1.1	Introduction	2
1.2	The Changing Face of Computing and the Task of the Computer Designer	4
1.3	Technology Trends	11
1.4	Cost, Price, and Their Trends	14
1.5	Measuring and Reporting Performance	24
1.6	Quantitative Principles of Computer Design	39
1.7	Putting It All Together: Performance and Price-Performance	48
1.8	Another View: Power Consumption and Efficiency as the Metric	56
1.9	Fallacies and Pitfalls	57
1.10	Concluding Remarks	65
1.11	Historical Perspective and References	67
	Exercises	74

1

Fundamentals of Computer Design

And now for something completely different.

Monty Python's Flying Circus

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 dollars. 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

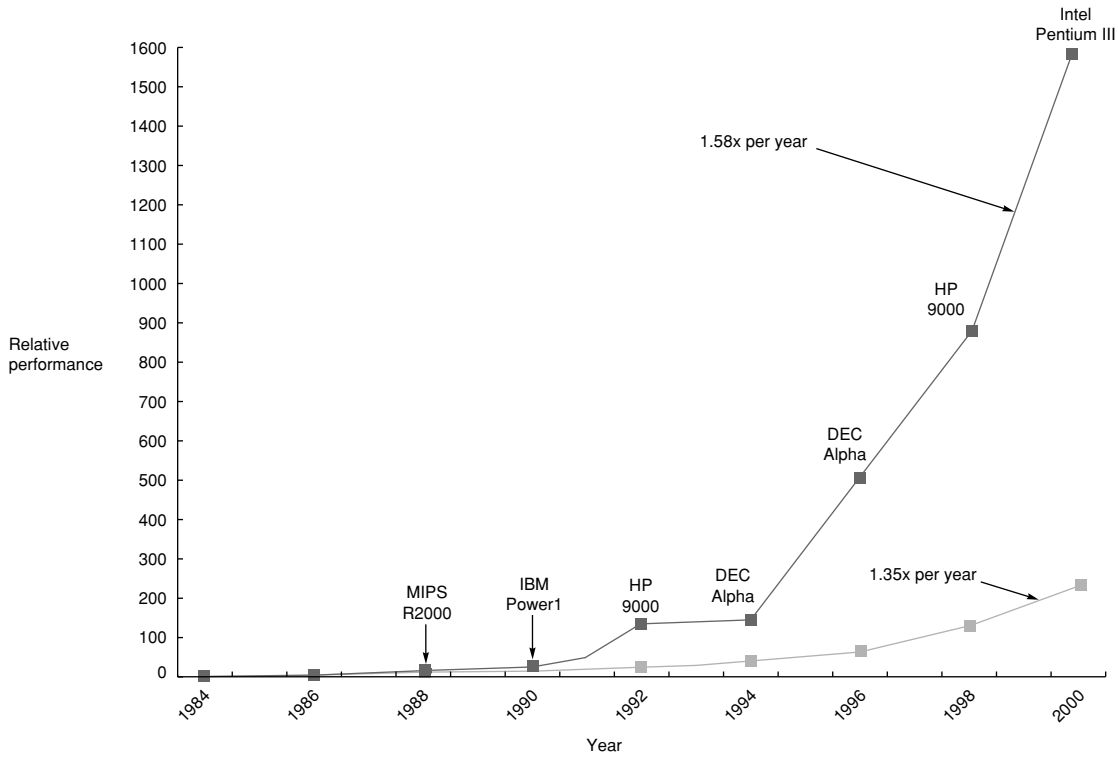

Figure 1.1 Growth in microprocessor performance since the mid-1980s has been substantially higher than in earlier years as shown by plotting SPECint performance. This chart plots relative performance as measured by the SPECint benchmarks with base of one being a VAX 11/780. Since SPEC has changed over 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 led to a difference in performance of about a factor of 15. Performance for floating-point-oriented calculations has increased even faster.

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, was about a factor of 15.

In the last few years, the tremendous improvement 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 processor. Beginning in the late 1990s, as transistor counts soared, the overhead (in transistors) of interpreting the more complex x86 architecture became negligible 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 we 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 time-sharing—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 time-sharing 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, 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 20 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 $1000 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 for a discussion of the issues affecting the 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, 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 because of 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 7.

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

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	$6450	$565	$283	$56.5
Credit card authorization	$2600	$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
Online 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 and that downtime is distributed uniformly. These data are from Kembel [2000] and were collected and analyzed by Contingency Planning Research.

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!

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 is 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).

Embedded Computers

Embedded computers—computers lodged in other devices where the presence of the computers is not immediately obvious—are the fastest growing portion of the computer market. These devices range from everyday machines (most microwaves, most washing machines, most printers, most networking switches, and all cars contain simple embedded 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 computers 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 part of the 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 dollars, to high-end embedded processors that cost hundreds of dollars and can execute a billion instructions per second 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 tends 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 a substantial portion of the system cost, and it is important to optimize memory size in such cases. Sometimes the application is expected to fit totally in the memory on the processor chip; other times the application 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. The designer uses a combined hardware/software solution that includes some custom hardware and an embedded processor core that is integrated with the custom hardware, often on the same chip.
2. The designer uses custom software running on an off-the-shelf embedded processor.
3. The designer uses a digital signal processor and custom software for the processor. *Digital signal processors* (DSPs) 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 architecture and the use 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 the 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	$1000–$10,000	$10,000–$10,000,000	$10–$100,000 (including network routers at the high end)
Price of microprocessor module	$100–$1000	$200–$2000 (per processor)	$0.20–$200 (per processor)
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. Note the wide range in system price for servers and embedded systems. For servers, this range arises from the need for very large-scale multiprocessor systems for high-end transaction processing and Web server applications. For embedded systems, one significant high-end application is a network router, which could include multiple processors as well as lots of memory and other electronics. 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 $5000—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. We 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 instruction set design. This challenge is particularly acute at the present, when the differences among instruction sets are small and when there are three rather distinct application 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 embedded processors with identical instruction set architectures but very different organizations are the NEC VR 5432 and the NEC VR 4122. Both processors implement the MIPS64 instruction set, but they have very different pipeline and cache organizations. In addition, the 4122 implements the floating-point instructions in software rather than hardware!

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 Celeron 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, which 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

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. G, H)
Commercial servers	Support for databases and transaction processing; enhancements for reliability and availability; support for scalability (Ch. 2, 6, 8)
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, 6)
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, 8)*
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. H), special arithmetic for graphics or signal processing
I/O bus	For I/O devices: Ultra ATA, Ultra SCSI, PCI (Ch. 7, 8)
Operating systems	UNIX, PalmOS, Windows, Windows NT, Windows CE, CISCO IOS
Networks	Support required for different networks: Ethernet, Infiniband (Ch. 8)
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.

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.

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 lead 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 addition, 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 7, 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 at about 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 sizes have decreased from 10 microns in 1971 to 0.18 microns in 2001. Since the transistor count per square millimeter of silicon is determined by the surface area of a transistor, 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 dimension 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 a 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 get 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 2 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 a watt, while a 2 GHz Pentium 4 consumes close to 100 watts. The fastest workstation and server microprocessors in 2001 consumed between 100 and 150 watts. Distributing the power, removing the heat, and preventing 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 significance: More than half the PCs sold in 1999 were priced at less than $1000, 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 the cost of a computer design and how these factors are changing over time.

The Impact of Time, Volume, and Commodification

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 5 and 10 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

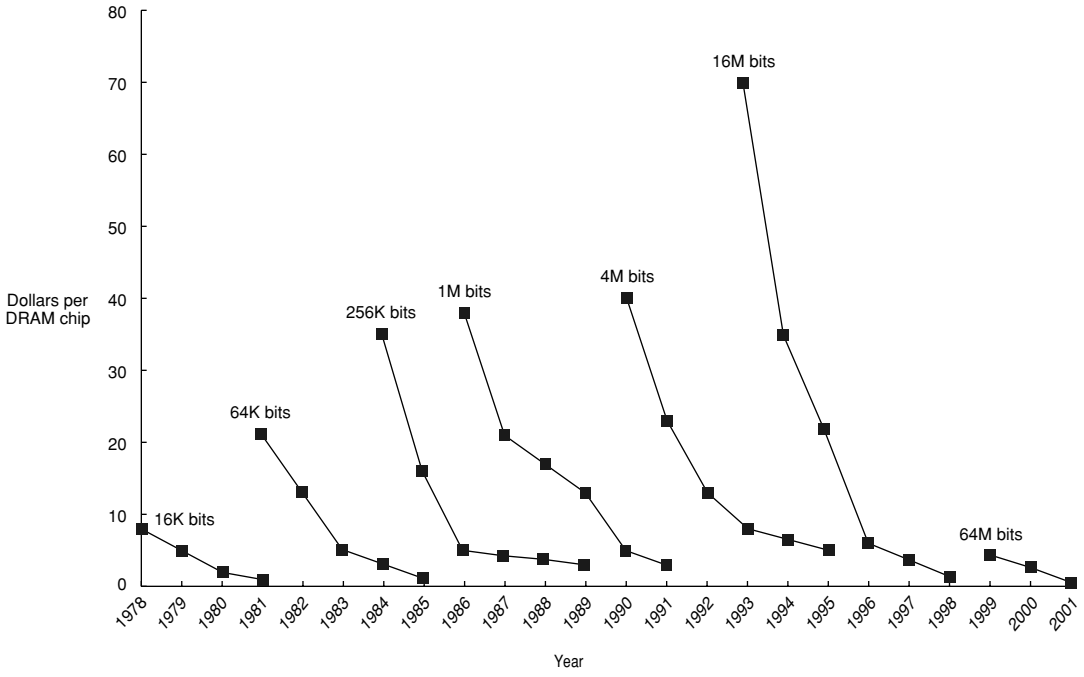

Figure 1.5 Prices of six generations of DRAMs (from 16K bits to 64M bits) over time in 1977 dollars, showing the learning curve at work. A 1977 dollar is worth about $2.95 in 2001; more than half of this inflation occurred in the five-year period of 1977–82, during which the value changed to $1.59. The 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.

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 number of vendors that ship virtually identical products and are highly com-

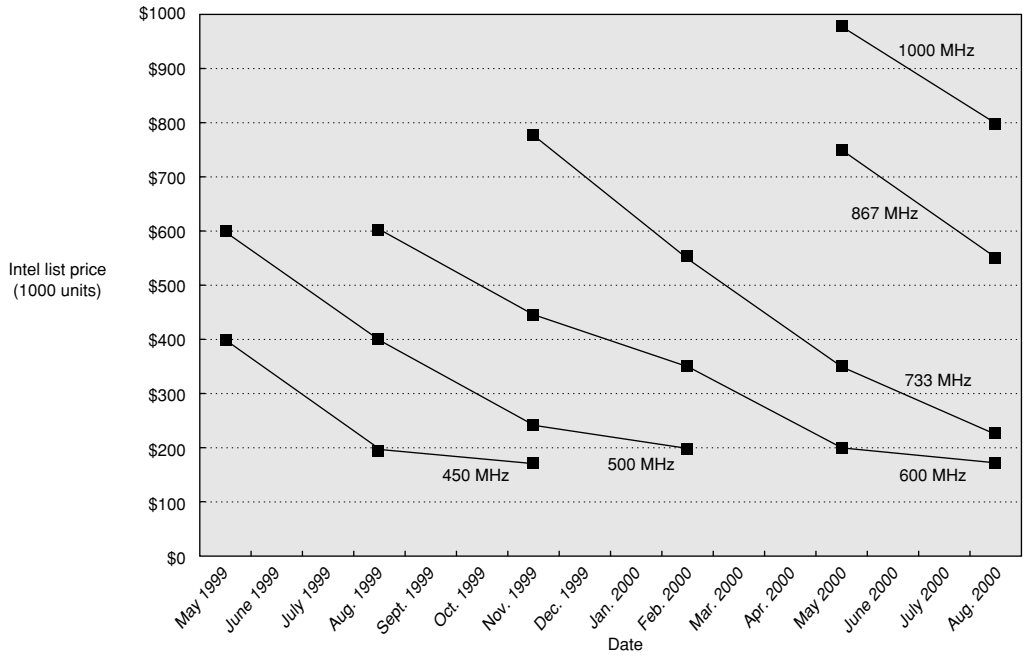

Figure 1.6 The price of an Intel Pentium III at a given frequency decreases over time as yield enhancements decrease the cost of a good die and competition forces price reductions. Data courtesy of *Microprocessor Report,* May 2000 issue. The most recent introductions will continue to decrease until 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.

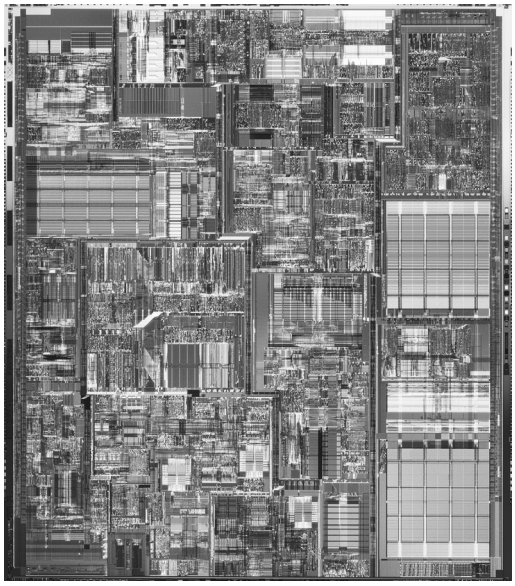

Figure 1.7 Photograph of an Intel Pentium 4 microprocessor die. (Courtesy Intel.)

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

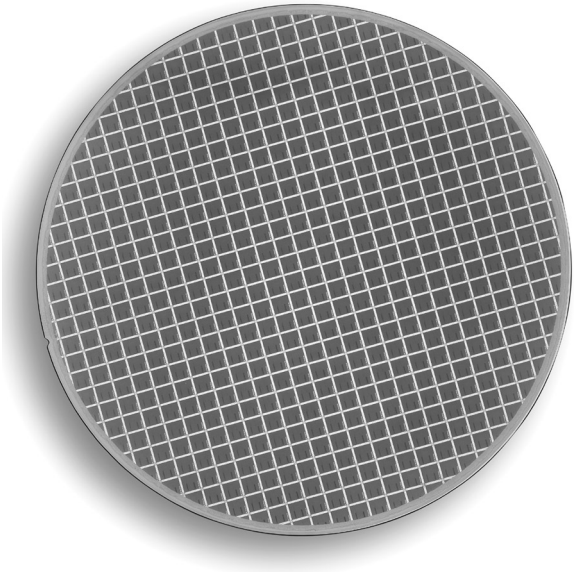

Figure 1.8 This 8-inch wafer contains 564 MIPS64 R20K processors implemented in a 0.18µ process. The R20K is an implementation of the MIPS64 architecture with instruction set extensions, called MIPS-3D, for use in three-dimensional graphics computations. The R20K is available at speeds from 500 to 750 MHz and is capable of executing two integer operations every clock cycle. Using the MIPS-3D instructions, the R20K can perform up to 3 billion floating-point operations per second. (Courtesy MIPS Technologies, Inc.)

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

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.

Learning 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:

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

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 inches) 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

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

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

$$\text{Die yield} = \left(1 + \frac{0.6 \times 1}{4.0}\right)^{-4} = 0.57$$

For the smaller die, it is

$$\text{Die yield} = \left(1 + \frac{0.6 \times 0.49}{4.0}\right)^{-4} = 0.75$$

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 366 good 1 cm^2 dies from the 30 cm wafer and 1014 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 for Exercise 1.8 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 accommodated. 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 four to six 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 would be around $5.42, while the cost per die of the 1 cm^2 die would be about $15.03, or almost 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, 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, it would appear that the cost of a die would grow with the fourth power of the die size. In practice, however, because the number of defects per unit area is small, the number of good dies per wafer, and hence the cost per die, grows roughly as the square of the die area. 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 1.8.

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 1 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 (which 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 $1000 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.

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	22%
	DRAM (128 MB)	5%
	Video card	5%
	Motherboard with basic I/O support, networking	5%
	Subtotal	**37%**
I/O devices	Keyboard and mouse	3%
	Monitor	19%
	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 $1000 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.

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.

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, and entire books have been written on the subject. The purpose of this section is to give you a simple introduction to what factors determine price, and to 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 difference 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 distinctive, competition is more intense, thus forcing lower prices and often lower profits, which in turn lead to a lower gross margin.

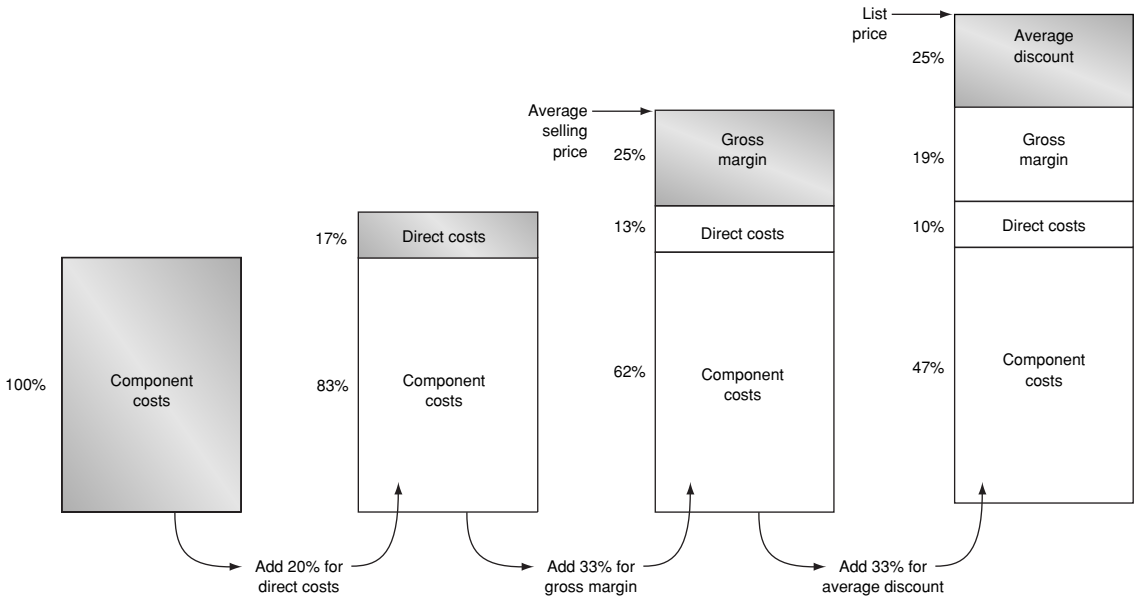

Figure 1.10 The components of price for a $1000 PC. Each increase is shown along the bottom as a tax on the prior price. The percentages of the new price for all elements are shown on the left of each column.

List price and average selling price are not the same, since companies typically offer volume discounts, lowering the average selling price. As personal computers became commodity products, the retail markups 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 would 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 preceding information 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—behave exactly like this. Between these extremes is *cost-performance design,* where the designer balances cost versus performance. Most of the PC market, the workstation 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

$$\frac{\text{Execution time}_Y}{\text{Execution time}_X} = n$$

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

$$n = \frac{\text{Execution time}_Y}{\text{Execution time}_X} = \frac{\frac{1}{\text{Performance}_Y}}{\frac{1}{\text{Performance}_X}} = \frac{\text{Performance}_X}{\text{Performance}_Y}$$

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 an 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. Our 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 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
Comments on the Whetstone benchmark (1976)

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 per-

formance 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 encounter 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 blocks for a benchmark, either with modifications to the application or with a script that acts as stimulus to the application. Applications are modified for one of 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 simulate application programs so as 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; 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. 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. All the SPEC benchmark suites are documented, together with reported results, at *www.spec.org*.

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

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 in Section 1.9 discusses the evolution of the benchmarks.) SPEC CPU2000, summarized in Figure 1.12, consists of a set of 11 integer benchmarks (CINT2000)

Benchmark name	Benchmark description
Business Winstone	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.
CC Winstone	Simulates multiple applications focused on content creation, such as Photoshop, Premiere, Navigator, and various audio-editing programs.
Winbench	Runs a variety of scripts that test CPU performance, video system performance, and disk performance using kernels focused on each subsystem.

Figure 1.11 A sample of some of the many PC benchmarks. The first two are scripts using real applications, and the last is a mixture of kernels and synthetic 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 services. For more information on these benchmarks, see *www.etestinglabs.com/benchmarks/*.

and 14 floating-point benchmarks (CFP2000). The SPEC benchmarks are real programs, 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 chromodynamics, 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 SPEC2000 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.

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

1. *Pro/Engineer*—A solid modeling application that does extensive 3D rendering. The input script is a model of a photocopying machine consisting of 370,000 triangles.
2. *SolidWorks 2001*—A 3D CAD/CAM design tool running a series of five tests varying from I/O intensive to CPU intensive. The largest input is a model of an assembly line consisting of 276,000 triangles.

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 three-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 SPEC CPU2000 benchmark suites. The 11 integer programs (all in C, except one in C++) are used for the CINT2000 measurement, while the 14 floating-point programs (6 in FORTRAN-77, 5 in C, and 3 in FORTRAN-90) are used for the CFP2000 measurement. See *www.spec.org* for more on these benchmarks.

3. *Unigraphics V15*—Based on an aircraft model and covering 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 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 7 discusses some file and I/O system benchmarks in detail.) SPECWeb 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 (TP) benchmarks measure the ability of a system to handle transactions, which consist of database accesses and updates. An airline reservation 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—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, preknowledge of the queries is taken for granted, and the DBMS system can be optimized to run these queries. TPC-W is a 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 *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 database 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.

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

Benchmark type	Number of kernels	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 (Bézier curve calculation, dithering, image rotation, text processing)
Telecommunications	6	Filtering and DSP benchmarks (autocorrelation, FFT, decoder, 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.

machine and 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 a 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 systems 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 the performance of compute-oriented benchmarks. The impact of compiler technology can be especially large when modification of the source is allowed (see the example with the EEMBC benchmarks in Figure 1.31 in Section 1.9) or when a benchmark is particularly susceptible to an optimization (see the example from SPEC described on page 58). 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 1.9, we'll see the tuning parameters for the optimized performance runs on this machine.

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 different approaches to addressing this question:

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 modifications 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 benchmarks, 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 changed, provided those changes become part of the production version.

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. These data are for the base CINT2000 report. The data are available online at *www.spec.org/osg/cpu2000/results/cpu2000.html.*

3. Source modifications are allowed. Several supercomputer benchmark suites allow modification of the source code. For example, the NAS supercomputer benchmarks specify the input and output and supply a version of the source, but vendors are allowed to rewrite the source, including changing the algorithms, as long as the modified version produces the same output. EEMBC also allows source-level changes to its benchmarks and reports these as "optimized" measurements, versus "out-of-the-box" measurements, which 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 in Section 1.9 shows the significant benefits from hand-coding on several different embedded 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 pro-

vide 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 another 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.

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.

	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].

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 indicate 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 $w_i = \dfrac{1}{\text{Time}_i \times \sum_{i=1}^{n} \left(\dfrac{1}{\text{Time}_j}\right)}$.

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 $\text{Execution time ratio}_i$ is the execution time, normalized to the reference machine, for the ith 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.

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 (Figure 1.16). 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

	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.

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 who 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 means 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 relative 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 may not 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 $\text{Fraction}_{\text{enhanced}} = 0.4$
$\text{Speedup}_{\text{enhanced}} = 10$

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

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 1.3 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 processors 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 alternatives.

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.

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* (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 IC × 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., $\text{IC}_i \div$ 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, are 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:

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

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

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

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 previous 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.

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 addresses 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 straight-line 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.

Principle of Locality

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 *principle of locality:* 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.

Principle of locality 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 is 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 typically 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 to logarithmic in the number of bits per operand.

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 in servers using TPC-C as the benchmark, and finally in 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 a 10/100M bit 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

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 these common elements. First, the systems offer different levels of expandability (with the Presario system being the least expandable, the Dell systems and Sunblade 100 being moderately expandable, and 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 2. In 2001 the Athlon sold 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 Section 1.4, is at work, especially for the Compaq and Dell systems. These prices were as of July 2001.

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 in Section 1.4. (See the further discussion on price variation in the caption of Figure 1.18.)

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.

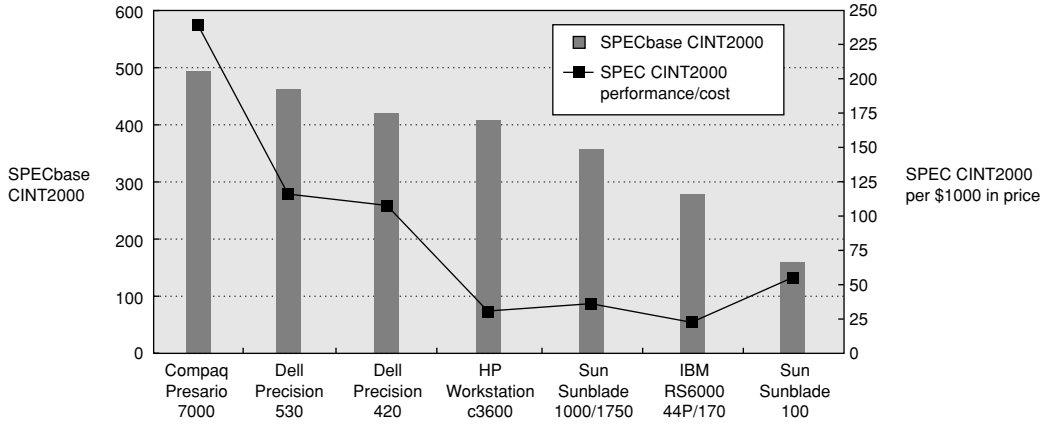

Figure 1.19 Performance and price-performance for seven systems are measured using SPEC CINT2000 as the benchmark. With the exception of the Sunblade 100 (Sun's low-end entry system), price-performance roughly parallels performance, contradicting the conventional wisdom—at least on the desktop—that higher-performance systems carry a disproportionate price premium. Price-performance is plotted as CINT2000 performance per $1000 in system cost. These performance numbers and prices were as of July 2001. The measurements are available online at *www.spec.org/osg/cpu2000/*.

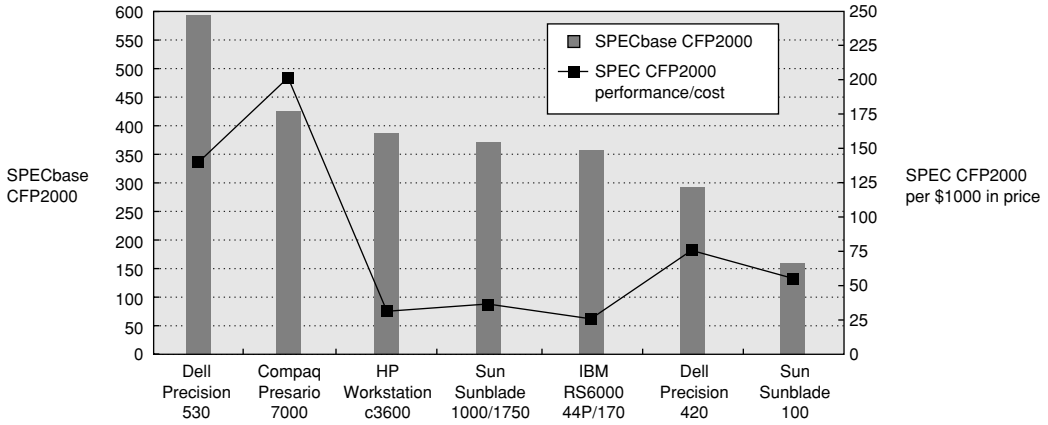

Figure 1.20 Performance and price-performance for seven systems are measured using SPEC CFP2000 as the benchmark. Price-performance is plotted as CFP2000 performance per $1000 in system cost. The dramatically improved floating-point performance of the Pentium 4 versus the Pentium III is clear in this figure. Price-performance partially parallels performance but not as clearly as in the case of the integer benchmarks. These performance numbers and prices were as of July 2001. The measurements are available online at *www.spec.org/osg/cpu2000/*.

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 make the performance of TPC-C particularly interesting. First, TPC-C is a reasonable approximation to a real OLTP application; although this makes benchmark setup 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 vendors 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 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 3, 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 price-performance to the others in the group. Chapters 6 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 three-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.

Vendor and system	CPUs	Database	OS	Price
IBM xSeries 370 c/s	280 Pentium III @ 900 MHz	Microsoft SQL Server 2000	Microsoft Windows Advanced Server	$15,543,346
Compaq AlphaServer GS 320	32 Alpha 21264 @ 1 GHz	Oracle 9i	Compaq Tru64 UNIX	$10,286,029
Fujitsu PRIMEPOWER 20000	48 SPARC64 GP @ 563 MHz	SymfoWARE Server Enterprise	Sun Solaris 8	$9,671,742
IBM pSeries 680 7017-S85	24 IBM RS64-IV @ 600 MHz	Oracle 8 v8.1.7.1	IBM AIX 4.3.3	$7,546,837
HP 9000 Enterprise Server	48 HP PA-RISC 8600 @ 552 MHz	Oracle8 v8.1.7.1	HP UX 11.i 64-bit	$8,522,104
IBM iSeries 400 840-2420	24 iSeries400 Model 840 @ 450 MHz	IBM DB2 for AS/400 V4	IBM OS/400 V4	$8,448,137
Dell PowerEdge 6400	3 Pentium III @ 700 MHz	Microsoft SQL Server 2000	Microsoft Windows 2000	$131,275
IBM xSeries 250 c/s	4 Pentium III @ 700 MHz	Microsoft SQL Server 2000	Microsoft Windows Advanced Server	$297,277
Compaq Proliant ML570 6/700 2	4 Pentium III @ 700 MHz	Microsoft SQL Server 2000	Microsoft Windows Advanced Server	$375,016
HP NetServer LH 6000	6 Pentium III @ 550 MHz	Microsoft SQL Server 2000	Microsoft Windows NT Enterprise	$372,805
NEC Express 5800/180	8 Pentium III @ 900 MHz	Microsoft SQL Server 2000	Microsoft Windows Advanced Server	$682,724
HP 9000 / L2000	4 PA-RISC 8500 @ 440 MHz	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 *www.tpc.org/*.

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

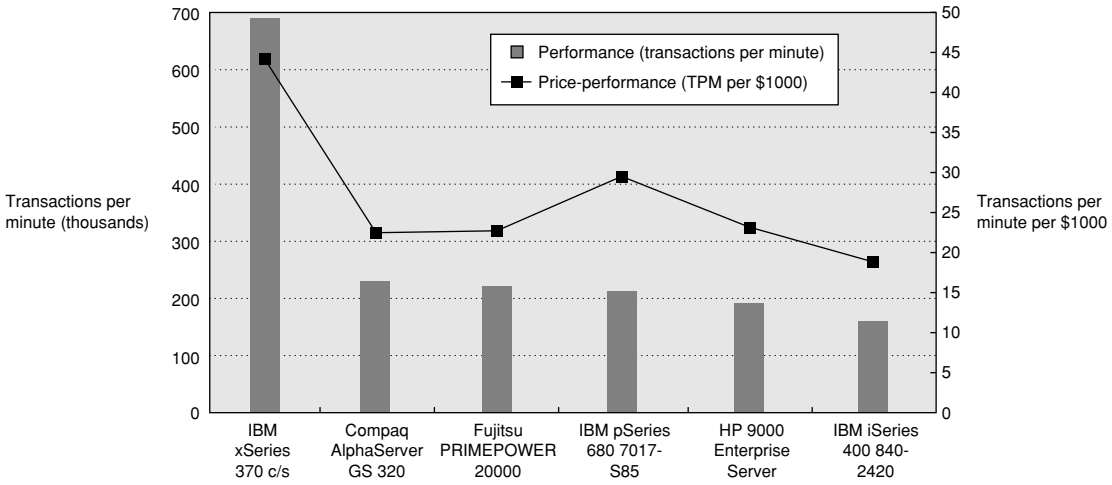

Figure 1.22 The performance (measured in thousands of transactions per minute) and the price-performance (measured in transactions per minute per $1000) are shown for six of the highest-performing systems using TPC-C as the benchmark. Interestingly, IBM occupies three of these six positions, with different hardware platforms (a cluster of Pentium IIIs, a Power III–based multiprocessor, and an AS 400–based multiprocessor.

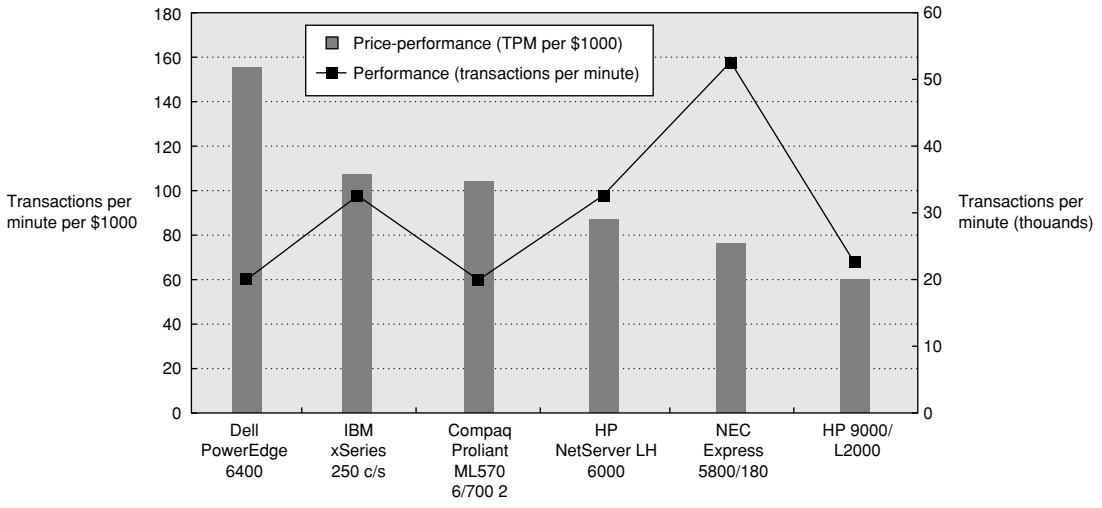

Figure 1.23 Price-performance (plotted as transactions per minute per $1000 of system cost) and overall performance (plotted as thousands of transactions per minute).

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 embedded 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 4122 is a low-end, low-power device used primarily in PDAs; in addition to the core computing functions, the 4122 provides a number of system functions, reducing the cost of the overall system.

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 4122), the clock rate is the primary factor in determining performance. For machines with widely differing cache structures (such as the presence

Processor	Instruction set	Processor clock rate (MHz)	Cache instruction/data on-chip secondary cache	Processor organization	Typical power (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	MIPS64	167	32K/32K	Pipelined: 2 issues/clock	2088	$25
NEC VR 4122	MIPS64	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 10, as we will see) and a wide range in price (roughly a factor of 4 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 VR 4122 include a hardware floating-point unit.

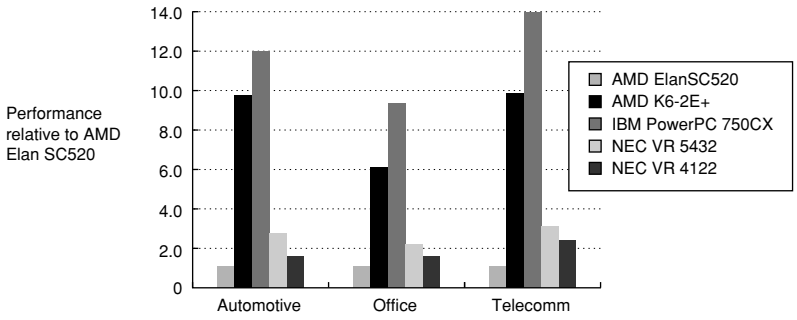

Figure 1.25 Relative performance of five different embedded processors for three of the five EEMBC benchmark suites. The performance is scaled relative to the AMD Elan SC520, so that the scores across the suites have a narrower range.

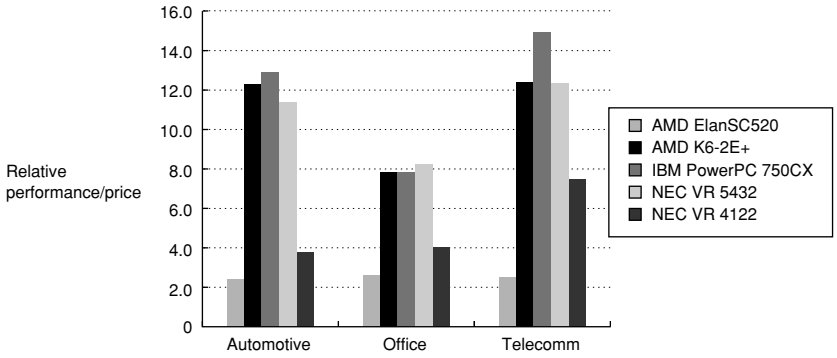

Figure 1.26 Relative price-performance of five different embedded processors for three of the five EEMBC benchmark suites, using only the price of the processor.

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 4122 at least competitive with the high-end IBM and AMD chips.

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 must 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, 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, which plots raw performance, and notice how different the results are. The NEC VR 4122 has a clear advantage in performance per watt, but is the second lowest

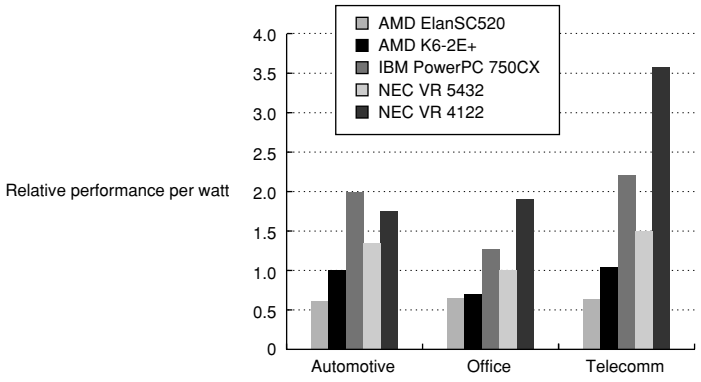

Figure 1.27 Relative performance per watt for the five embedded processors. The power is measured as typical operating power for the processor and does not include any interface chips.

performing processor! From the viewpoint of power consumption, the NEC VR 4122, 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 W typical, it is probably not suitable for most battery-based devices.

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 instruction set architecture (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

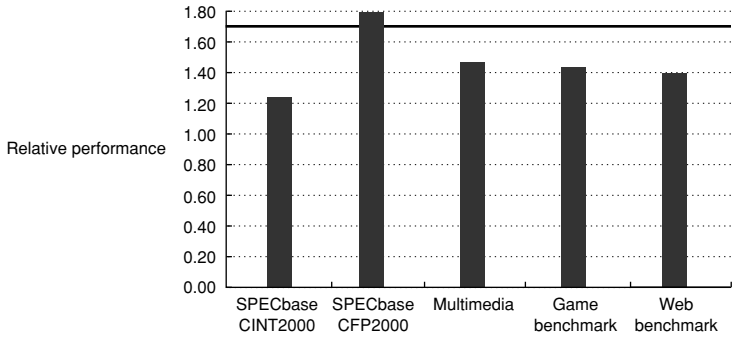

Figure 1.28 A comparison of the performance of the Pentium 4 (P4) relative to the Pentium III (P3) on five different sets of benchmark suites. The bars show the relative performance of a 1.7 GHz P4 versus a 1 GHz P3. The thick horizontal line at 1.7 shows how much faster a Pentium 4 at 1.7 GHz would 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.

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.

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 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 × 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 require 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 dif-

```
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. 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 *www.spec.org/osg/cpu2000/results/res1999q4/cpu2000-19991130-00012.html*.

ferent 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, a system tends to achieve closer to peak performance using the base flags. Similarly, as the benchmarks improve in quality, they become less susceptible to highly application-specific optimizations. Thus, the gap between peak and base, which in early times was often 20%, has narrowed.

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 run time 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 run time, 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 generation of the suite! To show how dramatically benchmarks must adapt over time, we summarize the status of the integer and FP benchmarks from SPEC89, -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 for two different machines in Figure 1.31. Both designers and users must be aware of this potentially large difference and not extrapolate performance for compiler-generated code from hand-coded 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 G 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.

Benchmark name	Integer or FP	SPEC89	SPEC92	SPEC95	SPEC2000
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	FP			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 shown. "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.

Machine	EEMBC benchmark set	Compiler-generated performance	Hand-coded performance	Ratio hand/ compiler
Trimedia 1300 @166 MHz	Consumer	23.3	110.0	4.7
BOPS Manta @ 136 MHz	Telecomm	2.6	225.8	86.8
TI TMS320C6203 @ 300 MHz	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 processors, 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.

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 two times faster
Vectorized FFT (vector lengths 64, 32, . . . , 2)	3.9 secs	7.7 secs	Cray two 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 G.

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 could have been a large fraction of both the purchase and operational costs of a system. For exam-

ple, for a medium-size database OLTP server, Microsoft OS software might run about $2000, while the Oracle software would run between $6000 and $9000 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 $8000 and $11,000. A midrange Dell server with 512 MB 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 G).
- 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. We 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 machines designed for a given class of applications. In such cases, the use of MIPS is less harmful since relative differences among MIPS ratings of machines with the same architecture and the same applications are more 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 72, 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 run time 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 A 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.

Chapter 6 focuses on 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.

In Chapters 7 and 8, 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 7 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 7, our focus is on examining the performance of disk storage systems for typical I/O-intensive workloads, like the OLTP 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 8 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 8 also describes clusters, which are growing in importance due to their suitability and efficiency for database and Web server applications.

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 on historical perspective closes each chapter in the text. This section provides historical background on some of the key ideas presented in the chapter. We 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 ten-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 (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 remained 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 [Burks, Goldstine, and von Newmann 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 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 World 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, Howlett, and Rota 1980]. The second, much larger and electronic machine, conceived and designed by Newman and Flowers, was called COLOSSUS [see Randall in Metropolis, Howlett, and Rota 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, Howlett, and Rota 1980], which was founded after the war to attempt to commercialize on the key ideas. ERA built 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, Howlett, and Rota 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, was 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 Atanasoff'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, Howlett, and Rota 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 follow-on machines, the 650, 704, and 705 (delivered in 1954 and 1955) were significant successes, each selling from 132 to 1800 computers.

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 pio-

neers. 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 G (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 layperson 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 instruction 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, 43–49.

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.

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).

Metropolis, N., J. Howlett, and G-C Rota, eds. [1980]. *A History of Computing in the Twentieth Century,* Academic Press, New York.

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).

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. Solutions to the "starred" exercises appear in Appendix B.

1.1 [15/15/15/15] <1.3, 1.4, 7.2> Computer system designers must be alert to the rapid change of computer technology. To see one example of how radically change can affect design, consider the evolution of DRAM and magnetic disk technologies since publication of the first edition of this text in 1990. At that time DRAM density had been improving for 10 years at a rate of about 60% per year, giving rise every third year to a new generation of DRAM chips with four times more capacity than before. Magnetic disk data recording density had been improving for 30 years at nearly 30% per year, doubling every three years.

a. [15] <1.3> The first edition posed a question much like this. Assume that cost per megabyte for either type of storage is proportional to density, that 1990 is the start of the 4M bit DRAM generation, and that in 1990 DRAM costs 20 times more per megabyte than disk. Using the well-established historic density improvement rates, create a table showing projected relative cost of each DRAM generation and disk from 1990 for six generations. What conclusion can be drawn about the future of disk drives in computer designs and about the magnetic disk industry from this projection?

b. [15] <1.4, 7.2> The conclusion supported by the result from part (a) is far from today's reality. Shortly before 1990 the change from inductive heads to thin film, and then magnetoresistive heads, allowed magnetic disk recording density to begin a 60% annual improvement trend, matching DRAM. Since about 1997, giant magnetoresistive effect heads have upped the rate to 100% per year, and, available to the mass market in 2001, antiferromagnetically coupled recording media should support or improve that rate for several years. Using data from Figures 1.5 and 7.4, plot the actual ratio of DRAM to disk price per unit of storage for each DRAM generation (3-year intervals) starting in 1983. Compare your answer with part (a) by including those data points on the graph. Assume that DRAM storage is built from the then-available chip size with the lowest cost per bit and that disk cost is the median cost for that year. Note that 1 GB = 1000 MB. Ignore the cost of any packaging, support hardware, and control hardware needed to incorporate DRAM and disk into a computer system.

c. [15] <1.3> Not only price, but disk physical volume and mass improve with recording density. Today's standard laptop computer disk drive bay is 10 cm long and 7 cm wide. Assume that a 100 MB disk in 1990 occupied 500 cc (cubic centimeters) and massed 1000 g (grams). If disk volume and mass had improved only 30% per year since 1990, what would the height (neglect mechanical constraints on disk drive shape) and mass of a 30 GB laptop computer disk be today? For comparison, actual typical height and mass values for 2001 are 1.25 cm and 100 g.

d. [15] <1.3, 1.4> Increasing disk recording density expands the range of software applications possible at a given computer price point. High-quality desktop digital video editing capability is available in 2001 on a $1000 PC. Five minutes of digital video consumes about 1 GB of storage, so the 20 GB disk of the PC in Figure 1.9 provides reasonable capacity. If disk density had improved only at 30% per year since 1990, but other PC component costs shown in Figure 1.9 were unchanged and the ratio of retail price to component cost given in Figure 1.10 was unaffected, approximately how much more would a desktop video PC cost in 2001?

1.2 [20/10/10/10/15] <1.6> In this exercise, assume that we are considering enhancing a machine by adding vector hardware to it. When a computation is run in vector mode on the vector hardware, it is 10 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 G, 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 the computation run time is spent in vector mode if a speedup of 2 is achieved?

d. [10] <1.6> What percentage of vectorization is needed to achieve one-half the maximum speedup attainable from using vector mode?

e. [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 hardware 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 as doubling vector hardware speed? Which investment would you recommend?

1.3 [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?

✪ 1.4 [12/10/Discussion] <1.6> Amdahl's Law implies that the ultimate goal of high-performance computer system design should be an enhancement that offers arbitrarily large speedup for all of the task time. Perhaps surprisingly, this goal can be approached quite closely with real computers and tasks. Section 3.5 describes how some branch instructions can, with high likelihood, be executed in zero time with a hardware enhancement called a branch-target buffer. Arbitrarily large speedup can be achieved for complex computational tasks when more efficient algorithms are developed. A classic example from the field of digital signal processing is the discrete Fourier transform (DFT) and the more efficient fast Fourier transform (FFT). How these two transforms work is not important here. All we need to know is that they compute the same result, and with an input of n floating-point data values, a DFT algorithm will execute approximately n^2 floating-point instructions, while the FFT algorithm will execute approximately $n \log_2 n$ floating-point instructions.

a. [12] <1.6> Ignore instructions other than floating point. What is the speedup gained by using the FFT instead of the DFT for an input of $n = 2^k$ floating-point values in the range $8 \le n \le 1024$ and also in the limit as $n \to \infty$?

b. [10] <1.6> When $n = 1024$, what is the percentage reduction in the number of executed floating-point instructions when using the FFT rather than the DFT?

c. [Discussion] <1.6> Despite the speedup achieved by processors with a branch-target buffer, not only do processors without such a buffer remain in production, new processor designs without this enhancement are still developed. Yet, once the FFT became known, the DFT was abandoned. Certainly speedup is desirable. What reasons can you think of to explain this asymmetry in use of a hardware and a software enhancement, and what does your answer say about the economics of hardware and algorithm technologies?

1.5 [15] <1.6> Show that the problem statements in the examples on pages 42 and 44 describe identical situations and equivalent design alternatives.

✪ 1.6 [15] <1.9> Dhrystone is a well-known integer benchmark. Computer A is measured to perform D_A executions of the Dhrystone benchmark per second, and to achieve a millions of instructions per second rate of $MIPS_A$ while doing Dhrystone. Computer B is measured to perform D_B executions of the Dhrystone benchmark per second. What is the fallacy in calculating the MIPS rating of computer B as $MIPS_B = MIPS_A \times (D_B / D_A)$?

1.7 [15/15/8] <1.9> A certain benchmark contains 195,578 floating-point operations, with the details shown in Figure 1.33.

The benchmark was run on an embedded processor after compilation with optimization turned on. The embedded processor is based on a current RISC processor that includes floating-point function units, but the embedded processor does not include floating point for reasons of cost, power consumption, and lack of need for floating point by the target applications. The compiler allows floating-point instructions to be calculated with the hardware units or using software routines, depending on compiler flags. The benchmark took 1.08 seconds on the

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 Occurrences of floating-point operations.

RISC processor and 13.6 seconds using software on its embedded version. Assume that the CPI using the RISC processor was measured to be 10, while the CPI of the embedded version of the processor was measured to be 6.

a. [15] <1.9> What is the total number of instructions executed for both runs?

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

c. [8] <1.9> On the average, how many integer instructions does it take to perform a floating-point operation in software?

1.8 [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 heat generated by the die. For example, a *plastic quad flat pack* (PQFP) dissipating less than 1 W, with 208 or fewer pins, and containing a die up to 1 cm on a side costs $2 in 2001. 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.

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 19. Assume a defect density of 0.5 defect per cm^2, a wafer yield of 95%, and $\alpha = 4$.

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> Using the additional assumptions 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.

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 0.3 per cm^2 and assuming that defect densities are 1.0 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 21064C with six levels of interconnect, estimate the cost of working, packaged, and tested die if $\alpha = 4$ and if $\alpha = 6$. Assume a defect density of 0.8 defects per cm^2.

Microprocessor	Die area (mm^2)	Pins	Technology	Estimated wafer cost ($)	Package
Alpha 21264C	115	524	CMOS, 0.18µ, 6M	4700	CLGA
Power3-II	163	1088	CMOS, 0.22µ, 6M	4000	SLC
Itanium	300	418	CMOS, 0.18µ, 6M	4900	PAC
MIPS R14000	204	527	CMOS, 0.25µ, 4M	3700	CPGA
UltraSPARC III	210	1368	CMOS, 0.15µ, 6M	5200	FC-LGA

Figure 1.34 Characteristics of microprocessors. About half of the pins are for power and ground connections. The technology entry is the process type, line width, and number of interconnect levels.

Package type	Pin count	Package cost ($)	Test time (secs)	Test cost per hour ($)
PAC	< 500	20	30	400
SLC	< 1100	20	20	420
Grid array (CLGA, CPGA, or FC-LGA)	< 500	20	20	400
Grid array (CLGA, CPGA, or FC-LGA)	< 1000	25	25	440
Grid array (CLGA, CPGA, or FC-LGA)	< 1500	30	30	480

Figure 1.35 Package and test characteristics.

1.9 [20/20] <1.4> On page 20 the concluding discussion about the die cost model claims that, for realistic die sizes and defect densities, die cost is better modeled as a function of (roughly) the die area squared rather than to the fourth power.

a. [20] <1.4> Using the model and a spreadsheet, determine the cost of dies ranging in area from 0.5 to 4 cm^2 and assuming a defect density of 0.6 and $\alpha = 4$. Next, use a mathematical analysis tool for fitting polynomial curves to fit the (die area, die cost) data pairs you computed in the spreadsheet. What is the lowest degree polynomial that is a close fit to the data?

b. [20] <1.4> Suppose defect densities were much higher: say, 2 defects per cm^2. Now what is lowest degree polynomial that is a close fit?

✪ 1.10 [15/15/10] <1.5, 1.9> Assume the two programs in Figure 1.15 each execute 100 million floating-point operations during execution on each of the three machines. If performance is expressed as a rate, then the average that tracks total execution time is the *harmonic mean,*

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

where Rate_i is a function of $1/\text{Time}_i$, the execution time for the ith of n programs in the workload.

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?

1.11 [12] <1.5> One reason people may incorrectly summarize rate data using 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?

1.12 [12] <1.5> For reasons similar to those in Exercise 1.11, some people use arithmetic mean instead of harmonic mean (see the definition of harmonic mean in Exercise 1.10). 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?

✪ 1.13 [10/10/10/10] <1.5> Sometimes we have a set of computer performance measurements that range from very slow to very fast execution. A single statistic, such as a mean, may not capture a useful sense of the data set as a whole. For example, the CPU pipeline and hard disk subsystem of a computer execute their respective basic processing steps at speeds that differ by a factor of typically 10^7. This is a speed difference in excess of that between a jet airliner in cruising flight (~1000 kilometers per hour) and a snail gliding on the long, thin leaf of an agapanthus (perhaps 1 meter per hour). Let's look at what happens when measurements with such a large range are summarized by a single number.

a. [10] <1.5> What are the arithmetic means of two sets of benchmark measurements, one with nine values of 10^7 and one value of 1 and the other set with nine values of 1 and one value of 10^7? How do these means compare with the data set medians? Which outlying data point affects the arithmetic mean more, a large or a small value?

b. [10] <1.5> What are the harmonic means (see Exercise 1.10 for the definition of harmonic mean) of the two sets of measurements specified in part (a)? How do these means compare with the data set medians? Which outlying data point affects the harmonic mean more, a large or a small value?

c. [10] <1.5> Which mean, arithmetic or harmonic, produces a statistic closest to the median?

d. [10] <1.5> Repeat parts (a) and (b) for two sets of 10 benchmark measurements with the outlying value only a factor of 2 larger or smaller. How representative of the entire set do the arithmetic and harmonic mean statistics seem for this narrow range of performance values?

1.14 [15/15] <1.5> A spreadsheet is useful for performing the computations of this exercise. Some of the results from the SPEC2000 Web site (*www.spec.org*) are shown in Figure 1.36. The *reference time* is the execution time for a particular computer system chosen by SPEC as a performance reference for all other tested systems. The *base ratio* is simply the run time for a benchmark divided into the reference time for that benchmark. The SPECfp_base2000 statistic is computed as the geometric mean of the base ratios. Let's see how a weighted arithmetic mean compares.

a. [15] <1.5> Calculate the weights for a workload so that running times on the reference computer will be equal for each of the 14 benchmarks in Figure 1.36.

SPEC CFP2000 program name	Reference time	Base ratio		
		Compaq AlphaServer ES40 Model 6/667	IBM eServer pSeries 640	Intel VC820
168.wupwise	1600	458	307	393
171.swim	3100	1079	227	406
172.mgrid	1800	525	284	246
173.applu	2100	386	311	244
177.mesa	1400	502	273	535
178.galgel	2900	445	380	295
179.art	2600	1238	924	379
183.equake	1300	220	528	233
187.facerec	1900	677	215	296
188.ammp	2200	405	272	283
189.lucas	2000	639	261	312
191.fma3d	2100	472	305	282
200.sixtrack	1100	273	205	169
301.apsi	2600	445	292	345
SPECfp_base2000 (geometric mean)		500	313	304

Figure 1.36 SPEC2000 performance for SPEC CFP2000. *Reference time* for each program is for a particular Sun Microsystems Ultra 10 computer configuration. *Base ratio* is the measured execution time of an executable generated by conservative compiler optimization, which is required to be identical for each program, divided into the reference time and is expressed as a percentage. SPECfp_base2000 is the geometric mean of the 14 base ratio values; it would be 100 for the reference computer system. The Compaq AlphaServer ES40 6/667 uses a 667 MHz Alpha 21164A microprocessor and an 8 MB off-chip tertiary cache. The IBM eServer pSeries 640 uses a 375 MHz Power3-II CPU and a 4 MB off-chip secondary cache. The Intel VC820 uses a 1000 MHz Pentium III processor with a 256 KB on-chip secondary cache. Data are from the SPEC Web site (*www.spec.org*).

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.

1.15 [15/20/15] <1.5> "The only consistent and reliable measure of performance is the execution time of real programs" [page 25].

a. [15] <1.5> For the execution time of a real program on a given computer system to have a meaningful value, two conditions must be satisfied. One has to do with the conditions within the computer system at the time of measurement, and the other has to do with the measured program itself. What are the conditions?

b. [20] <1.5> Programs such as operating systems, Web servers, device drivers, and TCP/IP stacks are intended to either not terminate or terminate only upon

an exceptional condition. Is throughput (work per unit time) a consistent and reliable performance measure for these programs? Why, or why not?

c. [15] <1.5> The fundamental unit of work that is of interest for programs such as Web servers and database systems is the transaction. Many computer systems are able to pipeline the processing of transactions, thus overlapping transaction execution times. What performance measurement error does the use of throughput rather than transaction execution time avoid?

✪ 1.16 [15/15/15] <1.6> Three enhancements with the following speedups are proposed for a new architecture:

$\text{Speedup}_1 = 30$

$\text{Speedup}_2 = 20$

$\text{Speedup}_3 = 15$

Only one enhancement is usable at a time.

a. [15] <1.6> If enhancements 1 and 2 are each usable for 25% of the time, what fraction of the time must enhancement 3 be used to achieve an overall speedup of 10?

b. [15] <1.6> Assume the enhancements can be used 25%, 35%, and 10% of the time for enhancements 1, 2, and 3, respectively. For what fraction of the reduced execution time is no enhancement in use?

c. [15] <1.6> Assume, for some benchmark, the possible 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?

1.17 [10/10/10/15/10] <1.6, 1.9> Your company has a benchmark that is considered representative of your typical applications. An embedded processor under consideration to support your task does not have a floating-point unit and must emulate each floating-point instruction by a sequence of integer instructions. This processor is rated at 120 MIPS on the benchmark. A third-party vendor offers a compatible coprocessor to boost performance. That coprocessor executes each floating-point instruction in hardware (i.e., no emulation is necessary). The processor/coprocessor combination 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 one floating-point instruction.

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

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

a. [10] <1.6, 1.9> Write an equation for the MIPS rating of each configuration using the symbols above.

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

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

d. [15] <1.6, 1.9> What is the MFLOPS rating of the system with the coprocessor?

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

✪ 1.18 [10/12] <1.6, 1.9> 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.

Let's examine the effects of this weighted MFLOPS measure. The SPEC CFP2000 171.swim program runs on the Compaq AlphaServer ES40 in 287 seconds. The number of floating-point operations executed in that program are listed in Figure 1.38.

a. [10] <1.6, 1.9> What is the native MFLOPS for 171.swim on a Compaq AlphaServer ES40?

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

1.19 [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 SPEC CINT2000 176.gcc on both machines. How well do peak MIPS predict performance of 176.gcc?

Real FP operations	Normalized FP operations
Add, Subtract, Compare, Multiply	1
Divide, Square root	4
Functions (Exponentiation, 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.

Floating-point operation	Times executed
`load`	77,033,084,546
`store`	22,823,523,329
`copy`	4,274,605,803
`add`	41,324,938,303
`sub`	21,443,753,876
`mul`	31,487,066,317
`div`	1,428,275,916
`convert`	11,760,563
Total	199,827,008,653

Figure 1.38 Floating-point operations in SPEC CFP2000 171.swim.

1.20 [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 SPEC CFP2000 171.swim benchmark on both machines. How well do peak MFLOPS predict performance of 171.swim?

1.21 [20/20/25] <1.7> Vendors often sell several models of a computer that have identical hardware with the sole exception of processor clock speed. The following questions explore the influence of clock speed on performance.

a. [20] <1.7> From the collection of computers with reported SPEC CFP2000 benchmark results at *www.spec.org/osg/cpu2000/results/*, choose a set of three computer models that are identical in tested configurations (both hardware and software) except for clock speed. For each pair of models, compare the clock speedup to the SPECint_base2000 benchmark speedup. How closely does benchmark performance track clock speed? Is this consistent with the description of the SPEC benchmarks on pages 28–30?

b. [20] <1.7> Now the workload for the computers in part (a) is as follows: a user launches a word-processing program, opens the file of an existing five-page text document, checks spelling, finds no errors, and finally prints the document to an inkjet printer. Suppose the execution time for this benchmark on the slowest clock rate model is 1 minute and 30 seconds, apportioned in this way: 5 seconds to load the word-processing program and the chosen document file from disk to memory, 5 seconds for the user to invoke spell checking, 1 second for spell checking to complete, 2 seconds for the user to absorb the information that there are no spelling errors, 5 seconds for the user to initiate the printing command, 2 seconds for the printing dialog box to appear, 2 seconds for the user to accept the default printing options and command that printing proceed, 8 seconds for the printer to start, and 1 minute to print the five pages.

User think time—the time it takes for a human to respond after waiting for a computer reply in interactive use—improves significantly when the computer can respond to a command quickly because the user maintains better mental focus. Assume that for computer response times less than 2 seconds, any computer response time improvement is matched by double that amount of improvement in the human response time, bounded by a 0.5 second minimum human response time.

What is the clock speedup and word-processing benchmark speedup for each pair of computer models? Discuss the importance of a faster processor for this workload.

c. [25] <1.7> Choose a desktop computer vendor that has a Web-based store and find the price for three systems that are configured identically except for processor clock rate. What is the relative price performance for each system if the workload execution time is determined only by processor clock speed ($ per MHz)? What is the relative price performance ($ per second) for each system if, during a workload execution time total of 100 seconds on the slowest system, the processor is busy 5% of the time and other system components and/or the user are busy the other 95% of the time?

1.22 [30] <1.5, 1.7> Find results from different benchmark sets, for example, PC versus SPEC benchmarks, and compare their performance measurements for two related processors, such as the Pentium III and Pentium 4. Discuss reasons for the differences in performance.

1.23 [20] <1.5, 1.8> Assume that typical power consumption for the 667 MHz Alpha 21164A, 375 MHz Power3-II, and 1000 MHz Pentium III processors is 50, 27, and 35 W, respectively. Using data from Figure 1.36 and scaling to the performance of the Pentium III, create a graph showing the relative performance and the relative performance per watt of these three processors for 171.swim, 183.equake, 301.apsi, and SPECfp_base2000.

1.24 [25] <1.4, 1.8> Design goals for a desktop computer system besides price and performance might include reducing size and noise. Assume that room air is available for cooling. Develop a simple model, similar to the cost model of Figure 1.10, that identifies the sources of additional system demands for power caused by a watt of processor power and includes the transition from passive, convective airflow to forced airflow cooling. Develop an analogous model showing the effect of processor power on system volume. Describe the effect that processor power consumption has on system noise and size.

1.25 [Discussion] <1.5> What is an interpretation of the geometric mean of execution times? What do you think are the advantages and disadvantages of using (a) total execution times versus (b) weighted arithmetic means of execution times using equal running time on the SPARC versus (c) geometric means of ratios of speed to the SPARC (used as the reference machine by SPEC2000)?

1.26 [30] <1.5> SPEC2000 programs are often compiled at levels of optimization that are almost never used by software that is sold commercially—and sometimes using compilers that no one would use in a real product. Rerun SPEC2000 programs on machines for which you can find official ratings, but this time run binaries of the programs compiled with simple optimization and no optimization. Does relative performance change? What do you conclude about the machines? About SPEC2000?

1.27 [Discussion] <1.5> PC benchmark suites use scripts to run programs as fast as possible, that is, with no user think time, the time a real user would spend understanding the current program output before providing the next user input. Also, to be sure to exercise new features of the latest version of the benchmark program, apparently they exercise every option once. What are the disadvantages of this approach? Can you think of compiler or architecture techniques that improve performance for real users but are penalized by this style of benchmarking?

1.28 [Discussion] <1.6> Amdahl's Law makes it clear that to deliver substantial performance improvement, a design enhancement must be usable a large fraction of the time. With this principle in mind, examine the table of contents for this text, determine the major themes of computer design that are covered and the ranking of specific techniques within the major topics, and discuss the extent to which Amdahl's Law is a useful dimension on which to organize the study of computer design.