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 past half century. In 1945, there were no stored-program computers. Today, a few thousand dollars will purchase a personal computer that has more performance, more main memory, and more disk storage than a computer bought in 1965 for $1 million. This rapid rate of improvement has come both from advances in the technology used to build computers and from innovation in computer design. While 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, lowered the cost and risk of bringing out a new architecture. These changes made it possible to successively develop a new set of architectures, called RISC architectures, in the early 1980s. Since the RISC-based microprocessors reached the market in the mid 1980s, these machines have grown in performance at an annual rate of over 50%. Figure 1.1 shows this difference in performance growth rates.

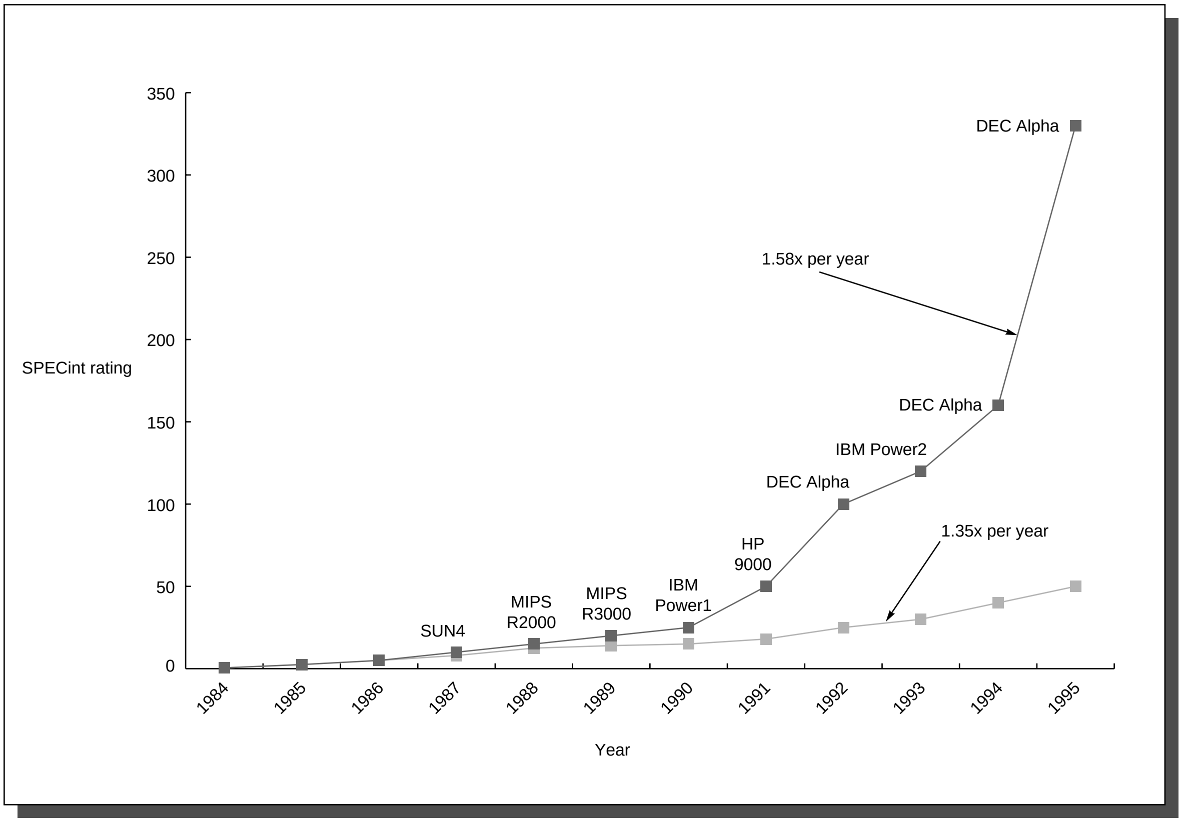

FIGURE 1.1 Growth in microprocessor performance since the mid 1980s has been substantially higher than in earlier years. This chart plots the performance as measured by the SPECint benchmarks. Prior to the mid 1980s, microprocessor performance growth was largely technology driven and averaged about 35% per year. The increase in growth since then is attributable to more advanced architectural ideas. By 1995 this growth leads to more than a factor of five difference in performance. Performance for floating-point-oriented calculations has increased even faster.

The effect of this dramatic growth rate has been twofold. First, it has significantly enhanced the capability available to computer users. As a simple example, consider the highest-performance workstation announced in 1993, an IBM Power-2 machine. Compared with a CRAY Y-MP supercomputer introduced in 1988 (probably the fastest machine in the world at that point), the workstation offers comparable performance on many floating-point programs (the performance for the SPEC floating-point benchmarks is similar) and better performance on integer programs for a price that is less than one-tenth of the supercomputer!

Second, this dramatic rate of improvement has led to the dominance of microprocessor-based computers across the entire range of the computer design. Workstations and PCs have emerged as major products in the computer industry. Minicomputers, which were traditionally made from off-the-shelf logic or from gate arrays, have been replaced by servers made using microprocessors. Mainframes are slowly being 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 1995, the difference between the highest-performance microprocessors and what would have been obtained by relying solely on technology is more than a factor of five. This text is about the architectural ideas and accompanying compiler improvements that have made this incredible growth rate possible. At the center of this dramatic revolution has been the development of a quantitative approach to computer design and analysis that uses empirical observations of programs, experimentation, and simulation as its tools. It is this style and approach to computer design that is reflected in this text.

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

1.2 The Task of a Computer Designer

The task the computer designer faces is a complex one: Determine what attributes are important for a new machine, then design a machine to maximize performance while staying within cost 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 insinuating that implementation is uninteresting or less challenging. The authors believe this view is not only incorrect, but is even responsible for mistakes in the design of new instruction sets. The architect's or designer's job is much more than instruction set design, and the technical hurdles in the other aspects of the project are certainly as challenging as those encountered in doing instruction set design. This is particularly true at the present when the differences among instruction sets are small (see Appendix C).

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 internal CPU (central processing unit—where arithmetic, logic, branching, and data transfer are implemented) design. For example, two machines with the same instruction set architecture but different organizations are the SPARCstation-2 and SPARCstation-20. *Hardware* is used to refer to the specifics of a machine. This would include 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, two versions of the Silicon Graphics Indy differ in clock rate and in detailed cache structure. 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 and performance goals. Often, they also have to determine what the functional requirements are, and this can be a major task. The requirements may be specific features, inspired by the market. Application software often drives the choice of certain functional requirements by determining how the machine will be used. If a large body of software exists for a certain instruction set architecture, the architect may decide that a new machine should implement an existing instruction set. The presence of a large market for a particular class of applications might encourage the designers to incorporate requirements that would make the machine competitive in that market. Figure 1.2 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 most common metrics involve cost and perfor-

Functional requirements	**Typical features required or supported**
Application area	Target of computer
General purpose	Balanced performance for a range of tasks (Ch 2,3,4,5)
Scientific	High-performance floating point (App A,B)
Commercial	Support for COBOL (decimal arithmetic); support for databases and transaction processing (Ch 2,7)
Level of software compatibility	Determines amount of existing software for machine
At programming language	Most flexible for designer; need new compiler (Ch 2,8)
Object code or binary compatible	Instruction set architecture is completely defined—little flexibility—but no investment needed in software or porting programs
Operating system requirements	Necessary features to support chosen OS (Ch 5,7)
Size of address space	Very important feature (Ch 5); may limit applications
Memory management	Required for modern OS; may be paged or segmented (Ch 5)
Protection	Different OS and application needs: page vs. segment protection (Ch 5)
Standards	Certain standards may be required by marketplace
Floating point	Format and arithmetic: IEEE, DEC, IBM (App A)
I/O bus	For I/O devices: VME, SCSI, Fiberchannel (Ch 7)
Operating systems	UNIX, DOS, or vendor proprietary
Networks	Support required for different networks: Ethernet, ATM (Ch 6)
Programming languages	Languages (ANSI C, Fortran 77, ANSI COBOL) affect instruction set (Ch 2)

FIGURE 1.2 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.

mance. Given some application domain, the architect can try to quantify the performance of the machine by a set of programs that are chosen to represent that application domain. Other measurable requirements may be important in some markets; reliability and fault tolerance are often crucial in transaction processing environments. Throughout this text we will focus on optimizing machine cost/performance.

In choosing between two designs, one factor that an architect must consider is design complexity. Complex designs take longer to complete, prolonging time to market. This means a design that takes longer will need to have higher performance to be competitive. The architect must be constantly aware of the impact of his design choices on the design time for both hardware and software.

In addition to performance, cost is the other key parameter in optimizing cost/performance. In addition to 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 and Computer Usage Trends

If an instruction set architecture is to be successful, it must be designed to survive changes in hardware technology, software technology, and application characteristics. The designer must be especially aware of trends in computer usage and in computer technology. After all, a successful new instruction set architecture may last decades—the core of the IBM mainframe has been in use since 1964. An architect must plan for technology changes that can increase the lifetime of a successful machine.

Trends in Computer Usage

The design of a computer is fundamentally affected both by how it will be used and by the characteristics of the underlying implementation technology. Changes in usage or in implementation technology affect the computer design in different ways, from motivating changes in the instruction set to shifting the payoff from important techniques such as pipelining or caching.

Trends in software technology and how programs will use the machine have a long-term impact on the instruction set architecture. One of the most important software trends is the increasing amount of memory used by programs and their data. The amount of memory needed by the average program has grown by a factor of 1.5 to 2 per year! This translates to a consumption of address bits at a rate of approximately 1/2 bit to 1 bit per year. This rapid rate of growth is driven both by the needs of programs as well as by the improvements in DRAM technology that continually improve the cost per bit. Underestimating address-space growth is often the major reason why an instruction set architecture must be abandoned. (For further discussion, see Chapter 5 on memory hierarchy.)

Another important software trend in the past 20 years has been the replacement of assembly language by high-level languages. This trend has resulted in a larger role for compilers, forcing compiler writers and architects to work together closely to build a competitive machine. Compilers have become the primary interface between user and machine.

In addition to this interface role, compiler technology has steadily improved, taking on newer functions and increasing the efficiency with which a program can be run on a machine. This improvement in compiler technology has included traditional optimizations, which we discuss in Chapter 2, as well as transformations aimed at improving pipeline behavior (Chapters 3 and 4) and memory system behavior (Chapter 5). How to balance the responsibility for efficient execution in modern processors between the compiler and the hardware continues to be one of the hottest architecture debates of the 1990s. Improvements in compiler technology played a major role in making vector machines (Appendix B) successful. The development of compiler technology for parallel machines is likely to have a large impact in the future.

Trends in Implementation Technology

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

- *Integrated circuit logic technology*—Transistor density increases by about 50% per year, quadrupling in just over three years. Increases in die size are less predictable, ranging from 10% to 25% per year. The combined effect is a growth rate in transistor count on a chip of between 60% and 80% per year. Device speed increases nearly as fast; however, metal technology used for wiring does not improve, causing cycle times to improve at a slower rate. We discuss this further in the next section.
- *Semiconductor DRAM*—Density increases by just under 60% per year, quadrupling in three years. Cycle time has improved very slowly, decreasing by about one-third in 10 years. Bandwidth per chip increases as the latency decreases. In addition, changes to the DRAM interface have also improved the bandwidth; these are discussed in Chapter 5. In the past, DRAM (dynamic random-access memory) technology has improved faster than logic technology. This difference has occurred because of reductions in the number of transistors per DRAM cell and the creation of specialized technology for DRAMs. As the improvement from these sources diminishes, the density growth in logic technology and memory technology should become comparable.
- *Magnetic disk technology*—Recently, disk density has been improving by about 50% per year, almost quadrupling in three years. Prior to 1990, density increased by about 25% per year, doubling in three years. It appears that disk technology will continue the faster density growth rate for some time to come. Access time has improved by one-third in 10 years. This technology is central to Chapter 6.

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 (two years of design and two years of production), key technologies, such as DRAM, change sufficiently that the designer must plan for these changes. Indeed, designers often design for the next technology, knowing that when a product begins shipping in volume that next technology may be the most cost-effective or may have performance advantages. Traditionally, cost has decreased very closely to the rate at which density increases.

These technology changes are not continuous but often occur in discrete steps. For example, DRAM sizes are always increased by factors of four because of the basic design structure. Thus, rather than doubling every 18 months, DRAM technology quadruples every three years. This stepwise change in technology leads to thresholds that can enable an implementation technique that was previously impossible. 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 eliminating chip crossings within the processor, a dramatic increase in cost/performance 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.

1.4 Cost and Trends in Cost

Although there are computer designs where costs tend to be ignored—specifically supercomputers—cost-sensitive designs are of growing importance. 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 complex. 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, specifically on the components of cost and the major trends. The Exercises and Examples use specific cost data that will change over time, though the basic determinants of cost are less time sensitive.

Entire books are written about costing, pricing strategies, and the impact of volume. This section can only introduce you to these topics by discussing some of the major factors that influence cost of a computer design and how these factors are changing over time.

The Impact of Time, Volume, Commodization, and Packaging

The cost of a manufactured computer component decreases over time even without major improvements in the basic implementation technology. The underlying principle that drives costs down is the *learning curve*—manufacturing costs decrease over time. The learning curve itself is best measured by change in *yield*—the percentage of manufactured devices that survives the testing procedure. Whether it is a chip, a board, or a system, designs that have twice the yield will have basically half the cost. Understanding how the learning curve will improve yield is key to projecting costs over the life of the product. As an example of the learning curve in action, the cost per megabyte of DRAM drops over the long term by 40% per year. A more dramatic version of the same information is shown in Figure 1.3, where the cost of a new DRAM chip is depicted over its lifetime. Between the start of a project and the shipping of a product, say two years, the cost of a new DRAM drops by a factor of between five and 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 1.3 discusses some of the long-term trends in DRAM cost.

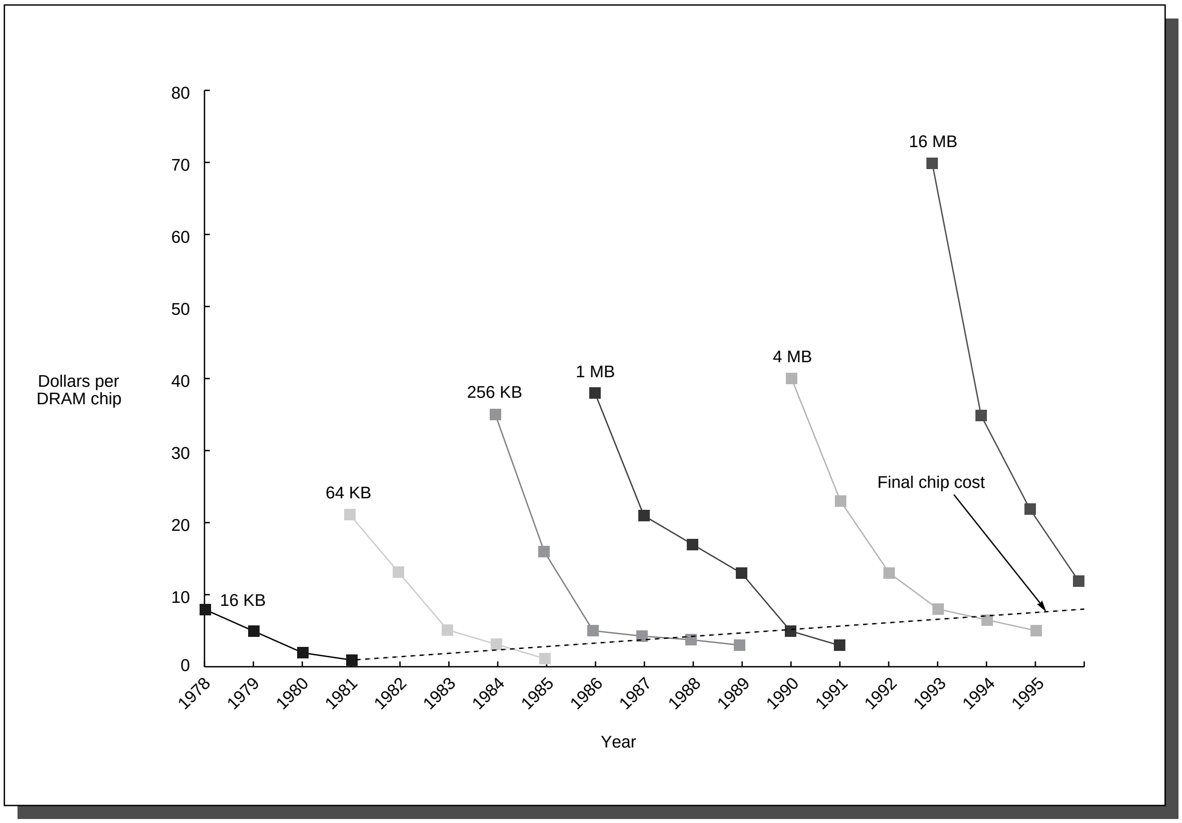

FIGURE 1.3 Prices of four generations of DRAMs over time in 1977 dollars, showing the learning curve at work. A 1977 dollar is worth about $2.44 in 1995; most of this inflation occurred in the period of 1977–82, during which the value changed to $1.61. The cost of a megabyte of memory has dropped *incredibly* during this period, from over $5000 in 1977 to just over $6 in 1995 (in 1977 dollars)! Each generation drops in constant dollar price by a factor of 8 to 10 over its lifetime. The increasing cost of fabrication equipment for each new generation has led to slow but steady increases in both the starting price of a technology and the eventual, lowest price. 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.

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, small disks, monitors, and keyboards. In the past 10 years, much of the low end of the computer business has become a commodity business focused on building IBM-compatible PCs. There are a variety of vendors that ship virtually identical products and are highly competitive. Of course, this competition decreases the gap between cost and selling price, but it also decreases cost. This occurs because a commodity market has both volume and a clear product definition. This 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.

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. We follow here the U.S. accounting approach to the costs of chips.

While 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 1.4 and 1.5). Thus the cost of a packaged integrated circuit is

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

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

FIGURE 1.4 Photograph of an 8-inch wafer containing Intel Pentium microprocessors. The die size is 480.7 mm^2 and the total number of dies is 63. (Courtesy Intel.)

FIGURE 1.5 Photograph of an 8-inch wafer containing PowerPC 601 microprocessors. The die size is 122 mm^2. The number of dies on the wafer is 200 after subtracting the test dies (the odd-looking dies that are scattered around). (Courtesy IBM.)

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

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

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 20 cm (≈ 8 inch) in diameter produces $3.14 \times 100 - (3.14 \times 20/1.41) = 269$ 1-cm dies.

EXAMPLE Find the number of dies per 20-cm wafer for a die that is 1.5 cm on a side.

ANSWER The total die area is 2.25 cm^2. Thus

$$\text{Dies per wafer} = \frac{\pi \times (20/2)^2}{2.25} - \frac{\pi \times 20}{\sqrt{2 \times 2.25}} = \frac{314}{2.25} - \frac{62.8}{2.12} = 110$$

■

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 and manufacturing defects that occur. In 1995, these values typically range between 0.6 and 1.2 per square centimeter, depending on the maturity of the process (recall the learning curve, mentioned earlier). Lastly, α is a parameter that corresponds roughly 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 $\alpha = 3.0$.

EXAMPLE Find the die yield for dies that are 1 cm on a side and 1.5 cm on a side, assuming a defect density of 0.8 per cm^2.

ANSWER The total die areas are 1 cm^2 and 2.25 cm^2. For the smaller die the yield is

$$\text{Die yield} = \left(1 + \frac{0.8 \times 1}{3}\right)^{-3} = 0.49$$

For the larger die, it is

$$\text{Die yield} = \left(1 + \frac{0.8 \times 2.25}{3}\right)^{-3} = 0.24$$

■

The bottom line is the number of good dies per wafer, which comes from multiplying dies per wafer by die yield. The examples above predict 132 good 1-cm^2 dies from the 20-cm wafer and 26 good 2.25-cm^2 dies. Most high-end microprocessors fall between these two sizes, with some being as large as 2.75 cm^2 in 1995. Low-end processors are sometimes as small as 0.8 cm^2, while processors used for embedded control (in printers, automobiles, etc.) are often just 0.5 cm^2. (Figure 1.22 on page 63 in the Exercises shows the die size and technology for several current microprocessors.) Occasionally dies become pad limited: the amount of die area is determined by the perimeter rather than the logic in the interior. This may lead to a higher yield, since defects in empty silicon are less serious!

Processing a 20-cm-diameter wafer in a leading-edge technology with 3–4 metal layers costs between \$3000 and \$4000 in 1995. Assuming a processed wafer cost of \$3500, the cost of the 1-cm^2 die is around \$27, while the cost per die of the 2.25-cm^2 die is about \$140, or slightly over 5 times the cost for a die that is 2.25 times larger.

What should a computer designer remember about chip costs? The manufacturing process dictates the wafer cost, wafer yield, α, and defects per unit area, so the sole control of the designer is die area. Since α is typically 3 for the advanced processes in use today, die costs are proportional to the fourth (or higher) power of the die area:

$$\text{Cost of die} = f\,(\text{Die area}^4)$$

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 part must be tested (to separate the good dies from the bad), packaged, and tested again after packaging. These steps all add costs. These processes and their contribution to cost are discussed and evaluated in Exercise 1.8.

Distribution of Cost in a System: An Example

To put the costs of silicon in perspective, Figure 1.6 shows the approximate cost breakdown for a color desktop machine in the late 1990s. While costs for units like DRAMs will surely drop over time from those in Figure 1.6, costs for units whose prices have already been cut, like displays and cabinets, will change very little. Furthermore, we can expect that future machines will have larger memories and disks, meaning that prices drop more slowly than the technology improvement.

The processor subsystem accounts for only 6% of the overall cost. Although in a mid-range or high-end design this number would be larger, the overall breakdown across major subsystems is likely to be similar.

System	Subsystem	Fraction of total
Cabinet	Sheet metal, plastic	1%
	Power supply, fans	2%
	Cables, nuts, bolts	1%
	Shipping box, manuals	0%
	Subtotal	**4%**
Processor board	Processor	6%
	DRAM (64 MB)	36%
	Video system	14%
	I/O system	3%
	Printed circuit board	1%
	Subtotal	**60%**
I/O devices	Keyboard and mouse	1%
	Monitor	22%
	Hard disk (1 GB)	7%
	DAT drive	6%
	Subtotal	**36%**

FIGURE 1.6 Estimated distribution of costs of the components in a low-end, late 1990s color desktop workstation assuming 100,000 units. Notice that the largest single item is memory! Costs for a high-end PC would be similar, except that the amount of memory might be 16–32 MB rather than 64 MB. This chart is based on data from Andy Bechtolsheim of Sun Microsystems, Inc. Touma [1993] discusses workstation costs and pricing.

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 with entire books written on the subject. The purpose of this section is to give you a simple introduction to what factors determine price and typical ranges for these factors.

The categories that make up price can be shown either as a tax on cost or as a percentage of the price. We will look at the information both ways. These differences between price and cost also depend on where in the computer marketplace a company is selling. To show these differences, Figures 1.7 and 1.8 on page 16 show 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 20% to 40% 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 20% to 55% of the average selling price, depending on the uniqueness of the product. Manufacturers of low-end PCs generally 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, phone order, or retail store) rather than salespeople. Third, because their products are less unique, competition is more intense, thus forcing lower prices and often lower profits, which in turn lead to a lower gross margin.

List price and average selling price are not the same. One reason for this is that companies offer volume discounts, lowering the average selling price. Also, if the product is to be sold in retail stores, as personal computers are, stores want to keep 40% to 50% of the list price for themselves. Thus, depending on the distribution system, the average selling price is typically 50% to 75% of the list price.

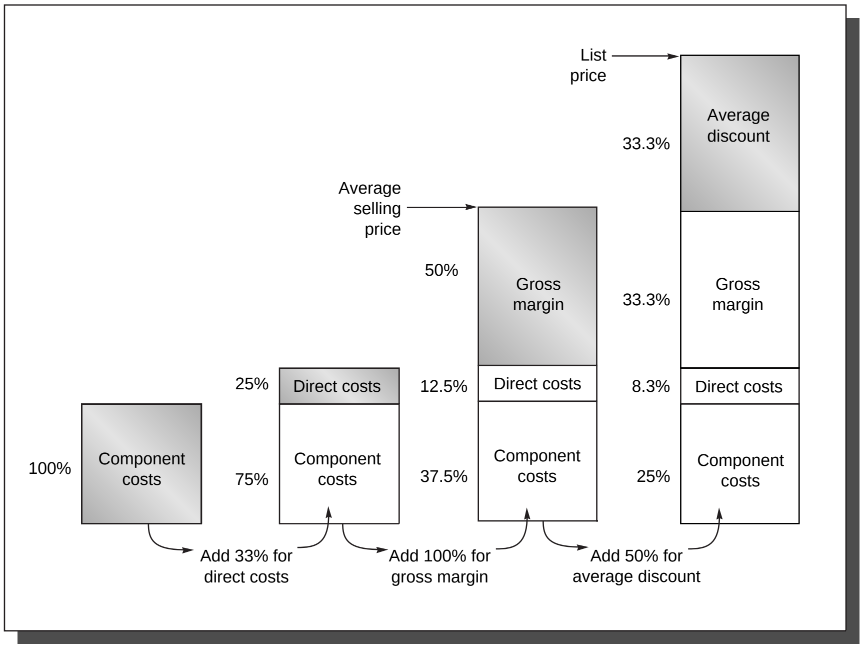

FIGURE 1.7 The components of price for a mid-range product in a workstation company. Each increase is shown along the bottom as a tax on the prior price. The percentages of the new price for all elements are shown on the left of each column.

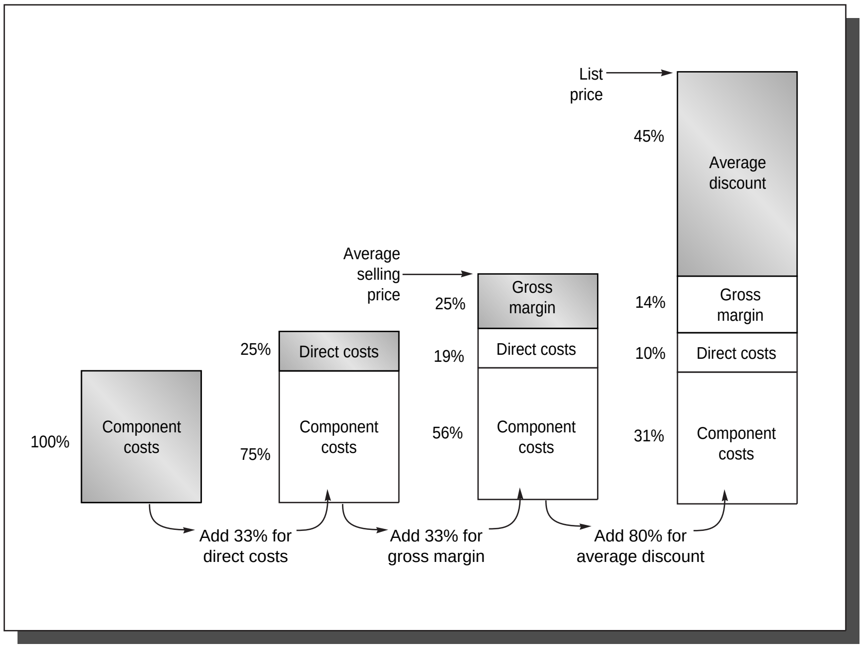

FIGURE 1.8 The components of price for a desktop product in a personal computer company. A larger average discount is used because of indirect selling, and a lower gross margin is required.

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

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

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

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

The issue of cost and cost/performance is a complex one. There is no single target for computer designers. At one extreme, *high-performance design* spares no cost in achieving its goal. Supercomputers have traditionally fit into this category. At the other extreme is *low-cost design*, where performance is sacrificed to achieve lowest cost. Computers like the IBM PC clones belong here. Between these extremes is *cost/performance design,* where the designer balances cost versus performance. Most of the workstation manufacturers 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. Even the supercomputer manufacturers have found that cost plays an increasing role. 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 computer user may say a computer is faster when a program runs in less time, while the computer center manager may say a computer is faster when it completes more jobs in an hour. The computer user is interested in reducing *response time*—the time between the start and the completion of an event—also referred to as *execution time*. The manager of a large data processing center may be interested in increasing *throughput*—the total amount of work done in a given time.

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

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

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

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

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

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

Whether we are interested in throughput or response time, the key measurement is time: The computer that performs the same amount of work in the least time is the fastest. The difference is whether we measure one task (response time) or many tasks (throughput). Unfortunately, time is not always the metric quoted in comparing the performance of computers. A number of popular measures have been adopted in the quest for a easily understood, universal measure of computer performance, with the result that a few innocent terms have been shanghaied from their well-defined environment and forced into a service for which they were never intended. The authors' position is that the only consistent and reliable measure of performance is the execution time of real programs, and that all proposed alternatives to time as the metric or to real programs as the items measured have eventually led to misleading claims or even mistakes in computer design. The dangers of a few popular alternatives are shown in *Fallacies and Pitfalls*, section 1.8.

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 concentrate on CPU performance in this chapter.

Choosing Programs to Evaluate Performance

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

Rick Richardson, *Clarification of Dhrystone* (1988)

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

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

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

1. *Real programs*—While 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 TeX, and CAD tools like Spice. Real programs have input, output, and options that a user can select when running the program.

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

3. *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.

4. *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 1.8 on page 44. 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 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, while 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 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. In the remainder of this section, we discuss the strengths and weaknesses of different methods for summarizing performance.

Benchmark suites are made of collections of programs, some of which may be kernels, but many of which are typically real programs. Figure 1.9 describes the programs in the popular SPEC92 benchmark suite used to characterize performance in the workstation and server markets.The programs in SPEC92 vary from collections of kernels (nasa7) to small, program fragments (tomcatv, ora, alvinn, swm256) to applications of varying size (spice2g6, gcc, compress). We will see data on many of these programs throughout this text. In the next subsection, we show how a SPEC92 report describes the machine, compiler, and OS configuration, while in section 1.8 we describe some of the pitfalls that have occurred in attempting to develop the benchmark suite and to prevent the benchmark circumvention that makes the results not useful for comparing performance among machines.

Benchmark	Source	Lines of code	Description
espresso	C	13,500	Minimizes Boolean functions.
li	C	7,413	A lisp interpreter written in C that solves the 8-queens problem.
eqntott	C	3,376	Translates a Boolean equation into a truth table.
compress	C	1,503	Performs data compression on a 1-MB file using Lempel-Ziv coding.
sc	C	8,116	Performs computations within a UNIX spreadsheet.
gcc	C	83,589	Consists of the GNU C compiler converting preprocessed files into optimized Sun-3 machine code.
spice2g6	FORTRAN	18,476	Circuit simulation package that simulates a small circuit.
doduc	FORTRAN	5,334	A Monte Carlo simulation of a nuclear reactor component.
mdljdp2	FORTRAN	4,458	A chemical application that solves equations of motion for a model of 500 atoms. This is similar to modeling a structure of liquid argon.
wave5	FORTRAN	7,628	A two-dimensional electromagnetic particle-in-cell simulation used to study various plasma phenomena. Solves equations of motion on a mesh involving 500,000 particles on 50,000 grid points for 5 time steps.
tomcatv	FORTRAN	195	A mesh generation program, which is highly vectorizable.
ora	FORTRAN	535	Traces rays through optical systems of spherical and plane surfaces.
mdljsp2	FORTRAN	3,885	Same as mdljdp2, but single precision.
alvinn	C	272	Simulates training of a neural network. Uses single precision.
ear	C	4,483	An inner ear model that filters and detects various sounds and generates speech signals. Uses single precision.
swm256	FORTRAN	487	A shallow water model that solves shallow water equations using finite difference equations with a 256×256 grid. Uses single precision.
su2cor	FORTRAN	2,514	Computes masses of elementary particles from Quark-Gluon theory.
hydro2d	FORTRAN	4,461	An astrophysics application program that solves hydrodynamical Navier Stokes equations to compute galactical jets.
nasa7	FORTRAN	1,204	Seven kernels do matrix manipulation, FFTs, Gaussian elimination, vortices creation.
fpppp	FORTRAN	2,718	A quantum chemistry application program used to calculate two electron integral derivatives.

FIGURE 1.9 The programs in the SPEC92 benchmark suites. The top six entries are the integer-oriented programs, from which the SPECint92 performance is computed. The bottom 14 are the floating-point-oriented benchmarks from which the SPECfp92 performance is computed.The floating-point programs use double precision unless stated otherwise. The amount of nonuser CPU activity varies from none (for most of the FP benchmarks) to significant (for programs like gcc and compress). In the performance measurements in this text, we use the five integer benchmarks (excluding sc) and five FP benchmarks: doduc, mdljdp2, ear, hydro2d, and su2cor.

Reporting Performance Results

The guiding principle of reporting performance measurements should be *reproducibility*—list everything another experimenter would need to duplicate the results. Compare descriptions of computer performance found in refereed scientific journals to descriptions of car performance found in magazines sold at supermarkets. Car magazines, in addition to supplying 20 performance metrics, list all optional equipment on the test car, the types of tires used in the performance test, and the date the test was made. Computer journals may have only seconds of execution labeled by the name of the program and the name and model of the computer—spice takes 187 seconds on an IBM RS/6000 Powerstation 590. Left to the reader's imagination are program input, version of the program, version of compiler, optimizing level of compiled code, version of operating system, amount of main memory, number and types of disks, version of the CPU—all of which make a difference in performance. In other words, car magazines have enough information about performance measurements to allow readers to duplicate results or to question the options selected for measurements, but computer journals often do not!

A SPEC benchmark report requires a fairly complete description of the machine, the compiler flags, as well as the publication of both the baseline and optimized results. As an example, Figure 1.10 shows portions of the SPECfp92 report for an IBM RS/6000 Powerstation 590. 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.

The importance of performance on the SPEC benchmarks motivated vendors to add many benchmark-specific flags when compiling SPEC programs; 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 1.10 shows the parameters for the baseline performance; in section 1.8, *Fallacies and Pitfalls*, we'll see the tuning parameters for the optimized performance runs on this machine.

Comparing and Summarizing Performance

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

Hardware		**Software**	
Model number	Powerstation 590	O/S and version	AIX version 3.2.5
CPU	66.67 MHz POWER2	Compilers and version	C SET++ for AIX C/C++ version 2.1 XL FORTRAN/6000 version 3.1
FPU	Integrated	Other software	See below
Number of CPUs	1	File system type	AIX/JFS
Primary cache	32KBI+256KBD off chip	System state	Single user
Secondary cache	None		
Other cache	None		
Memory	128 MB		
Disk subsystem	2x2.0 GB		
Other hardware	None		
SPECbase_fp92 tuning parameters/notes/summary of changes:			
FORTRAN flags: -O3 -qarch=pwrx -qhsflt -qnofold -bnso -BI:/lib/syscalss.exp			
C flags: -O3 -qarch=pwrx -Q -qtune=pwrx -qhssngl -bnso -bI:/lib/syscalls.exp			

FIGURE 1.10 The machine, software, and baseline tuning parameters for the SPECfp92 report on an IBM RS/6000 Powerstation 590. SPECfp92 means that this is the report for the floating-point (FP) benchmarks in the 1992 release (the earlier release was renamed SPEC89) The top part of the table describes the hardware and software. The bottom describes the compiler and options used for the baseline measurements, which must use one compiler and one set of flags for all the benchmarks in the same language. The tuning parameters and flags for the tuned SPEC92 performance are given in Figure 1.18 on page 49. Data from SPEC [1994].

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 1.11, taken from one of the articles, is an example of the confusion that can arise.

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

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.

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 *i*th program of a total of *n* in the workload. 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/ Time_i, the execution time for the *i*th of *n* programs 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 *i*th program in the workload and Time_i is the execution time of that program. Figure 1.12 shows the data from Figure 1.11 with three different weightings, each proportional to the execution time of a workload with a given mix. The *weighted harmonic mean* of rates will show the same relative performance as the weighted arithmetic means of execution times. The definition is

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

	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 1.12 Weighted arithmetic mean execution times using three weightings. 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 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

$$w_i = \frac{1}{\text{Time}_i \times \displaystyle\sum_{j=1}^{n} \left(\frac{1}{\text{Time}_j}\right)}.$$

Normalized Execution Time and the Pros and Cons of Geometric Means

A second approach to unequal mixture of programs in the workload is to normalize execution times to a reference machine and then take the average of the normalized execution times. This is the approach used by the SPEC benchmarks, where a base time on a VAX-11/780 is used for reference. This measurement gives a warm fuzzy feeling, because it suggests that performance of new programs can be predicted by simply multiplying this number times its performance on the reference machine.

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

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

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

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

As a result, taking either the ratio of the means or the mean of the ratios yields the same result. In contrast to arithmetic means, geometric means of normalized execution times are consistent no matter which machine is the reference. Hence, the arithmetic mean should *not* be used to average normalized execution times. Figure 1.13 shows some variations using both arithmetic and geometric means of normalized times.

	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 1.13 Execution times from Figure 1.11 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, while 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.

Because the weightings in weighted arithmetic means are set proportionate to execution times on a given machine, as in Figure 1.12, 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 1.13 suggest that for programs P1 and P2 the performance of machines A and B is the same, yet this would only be true for a workload that ran program P1 100 times for every occurrence of program P2 (see Figure 1.12 on page 26). The total execution time for such a workload suggests that machines A and B are about 50% faster than machine C, in contrast to the geometric mean, which says machine C is faster than A and B! In general there is *no workload* for three or more machines that will match the performance predicted by the geometric means of normalized execution times. Our original reason for examining geometric means of normalized performance was to avoid giving equal emphasis to the programs in our workload, but is this solution an improvement?

An additional drawback of using geometric mean as a method for summarizing performance for a benchmark suite (as SPEC92 does) is that it encourages hardware and software designers to focus their attention on the benchmarks where performance is easiest to improve rather than on the benchmarks that are slowest. For example, if some hardware or software improvement can cut the running time for a benchmark from 2 seconds to 1, the geometric mean will reward those designers with the same overall mark that it would give to designers that improve the running time on another benchmark in the suite from 10,000 seconds to 5000 seconds. Of course, everyone interested in running the second program thinks of the second batch of designers as their heroes and the first group as useless. Small programs are often easier to "crack," obtaining a large but unrepresentative performance improvement, and the use of geometric mean rewards such behavior more than a measure that reflects total running time.

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

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 that runs 10 times faster than the original machine but is only usable 40% 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 1.2 to see how wrong.)

Amdahl's Law can serve as a guide to how much an enhancement will improve performance and how to distribute resources to improve cost/performance. The goal, clearly, is to spend resources proportional to where time is spent. We can also use Amdahl's Law to compare two design alternatives, as the following Example shows.

EXAMPLE Implementations of floating-point (FP) square root vary significantly in performance. Suppose FP square root (FPSQR) is responsible for 20% of the execution time of a critical benchmark on a machine. One proposal is to add FPSQR hardware that will speed up this operation by a factor of 10. The other alternative is just to try to make all FP instructions run faster; FP instructions are responsible for a total of 50% of the execution time. The design team believes that they can make all FP instructions run two 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:

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

$$\text{Speedup}_{\text{FP}} = \frac{1}{(1-0.5)+\frac{0.5}{2.0}} = \frac{1}{0.75} = 1.33$$

Improving the performance of the FP operations overall is 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

Most 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., 2 ns) or by its rate (e.g., 500 MHz). 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):

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

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:

$$\text{CPU time} = \text{IC} \times \text{CPI} \times \text{Clock cycle time}$$

or

$$\text{CPU time} = \frac{\text{IC} \times \text{CPI}}{\text{Clock rate}}$$

Expanding the first formula into the units of measure 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 also 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

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

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

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

and overall CPI as

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

The latter form of the CPI calculation multiplies each individual CPI_i by the fraction of occurrences of that instruction in a program. CPI_i should be measured and not just calculated from a table in the back of a reference manual since it must include 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 easier to obtain.

EXAMPLE Suppose we have made the following measurements:

Frequency of FP operations = 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 reduce the CPI of FPSQR to 2 or to reduce the average CPI of all FP operations to 2. 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

$$\text{CPI}_{\text{new FP}} = (75\% \times 1.33) + (25\% \times 2.0) = 1.5$$

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

$$\begin{aligned}\text{Speedup}_{\text{new FP}} &= \frac{\text{CPU time}_{\text{original}}}{\text{CPU time}_{\text{new FP}}} = \frac{\text{IC} \times \text{Clock cycle} \times \text{CPI}_{\text{original}}}{\text{IC} \times \text{Clock cycle} \times \text{CPI}_{\text{new FP}}} \\ &= \frac{\text{CPI}_{\text{original}}}{\text{CPI}_{\text{new FP}}} = \frac{2.00}{1.5} = 1.33\end{aligned}$$

Happily, this is the same speedup we obtained using Amdahl’s Law on page 31. ■

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

Measuring the Components of CPU Performance

To use the CPU performance equation to determine performance, we need measurements of the individual components of the equation. Building and using tools to measure aspects of a design is a large part of a designer's job—at least for designers who base their decisions on quantitative principles!

To determine the clock cycle, we need only determine one number. Of course, this is easy for an existing CPU, but estimating the clock cycle time of a design in progress is very difficult. Low-level tools, called *timing estimators* or *timing verifiers*, are used to analyze the clock cycle time for a completed design. It is much more difficult to estimate the clock cycle time for a design that is not completed, or for an alternative for which no design exists. In practice, designers determine a target cycle time and estimate the impact on cycle time by examining what they believe to be the critical paths in a design. The difficulty is that control, rather than the data path of a processor, often turns out to be the critical path, and control is often the last thing to be done and the hardest to estimate timing for. So, designers rely heavily on estimates and on their experience and then do whatever is needed to try to make their clock cycle target. This sometimes means changing the organization so that the CPI of some instructions increases. Using the CPU performance equation, the impact of this trade-off can be calculated.

The other two components of the CPU performance equation are easier to measure. Measuring the instruction count for a program can be done if we have a compiler for the machine together with tools that measure the instruction set behavior. Of course, compilers for existing instruction set architectures are not a problem, and even changes to the architecture can be explored using modern compiler organizations that provide the ability to retarget the compiler easily. For new instruction sets, developing the compiler early is critical to making intelligent decisions in the design of the instruction set.

Once we have a compiled version of a program that we are interested in measuring, there are two major methods we can apply to obtain instruction count information. In most cases, we want to know not only the total instruction count, but also the frequency of different classes of instructions (called the *instruction mix*). The first way to obtain such data is an instruction set simulator that interprets the instructions. The major drawbacks of this approach are speed (since emulating the instruction set is slow) and the possible need to implement substantial infrastructure, since to handle large programs the simulator will need to provide support for operating system functions. One advantage of an instruction set simulator is that it can measure almost any aspect of instruction set behavior accurately and can also potentially simulate systems programs, such as the operating system. Typical instruction set simulators run from 10 to 1000 times slower than the program might, with the performance depending both on how carefully the simulator is written and on the relationship between the architectures of the simulated machine and host machine.

The alternative approach uses execution-based monitoring. In this approach, the binary program is modified to include *instrumentation code*, such as a counter in every basic block. The program is run and the counter values are recorded. It is then simple to determine the instruction distribution by examining the static version of the code and the values of the counters, which tell us how often each instruction is executed. This technique is obviously very fast, since the program is executed, rather than interpreted. Typical instrumentation code increases the execution time by 1.1 to 2.0 times. This technique is even usable when the architectures of the machine being simulated and the machine being used for the simulator differ. In such a case, the program that instruments the code does a simple translation between the instruction sets. This translation need not be very efficient—even a sloppy translation will usually lead to a much faster measurement system than complete simulation of the instruction set.

Measuring the CPI is more difficult, since it depends on the detailed processor organization as well as the instruction stream. For very simple processors, it may be possible to compute a CPI for every instruction from a table and simply multiply these values by the number of instances of each instruction type. However, this simplistic approach will not work with most modern processors. Since these processors were built using techniques such as pipelining and memory hierarchies, instructions do not have simple cycle counts but instead depend on the state of the processor when the instruction is executed. Designers often use average CPI values for instructions, but these average CPIs are computed by measuring the effects of the pipeline and cache structure.

To determine the CPI for an instruction in a modern processor, it is often useful to separate the component arising from the memory system and the component determined by the pipeline, assuming a perfect memory system. This is useful both because the simulation techniques for evaluating these contributions are different and because the memory system contribution is added as an average to all instructions, while the processor contribution is more likely to be instruction specific. Thus, we can compute the CPI for each instruction, *i*, as

$$\text{CPI}_i = \text{Pipeline CPI}_i + \text{Memory system CPI}_i$$

In the next section, we'll see how memory system CPI can be computed, at least for simple memory hierarchies. Chapter 5 discusses more sophisticated memory hierarchies and performance modeling.

The pipeline CPI is typically modeled by simulating the pipeline structure using the instruction stream. For simple pipelines, it may be sufficient to model the performance of each basic block individually, ignoring the cross basic block interactions. In such cases, the performance of each basic block, together with the frequency counts for each basic block, can be used to determine the overall CPI as well as the CPI for each instruction. In Chapter 3, we will examine simple pipeline structures where this approximation is valid. Since the pipeline behavior of each basic block is simulated only once, this is much faster than a full simulation of every instruction execution. Unfortunately, in our exploration of advanced pipelining in Chapter 4, we'll see that full simulations of the program are necessary to estimate the performance of sophisticated pipelines.

Using the CPU Performance Equations: More Examples

The real measure of computer performance is time. Changing the instruction set to lower the instruction count, for example, may lead to an organization with a slower clock cycle time that offsets the improvement in instruction count. When comparing two machines, you must look at all three components to understand relative performance.

EXAMPLE Suppose we are considering two alternatives for our conditional branch instructions, as follows:

CPU A: A condition code is set by a compare instruction and followed by a branch that tests the condition code.

CPU B: A compare is included in the branch.

On both CPUs, the conditional branch instruction takes 2 cycles, and all other instructions take 1 clock cycle. On CPU A, 20% of all instructions executed are conditional branches; since every branch needs a compare, another 20% of the instructions are compares. Because CPU A does not have the compare included in the branch, assume that its clock cycle time is 1.25 times faster than that of CPU B. Which CPU is faster? Suppose CPU A's clock cycle time was only 1.1 times faster?

ANSWER Since we are ignoring all systems issues, we can use the CPU performance formula:

$$\text{CPI}_A = 0.20 \times 2 + 0.80 \times 1 = 1.2$$

since 20% are branches taking 2 clock cycles and the rest of the instructions take 1 cycle each.The performance of CPU A is then

$$\text{CPU time}_A = \text{IC}_A \times 1.2 \times \text{Clock cycle time}_A$$

Clock cycle time$_B$ is 1.25 $\times$ Clock cycle time$_A$, since A has a clock rate that is 1.25 times higher. Compares are not executed in CPU B, so 20%/80% or 25% of the instructions are now branches taking 2 clock cycles, and the remaining 75% of the instructions take 1 cycle. Hence,

$$\text{CPI}_B = 0.25 \times 2 + 0.75 \times 1 = 1.25$$

Because CPU B doesn't execute compares, IC$_B$ = 0.8 $\times$ IC$_A$. Hence, the performance of CPU B is

$$\begin{aligned}\text{CPU time}_B &= \text{IC}_B \times \text{CPI}_B \times \text{Clock cycle time}_B \\ &= 0.8 \times \text{IC}_A \times 1.25 \times (1.25 \times \text{Clock cycle time}_A) \\ &= 1.25 \times \text{IC}_A \times \text{Clock cycle time}_A\end{aligned}$$

Under these assumptions, CPU A, with the shorter clock cycle time, is faster than CPU B, which executes fewer instructions.

If CPU A were only 1.1 times faster, then Clock cycle time$_B$ is $1.10 \times \text{Clock cycle time}_A$, and the performance of CPU B is

$$\begin{aligned}\text{CPU time}_B &= \text{IC}_B \times \text{CPI}_B \times \text{Clock cycle time}_B \\ &= 0.8 \times \text{IC}_A \times 1.25 \times (1.10 \times \text{Clock cycle time}_A) \\ &= 1.10 \times \text{IC}_A \times \text{Clock cycle time}_A\end{aligned}$$

With this improvement CPU B, which executes fewer instructions, is now faster. ■

Locality of Reference

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

To examine locality, 10 application programs in the SPEC92 benchmark suite were measured to determine what percentage of the instructions were responsible for 80% and for 90% of the instructions executed. The data are shown in Figure 1.14.

Locality of reference also applies to data accesses, though not as strongly as to code accesses. Two different types of locality have been observed. *Temporal locality* states that recently accessed items are likely to be accessed in the near future. Figure 1.14 shows one effect of temporal locality. *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 the next section.

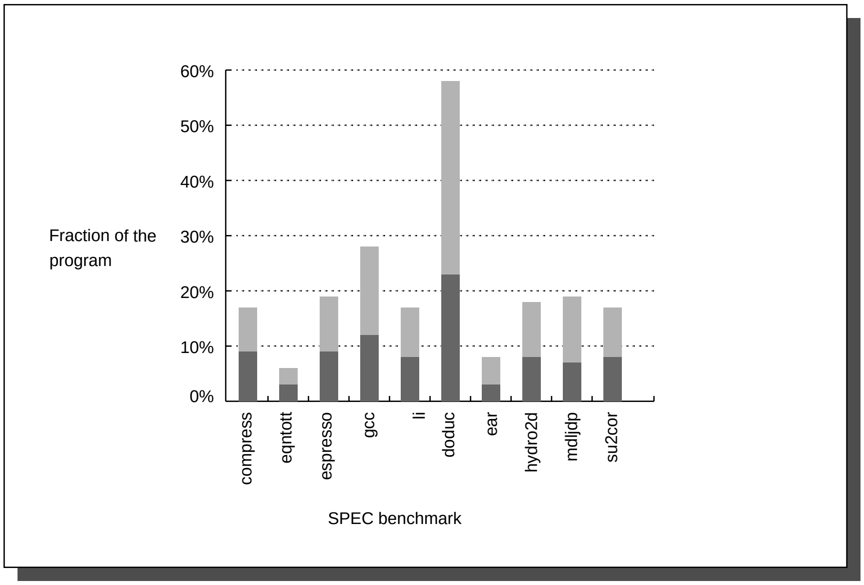

FIGURE 1.14 This plot shows what percentage of the instructions are responsible for 80% and for 90% of the instruction executions. The total bar height indicates the fraction of the instructions that account for 90% of the instruction executions while the dark portion indicates the fraction of the instructions responsible for 80% of the instruction executions. For example, in compress about 9% of the code accounts for 80% of the executed instructions and 16% accounts for 90% of the executed instructions. On average, 90% of the instruction executions comes from 10% of the instructions in the integer programs and 14% of the instructions in the FP programs. The programs are described in more detail in Figure 1.9 on page 22.

1.7 Putting It All Together: The Concept of Memory Hierarchy

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 first chapter, we discuss a key idea in memory systems that will be the sole focus of our attention in Chapter 5.

To begin, let's look at a simple axiom of hardware design: *Smaller is faster.* Smaller pieces of hardware will generally be faster than larger pieces. This simple principle is particularly applicable to memories built from the same technology for two reasons. First, in high-speed machines, signal propagation is a major cause of delay; larger memories have more signal delay and require more levels to decode addresses. Second, in most technologies we can obtain smaller memories that are faster than larger memories. This is primarily because the designer can use more power per memory cell in a smaller design. The fastest memories are generally available in smaller numbers of bits per chip at any point in time, and they cost substantially more per byte.

The important exception to the smaller-is-faster rule arises from differences in power consumption. Designs with higher power consumption will be faster and also usually larger. Such power differences can come from changes in technology, such as the use of ECL versus CMOS, or from a change in the design, such as the use of static memory cells rather than dynamic memory cells. If the power increase is sufficient, it can overcome the disadvantage arising from the size increase. Thus, the smaller-is-faster rule applies only when power differences do not exist or are taken into account.

Increasing memory bandwidth and decreasing the time to access memory are both crucial to system performance, and many of the organizational techniques we discuss will focus on these two metrics. How can we improve these two measures? The answer lies in combining the principles we discussed in this chapter together with the rule that smaller is faster.

The principle of locality of reference says that the data most recently used is very likely to be accessed again in the near future. Making the common case fast suggests that favoring accesses to such data will improve performance. Thus, we should try to keep recently accessed items in the fastest memory. Because smaller memories will be faster, we want to use smaller memories to try to hold the most recently accessed items close to the CPU and successively larger (and slower) memories as we move farther away from the CPU. Furthermore, we can also employ more expensive and higher-powered memory technologies for those memories closer to the CPU, because they are much smaller and the cost and power impact is lessened by the small size of the memories. This type of organization is called a *memory hierarchy.* Figure 1.15 shows a multilevel memory hierarchy, including typical sizes and speeds of access. Two important levels of the memory hierarchy are the cache and virtual memory.

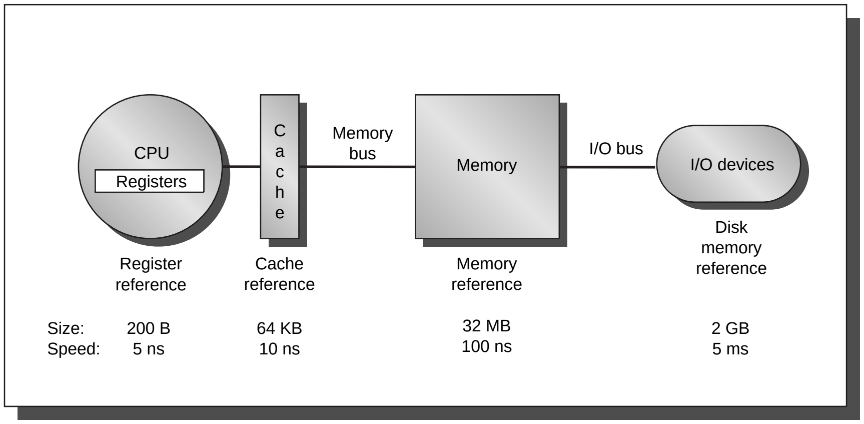

FIGURE 1.15 These are the levels in a typical memory hierarchy. As we move farther away from the CPU, the memory in the level becomes larger and slower. The sizes and access times are typical for a low- to mid-range desktop machine in late 1995. Figure 1.16 shows the wider range of values in use.

A *cache* is a small, fast memory located close to the CPU that holds the most recently accessed code or data. When the CPU finds a requested data item in the cache, it is called a *cache hit*. When the CPU does not find a data item it needs in the cache, a *cache miss* occurs. A fixed-size block of data, called a *block*, containing the requested word is retrieved from the main memory and placed into the cache. Temporal locality tells us that we are likely to need this word again in the near future, so placing it in the cache where it can be accessed quickly is useful. Because of spatial locality, there is high probability that the other data in the block will be needed soon.

The time required for the cache miss depends on both the latency of the memory and its bandwidth, which determines the time to retrieve the entire block. A cache miss, which is handled by hardware, usually causes the CPU to pause, or stall, until the data are available.

Likewise, not all objects referenced by a program need to reside in main memory. If the computer has *virtual memory*, then some objects may reside on disk. The address space is usually broken into fixed-size blocks, called *pages*. At any time, each page resides either in main memory or on disk. When the CPU references an item within a page that is not present in the cache or main memory, a *page fault* occurs, and the entire page is moved from the disk to main memory. Since page faults take so long, they are handled in software and the CPU is not stalled. The CPU usually switches to some other task while the disk access occurs. The cache and main memory have the same relationship as the main memory and disk.

Figure 1.16 shows the range of sizes and access times of each level in the memory hierarchy for machines ranging from low-end desktops to high-end servers. Chapter 5 focuses on memory hierarchy design and contains a detailed example of a real system hierarchy.

Level	**1**	**2**	**3**	**4**
Called	Registers	Cache	Main memory	Disk storage
Typical size	< 1 KB	< 4 MB	< 4 GB	> 1 GB
Implementation technology	Custom memory with multiple ports, CMOS or BiCMOS	On-chip or off-chip CMOS SRAM	CMOS DRAM	Magnetic disk
Access time (in ns)	2–5	3–10	80–400	5,000,000
Bandwidth (in MB/sec)	4000–32,000	800–5000	400–2000	4–32
Managed by	Compiler	Hardware	Operating system	Operating system/user
Backed by	Cache	Main memory	Disk	Tape

FIGURE 1.16 The typical levels in the hierarchy slow down and get larger as we move away from the CPU. Sizes are typical for a large workstation or small server. The implementation technology shows the typical technology used for these functions. The access time is given in nanoseconds for typical values in 1995; these times will decrease over time. Bandwidth is given in megabytes per second, assuming 64- to 256-bit paths between levels in the memory hierarchy. As we move to lower levels of the hierarchy, the access times increase, making it feasible to manage the transfer less responsively.

Performance of Caches: The Basics

Because of locality and the higher speed of smaller memories, a memory hierarchy can substantially improve performance. There are several ways that we can look at the performance of a memory hierarchy and its impact on CPU performance. Let's start with an example that uses Amdahl's Law to compare a system with and without a cache.

EXAMPLE Suppose a cache is 10 times faster than main memory, and suppose that the cache can be used 90% of the time. How much speedup do we gain by using the cache?

ANSWER This is a simple application of Amdahl's Law.

$$\text{Speedup} = \frac{1}{(1 - \text{\% of time cache can be used}) + \dfrac{\text{\% of time cache can be used}}{\text{Speedup using cache}}}$$

$$\text{Speedup} = \frac{1}{(1 - 0.9) + \dfrac{0.9}{10}}$$

$$\text{Speedup} = \frac{1}{0.19} \approx 5.3$$

Hence, we obtain a speedup from the cache of about 5.3 times. ■

In practice, we do not normally use Amdahl's Law for evaluating memory hierarchies. Most machines will include a memory hierarchy, and the key issue is really how to design that hierarchy, which depends on more detailed measurements. An alternative method is to expand our CPU execution time equation to account for the number of cycles during which the CPU is stalled waiting for a memory access, which we call the *memory stall cycles*. The performance is then the product of the clock cycle time and the sum of the CPU cycles and the memory stall cycles:

$$\text{CPU execution time} = (\text{CPU clock cycles} + \text{Memory stall cycles}) \times \text{Clock cycle}$$

This equation assumes that the CPU clock cycles include the time to handle a cache hit, and that the CPU is stalled during a cache miss. In Chapter 5, we will analyze memory hierarchies in more detail, examining both these assumptions.

The number of memory stall cycles depends on both the number of misses and the cost per miss, which is called the *miss penalty*:

$$\begin{aligned}\text{Memory stall cycles} &= \text{Number of misses} \times \text{Miss penalty}\\ &= \text{IC} \times \text{Misses per instruction} \times \text{Miss penalty}\\ &= \text{IC} \times \text{Memory references per instruction} \times \text{Miss rate} \times \text{Miss penalty}\end{aligned}$$

The advantage of the last form is that the components can be easily measured: We already know how to measure IC (instruction count), and measuring the number of memory references per instruction can be done in the same fashion, since each instruction requires an instruction access and we can easily decide if it requires a data access. The component *Miss rate* is simply the fraction of cache accesses that result in a miss (i.e., number of accesses that miss divided by number of accesses). Miss rates are typically measured with cache simulators that take a *trace* of the instruction and data references, simulate the cache behavior to determine which references hit and which miss, and then report the hit and miss totals. The miss rate is one of the most important measures of cache design, but, as we will see in Chapter 5, not the only measure.

EXAMPLE Assume we have a machine where the CPI is 2.0 when all memory accesses hit in the cache.The only data accesses are loads and stores, and these total 40% of the instructions. If the miss penalty is 25 clock cycles and the miss rate is 2%, how much faster would the machine be if all instructions were cache hits?

ANSWER First compute the performance for the machine that always hits:

$$\begin{aligned}\text{CPU execution time} &= (\text{CPU clock cycles} + \text{Memory stall cycles}) \times \text{Clock cycle}\\ &= (\text{IC} \times \text{CPI} + 0) \times \text{Clock cycle}\\ &= \text{IC} \times 2.0 \times \text{Clock cycle}\end{aligned}$$

Now for the machine with the real cache, first we compute memory stall cycles:

$$\begin{aligned}\text{Memory stall cycles} &= \text{IC} \times \text{Memory references per instruction} \times \text{Miss rate} \times \text{Miss penalty}\\ &= \text{IC} \times (1 + 0.4) \times 0.02 \times 25\\ &= \text{IC} \times 0.7\end{aligned}$$

where the middle term (1 + 0.4) represents one instruction access and 0.4 data accesses per instruction. The total performance is thus

$$\begin{aligned}\text{CPU execution time}_{\text{cache}} &= (\text{IC} \times 2.0 + \text{IC} \times 0.7) \times \text{Clock cycle}\\ &= 2.7 \times \text{IC} \times \text{Clock cycle}\end{aligned}$$

The performance ratio is the inverse of the execution times:

$$\frac{\text{CPU execution time}_{\text{cache}}}{\text{CPU execution time}} = \frac{2.7 \times \text{IC} \times \text{Clock cycle}}{2.0 \times \text{IC} \times \text{Clock cycle}}$$
$$= 1.35$$

The machine with no cache misses is 1.35 times faster. ■

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

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.

We can even see such anomalies with optimizing compilers.

EXAMPLE Assume we build an optimizing compiler for the load-store machine for which the measurements in Figure 1.17 have been made. The compiler discards 50% of the arithmetic logic unit (ALU) instructions, although it cannot reduce loads, stores, or branches. Ignoring systems issues and assuming a 2-ns clock cycle time (500-MHz clock rate) and 1.57 unoptimized CPI, what is the MIPS rating for optimized code versus unoptimized code? Does the ranking of MIPS agree with the ranking of execution time?

Instruction type	Frequency	Clock cycle count
ALU ops	43%	1
Loads	21%	2
Stores	12%	2
Branches	24%	2

FIGURE 1.17 Measurements of the load-store machine.

ANSWER We know that $\text{CPI}_{\text{unoptimized}} = 1.57$, so

$$\text{MIPS}_{\text{unoptimized}} = \frac{500\text{MHz}}{1.57 \times 10^6} = 318.5$$

The performance of unoptimized code is

$$\begin{aligned}\text{CPU time}_{\text{unoptimized}} &= \text{IC}_{\text{unoptimized}} \times 1.57 \times (2 \times 10^{-9}) \\ &= 3.14 \times 10^{-9} \times \text{IC}_{\text{unoptimized}}\end{aligned}$$

For optimized code:

$$\text{CPI}_{\text{optimized}} = \frac{(0.43/2) \times 1 + 0.21 \times 2 + 0.12 \times 2 + 0.24 \times 2}{1 - (0.43/2)} = 1.73$$

since half the ALU instructions are discarded (0.43/2) and the instruction count is reduced by the missing ALU instructions. Thus,

$$\text{MIPS}_{\text{optimized}} = \frac{500 \text{ MHz}}{1.73 \times 10^6} = 289.0$$

The performance of optimized code is

$$\begin{aligned}\text{CPU time}_{\text{optimized}} &= (0.785 \times \text{IC}_{\text{unoptimized}}) \times 1.73 \times (2 \times 10^{-9}) \\ &= 2.72 \times 10^{-9} \times \text{IC}_{\text{unoptimized}}\end{aligned}$$

The optimized code is 3.14/2.72 = 1.15 times faster, but its MIPS rating is lower: 289 versus 318! ■

As examples such as this one show, MIPS can fail to give a true picture of performance because it does not track execution time.

Fallacy: MFLOPS is a consistent and useful measure of performance.

Another popular alternative to execution time is *million floating-point operations per second*, abbreviated megaFLOPS or MFLOPS but always pronounced "megaflops." The formula for MFLOPS is simply the definition of the acronym:

$$\text{MFLOPS} = \frac{\text{Number of floating-point operations in a program}}{\text{Execution time in seconds} \times 10^6}$$

Clearly, a MFLOPS rating is dependent on the machine and on the program. Since MFLOPS is intended to measure floating-point performance, it is not applicable outside that range. Compilers, as an extreme example, have a MFLOPS rating near nil no matter how fast the machine, since compilers rarely use floating-point arithmetic.

This term is less innocent than MIPS. Based on operations rather than instructions, MFLOPS is intended to be a fair comparison between different machines. The belief is that the same program running on different computers would execute a different number of instructions but the same number of floating-point operations. Unfortunately, MFLOPS is not dependable because the set of floating-point operations is not consistent across machines. For example, the Cray C90 has no divide instruction, while the Intel Pentium has divide, square root, sine, and cosine. Another perceived problem is that the MFLOPS rating changes not only on the mixture of integer and floating-point operations but also on the mixture of fast and slow floating-point operations. For example, a program with 100% floating-point adds will have a higher rating than a program with 100% floating-point divides. (We discuss a proposed solution to this problem in Exercise 1.15 (b).)

Furthermore, like any other performance measure, the MFLOPS rating for a single program cannot be generalized to establish a single performance metric for a computer. Since MFLOPS is really just a constant divided by execution time for a specific program and specific input, MFLOPS is redundant to execution time, our principal measure of performance. And unlike execution time, it is tempting to characterize a machine with a single MIPS or MFLOPS rating without naming the program, specifying the I/O, or describing the versions of the OS and compilers.

Fallacy: Synthetic benchmarks predict performance for real programs.

The best known examples of such 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 4) and vectorization (Appendix B).
- Compilers can optimize a key piece of the Whetstone loop by noting the relationship between square root and exponential, even though this is very unlikely to occur in real programs. For example, one key loop contains the following FORTRAN code:

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

It could be compiled as if it were

```
X = EXP(ALOG(X)/(2×T1))
```

since

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

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

Fallacy: 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. While the baseline performance measurements restrict this (the rules for baseline tuning appear on pages 57–58), the tuned or optimized performance does not. In fact, benchmark-specific flags are allowed, even if they are illegal and lead to incorrect compilation in general! This has resulted in long lists of options, as Figure 1.18 shows. This incredible list of impenetrable options used in the tuned measurements for an IBM Powerstation 590, which is not significantly different from the option lists used by other vendors, makes it clear why the baseline measurements were needed. The performance difference between the baseline and tuned numbers can be substantial. For the SPECfp92 benchmarks on the Powerstation 590, the overall performance (which by SPEC92 rules is summarized by geometric mean) is 1.2 times higher for the optimized programs. For some benchmarks, however, the difference is considerably larger: For the nasa7 kernels, the optimized performance is 2.1 times higher than the baseline!

Benchmark engineering is sometimes applied to the runtime libraries. For example, SPEC92 added a spreadsheet to the SPEC92 integer benchmarks (called sc). Like any spreadsheet, sc spends a great deal of its time formatting data for the screen, a function that is handled in a UNIX runtime library. Normally such screen I/O is synchronous—each I/O is completed before the next one is done. This increases the runtime substantially. Several companies observed that when the benchmark is run, its output goes to a file, in which case the I/O need not be synchronous. Instead the I/O can be done to a memory buffer that is flushed to disk after the program completes, thus taking the I/O time out of the measure-

SPECfp92 Tuning parameters/Notes/Summary of changes:

```
Software: KAP for IBM FORTRAN Ver. 3.1 Beta, VAST-2 for XL FORTRAN Ver. 4.03 Beta, KAP for IBM C, Ver. 1.3
all: -O3 -qarch=pwrx -BI:/lib/syscalls.exp
013: -qnosave -P -Wp,-ea478,-Iindxx:dcsol,-Sv01.f:v06.f -lblas
015: -P -Wp,-ea478,-fz,-Isi:coeray,-Ssi.f:coeray.f -lblas
039: -Pk -Wp,-r=3,-inline,-ur=8,-ur2=2 00,-ind=2,-in11=2
034: -Pk -Wp,-r=3,-inline,-ur=4
047: -Q-Pk -Wp,-r=3,-o=4,-ag=a
048: -Pk -Wp,-inline,-r=3,-ur=2,-ur=10 0
052: -Q -Q-input-hidden -qhsflt -Dfloat=double -qassert-typeptr -qproclocal -qmaxmem=9999999 +K4 +Kargs=ur2=1
056: -qproclocal -Dfloat=double -qunroll=2 -qhsflt -qmaxmem=999999 +K4 -Kargs=-ar1=2:-ur2=5000
077: -O3 -qstrict -qarch=ppc -qmaxmem=-1 -Pk -Wp,-inline,-r=3,-ur=2,-ur2=9999
078: -qhsflt -P -Wp,-ea278,-fz,-me -qhot
089: -qnosave -qhssngl -Pk -Wp,-inline=trngv,-r=3,-ur=2,-ur2=9999
090: -P -Wp,-ea,-f1 -qhot
093: -DTIMES -P -Wp,-eaj78,-Rvpetst:vpenta:fftst -qfloat=nosqrt -lesslp2
094: -P -Wp,-ea78 -lesslp2
```

FIGURE 1.18 The tuning parameters for the SPECfp92 report on an IBM RS/6000 Powerstation 590. This is the portion of the SPEC report for the tuned performance corresponding to that in Figure 1.10 on page 24. These parameters describe the compiler and preprocessor (two versions of KAP and a version of VAST-2) as well as the options used for the tuned SPEC92 numbers. Each line shows the option used for one of the SPECfp92 benchmarks. The benchmarks are identified by number but appear in the same order as given in Figure 1.9 on page 22. Data from SPEC [1994].

ment loop. One company even went a step farther, realizing that the file is never read, and tossed the I/O completely. If the benchmark was meant to indicate real performance of a spreadsheet-like program, these “optimizations” have defeated such goals. Perhaps even worse than the fact that this creative engineering makes the program perform differently is that it makes it impossible to compare among vendors’ machines, which was the key reason SPEC was created.

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

Fallacy: Peak performance tracks observed performance.

One definition of peak performance is performance a machine is “guaranteed not to exceed.” The gap between peak performance and observed performance is typically a factor of 10 or more in supercomputers. (See Appendix B on vectors for an explanation.) Since the gap is so large, 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 at 236 MFLOPS and on the Cray X-MP at 115 MFLOPS. Although this suggests the S810 is 2.05 times faster than the X-MP, the X-MP runs a program with more typical vector lengths 1.97 times faster than the S810. These data are shown in Figure 1.19.

Measurement	Cray X-MP	Hitachi S810/20	Performance
A(i)=B(i)∗C(i)+D(i)∗E(i) (vector length 1000 done 100,000 times)	2.6 secs	1.3 secs	Hitachi 2.05 times faster
Vectorized FFT (vector lengths 64,32,…,2)	3.9 secs	7.7 secs	Cray 1.97 times faster

FIGURE 1.19 Measurements of peak performance and actual performance for the Hitachi S810/20 and the Cray X-MP. Data from pages 18–20 of Lubeck, Moore, and Mendez [1985]. Also see *Fallacies and Pitfalls* in Appendix B.

While the use of peak performance has been rampant in the supercomputer business, its use in the microprocessor business is just as misleading. For example, in 1994 DEC announced a version of the Alpha microprocessor capable of executing 1.2 billion instructions per second at its 300-MHz clock rate.The only way this processor can achieve this performance is for two integer instructions and two floating-point instructions to be executed each clock cycle. This machine has a peak performance that is almost 50 times the peak performance of the fastest microprocessor reported in the first SPEC benchmark report in 1989 (a MIPS M/2000, which had a 25-MHz clock). The overall SPEC92 number of the DEC Alpha processor, however, is only about 15 times higher on integer and 25 times higher on FP. This rate of performance improvement is still spectacular, even if peak performance is not a good indicator.

The authors hope that peak performance can be quarantined to the supercomputer industry and eventually eradicated from that domain. In any case, approaching supercomputer performance is not an excuse for adopting dubious supercomputer marketing habits.

1.9 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 with a hypothetical instruction set, called DLX, which is designed on the basis of observations of program behavior that we will make in the chapter.

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, with the exception of Chapter 4 on advanced pipelining. In later chapters, the *Crosscutting Issues* sections describe interactions between instruction sets and implementation techniques.

In Chapters 3 and 4 we turn our attention to pipelining, the most common implementation technique used for making faster processors. Pipelining overlaps the execution of instructions and thus can achieve lower CPIs and/or lower clock cycle times. As in Chapter 2, the CPU performance equation will be our guide in the evaluation of alternatives. Chapter 3 starts with a review of the basics of machine organization and control and moves through the basic ideas in pipelining, including the control of more complex floating-point pipelines. The chapter concludes with an examination and analysis of the R4000. At the end of Chapter 3, you will be able to understand the pipeline design of almost every processor built before 1990. Chapter 4 is an extensive examination of advanced pipelining techniques that attempt to get higher performance by exploiting more overlap among instructions than the simple pipelines in use in the 1980s. This chapter begins with an extensive discussion of basic concepts that will prepare you not only for the wide range of ideas examined in Chapter 4, but also to understand and analyze new techniques that will be introduced in the coming years. Chapter 4 uses examples that span about 20 years, drawing from the first modern supercomputers (the CDC 6600 and IBM 360/91) to the latest processors that first reached the market in 1995. Throughout Chapters 3 and 4, we will repeatedly look at techniques that rely either on clever hardware techniques or on sophisticated compiler technology. These alternatives are an exciting aspect of pipeline design, likely to continue through the decade of the 1990s.

In Chapter 5 we turn to the all-important area of memory system design. The *Putting It All Together* section in this chapter serves as a basic introduction. We will examine a wide range of techniques that conspire to make memory look infinitely large while still being as fast as possible. The simple equations we develop in this chapter will serve as a starting point for the quantitative evaluation of the many techniques used for memory system design. As in Chapters 3 and 4, we will see that hardware-software cooperation has become a key to high-performance memory systems, just as it has to high-performance pipelines.

In Chapters 6 and 7, we move away from a CPU-centric view and discuss issues in storage systems and in system interconnect. We apply a similar quantitative approach, but one based on observations of system behavior and using an end-to-end approach to performance analysis. Chapter 6 addresses the important issue of how to efficiently store and retrieve data using primarily lower-cost magnetic storage technologies. As we saw earlier, such technologies offer better cost per bit by a factor of 50–100 over DRAM. Magnetic storage is likely to remain advantageous wherever cost or nonvolatility (it keeps the information after the power is turned off) are important. In Chapter 6, our focus is on examining the performance of magnetic storage systems for typical I/O-intensive workloads, which are the counterpart to the CPU benchmarks we saw in this chapter. We extensively explore the idea of RAID-based systems, which use many small disks, arranged in a redundant fashion to achieve both high performance and high availability. Chapter 7 also discusses the primary interconnection technology used for I/O devices, namely buses. This chapter explores the topic of system interconnect more broadly, including large-scale MPP interconnects and networks used to allow separate computers to communicate. We put special emphasis on the emerging new networking standards developing around ATM.

Our final chapter returns to the issue of achieving higher performance through the use of multiple processors, or multiprocessors. Instead of using parallelism to overlap individual instructions, it 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 as well as those that are developing as this book goes to press.

We conclude this book with a variety of appendices that introduce you to important topics not covered in the eight chapters. Appendix A covers the topic of floating-point arithmetic—a necessary ingredient for any high-performance machine. The incorrect implementation of floating-point divide in the Intel Pentium processor, which led to an estimated impact in excess of $300 million, should serve as a clear reminder about the importance of floating point! Appendix B covers the topic of vector machines. In the scientific market, such machines are a viable alternative to the multiprocessors discussed in Chapter 8. Although vector machines do not dominate supercomputing the way they did in the 1980s, they still include many important concepts in pipelining, parallelism, and memory systems that are useful in different machine organizations. Appendix C surveys the most popular RISC instruction set architectures and contrasts the differences among them, using DLX as a starting point. Appendix D examines the popular

80x86 instruction set—the most heavily used instruction set architecture in existence. Appendix D compares the design of the 80x86 instruction set with that of the RISC machines described in Chapter 2 and in Appendix C. Finally, Appendix E discusses implementation issues in coherence protocols.

1.10 Historical Perspective and References

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

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

A section of historical perspectives closes each chapter in the text. This section provides historical background on some of the key ideas presented in the chapter. The authors may trace the development of an idea through a series of machines or describe significant projects. If you're interested in examining the initial development of an idea or machine or interested in further reading, references are provided at the end of the section.

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 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—far beyond the size of any computer built today. Each of the 20 10-digit registers was 2 feet long. In total, there were 18,000 vacuum tubes.

While the size was three orders of magnitude bigger than the size of 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.* The authors and several early inventors in the computer field believe that this term gives too much credit to von Neumann, who wrote up the ideas, and too little to the engineers, Eckert and Mauchly, who worked on the machines. For this reason, this term will not appear in this book.

In 1946, Maurice Wilkes of Cambridge University visited the Moore School to attend the latter part of a series of lectures on developments in electronic computers. When he returned to Cambridge, Wilkes decided to embark on a project to build a stored-program computer named EDSAC, for Electronic Delay Storage Automatic Calculator. The EDSAC 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 [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. 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 50 years ago, as most of the architectural concepts seen in modern computers are discussed there.

Recently, there has been some controversy about 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 that was never completely operational. Mauchly briefly visited Atanasoff before he built ENIAC. 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. One of the most important was the use of a binary representation for numbers. Atanasoff deserves much credit for his work, and he might fairly be given credit for the world's first special-purpose electronic computer. Another early machine that deserves some credit was a special-purpose machine built by Konrad Zuse in Germany in the late 1930s and early 1940s. This machine was electromechanical and, because of the war, never extensively pursued.

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

The Whirlwind project [Redmond and Smith 1980] began at MIT in 1947 and was aimed at applications in real-time radar signal processing. While 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.

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, where they built the UNIVAC I, designed to be sold as a general-purpose computer. First delivered 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 Boston, Massachusetts.

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, shipped in 1952 and eventually sold 19 units. 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.

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

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

Development of Quantitative Performance Measures: Successes and Failures

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

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

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

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

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

The popularity of the VAX-11/780 made it a popular reference machine for relative MIPS, especially since relative MIPS for a 1-MIPS computer is easy to calculate: If a machine was five times faster than the VAX-11/780, for that benchmark its rating would be 5 relative MIPS. The 1-MIPS rating was unquestioned for four years, until Joel Emer of DEC measured the VAX-11/780 under a time-sharing load. He found that the VAX-11/780 native MIPS rating was 0.5. Subsequent VAXes that run 3 native MIPS for some benchmarks were therefore called 6-MIPS machines because they run six times faster than the VAX-11/780. By the early 1980s, the term MIPS was almost universally used to mean relative MIPS.

The 1970s and 1980s marked the growth of the supercomputer industry, which was defined by high performance on floating-point-intensive programs. Average instruction time and MIPS were clearly inappropriate metrics for this industry, hence the invention of MFLOPS. 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. SPEC89 was replaced by SPEC92. This release enlarged the set of programs, made the inputs to some benchmarks bigger, and specified new run rules. To reduce the large number of benchmark-specific compiler flags and the use of targeted optimizations, in 1994 SPEC introduced rules for compilers and compilation switches to be used in determining the SPEC92 baseline performance:

1. The optimization options are safe: it is expected that they could generally be used on any program.
2. The same compiler and flags are used for all the benchmarks.
3. No assertion flags, which would tell the compiler some fact it could not derive, are allowed.
4. Flags that allow inlining of library routines normally considered part of the language are allowed, though other such inlining hints are disallowed by rule 5.
5. No program names or subroutine names are allowed in flags.
6. Feedback-based optimization is not allowed.
7. Flags that change the default size of a data item (for example, single precision to double precision) are not allowed.

Specifically permitted are flags that direct the compiler to compile for a particular implementation and flags that allow the compiler to relax certain numerical accuracy requirements (such as left-to-right evaluation). The intention is that the baseline results are what a casual user could achieve without extensive effort.

SPEC also has produced system-oriented benchmarks that can be used to benchmark a system including I/O and OS functions, as well as a throughput-oriented measure (SPECrate), suitable for servers. What has become clear is that maintaining the relevance of these benchmarks in an area of rapid performance improvement will be a continuing investment.

Implementation-Independent Performance Analysis

As the distinction between architecture and implementation pervaded the computing community in the 1970s, the question arose whether the performance of an architecture itself could be evaluated, as opposed to an implementation of the architecture. Many of the leading people in the field pursued this notion. One of the ambitious studies of this question performed at Carnegie Mellon University is summarized in Fuller and Burr [1977]. Three quantitative measures were invented to scrutinize architectures:

- S—Number of bytes for program code
- M—Number of bytes transferred between memory and the CPU during program execution for code and data (S measures size of code at compile time, while M is memory traffic during program execution.)
- R—Number of bytes transferred between registers in a canonical model of a CPU

Once these measures were taken, a weighting factor was applied to them to determine which architecture was “best.” The VAX architecture was designed in the height of popularity of the Carnegie Mellon study, and by those measures it does very well. Architectures created since 1985, however, have poorer measures than the VAX using these metrics, yet their implementations do well against the VAX implementations. For example, Figure 1.20 compares S, M, and CPU time for the VAXstation 3100, which uses the VAX instruction set, and the DECstation 3100, which doesn’t. The DECstation 3100 is about three to five times faster, even though its S measure is 35% to 70% worse and its M measure is 5% to 15% worse. The attempt to evaluate architecture independently of implementation was a valiant, if not successful, effort.

Program	S (code size in bytes)		M (megabytes code + data transferred)		CPU time (in secs)	
	VAX 3100	DEC 3100	VAX 3100	DEC 3100	VAX 3100	DEC 3100
Gnu C Compiler	409,600	688,128	18	21	291	90
Common TeX	158,720	217,088	67	78	449	95
spice	223,232	372,736	99	106	352	94

FIGURE 1.20 Code size and CPU time of the VAXstation 3100 and DECstation 3100 for Gnu C Compiler, TeX, and spice. Both machines were announced the same day by the same company, and they run the same operating system and similar technology. The difference is in the instruction sets, compilers, clock cycle time, and organization.

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

EXERCISES

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

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

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

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

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

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

1.2 [15/10] <1.6> Assume—as in the Amdahl’s Law Example on page 30—that we make an enhancement to a computer that improves some mode of execution by a factor of 10. Enhanced mode is used 50% of the time, measured as a percentage of the execution time *when* *the enhanced mode is in use*. Recall that Amdahl's Law depends on the fraction of the original, *unenhanced* execution time that could make use of enhanced mode. Thus, we cannot directly use this 50% measurement to compute speedup with Amdahl's Law.

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

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

1.3 [15] <1.6> Show that the problem statements in the Examples on page 31 and page 33 are the same.

1.4 [15] <1.6> Suppose we are considering a change to an instruction set. The base machine initially has only loads and stores to memory, and all operations work on the registers. Such machines are called *load-store* machines (see Chapter 2). Measurements of the load-store machine showing the *instruction mix* and clock cycle counts per instruction are given in Figure 1.17 on page 45.

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

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

1.5 [15] <1.7> Assume that we have a machine that with a perfect cache behaves as given in Figure 1.17.

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

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

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

Which is faster? Justify your decision quantitatively.

1.7 [15/15/8/12] <1.6,1.8> The Whetstone benchmark contains 195,578 basic floating-point operations in a single iteration, divided as shown in Figure 1.21.

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

Whetstone was run on a Sun 3/75 using the F77 compiler with optimization turned on. The Sun 3/75 is based on a Motorola 68020 running at 16.67 MHz, and it includes a floating-point coprocessor. The Sun compiler allows the floating point to be calculated with the coprocessor or using software routines, depending on compiler flags. A single iteration of Whetstone took 1.08 seconds using the coprocessor and 13.6 seconds using software. Assume that the CPI using the coprocessor was measured to be 10, while the CPI using software was measured to be 6.

a. [15] <1.6,1.8> What is the MIPS rating for both runs?

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

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

d. [12] <1.8> What is the MFLOPS rating for the Sun 3/75 with the floating-point coprocessor running Whetstone? (Assume all the floating-point operations in Figure 1.21 count as one operation.)

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

This exercise requires the information provided in Figure 1.22.

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

FIGURE 1.22 Characteristics of microprocessors. The technology entry is the process type, line width, and number of interconnect levels.

a. [15] <1.4> For each of the microprocessors in Figure 1.22, compute the number of good chips you would get per 20-cm wafer using the model on page 12. Assume a defect density of one defect per cm^2, a wafer yield of 95%, and assume $\alpha = 3$.

b. [10] <1.4> For each microprocessor in Figure 1.22, 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 1.22.

c. [15] <1.3> Both package cost and test cost are proportional to pin count. Using the additional assumption shown in Figure 1.23, 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 1.22 (total cost including packaging), assuming defect densities are 0.6 per cm^2 and assuming that defect densities are 1.2 per cm^2.

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

FIGURE 1.23 Package and test characteristics.

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

1.9 [12] <1.5> One reason people may incorrectly average rates with an arithmetic mean is that it always gives an answer greater than or equal to the geometric mean. Show that for any two positive integers, a and b, the arithmetic mean is always greater than or equal to the geometric mean. When are the two equal?

1.10 [12] <1.5> For reasons similar to those in Exercise 1.9, some people use arithmetic instead of the harmonic mean. Show that for any two positive rates, r and s, the arithmetic mean is always greater than or equal to the harmonic mean. When are the two equal?

1.11 [15/15] <1.5> Some of the SPECfp92 performance results from the SPEC92 Newsletter of June 1994 [SPEC 94] are shown in Figure 1.24. The SPECratio is simply the runtime for a benchmark divided into the VAX 11/780 time for that benchmark. The SPECfp92 number is computed as the geometric mean of the SPECratios. Let's see how a weighted arithmetic mean compares.

a. [15] <1.5> Calculate the weights for a workload so that running times on the VAX-11/780 will be equal for each of the 14 benchmarks (given in Figure 1.24).

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 1.24.

1.12 [15/15/15] <1.6,1.8> Three enhancements with the following speedups are proposed for a new architecture:

$\text{Speedup}_1 = 30$

$\text{Speedup}_2 = 20$

$\text{Speedup}_3 = 10$

Only one enhancement is usable at a time.

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

FIGURE 1.24 SPEC92 performance for SPECfp92. The DEC 3000 uses a 200-MHz Alpha microprocessor (21064) and a 2-MB off-chip cache. The IBM Powerstation 590 uses a 66.67-MHz Power-2. The Intel Xpress uses a 100-MHz Pentium with a 512-KB off-chip secondary cache. Data from SPEC [1994].

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

b. [15] <1.6,1.8> Assume the distribution of enhancement usage is 30%, 30%, and 20% for enhancements 1, 2, and 3, respectively. Assuming all three enhancements are in use, for what fraction of the reduced execution time is no enhancement in use?

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

1.13 [15/10/10/12/10] <1.6,1.8> Your company has a benchmark that is considered representative of your typical applications. One of the older-model workstations does not have a floating-point unit and must emulate each floating-point instruction by a sequence of integer instructions. This older-model workstation is rated at 120 MIPS on this benchmark. A third-party vendor offers an attached processor that is intended to give a “mid-life kicker” to your workstation. That attached processor executes each floating-point instruction on a dedicated processor (i.e., no emulation is necessary). The workstation/attached processor rates 80 MIPS on the same benchmark. The following symbols are used to answer parts (a)–(e) of this exercise.

I—Number of integer instructions executed on the benchmark.

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

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

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

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

a. [15] <1.6,1.8> Write an equation for the MIPS rating of each configuration using the symbols above. Document your equation.

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

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

d. [12] <1.6,1.8> What is the MFLOPS rating of the system with the attached processor board?

e. [10] <1.6,1.8> Your colleague wants to purchase the attached processor board even though the MIPS rating for the configuration using the board is less than that of the workstation alone. Is your colleague's evaluation correct? Defend your answer.

1.14 [15/15/10] <1.5,1.8> Assume the two programs in Figure 1.11 on page 24 each execute 100 million floating-point operations during execution.

a. [15] <1.5,1.8> Calculate the MFLOPS rating of each program.

b. [15] <1.5,1.8> Calculate the arithmetic, geometric, and harmonic means of MFLOPS for each machine.

c. [10] <1.5,1.8> Which of the three means matches the relative performance of total execution time?

1.15 [10/12] <1.8,1.6> One problem cited with MFLOPS as a measure is that not all FLOPS are created equal. To overcome this problem, normalized or weighted MFLOPS measures were developed. Figure 1.25 shows how the authors of the "Livermore Loops" benchmark calculate the number of normalized floating-point operations per program according to the operations actually found in the source code. Thus, the *native MFLOPS* rating is not the same as the *normalized MFLOPS* rating reported in the supercomputer literature, which has come as a surprise to a few computer designers.

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

FIGURE 1.25 Real versus normalized floating-point operations. The number of normalized floating-point operations per real operation in a program used by the authors of the Livermore FORTRAN Kernels, or "Livermore Loops," to calculate MFLOPS. A kernel with one Add, one Divide, and one Sin would be credited with 13 normalized floating-point operations. Native MFLOPS won't give the results reported for other machines on that benchmark.

Let’s examine the effects of this weighted MFLOPS measure. The spice program runs on the DECstation 3100 in 94 seconds. The number of floating-point operations executed in that program are listed in Figure 1.26.

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

FIGURE 1.26 Floating-point operations in spice.

a. [10] <1.8,1.6> What is the native MFLOPS for spice on a DECstation 3100?

b. [12] <1.8,1.6> Using the conversions in Figure 1.25, what is the normalized MFLOPS?

1.16 [30] <1.5,1.8> Devise a program in C that gets the peak MIPS rating for a computer. Run it on two machines to calculate the peak MIPS. Now run the SPEC92 gcc on both machines. How well do peak MIPS predict performance of gcc?

1.17 [30] <1.5,1.8> Devise a program in C or FORTRAN that gets the peak MFLOPS rating for a computer. Run it on two machines to calculate the peak MFLOPS. Now run the SPEC92 benchmark spice on both machines. How well do peak MFLOPS predict performance of spice?

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