`. The `ADDV.D` is dependent on the second `LV`, so it must come in yet a third convoy, and finally the `SV` depends on the `ADDV.D`, so it must go in a following convoy. This leads to the following layout of vector instructions into convoys:

1. `LV`
2. `MULVS.D` `LV`
3. `ADDV.D`
4. `SV`

The sequence requires four convoys and hence takes four chimes. Since the sequence takes a total of four chimes and there are two floating-point operations per result, the number of cycles per FLOP is 2 (ignoring any vector instruction issue overhead). Note that although we allow the `MULVS.D` and the `LV` both to execute in convoy 2, most vector machines will take 2 clock cycles to initiate the instructions.

The chime approximation is reasonably accurate for long vectors. For example, for 64-element vectors, the time in chimes is four, so the sequence would take about 256 clock cycles. The overhead of issuing convoy 2 in two separate clocks would be small.

Another source of overhead is far more significant than the issue limitation. The most important source of overhead ignored by the chime model is vector *start-up time*. The start-up time comes from the pipelining latency of the vector operation and is principally determined by how deep the pipeline is for the functional unit used. The start-up time increases the effective time to execute a convoy to more than one chime. Because of our assumption that convoys do not overlap in time, the start-up time delays the execution of subsequent convoys. Of course the instructions in successive convoys have either structural conflicts for some functional unit or are data dependent, so the assumption of no overlap is

reasonable. The actual time to complete a convoy is determined by the sum of the vector length and the start-up time. If vector lengths were infinite, this start-up overhead would be amortized, but finite vector lengths expose it, as the following example shows.

Example Assume the start-up overhead for functional units is shown in Figure G.4.

Show the time that each convoy can begin and the total number of cycles needed. How does the time compare to the chime approximation for a vector of length 64?

Answer Figure G.5 provides the answer in convoys, assuming that the vector length is n. One tricky question is when we assume the vector sequence is done; this determines whether the start-up time of the SV is visible or not. We assume that the instructions following cannot fit in the same convoy, and we have already assumed that convoys do not overlap. Thus the total time is given by the time until the last vector instruction in the last convoy completes. This is an approximation, and the start-up time of the last vector instruction may be seen in some sequences and not in others. For simplicity, we always include it.

The time per result for a vector of length 64 is 4 + (42/64) = 4.65 clock cycles, while the chime approximation would be 4. The execution time with start-up overhead is 1.16 times higher.

Unit	Start-up overhead (cycles)
Load and store unit	12
Multiply unit	7
Add unit	6

Figure G.4 Start-up overhead.

Convoy	Starting time	First-result time	Last-result time
1. LV	0	12	11 + n
2. MULVS.D LV	12 + n	12 + n + 12	23 + 2n
3. ADDV.D	24 + 2n	24 + 2n + 6	29 + 3n
4. SV	30 + 3n	30 + 3n + 12	41 + 4n

Figure G.5 Starting times and first- and last-result times for convoys 1 through 4. The vector length is n.

For simplicity, we will use the chime approximation for running time, incorporating start-up time effects only when we want more detailed performance or to illustrate the benefits of some enhancement. For long vectors, a typical situation, the overhead effect is not that large. Later in the appendix we will explore ways to reduce start-up overhead.

Start-up time for an instruction comes from the pipeline depth for the functional unit implementing that instruction. If the initiation rate is to be kept at 1 clock cycle per result, then

$$\text{Pipeline depth} = \left\lceil \frac{\text{Total functional unit time}}{\text{Clock cycle time}} \right\rceil$$

For example, if an operation takes 10 clock cycles, it must be pipelined 10 deep to achieve an initiation rate of one per clock cycle. Pipeline depth, then, is determined by the complexity of the operation and the clock cycle time of the processor. The pipeline depths of functional units vary widely—from 2 to 20 stages is not uncommon—although the most heavily used units have pipeline depths of 4–8 clock cycles.

For VMIPS, we will use the same pipeline depths as the Cray-1, although latencies in more modern processors have tended to increase, especially for loads. All functional units are fully pipelined. As shown in Figure G.6, pipeline depths are 6 clock cycles for floating-point add and 7 clock cycles for floating-point multiply. On VMIPS, as on most vector processors, independent vector operations using different functional units can issue in the same convoy.

Vector Load-Store Units and Vector Memory Systems

The behavior of the load-store vector unit is significantly more complicated than that of the arithmetic functional units. The start-up time for a load is the time to get the first word from memory into a register. If the rest of the vector can be supplied without stalling, then the vector initiation rate is equal to the rate at which new words are fetched or stored. Unlike simpler functional units, the initiation rate may not necessarily be 1 clock cycle because memory bank stalls can reduce effective throughput.

Operation	Start-up penalty
Vector add	6
Vector multiply	7
Vector divide	20
Vector load	12

Figure G.6 Start-up penalties on VMIPS. These are the start-up penalties in clock cycles for VMIPS vector operations.

Typically, penalties for start-ups on load-store units are higher than those for arithmetic functional units—over 100 clock cycles on some processors. For VMIPS we will assume a start-up time of 12 clock cycles, the same as the Cray-1. Figure G.6 summarizes the start-up penalties for VMIPS vector operations.

To maintain an initiation rate of 1 word fetched or stored per clock, the memory system must be capable of producing or accepting this much data. This is usually done by creating multiple memory banks, as discussed in Section 5.8. As we will see in the next section, having significant numbers of banks is useful for dealing with vector loads or stores that access rows or columns of data.

Most vector processors use memory banks rather than simple interleaving for three primary reasons:

1. Many vector computers support multiple loads or stores per clock, and the memory bank cycle time is often several times larger than the CPU cycle time. To support multiple simultaneous accesses, the memory system needs to have multiple banks and be able to control the addresses to the banks independently.
2. As we will see in the next section, many vector processors support the ability to load or store data words that are not sequential. In such cases, independent bank addressing, rather than interleaving, is required.
3. Many vector computers support multiple processors sharing the same memory system, and so each processor will be generating its own independent stream of addresses.

In combination, these features lead to a large number of independent memory banks, as shown by the following example.

Example The Cray T90 has a CPU clock cycle of 2.167 ns and in its largest configuration (Cray T932) has 32 processors each capable of generating four loads and two stores per CPU clock cycle. The CPU clock cycle is 2.167 ns, while the cycle time of the SRAMs used in the memory system is 15 ns. Calculate the minimum number of memory banks required to allow all CPUs to run at full memory bandwidth.

Answer The maximum number of memory references each cycle is 192 (32 CPUs times 6 references per CPU). Each SRAM bank is busy for 15/2.167 = 6.92 clock cycles, which we round up to 7 CPU clock cycles. Therefore we require a minimum of $192 \times 7 = 1344$ memory banks!

The Cray T932 actually has 1024 memory banks, and so the early models could not sustain full bandwidth to all CPUs simultaneously. A subsequent memory upgrade replaced the 15 ns asynchronous SRAMs with pipelined synchronous SRAMs that more than halved the memory cycle time, thereby providing sufficient bandwidth.

In Chapter 5 we saw that the desired access rate and the bank access time determined how many banks were needed to access a memory without a stall. The next example shows how these timings work out in a vector processor.

Example Suppose we want to fetch a vector of 64 elements starting at byte address 136, and a memory access takes 6 clocks. How many memory banks must we have to support one fetch per clock cycle? With what addresses are the banks accessed? When will the various elements arrive at the CPU?

Answer Six clocks per access require at least six banks, but because we want the number of banks to be a power of two, we choose to have eight banks. Figure G.7 shows the timing for the first few sets of accesses for an eight-bank system with a 6-clock-cycle access latency.

	Bank							
Cycle no.	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0		136						
1		busy	144					
2		busy	busy	152				
3		busy	busy	busy	160			
4		busy	busy	busy	busy	168		
5		busy	busy	busy	busy	busy	176	
6			busy	busy	busy	busy	busy	184
7	192			busy	busy	busy	busy	busy
8	busy	200			busy	busy	busy	busy
9	busy	busy	208			busy	busy	busy
10	busy	busy	busy	216			busy	busy
11	busy	busy	busy	busy	224			busy
12	busy	busy	busy	busy	busy	232		
13		busy	busy	busy	busy	busy	240	
14			busy	busy	busy	busy	busy	248
15	256			busy	busy	busy	busy	busy
16	busy	264			busy	busy	busy	busy

Figure G.7 Memory addresses (in bytes) by bank number and time slot at which access begins. Each memory bank latches the element address at the start of an access and is then busy for 6 clock cycles before returning a value to the CPU. Note that the CPU cannot keep all eight banks busy all the time because it is limited to supplying one new address and receiving one data item each cycle.

The timing of real memory banks is usually split into two different components, the access latency and the bank cycle time (or *bank busy time*). The access latency is the time from when the address arrives at the bank until the bank returns a data value, while the busy time is the time the bank is occupied with one request. The access latency adds to the start-up cost of fetching a vector from memory (the total memory latency also includes time to traverse the pipelined interconnection networks that transfer addresses and data between the CPU and memory banks). The bank busy time governs the effective bandwidth of a memory system because a processor cannot issue a second request to the same bank until the bank busy time has elapsed.

For simple unpipelined SRAM banks as used in the previous examples, the access latency and busy time are approximately the same. For a pipelined SRAM bank, however, the access latency is larger than the busy time because each element access only occupies one stage in the memory bank pipeline. For a DRAM bank, the access latency is usually shorter than the busy time because a DRAM needs extra time to restore the read value after the destructive read operation. For memory systems that support multiple simultaneous vector accesses or allow nonsequential accesses in vector loads or stores, the number of memory banks should be larger than the minimum; otherwise, memory bank conflicts will exist. We explore this in more detail in the next section.

G.3 Two Real-World Issues: Vector Length and Stride

This section deals with two issues that arise in real programs: What do you do when the vector length in a program is not exactly 64? How do you deal with nonadjacent elements in vectors that reside in memory? First, let's consider the issue of vector length.

Vector-Length Control

A vector-register processor has a natural vector length determined by the number of elements in each vector register. This length, which is 64 for VMIPS, is unlikely to match the real vector length in a program. Moreover, in a real program the length of a particular vector operation is often unknown at compile time. In fact, a single piece of code may require different vector lengths. For example, consider this code:

```
        do 10 i = 1,n
10      Y(i) = a * X(i) + Y(i)
```

The size of all the vector operations depends on `n`, which may not even be known until run time! The value of `n` might also be a parameter to a procedure containing the above loop and therefore be subject to change during execution.

The solution to these problems is to create a *vector-length register* (VLR). The VLR controls the length of any vector operation, including a vector load or store. The value in the VLR, however, cannot be greater than the length of the vector registers. This solves our problem as long as the real length is less than or equal to the *maximum vector length* (MVL) defined by the processor.

What if the value of `n` is not known at compile time, and thus may be greater than MVL? To tackle the second problem where the vector is longer than the maximum length, a technique called *strip mining* is used. Strip mining is the generation of code such that each vector operation is done for a size less than or equal to the MVL. We could strip-mine the loop in the same manner that we unrolled loops in Chapter 4: create one loop that handles any number of iterations that is a multiple of MVL and another loop that handles any remaining iterations, which must be less than MVL. In practice, compilers usually create a single strip-mined loop that is parameterized to handle both portions by changing the length. The strip-mined version of the DAXPY loop written in FORTRAN, the major language used for scientific applications, is shown with C-style comments:

```
        low = 1
        VL = (n mod MVL) /*find the odd-size piece*/
        do 1 j = 0,(n / MVL) /*outer loop*/
             do 10 i = low, low + VL - 1 /*runs for length VL*/
                 Y(i) = a * X(i) + Y(i) /*main operation*/
10           continue
             low = low + VL /*start of next vector*/
             VL = MVL /*reset the length to max*/
1       continue
```

The term `n/MVL` represents truncating integer division (which is what FORTRAN does) and is used throughout this section. The effect of this loop is to block the vector into segments that are then processed by the inner loop. The length of the first segment is `(n mod MVL)`, and all subsequent segments are of length `MVL`. This is depicted in Figure G.8.

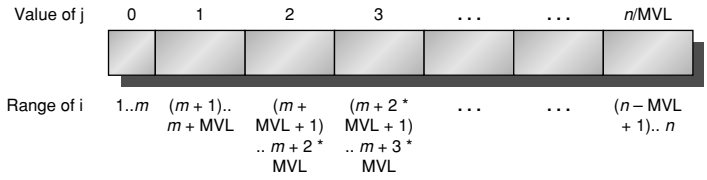


Figure G.8 A vector of arbitrary length processed with strip mining. All blocks but the first are of length MVL, utilizing the full power of the vector processor. In this figure, the variable *m* is used for the expression `(n mod MVL)`.

The inner loop of the preceding code is vectorizable with length `VL`, which is equal to either `(n mod MVL)` or MVL. The VLR register must be set twice—once at each place where the variable `VL` in the code is assigned. With multiple vector operations executing in parallel, the hardware must copy the value of VLR to the vector functional unit when a vector operation issues, in case VLR is changed for a subsequent vector operation.

Several vector ISAs have been developed that allow implementations to have different maximum vector-register lengths. For example, the IBM vector extension for the IBM 370 series mainframes supports an MVL of anywhere between 8 and 512 elements. A "load vector count and update" (VLVCU) instruction is provided to control strip-mined loops. The VLVCU instruction has a single scalar register operand that specifies the desired vector length. The vector-length register is set to the minimum of the desired length and the maximum available vector length, and this value is also subtracted from the scalar register, setting the condition codes to indicate if the loop should be terminated. In this way, object code can be moved unchanged between two different implementations while making full use of the available vector-register length within each strip-mined loop iteration.

In addition to the start-up overhead, we need to account for the overhead of executing the strip-mined loop. This strip-mining overhead, which arises from the need to reinitiate the vector sequence and set the VLR, effectively adds to the vector start-up time, assuming that a convoy does not overlap with other instructions. If that overhead for a convoy is 10 cycles, then the effective overhead per 64 elements increases by 10 cycles, or 0.15 cycles per element.

There are two key factors that contribute to the running time of a strip-mined loop consisting of a sequence of convoys:

1. The number of convoys in the loop, which determines the number of chimes. We use the notation T_{chime} for the execution time in chimes.
2. The overhead for each strip-mined sequence of convoys. This overhead consists of the cost of executing the scalar code for strip-mining each block, T_{loop}, plus the vector start-up cost for each convoy, T_{start}.

There may also be a fixed overhead associated with setting up the vector sequence the first time. In recent vector processors this overhead has become quite small, so we ignore it.

The components can be used to state the total running time for a vector sequence operating on a vector of length n, which we will call T_n:

$$T_n = \left\lceil \frac{n}{\text{MVL}} \right\rceil \times (T_{loop} + T_{start}) + n \times T_{chime}$$

The values of T_{start}, T_{loop}, and T_{chime} are compiler and processor dependent. The register allocation and scheduling of the instructions affect both what goes in a convoy and the start-up overhead of each convoy.

For simplicity, we will use a constant value for T_{loop} on VMIPS. Based on a variety of measurements of Cray-1 vector execution, the value chosen is 15 for T_{loop}. At first glance, you might think that this value is too small. The overhead in each loop requires setting up the vector starting addresses and the strides, incrementing counters, and executing a loop branch. In practice, these scalar instructions can be totally or partially overlapped with the vector instructions, minimizing the time spent on these overhead functions. The value of T_{loop} of course depends on the loop structure, but the dependence is slight compared with the connection between the vector code and the values of T_{chime} and T_{start}.

Example What is the execution time on VMIPS for the vector operation A = B × s, where s is a scalar and the length of the vectors A and B is 200?

Answer Assume the addresses of A and B are initially in `Ra` and `Rb`, s is in `Fs`, and recall that for MIPS (and VMIPS) R0 always holds 0. Since (200 mod 64) = 8, the first iteration of the strip-mined loop will execute for a vector length of 8 elements, and the following iterations will execute for a vector length of 64 elements. The starting byte addresses of the next segment of each vector is eight times the vector length. Since the vector length is either 8 or 64, we increment the address registers by 8 × 8 = 64 after the first segment and 8 × 64 = 512 for later segments. The total number of bytes in the vector is 8 × 200 = 1600, and we test for completion by comparing the address of the next vector segment to the initial address plus 1600. Here is the actual code:

```
        DADDUI   R2,R0,#1600  ;total # bytes in vector
        DADDU    R2,R2,Ra     ;address of the end of A vector
        DADDUI   R1,R0,#8     ;loads length of 1st segment
        MTC1     VLR,R1       ;load vector length in VLR
        DADDUI   R1,R0,#64    ;length in bytes of 1st segment
        DADDUI   R3,R0,#64    ;vector length of other segments
Loop:   LV       V1,Rb        ;load B
        MULVS.D  V2,V1,Fs     ;vector * scalar
        SV       Ra,V2        ;store A
        DADDU    Ra,Ra,R1     ;address of next segment of A
        DADDU    Rb,Rb,R1     ;address of next segment of B
        DADDUI   R1,R0,#512   ;load byte offset next segment
        MTC1     VLR,R3       ;set length to 64 elements
        DSUBU    R4,R2,Ra     ;at the end of A?
        BNEZ     R4,Loop      ;if not, go back
```

The three vector instructions in the loop are dependent and must go into three convoys, hence $T_{chime} = 3$. Let's use our basic formula:

$$T_n = \left\lceil \frac{n}{MVL} \right\rceil \times (T_{loop} + T_{start}) + n \times T_{chime}$$

$$T_{200} = 4 \times (15 + T_{start}) + 200 \times 3$$

$$T_{200} = 60 + (4 \times T_{start}) + 600 = 660 + (4 \times T_{start})$$

The value of T_{start} is the sum of

- The vector load start-up of 12 clock cycles
- A 7-clock-cycle start-up for the multiply
- A 12-clock-cycle start-up for the store

Thus, the value of T_{start} is given by

$$T_{start} = 12 + 7 + 12 = 31$$

So, the overall value becomes

$$T_{200} = 660 + 4 \times 31 = 784$$

The execution time per element with all start-up costs is then 784/200 = 3.9, compared with a chime approximation of three. In Section G.4, we will be more ambitious—allowing overlapping of separate convoys.

Figure G.9 shows the overhead and effective rates per element for the previous example (A = B × s) with various vector lengths. A chime counting model would lead to 3 clock cycles per element, while the two sources of overhead add 0.9 clock cycles per element in the limit.

The next few sections introduce enhancements that reduce this time. We will see how to reduce the number of convoys and hence the number of chimes using a technique called *chaining*. The loop overhead can be reduced by further overlapping the execution of vector and scalar instructions, allowing the scalar loop overhead in one iteration to be executed while the vector instructions in the previous instruction are completing. Finally, the vector start-up overhead can also be eliminated, using a technique that allows overlap of vector instructions in separate convoys.

Vector Stride

The second problem this section addresses is that the position in memory of adjacent elements in a vector may not be sequential. Consider the straightforward code for matrix multiply:

```
        do 10 i = 1,100
               do 10 j = 1,100
                      A(i,j) = 0.0
                      do 10 k = 1,100
10                    A(i,j) = A(i,j)+B(i,k)*C(k,j)
```

At the statement labeled 10 we could vectorize the multiplication of each row of `B` with each column of `C` and strip-mine the inner loop with `k` as the index variable.

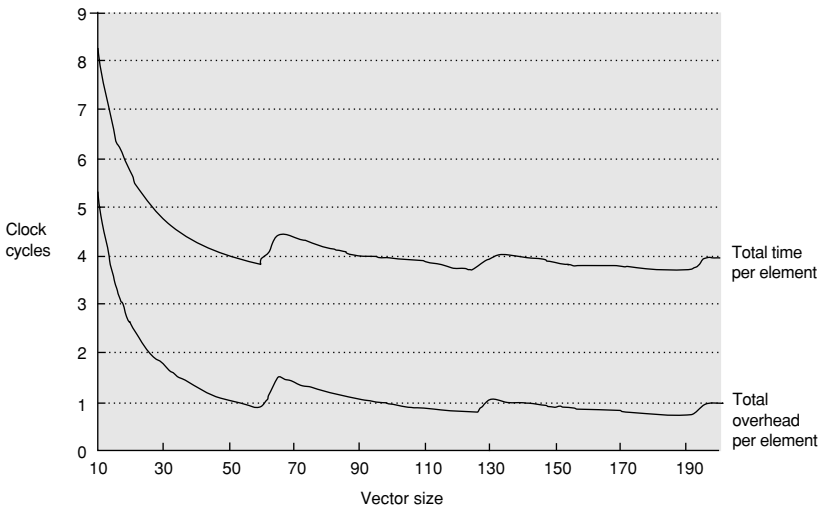


Figure G.9 The total execution time per element and the total overhead time per element versus the vector length for the example on page G-19. For short vectors the total start-up time is more than one-half of the total time, while for long vectors it reduces to about one-third of the total time. The sudden jumps occur when the vector length crosses a multiple of 64, forcing another iteration of the strip-mining code and execution of a set of vector instructions. These operations increase T_n by $T_{loop} + T_{start}$.

To do so, we must consider how adjacent elements in B and adjacent elements in C are addressed. As we discussed in Section 5.5, when an array is allocated memory, it is linearized and must be laid out in either row-major or column-major order. This linearization means that either the elements in the row or the elements in the column are not adjacent in memory. For example, if the preceding loop were written in FORTRAN, which allocates column-major order, the elements of B that are accessed by iterations in the inner loop are separated by the row size times 8 (the number of bytes per entry) for a total of 800 bytes. In Chapter 5, we saw that blocking could be used to improve the locality in cache-based systems. For vector processors without caches, we need another technique to fetch elements of a vector that are not adjacent in memory.

This distance separating elements that are to be gathered into a single register is called the *stride*. In the current example, using column-major layout for the matrices means that matrix C has a stride of 1, or 1 double word (8 bytes), separating successive elements, and matrix B has a stride of 100, or 100 double words (800 bytes).

Once a vector is loaded into a vector register it acts as if it had logically adjacent elements. Thus a vector-register processor can handle strides greater than one, called *nonunit strides,* using only vector-load and vector-store operations with stride capability. This ability to access nonsequential memory locations and

to reshape them into a dense structure is one of the major advantages of a vector processor over a cache-based processor. Caches inherently deal with unit stride data, so that while increasing block size can help reduce miss rates for large scientific data sets with unit stride, increasing block size can have a negative effect for data that is accessed with nonunit stride. While blocking techniques can solve some of these problems (see Section 5.5), the ability to efficiently access data that is not contiguous remains an advantage for vector processors on certain problems.

On VMIPS, where the addressable unit is a byte, the stride for our example would be 800. The value must be computed dynamically, since the size of the matrix may not be known at compile time, or—just like vector length—may change for different executions of the same statement. The vector stride, like the vector starting address, can be put in a general-purpose register. Then the VMIPS instruction `LVWS` (load vector with stride) can be used to fetch the vector into a vector register. Likewise, when a nonunit stride vector is being stored, `SVWS` (store vector with stride) can be used. In some vector processors the loads and stores always have a stride value stored in a register, so that only a single load and a single store instruction are required. Unit strides occur much more frequently than other strides and can benefit from special case handling in the memory system, and so are often separated from nonunit stride operations as in VMIPS.

Complications in the memory system can occur from supporting strides greater than one. In Chapter 5 we saw that memory accesses could proceed at full speed if the number of memory banks was at least as large as the bank busy time in clock cycles. Once nonunit strides are introduced, however, it becomes possible to request accesses from the same bank more frequently than the bank busy time allows. When multiple accesses contend for a bank, a memory bank conflict occurs and one access must be stalled. A bank conflict, and hence a stall, will occur if

$$\frac{\text{Number of banks}}{\text{Least common multiple (Stride, Number of banks)}} < \text{Bank busy time}$$

Example Suppose we have 8 memory banks with a bank busy time of 6 clocks and a total memory latency of 12 cycles. How long will it take to complete a 64-element vector load with a stride of 1? With a stride of 32?

Answer Since the number of banks is larger than the bank busy time, for a stride of 1, the load will take 12 + 64 = 76 clock cycles, or 1.2 clocks per element. The worst possible stride is a value that is a multiple of the number of memory banks, as in this case with a stride of 32 and 8 memory banks. Every access to memory (after the first one) will collide with the previous access and will have to wait for the 6-clock-cycle bank busy time. The total time will be 12 + 1 + 6 * 63 = 391 clock cycles, or 6.1 clocks per element.

Memory bank conflicts will not occur within a single vector memory instruction if the stride and number of banks are relatively prime with respect to each other and there are enough banks to avoid conflicts in the unit stride case. When there are no bank conflicts, multiword and unit strides run at the same rates. Increasing the number of memory banks to a number greater than the minimum to prevent stalls with a stride of length 1 will decrease the stall frequency for some other strides. For example, with 64 banks, a stride of 32 will stall on every other access, rather than every access. If we originally had a stride of 8 and 16 banks, every other access would stall; with 64 banks, a stride of 8 will stall on every eighth access. If we have multiple memory pipelines and/or multiple processors sharing the same memory system, we will also need more banks to prevent conflicts. Even machines with a single memory pipeline can experience memory bank conflicts on unit stride accesses between the last few elements of one instruction and the first few elements of the next instruction, and increasing the number of banks will reduce the probability of these interinstruction conflicts. In 2001, most vector supercomputers have at least 64 banks, and some have as many as 1024 in the maximum memory configuration. Because bank conflicts can still occur in nonunit stride cases, programmers favor unit stride accesses whenever possible.

A modern supercomputer may have dozens of CPUs, each with multiple memory pipelines connected to thousands of memory banks. It would be impractical to provide a dedicated path between each memory pipeline and each memory bank, and so typically a multistage switching network is used to connect memory pipelines to memory banks. Congestion can arise in this switching network as different vector accesses contend for the same circuit paths, causing additional stalls in the memory system.

G.4 Enhancing Vector Performance

In this section we present five techniques for improving the performance of a vector processor. The first, *chaining,* deals with making a sequence of dependent vector operations run faster, and originated in the Cray-1 but is now supported on most vector processors. The next two deal with expanding the class of loops that can be run in vector mode by combating the effects of conditional execution and sparse matrices with new types of vector instruction. The fourth technique increases the peak performance of a vector machine by adding more parallel execution units in the form of additional *lanes*. The fifth technique reduces start-up overhead by pipelining and overlapping instruction start-up.

Chaining—the Concept of Forwarding Extended to Vector Registers

Consider the simple vector sequence

```
MULV.D      V1,V2,V3
ADDV.D      V4,V1,V5
```

In VMIPS, as it currently stands, these two instructions must be put into two separate convoys, since the instructions are dependent. On the other hand, if the vector register, `V1` in this case, is treated not as a single entity but as a group of individual registers, then the ideas of forwarding can be conceptually extended to work on individual elements of a vector. This insight, which will allow the `ADDV.D` to start earlier in this example, is called *chaining*. Chaining allows a vector operation to start as soon as the individual elements of its vector source operand become available: The results from the first functional unit in the chain are "forwarded" to the second functional unit. In practice, chaining is often implemented by allowing the processor to read and write a particular register at the same time, albeit to different elements. Early implementations of chaining worked like forwarding, but this restricted the timing of the source and destination instructions in the chain. Recent implementations use *flexible chaining,* which allows a vector instruction to chain to essentially any other active vector instruction, assuming that no structural hazard is generated. Flexible chaining requires simultaneous access to the same vector register by different vector instructions, which can be implemented either by adding more read and write ports or by organizing the vector-register file storage into interleaved banks in a similar way to the memory system. We assume this type of chaining throughout the rest of this appendix.

Even though a pair of operations depend on one another, chaining allows the operations to proceed in parallel on separate elements of the vector. This permits the operations to be scheduled in the same convoy and reduces the number of chimes required. For the previous sequence, a sustained rate (ignoring start-up) of two floating-point operations per clock cycle, or one chime, can be achieved, even though the operations are dependent! The total running time for the above sequence becomes

$$\text{Vector length} + \text{Start-up time}_{\text{ADDV}} + \text{Start-up time}_{\text{MULV}}$$

Figure G.10 shows the timing of a chained and an unchained version of the above pair of vector instructions with a vector length of 64. This convoy requires one chime; however, because it uses chaining, the start-up overhead will be seen in the actual timing of the convoy. In Figure G.10, the total time for chained operation is 77 clock cycles, or 1.2 cycles per result. With 128 floating-point operations done in that time, 1.7 FLOPS per clock cycle are obtained. For the unchained version, there are 141 clock cycles, or 0.9 FLOPS per clock cycle.

Although chaining allows us to reduce the chime component of the execution time by putting two dependent instructions in the same convoy, it does not eliminate the start-up overhead. If we want an accurate running time estimate, we must count the start-up time both within and across convoys. With chaining, the number of chimes for a sequence is determined by the number of different vector functional units available in the processor and the number required by the appli-

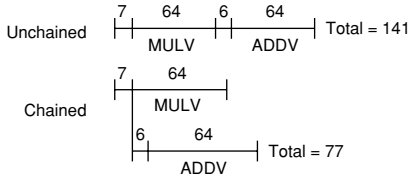


Figure G.10 Timings for a sequence of dependent vector operations ADDV and MULV, both unchained and chained. The 6- and 7-clock-cycle delays are the latency of the adder and multiplier.

cation. In particular, no convoy can contain a structural hazard. This means, for example, that a sequence containing two vector memory instructions must take at least two convoys, and hence two chimes, on a processor like VMIPS with only one vector load-store unit.

We will see in Section G.6 that chaining plays a major role in boosting vector performance. In fact, chaining is so important that every modern vector processor supports flexible chaining.

Conditionally Executed Statements

From Amdahl's Law, we know that the speedup on programs with low to moderate levels of vectorization will be very limited. Two reasons why higher levels of vectorization are not achieved are the presence of conditionals (if statements) inside loops and the use of sparse matrices. Programs that contain if statements in loops cannot be run in vector mode using the techniques we have discussed so far because the if statements introduce control dependences into a loop. Likewise, sparse matrices cannot be efficiently implemented using any of the capabilities we have seen so far. We discuss strategies for dealing with conditional execution here, leaving the discussion of sparse matrices to the following subsection.

Consider the following loop:

```
      do 100 i = 1, 64
             if (A(i).ne. 0) then
                    A(i) = A(i) – B(i)
             endif
100   continue
```

This loop cannot normally be vectorized because of the conditional execution of the body; however, if the inner loop could be run for the iterations for which $A(i) \neq 0$, then the subtraction could be vectorized. In Chapter 4, we saw that the conditionally executed instructions could turn such control dependences into data dependences, enhancing the ability to parallelize the loop. Vector processors can benefit from an equivalent capability for vectors.

The extension that is commonly used for this capability is *vector-mask control*. The vector-mask control uses a Boolean vector of length MVL to control the execution of a vector instruction just as conditionally executed instructions use a Boolean condition to determine whether an instruction is executed. When the *vector-mask register* is enabled, any vector instructions executed operate only on the vector elements whose corresponding entries in the vector-mask register are 1. The entries in the destination vector register that correspond to a 0 in the mask register are unaffected by the vector operation. If the vector-mask register is set by the result of a condition, only elements satisfying the condition will be affected. Clearing the vector-mask register sets it to all 1s, making subsequent vector instructions operate on all vector elements. The following code can now be used for the previous loop, assuming that the starting addresses of A and B are in `Ra` and `Rb`, respectively:

```
LV          V1,Ra         ;load vector A into V1
LV          V2,Rb         ;load vector B
L.D         F0,#0         ;load FP zero into F0
SNEVS.D     V1,F0         ;sets VM(i) to 1 if V1(i)!=F0
SUBV.D      V1,V1,V2      ;subtract under vector mask
CVM                       ;set the vector mask to all 1s
SV          Ra,V1         ;store the result in A
```

Most recent vector processors provide vector-mask control. The vector-mask capability described here is available on some processors, but others allow the use of the vector mask with only a subset of the vector instructions.

Using a vector-mask register does, however, have disadvantages. When we examined conditionally executed instructions, we saw that such instructions still require execution time when the condition is not satisfied. Nonetheless, the elimination of a branch and the associated control dependences can make a conditional instruction faster even if it sometimes does useless work. Similarly, vector instructions executed with a vector mask still take execution time, even for the elements where the mask is 0. Likewise, even with a significant number of 0s in the mask, using vector-mask control may still be significantly faster than using scalar mode. In fact, the large difference in potential performance between vector and scalar mode makes the inclusion of vector-mask instructions critical.

Second, in some vector processors the vector mask serves only to disable the storing of the result into the destination register, and the actual operation still occurs. Thus, if the operation in the previous example were a divide rather than a subtract and the test was on B rather than A, false floating-point exceptions might result since a division by 0 would occur. Processors that mask the operation as well as the storing of the result avoid this problem.

Sparse Matrices

There are techniques for allowing programs with sparse matrices to execute in vector mode. In a sparse matrix, the elements of a vector are usually stored in

some compacted form and then accessed indirectly. Assuming a simplified sparse structure, we might see code that looks like this:

```
        do      100 i = 1,n
100             A(K(i)) = A(K(i)) + C(M(i))
```

This code implements a sparse vector sum on the arrays A and C, using index vectors K and M to designate the nonzero elements of A and C. (A and C must have the same number of nonzero elements—n of them.) Another common representation for sparse matrices uses a bit vector to say which elements exist and a dense vector for the nonzero elements. Often both representations exist in the same program. Sparse matrices are found in many codes, and there are many ways to implement them, depending on the data structure used in the program.

A primary mechanism for supporting sparse matrices is *scatter-gather operations* using index vectors. The goal of such operations is to support moving between a dense representation (i.e., zeros are not included) and normal representation (i.e., the zeros are included) of a sparse matrix. A *gather* operation takes an *index vector* and fetches the vector whose elements are at the addresses given by adding a base address to the offsets given in the index vector. The result is a nonsparse vector in a vector register. After these elements are operated on in dense form, the sparse vector can be stored in expanded form by a *scatter* store, using the same index vector. Hardware support for such operations is called *scatter-gather* and appears on nearly all modern vector processors. The instructions LVI (load vector indexed) and SVI (store vector indexed) provide these operations in VMIPS. For example, assuming that Ra, Rc, Rk, and Rm contain the starting addresses of the vectors in the previous sequence, the inner loop of the sequence can be coded with vector instructions such as

```
LV          Vk,Rk           ;load K
LVI         Va,(Ra+Vk)      ;load A(K(I))
LV          Vm,Rm           ;load M
LVI         Vc,(Rc+Vm)      ;load C(M(I))
ADDV.D      Va,Va,Vc        ;add them
SVI         (Ra+Vk),Va      ;store A(K(I))
```

This technique allows code with sparse matrices to be run in vector mode. A simple vectorizing compiler could not automatically vectorize the source code above because the compiler would not know that the elements of K are distinct values, and thus that no dependences exist. Instead, a programmer directive would tell the compiler that it could run the loop in vector mode.

More sophisticated vectorizing compilers can vectorize the loop automatically without programmer annotations by inserting run time checks for data dependences. These run time checks are implemented with a vectorized software version of the advanced load address table (ALAT) hardware described in Chapter 4 for the Itanium processor. The associative ALAT hardware is replaced with a software hash table that detects if two element accesses within the same strip-mine iteration

are to the same address. If no dependences are detected, the strip-mine iteration can complete using the maximum vector length. If a dependence is detected, the vector length is reset to a smaller value that avoids all dependency violations, leaving the remaining elements to be handled on the next iteration of the strip-mined loop. Although this scheme adds considerable software overhead to the loop, the overhead is mostly vectorized for the common case where there are no dependences, and as a result the loop still runs considerably faster than scalar code (although much slower than if a programmer directive was provided).

A scatter-gather capability is included on many of the recent supercomputers. These operations often run more slowly than strided accesses because they are more complex to implement and are more susceptible to bank conflicts, but they are still much faster than the alternative, which may be a scalar loop. If the sparsity properties of a matrix change, a new index vector must be computed. Many processors provide support for computing the index vector quickly. The `CVI` (create vector index) instruction in VMIPS creates an index vector given a stride (m), where the values in the index vector are $0, m, 2 \times m, \ldots, 63 \times m$. Some processors provide an instruction to create a compressed index vector whose entries correspond to the positions with a 1 in the mask register. Other vector architectures provide a method to compress a vector. In VMIPS, we define the `CVI` instruction to always create a compressed index vector using the vector mask. When the vector mask is all 1s, a standard index vector will be created.

The indexed loads-stores and the `CVI` instruction provide an alternative method to support conditional vector execution. Here is a vector sequence that implements the loop we saw on page G-25:

```
LV          V1,Ra         ;load vector A into V1
L.D         F0,#0         ;load FP zero into F0
SNEVS.D     V1,F0         ;sets the VM to 1 if V1(i)!=F0
CVI         V2,#8         ;generates indices in V2
POP         R1,VM         ;find the number of 1's in VM
MTC1        VLR,R1        ;load vector-length register
CVM                       ;clears the mask
LVI         V3,(Ra+V2)    ;load the nonzero A elements
LVI         V4,(Rb+V2)    ;load corresponding B elements
SUBV.D      V3,V3,V4      ;do the subtract
SVI         (Ra+V2),V3    ;store A back
```

Whether the implementation using scatter-gather is better than the conditionally executed version depends on the frequency with which the condition holds and the cost of the operations. Ignoring chaining, the running time of the first version (on page G-25) is $5n + c_1$. The running time of the second version, using indexed loads and stores with a running time of one element per clock, is $4n + 4fn + c_2$, where f is the fraction of elements for which the condition is true (i.e., $A(i) \neq 0$). If we assume that the values of c_1 and c_2 are comparable, or that they are much smaller than n, we can find when this second technique is better.

$$\text{Time}_1 = 5(n)$$
$$\text{Time}_2 = 4n + 4fn$$

We want $\text{Time}_1 \geq \text{Time}_2$, so

$$5n \geq 4n + 4fn$$
$$\frac{1}{4} \geq f$$

That is, the second method is faster if less than one-quarter of the elements are nonzero. In many cases the frequency of execution is much lower. If the index vector can be reused, or if the number of vector statements within the if statement grows, the advantage of the scatter-gather approach will increase sharply.

Multiple Lanes

One of the greatest advantages of a vector instruction set is that it allows software to pass a large amount of parallel work to hardware using only a single short instruction. A single vector instruction can include tens to hundreds of independent operations yet be encoded in the same number of bits as a conventional scalar instruction. The parallel semantics of a vector instruction allows an implementation to execute these elemental operations using either a deeply pipelined functional unit, as in the VMIPS implementation we've studied so far, or by using an array of parallel functional units, or a combination of parallel and pipelined functional units. Figure G.11 illustrates how vector performance can be improved by using parallel pipelines to execute a vector add instruction.

The VMIPS instruction set has been designed with the property that all vector arithmetic instructions only allow element *N* of one vector register to take part in operations with element *N* from other vector registers. This dramatically simplifies the construction of a highly parallel vector unit, which can be structured as multiple parallel *lanes*. As with a traffic highway, we can increase the peak throughput of a vector unit by adding more lanes. The structure of a four-lane vector unit is shown in Figure G.12.

Each lane contains one portion of the vector-register file and one execution pipeline from each vector functional unit. Each vector functional unit executes vector instructions at the rate of one element group per cycle using multiple pipelines, one per lane. The first lane holds the first element (element 0) for all vector registers, and so the first element in any vector instruction will have its source and destination operands located in the first lane. This allows the arithmetic pipeline local to the lane to complete the operation without communicating with other lanes. Interlane wiring is only required to access main memory. This lack of interlane communication reduces the wiring cost and register file ports required to build a highly parallel execution unit, and helps explains why current vector supercomputers can complete up to 64 operations per cycle (2 arithmetic units and 2 load-store units across 16 lanes).

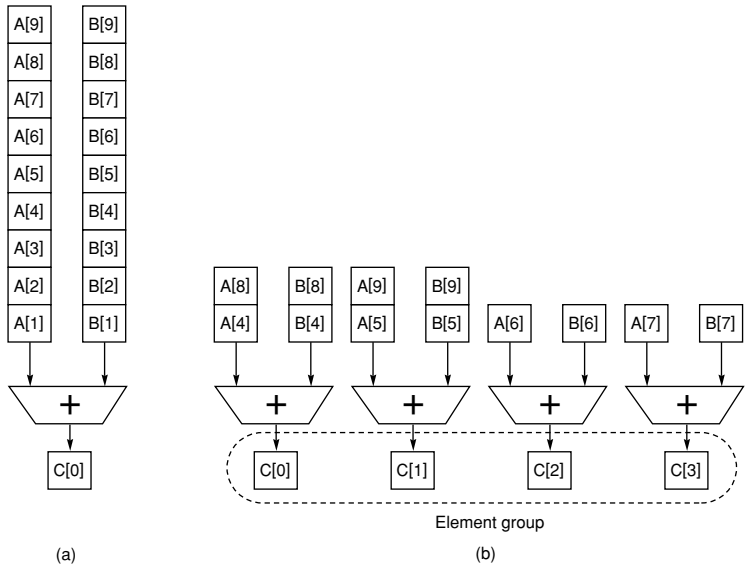


Figure G.11 Using multiple functional units to improve the performance of a single vector add instruction, C = A + B. The machine shown in (a) has a single add pipeline and can complete one addition per cycle. The machine shown in (b) has four add pipelines and can complete four additions per cycle. The elements within a single vector add instruction are interleaved across the four pipelines. The set of elements that move through the pipelines together is termed an *element group*. (Reproduced with permission from Asanovic [1998].)

Adding multiple lanes is a popular technique to improve vector performance as it requires little increase in control complexity and does not require changes to existing machine code. Several vector supercomputers are sold as a range of models that vary in the number of lanes installed, allowing users to trade price against peak vector performance. The Cray SV1 allows four two-lane CPUs to be ganged together using operating system software to form a single larger eight-lane CPU.

Pipelined Instruction Start-Up

Adding multiple lanes increases peak performance, but does not change start-up latency, and so it becomes critical to reduce start-up overhead by allowing the start of one vector instruction to be overlapped with the completion of preceding vector instructions. The simplest case to consider is when two vector instructions access a different set of vector registers. For example, in the code sequence

```
ADDV.D V1,V2,V3
ADDV.D V4,V5,V6
```

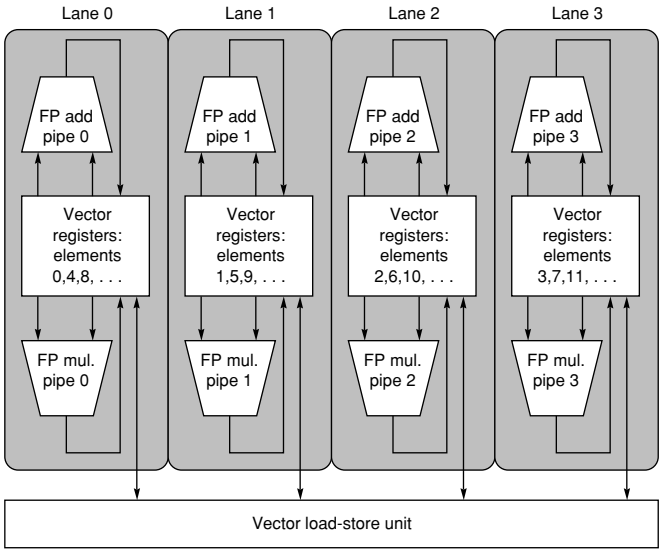


Figure G.12 Structure of a vector unit containing four lanes. The vector-register storage is divided across the lanes, with each lane holding every fourth element of each vector register. There are three vector functional units shown, an FP add, an FP multiply, and a load-store unit. Each of the vector arithmetic units contains four execution pipelines, one per lane, that act in concert to complete a single vector instruction. Note how each section of the vector-register file only needs to provide enough ports for pipelines local to its lane; this dramatically reduces the cost of providing multiple ports to the vector registers. The path to provide the scalar operand for vector-scalar instructions is not shown in this figure, but the scalar value must be broadcast to all lanes.

an implementation can allow the first element of the second vector instruction to immediately follow the last element of the first vector instruction down the FP adder pipeline. To reduce the complexity of control logic, some vector machines require some *recovery time* or *dead time* in between two vector instructions dispatched to the same vector unit. Figure G.13 is a pipeline diagram that shows both start-up latency and dead time for a single vector pipeline.

The following example illustrates the impact of this dead time on achievable vector performance.

Example The Cray C90 has two lanes but requires 4 clock cycles of dead time between any two vector instructions to the same functional unit, even if they have no data dependences. For the maximum vector length of 128 elements, what is the reduction in achievable peak performance caused by the dead time? What would be the reduction if the number of lanes were increased to 16?

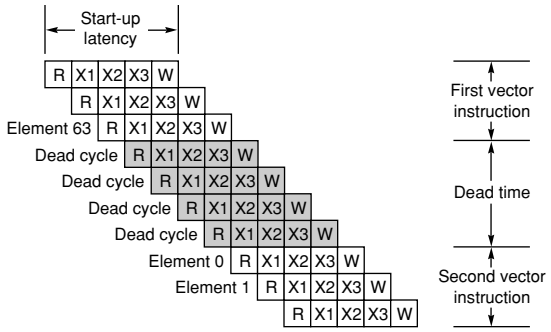


Figure G.13 Start-up latency and dead time for a single vector pipeline. Each element has a 5-cycle latency: 1 cycle to read the vector-register file, 3 cycles in execution, then 1 cycle to write the vector-register file. Elements from the same vector instruction can follow each other down the pipeline, but this machine inserts 4 cycles of dead time between two different vector instructions. The dead time can be eliminated with more complex control logic. (Reproduced with permission from Asanovic [1998].)

Answer A maximum length vector of 128 elements is divided over the two lanes and occupies a vector functional unit for 64 clock cycles. The dead time adds another 4 cycles of occupancy, reducing the peak performance to 64/(64 + 4) = 94.1% of the value without dead time. If the number of lanes is increased to 16, maximum length vector instructions will occupy a functional unit for only 128/16 = 8 cycles, and the dead time will reduce peak performance to 8/(8 + 4) = 66.6% of the value without dead time. In this second case, the vector units can never be more than 2/3 busy!

Pipelining instruction start-up becomes more complicated when multiple instructions can be reading and writing the same vector register, and when some instructions may stall unpredictably, for example, a vector load encountering memory bank conflicts. However, as both the number of lanes and pipeline latencies increase, it becomes increasingly important to allow fully pipelined instruction start-up.

G.5 Effectiveness of Compiler Vectorization

Two factors affect the success with which a program can be run in vector mode. The first factor is the structure of the program itself: Do the loops have true data dependences, or can they be restructured so as not to have such dependences? This factor is influenced by the algorithms chosen and, to some extent, by how they are coded. The second factor is the capability of the compiler. While no compiler can vectorize a loop where no parallelism among the loop iterations

exists, there is tremendous variation in the ability of compilers to determine whether a loop can be vectorized. The techniques used to vectorize programs are the same as those discussed in Chapter 4 for uncovering ILP; here we simply review how well these techniques work.

As an indication of the level of vectorization that can be achieved in scientific programs, let's look at the vectorization levels observed for the Perfect Club benchmarks. These benchmarks are large, real scientific applications. Figure G.14 shows the percentage of operations executed in vector mode for two versions of the code running on the Cray Y-MP. The first version is that obtained with just compiler optimization on the original code, while the second version has been extensively hand-optimized by a team of Cray Research programmers. The wide variation in the level of compiler vectorization has been observed by several studies of the performance of applications on vector processors.

The hand-optimized versions generally show significant gains in vectorization level for codes the compiler could not vectorize well by itself, with all codes now above 50% vectorization. It is interesting to note that for MG3D, FLO52, and DYFESM, the faster code produced by the Cray programmers had *lower* levels of vectorization. The level of vectorization is not sufficient by itself to determine performance. Alternative vectorization techniques might execute fewer

Benchmark name	Operations executed in vector mode, compiler-optimized	Operations executed in vector mode, hand-optimized	Speedup from hand optimization
BDNA	96.1%	97.2%	1.52
MG3D	95.1%	94.5%	1.00
FLO52	91.5%	88.7%	N/A
ARC3D	91.1%	92.0%	1.01
SPEC77	90.3%	90.4%	1.07
MDG	87.7%	94.2%	1.49
TRFD	69.8%	73.7%	1.67
DYFESM	68.8%	65.6%	N/A
ADM	42.9%	59.6%	3.60
OCEAN	42.8%	91.2%	3.92
TRACK	14.4%	54.6%	2.52
SPICE	11.5%	79.9%	4.06
QCD	4.2%	75.1%	2.15

Figure G.14 Level of vectorization among the Perfect Club benchmarks when executed on the Cray Y-MP [Vajapeyam 1991]. The first column shows the vectorization level obtained with the compiler, while the second column shows the results after the codes have been hand-optimized by a team of Cray Research programmers. Speedup numbers are not available for FLO52 and DYFESM as the hand-optimized runs used larger data sets than the compiler-optimized runs.

Processor	Compiler	Completely vectorized	Partially vectorized	Not vectorized
CDC CYBER 205	VAST-2 V2.21	62	5	33
Convex C-series	FC5.0	69	5	26
Cray X-MP	CFT77 V3.0	69	3	28
Cray X-MP	CFT V1.15	50	1	49
Cray-2	CFT2 V3.1a	27	1	72
ETA-10	FTN 77 V1.0	62	7	31
Hitachi S810/820	FORT77/HAP V20-2B	67	4	29
IBM 3090/VF	VS FORTRAN V2.4	52	4	44
NEC SX/2	FORTRAN77 / SX V.040	66	5	29

Figure G.15 Result of applying vectorizing compilers to the 100 FORTRAN test kernels. For each processor we indicate how many loops were completely vectorized, partially vectorized, and unvectorized. These loops were collected by Callahan, Dongarra, and Levine [1988]. Two different compilers for the Cray X-MP show the large dependence on compiler technology.

instructions, or keep more values in vector registers, or allow greater chaining and overlap among vector operations, and therefore improve performance even if the vectorization level remains the same or drops. For example, BDNA has almost the same level of vectorization in the two versions, but the hand-optimized code is over 50% faster.

There is also tremendous variation in how well different compilers do in vectorizing programs. As a summary of the state of vectorizing compilers, consider the data in Figure G.15, which shows the extent of vectorization for different processors using a test suite of 100 handwritten FORTRAN kernels. The kernels were designed to test vectorization capability and can all be vectorized by hand; we will see several examples of these loops in the exercises.

G.6 Putting It All Together: Performance of Vector Processors

In this section we look at different measures of performance for vector processors and what they tell us about the processor. To determine the performance of a processor on a vector problem we must look at the start-up cost and the sustained rate. The simplest and best way to report the performance of a vector processor on a loop is to give the execution time of the vector loop. For vector loops people often give the MFLOPS (millions of floating-point operations per second) rating rather than execution time. We use the notation R_n for the MFLOPS rating on a vector of length n. Using the measurements T_n (time) or R_n (rate) is equivalent if the number of FLOPS is agreed upon (see Chapter 1 for a longer discussion on MFLOPS). In any event, either measurement should include the overhead.

In this section we examine the performance of VMIPS on our DAXPY loop by looking at performance from different viewpoints. We will continue to compute the execution time of a vector loop using the equation developed in Section G.3. At the same time, we will look at different ways to measure performance using the computed time. The constant values for T_{loop} used in this section introduce some small amount of error, which will be ignored.

Measures of Vector Performance

Because vector length is so important in establishing the performance of a processor, length-related measures are often applied in addition to time and MFLOPS. These length-related measures tend to vary dramatically across different processors and are interesting to compare. (Remember, though, that *time* is always the measure of interest when comparing the relative speed of two processors.) Three of the most important length-related measures are

- R_∞—The MFLOPS rate on an infinite-length vector. Although this measure may be of interest when estimating peak performance, real problems do not have unlimited vector lengths, and the overhead penalties encountered in real problems will be larger.
- $N_{1/2}$—The vector length needed to reach one-half of R_∞. This is a good measure of the impact of overhead.
- N_v—The vector length needed to make vector mode faster than scalar mode. This measures both overhead and the speed of scalars relative to vectors.

Let's look at these measures for our DAXPY problem running on VMIPS. When chained, the inner loop of the DAXPY code in convoys looks like Figure G.16 (assuming that `Rx` and `Ry` hold starting addresses).

Recall our performance equation for the execution time of a vector loop with n elements, T_n:

$$T_n = \left\lceil \frac{n}{MVL} \right\rceil \times (T_{loop} + T_{start}) + n \times T_{chime}$$

Chaining allows the loop to run in three chimes (and no less, since there is one memory pipeline); thus $T_{chime} = 3$. If T_{chime} were a complete indication of performance, the loop would run at an MFLOPS rate of $2/3 \times$ clock rate (since there are 2 FLOPS per iteration). Thus, based only on the chime count, a 500 MHz VMIPS would run this loop at 333 MFLOPS assuming no strip-mining or start-up overhead. There are several ways to improve the performance: add additional vector

`LV V1,Rx`	`MULVS.D V2,V1,F0`	Convoy 1: chained load and multiply
`LV V3,Ry`	`ADDV.D V4,V2,V3`	Convoy 2: second load and add, chained
`SV Ry,V4`		Convoy 3: store the result

Figure G.16 The inner loop of the DAXPY code in chained convoys.

load-store units, allow convoys to overlap to reduce the impact of start-up overheads, and decrease the number of loads required by vector-register allocation. We will examine the first two extensions in this section. The last optimization is actually used for the Cray-1, VMIPS's cousin, to boost the performance by 50%. Reducing the number of loads requires an interprocedural optimization; we examine this transformation in Exercise G.6. Before we examine the first two extensions, let's see what the real performance, including overhead, is.

The Peak Performance of VMIPS on DAXPY

First, we should determine what the peak performance, R_∞, really is, since we know it must differ from the ideal 333 MFLOPS rate. For now, we continue to use the simplifying assumption that a convoy cannot start until all the instructions in an earlier convoy have completed; later we will remove this restriction. Using this simplification, the start-up overhead for the vector sequence is simply the sum of the start-up times of the instructions:

$$T_{start} = 12 + 7 + 12 + 6 + 12 = 49$$

Using MVL = 64, $T_{loop} = 15$, $T_{start} = 49$, and $T_{chime} = 3$ in the performance equation, and assuming that n is not an exact multiple of 64, the time for an n-element operation is

$$\begin{aligned} T_n &= \left\lceil \frac{n}{64} \right\rceil \times (15 + 49) + 3n \\ &\le (n + 64) + 3n \\ &= 4n + 64 \end{aligned}$$

The sustained rate is actually over 4 clock cycles per iteration, rather than the theoretical rate of 3 chimes, which ignores overhead. The major part of the difference is the cost of the start-up overhead for each block of 64 elements (49 cycles versus 15 for the loop overhead).

We can now compute R_∞ for a 500 MHz clock as

$$R_\infty = \lim_{n \to \infty} \left(\frac{\text{Operations per iteration} \times \text{Clock rate}}{\text{Clock cycles per iteration}} \right)$$

The numerator is independent of n, hence

$$R_\infty = \frac{\text{Operations per iteration} \times \text{Clock rate}}{\lim\limits_{n \to \infty} (\text{Clock cycles per iteration})}$$

$$\lim_{n \to \infty} (\text{Clock cycles per iteration}) = \lim_{n \to \infty} \left(\frac{T_n}{n} \right) = \lim_{n \to \infty} \left(\frac{4n + 64}{n} \right) = 4$$

$$R_\infty = \frac{2 \times 500 \text{ MHz}}{4} = 250 \text{ MFLOPS}$$

The performance without the start-up overhead, which is the peak performance given the vector functional unit structure, is now 1.33 times higher. In actuality the gap between peak and sustained performance for this benchmark is even larger!

Sustained Performance of VMIPS on the Linpack Benchmark

The Linpack benchmark is a Gaussian elimination on a 100×100 matrix. Thus, the vector element lengths range from 99 down to 1. A vector of length k is used k times. Thus, the average vector length is given by

$$\frac{\sum_{i=1}^{99} i^2}{\sum_{i=1}^{99} i} = 66.3$$

Now we can obtain an accurate estimate of the performance of DAXPY using a vector length of 66.

$$T_{66} = 2 \times (15 + 49) + 66 \times 3 = 128 + 198 = 326$$

$$R_{66} = \frac{2 \times 66 \times 500}{326} \text{ MFLOPS} = 202 \text{ MFLOPS}$$

The peak number, ignoring start-up overhead, is 1.64 times higher than this estimate of sustained performance on the real vector lengths. In actual practice, the Linpack benchmark contains a nontrivial fraction of code that cannot be vectorized. Although this code accounts for less than 20% of the time before vectorization, it runs at less than one-tenth of the performance when counted as FLOPS. Thus, Amdahl's Law tells us that the overall performance will be significantly lower than the performance estimated from analyzing the inner loop.

Since vector length has a significant impact on performance, the $N_{1/2}$ and N_v measures are often used in comparing vector machines.

Example What is $N_{1/2}$ for just the inner loop of DAXPY for VMIPS with a 500 MHz clock?

Answer Using R_∞ as the peak rate, we want to know the vector length that will achieve about 125 MFLOPS. We start with the formula for MFLOPS assuming that the measurement is made for $N_{1/2}$ elements:

$$\text{MFLOPS} = \frac{\text{FLOPS executed in } N_{1/2} \text{ iterations}}{\text{Clock cycles to execute } N_{1/2} \text{ iterations}} \times \frac{\text{Clock cycles}}{\text{Second}} \times 10^{-6}$$

$$125 = \frac{2 \times N_{1/2}}{T_{N_{1/2}}} \times 500$$

Simplifying this and then assuming $N_{1/2} \leq 64$, so that $T_{n \leq 64} = 1 \times 64 + 3 \times n$, yields

$$\begin{aligned} T_{N_{1/2}} &= 8 \times N_{1/2} \\ 1 \times 64 + 3 \times N_{1/2} &= 8 \times N_{1/2} \\ 5 \times N_{1/2} &= 64 \\ N_{1/2} &= 12.8 \end{aligned}$$

So $N_{1/2} = 13$; that is, a vector of length 13 gives approximately one-half the peak performance for the DAXPY loop on VMIPS.

Example What is the vector length, N_v, such that the vector operation runs faster than the scalar?

Answer Again, we know that $N_v < 64$. The time to do one iteration in scalar mode can be estimated as 10 + 12 + 12 + 7 + 6 +12 = 59 clocks, where 10 is the estimate of the loop overhead, known to be somewhat less than the strip-mining loop overhead. In the last problem, we showed that this vector loop runs in vector mode in time $T_{n \leq 64} = 64 + 3 \times n$ clock cycles. Therefore,

$$\begin{aligned} 64 + 3N_v &= 59N_v \\ N_v &= \left\lceil \frac{64}{56} \right\rceil \\ N_v &= 2 \end{aligned}$$

For the DAXPY loop, vector mode is faster than scalar as long as the vector has at least two elements. This number is surprisingly small, as we will see in the next section ("Fallacies and Pitfalls").

DAXPY Performance on an Enhanced VMIPS

DAXPY, like many vector problems, is memory limited. Consequently, performance could be improved by adding more memory access pipelines. This is the major architectural difference between the Cray X-MP (and later processors) and the Cray-1. The Cray X-MP has three memory pipelines, compared with the Cray-1's single memory pipeline, and the X-MP has more flexible chaining. How does this affect performance?

Example What would be the value of T_{66} for DAXPY on VMIPS if we added two more memory pipelines?

Answer With three memory pipelines all the instructions fit in one convoy and take one chime. The start-up overheads are the same, so

$$T_{66} = \left\lceil \frac{66}{64} \right\rceil \times (T_{loop} + T_{start}) + 66 \times T_{chime}$$

$$T_{66} = 2 \times (15 + 49) + 66 \times 1 = 194$$

With three memory pipelines, we have reduced the clock cycle count for sustained performance from 326 to 194, a factor of 1.7. Note the effect of Amdahl's Law: We improved the theoretical peak rate as measured by the number of chimes by a factor of 3, but only achieved an overall improvement of a factor of 1.7 in sustained performance.

Another improvement could come from allowing different convoys to overlap and also allowing the scalar loop overhead to overlap with the vector instructions. This requires that one vector operation be allowed to begin using a functional unit before another operation has completed, which complicates the instruction issue logic. Allowing this overlap eliminates the separate start-up overhead for every convoy except the first and hides the loop overhead as well.

To achieve the maximum hiding of strip-mining overhead, we need to be able to overlap strip-mined instances of the loop, allowing two instances of a convoy as well as possibly two instances of the scalar code to be in execution simultaneously. This requires the same techniques we looked at in Chapter 4 to avoid WAR hazards, although because no overlapped read and write of a single vector element is possible, copying can be avoided. This technique, called *tailgating,* was used in the Cray-2. Alternatively, we could unroll the outer loop to create several instances of the vector sequence using different register sets (assuming sufficient registers), just as we did in Chapter 4. By allowing maximum overlap of the convoys and the scalar loop overhead, the start-up and loop overheads will only be seen *once* per vector sequence, independent of the number of convoys and the instructions in each convoy. In this way a processor with vector registers can have both low start-up overhead for short vectors and high peak performance for very long vectors.

Example What would be the values of R_∞ and T_{66} for DAXPY on VMIPS if we added two more memory pipelines and allowed the strip-mining and start-up overheads to be fully overlapped?

Answer

$$R_\infty = \lim_{n \to \infty} \left(\frac{\text{Operations per iteration} \times \text{Clock rate}}{\text{Clock cycles per iteration}} \right)$$

$$\lim_{n \to \infty} (\text{Clock cycles per iteration}) = \lim_{n \to \infty} \left(\frac{T_n}{n} \right)$$

Since the overhead is only seen once, $T_n = n + 49 + 15 = n + 64$. Thus,

$$\lim_{n \to \infty}\left(\frac{T_n}{n}\right) = \lim_{n \to \infty}\left(\frac{n+64}{n}\right) = 1$$

$$R_\infty = \frac{2 \times 500 \text{ MHz}}{1} = 1000 \text{ MFLOPS}$$

Adding the extra memory pipelines and more flexible issue logic yields an improvement in peak performance of a factor of 4. However, $T_{66} = 130$, so for shorter vectors, the sustained performance improvement is about 326/130 = 2.5 times.

In summary, we have examined several measures of vector performance. Theoretical peak performance can be calculated based purely on the value of T_{chime} as

$$\frac{\text{Number of FLOPS per iteration} \times \text{Clock rate}}{T_{chime}}$$

By including the loop overhead, we can calculate values for peak performance for an infinite-length vector (R_∞) and also for sustained performance, R_n for a vector of length n, which is computed as

$$R_n = \frac{\text{Number of FLOPS per iteration} \times n \times \text{Clock rate}}{T_n}$$

Using these measures we also can find $N_{1/2}$ and N_V, which give us another way of looking at the start-up overhead for vectors and the ratio of vector to scalar speed. A wide variety of measures of performance of vector processors are useful in understanding the range of performance that applications may see on a vector processor.

G.7 Fallacies and Pitfalls

Pitfall *Concentrating on peak performance and ignoring start-up overhead.*

Early memory-memory vector processors such as the TI ASC and the CDC STAR-100 had long start-up times. For some vector problems, N_v could be greater than 100! On the CYBER 205, derived from the STAR-100, the start-up overhead for DAXPY is 158 clock cycles, substantially increasing the break-even point. With a single vector unit, which contains 2 memory pipelines, the CYBER 205 can sustain a rate of 2 clocks per iteration. The time for DAXPY for a vector of length n is therefore roughly $158 + 2n$. If the clock rates of the Cray-1 and the CYBER 205 were identical, the Cray-1 would be faster until $n > 64$. Because the Cray-1 clock is also faster (even though the 205 is newer), the crossover point is over 100. Comparing a four-lane CYBER 205 (the maximum-size processor) with the Cray X-MP that was delivered shortly after the 205, the 205 has a peak rate of two results per clock cycle—twice as fast as the X-MP. However, vectors must be longer than about 200 for the CYBER 205 to be faster.

The problem of start-up overhead has been a major difficulty for the memory-memory vector architectures, hence their lack of popularity.

Pitfall *Increasing vector performance, without comparable increases in scalar performance.*

This was a problem on many early vector processors, and a place where Seymour Cray rewrote the rules. Many of the early vector processors had comparatively slow scalar units (as well as large start-up overheads). Even today, processors with higher peak vector performance can be outperformed by a processor with lower vector performance but better scalar performance. Good scalar performance keeps down overhead costs (strip mining, for example) and reduces the impact of Amdahl's Law. A good example of this comes from comparing a fast scalar processor and a vector processor with lower scalar performance. The Livermore FORTRAN kernels are a collection of 24 scientific kernels with varying degrees of vectorization. Figure G.17 shows the performance of two different processors on this benchmark. Despite the vector processor's higher peak performance, its low scalar performance makes it slower than a fast scalar processor as measured by the harmonic mean. The next fallacy is closely related.

Fallacy *You can get vector performance without providing memory bandwidth.*

As we saw with the DAXPY loop, memory bandwidth is quite important. DAXPY requires 1.5 memory references per floating-point operation, and this ratio is typical of many scientific codes. Even if the floating-point operations took no time, a Cray-1 could not increase the performance of the vector sequence used, since it is memory limited. The Cray-1 performance on Linpack jumped when the compiler used clever transformations to change the computation so that values could be kept in the vector registers. This lowered the number of memory references per FLOP and improved the performance by nearly a factor of 2! Thus, the memory bandwidth on

Processor	Minimum rate for any loop (MFLOPS)	Maximum rate for any loop (MFLOPS)	Harmonic mean of all 24 loops (MFLOPS)
MIPS M/120-5	0.80	3.89	1.85
Stardent-1500	0.41	10.08	1.72

Figure G.17 Performance measurements for the Livermore FORTRAN kernels on two different processors. Both the MIPS M/120-5 and the Stardent-1500 (formerly the Ardent Titan-1) use a 16.7 MHz MIPS R2000 chip for the main CPU. The Stardent-1500 uses its vector unit for scalar FP and has about half the scalar performance (as measured by the minimum rate) of the MIPS M/120, which uses the MIPS R2010 FP chip. The vector processor is more than a factor of 2.5 times faster for a highly vectorizable loop (maximum rate). However, the lower scalar performance of the Stardent-1500 negates the higher vector performance when total performance is measured by the harmonic mean on all 24 loops.

the Cray-1 became sufficient for a loop that formerly required more bandwidth. This ability to reuse values from vector registers is another advantage of vector-register architectures compared with memory-memory vector architectures, which have to fetch all vector operands from memory, requiring even greater memory bandwidth.

G.8 Concluding Remarks

During the 1980s and 1990s, rapid performance increases in pipelined scalar processors led to a dramatic closing of the gap between traditional vector supercomputers and fast, pipelined, superscalar VLSI microprocessors. In 2002, it is possible to buy a complete desktop computer system for under $1000 that has a higher CPU clock rate than any available vector supercomputer, even those costing tens of millions of dollars. Although the vector supercomputers have lower clock rates, they support greater parallelism through the use of multiple lanes (up to 16 in the Japanese designs) versus the limited multiple issue of the superscalar microprocessors. Nevertheless, the peak floating-point performance of the low-cost microprocessors is within a factor of 4 of the leading vector supercomputer CPUs. Of course, high clock rates and high peak performance do not necessarily translate into sustained application performance. Main memory bandwidth is the key distinguishing feature between vector supercomputers and superscalar microprocessor systems. The fastest microprocessors in 2002 can sustain around 1 GB/sec of main memory bandwidth, while the fastest vector supercomputers can sustain around 50 GB/sec per CPU. For nonunit stride accesses the bandwidth discrepancy is even greater. For certain scientific and engineering applications, performance correlates directly with nonunit stride main memory bandwidth, and these are the applications for which vector supercomputers remain popular.

Providing this large nonunit stride memory bandwidth is one of the major expenses in a vector supercomputer, and traditionally SRAM was used as main memory to reduce the number of memory banks needed and to reduce vector start-up penalties. While SRAM has an access time several times lower than that of DRAM, it costs roughly 10 times as much per bit! To reduce main memory costs and to allow larger capacities, all modern vector supercomputers now use DRAM for main memory, taking advantage of new higher-bandwidth DRAM interfaces such as synchronous DRAM.

This adoption of DRAM for main memory (pioneered by Seymour Cray in the Cray-2) is one example of how vector supercomputers are adapting commodity technology to improve their price-performance. Another example is that vector supercomputers are now including vector data caches. Caches are not effective for all vector codes, however, and so these vector caches are designed to allow high main memory bandwidth even in the presence of many cache misses. For example, the cache on the Cray SV1 can support 384 outstanding cache misses per CPU, while for microprocessors 8–16 outstanding misses is a more typical maximum number.

Another example is the demise of bipolar ECL or gallium arsenide as technologies of choice for supercomputer CPU logic. Because of the huge investment in CMOS technology made possible by the success of the desktop computer, CMOS now offers competitive transistor performance with much greater transistor density and much reduced power dissipation compared with these more exotic technologies. As a result, all leading vector supercomputers are now built with the same CMOS technology as superscalar microprocessors. The primary reason that vector supercomputers now have lower clock rates than commodity microprocessors is that they are developed using standard cell ASIC techniques rather than full custom circuit design to reduce the engineering design cost. While a microprocessor design may sell tens of millions of copies and can amortize the design cost over this large number of units, a vector supercomputer is considered a success if over a hundred units are sold!

Conversely, superscalar microprocessor designs have begun to absorb some of the techniques made popular in earlier vector computer systems. Many multimedia applications contain code that can be vectorized, and as discussed in Chapter 2, most commercial microprocessor ISAs have added multimedia extensions that resemble short vector instructions. A common technique is to allow a wide 64-bit register to be split into smaller subwords that are operated on in parallel. This idea was used in the early TI ASC and CDC STAR-100 vector machines, where a 64-bit lane could be split into two 32-bit lanes to give higher performance on lower-precision data. Although the initial microprocessor multimedia extensions were very limited in scope, newer extensions such as AltiVec for the IBM/Motorola PowerPC and SSE2 for the Intel x86 processors have both increased the vector length to 128 bits (still small compared with the 4096 bits in a VMIPS vector register) and added better support for vector compilers. Vector instructions are particularly appealing for embedded processors because they support high degrees of parallelism at low cost and with low power dissipation, and have been used in several game machines such as the Nintendo-64 and the Sony Playstation 2 to boost graphics performance. We expect that microprocessors will continue to extend their support for vector operations, as this represents a much simpler approach to boosting performance for an important class of applications compared with the hardware complexity of increasing scalar instruction issue width, or the software complexity of managing multiple parallel processors.

G.9 Historical Perspective and References

The first vector processors were the CDC STAR-100 (see Hintz and Tate [1972]) and the TI ASC (see Watson [1972]), both announced in 1972. Both were memory-memory vector processors. They had relatively slow scalar units—the STAR used the same units for scalars and vectors—making the scalar pipeline extremely deep. Both processors had high start-up overhead and worked on vectors of several hundred to several thousand elements. The crossover between scalar and vector could be over 50 elements. It appears that not enough attention was paid to the role of Amdahl's Law on these two processors.

Cray, who worked on the 6600 and the 7600 at CDC, founded Cray Research and introduced the Cray-1 in 1976 (see Russell [1978]). The Cray-1 used a vector-register architecture to significantly lower start-up overhead and to reduce memory bandwidth requirements. He also had efficient support for nonunit stride and invented chaining. Most importantly, the Cray-1 was the fastest scalar processor in the world at that time. This matching of good scalar and vector performance was probably the most significant factor in making the Cray-1 a success. Some customers bought the processor primarily for its outstanding scalar performance. Many subsequent vector processors are based on the architecture of this first commercially successful vector processor. Baskett and Keller [1977] provide a good evaluation of the Cray-1.

In 1981, CDC started shipping the CYBER 205 (see Lincoln [1982]). The 205 had the same basic architecture as the STAR, but offered improved performance all around as well as expandability of the vector unit with up to four lanes, each with multiple functional units and a wide load-store pipe that provided multiple words per clock. The peak performance of the CYBER 205 greatly exceeded the performance of the Cray-1. However, on real programs, the performance difference was much smaller.

The CDC STAR processor and its descendant, the CYBER 205, were memory-memory vector processors. To keep the hardware simple and support the high bandwidth requirements (up to three memory references per floating-point operation), these processors did not efficiently handle nonunit stride. While most loops have unit stride, a nonunit stride loop had poor performance on these processors because memory-to-memory data movements were required to gather together (and scatter back) the nonadjacent vector elements; these operations used special scatter-gather instructions. In addition, there was special support for sparse vectors that used a bit vector to represent the zeros and nonzeros and a dense vector of nonzero values. These more complex vector operations were slow because of the long memory latency, and it was often faster to use scalar mode for sparse or nonunit stride operations. Schneck [1987] described several of the early pipelined processors (e.g., Stretch) through the first vector processors, including the 205 and Cray-1. Dongarra [1986] did another good survey, focusing on more recent processors.

In 1983, Cray Research shipped the first Cray X-MP (see Chen [1983]). With an improved clock rate (9.5 ns versus 12.5 ns on the Cray-1), better chaining support, and multiple memory pipelines, this processor maintained the Cray Research lead in supercomputers. The Cray-2, a completely new design configurable with up to four processors, was introduced later. A major feature of the Cray-2 was the use of DRAM, which made it possible to have very large memories. The first Cray-2 with its 256M word (64-bit words) memory contained more memory than the total of all the Cray machines shipped to that point! The Cray-2 had a much faster clock than the X-MP, but also much deeper pipelines; however, it lacked chaining, had an enormous memory latency, and had only one memory pipe per processor. In general, the Cray-2 is only faster than the Cray X-MP on problems that require its very large main memory.

The 1980s also saw the arrival of smaller-scale vector processors, called mini-supercomputers. Priced at roughly one-tenth the cost of a supercomputer ($0.5 to $1 million versus $5 to $10 million), these processors caught on quickly. Although many companies joined the market, the two companies that were most successful were Convex and Alliant. Convex started with the uniprocessor C-1 vector processor and then offered a series of small multiprocessors ending with the C-4 announced in 1994. The keys to the success of Convex over this period were their emphasis on Cray software capability, the effectiveness of their compiler (see Figure G.15), and the quality of their UNIX OS implementation. The C-4 was the last vector machine Convex sold; they switched to making large-scale multiprocessors using Hewlett-Packard RISC microprocessors and were bought by HP in 1995. Alliant [1987] concentrated more on the multiprocessor aspects; they built an eight-processor computer, with each processor offering vector capability. Alliant ceased operation in the early 1990s.

In the early 1980s, CDC spun out a group, called ETA, to build a new supercomputer, the ETA-10, capable of 10 GFLOPS. The ETA processor was delivered in the late 1980s (see Fazio [1987]) and used low-temperature CMOS in a configuration with up to 10 processors. Each processor retained the memory-memory architecture based on the CYBER 205. Although the ETA-10 achieved enormous peak performance, its scalar speed was not comparable. In 1989 CDC, the first supercomputer vendor, closed ETA and left the supercomputer design business.

In 1986, IBM introduced the System/370 vector architecture (see Moore et al. [1987]) and its first implementation in the 3090 Vector Facility. The architecture extends the System/370 architecture with 171 vector instructions. The 3090/VF is integrated into the 3090 CPU. Unlike most other vector processors, the 3090/VF routes its vectors through the cache.

In 1983, processor vendors from Japan entered the supercomputer marketplace, starting with the Fujitsu VP100 and VP200 (see Miura and Uchida [1983]), and later expanding to include the Hitachi S810 and the NEC SX/2 (see Watanabe [1987]). These processors have proved to be close to the Cray X-MP in performance. In general, these three processors have much higher peak performance than the Cray X-MP. However, because of large start-up overhead, their typical performance is often lower than the Cray X-MP (see Figure 1.32 in Chapter 1). The Cray X-MP favored a multiple-processor approach, first offering a two-processor version and later a four-processor. In contrast, the three Japanese processors had expandable vector capabilities.

In 1988, Cray Research introduced the Cray Y-MP—a bigger and faster version of the X-MP. The Y-MP allows up to eight processors and lowers the cycle time to 6 ns. With a full complement of eight processors, the Y-MP was generally the fastest supercomputer, though the single-processor Japanese supercomputers may be faster than a one-processor Y-MP. In late 1989 Cray Research was split into two companies, both aimed at building high-end processors available in the early 1990s. Seymour Cray headed the spin-off, Cray Computer Corporation,

until its demise in 1995. Their initial processor, the Cray-3, was to be implemented in gallium arsenide, but they were unable to develop a reliable and cost-effective implementation technology. A single Cray-3 prototype was delivered to the National Center for Atmospheric Research (NCAR) for evaluation purposes in 1993, but no paying customers were found for the design. The Cray-4 prototype, which was to have been the first processor to run at 1 GHz, was close to completion when the company filed for bankruptcy. Shortly before his tragic death in a car accident in 1996, Seymour Cray started yet another company, SRC Computers, to develop high-performance systems but this time using commodity components. In 2000, SRC announced the SRC-6 system that combines 512 Intel microprocessors, 5 billion gates of reconfigurable logic, and a high-performance vector-style memory system.

Cray Research focused on the C90, a new high-end processor with up to 16 processors and a clock rate of 240 MHz. This processor was delivered in 1991. Typical configurations are about $15 million. In 1993, Cray Research introduced their first highly parallel processor, the T3D, employing up to 2048 Digital Alpha 21064 microprocessors. In 1995, they announced the availability of both a new low-end vector machine, the J90, and a high-end machine, the T90. The T90 is much like the C90, but offers a clock that is twice as fast (460 MHz), using three-dimensional packaging and optical clock distribution. Like the C90, the T90 costs in the tens of millions, though a single CPU is available for $2.5 million. The T90 was the last bipolar ECL vector machine built by Cray. The J90 is a CMOS-based vector machine using DRAM memory starting at $250,000, but with typical configurations running about $1 million. In mid-1995, Cray Research was acquired by Silicon Graphics, and in 1998 released the SV1 system, which grafted considerably faster CMOS processors onto the J90 memory system, and which also added a data cache for vectors to each CPU to help meet the increased memory bandwidth demands. Silicon Graphics sold Cray Research to Tera Computer in 2000, and the joint company was renamed Cray Inc. Cray Inc. plans to release the SV2 in 2002, which will be based on a completely new vector ISA.

The Japanese supercomputer makers have continued to evolve their designs and have generally placed greater emphasis on increasing the number of lanes in their vector units. In 2001, the NEC SX/5 was generally held to be the fastest available vector supercomputer, with 16 lanes clocking at 312 MHz and with up to 16 processors sharing the same memory. The Fujitsu VPP5000 was announced in 2001 and also had 16 lanes and clocked at 300 MHz, but connected up to 128 processors in a distributed-memory cluster. In 2001, Cray Inc. announced that they would be marketing the NEC SX/5 machine in the United States, after many years in which Japanese supercomputers were unavailable to U.S. customers after the U.S. Commerce Department found NEC and Fujitsu guilty of bidding below cost for a 1996 NCAR supercomputer contract and imposed heavy import duties on their products.

The basis for modern vectorizing compiler technology and the notion of data dependence was developed by Kuck and his colleagues [1974] at the University of Illinois. Banerjee [1979] developed the test named after him. Padua and Wolfe [1986] give a good overview of vectorizing compiler technology.

Benchmark studies of various supercomputers, including attempts to understand the performance differences, have been undertaken by Lubeck, Moore, and Mendez [1985], Bucher [1983], and Jordan [1987]. In Chapter 1, we discussed several benchmark suites aimed at scientific usage and often employed for supercomputer benchmarking, including Linpack and the Lawrence Livermore Laboratories FORTRAN kernels. The University of Illinois coordinated the collection of a set of benchmarks for supercomputers, called the Perfect Club. In 1993, the Perfect Club was integrated into SPEC, which released a set of benchmarks, SPEChpc96, aimed at high-end scientific processing in 1996. The NAS parallel benchmarks developed at the NASA Ames Research Center [Bailey et al. 1991] have become a popular set of kernels and applications used for supercomputer evaluation.

In less than 30 years vector processors have gone from unproven, new architectures to playing a significant role in the goal to provide engineers and scientists with ever larger amounts of computing power. However, the enormous price-performance advantages of microprocessor technology are bringing this era to an end. Advanced superscalar microprocessors are approaching the peak performance of the fastest vector processors, and in 2001, most of the highest-performance machines in the world were large-scale multiprocessors based on these microprocessors. Vector supercomputers remain popular for certain applications including car crash simulation and weather prediction that rely heavily on scatter-gather performance over large data sets and for which effective massively parallel programs have yet to be written. Over time, we expect that microprocessors will support higher-bandwidth memory systems, and that more applications will be parallelized and/or tuned for cached multiprocessor systems. As the set of applications best suited for vector supercomputers shrinks, they will become less viable as commercial products and will eventually disappear. But vector processing techniques will likely survive as an integral part of future microprocessor architectures, with the currently popular SIMD multimedia extensions representing the first step in this direction.

References

Alliant Computer Systems Corp. [1987]. *Alliant FX/Series: Product Summary* (June), Acton, Mass.

Asanovic, K. [1998]. "Vector microprocessors," Ph.D. thesis, Computer Science Division, Univ. of California at Berkeley (May).

Bailey, D. H., E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga [1991]. "The NAS parallel benchmarks," *Int'l. J. Supercomputing Applications* 5, 63–73.

Banerjee, U. [1979]. "Speedup of ordinary programs," Ph.D. thesis, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign (October).

Baskett, F., and T. W. Keller [1977]. "An Evaluation of the Cray-1 Processor," in *High Speed Computer and Algorithm Organization,* D. J. Kuck, D. H. Lawrie, and A. H. Sameh, eds., Academic Press, San Diego, 71–84.

Brandt, M., J. Brooks, M. Cahir, T. Hewitt, E. Lopez-Pineda, and D. Sandness [2000]. *The Benchmarker's Guide for Cray SV1 Systems*. Cray Inc., Seattle, Wash.

Bucher, I. Y. [1983]. "The computational speed of supercomputers," *Proc. SIGMETRICS Conf. on Measuring and Modeling of Computer Systems,* ACM (August), 151–165.

Callahan, D., J. Dongarra, and D. Levine [1988]. "Vectorizing compilers: A test suite and results," *Supercomputing '88,* ACM/IEEE (November), Orlando, Fla., 98–105.

Chen, S. [1983]. "Large-scale and high-speed multiprocessor system for scientific applications," *Proc. NATO Advanced Research Work on High Speed Computing* (June); also in K. Hwang, ed., "Superprocessors: Design and applications," *IEEE* (August), 1984.

Dongarra, J. J. [1986]. "A survey of high performance processors," *COMPCON, IEEE* (March), 8–11.

Fazio, D. [1987]. "It's really much more fun building a supercomputer than it is simply inventing one," *COMPCON, IEEE* (February), 102–105.

Flynn, M. J. [1966]. "Very high-speed computing systems," *Proc. IEEE* 54:12 (December), 1901–1909.

Hintz, R. G., and D. P. Tate [1972]. "Control data STAR-100 processor design," *COMPCON, IEEE* (September), 1–4.

Jordan, K. E. [1987]. "Performance comparison of large-scale scientific processors: Scalar mainframes, mainframes with vector facilities, and supercomputers," *Computer* 20:3 (March), 10–23.

Kuck, D., P. P. Budnik, S.-C. Chen, D. H. Lawrie, R. A. Towle, R. E. Strebendt, E. W. Davis, Jr., J. Han, P. W. Kraska, and Y. Muraoka [1974]. "Measurements of parallelism in ordinary FORTRAN programs," *Computer* 7:1 (January), 37–46.

Lincoln, N. R. [1982]. "Technology and design trade offs in the creation of a modern supercomputer," *IEEE Trans. on Computers* C-31:5 (May), 363–376.

Lubeck, O., J. Moore, and R. Mendez [1985]. "A benchmark comparison of three supercomputers: Fujitsu VP-200, Hitachi S810/20, and Cray X-MP/2," *Computer* 18:1 (January), 10–29.

Miranker, G. S., J. Rubenstein, and J. Sanguinetti [1988]. "Squeezing a Cray-class supercomputer into a single-user package," *COMPCON, IEEE* (March), 452–456.

Miura, K., and K. Uchida [1983]. "FACOM vector processing system: VP100/200," *Proc. NATO Advanced Research Work on High Speed Computing* (June); also in K. Hwang, ed., "Superprocessors: Design and applications," *IEEE* (August 1984), 59–73.

Moore, B., A. Padegs, R. Smith, and W. Bucholz [1987]. "Concepts of the System/370 vector architecture," *Proc. 14th Symposium on Computer Architecture* (June), ACM/IEEE, Pittsburgh, 282–292.

Padua, D., and M. Wolfe [1986]. "Advanced compiler optimizations for supercomputers," *Comm. ACM* 29:12 (December), 1184–1201.

Russell, R. M. [1978]. "The Cray-1 processor system," *Comm. of the ACM* 21:1 (January), 63–72.

Schneck, P. B. [1987]. *Superprocessor Architecture,* Kluwer Academic Publishers, Norwell, Mass.

Smith, B. J. [1981]. "Architecture and applications of the HEP multiprocessor system," *Real-Time Signal Processing IV* 298 (August), 241–248.

Sporer, M., F. H. Moss, and C. J. Mathais [1988]. "An introduction to the architecture of the Stellar Graphics supercomputer," *COMPCON, IEEE* (March), 464.

Vajapeyam, S. [1991]. "Instruction-level characterization of the Cray Y-MP processor," Ph.D. thesis, Computer Sciences Department, University of Wisconsin-Madison.

Watanabe, T. [1987]. "Architecture and performance of the NEC supercomputer SX system," *Parallel Computing* 5, 247–255.

Watson, W. J. [1972]. "The TI ASC—a highly modular and flexible super processor architecture," *Proc. AFIPS Fall Joint Computer Conf.,* 221–228.

Exercises

In these exercises assume VMIPS has a clock rate of 500 MHz and that T_{loop} = 15. Use the start-up times from Figure G.4, and assume that the store latency is always included in the running time.

G.1 [10] <G.1, G.2> Write a VMIPS vector sequence that achieves the peak MFLOPS performance of the processor (use the functional unit and instruction description in Section G.2). Assuming a 500-MHz clock rate, what is the peak MFLOPS?

G.2 [20/15/15] <G.1–G.6> Consider the following vector code run on a 500-MHz version of VMIPS for a fixed vector length of 64:

```
LV       V1,Ra
MULV.D   V2,V1,V3
ADDV.D   V4,V1,V3
SV       Rb,V2
SV       Rc,V4
```

Ignore all strip-mining overhead, but assume that the store latency must be included in the time to perform the loop. The entire sequence produces 64 results.

a. [20] <G.1–G.4> Assuming no chaining and a single memory pipeline, how many chimes are required? How many clock cycles per result (including both stores as one result) does this vector sequence require, including start-up overhead?

b. [15] <G.1–G.4> If the vector sequence is chained, how many clock cycles per result does this sequence require, including overhead?

c. [15] <G.1–G.6> Suppose VMIPS had three memory pipelines and chaining. If there were no bank conflicts in the accesses for the above loop, how many clock cycles are required per result for this sequence?

G.3 [20/20/15/15/20/20/20] <G.2–G.6> Consider the following FORTRAN code:

```
      do 10 i=1,n
            A(i) = A(i) + B(i)
            B(i) = x * B(i)
10    continue
```

Use the techniques of Section G.6 to estimate performance throughout this exercise, assuming a 500-MHz version of VMIPS.

a. [20] <G.2–G.6> Write the best VMIPS vector code for the inner portion of the loop. Assume `x` is in F0 and the addresses of `A` and `B` are in `Ra` and `Rb`, respectively.

b. [20] <G.2–G.6> Find the total time for this loop on VMIPS (T_{100}). What is the MFLOPS rating for the loop (R_{100})?

c. [15] <G.2–G.6> Find R_∞ for this loop.

d. [15] <G.2–G.6> Find $N_{1/2}$ for this loop.

e. [20] <G.2–G.6> Find N_v for this loop. Assume the scalar code has been pipeline scheduled so that each memory reference takes six cycles and each FP operation takes three cycles. Assume the scalar overhead is also T_{loop}.

f. [20] <G.2–G.6> Assume VMIPS has two memory pipelines. Write vector code that takes advantage of the second memory pipeline. Show the layout in convoys.

g. [20] <G.2–G.6> Compute T_{100} and R_{100} for VMIPS with two memory pipelines.

G.4 [20/10] <G.3> Suppose we have a version of VMIPS with eight memory banks (each a double word wide) and a memory access time of eight cycles.

a. [20] <G.3> If a load vector of length 64 is executed with a stride of 20 double words, how many cycles will the load take to complete?

b. [10] <G.3> What percentage of the memory bandwidth do you achieve on a 64-element load at stride 20 versus stride 1?

G.5 [12/12] <G.5–G.6> Consider the following loop:

```
      C = 0.0
      do 10 i=1,64
            A(i) = A(i) + B(i)
            C = C + A(i)
10    continue
```

a. [12] <G.5–G.6> Split the loop into two loops: one with no dependence and one with a dependence. Write these loops in FORTRAN—as a source-to-source transformation. This optimization is called *loop fission*.

b. [12] <G.5–G.6> Write the VMIPS vector code for the loop without a dependence.

G.6 [20/15/20/20] <G.5–G.6> The compiled Linpack performance of the Cray-1 (designed in 1976) was almost doubled by a better compiler in 1989. Let's look at a simple example of how this might occur. Consider the DAXPY-like loop (where *k* is a parameter to the procedure containing the loop):

```
      do 10 i=1,64
            do 10 j=1,64
            Y(k,j) = a*X(i,j) + Y(k,j)
10    continue
```

a. [20] <G.5–G.6> Write the *straightforward* code sequence for just the inner loop in VMIPS vector instructions.

b. [15] <G.5–G.6> Using the techniques of Section G.6, estimate the performance of this code on VMIPS by finding T_{64} in clock cycles. You may assume

that T_{loop} of overhead is incurred for each iteration of the outer loop. What limits the performance?

c. [20] <G.5–G.6> Rewrite the VMIPS code to reduce the performance limitation; show the resulting inner loop in VMIPS vector instructions. (*Hint:* Think about what establishes T_{chime}; can you affect it?) Find the total time for the resulting sequence.

d. [20] <G.5–G.6> Estimate the performance of your new version, using the techniques of Section G.6 and finding T_{64}.

G.7 [15/15/25] <G.4> Consider the following code.

```
      do 10 i=1,64
             if (B(i) .ne. 0) then
                    A(i) = A(i) / B(i)
10    continue
```

Assume that the addresses of A and B are in Ra and Rb, respectively, and that F0 contains 0.

a. [15] <G.4> Write the VMIPS code for this loop using the vector-mask capability.

b. [15] <G.4> Write the VMIPS code for this loop using scatter-gather.

c. [25] <G.4> Estimate the performance (T_{100} in clock cycles) of these two vector loops, assuming a divide latency of 20 cycles. Assume that all vector instructions run at one result per clock, independent of the setting of the vector-mask register. Assume that 50% of the entries of B are 0. Considering hardware costs, which would you build if the above loop were typical?

G.8 [15/20/15/15] <G.1–G.6> In "Fallacies and Pitfalls" of Chapter 1, we saw that the difference between peak and sustained performance could be large: For one problem, a Hitachi S810 had a peak speed twice as high as that of the Cray X-MP, while for another more realistic problem, the Cray X-MP was twice as fast as the Hitachi processor. Let's examine why this might occur using two versions of VMIPS and the following code sequences:

```
C           Code sequence 1
      do 10 i=1,10000
             A(i) = x * A(i) + y * A(i)
10    continue
C           Code sequence 2
      do 10 i=1,100
             A(i) = x * A(i)
10    continue
```

Assume there is a version of VMIPS (call it VMIPS-II) that has two copies of every floating-point functional unit with full chaining among them. Assume that both VMIPS and VMIPS-II have two load-store units. Because of the extra func-

tional units and the increased complexity of assigning operations to units, all the overheads (T_{loop} and T_{start}) are doubled.

a. [15] <G.1–G.6> Find the number of clock cycles for code sequence 1 on VMIPS.

b. [20] <G.1–G.6> Find the number of clock cycles on code sequence 1 for VMIPS-II. How does this compare to VMIPS?

c. [15] <G.1–G.6> Find the number of clock cycles on code sequence 2 for VMIPS.

d. [15] <G.1–G.6> Find the number of clock cycles on code sequence 2 for VMIPS-II. How does this compare to VMIPS?

G.9 [20] <G.5> Here is a tricky piece of code with two-dimensional arrays. Does this loop have dependences? Can these loops be written so they are parallel? If so, how? Rewrite the *source* code so that it is clear that the loop can be vectorized, if possible.

```
      do 290 j = 2,n
            do 290 i = 2,j
                  aa(i,j)= aa(i-1,j)*aa(i-1,j)+bb(i,j)
290   continue
```

G.10 [12/15] <G.5> Consider the following loop:

```
      do 10 i = 2,n
         A(i) = B
10       C(i) = A(i-1)
```

a. [12] <G.5> Show there is a loop-carried dependence in this code fragment.

b. [15] <G.5> Rewrite the code in FORTRAN so that it can be vectorized as two separate vector sequences.

G.11 [15/25/25] <G.5> As we saw in Section G.5, some loop structures are not easily vectorized. One common structure is a *reduction*—a loop that reduces an array to a single value by repeated application of an operation. This is a special case of a recurrence. A common example occurs in dot product:

```
      dot = 0.0
      do 10 i=1,64
10          dot = dot + A(i) * B(i)
```

This loop has an obvious loop-carried dependence (on `dot`) and cannot be vectorized in a straightforward fashion. The first thing a good vectorizing compiler would do is split the loop to separate out the vectorizable portion and the recurrence and perhaps rewrite the loop as

```
      do 10 i=1,64
10          dot(i) = A(i) * B(i)
      do 20 i=2,64
20          dot(1) = dot(1) + dot(i)
```

The variable `dot` has been expanded into a vector; this transformation is called *scalar expansion*. We can try to vectorize the second loop either relying strictly on the compiler (part (a)) or with hardware support as well (part (b)). There is an important caveat in the use of vector techniques for reduction. To make reduction work, we are relying on the associativity of the operator being used for the reduction. Because of rounding and finite range, however, floating-point arithmetic is not strictly associative. For this reason, most compilers require the programmer to indicate whether associativity can be used to more efficiently compile reductions.

a. [15] <G.5> One simple scheme for compiling the loop with the recurrence is to add sequences of progressively shorter vectors—two 32-element vectors, then two 16-element vectors, and so on. This technique has been called *recursive doubling*. It is faster than doing all the operations in scalar mode. Show how the FORTRAN code would look for execution of the second loop in the preceding code fragment using recursive doubling.

b. [25] <G.5> In some vector processors, the vector registers are addressable, and the operands to a vector operation may be two different parts of the same vector register. This allows another solution for the reduction, called *partial sums*. The key idea in partial sums is to reduce the vector to *m* sums where *m* is the total latency through the vector functional unit, including the operand read and write times. Assume that the VMIPS vector registers are addressable (e.g., you can initiate a vector operation with the operand V1(16), indicating that the input operand began with element 16). Also, assume that the total latency for adds, including operand read and write, is eight cycles. Write a VMIPS code sequence that reduces the contents of V1 to eight partial sums. It can be done with one vector operation.

c. Discuss how adding the extension in part (b) would affect a machine that had multiple lanes.

G.12 [40] <G.2–G.4> Extend the MIPS simulator to be a VMIPS simulator, including the ability to count clock cycles. Write some short benchmark programs in MIPS and VMIPS assembly language. Measure the speedup on VMIPS, the percentage of vectorization, and usage of the functional units.

G.13 [50] <G.5> Modify the MIPS compiler to include a dependence checker. Run some scientific code and loops through it and measure what percentage of the statements could be vectorized.

G.14 [Discussion] Some proponents of vector processors might argue that the vector processors have provided the best path to ever-increasing amounts of processor power by focusing their attention on boosting peak vector performance. Others would argue that the emphasis on peak performance is misplaced because an increasing percentage of the programs are dominated by nonvector performance. (Remember Amdahl's Law?) The proponents would respond that programmers should work to make their programs vectorizable. What do you think about this argument?

G.15 [Discussion] Consider the points raised in "Concluding Remarks" (Section G.8). This topic—the relative advantages of pipelined scalar processors versus FP vector processors—was the source of much debate in the 1990s. What advantages do you see for each side? What would you do in this situation?

H.1	Introduction	H-2
H.2	Basic Techniques of Integer Arithmetic	H-2
H.3	Floating Point	H-13
H.4	Floating-Point Multiplication	H-17
H.5	Floating-Point Addition	H-21
H.6	Division and Remainder	H-27
H.7	More on Floating-Point Arithmetic	H-33
H.8	Speeding Up Integer Addition	H-37
H.9	Speeding Up Integer Multiplication and Division	H-45
H.10	Putting It All Together	H-58
H.11	Fallacies and Pitfalls	H-62
H.12	Historical Perspective and References	H-63
	Exercises	H-69

H

Computer Arithmetic

by David Goldberg
Xerox Palo Alto Research Center

The Fast drives out the Slow even if the Fast is wrong.

W. Kahan

© 2003 Elsevier Science (USA). All rights reserved.

H.1 Introduction

Although computer arithmetic is sometimes viewed as a specialized part of CPU design, it is a very important part. This was brought home for Intel in 1994 when their Pentium chip was discovered to have a bug in the divide algorithm. This floating-point flaw resulted in a flurry of bad publicity for Intel and also cost them a lot of money. Intel took a $300 million write-off to cover the cost of replacing the buggy chips.

In this appendix we will study some basic floating-point algorithms, including the division algorithm used on the Pentium. Although a tremendous variety of algorithms have been proposed for use in floating-point accelerators, actual implementations are usually based on refinements and variations of the few basic algorithms presented here. In addition to choosing algorithms for addition, subtraction, multiplication, and division, the computer architect must make other choices. What precisions should be implemented? How should exceptions be handled? This appendix will give you the background for making these and other decisions.

Our discussion of floating point will focus almost exclusively on the IEEE floating-point standard (IEEE 754) because of its rapidly increasing acceptance. Although floating-point arithmetic involves manipulating exponents and shifting fractions, the bulk of the time in floating-point operations is spent operating on fractions using integer algorithms (but not necessarily sharing the hardware that implements integer instructions). Thus, after our discussion of floating point, we will take a more detailed look at integer algorithms.

Some good references on computer arithmetic, in order from least to most detailed, are Chapter 4 of Patterson and Hennessy [1994]; Chapter 7 of Hamacher, Vranesic, and Zaky [1984]; Gosling [1980]; and Scott [1985].

H.2 Basic Techniques of Integer Arithmetic

Readers who have studied computer arithmetic before will find most of this section to be review.

Ripple-Carry Addition

Adders are usually implemented by combining multiple copies of simple components. The natural components for addition are *half adders* and *full adders*. The half adder takes two bits a and b as input and produces a sum bit s and a carry bit c_{out} as output. Mathematically, $s = (a + b) \bmod 2$, and $c_{\text{out}} = \lfloor (a + b)/2 \rfloor$, where $\lfloor\ \rfloor$ is the floor function. As logic equations, $s = a\overline{b} + \overline{a}b$ and $c_{\text{out}} = ab$, where ab means $a \wedge b$ and $a + b$ means $a \vee b$. The half adder is also called a (2,2) adder, since it takes two inputs and produces two outputs. The full adder is a (3,2) adder and is defined by $s = (a + b + c) \bmod 2$, $c_{\text{out}} = \lfloor (a + b + c)/2 \rfloor$, or the logic equations

H.2.1 $s = a\bar{b}\,\bar{c} + \bar{a}b\bar{c} + \bar{a}\bar{b}c + abc$

H.2.2 $c_{\text{out}} = ab + ac + bc$

The principal problem in constructing an adder for n-bit numbers out of smaller pieces is propagating the carries from one piece to the next. The most obvious way to solve this is with a *ripple-carry adder,* consisting of n full adders, as illustrated in Figure H.1. (In the figures in this appendix, the least-significant bit is always on the right.) The inputs to the adder are $a_{n-1}a_{n-2}\cdots a_0$ and $b_{n-1}b_{n-2}\cdots b_0$, where $a_{n-1}a_{n-2}\cdots a_0$ represents the number $a_{n-1}\,2^{n-1} + a_{n-2}\,2^{n-2} + \cdots + a_0$. The c_{i+1} output of the ith adder is fed into the c_{i+1} input of the next adder (the $(i + 1)$-th adder) with the lower-order carry-in c_0 set to 0. Since the low-order carry-in is wired to 0, the low-order adder could be a half adder. Later, however, we will see that setting the low-order carry-in bit to 1 is useful for performing subtraction.

In general, the time a circuit takes to produce an output is proportional to the maximum number of logic levels through which a signal travels. However, determining the exact relationship between logic levels and timings is highly technology dependent. Therefore, when comparing adders we will simply compare the number of logic levels in each one. How many levels are there for a ripple-carry adder? It takes two levels to compute c_1 from a_0 and b_0. Then it takes two more levels to compute c_2 from c_1, a_1, b_1, and so on, up to c_n. So there are a total of $2n$ levels. Typical values of n are 32 for integer arithmetic and 53 for double-precision floating point. The ripple-carry adder is the slowest adder, but also the cheapest. It can be built with only n simple cells, connected in a simple, regular way.

Because the ripple-carry adder is relatively slow compared with the designs discussed in Section H.8, you might wonder why it is used at all. In technologies like CMOS, even though ripple adders take time O(n), the constant factor is very small. In such cases short ripple adders are often used as building blocks in larger adders.

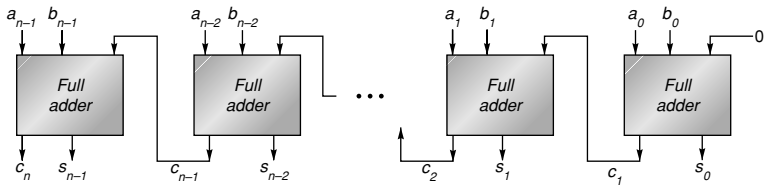


Figure H.1 Ripple-carry adder, consisting of *n* full adders. The carry-out of one full adder is connected to the carry-in of the adder for the next most-significant bit. The carries ripple from the least-significant bit (on the right) to the most-significant bit (on the left).

Radix-2 Multiplication and Division

The simplest multiplier computes the product of two unsigned numbers, one bit at a time, as illustrated in Figure H.2(a). The numbers to be multiplied are $a_{n-1}a_{n-2} \cdots a_0$ and $b_{n-1}b_{n-2} \cdots b_0$, and they are placed in registers A and B, respectively. Register P is initially 0. Each multiply step has two parts.

Multiply Step

(*i*) If the least-significant bit of A is 1, then register B, containing $b_{n-1}b_{n-2} \cdots b_0$, is added to P; otherwise $00 \cdots 00$ is added to P. The sum is placed back into P.

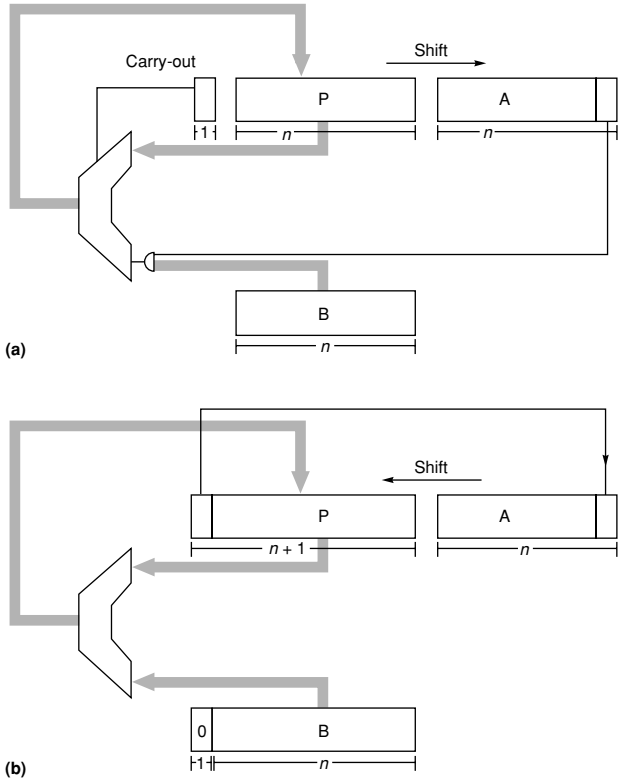


Figure H.2 Block diagram of (a) multiplier and (b) divider for *n*-bit unsigned integers. Each multiplication step consists of adding the contents of P to either B or 0 (depending on the low-order bit of A), replacing P with the sum, and then shifting both P and A one bit right. Each division step involves first shifting P and A one bit left, subtracting B from P, and, if the difference is nonnegative, putting it into P. If the difference is nonnegative, the low-order bit of A is set to 1.

(*ii*) Registers P and A are shifted right, with the carry-out of the sum being moved into the high-order bit of P, the low-order bit of P being moved into register A, and the rightmost bit of A, which is not used in the rest of the algorithm, being shifted out.

After n steps, the product appears in registers P and A, with A holding the lower-order bits.

The simplest divider also operates on unsigned numbers and produces the quotient bits one at a time. A hardware divider is shown in Figure H.2(b). To compute a/b, put a in the A register, b in the B register, 0 in the P register, and then perform n divide steps. Each divide step consists of four parts:

Divide Step

(*i*) Shift the register pair (P,A) one bit left.

(*ii*) Subtract the content of register B (which is $b_{n-1}b_{n-2}\cdots b_0$) from register P, putting the result back into P.

(*iii*) If the result of step 2 is negative, set the low-order bit of A to 0, otherwise to 1.

(*iv*) If the result of step 2 is negative, restore the old value of P by adding the contents of register B back into P.

After repeating this process n times, the A register will contain the quotient, and the P register will contain the remainder. This algorithm is the binary version of the paper-and-pencil method; a numerical example is illustrated in Figure H.3(a).

Notice that the two block diagrams in Figure H.2 are very similar. The main difference is that the register pair (P,A) shifts right when multiplying and left when dividing. By allowing these registers to shift bidirectionally, the same hardware can be shared between multiplication and division.

The division algorithm illustrated in Figure H.3(a) is called *restoring,* because if subtraction by b yields a negative result, the P register is restored by adding b back in. The restoring algorithm has a variant that skips the restoring step and instead works with the resulting negative numbers. Each step of this *nonrestoring* algorithm has three parts:

Nonrestoring Divide Step

If P is negative,

(*i-a*) Shift the register pair (P,A) one bit left.

(*ii-a*) Add the contents of register B to P.

Else,

(*i-b*) Shift the register pair (P,A) one bit left.

(*ii-b*) Subtract the contents of register B from P.

(*iii*) If P is negative, set the low-order bit of A to 0, otherwise set it to 1.

After repeating this n times, the quotient is in A. If P is nonnegative, it is the remainder. Otherwise, it needs to be restored (i.e., add b), and then it will be the remainder. A numerical example is given in Figure H.3(b). Since (*i-a*) and (*i-b*)

(a)

P	A	
00000	1110	Divide 14 = 1110_2 by 3 = 11_2. B always contains 0011_2.
00001	110	step 1(i): shift.
–00011		step 1(ii): subtract.
–00010	110**0**	step 1(iii): result is negative, set quotient bit to 0.
00001	110**0**	step 1(iv): restore.
00011	10**0**	step 2(i): shift.
–00011		step 2(ii): subtract.
00000	10**01**	step 2(iii): result is nonnegative, set quotient bit to 1.
00001	0**01**	step 3(i): shift.
–00011		step 3(ii): subtract.
–00010	0**010**	step 3(iii): result is negative, set quotient bit to 0.
00001	0**010**	step 3(iv): restore.
00010	**010**	step 4(i): shift.
–00011		step 4(ii): subtract.
–00001	**0100**	step 4(iii): result is negative, set quotient bit to 0.
00010	**0100**	step 4(iv): restore. The quotient is 0100_2 and the remainder is 00010_2.

(b)

00000	1110	Divide 14 = 1110_2 by 3 = 11_2. B always contains 0011_2.
00001	110	step 1(i-b): shift.
+11101		step 1(ii-b): subtract b (add two's complement).
11110	110**0**	step 1(iii): P is negative, so set quotient bit 0.
11101	10**0**	step 2(i-a): shift.
+00011		step 2(ii-a): add b.
00000	10**01**	step 2(iii): P is nonnegative, so set quotient bit to 1.
00001	0**01**	step 3(i-b): shift.
+11101		step 3(ii-b): subtract b.
11110	0**010**	step 3(iii): P is negative, so set quotient bit to 0.
11100	**010**	step 4(i-a): shift.
+00011		step 4(ii-a): add b.
11111	**0100**	step 4(iii): P is negative, so set quotient bit to 0.
+00011		Remainder is negative, so do final restore step.
00010		The quotient is 0100_2 and the remainder is 00010_2.

Figure H.3 Numerical example of (a) restoring division and (b) nonrestoring division.

are the same, you might be tempted to perform this common step first, and then test the sign of P. That doesn't work, since the sign bit can be lost when shifting.

The explanation for why the nonrestoring algorithm works is this. Let r_k be the contents of the (P,A) register pair at step k, ignoring the quotient bits (which

are simply sharing the unused bits of register A). In Figure H.3(a), initially A contains 14, so $r_0 = 14$. At the end of the first step, $r_1 = 28$, and so on. In the restoring algorithm, part (i) computes $2r_k$ and then part (ii) $2r_k - 2^n b$ ($2^n b$ since b is subtracted from the left half). If $2r_k - 2^n b \geq 0$, both algorithms end the step with identical values in (P,A). If $2r_k - 2^n b < 0$, then the restoring algorithm restores this to $2r_k$, and the next step begins by computing $r_{\text{res}} = 2(2r_k) - 2^n b$. In the nonrestoring algorithm, $2r_k - 2^n b$ is kept as a negative number, and in the next step $r_{\text{nonres}} = 2(2r_k - 2^n b) + 2^n b = 4r_k - 2^n b = r_{\text{res}}$. Thus (P,A) has the same bits in both algorithms.

If a and b are unsigned n-bit numbers, hence in the range $0 \leq a,b \leq 2^n - 1$, then the multiplier in Figure H.2 will work if register P is n bits long. However, for division, P must be extended to $n + 1$ bits in order to detect the sign of P. Thus the adder must also have $n + 1$ bits.

Why would anyone implement restoring division, which uses the same hardware as nonrestoring division (the control is slightly different) but involves an extra addition? In fact, the usual implementation for restoring division doesn't actually perform an add in step (*iv*). Rather, the sign resulting from the subtraction is tested at the output of the adder, and only if the sum is nonnegative is it loaded back into the P register.

As a final point, before beginning to divide, the hardware must check to see whether the divisor is 0.

Signed Numbers

There are four methods commonly used to represent signed n-bit numbers: *sign magnitude, two's complement, one's complement,* and *biased*. In the sign magnitude system, the high-order bit is the sign bit, and the low-order $n - 1$ bits are the magnitude of the number. In the two's complement system, a number and its negative add up to 2^n. In one's complement, the negative of a number is obtained by complementing each bit (or alternatively, the number and its negative add up to $2^n - 1$). In each of these three systems, nonnegative numbers are represented in the usual way. In a biased system, nonnegative numbers do not have their usual representation. Instead, all numbers are represented by first adding them to the bias, and then encoding this sum as an ordinary unsigned number. Thus a negative number k can be encoded as long as k + bias ≥ 0. A typical value for the bias is 2^{n-1}.

Example Using 4-bit numbers ($n = 4$), if $k = 3$ (or in binary, $k = 0011_2$), how is $-k$ expressed in each of these formats?

Answer In signed magnitude, the leftmost bit in $k = \mathbf{0}011_2$ is the sign bit, so flip it to 1: $-k$ is represented by $\mathbf{1}011_2$. In two's complement, $k + 1101_2 = 2^n = 16$. So $-k$ is represented by 1101_2. In one's complement, the bits of $k = 0011_2$ are flipped, so $-k$ is represented by 1100_2. For a biased system, assuming a bias of $2^{n-1} = 8$, k is represented by k + bias $= 1011_2$, and $-k$ by $-k$ + bias $= 0101_2$.

The most widely used system for representing integers, two's complement, is the system we will use here. One reason for the popularity of two's complement is that it makes signed addition easy: Simply discard the carry-out from the high-order bit. To add 5 + −2, for example, add 0101_2 and 1110_2 to obtain 0011_2, resulting in the correct value of 3. A useful formula for the value of a two's complement number $a_{n-1}a_{n-2}\cdots a_1a_0$ is

H.2.3
$$-a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + \cdots + a_1 2^1 + a_0$$

As an illustration of this formula, the value of 1101_2 as a 4-bit two's complement number is $-1\cdot2^3 + 1\cdot2^2 + 0\cdot2^1 + 1\cdot2^0 = -8 + 4 + 1 = -3$, confirming the result of the example above.

Overflow occurs when the result of the operation does not fit in the representation being used. For example, if unsigned numbers are being represented using 4 bits, then $6 = 0110_2$ and $11 = 1011_2$. Their sum (17) overflows because its binary equivalent (10001_2) doesn't fit into 4 bits. For unsigned numbers, detecting overflow is easy; it occurs exactly when there is a carry-out of the most-significant bit. For two's complement, things are trickier: Overflow occurs exactly when the carry into the high-order bit is different from the (to be discarded) carry-out of the high-order bit. In the example of 5 + −2 above, a 1 is carried both into and out of the leftmost bit, avoiding overflow.

Negating a two's complement number involves complementing each bit and then adding 1. For instance, to negate 0011_2, complement it to get 1100_2 and then add 1 to get 1101_2. Thus, to implement $a - b$ using an adder, simply feed a and $\overline{b}$ (where $\overline{b}$ is the number obtained by complementing each bit of b) into the adder and set the low-order, carry-in bit to 1. This explains why the rightmost adder in Figure H.1 is a full adder.

Multiplying two's complement numbers is not quite as simple as adding them. The obvious approach is to convert both operands to be nonnegative, do an unsigned multiplication, and then (if the original operands were of opposite signs) negate the result. Although this is conceptually simple, it requires extra time and hardware. Here is a better approach: Suppose that we are multiplying a times b using the hardware shown in Figure H.2(a). Register A is loaded with the number a; B is loaded with b. Since the content of register B is always b, we will use B and b interchangeably. If B is potentially negative but A is nonnegative, the only change needed to convert the unsigned multiplication algorithm into a two's complement one is to ensure that when P is shifted, it is shifted arithmetically; that is, the bit shifted into the high-order bit of P should be the sign bit of P (rather than the carry-out from the addition). Note that our n-bit-wide adder will now be adding n-bit two's complement numbers between -2^{n-1} and $2^{n-1} - 1$.

Next, suppose a is negative. The method for handling this case is called *Booth recoding*. Booth recoding is a very basic technique in computer arithmetic and will play a key role in Section H.9. The algorithm on page H-4 computes $a \times b$ by examining the bits of a from least significant to most significant. For example, if $a = 7 = 0111_2$, then step (i) will successively add B, add B, add B, and add 0. Booth recoding "recodes" the number 7 as $8 - 1 = 1000_2 - 0001_2 = 100\overline{1}$, where

$\bar{1}$ represents −1. This gives an alternate way to compute $a \times b$; namely, successively subtract B, add 0, add 0, and add B. This is more complicated than the unsigned algorithm on page H-4, since it uses both addition and subtraction. The advantage shows up for negative values of a. With the proper recoding, we can treat a as though it were unsigned. For example, take $a = -4 = 1100_2$. Think of 1100_2 as the unsigned number 12, and recode it as $12 = 16 - 4 = 10000_2 - 0100_2 = 10\bar{1}00$. If the multiplication algorithm is only iterated n times ($n = 4$ in this case), the high-order digit is ignored, and we end up subtracting $0100_2 = 4$ times the multiplier—exactly the right answer. This suggests that multiplying using a recoded form of a will work equally well for both positive and negative numbers. And indeed, to deal with negative values of a, all that is required is to sometimes subtract b from P, instead of adding either b or 0 to P. Here are the precise rules: If the initial content of A is $a_{n-1} \cdots a_0$, then at the ith multiply step, the low-order bit of register A is a_i, and step (i) in the multiplication algorithm becomes

I. If $a_i = 0$ and $a_{i-1} = 0$, then add 0 to P.

II. If $a_i = 0$ and $a_{i-1} = 1$, then add B to P.

III. If $a_i = 1$ and $a_{i-1} = 0$, then subtract B from P.

IV. If $a_i = 1$ and $a_{i-1} = 1$, then add 0 to P.

For the first step, when $i = 0$, take a_{i-1} to be 0.

Example When multiplying −6 times −5, what is the sequence of values in the (P,A) register pair?

Answer See Figure H.4.

P	A	
0000	1010	Put $-6 = 1010_2$ into A, $-5 = 1011_2$ into B.
0000	1010	step 1(i): $a_0 = a_{-1} = 0$, so from rule I add 0.
0000	0101	step 1(ii): shift.
+ 0101		step 2(i): $a_1 = 1$, $a_0 = 0$. Rule III says subtract b (or add $-b = -1011_2 = 0101_2$).
0101	0101	
0010	1010	step 2(ii): shift.
+ 1011		step 3(i): $a_2 = 0$, $a_1 = 1$. Rule II says add b (1011).
1101	1010	
1110	1101	step 3(ii): shift. (Arithmetic shift—load 1 into leftmost bit.)
+ 0101		step 4(i): $a_3 = 1$, $a_2 = 0$. Rule III says subtract b.
0011	1101	
0001	1110	step 4(ii): shift. Final result is $00011110_2 = 30$.

Figure H.4 Numerical example of Booth recoding. Multiplication of $a = -6$ by $b = -5$ to get 30.

The four cases above can be restated as saying that in the *i*th step you should add $(a_{i-1} - a_i)$B to P. With this observation, it is easy to verify that these rules work, because the result of all the additions is

$$\sum_{i=0}^{n-1} b(a_{i-1} - a_i)2^i = b(-a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + \ldots + a_1 2 + a_0) + ba_{-1}$$

Using Equation H.2.3 (page H-8) together with $a_{-1} = 0$, the right-hand side is seen to be the value of $b \times a$ as a two's complement number.

The simplest way to implement the rules for Booth recoding is to extend the A register one bit to the right so that this new bit will contain a_{i-1}. Unlike the naive method of inverting any negative operands, this technique doesn't require extra steps or any special casing for negative operands. It has only slightly more control logic. If the multiplier is being shared with a divider, there will already be the capability for subtracting *b,* rather than adding it. To summarize, a simple method for handling two's complement multiplication is to pay attention to the sign of P when shifting it right, and to save the most recently shifted-out bit of A to use in deciding whether to add or subtract *b* from P.

Booth recoding is usually the best method for designing multiplication hardware that operates on signed numbers. For hardware that doesn't directly implement it, however, performing Booth recoding in software or microcode is usually too slow because of the conditional tests and branches. If the hardware supports arithmetic shifts (so that negative *b* is handled correctly), then the following method can be used. Treat the multiplier *a* as if it were an unsigned number, and perform the first $n - 1$ multiply steps using the algorithm on page H-4. If $a < 0$ (in which case there will be a 1 in the low-order bit of the A register at this point), then subtract *b* from P; otherwise ($a \geq 0$) neither add nor subtract. In either case, do a final shift (for a total of *n* shifts). This works because it amounts to multiplying *b* by $-a_{n-1}\,2^{n-1} + \cdots + a_1 2 + a_0$, which is the value of $a_{n-1} \cdots a_0$ as a two's complement number by Equation H.2.3. If the hardware doesn't support arithmetic shift, then converting the operands to be nonnegative is probably the best approach.

Two final remarks: A good way to test a signed-multiply routine is to try $-2^{n-1} \times -2^{n-1}$, since this is the only case that produces a $2n - 1$ bit result. Unlike multiplication, division is usually performed in hardware by converting the operands to be nonnegative and then doing an unsigned divide. Because division is substantially slower (and less frequent) than multiplication, the extra time used to manipulate the signs has less impact than it does on multiplication.

Systems Issues

When designing an instruction set, a number of issues related to integer arithmetic need to be resolved. Several of them are discussed here.

First, what should be done about integer overflow? This situation is complicated by the fact that detecting overflow differs depending on whether the oper-

ands are signed or unsigned integers. Consider signed arithmetic first. There are three approaches: Set a bit on overflow, trap on overflow, or do nothing on overflow. In the last case, software has to check whether or not an overflow occurred. The most convenient solution for the programmer is to have an enable bit. If this bit is turned on, then overflow causes a trap. If it is turned off, then overflow sets a bit (or alternatively, have two different add instructions). The advantage of this approach is that both trapping and nontrapping operations require only one instruction. Furthermore, as we will see in Section H.7, this is analogous to how the IEEE floating-point standard handles floating-point overflow. Figure H.5 shows how some common machines treat overflow.

What about unsigned addition? Notice that none of the architectures in Figure H.5 traps on unsigned overflow. The reason for this is that the primary use of unsigned arithmetic is in manipulating addresses. It is convenient to be able to subtract from an unsigned address by adding. For example, when $n = 4$, we can subtract 2 from the unsigned address $10 = 1010_2$ by adding $14 = 1110_2$. This generates an overflow, but we would not want a trap to be generated.

A second issue concerns multiplication. Should the result of multiplying two n-bit numbers be a $2n$-bit result, or should multiplication just return the low-order n bits, signaling overflow if the result doesn't fit in n bits? An argument in favor of an n-bit result is that in virtually all high-level languages, multiplication is an operation in which arguments are integer variables and the result is an integer variable of the same type. Therefore, compilers won't generate code that utilizes a double-precision result. An argument in favor of a $2n$-bit result is that it can be used by an assembly language routine to substantially speed up multiplication of multiple-precision integers (by about a factor of 3).

A third issue concerns machines that want to execute one instruction every cycle. It is rarely practical to perform a multiplication or division in the same amount of time that an addition or register-register move takes. There are three possible approaches to this problem. The first is to have a single-cycle *multiply-step* instruction. This might do one step of the Booth algorithm. The second

Machine	Trap on signed overflow?	Trap on unsigned overflow?	Set bit on signed overflow?	Set bit on unsigned overflow?
VAX	If enable is on	No	Yes. `Add` sets V bit.	Yes. `Add` sets C bit.
IBM 370	If enable is on	No	Yes. `Add` sets cond code.	Yes. Logical `add` sets cond code.
Intel 8086	No	No	Yes. `Add` sets V bit.	Yes. `Add` sets C bit.
MIPS R3000	Two `add` instructions: one always traps, the other never does.	No	No. Software must deduce it from sign of operands and result.	
SPARC	No	No	`Addcc` sets V bit. `Add` does not.	`Addcc` sets C bit. `Add` does not.

Figure H.5 Summary of how various machines handle integer overflow. Both the 8086 and SPARC have an instruction that traps if the V bit is set, so the cost of trapping on overflow is one extra instruction.

approach is to do integer multiplication in the floating-point unit and have it be part of the floating-point instruction set. (This is what DLX does.) The third approach is to have an autonomous unit in the CPU do the multiplication. In this case, the result either can be guaranteed to be delivered in a fixed number of cycles—and the compiler charged with waiting the proper amount of time—or there can be an interlock. The same comments apply to division as well. As examples, the original SPARC had a multiply-step instruction but no divide-step instruction, while the MIPS R3000 has an autonomous unit that does multiplication and division (newer versions of the SPARC architecture added an integer multiply instruction). The designers of the HP Precision Architecture did an especially thorough job of analyzing the frequency of the operands for multiplication and division, and they based their multiply and divide steps accordingly. (See Magenheimer et al. [1988] for details.)

The final issue involves the computation of integer division and remainder for negative numbers. For example, what is –5 DIV 3 and –5 MOD 3? When computing x DIV y and x MOD y, negative values of x occur frequently enough to be worth some careful consideration. (On the other hand, negative values of y are quite rare.) If there are built-in hardware instructions for these operations, they should correspond to what high-level languages specify. Unfortunately, there is no agreement among existing programming languages. See Figure H.6.

One definition for these expressions stands out as clearly superior; namely, x DIV $y = \lfloor x/y \rfloor$, so that 5 DIV 3 = 1, –5 DIV 3 = –2. And MOD should satisfy $x = (x \text{ DIV } y) \times y + x \text{ MOD } y$, so that x MOD $y \geq 0$. Thus 5 MOD 3 = 2, and –5 MOD 3 = 1. Some of the many advantages of this definition are as follows:

1. A calculation to compute an index into a hash table of size N can use MOD N and be guaranteed to produce a valid index in the range from 0 to $N - 1$.
2. In graphics, when converting from one coordinate system to another, there is no "glitch" near 0. For example, to convert from a value x expressed in a system that uses 100 dots per inch to a value y on a bitmapped display with 70 dots per inch, the formula $y = (70 \times x)$ DIV 100 maps one or two x coordinates into each y coordinate. But if DIV were defined as in Pascal to be x/y rounded to 0, then 0 would have three different points (–1, 0, 1) mapped into it.

Language	Division	Remainder
FORTRAN	–5/3 = –1	MOD(–5, 3) = –2
Pascal	–5 DIV 3 = –1	–5 MOD 3 = 1
Ada	–5/3 = –1	–5 MOD 3 = 1 –5 REM 3 = –2
C	–5/3 undefined	–5 % 3 undefined
Modula-3	–5 DIV 3 = –2	–5 MOD 3 = 1

Figure H.6 Examples of integer division and integer remainder in various programming languages.

3. x MOD 2^k is the same as performing a bitwise AND with a mask of k bits, and x DIV 2^k is the same as doing a k-bit arithmetic right shift.

Finally, a potential pitfall worth mentioning concerns multiple-precision addition. Many instruction sets offer a variant of the `add` instruction that adds three operands: two n-bit numbers together with a third single-bit number. This third number is the carry from the previous addition. Since the multiple-precision number will typically be stored in an array, it is important to be able to increment the array pointer without destroying the carry bit.

H.3 Floating Point

Many applications require numbers that aren't integers. There are a number of ways that nonintegers can be represented. One is to use *fixed point;* that is, use integer arithmetic and simply imagine the binary point somewhere other than just to the right of the least-significant digit. Adding two such numbers can be done with an integer add, whereas multiplication requires some extra shifting. Other representations that have been proposed involve storing the logarithm of a number and doing multiplication by adding the logarithms, or using a pair of integers (a,b) to represent the fraction a/b. However, only one noninteger representation has gained widespread use, and that is *floating point*. In this system, a computer word is divided into two parts, an exponent and a significand. As an example, an exponent of -3 and significand of 1.5 might represent the number $1.5 \times 2^{-3} = 0.1875$. The advantages of standardizing a particular representation are obvious. Numerical analysts can build up high-quality software libraries, computer designers can develop techniques for implementing high-performance hardware, and hardware vendors can build standard accelerators. Given the predominance of the floating-point representation, it appears unlikely that any other representation will come into widespread use.

The semantics of floating-point instructions are not as clear-cut as the semantics of the rest of the instruction set, and in the past the behavior of floating-point operations varied considerably from one computer family to the next. The variations involved such things as the number of bits allocated to the exponent and significand, the range of exponents, how rounding was carried out, and the actions taken on exceptional conditions like underflow and overflow. Computer architecture books used to dispense advice on how to deal with all these details, but fortunately this is no longer necessary. That's because the computer industry is rapidly converging on the format specified by IEEE standard 754-1985 (also an international standard, IEC 559). The advantages of using a standard variant of floating point are similar to those for using floating point over other noninteger representations.

IEEE arithmetic differs from many previous arithmetics in the following major ways:

1. When rounding a "halfway" result to the nearest floating-point number, it picks the one that is even.
2. It includes the *special values* NaN, ∞, and −∞.
3. It uses *denormal* numbers to represent the result of computations whose value is less than $1.0 \times 2^{E_{\min}}$.
4. It rounds to nearest by default, but it also has three other rounding modes.
5. It has sophisticated facilities for handling exceptions.

To elaborate on (1), note that when operating on two floating-point numbers, the result is usually a number that cannot be exactly represented as another floating-point number. For example, in a floating-point system using base 10 and two significant digits, $6.1 \times 0.5 = 3.05$. This needs to be rounded to two digits. Should it be rounded to 3.0 or 3.1? In the IEEE standard, such halfway cases are rounded to the number whose low-order digit is even. That is, 3.05 rounds to 3.0, not 3.1. The standard actually has four *rounding modes*. The default is *round to nearest,* which rounds ties to an even number as just explained. The other modes are round toward 0, round toward +∞, and round toward –∞.

We will elaborate on the other differences in following sections. For further reading, see IEEE [1985], Cody et al. [1984], and Goldberg [1991].

Special Values and Denormals

Probably the most notable feature of the standard is that by default a computation continues in the face of exceptional conditions, such as dividing by 0 or taking the square root of a negative number. For example, the result of taking the square root of a negative number is a *NaN* (*N*ot *a N*umber), a bit pattern that does not represent an ordinary number. As an example of how NaNs might be useful, consider the code for a zero finder that takes a function F as an argument and evaluates F at various points to determine a zero for it. If the zero finder accidentally probes outside the valid values for F, F may well cause an exception. Writing a zero finder that deals with this case is highly language and operating-system dependent, because it relies on how the operating system reacts to exceptions and how this reaction is mapped back into the programming language. In IEEE arithmetic it is easy to write a zero finder that handles this situation and runs on many different systems. After each evaluation of F, it simply checks to see whether F has returned a NaN; if so, it knows it has probed outside the domain of F.

In IEEE arithmetic, if the input to an operation is a NaN, the output is NaN (e.g., 3 + NaN = NaN). Because of this rule, writing floating-point subroutines that can accept NaN as an argument rarely requires any special case checks. For example, suppose that arccos is computed in terms of arctan, using the formula $\arccos x = 2 \arctan(\sqrt{(1-x)/(1+x)})$. If arctan handles an argument of NaN properly, arccos will automatically do so too. That's because if x is a NaN, $1+x$, $1-x$, $(1+x)/(1-x)$, and $\sqrt{(1-x)/(1+x)}$ will also be NaNs. No checking for NaNs is required.

While the result of $\sqrt{-1}$ is a NaN, the result of 1/0 is not a NaN, but $+\infty$, which is another special value. The standard defines arithmetic on infinities (there is both $+\infty$ and $-\infty$) using rules such as $1/\infty = 0$. The formula $\arccos x = 2 \arctan\left(\sqrt{(1-x)/(1+x)}\right)$ illustrates how infinity arithmetic can be used. Since $\arctan x$ asymptotically approaches $\pi/2$ as x approaches ∞, it is natural to define $\arctan(\infty) = \pi/2$, in which case $\arccos(-1)$ will automatically be computed correctly as $2 \arctan(\infty) = \pi$.

The final kind of special values in the standard are *denormal* numbers. In many floating-point systems, if $E_{\min}$ is the smallest exponent, a number less than $1.0 \times 2^{E_{\min}}$ cannot be represented, and a floating-point operation that results in a number less than this is simply flushed to 0. In the IEEE standard, on the other hand, numbers less than $1.0 \times 2^{E_{\min}}$ are represented using significands less than 1. This is called *gradual underflow*. Thus, as numbers decrease in magnitude below $2^{E_{\min}}$, they gradually lose their significance and are only represented by 0 when all their significance has been shifted out. For example, in base 10 with four significant figures, let $x = 1.234 \times 10^{E_{\min}}$. Then $x/10$ will be rounded to $0.123 \times 10^{E_{\min}}$, having lost a digit of precision. Similarly $x/100$ rounds to $0.012 \times 10^{E_{\min}}$, and $x/1000$ to $0.001 \times 10^{E_{\min}}$, while $x/10000$ is finally small enough to be rounded to 0. Denormals make dealing with small numbers more predictable by maintaining familiar properties such as $x = y \Leftrightarrow x - y = 0$. For example, in a flush-to-zero system (again in base 10 with four significant digits), if $x = 1.256 \times 10^{E_{\min}}$ and $y = 1.234 \times 10^{E_{\min}}$, then $x - y = 0.022 \times 10^{E_{\min}}$, which flushes to zero. So even though $x \neq y$, the computed value of $x - y = 0$. This never happens with gradual underflow. In this example, $x - y = 0.022 \times 10^{E_{\min}}$ is a denormal number, and so the computation of $x - y$ is exact.

Representation of Floating-Point Numbers

Let us consider how to represent single-precision numbers in IEEE arithmetic. Single-precision numbers are stored in 32 bits: 1 for the sign, 8 for the exponent, and 23 for the fraction. The exponent is a signed number represented using the bias method (see the subsection "Signed Numbers," page H-7) with a bias of 127. The term *biased exponent* refers to the unsigned number contained in bits 1 through 8 and *unbiased exponent* (or just exponent) means the actual power to which 2 is to be raised. The fraction represents a number less than 1, but the *significand* of the floating-point number is 1 plus the fraction part. In other words, if e is the biased exponent (value of the exponent field) and f is the value of the fraction field, the number being represented is $1.f \times 2^{e-127}$.

Example What single-precision number does the following 32-bit word represent?

1 10000001 01000000000000000000000

Answer Considered as an unsigned number, the exponent field is 129, making the value of the exponent 129 − 127 = 2. The fraction part is $.01_2 = .25$, making the significand 1.25. Thus, this bit pattern represents the number $-1.25 \times 2^2 = -5$.

The fractional part of a floating-point number (.25 in the example above) must not be confused with the significand, which is 1 plus the fractional part. The leading 1 in the significand $1.f$ does not appear in the representation; that is, the leading bit is implicit. When performing arithmetic on IEEE format numbers, the fraction part is usually *unpacked,* which is to say the implicit 1 is made explicit.

Figure H.7 summarizes the parameters for single (and other) precisions. It shows the exponents for single precision to range from –126 to 127; accordingly, the biased exponents range from 1 to 254. The biased exponents of 0 and 255 are used to represent special values. This is summarized in Figure H.8. When the biased exponent is 255, a zero fraction field represents infinity, and a nonzero fraction field represents a NaN. Thus, there is an entire family of NaNs. When the biased exponent and the fraction field are 0, then the number represented is 0. Because of the implicit leading 1, ordinary numbers always have a significand greater than or equal to 1. Thus, a special convention such as this is required to represent 0. Denormalized numbers are implemented by having a word with a zero exponent field represent the number $0.f \times 2^{E_{\min}}$.

	Single	Single extended	Double	Double extended
p (bits of precision)	24	≥ 32	53	≥ 64
$E_{\max}$	127	≥ 1023	1023	≥ 16383
$E_{\min}$	–126	≤ –1022	–1022	≤ –16382
Exponent bias	127		1023	

Figure H.7 Format parameters for the IEEE 754 floating-point standard. The first row gives the number of bits in the significand. The blanks are unspecified parameters.

Exponent	Fraction	Represents
$e = E_{\min} - 1$	$f = 0$	±0
$e = E_{\min} - 1$	$f \neq 0$	$0.f \times 2^{E_{\min}}$
$E_{\min} \leq e \leq E_{\max}$	—	$1.f \times 2^e$
$e = E_{\max} + 1$	$f = 0$	± ∞
$e = E_{\max} + 1$	$f \neq 0$	NaN

Figure H.8 Representation of special values. When the exponent of a number falls outside the range $E_{\min} \leq e \leq E_{\max}$, then that number has a special interpretation as indicated in the table.

The primary reason why the IEEE standard, like most other floating-point formats, uses biased exponents is that it means nonnegative numbers are ordered in the same way as integers. That is, the magnitude of floating-point numbers can be compared using an integer comparator. Another (related) advantage is that 0 is represented by a word of all 0's. The downside of biased exponents is that adding them is slightly awkward, because it requires that the bias be subtracted from their sum.

H.4 Floating-Point Multiplication

The simplest floating-point operation is multiplication, so we discuss it first. A binary floating-point number x is represented as a significand and an exponent, $x = s \times 2^e$. The formula

$$(s_1 \times 2^{e1}) \bullet (s_2 \times 2^{e2}) = (s_1 \bullet s_2) \times 2^{e1+e2}$$

shows that a floating-point multiply algorithm has several parts. The first part multiplies the significands using ordinary integer multiplication. Because floating-point numbers are stored in sign magnitude form, the multiplier need only deal with unsigned numbers (although we have seen that Booth recoding handles signed two's complement numbers painlessly). The second part rounds the result. If the significands are unsigned p-bit numbers (e.g., $p = 24$ for single precision), then the product can have as many as $2p$ bits and must be rounded to a p-bit number. The third part computes the new exponent. Because exponents are stored with a bias, this involves subtracting the bias from the sum of the biased exponents.

Example How does the multiplication of the single-precision numbers

$$1\ 10000010\ 000\ldots = -1 \times 2^3$$
$$0\ 10000011\ 000\ldots = 1 \times 2^4$$

proceed in binary?

Answer When unpacked, the significands are both 1.0, their product is 1.0, and so the result is of the form

$$1\ ????????\ 000\ldots$$

To compute the exponent, use the formula

$$\text{biased exp}\,(e_1 + e_2) = \text{biased exp}(e_1) + \text{biased exp}(e_2) - \text{bias}$$

From Figure H.7, the bias is $127 = 01111111_2$, so in two's complement -127 is 10000001_2. Thus the biased exponent of the product is

$$\begin{array}{r} 10000010 \\ 10000011 \\ +\ 10000001 \\ \hline 10000110 \end{array}$$

Since this is 134 decimal, it represents an exponent of 134 – bias = 134 – 127 = 7, as expected.

The interesting part of floating-point multiplication is rounding. Some of the different cases that can occur are illustrated in Figure H.9. Since the cases are similar in all bases, the figure uses human-friendly base 10, rather than base 2.

In the figure, $p = 3$, so the final result must be rounded to three significant digits. The three most-significant digits are in boldface. The fourth most-significant digit (marked with an arrow) is the *round* digit, denoted by *r*.

If the round digit is less than 5, then the bold digits represent the rounded result. If the round digit is greater than 5 (as in (a)), then 1 must be added to the least-significant bold digit. If the round digit is exactly 5 (as in (b)), then additional digits must be examined to decide between truncation or incrementing by 1. It is only necessary to know if any digits past 5 are nonzero. In the algorithm below, this will be recorded in a *sticky bit*. Comparing (a) and (b) in the figure shows that there are two possible positions for the round digit (relative to the least-significant digit of the product). Case (c) illustrates that when adding 1 to the least-significant bold digit, there may be a carry-out. When this happens, the final significand must be 10.0.

There is a straightforward method of handling rounding using the multiplier of Figure H.2 (page H-4) together with an extra sticky bit. If *p* is the number of bits in the significand, then the A, B, and P registers should be *p* bits wide. Multiply the two significands to obtain a 2*p*-bit product in the (P,A) registers (see

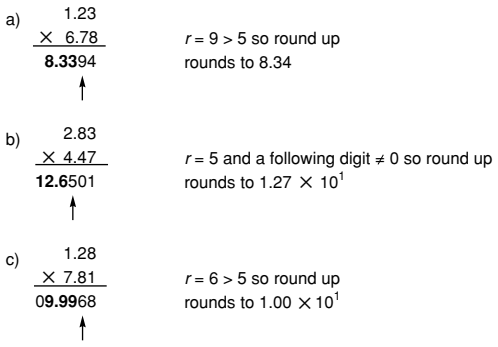


Figure H.9 Examples of rounding a multiplication. Using base 10 and $p = 3$, parts (a) and (b) illustrate that the result of a multiplication can have either $2p - 1$ or $2p$ digits, and hence the position where a 1 is added when rounding up (just left of the arrow) can vary. Part (c) shows that rounding up can cause a carry-out.

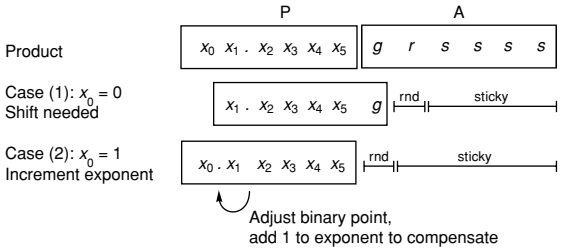


Figure H.10 The two cases of the floating-point multiply algorithm. The top line shows the contents of the P and A registers after multiplying the significands, with $p = 6$. In case (1), the leading bit is 0, and so the P register must be shifted. In case (2), the leading bit is 1, no shift is required, but both the exponent and the round and sticky bits must be adjusted. The sticky bit is the logical OR of the bits marked *s*.

Figure H.10). During the multiplication, the first $p - 2$ times a bit is shifted into the A register, OR it into the sticky bit. This will be used in halfway cases. Let s represent the sticky bit, g (for guard) the most-significant bit of A, and r (for round) the second most-significant bit of A. There are two cases:

1. The high-order bit of P is 0. Shift P left 1 bit, shifting in the g bit from A. Shifting the rest of A is not necessary.
2. The high-order bit of P is 1. Set $s := s \vee r$ and $r := g$, and add 1 to the exponent.

Now if $r = 0$, P is the correctly rounded product. If $r = 1$ and $s = 1$, then P + 1 is the product (where by P + 1 we mean adding 1 to the least-significant bit of P). If $r = 1$ and $s = 0$, we are in a halfway case, and round up according to the least-significant bit of P. As an example, apply the decimal version of these rules to Figure H.9(b). After the multiplication, P = 126 and A = 501, with $g = 5$, $r = 0$, $s = 1$. Since the high-order digit of P is nonzero, case (2) applies and $r := g$, so that $r = 5$, as the arrow indicates in Figure H.9. Since $r = 5$, we could be in a halfway case, but $s = 1$ indicates that the result is in fact slightly over 1/2, so add 1 to P to obtain the correctly rounded product.

The precise rules for rounding depend on the rounding mode and are given in Figure H.11. Note that P is nonnegative, that is, it contains the magnitude of the result. A good discussion of more efficient ways to implement rounding is in Santoro, Bewick, and Horowitz [1989].

Example In binary with $p = 4$, show how the multiplication algorithm computes the product -5×10 in each of the four rounding modes.

Answer In binary, -5 is $-1.010_2 \times 2^2$ and $10 = 1.010_2 \times 2^3$. Applying the integer multiplication algorithm to the significands gives 01100100_2, so P = 0110_2, A = 0100_2,

$g = 0$, $r = 1$, and $s = 0$. The high-order bit of P is 0, so case (1) applies. Thus P becomes 1100_2, and since the result is negative, Figure H.11 gives

round to -∞	1101_2	add 1 since $r \vee s = 1/0 =$ TRUE	
round to +∞	1100_2		
round to 0	1100_2		
round to nearest	1100_2	no add since	$r \wedge p_0 = 1 \wedge 0 =$ FALSE and
			$r \wedge s = 1 \wedge 0 =$ FALSE

The exponent is 2 + 3 = 5, so the result is $-1.100_2 \times 2^5 = -48$, except when rounding to $-\infty$, in which case it is $-1.101_2 \times 2^5 = -52$.

Overflow occurs when the rounded result is too large to be represented. In single precision, this occurs when the result has an exponent of 128 or higher. If e_1 and e_2 are the two biased exponents, then $1 \le e_i \le 254$, and the exponent calculation $e_1 + e_2 - 127$ gives numbers between 1 + 1 − 127 and 254 + 254 − 127, or between −125 and 381. This range of numbers can be represented using 9 bits. So one way to detect overflow is to perform the exponent calculations in a 9-bit adder (see Exercise H.12). Remember that you must check for overflow *after* rounding—the example in Figure H.9(c) shows that this can make a difference.

Denormals

Checking for underflow is somewhat more complex because of denormals. In single precision, if the result has an exponent less than −126, that does not necessarily indicate underflow, because the result might be a denormal number. For example, the product of (1×2^{-64}) with (1×2^{-65}) is 1×2^{-129}, and −129 is below the legal exponent limit. But this result is a valid denormal number, namely, 0.125×2^{-126}. In general, when the unbiased exponent of a product dips below −126, the resulting product must be shifted right and the exponent incremented until the

Rounding mode	Sign of result ≥ 0	Sign of result < 0
–∞		+1 if $r \vee s$
+∞	+1 if $r \vee s$	
0		
Nearest	+1 if $r \wedge p_0$ or $r \wedge s$	+1 if $r \wedge p_0$ or $r \wedge s$

Figure H.11 Rules for implementing the IEEE rounding modes. Let *S* be the magnitude of the preliminary result. Blanks mean that the *p* most-significant bits of *S* are the actual result bits. If the condition listed is true, add 1 to the *p*th most-significant bit of *S*. The symbols *r* and *s* represent the round and sticky bits, while p_0 is the *p*th most-significant bit of *S*.

exponent reaches –126. If this process causes the entire significand to be shifted out, then underflow has occurred. The precise definition of underflow is somewhat subtle—see Section H.7 for details.

When one of the operands of a multiplication is denormal, its significand will have leading zeros, and so the product of the significands will also have leading zeros. If the exponent of the product is less than –126, then the result is denormal, so right-shift and increment the exponent as before. If the exponent is greater than –126, the result may be a normalized number. In this case, *left*-shift the product (while decrementing the exponent) until either it becomes normalized or the exponent drops to –126.

Denormal numbers present a major stumbling block to implementing floating-point multiplication, because they require performing a variable shift in the multiplier, which wouldn't otherwise be needed. Thus, high-performance, floating-point multipliers often do not handle denormalized numbers, but instead trap, letting software handle them. A few practical codes frequently underflow, even when working properly, and these programs will run quite a bit slower on systems that require denormals to be processed by a trap handler.

So far we haven't mentioned how to deal with operands of zero. This can be handled by either testing both operands before beginning the multiplication or testing the product afterward. If you test afterward, be sure to handle the case $0 \times \infty$ properly: This results in NaN, not 0. Once you detect that the result is 0, set the biased exponent to 0. Don't forget about the sign. The sign of a product is the XOR of the signs of the operands, even when the result is 0.

Precision of Multiplication

In the discussion of integer multiplication, we mentioned that designers must decide whether to deliver the low-order word of the product or the entire product. A similar issue arises in floating-point multiplication, where the exact product can be rounded to the precision of the operands or to the next higher precision. In the case of integer multiplication, none of the standard high-level languages contains a construct that would generate a "single times single gets double" instruction. The situation is different for floating point. Many languages allow assigning the product of two single-precision variables to a double-precision one, and the construction can also be exploited by numerical algorithms. The best-known case is using iterative refinement to solve linear systems of equations.

H.5 Floating-Point Addition

Typically, a floating-point operation takes two inputs with p bits of precision and returns a p-bit result. The ideal algorithm would compute this by first performing the operation exactly, and then rounding the result to p bits (using the current rounding mode). The multiplication algorithm presented in the previous section follows this strategy. Even though hardware implementing IEEE arithmetic must

return the same result as the ideal algorithm, it doesn't need to actually perform the ideal algorithm. For addition, in fact, there are better ways to proceed. To see this, consider some examples.

First, the sum of the binary 6-bit numbers 1.10011_2 and $1.10001_2 \times 2^{-5}$: When the summands are shifted so they have the same exponent, this is

$$\begin{array}{r} 1.10011 \\ +\quad .0000110001 \\ \hline \end{array}$$

Using a 6-bit adder (and discarding the low-order bits of the second addend) gives

$$\begin{array}{r} 1.10011 \\ +\quad .00001 \\ \hline 1.10100 \end{array}$$

The first discarded bit is 1. This isn't enough to decide whether to round up. The rest of the discarded bits, 0001, need to be examined. Or actually, we just need to record whether any of these bits are nonzero, storing this fact in a sticky bit just as in the multiplication algorithm. So for adding two p-bit numbers, a p-bit adder is sufficient, as long as the first discarded bit (round) and the OR of the rest of the bits (sticky) are kept. Then Figure H.11 can be used to determine if a roundup is necessary, just as with multiplication. In the example above, sticky is 1, so a roundup is necessary. The final sum is 1.10101_2.

Here's another example:

$$\begin{array}{r} 1.11011 \\ +\quad .0101001 \\ \hline \end{array}$$

A 6-bit adder gives

$$\begin{array}{r} 1.11011 \\ +\quad .01010 \\ \hline 10.00101 \end{array}$$

Because of the carry-out on the left, the round bit isn't the first discarded bit; rather, it is the low-order bit of the sum (1). The discarded bits, 01, are OR'ed together to make sticky. Because round and sticky are both 1, the high-order 6 bits of the sum, 10.0010_2, must be rounded up for the final answer of 10.0011_2.

Next, consider subtraction and the following example:

$$\begin{array}{r} 1.00000 \\ -\quad .00000101111 \\ \hline \end{array}$$

The simplest way of computing this is to convert $-.00000101111_2$ to its two's complement form, so the difference becomes a sum

$$\begin{array}{r} 1.00000 \\ +\ 1.11111010001 \\ \hline \end{array}$$

Computing this sum in a 6-bit adder gives

$$\begin{array}{r} 1.00000 \\ +\ 1.11111 \\ \hline 0.11111 \end{array}$$

Because the top bits canceled, the first discarded bit (the guard bit) is needed to fill in the least-significant bit of the sum, which becomes 0.111110_2, and the second discarded bit becomes the round bit. This is analogous to case (1) in the multiplication algorithm (see page H-19). The round bit of 1 isn't enough to decide whether to round up. Instead, we need to OR all the remaining bits (0001) into a sticky bit. In this case, sticky is 1, so the final result must be rounded up to 0.111111. This example shows that if subtraction causes the most-significant bit to cancel, then one guard bit is needed. It is natural to ask whether two guard bits are needed for the case when the *two* most-significant bits cancel. The answer is no, because if x and y are so close that the top two bits of $x - y$ cancel, then $x - y$ will be exact, so guard bits aren't needed at all.

To summarize, addition is more complex than multiplication because, depending on the signs of the operands, it may actually be a subtraction. If it is an addition, there can be carry-out on the left, as in the second example. If it is subtraction, there can be cancellation, as in the third example. In each case, the position of the round bit is different. However, we don't need to compute the exact sum and then round. We can infer it from the sum of the high-order p bits together with the round and sticky bits.

The rest of this section is devoted to a detailed discussion of the floating-point addition algorithm. Let a_1 and a_2 be the two numbers to be added. The notations e_i and s_i are used for the exponent and significand of the addends a_i. This means that the floating-point inputs have been unpacked and that s_i has an explicit leading bit. To add a_1 and a_2, perform these eight steps.

1. If $e_1 < e_2$, swap the operands. This ensures that the difference of the exponents satisfies $d = e_1 - e_2 \geq 0$. Tentatively set the exponent of the result to e_1.
2. If the signs of a_1 and a_2 differ, replace s_2 by its two's complement.
3. Place s_2 in a p-bit register and shift it $d = e_1 - e_2$ places to the right (shifting in 1's if s_2 was complemented in the previous step). From the bits shifted out, set g to the most-significant bit, r to the next most-significant bit, and set sticky to the OR of the rest.
4. Compute a preliminary significand $S = s_1 + s_2$ by adding s_1 to the p-bit register containing s_2. If the signs of a_1 and a_2 are different, the most-significant bit of S is 1, and there was no carry-out, then S is negative. Replace S with its two's complement. This can only happen when $d = 0$.
5. Shift S as follows. If the signs of a_1 and a_2 are the same and there was a carry-out in step 4, shift S right by one, filling in the high-order position with 1 (the

carry-out). Otherwise shift it left until it is normalized. When left-shifting, on the first shift fill in the low-order position with the g bit. After that, shift in zeros. Adjust the exponent of the result accordingly.

6. Adjust r and s. If S was shifted right in step 5, set r := low-order bit of S before shifting and $s := g$ OR r OR s. If there was no shift, set $r := g$, $s := r$ OR s. If there was a single left shift, don't change r and s. If there were two or more left shifts, $r := 0$, $s := 0$. (In the last case, two or more shifts can only happen when a_1 and a_2 have opposite signs and the same exponent, in which case the computation $s_1 + s_2$ in step 4 will be exact.)
7. Round S using Figure H.11; namely, if a table entry is nonempty, add 1 to the low-order bit of S. If rounding causes carry-out, shift S right and adjust the exponent. This is the significand of the result.
8. Compute the sign of the result. If a_1 and a_2 have the same sign, this is the sign of the result. If a_1 and a_2 have different signs, then the sign of the result depends on which of a_1, a_2 is negative, whether there was a swap in step 1, and whether S was replaced by its two's complement in step 4. See Figure H.12.

Example Use the algorithm to compute the sum $(-1.001_2 \times 2^{-2}) + (-1.111_2 \times 2^0)$

Answer $s_1 = 1.001$, $e_1 = -2$, $s_2 = 1.111$, $e_2 = 0$

1. $e_1 < e_2$, so swap. $d = 2$. Tentative exp = 0.
2. Signs of both operands negative, don't negate s_2.
3. Shift s_2 (1.001 after swap) right by 2, giving $s_2 = .010$, $g = 0$, $r = 1$, $s = 0$.
4.
$$\begin{array}{r} 1.111 \\ +\quad .010 \\ \hline (1)0.001 \end{array}$$
$S = 0.001$, with a carry-out.
5. Carry-out, so shift S right, $S = 1.000$, exp = exp + 1, so exp = 1.

swap	compl	sign(a_1)	sign(a_2)	sign(result)
Yes		+	–	–
Yes		–	+	+
No	No	+	–	+
No	No	–	+	–
No	Yes	+	–	–
No	Yes	–	+	+

Figure H.12 Rules for computing the sign of a sum when the addends have different signs. The *swap* column refers to swapping the operands in step 1, while the *compl* column refers to performing a two's complement in step 4. Blanks are "don't care."

6. r = low-order bit of sum = 1, $s = g \vee r \vee s = 0 \vee 1 \vee 0 = 1$.
7. r AND s = TRUE, so Figure H.11 says round up, $S = S + 1$ or $S = 1.001$.
8. Both signs negative, so sign of result is negative. Final answer: $-S \times 2^{\text{exp}} = 1.001_2 \times 2^1$.

Example Use the algorithm to compute the sum $(-1.010_2) + 1.100_2$

Answer $s_1 = 1.010$, $e_1 = 0$, $s_2 = 1.100$, $e_2 = 0$

1. No swap, $d = 0$, tentative exp = 0.
2. Signs differ, replace s_2 with 0.100.
3. $d = 0$, so no shift. $r = g = s = 0$.
4.
$$\begin{array}{r} 1.010 \\ +\ 0.100 \\ \hline 1.110 \end{array}$$
Signs are different, most-significant bit is 1, no carry-out, so must two's complement sum, giving $S = 0.010$.
5. Shift left twice, so $S = 1.000$, exp = exp − 2, or exp = −2.
6. Two left shifts, so $r = g = s = 0$.
7. No addition required for rounding.
8. Answer is sign $\times S \times 2^{\text{exp}}$ or sign $\times 1.000 \times 2^{-2}$. Get sign from Figure H.12. Since complement but no swap and sign(a_1) is −, the sign of sum is +. Thus answer = $1.000_2 \times 2^{-2}$.

Speeding Up Addition

Let's estimate how long it takes to perform the algorithm above. Step 2 may require an addition, step 4 requires one or two additions, and step 7 may require an addition. If it takes T time units to perform a p-bit add (where $p = 24$ for single precision, 53 for double), then it appears the algorithm will take at least $4T$ time units. But that is too pessimistic. If step 4 requires two adds, then a_1 and a_2 have the same exponent and different signs. But in that case the difference is exact, and so no roundup is required in step 7. Thus only three additions will ever occur. Similarly, it appears that a variable shift may be required both in step 3 and step 5. But if $|e_1 - e_2| \leq 1$, then step 3 requires a right shift of at most one place, so only step 5 needs a variable shift. And if $|e_1 - e_2| > 1$, then step 3 needs a variable shift, but step 5 will require a left shift of at most one place. So only a single variable shift will be performed. Still, the algorithm requires three sequential adds, which, in the case of a 53-bit double-precision significand, can be rather time consuming.

A number of techniques can speed up addition. One is to use pipelining. The "Putting It All Together" section gives examples of how some commercial chips pipeline addition. Another method (used on the Intel 860 [Kohn and Fu 1989]) is to perform two additions in parallel. We now explain how this reduces the latency from $3T$ to T.

There are three cases to consider. First, suppose that both operands have the same sign. We want to combine the addition operations from steps 4 and 7. The position of the high-order bit of the sum is not known ahead of time, because the addition in step 4 may or may not cause a carry-out. Both possibilities are accounted for by having two adders. The first adder assumes the add in step 4 will not result in a carry-out. Thus the values of r and s can be computed before the add is actually done. If r and s indicate a roundup is necessary, the first adder will compute $S = s_1 + s_2 + 1$, where the notation +1 means adding 1 at the position of the least-significant bit of s_1. This can be done with a regular adder by setting the low-order carry-in bit to 1. If r and s indicate no roundup, the adder computes $S = s_1 + s_2$ as usual. One extra detail: when $r = 1$, $s = 0$, you will also need to know the low-order bit of the sum, which can also be computed in advance very quickly. The second adder covers the possibility that there will be carry-out. The values of r and s and the position where the roundup 1 is added are different from above, but again they can be quickly computed in advance. It is not known whether there will be a carry-out until after the add is actually done, but that doesn't matter. By doing both adds in parallel, one adder is guaranteed to reduce the correct answer.

The next case is when a_1 and a_2 have opposite signs, but the same exponent. The sum $a_1 + a_2$ is exact in this case (no roundup is necessary), but the sign isn't known until the add is completed. So don't compute the two's complement (which requires an add) in step 2, but instead compute $\bar{s}_1 + s_2 + 1$ and $s_1 + \bar{s}_2 + 1$ in parallel. The first sum has the result of simultaneously complementing s_1 and computing the sum, resulting in $s_2 - s_1$. The second sum computes $s_1 - s_2$. One of these will be nonnegative and hence the correct final answer. Once again, all the additions are done in one step using two adders operating in parallel.

The last case, when a_1 and a_2 have opposite signs and different exponents, is more complex. If $|e_1 - e_2| > 1$, the location of the leading bit of the difference is in one of two locations, so there are two cases just as in addition. When $|e_1 - e_2| = 1$, cancellation is possible and the leading bit could be almost anywhere. However, only if the leading bit of the difference is in the same position as the leading bit of s_1 could a roundup be necessary. So one adder assumes a roundup, the other assumes no roundup. Thus the addition of step 4 and the rounding of step 7 can be combined. However, there is still the problem of the addition in step 2!

To eliminate this addition, consider the following diagram of step 4:

```
          |—— p ——|
s1        1.xxxxxxx
s2    –        1yyzzzzz
```

If the bits marked z are all 0, then the high-order p bits of $S = s_1 - s_2$ can be computed as $s_1 + \bar{s}_2 + 1$. If at least one of the z bits is 1, use $s_1 + \bar{s}_2$. So $s_1 - s_2$ can be

computed with one addition. However, we still don't know g and r for the two's complement of s_2, which are needed for rounding in step 7.

To compute $s_1 - s_2$ and get the proper g and r bits, combine steps 2 and 4 as follows. Don't complement s_2 in step 2. Extend the adder used for computing S two bits to the right (call the extended sum S'). If the preliminary sticky bit (computed in step 3) is 1, compute $S' = s_1' + \bar{s}_2'$, where s_1' has two 0 bits tacked onto the right, and s_2' has preliminary g and r appended. If the sticky bit is 0, compute $s_1' + \bar{s}_2' + 1$. Now the two low-order bits of S' have the correct values of g and r (the sticky bit was already computed properly in step 3). Finally, this modification can be combined with the modification that combines the addition from steps 4 and 7 to provide the final result in time T, the time for one addition.

A few more details need to be considered, as discussed in Santoro, Bewick, and Horowitz [1989] and Exercise H.17. Although the Santoro paper is aimed at multiplication, much of the discussion applies to addition as well. Also relevant is Exercise H.19, which contains an alternate method for adding signed magnitude numbers.

Denormalized Numbers

Unlike multiplication, for addition very little changes in the preceding description if one of the inputs is a denormal number. There must be a test to see if the exponent field is 0. If it is, then when unpacking the significand there will not be a leading 1. By setting the biased exponent to 1 when unpacking a denormal, the algorithm works unchanged.

To deal with denormalized outputs, step 5 must be modified slightly. Shift S until it is normalized, or until the exponent becomes $E_{\min}$ (that is, the biased exponent becomes 1). If the exponent is $E_{\min}$ and, after rounding, the high-order bit of S is 1, then the result is a normalized number and should be packed in the usual way, by omitting the 1. If, on the other hand, the high-order bit is 0, the result is denormal. When the result is unpacked, the exponent field must be set to 0. Section H.7 discusses the exact rules for detecting underflow.

Incidentally, detecting overflow is very easy. It can only happen if step 5 involves a shift right and the biased exponent at that point is bumped up to 255 in single precision (or 2047 for double precision), or if this occurs after rounding.

H.6 Division and Remainder

In this section, we'll discuss floating-point division and remainder.

Iterative Division

We earlier discussed an algorithm for integer division. Converting it into a floating-point division algorithm is similar to converting the integer multiplication algorithm into floating point. The formula

$$(s_1 \times 2^{e_1}) / (s_2 \times 2^{e_2}) = (s_1 / s_2) \times 2^{e_1 - e_2}$$

shows that if the divider computes s_1/s_2, then the final answer will be this quotient multiplied by $2^{e_1-e_2}$. Referring to Figure H.2(b) (page H-4), the alignment of operands is slightly different from integer division. Load s_2 into B and s_1 into P. The A register is not needed to hold the operands. Then the integer algorithm for division (with the one small change of skipping the very first left shift) can be used, and the result will be of the form $q_0.q_1\cdots$. To round, simply compute two additional quotient bits (guard and round) and use the remainder as the sticky bit. The guard digit is necessary because the first quotient bit might be 0. However, since the numerator and denominator are both normalized, it is not possible for the two most-significant quotient bits to be 0. This algorithm produces one quotient bit in each step.

A different approach to division converges to the quotient at a quadratic rather than a linear rate. An actual machine that uses this algorithm will be discussed in Section H.10. First, we will describe the two main iterative algorithms, and then we will discuss the pros and cons of iteration when compared with the direct algorithms. There is a general technique for constructing iterative algorithms, called *Newton's iteration*, shown in Figure H.13. First, cast the problem in the form of finding the zero of a function. Then, starting from a guess for the zero, approximate the function by its tangent at that guess and form a new guess based on where the tangent has a zero. If x_i is a guess at a zero, then the tangent line has the equation

$$y - f(x_i) = f'(x_i)(x - x_i)$$

This equation has a zero at

$$x = x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \qquad \textbf{H.6.1}$$

To recast division as finding the zero of a function, consider $f(x) = x^{-1} - b$. Since the zero of this function is at $1/b$, applying Newton's iteration to it will give

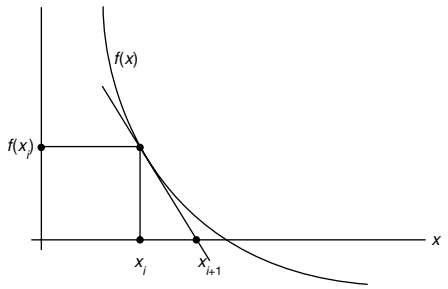


Figure H.13 Newton's iteration for zero finding. If x_i is an estimate for a zero of f, then x_{i+1} is a better estimate. To compute x_{i+1}, find the intersection of the x-axis with the tangent line to f at $f(x_i)$.

an iterative method of computing $1/b$ from b. Using $f'(x) = -1/x^2$, Equation H.6.1 becomes

H.6.2
$$x_{i+1} = x_i - \frac{1/x_i - b}{-1/x_i^2} = x_i + x_i - x_i^2 b = x_i(2 - x_i b)$$

Thus, we could implement computation of a/b using the following method:

1. Scale b to lie in the range $1 \le b < 2$ and get an approximate value of $1/b$ (call it x_0) using a table lookup.
2. Iterate $x_{i+1} = x_i(2 - x_i b)$ until reaching an x_n that is accurate enough.
3. Compute ax_n and reverse the scaling done in step 1.

Here are some more details. How many times will step 2 have to be iterated? To say that x_i is accurate to p bits means that $\left|(x_i - 1/b)/(1/b)\right| = 2^{-p}$, and a simple algebraic manipulation shows that when this is so, then $(x_{i+1} - 1/b)/(1/b) = 2^{-2p}$. Thus the number of correct bits doubles at each step. Newton's iteration is *self-correcting* in the sense that making an error in x_i doesn't really matter. That is, it treats x_i as a guess at $1/b$ and returns x_{i+1} as an improvement on it (roughly doubling the digits). One thing that would cause x_i to be in error is rounding error. More importantly, however, in the early iterations we can take advantage of the fact that we don't expect many correct bits by performing the multiplication in reduced precision, thus gaining speed without sacrificing accuracy. Another application of Newton's iteration is discussed in Exercise H.20.

The second iterative division method is sometimes called *Goldschmidt's algorithm*. It is based on the idea that to compute a/b, you should multiply the numerator and denominator by a number r with $rb \approx 1$. In more detail, let $x_0 = a$ and $y_0 = b$. At each step compute $x_{i+1} = r_i x_i$ and $y_{i+1} = r_i y_i$. Then the quotient $x_{i+1}/y_{i+1} = x_i/y_i = a/b$ is constant. If we pick r_i so that $y_i \to 1$, then $x_i \to a/b$, so the x_i converge to the answer we want. This same idea can be used to compute other functions. For example, to compute the square root of a, let $x_0 = a$ and $y_0 = a$, and at each step compute $x_{i+1} = r_i^2 x_i$, $y_{i+1} = r_i y_i$. Then $x_{i+1}/y_{i+1}^2 = x_i/y_i^2 = 1/a$, so if the r_i are chosen to drive $x_i \to 1$, then $y_i \to \sqrt{a}$. This technique is used to compute square roots on the TI 8847.

Returning to Goldschmidt's division algorithm, set $x_0 = a$ and $y_0 = b$, and write $b = 1 - \delta$, where $|\delta| < 1$. If we pick $r_0 = 1 + \delta$, then $y_1 = r_0 y_0 = 1 - \delta^2$. We next pick $r_1 = 1 + \delta^2$, so that $y_2 = r_1 y_1 = 1 - \delta^4$, and so on. Since $|\delta| < 1$, $y_i \to 1$. With this choice of r_i, the x_i will be computed as $x_{i+1} = r_i x_i = (1 + \delta^{2^i})x_i = (1 + (1 - b)^{2^i})x_i$, or

H.6.3
$$x_{i+1} = a\,[1 + (1 - b)]\,[1 + (1 - b)^2]\,[1 + (1 - b)^4] \cdots [1 + (1 - b)^{2^i}]$$

There appear to be two problems with this algorithm. First, convergence is slow when b is not near 1 (that is, δ is not near 0); and second, the formula isn't self-correcting—since the quotient is being computed as a product of independent terms, an error in one of them won't get corrected. To deal with slow

convergence, if you want to compute a/b, look up an approximate inverse to b (call it b'), and run the algorithm on ab'/bb'. This will converge rapidly since $bb' \approx 1$.

To deal with the self-correction problem, the computation should be run with a few bits of extra precision to compensate for rounding errors. However, Goldschmidt's algorithm does have a weak form of self-correction, in that the precise value of the r_i does not matter. Thus, in the first few iterations, when the full precision of $1 - \delta^{2^i}$ is not needed you can choose r_i to be a truncation of $1 + \delta^{2^i}$, which may make these iterations run faster without affecting the speed of convergence. If r_i is truncated, then y_i is no longer exactly $1 - \delta^{2^i}$. Thus, Equation H.6.3 can no longer be used, but it is easy to organize the computation so that it does not depend on the precise value of r_i. With these changes, Goldschmidt's algorithm is as follows (the notes in brackets show the connection with our earlier formulas).

1. Scale a and b so that $1 \le b < 2$.
2. Look up an approximation to $1/b$ (call it b') in a table.
3. Set $x_0 = ab'$ and $y_0 = bb'$.
4. Iterate until x_i is close enough to a/b:

   Loop

   $r \approx 2 - y$ [if $y_i = 1 + \delta_i$, then $r \approx 1 - \delta_i$]

   $y = y \times r$ [$y_{i+1} = y_i \times r \approx 1 - \delta_i^2$]

   $x_{i+1} = x_i \times r$ [$x_{i+1} = x_i \times r$]

   End loop

The two iteration methods are related. Suppose in Newton's method that we unroll the iteration and compute each term x_{i+1} directly in terms of b, instead of recursively in terms of x_i. By carrying out this calculation (see Exercise H.22), we discover that

$$x_{i+1} = x_0(2 - x_0 b)\,[(1 + (x_0 b - 1)^2]\,[1 + (x_0 b - 1)^4] \ldots [1 + (x_0 b - 1)^{2^i}]$$

This formula is very similar to Equation H.6.3. In fact they are identical if a, b in H.6.3 are replaced with ax_0, bx_0 and $a = 1$. Thus if the iterations were done to infinite precision, the two methods would yield exactly the same sequence x_i.

The advantage of iteration is that it doesn't require special divide hardware. Instead, it can use the multiplier (which, however, requires extra control). Further, on each step, it delivers twice as many digits as in the previous step—unlike ordinary division, which produces a fixed number of digits at every step.

There are two disadvantages with inverting by iteration. The first is that the IEEE standard requires division to be correctly rounded, but iteration only delivers a result that is close to the correctly rounded answer. In the case of Newton's

iteration, which computes $1/b$ instead of a/b directly, there is an additional problem. Even if $1/b$ were correctly rounded, there is no guarantee that a/b will be. An example in decimal with $p = 2$ is $a = 13$, $b = 51$. Then $a/b = .2549\ldots$, which rounds to .25. But $1/b = .0196\ldots$, which rounds to .020, and then $a \times .020 = .26$, which is off by 1. The second disadvantage is that iteration does not give a remainder. This is especially troublesome if the floating-point divide hardware is being used to perform integer division, since a remainder operation is present in almost every high-level language.

Traditional folklore has held that the way to get a correctly rounded result from iteration is to compute $1/b$ to slightly more than $2p$ bits, compute a/b to slightly more than $2p$ bits, and then round to p bits. However, there is a faster way, which apparently was first implemented on the TI 8847. In this method, a/b is computed to about 6 extra bits of precision, giving a preliminary quotient q. By comparing qb with a (again with only 6 extra bits), it is possible to quickly decide whether q is correctly rounded or whether it needs to be bumped up or down by 1 in the least-significant place. This algorithm is explored further in Exercise H.21.

One factor to take into account when deciding on division algorithms is the relative speed of division and multiplication. Since division is more complex than multiplication, it will run more slowly. A common rule of thumb is that division algorithms should try to achieve a speed that is about one-third that of multiplication. One argument in favor of this rule is that there are real programs (such as some versions of spice) where the ratio of division to multiplication is 1:3. Another place where a factor of 3 arises is in the standard iterative method for computing square root. This method involves one division per iteration, but it can be replaced by one using three multiplications. This is discussed in Exercise H.20.

Floating-Point Remainder

For nonnegative integers, integer division and remainder satisfy

$$a = (a \text{ DIV } b)b + a \text{ REM } b,\ 0 \le a \text{ REM } b < b$$

A floating-point remainder x REM y can be similarly defined as $x = \text{INT}(x/y)y + x$ REM y. How should x/y be converted to an integer? The IEEE remainder function uses the round-to-even rule. That is, pick $n = \text{INT}\ (x/y)$ so that $|x/y - n| \le 1/2$. If two different n satisfy this relation, pick the even one. Then REM is defined to be $x - yn$. Unlike integers where $0 \le a$ REM $b < b$, for floating-point numbers $|x \text{ REM } y| \le y/2$. Although this defines REM precisely, it is not a practical operational definition, because n can be huge. In single precision, n could be as large as $2^{127}/2^{-126} = 2^{253} \approx 10^{76}$.

There is a natural way to compute REM if a direct division algorithm is used. Proceed as if you were computing x/y. If $x = s_1 2^{e_1}$ and $y = s_2 2^{e_2}$ and the divider is as in Figure H.2(b) (page H-4), then load s_1 into P and s_2 into B. After $e_1 - e_2$ division steps, the P register will hold a number r of the form $x - yn$ satisfying

$0 \le r < y$. Since the IEEE remainder satisfies $|\text{REM}| \le y/2$, REM is equal to either r or $r - y$. It is only necessary to keep track of the last quotient bit produced, which is needed to resolve halfway cases. Unfortunately, $e_1 - e_2$ can be a lot of steps, and floating-point units typically have a maximum amount of time they are allowed to spend on one instruction. Thus, it is usually not possible to implement REM directly. None of the chips discussed in Section H.10 implements REM, but they could by providing a remainder-step instruction—this is what is done on the Intel 8087 family. A remainder step takes as arguments two numbers x and y, and performs divide steps until either the remainder is in P or n steps have been performed, where n is a small number, such as the number of steps required for division in the highest-supported precision. Then REM can be implemented as a software routine that calls the REM step instruction $\lfloor (e_1 - e_2)/n \rfloor$ times, initially using x as the numerator, but then replacing it with the remainder from the previous REM step.

REM can be used for computing trigonometric functions. To simplify things, imagine that we are working in base 10 with five significant figures, and consider computing $\sin x$. Suppose that $x = 7$. Then we can reduce by $\pi = 3.1416$ and compute $\sin(7) = \sin(7 - 2 \times 3.1416) = \sin(0.7168)$ instead. But suppose we want to compute $\sin(2.0 \times 10^5)$. Then $2 \times 10^5 / 3.1416 = 63661.8$, which in our five-place system comes out to be 63662. Since multiplying 3.1416 times 63662 gives 200000.5392, which rounds to 2.0000×10^5, argument reduction reduces 2×10^5 to 0, which is not even close to being correct. The problem is that our five-place system does not have the precision to do correct argument reduction. Suppose we had the REM operator. Then we could compute 2×10^5 REM 3.1416 and get −.53920. However, this is still not correct because we used 3.1416, which is an approximation for π. The value of 2×10^5 REM π is −.071513.

Traditionally, there have been two approaches to computing periodic functions with large arguments. The first is to return an error for their value when x is large. The second is to store π to a very large number of places and do exact argument reduction. The REM operator is not much help in either of these situations. There is a third approach that has been used in some math libraries, such as the Berkeley UNIX 4.3bsd release. In these libraries, π is computed to the nearest floating-point number. Let's call this machine π, and denote it by π'. Then when computing $\sin x$, reduce x using x REM π'. As we saw in the above example, x REM π' is quite different from x REM π when x is large, so that computing $\sin x$ as $\sin(x \text{ REM } \pi')$ will not give the exact value of $\sin x$. However, computing trigonometric functions in this fashion has the property that all familiar identities (such as $\sin^2 x + \cos^2 x = 1$) are true to within a few rounding errors. Thus, using REM together with machine π provides a simple method of computing trigonometric functions that is accurate for small arguments and still may be useful for large arguments.

When REM is used for argument reduction, it is very handy if it also returns the low-order bits of n (where x REM $y = x - ny$). This is because a practical implementation of trigonometric functions will reduce by something smaller than 2π. For example, it might use $\pi/2$, exploiting identities such as $\sin(x - \pi/2) = -\cos$

x, $\sin(x - \pi) = -\sin x$. Then the low bits of n are needed to choose the correct identity.

H.7 More on Floating-Point Arithmetic

Before leaving the subject of floating-point arithmetic, we present a few additional topics.

Fused Multiply-Add

Probably the most common use of floating-point units is performing matrix operations, and the most frequent matrix operation is multiplying a matrix times a matrix (or vector), which boils down to computing an inner product, $x_1 \cdot y_1 + x_2 \cdot y_2 + \ldots + x_n \cdot y_n$. Computing this requires a series of multiply-add combinations.

Motivated by this, the IBM RS/6000 introduced a single instruction that computes $ab + c$, the *fused multiply-add*. Although this requires being able to read three operands in a single instruction, it has the potential for improving the performance of computing inner products.

The fused multiply-add computes $ab + c$ exactly and then rounds. Although rounding only once increases the accuracy of inner products somewhat, that is not its primary motivation. There are two main advantages of rounding once. First, as we saw in the previous sections, rounding is expensive to implement because it may require an addition. By rounding only once, an addition operation has been eliminated. Second, the extra accuracy of fused multiply-add can be used to compute correctly rounded division and square root when these are not available directly in hardware. Fused multiply-add can also be used to implement efficient floating-point multiple-precision packages.

The implementation of correctly rounded division using fused multiply-add has many details, but the main idea is simple. Consider again the example from Section H.6 (page H-31), which was computing a/b with $a = 13$, $b = 51$. Then $1/b$ rounds to $b' = .020$, and ab' rounds to $q' = .26$, which is not the correctly rounded quotient. Applying fused multiply-add twice will correctly adjust the result, via the formulas

$$r = a - bq'$$
$$q'' = q' + rb'$$

Computing to two-digit accuracy, $bq' = 51 \times .26$ rounds to 13, and so $r = a - bq'$ would be 0, giving no adjustment. But using fused multiply-add gives $r = a - bq' = 13 - (51 \times .26) = -.26$, and then $q'' = q' + rb' = .26 - .0052 = .2548$, which rounds to the correct quotient, .25. More details can be found in the papers by Montoye, Hokenek, and Runyon [1990] and Markstein [1990].

Precisions

The standard specifies four precisions: *single, single extended, double,* and *double extended*. The properties of these precisions are summarized in Figure H.7 (page H-16). Implementations are not required to have all four precisions, but are encouraged to support either the combination of single and single extended or all of single, double, and double extended. Because of the widespread use of double precision in scientific computing, double precision is almost always implemented. Thus the computer designer usually only has to decide whether to support double extended and, if so, how many bits it should have.

The Motorola 68882 and Intel 387 coprocessors implement extended precision using the smallest allowable size of 80 bits (64 bits of significand). However, many of the more recently designed, high-performance floating-point chips do not implement 80-bit extended precision. One reason is that the 80-bit width of extended precision is awkward for 64-bit buses and registers. Some new architectures, such as SPARC V8 and PA-RISC, specify a 128-bit extended (or *quad*) precision. They have established a *de facto* convention for quad that has 15 bits of exponent and 113 bits of significand.

Although most high-level languages do not provide access to extended precision, it is very useful to writers of mathematical software. As an example, consider writing a library routine to compute the length of a vector (x,y) in the plane, namely, $\sqrt{x^2 + y^2}$. If x is larger than $2^{E_{\max}/2}$, then computing this in the obvious way will overflow. This means that either the allowable exponent range for this subroutine will be cut in half or a more complex algorithm using scaling will have to be employed. But if extended precision is available, then the simple algorithm will work. Computing the length of a vector is a simple task, and it is not difficult to come up with an algorithm that doesn't overflow. However, there are more complex problems for which extended precision means the difference between a simple, fast algorithm and a much more complex one. One of the best examples of this is binary-to-decimal conversion. An efficient algorithm for binary-to-decimal conversion that makes essential use of extended precision is very readably presented in Coonen [1984]. This algorithm is also briefly sketched in Goldberg [1991]. Computing accurate values for transcendental functions is another example of a problem that is made much easier if extended precision is present.

One very important fact about precision concerns *double rounding*. To illustrate in decimals, suppose that we want to compute 1.9×0.66, and that single precision is two digits, while extended precision is three digits. The exact result of the product is 1.254. Rounded to extended precision, the result is 1.25. When further rounded to single precision, we get 1.2. However, the result of 1.9×0.66 correctly rounded to single precision is 1.3. Thus, rounding twice may not produce the same result as rounding once. Suppose you want to build hardware that only does double-precision arithmetic. Can you simulate single precision by computing first in double precision and then rounding to single? The above example suggests that you can't. However, double rounding is not always dangerous. In fact, the following rule is true (this is not easy to prove, but see Exercise H.25).

If x *and* y *have* p*-bit significands, and* x + y *is computed exactly and then rounded to* q *places, a second rounding to* p *places will not change the answer if* $q \geq 2p + 2$. *This is true not only for addition, but also for multiplication, division, and square root.*

In our example above, $q = 3$ and $p = 2$, so $q \geq 2p + 2$ is not true. On the other hand, for IEEE arithmetic, double precision has $q = 53$, $p = 24$, so $q = 53 \geq 2p + 2 = 50$. Thus, single precision can be implemented by computing in double precision—that is, computing the answer exactly and then rounding to double—and then rounding to single precision.

Exceptions

The IEEE standard defines five exceptions: underflow, overflow, divide by zero, inexact, and invalid. By default, when these exceptions occur, they merely set a flag and the computation continues. The flags are *sticky,* meaning that once set they remain set until explicitly cleared. The standard strongly encourages implementations to provide a trap-enable bit for each exception. When an exception with an enabled trap handler occurs, a user trap handler is called, and the value of the associated exception flag is undefined. In Section H.3 we mentioned that $\sqrt{-3}$ has the value NaN and 1/0 is ∞. These are examples of operations that raise an exception. By default, computing $\sqrt{-3}$ sets the invalid flag and returns the value NaN. Similarly 1/0 sets the divide-by-zero flag and returns ∞.

The underflow, overflow, and divide-by-zero exceptions are found in most other systems. The *invalid exception* is for the result of operations such as $\sqrt{-1}$, 0/0, or $\infty - \infty$, which don't have any natural value as a floating-point number or as $\pm\infty$. The *inexact exception* is peculiar to IEEE arithmetic and occurs either when the result of an operation must be rounded or when it overflows. In fact, since 1/0 and an operation that overflows both deliver ∞, the exception flags must be consulted to distinguish between them. The inexact exception is an unusual "exception," in that it is not really an exceptional condition because it occurs so frequently. Thus, enabling a trap handler for the inexact exception will most likely have a severe impact on performance. Enabling a trap handler doesn't affect whether an operation is exceptional except in the case of underflow. This is discussed below.

The IEEE standard assumes that when a trap occurs, it is possible to identify the operation that trapped and its operands. On machines with pipelining or multiple arithmetic units, when an exception occurs, it may not be enough to simply have the trap handler examine the program counter. Hardware support may be necessary to identify exactly which operation trapped.

Another problem is illustrated by the following program fragment.

```
r1 = r2 / r3
r2 = r4 + r5
```

These two instructions might well be executed in parallel. If the divide traps, its argument `r2` could already have been overwritten by the addition, especially since addition is almost always faster than division. Computer systems that support trapping in the IEEE standard must provide some way to save the value of `r2`, either in hardware or by having the compiler avoid such a situation in the first place. This kind of problem is not peculiar to floating point. In the sequence

```
r1 = 0(r2)
r2 = r3
```

it would be efficient to execute `r2 = r3` while waiting for memory. But if accessing `0(r2)` causes a page fault, `r2` might no longer be available for restarting the instruction `r1 = 0(r2)`.

One approach to this problem, used in the MIPS R3010, is to identify instructions that may cause an exception early in the instruction cycle. For example, an addition can overflow only if one of the operands has an exponent of $E_{\max}$, and so on. This early check is conservative: It might flag an operation that doesn't actually cause an exception. However, if such false positives are rare, then this technique will have excellent performance. When an instruction is tagged as being possibly exceptional, special code in a trap handler can compute it without destroying any state. Remember that all these problems occur only when trap handlers are enabled. Otherwise, setting the exception flags during normal processing is straightforward.

Underflow

We have alluded several times to the fact that detection of underflow is more complex than for the other exceptions. The IEEE standard specifies that if an underflow trap handler is enabled, the system must trap if the result is denormal. On the other hand, if trap handlers are disabled, then the underflow flag is set only if there is a loss of accuracy—that is, if the result must be rounded. The rationale is, if no accuracy is lost on an underflow, there is no point in setting a warning flag. But if a trap handler is enabled, the user might be trying to simulate flush-to-zero and should therefore be notified whenever a result dips below $1.0 \times 2^{E\min}$.

So if there is no trap handler, the underflow exception is signaled only when the result is denormal and inexact. But the definitions of *denormal* and *inexact* are both subject to multiple interpretations. Normally, inexact means there was a result that couldn't be represented exactly and had to be rounded. Consider the example (in a base 2 floating-point system with 3-bit significands) of $(1.11_2 \times 2^{-2}) \times (1.11_2 \times 2^{E\min}) = 0.110001_2 \times 2^{E\min}$, with round to nearest in effect. The delivered result is $0.11_2 \times 2^{E\min}$, which had to be rounded, causing inexact to be signaled. But is it correct to also signal underflow? Gradual underflow loses significance because the exponent range is bounded. If the exponent range were unbounded, the delivered result would be $1.10_2 \times 2^{E\min-1}$, exactly the same answer obtained with gradual underflow. The fact that denormalized numbers

have fewer bits in their significand than normalized numbers therefore doesn't make any difference in this case. The commentary to the standard [Cody et al. 1984] encourages this as the criterion for setting the underflow flag. That is, it should be set whenever the delivered result is different from what would be delivered in a system with the same fraction size, but with a very large exponent range. However, owing to the difficulty of implementing this scheme, the standard allows setting the underflow flag whenever the result is denormal and different from the infinitely precise result.

There are two possible definitions of what it means for a result to be denormal. Consider the example of $1.10_2 \times 2^{-1}$ multiplied by $1.01_2 \times 2^{E_{\min}}$. The exact product is $0.1111 \times 2^{E_{\min}}$. The rounded result is the normal number $1.00_2 \times 2^{E_{\min}}$. Should underflow be signaled? Signaling underflow means that you are using the *before rounding* rule, because the result was denormal before rounding. Not signaling underflow means that you are using the *after rounding* rule, because the result is normalized after rounding. The IEEE standard provides for choosing either rule; however, the one chosen must be used consistently for all operations.

To illustrate these rules, consider floating-point addition. When the result of an addition (or subtraction) is denormal, it is always exact. Thus the underflow flag never needs to be set for addition. That's because if traps are not enabled, then no exception is raised. And if traps are enabled, the value of the underflow flag is undefined, so again it doesn't need to be set.

One final subtlety should be mentioned concerning underflow. When there is no underflow trap handler, the result of an operation on p-bit numbers that causes an underflow is a denormal number with $p - 1$ or fewer bits of precision. When traps are enabled, the trap handler is provided with the result of the operation rounded to p bits and with the exponent wrapped around. Now there is a potential double-rounding problem. If the trap handler wants to return the denormal result, it can't just round its argument, because that might lead to a double-rounding error. Thus, the trap handler must be passed at least one extra bit of information if it is to be able to deliver the correctly rounded result.

H.8 Speeding Up Integer Addition

The previous section showed that many steps go into implementing floating-point operations. However, each floating-point operation eventually reduces to an integer operation. Thus, increasing the speed of integer operations will also lead to faster floating point.

Integer addition is the simplest operation and the most important. Even for programs that don't do explicit arithmetic, addition must be performed to increment the program counter and to calculate addresses. Despite the simplicity of addition, there isn't a single best way to perform high-speed addition. We will discuss three techniques that are in current use: carry-lookahead, carry-skip, and carry-select.

Carry-Lookahead

An n-bit adder is just a combinational circuit. It can therefore be written by a logic formula whose form is a sum of products and can be computed by a circuit with two levels of logic. How do you figure out what this circuit looks like? From Equation H.2.1 (page H-3) the formula for the ith sum can be written as

H.8.1 $$s_i = a_i \bar{b}_i \bar{c}_i + \bar{a}_i b_i \bar{c}_i + \bar{a}_i \bar{b}_i c_i + a_i b_i c_i$$

where ci is both the carry-in to the ith adder and the carry-out from the $(i-1)$-st adder.

The problem with this formula is that although we know the values of a_i and b_i—they are inputs to the circuit—we don't know c_i. So our goal is to write c_i in terms of a_i and b_i. To accomplish this, we first rewrite Equation H.2.2 (page H-3) as

H.8.2 $$c_i = g_{i-1} + p_{i-1} c_{i-1}, \quad g_{i-1} = a_{i-1} b_{i-1}, \quad p_{i-1} = a_{i-1} + b_{i-1}$$

Here is the reason for the symbols p and g: If g_{i-1} is true, then c_i is certainly true, so a carry is *generated*. Thus, g is for generate. If p_{i-1} is true, then if c_{i-1} is true, it is *propagated* to c_i. Start with Equation H.8.1 and use Equation H.8.2 to replace c_i with $g_{i-1} + p_{i-1}c_{i-1}$. Then, use Equation H.8.2 with $i - 1$ in place of i to replace c_{i-1} with $c_{i-2,}$ and so on. This gives the result

H.8.3 $$c_i = g_{i-1} + p_{i-1} g_{i-2} + p_{i-1} p_{i-2} g_{i-3} + \cdots + p_{i-1} p_{i-2} \cdots p_1 g_0 + p_{i-1} p_{i-2} \cdots p_1 p_0 c_0$$

An adder that computes carries using Equation H.8.3 is called a *carry-lookahead adder,* or CLA. A CLA requires one logic level to form p and g, two levels to form the carries, and two for the sum, for a grand total of five logic levels. This is a vast improvement over the $2n$ levels required for the ripple-carry adder.

Unfortunately, as is evident from Equation H.8.3 or from Figure H.14, a carry-lookahead adder on n bits requires a fan-in of $n + 1$ at the OR gate as well as at the rightmost AND gate. Also, the p_{n-1} signal must drive n AND gates. In addition, the rather irregular structure and many long wires of Figure H.14 make it impractical to build a full carry-lookahead adder when n is large.

However, we can use the carry-lookahead idea to build an adder that has about $\log_2 n$ logic levels (substantially fewer than the $2n$ required by a ripple-carry adder) and yet has a simple, regular structure. The idea is to build up the p's and g's in steps. We have already seen that

$$c_1 = g_0 + c_0 p_0$$

This says there is a carry-out of the 0th position (c_1) either if there is a carry generated in the 0th position or if there is a carry into the 0th position and the carry propagates. Similarly,

$$c_2 = G_{01} + P_{01} c_0$$

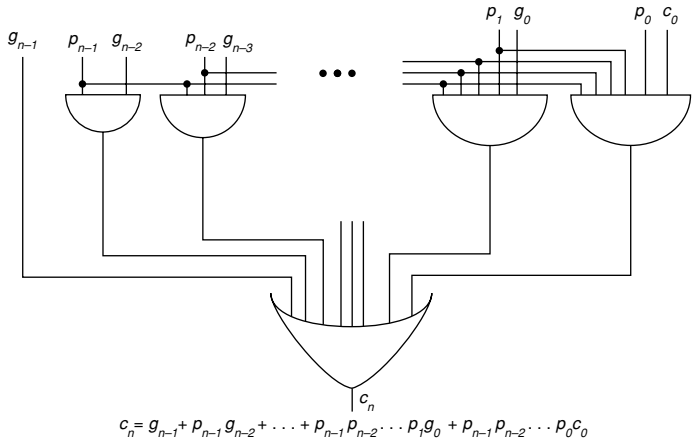


Figure H.14 Pure carry-lookahead circuit for computing the carry-out c_n of an *n*-bit adder.

G_{01} means there is a carry generated out of the block consisting of the first two bits. P_{01} means that a carry propagates through this block. P and G have the following logic equations:

$$G_{01} = g_1 + p_1g_0$$

$$P_{01} = p_1p_0$$

More generally, for any j with $i < j$, $j + 1 < k$, we have the recursive relations

H.8.4 $$c_{k+1} = G_{ik} + P_{ik}c_i$$

H.8.5 $$G_{ik} = G_{j+1,k} + P_{j+1,k}G_{ij}$$

H.8.6 $$P_{ik} = P_{ij}\,P_{j+1,k}$$

Equation H.8.5 says that a carry is generated out of the block consisting of bits i through k inclusive if it is generated in the high-order part of the block $(j + 1, k)$ or if it is generated in the low-order part of the block (i,j) and then propagated through the high part. These equations will also hold for $i \le j < k$ if we set $G_{ii} = g_i$ and $P_{ii} = p_i$.

Example Express P_{03} and G_{03} in terms of p's and g's.

Answer Using Equation H.8.6, $P_{03} = P_{01}P_{23} = P_{00}P_{11}P_{22}P_{33}$. Since $P_{ii} = p_i$, $P_{03} = p_0p_1p_2p_3$. For G_{03}, Equation H.8.5 says $G_{03} = G_{23} + P_{23}G_{01} = (G_{33} + P_{33}G_{22}) + (P_{22}P_{33})(G_{11} + P_{11}G_{00}) = g_3 + p_3g_2 + p_3\,p_2\,g_1 + p_3\,p_2\,p_1g_0$.

With these preliminaries out of the way, we can now show the design of a practical CLA. The adder consists of two parts. The first part computes various values of P and G from p_i and g_i, using Equations H.8.5 and H.8.6; the second part uses these P and G values to compute all the carries via Equation H.8.4. The first part of the design is shown in Figure H.15. At the top of the diagram, input numbers $a_7 \ldots a_0$ and $b_7 \ldots b_0$ are converted to p's and g's using cells of type 1. Then various P's and G's are generated by combining cells of type 2 in a binary tree structure. The second part of the design is shown in Figure H.16. By feeding c_0 in at the bottom of this tree, all the carry bits come out at the top. Each cell must know a pair of (P,G) values in order to do the conversion, and the value it needs is written inside the cells. Now compare Figures H.15 and H.16. There is a one-to-one correspondence between cells, and the value of (P,G) needed by the carry-generating cells is exactly the value known by the corresponding (P,G)-generating cells. The combined cell is shown in Figure H.17. The numbers to be added flow into the top and downward through the tree, combining with c_0 at the bottom and flowing back up the tree to form the carries. Note that one thing is missing from Figure H.17: a small piece of extra logic to compute c_8 for the carry-out of the adder.

The bits in a CLA must pass through about $\log_2 n$ logic levels, compared with $2n$ for a ripple-carry adder. This is a substantial speed improvement, especially for a large n. Whereas the ripple-carry adder had n cells, however, the CLA has

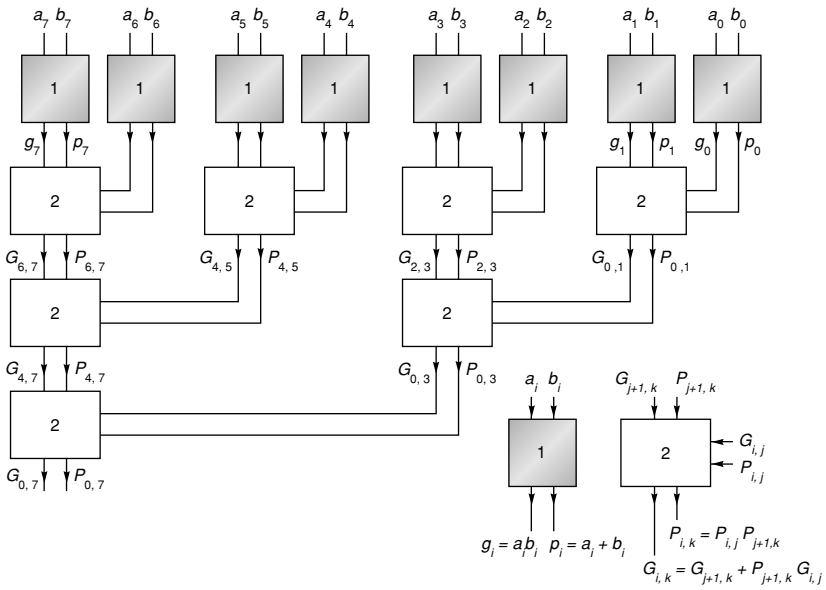


Figure H.15 First part of carry-lookahead tree. As signals flow from the top to the bottom, various values of P and G are computed.

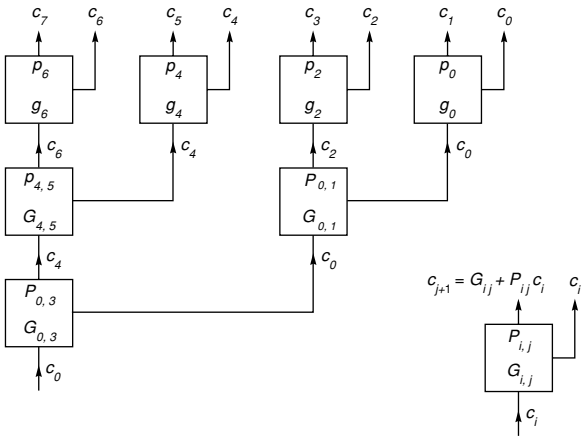


Figure H.16 Second part of carry-lookahead tree. Signals flow from the bottom to the top, combining with *P* and *G* to form the carries.

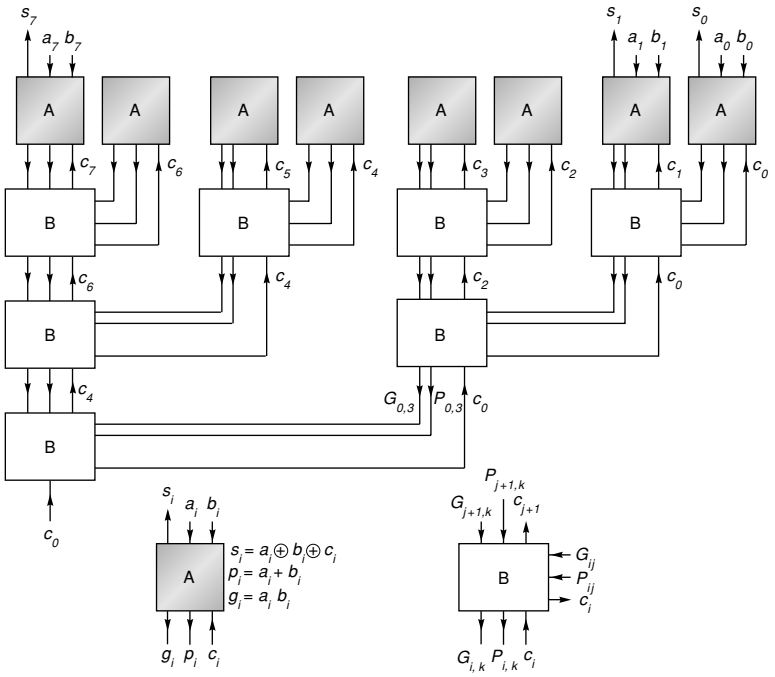


Figure H.17 Complete carry-lookahead tree adder. This is the combination of Figures H.15 and H.16. The numbers to be added enter at the top, flow to the bottom to combine with c_0, and then flow back up to compute the sum bits.

$2n$ cells, although in our layout they will take $n \log n$ space. The point is that a small investment in size pays off in a dramatic improvement in speed.

A number of technology-dependent modifications can improve CLAs. For example, if each node of the tree has three inputs instead of two, then the height of the tree will decrease from $\log_2 n$ to $\log_3 n$. Of course, the cells will be more complex and thus might operate more slowly, negating the advantage of the decreased height. For technologies where rippling works well, a hybrid design might be better. This is illustrated in Figure H.18. Carries ripple between adders at the top level, while the "B" boxes are the same as those in Figure H.17. This design will be faster if the time to ripple between four adders is faster than the time it takes to traverse a level of "B" boxes. (To make the pattern more clear, Figure H.18 shows a 16-bit adder, so the 8-bit adder of Figure H.17 corresponds to the right half of Figure H.18.)

Carry-Skip Adders

A *carry-skip adder* sits midway between a ripple-carry adder and a carry-lookahead adder, both in terms of speed and cost. (A carry-skip adder is not called a CSA, as that name is reserved for carry-save adders.) The motivation for this adder comes from examining the equations for P and G. For example,

$$P_{03} = p_0 p_1 p_2 p_3$$

$$G_{03} = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0$$

Computing P is much simpler than computing G, and a carry-skip adder only computes the P's. Such an adder is illustrated in Figure H.19. Carries begin rippling simultaneously through each block. If any block generates a carry, then the carry-out of a block will be true, even though the carry-in to the block may not be

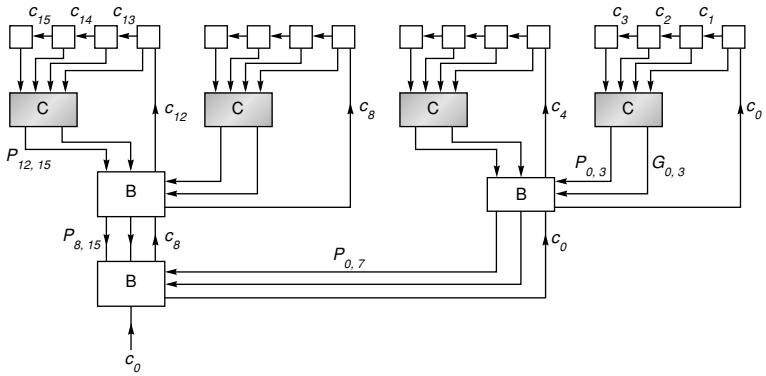


Figure H.18 Combination of CLA and ripple-carry adder. In the top row, carries ripple within each group of four boxes.

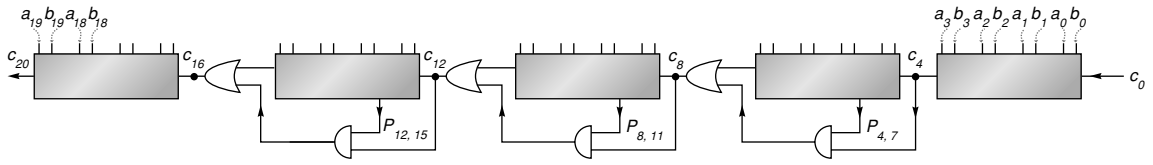


Figure H.19 Carry-skip adder. This is a 20-bit carry-skip adder ($n = 20$) with each block 4-bits wide ($k = 4$).

correct yet. If at the start of each add operation the carry-in to each block is 0, then no spurious carry-outs will be generated. Thus, the carry-out of each block can thus be thought of as if it were the G signal. Once the carry-out from the least-significant block is generated, it not only feeds into the next block, but is also fed through the AND gate with the P signal from that next block. If the carry-out and P signals are both true, then the carry *skips* the second block and is ready to feed into the third block, and so on. The carry-skip adder is only practical if the carry-in signals can be easily cleared at the start of each operation—for example, by precharging in CMOS.

To analyze the speed of a carry-skip adder, let's assume that it takes 1 time unit for a signal to pass through two logic levels. Then it will take k time units for a carry to ripple across a block of size k, and it will take 1 time unit for a carry to skip a block. The longest signal path in the carry-skip adder starts with a carry being generated at the 0th position. If the adder is n bits wide, then it takes k time units to ripple through the first block, $n/k - 2$ time units to skip blocks, and k more to ripple through the last block. To be specific: if we have a 20-bit adder broken into groups of 4 bits, it will take $4 + (20/4 - 2) + 4 = 11$ time units to perform an add. Some experimentation reveals that there are more efficient ways to divide 20 bits into blocks. For example, consider five blocks with the least-significant 2 bits in the first block, the next 5 bits in the second block, followed by blocks of size 6, 5, and 2. Then the add time is reduced to 9 time units. This illustrates an important general principle. For a carry-skip adder, making the interior blocks larger will speed up the adder. In fact, the same idea of varying the block sizes can sometimes speed up other adder designs as well. Because of the large amount of rippling, a carry-skip adder is most appropriate for technologies where rippling is fast.

Carry-Select Adder

A *carry-select adder* works on the following principle: Two additions are performed in parallel, one assuming the carry-in is 0 and the other assuming the carry-in is 1. When the carry-in is finally known, the correct sum (which has been precomputed) is simply selected. An example of such a design is shown in Figure H.20. An 8-bit adder is divided into two halves, and the carry-out from the lower half is used to select the sum bits from the upper half. If each block is computing its sum using rippling (a linear time algorithm), then the design in Figure H.20 is

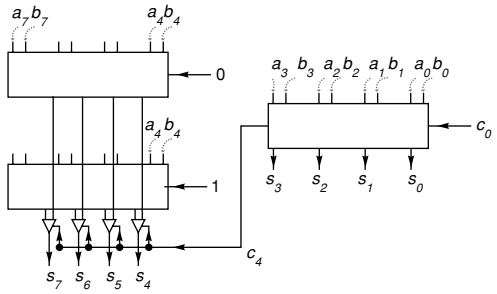


Figure H.20 Simple carry-select adder. At the same time that the sum of the low-order 4 bits is being computed, the high-order bits are being computed twice in parallel: once assuming that $c_4 = 0$ and once assuming $c_4 = 1$.

twice as fast at 50% more cost. However, note that the c_4 signal must drive many muxes, which may be very slow in some technologies. Instead of dividing the adder into halves, it could be divided into quarters for a still further speedup. This is illustrated in Figure H.21. If it takes k time units for a block to add k-bit numbers, and if it takes 1 time unit to compute the mux input from the two carry-out signals, then for optimal operation each block should be 1 bit wider than the next, as shown in Figure H.21. Therefore, as in the carry-skip adder, the best design involves variable-size blocks.

As a summary of this section, the asymptotic time and space requirements for the different adders are given in Figure H.22. (The times for carry-skip and carry-select come from a careful choice of block size. See Exercise H.26 for the carry-skip adder.) These different adders shouldn't be thought of as disjoint choices, but rather as building blocks to be used in constructing an adder. The utility of these different building blocks is highly dependent on the technology used. For example, the carry-select adder works well when a signal can drive many muxes, and the carry-skip adder is attractive in technologies where signals can be cleared at the start of each operation. Knowing the asymptotic behavior of adders is use-

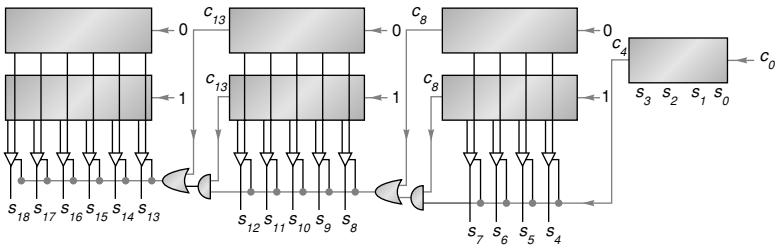


Figure H.21 Carry-select adder. As soon as the carry-out of the rightmost block is known, it is used to select the other sum bits.

Adder	Time	Space
Ripple	O(n)	O(n)
CLA	O(log n)	O(n log n)
Carry-skip	O($\sqrt{n}$)	O(n)
Carry-select	O($\sqrt{n}$)	O(n)

Figure H.22 Asymptotic time and space requirements for four different types of adders.

ful in understanding them, but relying too much on that behavior is a pitfall. The reason is that asymptotic behavior is only important as n grows very large. But n for an adder is the bits of precision, and double precision today is the same as it was 20 years ago—about 53 bits. Although it is true that as computers get faster, computations get longer—and thus have more rounding error, which in turn requires more precision—this effect grows very slowly with time.

H.9 Speeding Up Integer Multiplication and Division

The multiplication and division algorithms presented in Section H.2 are fairly slow, producing 1 bit per cycle (although that cycle might be a fraction of the CPU instruction cycle time). In this section we discuss various techniques for higher-performance multiplication and division, including the division algorithm used in the Pentium chip.

Shifting over Zeros

Although the technique of shifting over zeros is not currently used much, it is instructive to consider. It is distinguished by the fact that its execution time is operand dependent. Its lack of use is primarily attributable to its failure to offer enough speedup over bit-at-a-time algorithms. In addition, pipelining, synchronization with the CPU, and good compiler optimization are difficult with algorithms that run in variable time. In multiplication, the idea behind shifting over zeros is to add logic that detects when the low-order bit of the A register is 0 (see Figure H.2(a) on page H-4) and, if so, skips the addition step and proceeds directly to the shift step—hence the term *shifting over zeros*.

What about shifting for division? In nonrestoring division, an ALU operation (either an addition or subtraction) is performed at every step. There appears to be no opportunity for skipping an operation. But think about division this way: To compute a/b, subtract multiples of b from a, and then report how many subtractions were done. At each stage of the subtraction process the remainder must fit into the P register of Figure H.2(b) (page H-4). In the case when the remainder is a small positive number, you normally subtract b; but suppose instead you only

shifted the remainder and subtracted b the next time. As long as the remainder was sufficiently small (its high-order bit 0), after shifting it still would fit into the P register, and no information would be lost. However, this method does require changing the way we keep track of the number of times b has been subtracted from a. This idea usually goes under the name of *SRT division,* for Sweeney, Robertson, and Tocher, who independently proposed algorithms of this nature. The main extra complication of SRT division is that the quotient bits cannot be determined immediately from the sign of P at each step, as they can be in ordinary nonrestoring division.

More precisely, to divide a by b where a and b are n-bit numbers, load a and b into the A and B registers, respectively, of Figure H.2 (page H-4).

SRT Division

1. If B has k leading zeros when expressed using n bits, shift all the registers left k bits.
2. For $i = 0, n - 1$,
   (a) If the top three bits of P are equal, set $q_i = 0$ and shift (P,A) one bit left.
   (b) If the top three bits of P are not all equal and P is negative, set $q_i = -1$ (also written as $\overline{1}$), shift (P,A) one bit left, and add B.
   (c) Otherwise set $q_i = 1$, shift (P,A) one bit left, and subtract B.

   End loop
3. If the final remainder is negative, correct the remainder by adding B, and correct the quotient by subtracting 1 from q_0. Finally, the remainder must be shifted k bits right, where k is the initial shift.

A numerical example is given in Figure H.23. Although we are discussing integer division, it helps in explaining the algorithm to imagine the binary point just left of the most-significant bit. This changes Figure H.23 from $01000_2/0011_2$ to $0.1000_2/.0011_2$. Since the binary point is changed in both the numerator and denominator, the quotient is not affected. The (P,A) register pair holds the remainder and is a two's complement number. For example, if P contains 11110_2 and A = 0, then the remainder is $1.1110_2 = -1/8$. If r is the value of the remainder, then $-1 \le r < 1$.

Given these preliminaries, we can now analyze the SRT division algorithm. The first step of the algorithm shifts b so that $b \ge 1/2$. The rule for which ALU operation to perform is this: If $-1/4 \le r < 1/4$ (true whenever the top three bits of P are equal), then compute $2r$ by shifting (P,A) left one bit; else if $r < 0$ (and hence $r < -1/4$, since otherwise it would have been eliminated by the first condition), then compute $2r + b$ by shifting and then adding, else $r \ge 1/4$ and subtract b from $2r$. Using $b \ge 1/2$, it is easy to check that these rules keep $-1/2 \le r < 1/2$. For nonrestoring division, we only have $|r| \le b$, and we need P to be $n + 1$ bits wide. But for SRT division, the bound on r is tighter, namely, $-1/2 \le r < 1/2$. Thus, we can save a bit by eliminating the high-order bit of P (and b and the adder). In par-

P	A	
00000	1000	Divide 8 = 1000 by 3 = 0011. B contains 0011.
00010	0000	Step 1: B had two leading 0s, so shift left by 2. B now contains 1100.
		Step 2.1: Top three bits are equal. This is case (a), so
00100	000**0**	set $q_0 = 0$ and shift.
		Step 2.2: Top three bits not equal and P ≥ 0 is case (c), so
01000	00**01**	set $q_1 = 1$ and shift.
+ 10100		Subtract B.
11100	00**01**	Step 2.3: Top bits equal is case (a), so
11000	0**010**	set $q_2 = 0$ and shift.
		Step 2.4: Top three bits unequal is case (b), so
10000	**010$\overline{1}$**	set $q_3 = -1$ and shift.
+ 01100		Add B.
11100		Step 3. Remainder is negative so restore it and subtract 1 from *q*.
+ 01100		
01000		Must undo the shift in step 1, so right-shift by 2 to get true remainder.
		Remainder = 10, quotient = 010$\overline{1}$ – 1 = 0010.

Figure H.23 SRT division of $1000_2/0011_2$. The quotient bits are shown in bold, using the notation $\overline{1}$ for –1.

ticular, the test for equality of the top three bits of P becomes a test on just two bits.

The algorithm might change slightly in an implementation of SRT division. After each ALU operation, the P register can be shifted as many places as necessary to make either $r \geq 1/4$ or $r < -1/4$. By shifting k places, k quotient bits are set equal to zero all at once. For this reason SRT division is sometimes described as one that keeps the remainder normalized to $|r| \geq 1/4$.

Notice that the value of the quotient bit computed in a given step is based on which operation is performed in that step (which in turn depends on the result of the operation from the previous step). This is in contrast to nonrestoring division, where the quotient bit computed in the *i*th step depends on the result of the operation in the same step. This difference is reflected in the fact that when the final remainder is negative, the last quotient bit must be adjusted in SRT division, but not in nonrestoring division. However, the key fact about the quotient bits in SRT division is that they can include $\overline{1}$. Although Figure H.23 shows the quotient bits being stored in the low-order bits of A, an actual implementation can't do this because you can't fit the three values –1, 0, 1 into one bit. Furthermore, the quotient must be converted to ordinary two's complement in a full adder. A common way to do this is to accumulate the positive quotient bits in one register and the negative quotient bits in another, and then subtract the two registers after all the bits are known. Because there is more than one way to write a number in terms of the digits –1, 0, 1, SRT division is said to use a *redundant* quotient representation.

The differences between SRT division and ordinary nonrestoring division can be summarized as follows:

1. ALU decision rule: In nonrestoring division, it is determined by the sign of P; in SRT, it is determined by the two most-significant bits of P.

2. Final quotient: In nonrestoring division, it is immediate from the successive signs of P; in SRT, there are three quotient digits (1, 0, $\overline{1}$), and the final quotient must be computed in a full n-bit adder.
3. Speed: SRT division will be faster on operands that produce zero quotient bits.

The simple version of the SRT division algorithm given above does not offer enough of a speedup to be practical in most cases. However, later on in this section we will study variants of SRT division that are quite practical.

Speeding Up Multiplication with a Single Adder

As mentioned before, shifting-over-zero techniques are not used much in current hardware. We now discuss some methods that are in widespread use. Methods that increase the speed of multiplication can be divided into two classes: those that use a single adder and those that use multiple adders. Let's first discuss techniques that use a single adder.

In the discussion of addition we noted that, because of carry propagation, it is not practical to perform addition with two levels of logic. Using the cells of Figure H.17, adding two 64-bit numbers will require a trip through seven cells to compute the P's and G's, and seven more to compute the carry bits, which will require at least 28 logic levels. In the simple multiplier of Figure H.2 on page H-4, each multiplication step passes through this adder. The amount of computation in each step can be dramatically reduced by using *carry-save adders* (CSAs). A carry-save adder is simply a collection of n independent full adders. A multiplier using such an adder is illustrated in Figure H.24. Each circle marked "+" is a single-bit full adder, and each box represents one bit of a register. Each addition operation results in a pair of bits, stored in the sum and carry parts of P. Since each add is independent, only two logic levels are involved in the add—a vast improvement over 28.

To operate the multiplier in Figure H.24, load the sum and carry bits of P with zero and perform the first ALU operation. (If Booth recoding is used, it might be a subtraction rather than an addition.) Then shift the low-order sum bit of P into A, as well as shifting A itself. The $n - 1$ high-order bits of P don't need to be shifted because on the next cycle the sum bits are fed into the next lower-order adder. Each addition step is substantially increased in speed, since each add cell is working independently of the others, and no carry is propagated.

There are two drawbacks to carry-save adders. First, they require more hardware because there must be a copy of register P to hold the carry outputs of the adder. Second, after the last step, the high-order word of the result must be fed into an ordinary adder to combine the sum and carry parts. One way to accomplish this is by feeding the output of P into the adder used to perform the addition operation. Multiplying with a carry-save adder is sometimes called redundant multiplication because P is represented using two registers. Since there

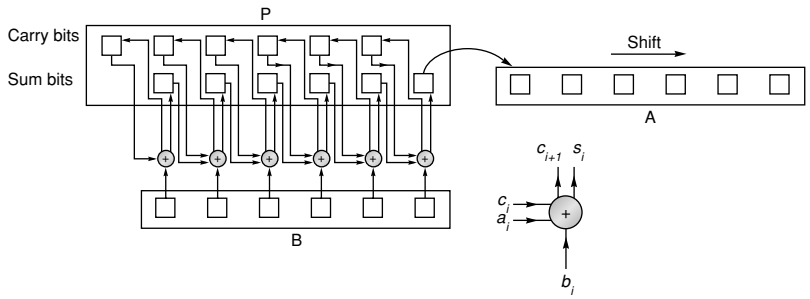


Figure H.24 Carry-save multiplier. Each circle represents a (3,2) adder working independently. At each step, the only bit of P that needs to be shifted is the low-order sum bit.

are many ways to represent P as the sum of two registers, this representation is redundant. The term *carry-propagate adder* (CPA) is used to denote an adder that is not a CSA. A propagate adder may propagate its carries using ripples, carry-lookahead, or some other method.

Another way to speed up multiplication without using extra adders is to examine k low-order bits of A at each step, rather than just one bit. This is often called *higher-radix multiplication*. As an example, suppose that $k = 2$. If the pair of bits is 00, add 0 to P; if it is 01, add B. If it is 10, simply shift b one bit left before adding it to P. Unfortunately, if the pair is 11, it appears we would have to compute $b + 2b$. But this can be avoided by using a higher-radix version of Booth recoding. Imagine A as a base 4 number: When the digit 3 appears, change it to $\overline{1}$ and add 1 to the next higher digit to compensate. An extra benefit of using this scheme is that just like ordinary Booth recoding, it works for negative as well as positive integers (Section H.2).

The precise rules for radix-4 Booth recoding are given in Figure H.25. At the ith multiply step, the two low-order bits of the A register contain a_{2i} and a_{2i+1}. These two bits, together with the bit just shifted out (a_{2i-1}), are used to select the multiple of b that must be added to the P register. A numerical example is given in Figure H.26. Another name for this multiplication technique is *overlapping triplets,* since it looks at 3 bits to determine what multiple of b to use, whereas ordinary Booth recoding looks at 2 bits.

Besides having more complex control logic, overlapping triplets also requires that the P register be 1 bit wider to accommodate the possibility of $2b$ or $-2b$ being added to it. It is possible to use a radix-8 (or even higher) version of Booth recoding. In that case, however, it would be necessary to use the multiple 3B as a potential summand. Radix-8 multipliers normally compute 3B once and for all at the beginning of a multiplication operation.

Low-order bits of A		Last bit shifted out	
$2i + 1$	$2i$	$2i - 1$	Multiple
0	0	0	0
0	0	1	$+b$
0	1	0	$+b$
0	1	1	$+2b$
1	0	0	$-2b$
1	0	1	$-b$
1	1	0	$-b$
1	1	1	0

Figure H.25 Multiples of *b* to use for radix-4 Booth recoding. For example, if the two low-order bits of the A register are both 1, and the last bit to be shifted out of the A register is 0, then the correct multiple is *–b,* obtained from the second-to-last row of the table.

```
   P      A    L
 00000   1001        Multiply –7 = 1001 times –5 = 1011. B contains 1011.
+ 11011              Low-order bits of A are 0, 1; L = 0, so add B.
 11011   1001
 11110   1110   0    Shift right by two bits, shifting in 1s on the left.
+ 01010              Low-order bits of A are 1, 0; L = 0, so add –2b.
 01000   1110   0
 00010   0011   1    Shift right by two bits.
                     Product is 35 = 0100011.
```

Figure H.26 Multiplication of –7 times –5 using radix-4 Booth recoding. The column labeled L contains the last bit shifted out the right end of A.

Faster Multiplication with Many Adders

If the space for many adders is available, then multiplication speed can be improved. Figure H.27 shows a simple array multiplier for multiplying two 5-bit numbers, using three CSAs and one propagate adder. Part (a) is a block diagram of the kind we will use throughout this section. Parts (b) and (c) show the adder in more detail. All the inputs to the adder are shown in (b); the actual adders with their interconnections are shown in (c). Each row of adders in (c) corresponds to a box in (a). The picture is "twisted" so that bits of the same significance are in the same column. In an actual implementation, the array would most likely be laid out as a square instead.

The array multiplier in Figure H.27 performs the same number of additions as the design in Figure H.24, so its latency is not dramatically different from that of a single carry-save adder. However, with the hardware in Figure H.27, multiplication can be pipelined, increasing the total throughput. On the other hand,

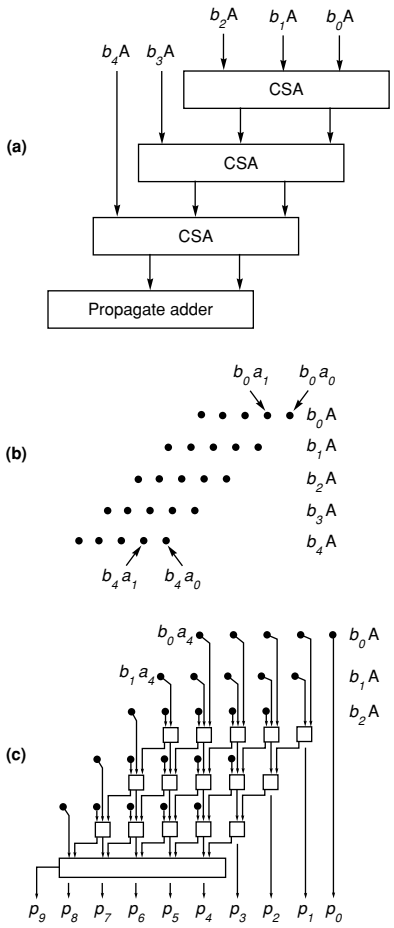


Figure H.27 An array multiplier. The 5-bit number in A is multiplied by $b_4b_3b_2b_1b_0$. Part (a) shows the block diagram, (b) shows the inputs to the array, and (c) expands the array to show all the adders.

although this level of pipelining is sometimes used in array processors, it is not used in any of the single-chip, floating-point accelerators discussed in Section H.10. Pipelining is discussed in general in Appendix A and by Kogge [1981] in the context of multipliers.

Sometimes the space budgeted on a chip for arithmetic may not hold an array large enough to multiply two double-precision numbers. In this case, a popular design is to use a two-pass arrangement such as the one shown in Figure H.28. The first pass through the array "retires" 5 bits of B. Then the result of this first pass is fed back into the top to be combined with the next three summands. The result of this second pass is then fed into a CPA. This design, however, loses the ability to be pipelined.

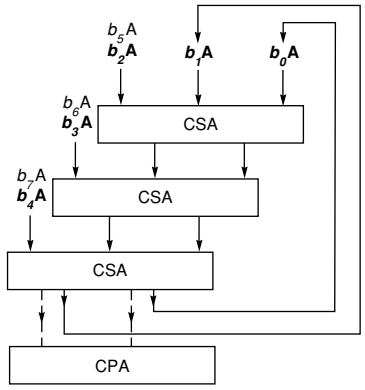


Figure H.28 Multipass array multiplier. Multiplies two 8-bit numbers with about half the hardware that would be used in a one-pass design like that of Figure H.27. At the end of the second pass, the bits flow into the CPA. The inputs used in the first pass are marked in bold.

If arrays require as many addition steps as the much cheaper arrangements in Figures H.2 and H.24, why are they so popular? First of all, using an array has a smaller latency than using a single adder—because the array is a combinational circuit, the signals flow through it directly without being clocked. Although the two-pass adder of Figure H.28 would normally still use a clock, the cycle time for passing through k arrays can be less than k times the clock that would be needed for designs like the ones in Figures H.2 or H.24. Second, the array is amenable to various schemes for further speedup. One of them is shown in Figure H.29. The idea of this design is that two adds proceed in parallel or, to put it another way, each stream passes through only half the adders. Thus, it runs at almost twice the speed of the multiplier in Figure H.27. This *even*/*odd* multiplier is popular in VLSI because of its regular structure. Arrays can also be speeded up using asynchronous logic. One of the reasons why the multiplier of Figure H.2 (page H-4) needs a clock is to keep the output of the adder from feeding back into the input of the adder before the output has fully stabilized. Thus, if the array in Figure H.28 is long enough so that no signal can propagate from the top through the bottom in the time it takes for the first adder to stabilize, it may be possible to avoid clocks altogether. Williams et al. [1987] discuss a design using this idea, although it is for dividers instead of multipliers.

The techniques of the previous paragraph still have a multiply time of O(n), but the time can be reduced to log n using a tree. The simplest tree would combine pairs of summands $b_0\text{A} \cdots b_{n-1}\text{A}$, cutting the number of summands from n to n/2. Then these n/2 numbers would be added in pairs again, reducing to n/4, and so on, and resulting in a single sum after log n steps. However, this simple binary tree idea doesn't map into full (3,2) adders, which reduce three inputs to two rather than reducing two inputs to one. A tree that does use full adders, known as a *Wallace tree,* is shown in Figure H.30. When computer arithmetic units were

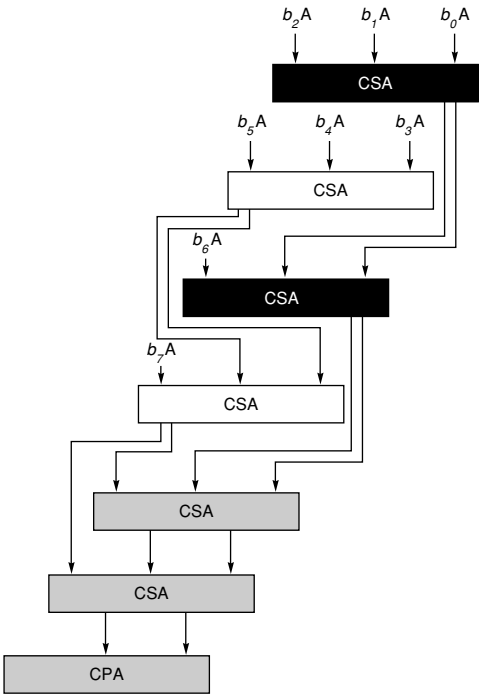


Figure H.29 Even/odd array. The first two adders work in parallel. Their results are fed into the third and fourth adders, which also work in parallel, and so on.

built out of MSI parts, a Wallace tree was the design of choice for high-speed multipliers. There is, however, a problem with implementing it in VLSI. If you try to fill in all the adders and paths for the Wallace tree of Figure H.30, you will discover that it does not have the nice, regular structure of Figure H.27. This is why VLSI designers have often chosen to use other log n designs such as the *binary tree multiplier*, which is discussed next.

The problem with adding summands in a binary tree is coming up with a (2,1) adder that combines two digits and produces a single-sum digit. Because of carries, this isn't possible using binary notation, but it can be done with some other representation. We will use the *signed-digit representation* 1, $\overline{1}$, and 0, which we used previously to understand Booth's algorithm. This representation has two costs. First, it takes 2 bits to represent each signed digit. Second, the algorithm for adding two signed-digit numbers a_i and b_i is complex and requires examining $a_i a_{i-1} a_{i-2}$ and $b_i b_{i-1} b_{i-2}$. Although this means you must look 2 bits back, in binary addition you might have to look an arbitrary number of bits back because of carries.

We can describe the algorithm for adding two signed-digit numbers as follows. First, compute sum and carry bits s_i and c_{i+1} using Figure H.31. Then compute the final sum as $s_i + c_i$. The tables are set up so that this final sum does not generate a carry.

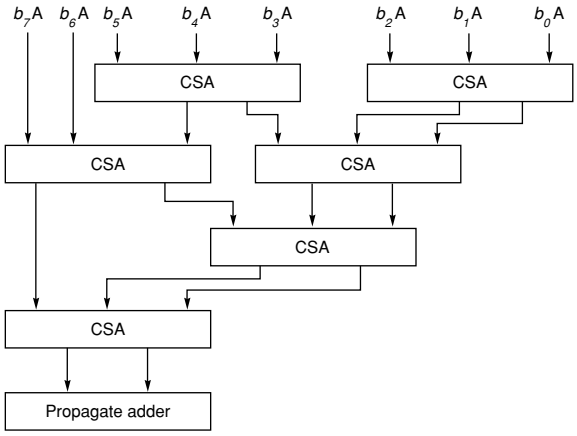


Figure H.30 Wallace tree multiplier. An example of a multiply tree that computes a product in 0(log *n*) steps.

$$\begin{array}{r} 1 \\ +\,1 \\ \hline 1\,0 \end{array} \qquad \begin{array}{r} 1 \\ +\,\bar{1} \\ \hline 0\,0 \end{array} \qquad \begin{array}{r} \bar{1} \\ +\,\bar{1} \\ \hline \bar{1}\,0 \end{array} \qquad \begin{array}{r} 0 \\ +\,0 \\ \hline 0\,0 \end{array} \qquad \begin{array}{r} 1\ x \\ +\,0\ y \\ \hline 1\,\bar{1} \\ 0\,1 \end{array} \begin{array}{l} \text{if } x \geq 0 \text{ and } y \geq 0 \\ \text{otherwise} \end{array} \qquad \begin{array}{r} \bar{1}\ x \\ +\,0\ y \\ \hline 0\,\bar{1} \\ \bar{1}\,1 \end{array} \begin{array}{l} \text{if } x \geq 0 \text{ and } y \geq 0 \\ \text{otherwise} \end{array}$$

Figure H.31 Signed-digit addition table. The leftmost sum shows that when computing 1 + 1, the sum bit is 0 and the carry bit is 1.

Example What is the sum of the signed-digit numbers $1\bar{1}0_2$ and 001_2?

Answer The two low-order bits sum to $0 + 1 = 1\bar{1}$, the next pair sums to $\bar{1} + 0 = 0\bar{1}$, and the high-order pair sums to $1 + 0 = 01$, so the sum is $1\bar{1} + 0\bar{1}0 + 0100 = 10\bar{1}_2$.

This, then, defines a (2,1) adder. With this in hand, we can use a straightforward binary tree to perform multiplication. In the first step it adds $b_0\text{A} + b_1\text{A}$ in parallel with $b_2\text{A} + b_3\text{A}, \ldots, b_{n-2}\text{A} + b_{n-1}\text{A}$. The next step adds the results of these sums in pairs, and so on. Although the final sum must be run through a carry-propagate adder to convert it from signed-digit form to two's complement, this final add step is necessary in any multiplier using CSAs.

To summarize, both Wallace trees and signed-digit trees are log *n* multipliers. The Wallace tree uses fewer gates but is harder to lay out. The signed-digit tree has a more regular structure, but requires 2 bits to represent each digit and has more complicated add logic. As with adders, it is possible to combine different multiply techniques. For example, Booth recoding and arrays can be combined.

In Figure H.27 instead of having each input be b_iA, we could have it be $b_i b_{i-1}$A. To avoid having to compute the multiple 3*b,* we can use Booth recoding.

Faster Division with One Adder

The two techniques we discussed for speeding up multiplication with a single adder were carry-save adders and higher-radix multiplication. However, there is a difficulty when trying to utilize these approaches to speed up nonrestoring division. If the adder in Figure H.2(b) on page H-4 is replaced with a carry-save adder, then P will be replaced with two registers, one for the sum bits and one for the carry bits (compare with the multiplier in Figure H.24). At the end of each cycle, the sign of P is uncertain (since P is the unevaluated sum of the two registers), yet it is the sign of P that is used to compute the quotient digit and decide the next ALU operation. When a higher radix is used, the problem is deciding what value to subtract from P. In the paper-and-pencil method, you have to guess the quotient digit. In binary division there are only two possibilities. We were able to finesse the problem by initially guessing one and then adjusting the guess based on the sign of P. This doesn't work in higher radices because there are more than two possible quotient digits, rendering quotient selection potentially quite complicated: You would have to compute all the multiples of *b* and compare them to P.

Both the carry-save technique and higher-radix division can be made to work if we use a redundant quotient representation. Recall from our discussion of SRT division (page H-46) that by allowing the quotient digits to be −1, 0, or 1, there is often a choice of which one to pick. The idea in the previous algorithm was to choose 0 whenever possible, because that meant an ALU operation could be skipped. In carry-save division, the idea is that, because the remainder (which is the value of the (P,A) register pair) is not known exactly (being stored in carry-save form), the exact quotient digit is also not known. But thanks to the redundant representation, the remainder doesn't have to be known precisely in order to pick a quotient digit. This is illustrated in Figure H.32, where the *x* axis represents r_i,

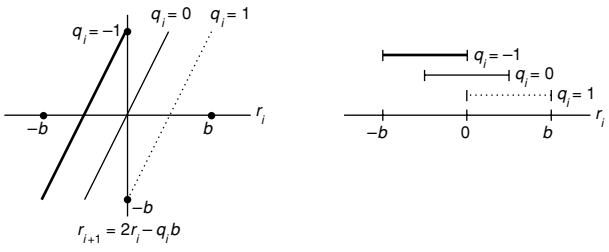


Figure H.32 Quotient selection for radix-2 division. The *x*-axis represents the *i*th remainder, which is the quantity in the (P,A) register pair. The *y*-axis shows the value of the remainder after one additional divide step. Each bar on the right-hand graph gives the range of r_i values for which it is permissible to select the associated value of q_i.

the remainder after i steps. The line labeled $q_i = 1$ shows the value that r_{i+1} would be if we chose $q_i = 1$, and similarly for the lines $q_i = 0$ and $q_i = -1$. We can choose any value for q_i, as long as $r_{i+1} = 2r_i - q_i b$ satisfies $|r_{i+1}| \leq b$. The allowable ranges are shown in the right half of Figure H.32. This shows that you don't need to know the precise value of r_i in order to choose a quotient digit q_i. You only need to know that r lies in an interval small enough to fit entirely within one of the overlapping bars shown in the right half of Figure H.32.

This is the basis for using carry-save adders. Look at the high-order bits of the carry-save adder and sum them in a propagate adder. Then use this approximation of r (together with the divisor, b) to compute q_i, usually by means of a lookup table. The same technique works for higher-radix division (whether or not a carry-save adder is used). The high-order bits P can be used to index a table that gives one of the allowable quotient digits.

The design challenge when building a high-speed SRT divider is figuring out how many bits of P and B need to be examined. For example, suppose that we take a radix of 4, use quotient digits of 2, 1, 0, $\overline{1}$, $\overline{2}$, but have a propagate adder. How many bits of P and B need to be examined? Deciding this involves two steps. For ordinary radix-2 nonrestoring division, because at each stage $|r| \leq b$, the P buffer won't overflow. But for radix 4, $r_{i+1} = 4r_i - q_i b$ is computed at each stage, and if r_i is near b, then $4r_i$ will be near $4b$, and even the largest quotient digit will not bring r back to the range $|r_{i+1}| \leq b$. In other words, the remainder might grow without bound. However, restricting $|r_i| \leq 2b/3$ makes it easy to check that r_i will stay bounded.

After figuring out the bound that r_i must satisfy, we can draw the diagram in Figure H.33, which is analogous to Figure H.32. For example, the diagram shows that if r_i is between $(1/12)b$ and $(5/12)b$, we can pick $q = 1$, and so on. Or to put it another way, if r/b is between 1/12 and 5/12, we can pick $q = 1$. Suppose the

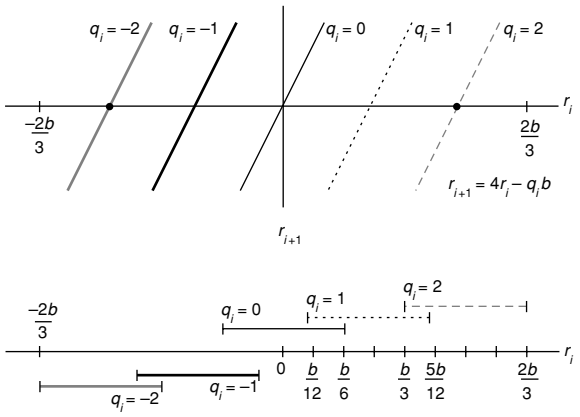


Figure H.33 Quotient selection for radix-4 division with quotient digits –2, –1, 0, 1, 2.

divider examines 5 bits of P (including the sign bit) and 4 bits of *b* (ignoring the sign, since it is always nonnegative). The interesting case is when the high bits of P are 00011*xxx*···, while the high bits of *b* are 1001*xxx*···. Imagine the binary point at the left end of each register. Since we truncated, *r* (the value of P concatenated with A) could have a value from 0.0011_2 to 0.0100_2, and *b* could have a value from $.1001_2$ to $.1010_2$. Thus *r*/*b* could be as small as $0.0011_2/.1010_2$ or as large as $0.0100_2/.1001_2$. But $0.0011_2/.1010_2 = 3/10 < 1/3$ would require a quotient bit of 1, while $0.0100_2/.1001_2 = 4/9 > 5/12$ would require a quotient bit of 2. In other words, 5 bits of P and 4 bits of *b* aren't enough to pick a quotient bit. It turns out that 6 bits of P and 4 bits of *b* are enough. This can be verified by writing a simple program that checks all the cases. The output of such a program is shown in Figure H.34.

b	Range of P		*q*	*b*	Range of P		*q*
8	−12	−7	−2	12	−18	−10	−2
8	−6	−3	−1	12	−10	−4	−1
8	−2	1	0	12	−4	3	0
8	2	5	1	12	3	9	1
8	6	11	2	12	9	17	2
9	−14	−8	−2	13	−19	−11	−2
9	−7	−3	−1	13	−10	−4	−1
9	−3	2	0	13	−4	3	0
9	2	6	1	13	3	9	1
9	7	13	2	13	10	18	2
10	−15	−9	−2	14	−20	−11	−2
10	−8	−3	−1	14	−11	−4	−1
10	−3	2	0	14	−4	3	0
10	2	7	1	14	3	10	1
10	8	14	2	14	10	19	2
11	−16	−9	−2	15	−22	−12	−2
11	−9	−3	−1	15	−12	−4	−1
11	−3	2	0	15	−5	4	0
11	2	8	1	15	3	11	1
11	8	15	2	15	11	21	2

Figure H.34 Quotient digits for radix-4 SRT division with a propagate adder. The top row says that if the high-order 4 bits of *b* are $1000_2 = 8$, and if the top 6 bits of P are between $110100_2 = -12$ and $111001_2 = -7$, then −2 is a valid quotient digit.

P	A		
000000000	10010101	Divide 149 by 5. B contains 00000101.	
000010010	10100000	Step 1:	B had 5 leading 0s, so shift left by 5. B now contains 10100000, so use $b = 10$ section of table.
		Step 2.1:	Top 6 bits of P are 2, so shift left by 2. From table, can pick q to be
001001010	1000000		0 or 1. Choose $q_0 = 0$.
		Step 2.2:	Top 6 bits of P are 9, so
100101010	000002		shift left 2. $q_1 = 2$.
+ 011000000			Subtract $2b$.
111101010	000002	Step 2.3:	Top bits = –3, so
110101000	00020		shift left 2. Can pick 0 or –1 for q, pick $q_2 = 0$.
		Step 2.4:	Top bits = –11, so
010100000	0202		shift left 2. $q_3 = -2$.
+ 101000000			Add $2b$.
111100000		Step 3:	Remainder is negative, so restore
+ 010100000			by adding b and subtract 1 from q.
010000000		Answer:	$q = 020\overline{2} - 1 = 29$.
			To get remainder, undo shift in step 1 so remainder = 010000000 >> 5 = 4.

Figure H.35 Example of radix-4 SRT division. Division of 149 by 5.

Example Using 8-bit registers, compute 149/5 using radix-4 SRT division.

Answer Follow the SRT algorithm on page H-46, but replace the quotient selection rule in step 2 with one that uses Figure H.34. See Figure H.35.

The Pentium uses a radix-4 SRT division algorithm like the one just presented, except that it uses a carry-save adder. Exercises H.34(c) and H.35 explore this in detail. Although these are simple cases, all SRT analyses proceed in the same way. First compute the range of r_i, then plot r_i against r_{i+1} to find the quotient ranges, and finally write a program to compute how many bits are necessary. (It is sometimes also possible to compute the required number of bits analytically.) Various details need to be considered in building a practical SRT divider. For example, the quotient lookup table has a fairly regular structure, which means it is usually cheaper to encode it as a PLA rather than in ROM. For more details about SRT division, see Burgess and Williams [1995].

H.10 Putting It All Together

In this section, we will compare the Weitek 3364, the MIPS R3010, and the Texas Instruments 8847 (see Figures H.36 and H.37). In many ways, these are ideal chips to compare. They each implement the IEEE standard for addition, subtraction, multiplication, and division on a single chip. All were introduced in 1988 and run with a cycle time of about 40 nanoseconds. However, as we will see, they use quite different algorithms. The Weitek chip is well described in Birman et al.

Features	MIPS R3010	Weitek 3364	TI 8847
Clock cycle time (ns)	40	50	30
Size (mil^2)	114,857	147,600	156,180
Transistors	75,000	165,000	180,000
Pins	84	168	207
Power (watts)	3.5	1.5	1.5
Cycles/add	2	2	2
Cycles/mult	5	2	3
Cycles/divide	19	17	11
Cycles/square root	–	30	14

Figure H.36 Summary of the three floating-point chips discussed in this section. The cycle times are for production parts available in June 1989. The cycle counts are for double-precision operations.

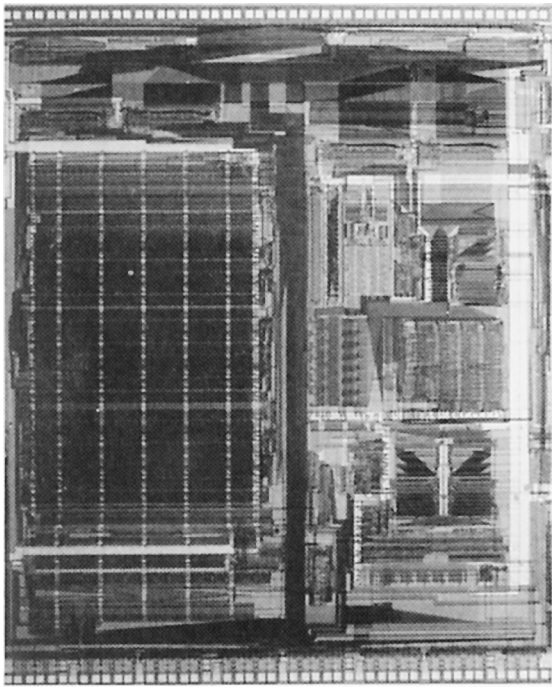
[1990], the MIPS chip is described in less detail in Rowen, Johnson, and Ries [1988], and details of the TI chip can be found in Darley et al. [1989].

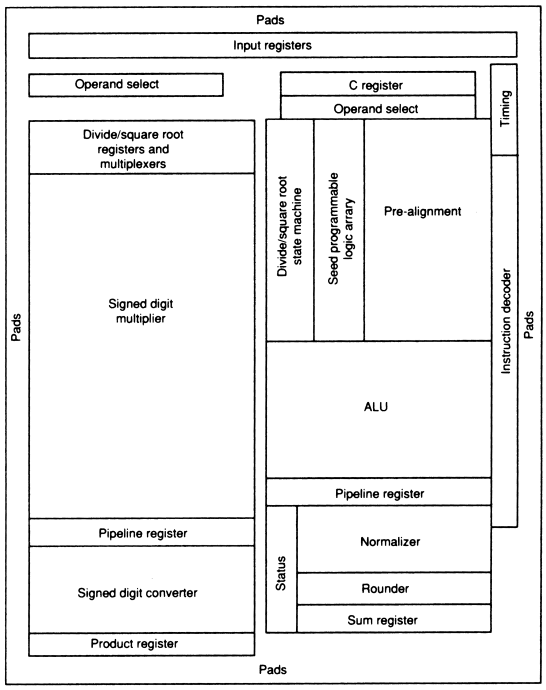
These three chips have a number of things in common. They perform addition and multiplication in parallel, and they implement neither extended precision nor a remainder step operation. (Recall from Section H.6 that it is easy to implement the IEEE remainder function in software if a remainder step instruction is available.) The designers of these chips probably decided not to provide extended precision because the most influential users are those who run portable codes, which can't rely on extended precision. However, as we have seen, extended precision can make for faster and simpler math libraries.

In the summary of the three chips given in Figure H.36, note that a higher transistor count generally leads to smaller cycle counts. Comparing the cycles/op numbers needs to be done carefully, because the figures for the MIPS chip are those for a complete system (R3000/3010 pair), while the Weitek and TI numbers are for stand-alone chips and are usually larger when used in a complete system.

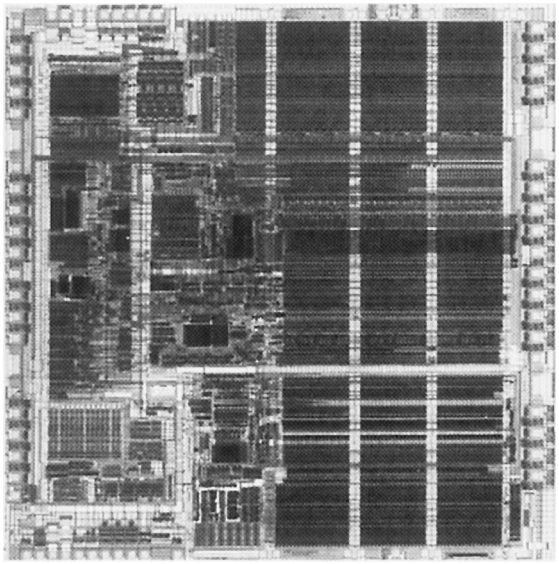
The MIPS chip has the fewest transistors of the three. This is reflected in the fact that it is the only chip of the three that does not have any pipelining or hardware square root. Further, the multiplication and addition operations are not completely independent because they share the carry-propagate adder that performs the final rounding (as well as the rounding logic).

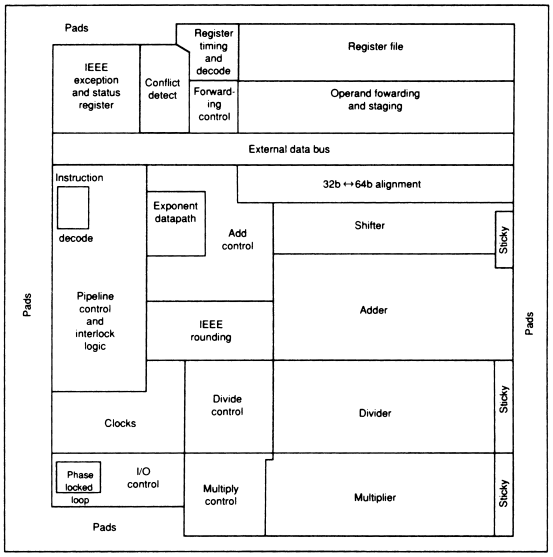
Addition on the R3010 uses a mixture of ripple, CLA, and carry-select. A carry-select adder is used in the fashion of Figure H.20 (page H-44). Within each half, carries are propagated using a hybrid ripple-CLA scheme of the type indicated in Figure H.18 (page H-42). However, this is further tuned by varying the size of each block, rather than having each fixed at 4 bits (as they are in Figure H.18). The multiplier is midway between the designs of Figures H.2 (page H-4) and H.27 (page H-51). It has an array just large enough so that output can be fed back into the input without having to be clocked. Also, it uses radix-4 Booth





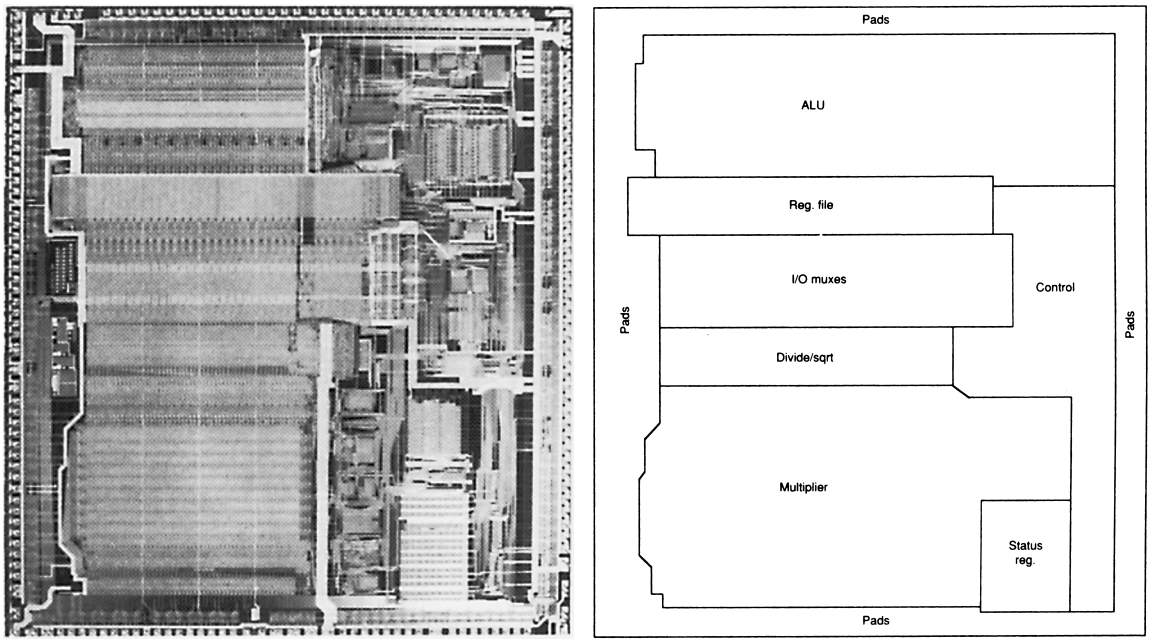
TI 8847





MIPS R3010

Figure H.37 Chip layout for the TI 8847, MIPS R3010, and Weitek 3364. In the left-hand columns are the photomicrographs; the right-hand columns show the corresponding floor plans.



Weitek 3364

Figure H.37 (*Continued.*)

recoding and the even/odd technique of Figure H.29 (page H-53). The R3010 can do a divide and multiply in parallel (like the Weitek chip but unlike the TI chip). The divider is a radix-4 SRT method with quotient digits –2, –1, 0, 1, and 2, and is similar to that described in Taylor [1985]. Double-precision division is about four times slower than multiplication. The R3010 shows that for chips using an O(n) multiplier, an SRT divider can operate fast enough to keep a reasonable ratio between multiply and divide.

The Weitek 3364 has independent add, multiply, and divide units. It also uses radix-4 SRT division. However, the add and multiply operations on the Weitek chip are pipelined. The three addition stages are (1) exponent compare, (2) add followed by shift (or vice versa), and (3) final rounding. Stages (1) and (3) take only a half-cycle, allowing the whole operation to be done in two cycles, even though there are three pipeline stages. The multiplier uses an array of the style of Figure H.28 but uses radix-8 Booth recoding, which means it must compute 3 times the multiplier. The three multiplier pipeline stages are (1) compute 3b, (2) pass through array, and (3) final carry-propagation add and round. Single precision passes through the array once, double precision twice. Like addition, the latency is two cycles.

The Weitek chip uses an interesting addition algorithm. It is a variant on the carry-skip adder pictured in Figure H.19 (page H-43). However, P_{ij}, which is the

logical AND of many terms, is computed by rippling, performing one AND per ripple. Thus, while the carries propagate left within a block, the value of P_{ij} is propagating right within the next block, and the block sizes are chosen so that both waves complete at the same time. Unlike the MIPS chip, the 3364 has hardware square root, which shares the divide hardware. The ratio of double-precision multiply to divide is 2:17. The large disparity between multiply and divide is due to the fact that multiplication uses radix-8 Booth recoding, while division uses a radix-4 method. In the MIPS R3010, multiplication and division use the same radix.

The notable feature of the TI 8847 is that it does division by iteration (using the Goldschmidt algorithm discussed in Section H.6). This improves the speed of division (the ratio of multiply to divide is 3:11), but means that multiplication and division cannot be done in parallel as on the other two chips. Addition has a two-stage pipeline. Exponent compare, fraction shift, and fraction addition are done in the first stage, normalization and rounding in the second stage. Multiplication uses a binary tree of signed-digit adders and has a three-stage pipeline. The first stage passes through the array, retiring half the bits; the second stage passes through the array a second time; and the third stage converts from signed-digit form to two's complement. Since there is only one array, a new multiply operation can only be initiated in every other cycle. However, by slowing down the clock, two passes through the array can be made in a single cycle. In this case, a new multiplication can be initiated in each cycle. The 8847 adder uses a carry-select algorithm rather than carry-lookahead. As mentioned in Section H.6, the TI carries 60 bits of precision in order to do correctly rounded division.

These three chips illustrate the different trade-offs made by designers with similar constraints. One of the most interesting things about these chips is the diversity of their algorithms. Each uses a different add algorithm, as well as a different multiply algorithm. In fact, Booth recoding is the only technique that is universally used by all the chips.

H.11 Fallacies and Pitfalls

Fallacy *Underflows rarely occur in actual floating-point application code.*

Although most codes rarely underflow, there are actual codes that underflow frequently. SDRWAVE [Kahaner 1988], which solves a one-dimensional wave equation, is one such example. This program underflows quite frequently, even when functioning properly. Measurements on one machine show that adding hardware support for gradual underflow would cause SDRWAVE to run about 50% faster.

Fallacy *Conversions between integer and floating point are rare.*

In fact, in spice they are as frequent as divides. The assumption that conversions are rare leads to a mistake in the SPARC version 8 instruction set, which does not provide an instruction to move from integer registers to floating-point registers.

Pitfall *Don't increase the speed of a floating-point unit without increasing its memory bandwidth.*

A typical use of a floating-point unit is to add two vectors to produce a third vector. If these vectors consist of double-precision numbers, then each floating-point add will use three operands of 64 bits each, or 24 bytes of memory. The memory bandwidth requirements are even greater if the floating-point unit can perform addition and multiplication in parallel (as most do).

Pitfall *–x is not the same as 0 – x.*

This is a fine point in the IEEE standard that has tripped up some designers. Because floating-point numbers use the sign magnitude system, there are two zeros, +0 and –0. The standard says that $0 - 0 = +0$, whereas $-(0) = -0$. Thus $-x$ is not the same as $0 - x$ when $x = 0$.

H.12 Historical Perspective and References

The earliest computers used fixed point rather than floating point. In "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument," Burks, Goldstine, and von Neumann [1946] put it like this:

> There appear to be two major purposes in a "floating" decimal point system both of which arise from the fact that the number of digits in a word is a constant fixed by design considerations for each particular machine. The first of these purposes is to retain in a sum or product as many significant digits as possible and the second of these is to free the human operator from the burden of estimating and inserting into a problem "scale factors" — multiplicative constants which serve to keep numbers within the limits of the machine.
>
> There is, of course, no denying the fact that human time is consumed in arranging for the introduction of suitable scale factors. We only argue that the time so consumed is a very small percentage of the total time we will spend in preparing an interesting problem for our machine. The first advantage of the floating point is, we feel, somewhat illusory. In order to have such a floating point, one must waste memory capacity which could otherwise be used for carrying more digits per word. It would therefore seem to us not at all clear whether the modest advantages of a floating binary point offset the loss of memory capacity and the increased complexity of the arithmetic and control circuits.

This enables us to see things from the perspective of early computer designers, who believed that saving computer time and memory were more important than saving programmer time.

The original papers introducing the Wallace tree, Booth recoding, SRT division, overlapped triplets, and so on, are reprinted in Swartzlander [1990]. A good explanation of an early machine (the IBM 360/91) that used a pipelined Wallace tree, Booth recoding, and iterative division is in Anderson et al. [1967]. A discussion of the average time for single-bit SRT division is in Freiman [1961]; this is one of the few interesting historical papers that does not appear in Swartzlander.

The standard book of Mead and Conway [1980] discouraged the use of CLAs as not being cost-effective in VLSI. The important paper by Brent and Kung [1982] helped combat that view. An example of a detailed layout for CLAs can be found in Ngai and Irwin [1985] or in Weste and Eshraghian [1993], and a more theoretical treatment is given by Leighton [1992]. Takagi, Yasuura, and Yajima [1985] provide a detailed description of a signed-digit tree multiplier.

Before the ascendancy of IEEE arithmetic, many different floating-point formats were in use. Three important ones were used by the IBM 370, the DEC VAX, and the Cray. Here is a brief summary of these older formats. The VAX format is closest to the IEEE standard. Its single-precision format (F format) is like IEEE single precision in that it has a hidden bit, 8 bits of exponent, and 23 bits of fraction. However, it does not have a sticky bit, which causes it to round halfway cases up instead of to even. The VAX has a slightly different exponent range from IEEE single: $E_{\min}$ is −128 rather than −126 as in IEEE, and $E_{\max}$ is 126 instead of 127. The main differences between VAX and IEEE are the lack of special values and gradual underflow. The VAX has a reserved operand, but it works like a signaling NaN: It traps whenever it is referenced. Originally, the VAX's double precision (D format) also had 8 bits of exponent. However, as this is too small for many applications, a G format was added; like the IEEE standard, this format has 11 bits of exponent. The VAX also has an H format, which is 128 bits long.

The IBM 370 floating-point format uses base 16 rather than base 2. This means it cannot use a hidden bit. In single precision, it has 7 bits of exponent and 24 bits (6 hex digits) of fraction. Thus, the largest representable number is $16^{2^7} = 2^{4 \times 2^7} = 2^{2^9}$, compared with 2^{2^8} for IEEE. However, a number that is normalized in the hexadecimal sense only needs to have a nonzero leading digit. When interpreted in binary, the three most-significant bits could be zero. Thus, there are potentially fewer than 24 bits of significance. The reason for using the higher base was to minimize the amount of shifting required when adding floating-point numbers. However, this is less significant in current machines, where the floating-point add time is usually fixed independently of the operands. Another difference between 370 arithmetic and IEEE arithmetic is that the 370 has neither a round digit nor a sticky digit, which effectively means that it truncates rather than rounds. Thus, in many computations, the result will systematically be too small. Unlike the VAX and IEEE arithmetic, every bit pattern is a valid number. Thus, library routines must establish conventions for what to return in case of errors. In the IBM FORTRAN library, for example, $\sqrt{-4}$ returns 2!

Arithmetic on Cray computers is interesting because it is driven by a motivation for the highest possible floating-point performance. It has a 15-bit exponent field and a 48-bit fraction field. Addition on Cray computers does not have a guard digit, and multiplication is even less accurate than addition. Thinking of

multiplication as a sum of p numbers, each $2p$ bits long, Cray computers drop the low-order bits of each summand. Thus, analyzing the exact error characteristics of the multiply operation is not easy. Reciprocals are computed using iteration, and division of a by b is done by multiplying a times $1/b$. The errors in multiplication and reciprocation combine to make the last three bits of a divide operation unreliable. At least Cray computers serve to keep numerical analysts on their toes!

The IEEE standardization process began in 1977, inspired mainly by W. Kahan and based partly on Kahan's work with the IBM 7094 at the University of Toronto [Kahan 1968]. The standardization process was a lengthy affair, with gradual underflow causing the most controversy. (According to Cleve Moler, visitors to the United States were advised that the sights not to be missed were Las Vegas, the Grand Canyon, and the IEEE standards committee meeting.) The standard was finally approved in 1985. The Intel 8087 was the first major commercial IEEE implementation and appeared in 1981, before the standard was finalized. It contains features that were eliminated in the final standard, such as projective bits. According to Kahan, the length of double-extended precision was based on what could be implemented in the 8087. Although the IEEE standard was not based on any existing floating-point system, most of its features were present in some other system. For example, the CDC 6600 reserved special bit patterns for INDEFINITE and INFINITY, while the idea of denormal numbers appears in Goldberg [1967] as well as in Kahan [1968]. Kahan was awarded the 1989 Turing prize in recognition of his work on floating point.

Although floating point rarely attracts the interest of the general press, newspapers were filled with stories about floating-point division in November 1994. A bug in the division algorithm used on all of Intel's Pentium chips had just come to light. It was discovered by Thomas Nicely, a math professor at Lynchburg College in Virginia. Nicely found the bug when doing calculations involving reciprocals of prime numbers. News of Nicely's discovery first appeared in the press on the front page of the November 7 issue of *Electronic Engineering Times*. Intel's immediate response was to stonewall, asserting that the bug would only affect theoretical mathematicians. Intel told the press, "This doesn't even qualify as an errata . . . even if you're an engineer, you're not going to see this."

Under more pressure, Intel issued a white paper, dated November 30, explaining why they didn't think the bug was significant. One of their arguments was based on the fact that if you pick two floating-point numbers at random and divide one into the other, the chance that the resulting quotient will be in error is about 1 in 9 billion. However, Intel neglected to explain why they thought that the typical customer accessed floating-point numbers randomly.

Pressure continued to mount on Intel. One sore point was that Intel had known about the bug before Nicely discovered it, but had decided not to make it public. Finally, on December 20, Intel announced that they would unconditionally replace any Pentium chip that used the faulty algorithm and that they would take an unspecified charge against earnings, which turned out to be $300 million.

The Pentium uses a simple version of SRT division as discussed in Section H.9. The bug was introduced when they converted the quotient lookup table to a PLA. Evidently there were a few elements of the table containing the quotient

digit 2 that Intel thought would never be accessed, and they optimized the PLA design using this assumption. The resulting PLA returned 0 rather than 2 in these situations. However, those entries were really accessed, and this caused the division bug. Even though the effect of the faulty PLA was to cause 5 out of 2048 table entries to be wrong, the Pentium only computes an incorrect quotient 1 out of 9 billion times on random inputs. This is explored in Exercise H.34.

References

Anderson, S. F., J. G. Earle, R. E. Goldschmidt, and D. M. Powers [1967]. "The IBM System/360 Model 91: Floating-point execution unit," *IBM J. Research and Development* 11, 34–53. Reprinted in Swartzlander [1990].
Good description of an early high-performance floating-point unit that used a pipelined Wallace tree multiplier and iterative division.

Bell, C. G., and A. Newell [1971]. *Computer Structures: Readings and Examples,* McGraw-Hill, New York.

Birman, M., A. Samuels, G. Chu, T. Chuk, L. Hu, J. McLeod, and J. Barnes [1990]. "Developing the WRL3170/3171 SPARC floating-point coprocessors," *IEEE Micro* 10:1, 55–64.
These chips have the same floating-point core as the Weitek 3364, and this paper has a fairly detailed description of that floating-point design.

Brent, R. P., and H. T. Kung [1982]. "A regular layout for parallel adders," *IEEE Trans. on Computers* C-31, 260–264.
This is the paper that popularized CLAs in VLSI.

Burgess, N., and T. Williams [1995]. "Choices of operand truncation in the SRT division algorithm," *IEEE Trans. on Computers* 44:7.
Analyzes how many bits of divisor and remainder need to be examined in SRT division.

Burks, A. W., H. H. Goldstine, and J. von Neumann [1946]. "Preliminary discussion of the logical design of an electronic computing instrument," Report to the U.S. Army Ordnance Department, p. 1; also appears in *Papers of John von Neumann*, W. Aspray and A. Burks, eds., MIT Press, Cambridge, Mass., and Tomash Publishers, Los Angeles, 1987, 97–146.

Cody, W. J., J. T. Coonen, D. M. Gay, K. Hanson, D. Hough, W. Kahan, R. Karpinski, J. Palmer, F. N. Ris, and D. Stevenson [1984]. "A proposed radix- and word-length-independent standard for floating-point arithmetic," *IEEE Micro* 4:4, 86–100.
Contains a draft of the 854 standard, which is more general than 754. The significance of this article is that it contains commentary on the standard, most of which is equally relevant to 754. However, be aware that there are some differences between this draft and the final standard.

Coonen, J. [1984]. *Contributions to a Proposed Standard for Binary Floating-Point Arithmetic,* Ph.D. thesis, Univ. of Calif., Berkeley.
The only detailed discussion of how rounding modes can be used to implement efficient binary decimal conversion.

Darley, H. M., et al. [1989]. "Floating point/integer processor with divide and square root functions," U.S. Patent 4,878,190, October 31, 1989.
Pretty readable as patents go. Gives a high-level view of the TI 8847 chip, but doesn't have all the details of the division algorithm.

Demmel, J. W., and X. Li [1994]. "Faster numerical algorithms via exception handling," *IEEE Trans. on Computers* 43:8, 983–992.
A good discussion of how the features unique to IEEE floating point can improve the performance of an important software library.

Freiman, C. V. [1961]. "Statistical analysis of certain binary division algorithms," *Proc. IRE* 49:1, 91–103.
Contains an analysis of the performance of shifting-over-zeros SRT division algorithm.

Goldberg, D. [1991]. "What every computer scientist should know about floating-point arithmetic," *Computing Surveys* 23:1, 5–48.
Contains an in-depth tutorial on the IEEE standard from the software point of view.

Goldberg, I. B. [1967]. "27 bits are not enough for 8-digit accuracy," *Comm. ACM* 10:2, 105–106.
This paper proposes using hidden bits and gradual underflow.

Gosling, J. B. [1980]. *Design of Arithmetic Units for Digital Computers,* Springer-Verlag, New York.
A concise, well-written book, although it focuses on MSI designs.

Hamacher, V. C., Z. G. Vranesic, and S. G. Zaky [1984]. *Computer Organization,* 2nd ed., McGraw-Hill, New York.
Introductory computer architecture book with a good chapter on computer arithmetic.

Hwang, K. [1979]. *Computer Arithmetic: Principles, Architecture, and Design,* Wiley, New York.
This book contains the widest range of topics of the computer arithmetic books.

IEEE [1985]. "IEEE standard for binary floating-point arithmetic," *SIGPLAN Notices* 22:2, 9–25.
IEEE 754 is reprinted here.

Kahan, W. [1968]. "7094-II system support for numerical analysis," *SHARE Secretarial Distribution* SSD-159.
This system had many features that were incorporated into the IEEE floating-point standard.

Kahaner, D. K. [1988]. "Benchmarks for 'real' programs," *SIAM News* (November).
The benchmark presented in this article turns out to cause many underflows.

Knuth, D. [1981]. *The Art of Computer Programming,* vol. II, 2nd ed., Addison-Wesley, Reading, Mass.
Has a section on the distribution of floating-point numbers.

Kogge, P. [1981]. *The Architecture of Pipelined Computers,* McGraw-Hill, New York.
Has a brief discussion of pipelined multipliers.

Kohn, L., and S.-W. Fu [1989]. "A 1,000,000 transistor microprocessor," *IEEE Int'l Solid-State Circuits Conf.*, 54–55.
There are several articles about the i860, but this one contains the most details about its floating-point algorithms.

Koren, I. [1989]. *Computer Arithmetic Algorithms,* Prentice Hall, Englewood Cliffs, N.J.

Leighton, F. T. [1992]. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes,* Morgan Kaufmann, San Francisco.
This is an excellent book, with emphasis on the complexity analysis of algorithms. Section 1.2.1 has a nice discussion of carry-lookahead addition on a tree.

Magenheimer, D. J., L. Peters, K. W. Pettis, and D. Zuras [1988]. "Integer multiplication and division on the HP Precision architecture," *IEEE Trans. on Computers* 37:8, 980–990.
Gives rationale for the integer- and divide-step instructions in the Precision architecture.

Markstein, P. W. [1990]. "Computation of elementary functions on the IBM RISC System/6000 processor," *IBM J. of Research and Development* 34:1, 111–119.
Explains how to use fused muliply-add to compute correctly rounded division and square root.

Mead, C., and L. Conway [1980]. *Introduction to VLSI Systems,* Addison-Wesley, Reading, Mass.

Montoye, R. K., E. Hokenek, and S. L. Runyon [1990]. "Design of the IBM RISC System/6000 floating-point execution," *IBM J. of Research and Development* 34:1, 59–70.
Describes one implementation of fused multiply-add.

Ngai, T.-F., and M. J. Irwin [1985]. "Regular, area-time efficient carry-lookahead adders," *Proc. Seventh IEEE Symposium on Computer Arithmetic,* 9–15.
Describes a CLA like that of Figure H.17, where the bits flow up and then come back down.

Patterson, D. A., and J. L. Hennessy [1994]. *Computer Organization and Design: The Hardware/Software Interface,* Morgan Kaufmann, San Francisco.
Chapter 4 is a gentler introduction to the first third of this appendix.

Peng, V., S. Samudrala, and M. Gavrielov [1987]. "On the implementation of shifters, multipliers, and dividers in VLSI floating point units," *Proc. Eighth IEEE Symposium on Computer Arithmetic,* 95–102.
Highly recommended survey of different techniques actually used in VLSI designs.

Rowen, C., M. Johnson, and P. Ries [1988]. "The MIPS R3010 floating-point coprocessor," *IEEE Micro,* 53–62 (June).

Santoro, M. R., G. Bewick, and M. A. Horowitz [1989]. "Rounding algorithms for IEEE multipliers," *Proc. Ninth IEEE Symposium on Computer Arithmetic,* 176–183.
A very readable discussion of how to efficiently implement rounding for floating-point multiplication.

Scott, N. R. [1985]. *Computer Number Systems and Arithmetic,* Prentice Hall, Englewood Cliffs, N.J.

Swartzlander, E., ed. [1990]. *Computer Arithmetic,* IEEE Computer Society Press, Los Alamitos, Calif.
A collection of historical papers in two volumes.

Takagi, N., H. Yasuura, and S. Yajima [1985]."High-speed VLSI multiplication algorithm with a redundant binary addition tree," *IEEE Trans. on Computers* C-34:9, 789–796.
A discussion of the binary tree signed multiplier that was the basis for the design used in the TI 8847.

Taylor, G. S. [1981]. "Compatible hardware for division and square root," *Proc. Fifth IEEE Symposium on Computer Arithmetic,* 127–134.
Good discussion of a radix-4 SRT division algorithm.

Taylor, G. S. [1985]. "Radix 16 SRT dividers with overlapped quotient selection stages," *Proc. Seventh IEEE Symposium on Computer Arithmetic,* 64–71.
Describes a very sophisticated high-radix division algorithm.

Weste, N., and K. Eshraghian [1993]. *Principles of CMOS VLSI Design: A Systems Perspective,* 2nd ed., Addison-Wesley, Reading, Mass.
This textbook has a section on the layouts of various kinds of adders.

Williams, T. E., M. Horowitz, R. L. Alverson, and T. S. Yang [1987]. "A self-timed chip for division," *Advanced Research in VLSI, Proc. 1987 Stanford Conf.,* MIT Press, Cambridge, Mass.
Describes a divider that tries to get the speed of a combinational design without using the area that would be required by one.

Exercises

H.1 [12] <H.2> Using n bits, what is the largest and smallest integer that can be represented in the two's complement system?

H.2 [20/25] <H.2> In the subsection "Signed Numbers" (page H-7), it was stated that two's complement overflows when the carry into the high-order bit position is different from the carry-out from that position.

a. [20] <H.2> Give examples of pairs of integers for all four combinations of carry-in and carry-out. Verify the rule stated above.

b. [25] <H.2> Explain why the rule is always true.

H.3 [12] <H.2> Using 4-bit binary numbers, multiply -8×-8 using Booth recoding.

H.4 [15] <H.2> Equations H.2.1 and H.2.2 are for adding two n-bit numbers. Derive similar equations for subtraction, where there will be a borrow instead of a carry.

H.5 [25] <H.2> On a machine that doesn't detect integer overflow in hardware, show how you would detect overflow on a signed addition operation in software.

H.6 [15/15/20] <H.3> Represent the following numbers as single-precision and double-precision IEEE floating-point numbers.

a. [15] <H.3> 10.

b. [15] <H.3> 10.5.

c. [20] <H.3> 0.1.

H.7 [12/12/12/12/12] <H.3> Below is a list of floating-point numbers. In single precision, write down each number in binary, in decimal, and give its representation in IEEE arithmetic.

a. [12] <H.3> The largest number less than 1.

b. [12] <H.3> The largest number.

c. [12] <H.3> The smallest positive normalized number.

d. [12] <H.3> The largest denormal number.

e. [12] <H.3> The smallest positive number.

H.8 [15] <H.3> Is the ordering of nonnegative floating-point numbers the same as integers when denormalized numbers are also considered?

H.9 [20] <H.3> Write a program that prints out the bit patterns used to represent floating-point numbers on your favorite computer. What bit pattern is used for NaN?

H.10 [15] <H.4> Using $p = 4$, show how the binary floating-point multiply algorithm computes the product of 1.875×1.875.

H.11 [12/10] <H.4> Concerning the addition of exponents in floating-point multiply:

a. [12] <H.4> What would the hardware that implements the addition of exponents look like?

b. [10] <H.4> If the bias in single precision were 129 instead of 127, would addition be harder or easier to implement?

H.12 [15/12] <H.4> In the discussion of overflow detection for floating-point multiplication, it was stated that (for single precision) you can detect an overflowed exponent by performing exponent addition in a 9-bit adder.

a. [15] <H.4> Give the exact rule for detecting overflow.

b. [12] <H.4> Would overflow detection be any easier if you used a 10-bit adder instead?

H.13 [15/10] <H.4> Floating-point multiplication:

a. [15] <H.4> Construct two single-precision floating-point numbers whose product doesn't overflow until the final rounding step.

b. [10] <H.4> Is there any rounding mode where this phenomenon cannot occur?

H.14 [15] <H.4> Give an example of a product with a denormal operand but a normalized output. How large was the final shifting step? What is the maximum possible shift that can occur when the inputs are double-precision numbers?

H.15 [15] <H.5> Use the floating-point addition algorithm on page H-23 to compute $1.010_2 - .1001_2$ (in 4-bit precision) .

H.16 [10/15/20/20/20] <H.5> In certain situations, you can be sure that $a + b$ is exactly representable as a floating-point number, that is, no rounding is necessary.

a. [10] <H.5> If a, b have the same exponent and different signs, explain why $a + b$ is exact. This was used in the subsection "Speeding Up Addition" on page H-25.

b. [15] <H.5> Give an example where the exponents differ by 1, a and b have different signs, and $a + b$ is not exact.

c. [20] <H.5> If $a \geq b \geq 0$, and the top two bits of a cancel when computing $a - b$, explain why the result is exact (this fact is mentioned on page H-23).

d. [20] <H.5> If $a \geq b \geq 0$, and the exponents differ by 1, show that $a - b$ is exact unless the high order bit of $a - b$ is in the same position as that of a (mentioned in "Speeding Up Addition," page H-25).

e. [20] <H.5> If the result of $a - b$ or $a + b$ is denormal, show that the result is exact (mentioned in the subsection "Underflow," page H-36).

H.17 [15/20] <H.5> Fast floating-point addition (using parallel adders) for $p = 5$.

a. [15] <H.5> Step through the fast addition algorithm for $a + b$, where $a = 1.0111_2$ and $b = .11011_2$.

b. [20] <H.5> Suppose the rounding mode is toward $+\infty$. What complication arises in the above example for the adder that assumes a carry-out? Suggest a solution.

H.18 [12] <H.4, H.5> How would you use two parallel adders to avoid the final round-up addition in floating-point multiplication?

H.19 [30/10] <H.5> This problem presents a way to reduce the number of addition steps in floating-point addition from three to two using only a single adder.

a. [30] <H.5> Let A and B be integers of opposite signs, with a and b their magnitudes. Show that the following rules for manipulating the unsigned numbers a and b gives $A + B$.

1. Complement one of the operands.
2. Use end-around carry to add the complemented operand and the other (uncomplemented) one.
3. If there was a carry-out, the sign of the result is the sign associated with the uncomplemented operand.
4. Otherwise, if there was no carry-out, complement the result, and give it the sign of the complemented operand.

b. [10] <H.5> Use the above to show how steps 2 and 4 in the floating-point addition algorithm on page H-23 can be performed using only a single addition.

H.20 [20/15/20/15/20/15] <H.6> Iterative square root.

a. [20] <H.6> Use Newton's method to derive an iterative algorithm for square root. The formula will involve a division.

b. [15] <H.6> What is the fastest way you can think of to divide a floating-point number by 2?

c. [20] <H.6> If division is slow, then the iterative square root routine will also be slow. Use Newton's method on $f(x) = 1/x^2 - a$ to derive a method that doesn't use any divisions.

d. [15] <H.6> Assume that the ratio division by 2 : floating-point add : floating-point multiply is 1:2:4. What ratios of multiplication time to divide time makes each iteration step in the method of part (c) faster than each iteration in the method of part (a)?

e. [20] <H.6> When using the method of part (a), how many bits need to be in the initial guess in order to get double-precision accuracy after three iterations? (You may ignore rounding error.)

f. [15] <H.6> Suppose that when spice runs on the TI 8847, it spends 16.7% of its time in the square root routine (this percentage has been measured on other machines). Using the values in Figure H.36 and assuming three iterations, how much slower would spice run if square root were implemented in software using the method of part(a)?

H.21 [10/20/15/15/15] <H.6> Correctly rounded iterative division. Let a and b be floating-point numbers with p-bit significands ($p = 53$ in double precision). Let q be the exact quotient $q = a/b$, $1 \le q < 2$. Suppose that $\overline{q}$ is the result of an iteration process, that $\overline{q}$ has a few extra bits of precision, and that $0 < q - \overline{q} < 2^{-p}$. For the following, it is important that $\overline{q} < q$, even when q can be exactly represented as a floating-point number.

a. [10] <H.6> If x is a floating-point number, and $1 \le x < 2$, what is the next representable number after x?

b. [20] <H.6> Show how to compute q' from $\overline{q}$, where q' has $p + 1$ bits of precision and $\left| q - q' \right| < 2^{-p}$.

c. [15] <H.6> Assuming round to nearest, show that the correctly rounded quotient is either q', $q' - 2^{-p}$, or $q' + 2^{-p}$.

d. [15] <H.6> Give rules for computing the correctly rounded quotient from q' based on the low-order bit of q' and the sign of $a - bq'$.

e. [15] <H.6> Solve part(c) for the other three rounding modes.

H.22 [15] <H.6> Verify the formula on page H-30. [*Hint:* If $x_n = x_0(2 - x_0 b) \times \prod_{i=1,n} [1 + (1 - x_0 b)^{2^i}]$, then $2 - x_n b = 2 - x_0 b(2 - x_0 b) \prod[1 + (1 - x_0 b)^{2^i}] = 2 - [1 - (1 - x_0 b)^2] \prod[1 + (1 - x_0 b)^{2^i}]$.]

H.23 [15] <H.7> Our example that showed that double rounding can give a different answer from rounding once used the round-to-even rule. If halfway cases are always rounded up, is double rounding still dangerous?

H.24 [10/10/20/20] <H.7> Some of the cases of the italicized statement in the "Precisions" subsection (page H-34) aren't hard to demonstrate.

a. [10] <H.7> What form must a binary number have if rounding to q bits followed by rounding to p bits gives a different answer than rounding directly to p bits?

b. [10] <H.7> Show that for multiplication of p-bit numbers, rounding to q bits followed by rounding to p bits is the same as rounding immediately to p bits if $q \ge 2p$.

c. [20] <H.7> If a and b are p-bit numbers with the same sign, show that rounding $a + b$ to q bits followed by rounding to p bits is the same as rounding immediately to p bits if $q \ge 2p + 1$.

d. [20] <H.7> Do part (c) when a and b have opposite signs.

H.25 [Discussion] <H.7> In the MIPS approach to exception handling, you need a test for determining whether two floating-point operands could cause an exception. This should be fast and also not have too many false positives. Can you come up

with a practical test? The performance cost of your design will depend on the distribution of floating-point numbers. This is discussed in Knuth [1981] and the Hamming paper in Swartzlander [1990].

H.26 [12/12/10] <H.8> Carry-skip adders.

a. [12] <H.8> Assuming that time is proportional to logic levels, how long does it take an n-bit adder divided into (fixed) blocks of length k bits to perform an addition?

b. [12] <H.8> What value of k gives the fastest adder?

c. [10] <H.8> Explain why the carry-skip adder takes time $0(\sqrt{n})$.

H.27 [10/15/20] <H.8> Complete the details of the block diagrams for the following adders.

a. [10] <H.8> In Figure H.15, show how to implement the "1" and "2" boxes in terms of AND and OR gates.

b. [15] <H.8> In Figure H.18, what signals need to flow from the adder cells in the top row into the "C" cells? Write the logic equations for the "C" box.

c. [20] <H.8> Show how to extend the block diagram in H.17 so it will produce the carry-out bit c_8.

H.28 [15] <H.9> For ordinary Booth recoding, the multiple of b used in the ith step is simply $a_{i-1} - a_i$. Can you find a similar formula for radix-4 Booth recoding (overlapped triplets)?

H.29 [20] <H.9> Expand Figure H.29 in the fashion of H.27, showing the individual adders.

H.30 [25] <H.9> Write out the analog of Figure H.25 for radix-8 Booth recoding.

H.31 [18] <H.9> Suppose that $a_{n-1}. . .a_1a_0$ and $b_{n-1}. . .b_1b_0$ are being added in a signed-digit adder as illustrated in the example on page H-54. Write a formula for the ith bit of the sum, s_i, in terms of a_i, a_{i-1}, a_{i-2}, b_i, b_{i-1}, and b_{i-2}.

H.32 [15] <H.9> The text discussed radix-4 SRT division with quotient digits of −2, −1, 0, 1, 2. Suppose that 3 and −3 are also allowed as quotient digits. What relation replaces $|r_i| \le 2b/3$?

H.33 [25/20/30] <H.9> Concerning the SRT division table, Figure H.34:

a. [25] <H.9> Write a program to generate the results of Figure H.34.

b. [20] <H.9> Note that Figure H.34 has a certain symmetry with respect to positive and negative values of P. Can you find a way to exploit the symmetry and only store the values for positive P?

c. [30] <H.9> Suppose a carry-save adder is used instead of a propagate adder. The input to the quotient lookup table will be k bits of divisor and l bits of remainder, where the remainder bits are computed by summing the top l bits of the sum and carry registers. What are k and l? Write a program to generate the analog of Figure H.34.

H.34 [12/12/12]<H.9, H.12>The first several million Pentium chips produced had a flaw that caused division to sometimes return the wrong result. The Pentium uses a radix-4 SRT algorithm similar to the one illustrated in the example on page H-58 (but with the remainder stored in carry-save format: see Exercise H.33(c)). According to Intel, the bug was due to five incorrect entries in the quotient lookup table.

a. [12] <H.9, H.12> The bad entries should have had a quotient of plus or minus 2, but instead had a quotient of 0. Because of redundancy, it's conceivable that the algorithm could "recover" from a bad quotient digit on later iterations. Show that this is not possible for the Pentium flaw.

b. [12] <H.9, H.12> Since the operation is a floating-point divide rather than an integer divide, the SRT division algorithm on page H-46 must be modified in two ways. First, step 1 is no longer needed, since the divisor is already normalized. Second, the very first remainder may not satisfy the proper bound ($|r| \leq 2b/3$ for Pentium, see page H-56). Show that skipping the very first left shift in step 2(a) of the SRT algorithm will solve this problem.

c. [12] <H.9, H.12> If the faulty table entries were indexed by a remainder that could occur at the very first divide step (when the remainder is the divisor), random testing would quickly reveal the bug. This didn't happen. What does that tell you about the remainder values that index the faulty entries?

H.35 [12/12/12] <H.6, H.9> The discussion of the remainder-step instruction assumed that division was done using a bit-at-a-time algorithm. What would have to change if division were implemented using a higher-radix method?

H.36 [25] <H.9> In the array of Figure H.28, the fact that an array can be pipelined is not exploited. Can you come up with a design that feeds the output of the bottom CSA into the bottom CSAs instead of the top one, and that will run faster than the arrangement of Figure H.28?

I.1 Implementation Issues for the Snooping Coherence Protocol I-2
I.2 Implementation Issues in the Distributed Directory Protocol I-6
Exercises I-12

I

Implementing Coherence Protocols

The devil is in the details.

Classic Proverb

© 2003 Elsevier Science (USA). All rights reserved.

I.1 Implementation Issues for the Snooping Coherence Protocol

The major complication in actually using the snooping coherence protocol from Section 6.3 is that write misses are not atomic: The operation of detecting a write miss, obtaining the bus, getting the most recent value, and updating the cache cannot be done as if it took a single cycle. In particular, two processors cannot both use the bus at the same time. Thus, we must decompose the write into several steps that may be separated in time, but will still preserve correct execution. The first step detects the miss and requests the bus. The second step acquires the bus, places the miss on the bus, gets the data, and completes the write. Each of these two steps is atomic, but the cache block does not become exclusive until the second step has begun. As long as we do not change the block to exclusive or allow the cache update to proceed before the bus is acquired, writes to the same cache block will serialize when they reach the second step of the coherence protocol. Unfortunately, this two-step process does introduce new complications in the protocol.

Figure I.1 shows the actual finite-state diagram for implementing coherence for this two-step process under the assumption that a bus transaction is atomic once the bus is acquired. This assumption simply means that the bus is not a split transaction, and once it is acquired any requests are processed before another processor can acquire the bus. We discuss the complexities of a split-transaction bus shortly. In the simplest implementation, the finite-state machine in Figure I.1 is simply replicated for each block in the cache. Since there is no interaction among operations on different cache blocks, this replication of the controller works. Replicating the controller is not necessary, but before we see why, let's make sure we understand how the finite-state controller in Figure I.1 operates.

The additional states in Figure I.1 over those in Figure 6.12 on page 559 are all transient: The controller will leave those states when the bus is available. Four of the states are pending write-back states that arise because in a write-back cache when a block is replaced (or invalidated) it must be written back to the memory. Four events can cause such a write back:

1. A write miss on the bus by another processor for this exclusive block.
2. A CPU read miss that forces the exclusive block to be replaced.
3. A CPU write miss that forces the exclusive block to be replaced.
4. A read miss on the bus by another processor for this block.

In each of the cases the next state differs, hence there are four separate pending write-back states with four different successor states.

Logically replicating the controller for each cache block allows correct operation if two conditions hold (in addition to our base assumption that the processor blocks until a cache access completes):

1. An operation on the bus for a cache block and a pending operation for a different cache block are noninterfering.

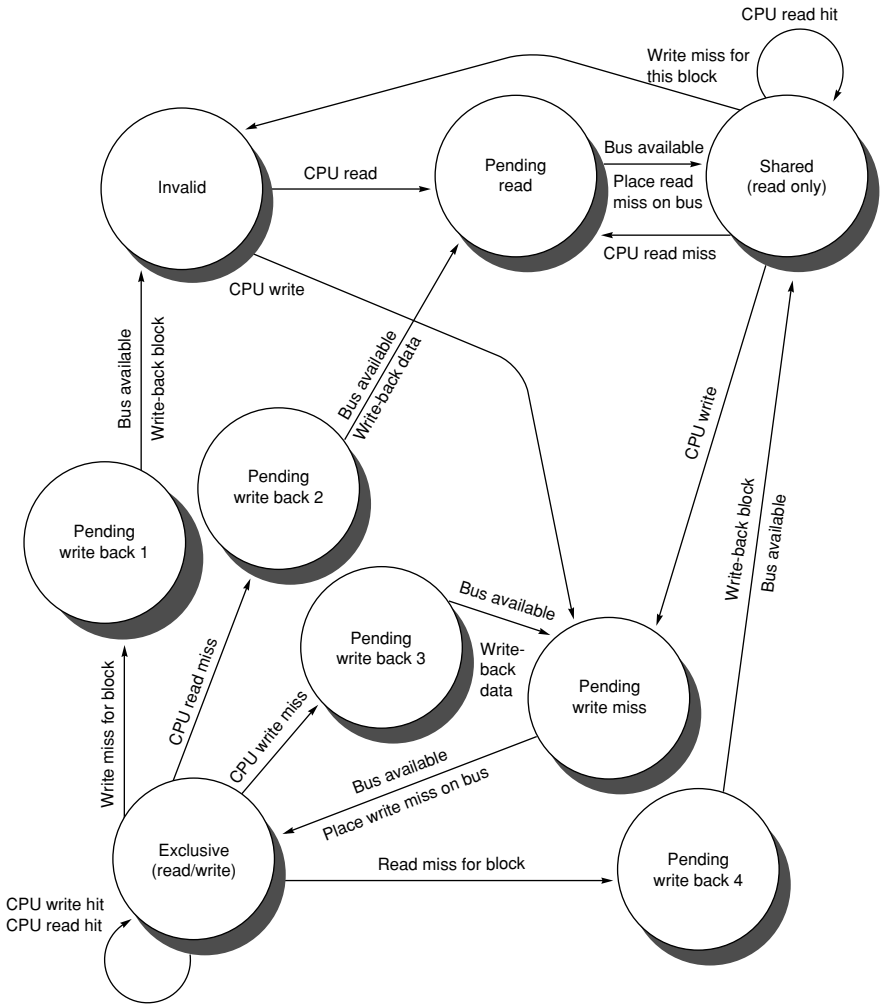


Figure I.1 A finite-state controller for a simple cache coherence scheme with a write-back cache. The engine that implements this controller must be reentrant, that is, it must handle multiple requests for different cache lines that are overlapped in time. The diagram assumes the processor stalls until a request is completed, but other transactions must be handled. This controller also assumes that a transition to a new state that involves a bus access does not complete until the bus access is completed. Notice that if we did not require a processor to generate a write miss when it transitioned from the shared to exclusive state, it might not obtain the latest value of a cache block, since some other processor may have updated that block. In a protocol using ownership or upgrade transitions, we will need to be able to transition out of the pending write state and restart an access if a conflicting write obtains the bus first.

2. The controller in Figure I.1 correctly deals with the cases when a pending operation and a bus operation are for the same block.

The first condition is certainly true, since operations for different blocks may proceed in any order and do not affect the state transitions for the other block. To see why the second condition is true, consider each of the pending states and what happens if a conflicting access occurs:

- Pending write back 1—The cache is writing back the data to eliminate it anyway, so a read or write miss for the block has no new effect. Notice, however, that the pending cache *must* use the bus cycle generated by the read or write miss to complete the write back. Otherwise, there will be no response to the miss, since the pending cache still has the only copy of the cache block. When it sees that the address of a miss matches the address of the block it is waiting to write back, it recognizes that the bus is available, writes the data, and transitions its state. This applies to all the pending write-back states.
- Pending write back 2, 3—The cache is eliminating a block in the exclusive state, so another miss for that block simply allows the write back to occur immediately. If the read or write miss on the bus is for the new block that the processor is trying to share, there is no interaction, since the processor does not yet have a copy of the block.
- Pending write back 4—In this case the processor is surrendering an exclusive block and simply completes the write back.
- Pending read, pending write miss —The processor does not yet have a copy of the block that it is waiting for, so a read or write miss for that block has no effect. Since the waiting cache still needs to place a miss on the bus and fetch the block, it is guaranteed to get a new copy.

With these additional states and our assumptions that the bus operates atomically, that misses always cause the state to be updated, and that the processor blocks until an access completes, our coherence implementation is both deadlock-free and correct. If some fairness guarantee is made for bus access, then this controller is also free of *livelock*. Livelock occurs when some portion of a computation cannot make progress, though other portions can. If one processor could be denied the bus indefinitely, then that processor could never make progress in its computation. Some guarantee of fairness on bus access prevents this.

There is still, however, one more critical implementation detail related to the bus transactions and what happens when a miss is processed. The key difference between the cache coherence case and the standard uniprocessor case occurs when the block is exclusive in some cache. Because it is a write-back cache, the memory copy is stale. In this case, the coherence unit will retrieve the block (called an *intervention*) and generate a write back. Since the memory does not know the state of the block, it will attempt to respond to the request as well. Since the data have been updated, the cache and processor will each attempt to drive the bus with different values. To prevent this, a line is added to the bus (often called the shared line) to coordinate the response. When the processor detects that it has a copy in the exclusive state, it signals the memory on this line and the memory

aborts the transaction. When the write back occurs, the memory gets the data and updates its copy. Since it is difficult to bound the amount of time that it can take to snoop the local cache copy, this line is usually implemented as a wired-OR with each processor holding its input low until it knows it does not have the block in exclusive state. The memory waits for the line to go high, indicating that no cache has the copy in the exclusive state, before putting data on the bus.

If the bus had a split-transaction capability then we could not assume that a response would occur immediately. In fact, implementing a split transaction with coherence is significantly more complex. One complication arises from the fact that we must number and track bus transactions, so that a controller knows when a bus action is a response to its request. Another complication is dealing with races that can arise because two operations for the same cache block could potentially be outstanding simultaneously. An example illustrates this complication best. What happens when two processors try to write a word in the same cache block? Without split transactions, one of the operations reaches the bus first and the other must change the state of the block to invalid and try the operation again. Only one of the transactions is outstanding on the bus at any point.

Example Suppose we have a split-transaction bus and no cache has a copy of a particular block. Show how when both P1 and P2 try to write a word in that block, we can get an incorrect result using the protocol in Figure I.1 on page I-3.

Answer With the protocol in Figure I.1, the following sequence of events could occur:

1. P1 places a write miss for the block on the bus. Since P2 has the data in the invalid state, nothing occurs.
2. P2 places its write miss on the bus; again, since no copy exists, no state changes are needed.
3. The memory responds to P1's request. P1 places the block in the exclusive state and writes the word into the block.
4. The memory responds to P2's request. P2 places the block in the exclusive state and writes the word into the block.

Disaster! Two caches now have the same block in the exclusive state and memory will be inconsistent.

How can this race be avoided? The simplest way is to use the broadcast capability of the bus. All coherence controllers track all bus accesses. In a split-transaction bus, the transactions must be tagged with the processor identity (or a transaction number), so that a processor can identify a reply to its request. Every controller can keep track of the memory address of any outstanding bus requests, since it can see the request and the corresponding reply on the bus. When the local processor generates a miss, the controller does not place the miss request on the bus until there are no outstanding requests for the same cache block. This will

force P2 in the above example to wait for P1's access to complete, allowing P1 to place the data in the exclusive state (and write the word into the block). The miss request from P2 will then cause P1 to do a write back and move the block to the invalid state. Alternatively, we could have each processor buffer only its own requests and track the responses to others. If the address of the requested block were included in the reply, then the second processor to request the block could ignore the reply and reissue its request.

These race conditions are what make implementing coherence even more tricky as the interconnection mechanism becomes more sophisticated. As we will see in the next section, such problems are slightly worse in a directory-based system that does not have a broadcast mechanism like a bus, which can be used to order all accesses.

I.2 Implementation Issues in the Distributed Directory Protocol

One further source of complexity of a directory protocol comes from the lack of atomicity in transactions. Several of the operations that are atomic in a bus-based snoopy protocol cannot be atomic in a directory-based machine. For example, a read miss, which is atomic in the snoopy protocol, cannot be atomic, since it requires messages to be sent to remote directories and caches. In fact, if we attempt to implement these operations in an atomic fashion in a distributed-memory machine, we can have deadlock. Recall from Chapter 6 that a deadlock means that the machine has reached a state from which it cannot make forward progress. This is easy to see with an example.

Example Show how deadlock can occur if a node treats a read miss as atomic and hence is unable to respond to other requests until the read miss is completed.

Answer Assume that two nodes P1 and P2 each have exclusive copies of cache blocks X1 and X2 that have different home directories. Consider the following sequence of events shown in Figure I.2.

Events caused by P1 activity	Events caused by P2 activity
P1 read miss for X2	P2 read miss for X1
Directory for X2 receives read miss and generates a fetch that is sent to P2	Directory for X1 receives read miss and generates a fetch that is sent to P1
Fetch arrives at P1, waits for completion of atomic read miss	Fetch arrives at P2, waits for completion of atomic read miss

Figure I.2 Events caused by P1 and P2 leading to deadlock.

At this point the nodes are deadlocked. In this case, since the requests are for separate blocks, deadlock can be avoided by duplicating the controller for each block. This allows the controllers to accept a request for one block while a request for another block is in process. In practice, complications arise because requests for the same block can collide, as we will see shortly.

The almost complete lack of atomicity in transactions causes most of the complexities in translating these state transition diagrams into actual finite-state controllers. There are two assumptions about the interconnection network that significantly simplify the implementation. First, we assume that the network provides point-to-point *in-order delivery* of messages. This means that two messages sent from a single node to another node arrive in the order they were sent. No assumptions are made about messages originating from, or destined to, different nodes. Second, we assume the network has unlimited buffering. This second assumption means that a message can always be accepted into the network. This reduces the possibility for deadlock and allows us to treat some nonatomic action, where we would need to be able to deal with a full set of network buffers, as atomic actions. Of course, we also assume that the network delivers all messages within a finite time.

While the first assumption, in-order transmission, is quite reasonable and is, in fact, true in many machines, the second assumption, unlimited buffering, is not true. Actually, the network need only be capable of buffering a finite number of messages, since we still assume that processors block on misses. In practice, this number may still be large and unreasonable, so later in the section we will discuss what has to change to eliminate the assumption that a message can always be accepted, while still preventing deadlock.

We also assume that the coherence controller is duplicated for each cache block (to avoid having to deal with unrelated transactions) and that a state transition only completes when a message has been transmitted and a data value reply received (when needed). This last assumption simply means that we do not allow the CPU to continue and read or write a cache block until the read or write miss is satisfied by a data value reply message. This simply eliminates a transition state that waits for the block to arrive. Because we are assuming unlimited buffering, we also assume that an outgoing message can always be transmitted before the next incoming message is accepted.

Under these assumptions the state transition diagram of Figure 6.29 on page 581 can be used for the coherence controller at the cache with one small addition: The controller simply throws away any incoming transactions, other than the data value reply, while waiting for a read or write miss. Let's look at each possible case that can arise while the cache is waiting for a response from the directory. Cases where the cache is transitioning the block to invalid, either from the shared or exclusive state, do not matter, since any incoming signals for this block do not affect the block once it is invalid. Hence, we need only consider cases where the processor is transitioning to the shared or exclusive state. There are two such cases:

- CPU read miss from either invalid or exclusive—The directory will not reply until the block is available. Furthermore, since any write back of an exclusive entry for this block has been done, the controller can ignore any requests.
- CPU write miss—Any required write back is done first and the processor is stalled. Since it cannot hold a block exclusive in this cache entry, it can ignore requests for this block until the write miss is satisfied from the directory.

The directory case is more complex to explain, since multiple cache controllers may send a message for the same block close to the same time. These operations must be serialized. Unlike the snoopy case where every controller sees every request on the bus at the same time, the individual caches only know what has happened when they are notified by the directory. Because the directory serializes the messages when it receives them and because all write misses for a given cache block go to the same directory, writes will be serialized by the home directory.

Thus, the directory controller's main problem is to deal with the distributed representation of the cache state. Since the directory must wait for the completion of certain operations, such as sending invalidates and fetching a cache block before transitioning state, most potential races are eliminated. Because we assume unlimited buffering, the directory can always complete a transaction before accepting the next incoming message. For this reason, the state transition diagram in Figure 6.30 can be used as an implementation. To see why, we must consider cases where the directory and the local cache do not agree on the state of a block. The cache can only have a block in a less restricted state than the directory believes the block is in, because transitioning to exclusive from invalid or shared, or to shared from invalid, requires a message to the directory and a reply. Thus, the only cases to consider are

- Local cache state is invalid, directory state is exclusive—The cache controller must have performed a data write back of the block (see Figure 6.29). Hence the directory will shortly obtain the block. Furthermore no invalidation is needed, since the block has been replaced.
- Local cache state is invalid, directory state is shared (the local cache is replacing the line)—The directory will send an invalidate, which may be ignored, since the block has been replaced. Some directory protocols send a *replacement hint* message when a shared line is replaced. Such messages are used to eliminate unnecessary invalidates and to reduce the state needed in the directory.
- Local cache state is shared, directory state is exclusive—The write back has already been done and the block has been replaced, so a fetch/invalidate, which could be sent by the directory, can be ignored.

Hence, the protocol operates correctly with infinite buffering.

Dealing with Finite Buffering

What happens when the network does not have unlimited buffering? The major implication of this limit is that a cache or directory controller may be unable to complete a message send. This could lead to deadlock. The example on page I-6 showed such a deadlock case. Even if we assume a separate controller for each cache block, so that the requests do not interfere in the controller, the example will deadlock if there are no buffers available to send the replies.

The occurrence of such a deadlock is based on three properties, which characterize many deadlock situations:

1. More than one resource is needed to complete a transaction: Buffers are needed to generate requests, create replies, and accept replies.
2. Resources are held until a nonatomic transaction completes: The buffer used to create the reply cannot be freed until the reply is accepted.
3. There is no global partial order on the acquisition of resources: Nodes can generate requests and replies at will.

These characteristics lead to deadlock, and avoiding deadlock requires breaking one of these properties. Imposing a global partial order, the solution used in a bus-based system, is unworkable in a larger-scale, distributed machine. Freeing up resources without completing a transaction is difficult, since the transaction must be completely backed out and cannot be left half-finished. Hence, our approach will be to try to resolve the need for multiple resources. We cannot simply eliminate this need, but we can try to ensure that the resources will always be available.

One way to ensure that a transaction can always complete is to guarantee that there are always buffers to accept messages. Although this is possible for a small machine with processors that block on a cache miss, it may not be very practical, since a single write could generate many invalidate messages. In addition, features such as prefetch would increase the amount of buffering required. There is an alternative strategy, which most systems use, and which ensures that a transaction will not actually be initiated until we can guarantee that it has the resources to complete. The strategy has four parts:

1. A separate network (physical or virtual) is used for requests and replies, where a reply is any message that a controller waits for in transitioning between states. This ensures that new requests cannot block replies that will free up buffers.
2. Every request that expects a reply allocates space to accept the reply when the request is generated. If no space is available, the request waits. This ensures that a node can always accept a reply message, which will allow the replying node to free its buffer.

3. Any controller can reject (usually with a *negative acknowledge* or *NAK*) any request, but it can never NAK a reply. This prevents a transaction from starting if the controller cannot guarantee that it has buffer space for the reply.
4. Any request that receives a NAK in response is simply retried.

To understand why this is sufficient to prevent deadlock, let's first consider our earlier example. Because a write miss is a request that requires a reply, the space to accept the reply is preallocated. Hence, both nodes will have space for the reply. Since the networks are separate, a reply can be received even if no more space is available for requests. Since the requests are for two different blocks, the separate coherence controllers handle the requests. If the accesses are for the same address, then they are serialized at the directory and no problem exists.

To see that there are no deadlocks more generally, we must ensure that all replies can be accepted, and that every request is eventually serviced. Since a cache controller or directory controller can have at most one request needing a reply outstanding, it can always accept the reply when it returns. To see that every request is eventually serviced, we need only show that any request could be completed. Since every request starts with a read or write miss at a cache, it is sufficient to show that any read or write miss is eventually serviced. Since the write miss case includes the actions for a read miss as a subset, we focus on showing the write misses are serviced. The simplest situation is when the block is uncached; since that case is subsumed by the case when the block is shared, we focus on the shared and exclusive cases. Let's consider the case where the block is shared:

- The CPU attempts to do a write and generates a write miss that is sent to the directory. At this point the processor is stalled.
- The write miss is sent to the directory controller for this memory block. Note that although one cache controller handles all the requests for a given cache block, regardless of its memory contents, there is a controller for every memory block. Thus the only conflict at the directory controller is when two requests arrive for the same block. This is critical to the deadlock-free operation of the controller and needs to be addressed in an implementation using a single controller.
- Now consider what happens at the directory controller: Suppose the write miss is the next thing to arrive at the directory controller. The controller sends out the invalidates, which can always be accepted if the controller for this block is idle. If the controller is not idle, then the processor must be stalled. Since the processor is stalled, it must have generated a read or write miss. If it generated a read miss, then it has either displaced this block or does not have a copy. If it does not have a copy, then it has sent a read miss and cannot continue until the read miss is processed by the directory (the read miss will not be handled until the write miss is). If the controller has replaced the block, then we need not worry about it. If the controller is idle, then an invalidate occurs, and the copy is eliminated.

The case where the block is exclusive is somewhat trickier. Our analysis begins when the write miss arrives at the directory controller for processing. There are two cases to consider:

- The directory controller sends a fetch/invalidate message to the processor where it arrives to find the cache controller idle and the block in the exclusive state. The cache controller sends a data write back to the home directory and makes its state invalid. This reply arrives at the home directory controller, which can always accept the reply, since it preallocated the buffer. The directory controller sends back the data to the requesting processor, which can always accept the reply; after the cache is updated the requesting cache controller restarts the processor.
- The directory controller sends a fetch/invalidate message to the node indicated as owner. When the message arrives at the owner node, it finds that this cache controller has taken a read or write miss that caused the block to be replaced. In this case, the cache controller has already sent the block to the home directory with a data write back and made the data unavailable. Since this is exactly the effect of the fetch/invalidate message, the protocol operates correctly in this case as well.

We have shown that our coherence mechanism operates correctly when controllers are replicated and when responses can be NAKed and retried. Both of these assumptions generate some problems in the implementation.

Implementing the Directory Controllers

First, let's consider how these controllers, which we have assumed are replicated, can be built without actually replicating them. On the side of the cache controllers, because the processors stall, the actual implementation is quite similar to what was needed for the snoopy controller. We can simply add the transient states just as we did for the snoopy case and note that a transaction for a different cache block can be handled while the current processor-generated operation is pending. Since a processor blocks on a request, at most one pending operation need be dealt with.

On the side of the directory controller, things are more complicated. The difficulty arises from the way we handle the retrieval and return of a block. In particular, during the time a directory retrieves an exclusive block and returns it to the requesting node, the directory must accommodate other transactions. Otherwise, integrating the directory controllers for different cache blocks will lead to the possibility of deadlock. Because of this situation, the directory controller must be reentrant, that is, it must be capable of suspending its execution while waiting for a reply and accept another transaction. The only place this must occur is in response to read or write misses, while waiting for a response from the owner. This leads to three important observations:

1. The state of the controller need only be saved and restored while either a fetch or a fetch/invalidate operation is outstanding.
2. The implementation can bound the number of outstanding transactions being handled in the directory, by simply NAKing read or write miss requests that could cause the number of outstanding requests to be exceeded.
3. If instead of returning the data through the directory, the owner node forwards the data directly to the requester (as well as returning it to the directory), we can eliminate the need for the directory to handle more than one outstanding request. This motivation, in addition to the reduction of latency, is the reason for using the forwarding style of protocol. The forwarding-style protocol introduces another type of problem that we discuss in the exercises.

The major remaining implementation difficulty is to handle NAKs. One alternative is for each processor to keep track of its outstanding transactions, so it knows, when the NAK is received, what the requested transaction was. The alternative is to bundle the original request into the NAK, so that the controller receiving the NAK can determine what the original request was. Because every request allocates a slot to receive a reply and a NAK is a reply, NAKs can always be received. In fact, the buffer holding the return slot for the request can also hold information about the request, allowing the processor to reissue the request if it is NAKed.

This completes the implementation of the directory scheme. In practice, great care is required to implement these protocols correctly and to avoid deadlock. The key ideas we have seen in this section—dealing with nonatomicity and finite buffering—are critical to ensuring a correct implementation. Designers have found that both formal and informal verification techniques are helpful for ensuring that implementations are correct.

Exercises

I.1 [20] <6.5, I.2> The Convex Exemplar is a coherent shared-memory machine organized as a ring of eight-processor clusters. Describe a protocol for this machine, assuming that the ring can be snooped and that a directory sits at the junction of the ring and can also be interrogated from inside the cluster. How much directory storage is needed? If the coherence misses are uniformly distributed and the capacity misses are all within a cluster, what is the average memory access time for Ocean running on 64 processors?

I.2 [15/20] <6.5, I.2> As we discussed in Section I.2, many DSM machines use a forwarding protocol, where a write miss request to a remote dirty block is forwarded to the node that has the copy of the block. The remote node then generates both a write-back operation and a data value reply.

a. [15] <6.5> Modify the state diagrams of Figures 6.29 and 6.30 so that the diagrams implement a forwarding protocol.

b. [20] <6.5, I.2> Forwarding protocols introduce a race condition into the protocol. Describe this race condition. Show how NAKs can be used to resolve the race condition.

I.3 [20] <6.5, I.2> Supporting lock-up free caches can have different implications for coherence protocols. Show how, without additional changes, allowing multiple outstanding misses from a node in a DSM can lead to deadlock—even if buffering is unlimited.