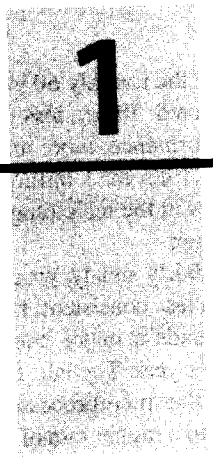

Fundamentals of Computer Design

And now for something completely different.

Monty Python's Flying Circus

1.1 Introduction

Computer technology has made incredible progress in the roughly 60 years since the first general-purpose electronic computer was created. Today, less than $500 will purchase a personal computer that has more performance, more main memory, and more disk storage than a computer bought in 1985 for 1 million dollars. This rapid improvement has come both from advances in the technology used to build computers and from innovation in computer design.

Although technological improvements have been fairly steady, progress arising from better computer architectures has been much less consistent. During the first 25 years of electronic computers, both forces made a major contribution, delivering performance improvement of about 25% per year. The late 1970s saw the emergence of the microprocessor. The ability of the microprocessor to ride the improvements in integrated circuit technology led to a higher rate of improvement—roughly 35% growth per year in performance.

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

These changes made it possible to develop successfully a new set of architectures with simpler instructions, called RISC (Reduced Instruction Set Computer) architectures, in the early 1980s. The RISC-based machines focused the attention of designers on two critical performance techniques, the exploitation of *instruction-level parallelism* (initially through pipelining and later through multiple instruction issue) and the use of caches (initially in simple forms and later using more sophisticated organizations and optimizations).

The RISC-based computers raised the performance bar, forcing prior architectures to keep up or disappear. The Digital Equipment Vax could not, and so it was replaced by a RISC architecture. Intel rose to the challenge, primarily by translating x86 (or IA-32) instructions into RISC-like instructions internally, allowing it to adopt many of the innovations first pioneered in the RISC designs. As transistor counts soared in the late 1990s, the hardware overhead of translating the more complex x86 architecture became negligible.

Figure 1.1 shows that the combination of architectural and organizational enhancements led to 16 years of sustained growth in performance at an annual rate of over 50%—a rate that is unprecedented in the computer industry.

The effect of this dramatic growth rate in the 20th century has been twofold. First, it has significantly enhanced the capability available to computer users. For many applications, the highest-performance microprocessors of today outperform the supercomputer of less than 10 years ago.

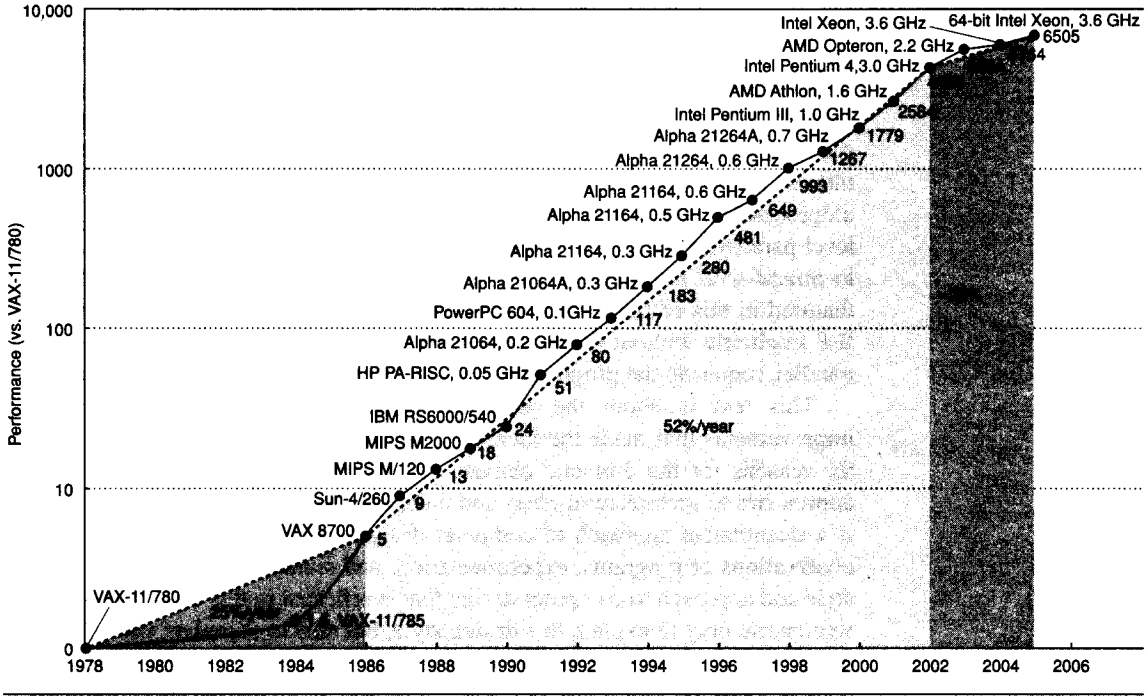

Figure 1.1 Growth in processor performance since the mid-1980s. This chart plots performance relative to the VAX 11/780 as measured by the SPECint benchmarks (see Section 1.8). Prior to the mid-1980s, processor performance growth was largely technology driven and averaged about 25% per year. The increase in growth to about 52% since then is attributable to more advanced architectural and organizational ideas. By 2002, this growth led to a difference in performance of about a factor of seven. Performance for floating-point-oriented calculations has increased even faster. Since 2002, the limits of power, available instruction-level parallelism, and long memory latency have slowed uniprocessor performance recently, to about 20% per year. Since SPEC has changed over the years, performance of newer machines is estimated by a scaling factor that relates the performance for two different versions of SPEC (e.g., SPEC92, SPEC95, and SPEC2000).

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

These innovations led to a renaissance in computer design, which emphasized both architectural innovation and efficient use of technology improvements. This rate of growth has compounded so that by 2002, high-performance microprocessors are about seven times faster than what would have been obtained by relying solely on technology, including improved circuit design.

However, Figure 1.1 also shows that this 16-year renaissance is over. Since 2002, processor performance improvement has dropped to about 20% per year due to the triple hurdles of maximum power dissipation of air-cooled chips, little instruction-level parallelism left to exploit efficiently, and almost unchanged memory latency. Indeed, in 2004 Intel canceled its high-performance uniprocessor projects and joined IBM and Sun in declaring that the road to higher performance would be via multiple processors per chip rather than via faster uniprocessors. This signals a historic switch from relying solely on instruction-level parallelism (ILP), the primary focus of the first three editions of this book, to *thread-level parallelism* (TLP) and *data-level parallelism* (DLP), which are featured in this edition. Whereas the compiler and hardware conspire to exploit ILP implicitly without the programmer's attention, TLP and DLP are explicitly parallel, requiring the programmer to write parallel code to gain performance.

This text is about the architectural ideas and accompanying compiler improvements that made the incredible growth rate possible in the last century, the reasons for the dramatic change, and the challenges and initial promising approaches to architectural ideas and compilers for the 21st century. At the core is 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. This book was written not only to explain this design style, but also to stimulate you to contribute to this progress. We believe the approach will work for explicitly parallel computers of the future just as it worked for the implicitly parallel computers of the past.

1.2 Classes of Computers

In the 1960s, the dominant form of computing was on large mainframes—computers costing millions of dollars and stored in computer rooms with multiple operators overseeing their support. Typical applications included business data processing and large-scale scientific computing. The 1970s saw the birth of the minicomputer, a smaller-sized computer initially focused on applications in scientific laboratories, but rapidly branching out with the popularity of time-sharing—multiple users sharing a computer interactively through independent terminals. That decade also saw the emergence of supercomputers, which were high-performance computers for scientific computing. Although few in number, they were important historically because they pioneered innovations that later trickled down to less expensive computer classes. The 1980s saw the rise of the desktop computer based on microprocessors, in the form of both personal computers and workstations. The individually owned desktop computer replaced time-sharing and led to the rise of servers—computers that provided larger-scale services such as reliable, long-term file storage and access, larger memory, and more computing power. The 1990s saw the emergence of the Internet and the World Wide Web, the first successful handheld computing devices (personal digi-

Feature	Desktop	Server	Embedded
Price of system	$500–$5000	$5000–$5,000,000	$10–$100,000 (including network routers at the high end)
Price of microprocessor module	$50–$500 (per processor)	$200–$10,000 (per processor)	$0.01–$100 (per processor)
Critical system design issues	Price-performance, graphics performance	Throughput, availability, scalability	Price, power consumption, application-specific performance

Figure 1.2 A summary of the three mainstream computing classes and their system characteristics. Note the wide range in system price for servers and embedded systems. For servers, this range arises from the need for very large-scale multiprocessor systems for high-end transaction processing and Web server applications. The total number of embedded processors sold in 2005 is estimated to exceed 3 billion if you include 8-bit and 16-bit microprocessors. Perhaps 200 million desktop computers and 10 million servers were sold in 2005.

tal assistants or PDAs), and the emergence of high-performance digital consumer electronics, from video games to set-top boxes. The extraordinary popularity of cell phones has been obvious since 2000, with rapid improvements in functions and sales that far exceed those of the PC. These more recent applications use *embedded computers*, where computers are lodged in other devices and their presence is not immediately obvious.

These changes have set the stage for a dramatic change in how we view computing, computing applications, and the computer markets in this new century. Not since the creation of the personal computer more than 20 years ago have we seen such dramatic changes in the way computers appear and in how they are used. These changes in computer use have led to three different computing markets, each characterized by different applications, requirements, and computing technologies. Figure 1.2 summarizes these mainstream classes of computing environments and their important characteristics.

Desktop Computing

The first, and still the largest market in dollar terms, is desktop computing. Desktop computing spans from low-end systems that sell for under $500 to high-end, heavily configured workstations that may sell for $5000. Throughout this range in price and capability, the desktop market tends to be driven to optimize *price-performance*. This combination of performance (measured primarily in terms of compute performance and graphics performance) and price of a system is what matters most to customers in this market, and hence to computer designers. As a result, the newest, highest-performance microprocessors and cost-reduced microprocessors often appear first in desktop systems (see Section 1.6 for a discussion of the issues affecting the cost of computers).

Desktop computing also tends to be reasonably well characterized in terms of applications and benchmarking, though the increasing use of Web-centric, interactive applications poses new challenges in performance evaluation.

Servers

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

For servers, different characteristics are important. First, dependability is critical. (We discuss dependability in Section 1.7.) Consider the servers running Google, taking orders for Cisco, or running auctions on eBay. Failure of such server systems is far more catastrophic than failure of a single desktop, since these servers must operate seven days a week, 24 hours a day. Figure 1.3 estimates revenue costs of downtime as of 2000. To bring costs up-to-date, Amazon.com had $2.98 billion in sales in the fall quarter of 2005. As there were about 2200 hours in that quarter, the average revenue per hour was $1.35 million. During a peak hour for Christmas shopping, the potential loss would be many times higher.

Hence, the estimated costs of an unavailable system are high, yet Figure 1.3 and the Amazon numbers are purely lost revenue and do not account for lost employee productivity or the cost of unhappy customers.

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

Lastly, servers are designed for efficient throughput. That is, the overall performance of the server—in terms of transactions per minute or Web pages served

		Annual losses (millions of $) with downtime of		
Application	**Cost of downtime per hour (thousands of $)**	**1% (87.6 hrs/yr)**	**0.5% (43.8 hrs/yr)**	**0.1% (8.8 hrs/yr)**
Brokerage operations	$6450	$565	$283	$56.5
Credit card authorization	$2600	$228	$114	$22.8
Package shipping services	$150	$13	$6.6	$1.3
Home shopping channel	$113	$9.9	$4.9	$1.0
Catalog sales center	$90	$7.9	$3.9	$0.8
Airline reservation center	$89	$7.9	$3.9	$0.8
Cellular service activation	$41	$3.6	$1.8	$0.4
Online network fees	$25	$2.2	$1.1	$0.2
ATM service fees	$14	$1.2	$0.6	$0.1

Figure 1.3 The cost of an unavailable system is shown by analyzing the cost of downtime (in terms of immediately lost revenue), assuming three different levels of availability, and that downtime is distributed uniformly. These data are from Kembel [2000] and were collected and analyzed by Contingency Planning Research.

per second—is what is crucial. Responsiveness to an individual request remains important, but overall efficiency and cost-effectiveness, as determined by how many requests can be handled in a unit time, are the key metrics for most servers. We return to the issue of assessing performance for different types of computing environments in Section 1.8.

A related category is *supercomputers*. They are the most expensive computers, costing tens of millions of dollars, and they emphasize floating-point performance. Clusters of desktop computers, which are discussed in Appendix H, have largely overtaken this class of computer. As clusters grow in popularity, the number of conventional supercomputers is shrinking, as are the number of companies who make them.

Embedded Computers

Embedded computers are the fastest growing portion of the computer market. These devices range from everyday machines—most microwaves, most washing machines, most printers, most networking switches, and all cars contain simple embedded microprocessors—to handheld digital devices, such as cell phones and smart cards, to video games and digital set-top boxes.

Embedded computers have the widest spread of processing power and cost. They include 8-bit and 16-bit processors that may cost less than a dime, 32-bit microprocessors that execute 100 million instructions per second and cost under $5, and high-end processors for the newest video games or network switches that cost $100 and can execute a billion instructions per second. Although the range of computing power in the embedded computing market is very large, price is a key factor in the design of computers for this space. Performance requirements do exist, of course, but the primary goal is often meeting the performance need at a minimum price, rather than achieving higher performance at a higher price.

Often, the performance requirement in an embedded application is real-time execution. A *real-time performance* requirement is when a segment of the application has an absolute maximum execution time. For example, in a digital set-top box, the time to process each video frame is limited, since the processor must accept and process the next frame shortly. In some applications, a more nuanced requirement exists: the average time for a particular task is constrained as well as the number of instances when some maximum time is exceeded. Such approaches—sometimes called *soft real-time*—arise when it is possible to occasionally miss the time constraint on an event, as long as not too many are missed. Real-time performance tends to be highly application dependent.

Two other key characteristics exist in many embedded applications: the need to minimize memory and the need to minimize power. In many embedded applications, the memory can be a substantial portion of the system cost, and it is important to optimize memory size in such cases. Sometimes the application is expected to fit totally in the memory on the processor chip; other times the

application needs to fit totally in a small off-chip memory. In any event, the importance of memory size translates to an emphasis on code size, since data size is dictated by the application.

Larger memories also mean more power, and optimizing power is often critical in embedded applications. Although the emphasis on low power is frequently driven by the use of batteries, the need to use less expensive packaging—plastic versus ceramic—and the absence of a fan for cooling also limit total power consumption. We examine the issue of power in more detail in Section 1.5.

Most of this book applies to the design, use, and performance of embedded processors, whether they are off-the-shelf microprocessors or microprocessor cores, which will be assembled with other special-purpose hardware.

Indeed, the third edition of this book included examples from embedded computing to illustrate the ideas in every chapter. Alas, most readers found these examples unsatisfactory, as the data that drives the quantitative design and evaluation of desktop and server computers has not yet been extended well to embedded computing (see the challenges with EEMBC, for example, in Section 1.8). Hence, we are left for now with qualitative descriptions, which do not fit well with the rest of the book. As a result, in this edition we consolidated the embedded material into a single appendix. We believe this new appendix (Appendix D) improves the flow of ideas in the text while still allowing readers to see how the differing requirements affect embedded computing.

1.3 Defining Computer Architecture

The task the computer designer faces is a complex one: Determine what attributes are important for a new computer, then design a computer to maximize performance while staying within cost, power, and availability 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.

We believe this view is incorrect. 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 likely more challenging than those encountered in instruction set design. We'll quickly review instruction set architecture before describing the larger challenges for the computer architect.

Instruction Set Architecture

We use the term *instruction set architecture* (ISA) to refer to the actual programmer-visible instruction set in this book. The ISA serves as the boundary between the

software and hardware. This quick review of ISA will use examples from MIPS and 80x86 to illustrate the seven dimensions of an ISA. Appendices B and J give more details on MIPS and the 80x86 ISAs.

1. *Class of ISA*—Nearly all ISAs today are classified as general-purpose register architectures, where the operands are either registers or memory locations. The 80x86 has 16 general-purpose registers and 16 that can hold floating-point data, while MIPS has 32 general-purpose and 32 floating-point registers (see Figure 1.4). The two popular versions of this class are *register-memory* ISAs such as the 80x86, which can access memory as part of many instructions, and *load-store* ISAs such as MIPS, which can access memory only with load or store instructions. All recent ISAs are load-store.
2. *Memory addressing*—Virtually all desktop and server computers, including the 80x86 and MIPS, use byte addressing to access memory operands. Some architectures, like MIPS, require that objects must be *aligned*. An access to an object of size s bytes at byte address A is aligned if $A \bmod s = 0$. (See Figure B.5 on page B-9.) The 80x86 does not require alignment, but accesses are generally faster if operands are aligned.
3. *Addressing modes*—In addition to specifying registers and constant operands, addressing modes specify the address of a memory object. MIPS addressing modes are Register, Immediate (for constants), and Displacement, where a constant offset is added to a register to form the memory address. The 80x86 supports those three plus three variations of displacement: no register (absolute), two registers (based indexed with displacement), two registers where

Name	Number	Use	Preserved across a call?
$zero	0	The constant value 0	N.A.
$at	1	Assembler temporary	No
$v0–$v1	2–3	Values for function results and expression evaluation	No
$a0–$a3	4–7	Arguments	No
$t0–$t7	8–15	Temporaries	No
$s0–$s7	16–23	Saved temporaries	Yes
$t8–$t9	24–25	Temporaries	No
$k0–$k1	26–27	Reserved for OS kernel	No
$gp	28	Global pointer	Yes
$sp	29	Stack pointer	Yes
$fp	30	Frame pointer	Yes
$ra	31	Return address	Yes

Figure 1.4 MIPS registers and usage conventions. In addition to the 32 general-purpose registers (R0–R31), MIPS has 32 floating-point registers (F0–F31) that can hold either a 32-bit single-precision number or a 64-bit double-precision number.

one register is multiplied by the size of the operand in bytes (based with scaled index and displacement). It has more like the last three, minus the displacement field: register indirect, indexed, and based with scaled index.

4. *Types and sizes of operands*—Like most ISAs, MIPS and 80x86 support operand sizes of 8-bit (ASCII character), 16-bit (Unicode character or half word), 32-bit (integer or word), 64-bit (double word or long integer), and IEEE 754 floating point in 32-bit (single precision) and 64-bit (double precision). The 80x86 also supports 80-bit floating point (extended double precision).
5. *Operations*—The general categories of operations are data transfer, arithmetic logical, control (discussed next), and floating point. MIPS is a simple and easy-to-pipeline instruction set architecture, and it is representative of the RISC architectures being used in 2006. Figure 1.5 summarizes the MIPS ISA. The 80x86 has a much richer and larger set of operations (see Appendix J).
6. *Control flow instructions*—Virtually all ISAs, including 80x86 and MIPS, support conditional branches, unconditional jumps, procedure calls, and returns. Both use PC-relative addressing, where the branch address is specified by an address field that is added to the PC. There are some small differences. MIPS conditional branches (`BE`, `BNE`, etc.) test the contents of registers, while the 80x86 branches (`JE`, `JNE`, etc.) test condition code bits set as side effects of arithmetic/logic operations. MIPS procedure call (`JAL`) places the return address in a register, while the 80x86 call (`CALLF`) places the return address on a stack in memory.
7. *Encoding an ISA*—There are two basic choices on encoding: *fixed length* and *variable length*. All MIPS instructions are 32 bits long, which simplifies instruction decoding. Figure 1.6 shows the MIPS instruction formats. The 80x86 encoding is variable length, ranging from 1 to 18 bytes. Variable-length instructions can take less space than fixed-length instructions, so a program compiled for the 80x86 is usually smaller than the same program compiled for MIPS. Note that choices mentioned above will affect how the instructions are encoded into a binary representation. For example, the number of registers and the number of addressing modes both have a significant impact on the size of instructions, as the register field and addressing mode field can appear many times in a single instruction.

The other challenges facing the computer architect beyond ISA design are particularly acute at the present, when the differences among instruction sets are small and when there are distinct application areas. Therefore, starting with this edition, the bulk of instruction set material beyond this quick review is found in the appendices (see Appendices B and J).

We use a subset of MIPS64 as the example ISA in this book.

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

Figure 1.5 Subset of the instructions in MIPS64. SP = single precision; DP = double precision. Appendix B gives much more detail on MIPS64. For data, the most significant bit number is 0; least is 63.

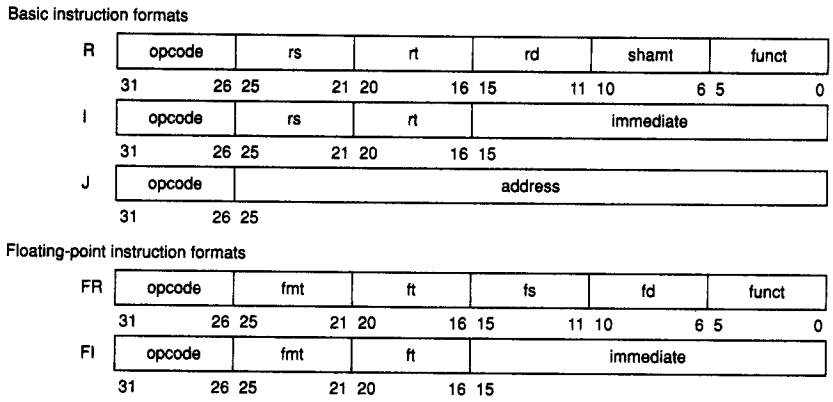

Figure 1.6 MIPS64 instruction set architecture formats. All instructions are 32 bits long. The R format is for integer register-to-register operations, such as DADDU, DSUBU, and so on. The I format is for data transfers, branches, and immediate instructions, such as LD, SD, BEQZ, and DADDIs. The J format is for jumps, the FR format for floating point operations, and the FI format for floating point branches.

The Rest of Computer Architecture: Designing the Organization and Hardware to Meet Goals and Functional Requirements

The implementation of a computer 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 memory interconnect, and the design of the internal processor or CPU (central processing unit—where arithmetic, logic, branching, and data transfer are implemented). For example, two processors with the same instruction set architectures but very different organizations are the AMD Opteron 64 and the Intel Pentium 4. Both processors implement the x86 instruction set, but they have very different pipeline and cache organizations.

Hardware refers to the specifics of a computer, including the detailed logic design and the packaging technology of the computer. Often a line of computers contains computers with identical instruction set architectures and nearly identical organizations, but they differ in the detailed hardware implementation. For example, the Pentium 4 and the Mobile Pentium 4 are nearly identical, but offer different clock rates and different memory systems, making the Mobile Pentium 4 more effective for low-end computers.

In this book, the word *architecture* covers all three aspects of computer design—instruction set architecture, organization, and hardware.

Computer architects must design a computer to meet functional requirements as well as price, power, performance, and availability goals. Figure 1.7 summarizes requirements to consider in designing a new computer. Often, architects

Functional requirements	Typical features required or supported
Application area	*Target of computer*
General-purpose desktop	Balanced performance for a range of tasks, including interactive performance for graphics, video, and audio (Ch. 2, 3, 5, App. B)
Scientific desktops and servers	High-performance floating point and graphics (App. I)
Commercial servers	Support for databases and transaction processing; enhancements for reliability and availability; support for scalability (Ch. 4, App. B, E)
Embedded computing	Often requires special support for graphics or video (or other application-specific extension); power limitations and power control may be required (Ch. 2, 3, 5, App. B)
Level of software compatibility	*Determines amount of existing software for computer*
At programming language	Most flexible for designer; need new compiler (Ch. 4, App. B)
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, App. E)*
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; virtual machines (Ch. 5)
Standards	*Certain standards may be required by marketplace*
Floating point	Format and arithmetic: IEEE 754 standard (App. I), special arithmetic for graphics or signal processing
I/O interfaces	For I/O devices: Serial ATA, Serial Attach SCSI, PCI Express (Ch. 6, App. E)
Operating systems	UNIX, Windows, Linux, CISCO IOS
Networks	Support required for different networks: Ethernet, Infiniband (App. E)
Programming languages	Languages (ANSI C, C++, Java, FORTRAN) affect instruction set (App. B)

Figure 1.7 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 specific examples. The right-hand column also contains references to chapters and appendices that deal with the specific issues.

also must determine what the functional requirements are, which can be a major task. The requirements may be specific features inspired by the market. Application software often drives the choice of certain functional requirements by determining how the computer will be used. If a large body of software exists for a certain instruction set architecture, the architect may decide that a new computer 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 computer competitive in that market. Many of these requirements and features are examined in depth in later chapters.

Architects must also be aware of important trends in both the technology and the use of computers, as such trends not only affect future cost, but also the longevity of an architecture.

1.4 Trends in Technology

If an instruction set architecture is to be successful, it must be designed to survive rapid changes in computer technology. After all, a successful new instruction set architecture may last decades—for example, the core of the IBM mainframe has been in use for more than 40 years. An architect must plan for technology changes that can increase the lifetime of a successful computer.

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

- *Integrated circuit logic technology*—Transistor density increases by about 35% per year, quadrupling in somewhat over four years. Increases in die size are less predictable and slower, ranging from 10% to 20% per year. The combined effect is a growth rate in transistor count on a chip of about 40% to 55% per year. Device speed scales more slowly, as we discuss below.
- *Semiconductor DRAM* (dynamic random-access memory)—Capacity increases by about 40% per year, doubling roughly every two years.
- *Magnetic disk technology*—Prior to 1990, density increased by about 30% per year, doubling in three years. It rose to 60% per year thereafter, and increased to 100% per year in 1996. Since 2004, it has dropped back to 30% per year. Despite this roller coaster of rates of improvement, disks are still 50–100 times cheaper per bit than DRAM. This technology is central to Chapter 6, and we discuss the trends in detail there.
- *Network technology*—Network performance depends both on the performance of switches and on the performance of the transmission system. We discuss the trends in networking in Appendix E.

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

Although technology improves continuously, the impact of these improvements can be in discrete leaps, as a threshold that allows a new capability is reached. For example, when MOS technology reached a point in the early 1980s where between 25,000 and 50,000 transistors could fit on a single chip, it became possible to build a single-chip, 32-bit microprocessor. By the late 1980s, first-level caches could go on chip. By eliminating chip crossings within the processor and between the processor and the cache, a dramatic improvement in cost-performance and power-performance was possible. This design was simply infea-

sible 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.

Performance Trends: Bandwidth over Latency

As we shall see in Section 1.8, *bandwidth* or *throughput* is the total amount of work done in a given time, such as megabytes per second for a disk transfer. In contrast, *latency* or *response time* is the time between the start and the completion of an event, such as milliseconds for a disk access. Figure 1.8 plots the relative improvement in bandwidth and latency for technology milestones for microprocessors, memory, networks, and disks. Figure 1.9 describes the examples and milestones in more detail. Clearly, bandwidth improves much more rapidly than latency.

Performance is the primary differentiator for microprocessors and networks, so they have seen the greatest gains: 1000–2000X in bandwidth and 20–40X in latency. Capacity is generally more important than performance for memory and disks, so capacity has improved most, yet their bandwidth advances of 120–140X are still much greater than their gains in latency of 4–8X. Clearly, bandwidth has outpaced latency across these technologies and will likely continue to do so.

A simple rule of thumb is that bandwidth grows by at least the square of the improvement in latency. Computer designers should make plans accordingly.

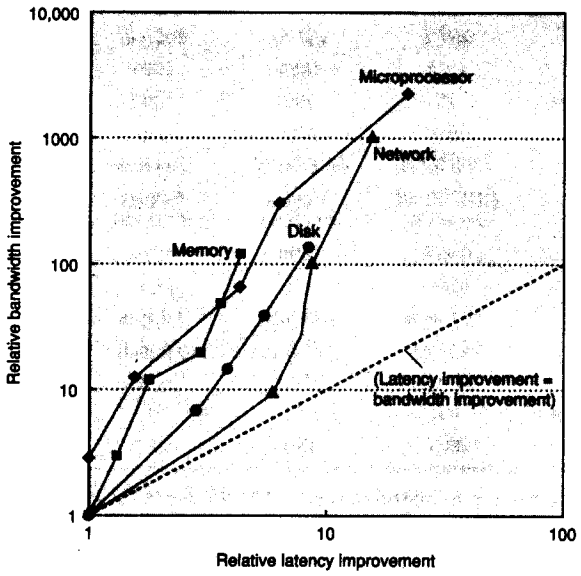

Figure 1.8 Log-log plot of bandwidth and latency milestones from Figure 1.9 relative to the first milestone. Note that latency improved about 10X while bandwidth improved about 100X to 1000X. From Patterson [2004].

Microprocessor	16-bit address/bus, microcoded	32-bit address.bus, microcoded	5-stage pipeline, on-chip I & D caches, FPU	2-way superscalar, 64-bit bus	Out-of-order 3-way superscalar	Out-of-order superpipelined, on-chip 1.2 cache
Product	Intel 80286	Intel 80386	Intel 80486	Intel Pentium	Intel Pentium Pro	Intel Pentium 4
Year	1982	1985	1989	1993	1997	2001
Die size (mm^2)	47	43	81	90	308	217
Transistors	134,000	275,000	1,200,000	3,100,000	5,500,000	42,000,000
Pins	68	132	168	273	387	423
Latency (clocks)	6	5	5	5	10	22
Bus width (bits)	16	32	32	64	64	64
Clock rate (MHz)	12.5	16	25	66	200	1500
Bandwidth (MIPS)	2	6	25	132	600	4500
Latency (ns)	320	313	200	76	50	15
Memory module	DRAM	Page mode DRAM	Fast page mode DRAM	Fast page mode DRAM	Synchronous DRAM	Double data rate SDRAM
Module width (bits)	16	16	32	64	64	64
Year	1980	1983	1986	1993	1997	2000
Mbits/DRAM chip	0.06	0.25	1	16	64	256
Die size (mm^2)	35	45	70	130	170	204
Pins/DRAM chip	16	16	18	20	54	66
Bandwidth (MBit/sec)	13	40	160	267	640	1600
Latency (ns)	225	170	125	75	62	52
Local area network	Ethernet	Fast Ethernet	Gigabit Ethernet	10 Gigabit Ethernet		
IEEE standard	802.3	803.3u	802.3ab	802.3ac		
Year	1978	1995	1999	2003		
Bandwidth (MBit/sec)	10	100	1000	10000		
Latency (μsec)	3000	500	340	190		
Hard disk	3600 RPM	5400 RPM	7200 RPM	10,000 RPM	15,000 RPM	
Product	CDC WrenI 94145-36	Seagate ST41600	Seagate ST15150	Seagate ST39102	Seagate ST373453	
Year	1983	1990	1994	1998	2003	
Capacity (GB)	0.03	1.4	4.3	9.1	73.4	
Disk form factor	5.25 inch	5.25 inch	3.5 inch	3.5 inch	3.5 inch	
Media diameter	5.25 inch	5.25 inch	3.5 inch	3.0 inch	2.5 inch	
Interface	ST-412	SCSI	SCSI	SCSI	SCSI	
Bandwidth (MBit/sec)	0.6	4	9	24	86	
Latency (ms)	48.3	17.1	12.7	8.8	5.7	

Figure 1.9 Performance milestones over 20 to 25 years for microprocessors, memory, networks, and disks. The microprocessor milestones are six generations of IA-32 processors, going from a 16-bit bus, microcoded 80286 to a 64-bit bus, superscalar, out-of-order execution, superpipelined Pentium 4. Memory module milestones go from 16-bit-wide, plain DRAM to 64-bit-wide double data rate synchronous DRAM. Ethernet advanced from 10 Mb/sec to 10 Gb/sec. Disk milestones are based on rotation speed, improving from 3600 RPM to 15,000 RPM. Each case is best-case bandwidth, and latency is the time for a simple operation assuming no contention. From Patterson [2004].

Scaling of Transistor Performance and Wires

Integrated circuit processes are characterized by the *feature size,* which is the minimum size of a transistor or a wire in either the x or y dimension. Feature sizes have decreased from 10 microns in 1971 to 0.09 microns in 2006; in fact, we have switched units, so production in 2006 is now referred to as "90 nanometers," and 65 nanometer chips are underway. Since the transistor count per square millimeter of silicon is determined by the surface area of a transistor, the density of transistors increases quadratically with a linear decrease in feature size.

The increase in transistor performance, however, is more complex. As feature sizes shrink, devices shrink quadratically in the horizontal dimension and also shrink in the vertical dimension. The shrink in the vertical dimension requires a reduction in operating voltage to maintain correct operation and reliability of the transistors. This combination of scaling factors leads to a complex interrelationship between transistor performance and process feature size. To a first approximation, transistor performance improves linearly with decreasing feature size.

The fact that transistor count improves quadratically with a linear improvement in transistor performance is both the challenge and the opportunity for which computer architects were created! In the early days of microprocessors, the higher rate of improvement in density was used to move quickly from 4-bit, to 8-bit, to 16-bit, to 32-bit microprocessors. More recently, density improvements have supported the introduction of 64-bit microprocessors as well as many of the innovations in pipelining and caches found in Chapters 2, 3, and 5.

Although transistors generally improve in performance with decreased feature size, wires in an integrated circuit do not. In particular, the signal delay for a wire increases in proportion to the product of its resistance and capacitance. Of course, as feature size shrinks, wires get shorter, but the resistance and capacitance per unit length get worse. This relationship is complex, since both resistance and capacitance depend on detailed aspects of the process, the geometry of a wire, the loading on a wire, and even the adjacency to other structures. There are occasional process enhancements, such as the introduction of copper, which provide one-time improvements in wire delay.

In general, however, wire delay scales poorly compared to transistor performance, creating additional challenges for the designer. In the past few years, wire delay has become a major design limitation for large integrated circuits and is often more critical than transistor switching delay. Larger and larger fractions of the clock cycle have been consumed by the propagation delay of signals on wires. In 2001, the Pentium 4 broke new ground by allocating 2 stages of its 20+-stage pipeline just for propagating signals across the chip.

1.5 Trends in Power in Integrated Circuits

Power also provides challenges as devices are scaled. First, power must be brought in and distributed around the chip, and modern microprocessors use

hundreds of pins and multiple interconnect layers for just power and ground. Second, power is dissipated as heat and must be removed.

For CMOS chips, the traditional dominant energy consumption has been in switching transistors, also called *dynamic power*. The power required per transistor is proportional to the product of the load capacitance of the transistor, the square of the voltage, and the frequency of switching, with watts being the unit:

$$\text{Power}_{\text{dynamic}} = 1/2 \times \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency switched}$$

Mobile devices care about battery life more than power, so energy is the proper metric, measured in joules:

$$\text{Energy}_{\text{dynamic}} = \text{Capacitive load} \times \text{Voltage}^2$$

Hence, dynamic power and energy are greatly reduced by lowering the voltage, and so voltages have dropped from 5V to just over 1V in 20 years. The capacitive load is a function of the number of transistors connected to an output and the technology, which determines the capacitance of the wires and the transistors. For a fixed task, slowing clock rate reduces power, but not energy.

Example Some microprocessors today are designed to have adjustable voltage, so that a 15% reduction in voltage may result in a 15% reduction in frequency. What would be the impact on dynamic power?

Answer Since the capacitance is unchanged, the answer is the ratios of the voltages and frequencies:

$$\frac{\text{Power}_{\text{new}}}{\text{Power}_{\text{old}}} = \frac{(\text{Voltage} \times 0.85)^2 \times (\text{Frequency switched} \times 0.85)}{\text{Voltage}^2 \times \text{Frequency switched}} = 0.85^3 = 0.61$$

thereby reducing power to about 60% of the original.

As we move from one process to the next, the increase in the number of transistors switching, and the frequency with which they switch, dominates the decrease in load capacitance and voltage, leading to an overall growth in power consumption and energy. The first microprocessors consumed tenths of a watt, while a 3.2 GHz Pentium 4 Extreme Edition consumes 135 watts. Given that this heat must be dissipated from a chip that is about 1 cm on a side, we are reaching the limits of what can be cooled by air. Several Intel microprocessors have temperature diodes to reduce activity automatically if the chip gets too hot. For example, they may reduce voltage and clock frequency or the instruction issue rate.

Distributing the power, removing the heat, and preventing hot spots have become increasingly difficult challenges. Power is now the major limitation to using transistors; in the past it was raw silicon area. As a result of this limitation, most microprocessors today turn off the clock of inactive modules to save energy

and dynamic power. For example, if no floating-point instructions are executing, the clock of the floating-point unit is disabled.

Although dynamic power is the primary source of power dissipation in CMOS, static power is becoming an important issue because leakage current flows even when a transistor is off:

$$\text{Power}_{\text{static}} = \text{Current}_{\text{static}} \times \text{Voltage}$$

Thus, increasing the number of transistors increases power even if they are turned off, and leakage current increases in processors with smaller transistor sizes. As a result, very low power systems are even gating the voltage to inactive modules to control loss due to leakage. In 2006, the goal for leakage is 25% of the total power consumption, with leakage in high-performance designs sometimes far exceeding that goal. As mentioned before, the limits of air cooling have led to exploration of multiple processors on a chip running at lower voltages and clock rates.

1.6 Trends in Cost

Although there are computer designs where costs tend to be less important—specifically supercomputers—cost-sensitive designs are of growing significance. Indeed, in the past 20 years, the use of technology improvements to lower cost, as well as increase performance, has been a major theme in the computer industry.

Textbooks often ignore the cost half of cost-performance because costs change, thereby dating books, and because the issues are subtle and differ across industry segments. Yet an understanding of cost and its factors is essential for designers to 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 discusses the major factors that influence the cost of a computer and how these factors are changing over time.

The Impact of Time, Volume, and Commodification

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

Understanding how the learning curve improves yield is critical to projecting costs over a product's life. One example is that the price per megabyte of DRAM has dropped over the long term by 40% per year. Since DRAMs tend to be priced

in close relationship to cost—with the exception of periods when there is a shortage or an oversupply—price and cost of DRAM track closely.

Microprocessor prices also drop over time, but because they are less standardized than DRAMs, the relationship between price and cost is more complex. In a period of significant competition, price tends to track cost closely, although microprocessor vendors probably rarely sell at a loss. Figure 1.10 shows processor price trends for Intel microprocessors.

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. Moreover, volume decreases the amount of development cost that must be amortized by each computer, thus allowing cost and selling price to be closer.

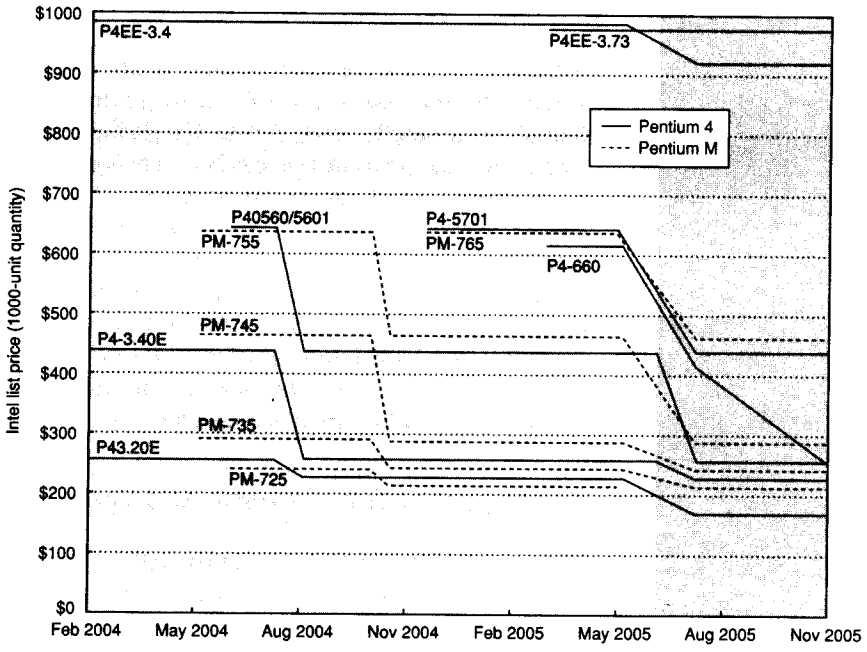

Figure 1.10 The price of an Intel Pentium 4 and Pentium M at a given frequency decreases over time as yield enhancements decrease the cost of a good die and competition forces price reductions. The most recent introductions will continue to decrease until they reach similar prices to the lowest-cost parts available today ($200). Such price decreases assume a competitive environment where price decreases track cost decreases closely. Data courtesy of *Microprocessor Report*, May 2005.

Commodities are products that are sold by multiple vendors in large volumes and are essentially identical. Virtually all the products sold on the shelves of grocery stores are commodities, as are standard DRAMs, disks, monitors, and keyboards. In the past 15 years, much of the low end of the computer business has become a commodity business focused on building desktop and laptop computers running Microsoft Windows.

Because many vendors ship virtually identical products, it is highly competitive. Of course, this competition decreases the gap between cost and selling price, but it also decreases cost. Reductions occur because a commodity market has both volume and a clear product definition, which allows multiple suppliers to compete in building components for the commodity product. As a result, the overall product cost is lower because of the competition among the suppliers of the components and the volume efficiencies the suppliers can achieve. This has led to the low end of the computer business being able to achieve better price-performance than other sectors and yielded greater growth at the low end, although with very limited profits (as is typical in any commodity business).

Cost of an Integrated Circuit

Why would a computer architecture book have a section on integrated circuit costs? In an increasingly competitive computer marketplace where standard parts—disks, DRAMs, and so on—are becoming a significant portion of any system's cost, integrated circuit costs are becoming a greater portion of the cost that varies between computers, especially in the high-volume, cost-sensitive portion of the market. Thus, computer designers must understand the costs of chips to understand the costs of current computers.

Although the costs of integrated circuits have dropped exponentially, the basic process of silicon manufacture is unchanged: A *wafer* is still tested and chopped into *dies* that are packaged (see Figures 1.11 and 1.12). 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.

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

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

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

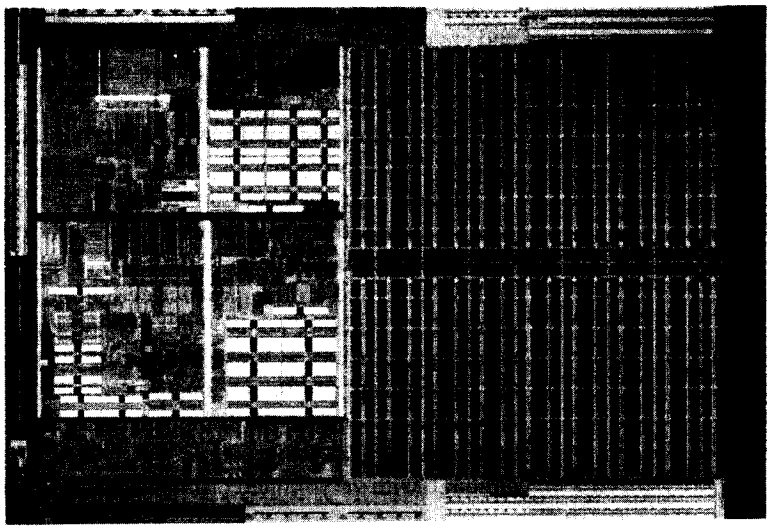

Figure 1.11 Photograph of an AMD Opteron microprocessor die. (Courtesy AMD.)

The number of dies per wafer is approximately 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.

Example Find the number of dies per 300 mm (30 cm) wafer for a die that is 1.5 cm on a side.

Answer The die area is 2.25 cm^2. Thus

$$\text{Dies per wafer} = \frac{\pi \times (30/2)^2}{2.25} - \frac{\pi \times 30}{\sqrt{2 \times 2.25}} = \frac{706.9}{2.25} - \frac{94.2}{2.12} = 270$$

However, this only gives the maximum number of dies per wafer. The critical question is: What is the fraction of good dies on a wafer number, or the *die yield*? A simple 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:

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

Figure 1.12 This 300mm wafer contains 117 AMD Opteron chips implemented in a 90 nm process. (Courtesy AMD.)

The formula is an empirical model developed by looking at the yield of many manufacturing lines. *Wafer yield* accounts for wafers that are completely bad and so need not be tested. For simplicity, we'll just assume the wafer yield is 100%. Defects per unit area is a measure of the random manufacturing defects that occur. In 2006, these value is typically 0.4 defects per square centimeter for 90 nm, as it depends on the maturity of the process (recall the learning curve, mentioned earlier). Lastly, α is a parameter that corresponds roughly to the number of critical masking levels, a measure of manufacturing complexity. For multilevel metal CMOS processes in 2006, a good estimate is $\alpha = 4.0$.

Example Find the die yield for dies that are 1.5 cm on a side and 1.0 cm on a side, assuming a defect density of 0.4 per cm^2 and α is 4.

Answer The total die areas are 2.25 cm^2 and 1.00 cm^2. For the larger die, the yield is

$$\text{Die yield} = \left(1 + \frac{0.4 \times 2.25}{4.0}\right)^{-4} = 0.44$$

For the smaller die, it is $\text{Die yield} = \left(1 + \frac{0.4 \times 1.00}{4.0}\right)^{-4} = 0.68$

That is, less than half of all the large die are good but more than two-thirds of the small die are good.

The bottom line is the number of good dies per wafer, which comes from multiplying dies per wafer by die yield to incorporate the effects of defects. The examples above predict about 120 good 2.25 cm^2 dies from the 300 mm wafer and 435 good 1.00 cm^2 dies. Many 32-bit and 64-bit microprocessors in a modern 90 nm technology fall between these two sizes. Low-end embedded 32-bit processors are sometimes as small as 0.25 cm^2, and processors used for embedded control (in printers, automobiles, etc.) are often less than 0.1 cm^2.

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

Processing of a 300 mm (12-inch) diameter wafer in a leading-edge technology costs between $5000 and $6000 in 2006. Assuming a processed wafer cost of $5500, the cost of the 1.00 cm^2 die would be around $13, but the cost per die of the 2.25 cm^2 die would be about $46, or almost four times the cost for a die that is a little over twice as large.

What should a computer designer remember about chip costs? The manufacturing process dictates the wafer cost, wafer yield, and defects per unit area, so the sole control of the designer is die area. In practice, because the number of defects per unit area is small, the number of good dies per wafer, and hence the cost per die, grows roughly as the square of the die area. The computer designer affects die size, and hence cost, both by what functions are included on or excluded from the die and by the number of I/O pins.

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

The above analysis has focused on the variable costs of producing a functional die, which is appropriate for high-volume integrated circuits. There is, however, one very important part of the fixed cost that can significantly affect the

cost of an integrated circuit for low volumes (less than 1 million parts), namely, the cost of a mask set. Each step in the integrated circuit process requires a separate mask. Thus, for modern high-density fabrication processes with four to six metal layers, mask costs exceed $1 million. Obviously, this large fixed cost affects the cost of prototyping and debugging runs and, for small-volume production, can be a significant part of the production cost. Since mask costs are likely to continue to increase, designers may incorporate reconfigurable logic to enhance the flexibility of a part, or choose to use gate arrays (which have fewer custom mask levels) and thus reduce the cost implications of masks.

Cost versus Price

With the commoditization of the computers, the margin between the cost to the manufacture a product and the price the product sells for has been shrinking. Those margins pay for a company's research and development (R&D), marketing, sales, manufacturing equipment maintenance, building rental, cost of financing, pretax profits, and taxes. 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.

1.7 Dependability

Historically, integrated circuits were one of the most reliable components of a computer. Although their pins may be vulnerable, and faults may occur over communication channels, the error rate inside the chip was very low. That conventional wisdom is changing as we head to feature sizes of 65 nm and smaller, as both transient faults and permanent faults will become more commonplace, so architects must design systems to cope with these challenges. This section gives an quick overview of the issues in dependability, leaving the official definition of the terms and approaches to Section 6.3.

Computers are designed and constructed at different layers of abstraction. We can descend recursively down through a computer seeing components enlarge themselves to full subsystems until we run into individual transistors. Although some faults are widespread, like the loss of power, many can be limited to a single component in a module. Thus, utter failure of a module at one level may be considered merely a component error in a higher-level module. This distinction is helpful in trying to find ways to build dependable computers.

One difficult question is deciding when a system is operating properly. This philosophical point became concrete with the popularity of Internet services. Infrastructure providers started offering *Service Level Agreements* (SLA) or *Service Level Objectives* (SLO) to guarantee that their networking or power service would be dependable. For example, they would pay the customer a penalty if they did not meet an agreement more than some hours per month. Thus, an SLA could be used to decide whether the system was up or down.

Systems alternate between two states of service with respect to an SLA:

1. *Service accomplishment,* where the service is delivered as specified
2. *Service interruption,* where the delivered service is different from the SLA

Transitions between these two states are caused by *failures* (from state 1 to state 2) or *restorations* (2 to 1). Quantifying these transitions leads to the two main measures of dependability:

- *Module reliability* is a measure of the continuous service accomplishment (or, equivalently, of the time to failure) from a reference initial instant. Hence, the *mean time to failure* (MTTF) is a reliability measure. The reciprocal of MTTF is a rate of failures, generally reported as failures per billion hours of operation, or *FIT* (for *failures in time).*Thus, an MTTF of 1,000,000 hours equals $10^9/10^6$ or 1000 FIT. Service interruption is measured as *mean time to repair* (MTTR). *Mean time between failures* (MTBF) is simply the sum of MTTF + MTTR. Although MTBF is widely used, MTTF is often the more appropriate term. If a collection of modules have exponentially distributed lifetimes—meaning that the age of a module is not important in probability of failure—the overall failure rate of the collection is the sum of the failure rates of the modules.
- *Module availability* is a measure of the service accomplishment with respect to the alternation between the two states of accomplishment and interruption. For nonredundant systems with repair, module availability is

$$\text{Module availability} = \frac{\text{MTTF}}{(\text{MTTF} + \text{MTTR})}$$

Note that reliability and availability are now quantifiable metrics, rather than synonyms for dependability. From these definitions, we can estimate reliability of a system quantitatively if we make some assumptions about the reliability of components and that failures are independent.

Example Assume a disk subsystem with the following components and MTTF:

- 10 disks, each rated at 1,000,000-hour MTTF
- 1 SCSI controller, 500,000-hour MTTF
- 1 power supply, 200,000-hour MTTF
- 1 fan, 200,000-hour MTTF
- 1 SCSI cable, 1,000,000-hour MTTF

Using the simplifying assumptions that the lifetimes are exponentially distributed and that failures are independent, compute the MTTF of the system as a whole.

Answer The sum of the failure rates is

$$\text{Failure rate}_{\text{system}} = 10 \times \frac{1}{1{,}000{,}000} + \frac{1}{500{,}000} + \frac{1}{200{,}000} + \frac{1}{200{,}000} + \frac{1}{1{,}000{,}000}$$

$$= \frac{10+2+5+5+1}{1{,}000{,}000 \text{ hours}} = \frac{23}{1{,}000{,}000} = \frac{23{,}000}{1{,}000{,}000{,}000 \text{ hours}}$$

or 23,000 FIT. The MTTF for the system is just the inverse of the failure rate:

$$\text{MTTF}_{\text{system}} = \frac{1}{\text{Failure rate}_{\text{system}}} = \frac{1{,}000{,}000{,}000 \text{ hours}}{23{,}000} = 43{,}500 \text{ hours}$$

or just under 5 years.

The primary way to cope with failure is redundancy, either in time (repeat the operation to see if it still is erroneous) or in resources (have other components to take over from the one that failed). Once the component is replaced and the system fully repaired, the dependability of the system is assumed to be as good as new. Let's quantify the benefits of redundancy with an example.

Example Disk subsystems often have redundant power supplies to improve dependability. Using the components and MTTFs from above, calculate the reliability of a redundant power supply. Assume one power supply is sufficient to run the disk subsystem and that we are adding one redundant power supply.

Answer We need a formula to show what to expect when we can tolerate a failure and still provide service. To simplify the calculations, we assume that the lifetimes of the components are exponentially distributed and that there is no dependency between the component failures. MTTF for our redundant power supplies is the mean time until one power supply fails divided by the chance that the other will fail before the first one is replaced. Thus, if the chance of a second failure before repair is small, then MTTF of the pair is large.

Since we have two power supplies and independent failures, the mean time until one disk fails is $\text{MTTF}_{\text{power supply}} / 2$. A good approximation of the probability of a second failure is MTTR over the mean time until the other power supply fails. Hence, a reasonable approximation for a redundant pair of power supplies is

$$\text{MTTF}_{\text{power supply pair}} = \frac{\text{MTTF}_{\text{power supply}}/2}{\frac{\text{MTTR}_{\text{power supply}}}{\text{MTTF}_{\text{power supply}}}} = \frac{\text{MTTF}^2_{\text{power supply}}/2}{\text{MTTR}_{\text{power supply}}} = \frac{\text{MTTF}^2_{\text{power supply}}}{2 \times \text{MTTR}_{\text{power supply}}}$$

Using the MTTF numbers above, if we assume it takes on average 24 hours for a human operator to notice that a power supply has failed and replace it, the reliability of the fault tolerant pair of power supplies is

$$\text{MTTF}_{\text{power supply pair}} = \frac{\text{MTTF}^2_{\text{power supply}}}{2 \times \text{MTTR}_{\text{power supply}}} = \frac{200{,}000^2}{2 \times 24} \cong 830{,}000{,}000$$

making the pair about 4150 times more reliable than a single power supply.

Having quantified the cost, power, and dependability of computer technology, we are ready to quantify performance.

1.8 Measuring, Reporting, and Summarizing Performance

When we say one computer is faster than another is, what do we mean? The user of a desktop computer may say a computer is faster when a program runs in less time, while an Amazon.com administrator may say a computer is faster when it completes more transactions per 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 administrator 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 computers, say, X and Y. The phrase "X is faster than Y" is used here to mean that the response time or execution time is lower on X than on Y for the given task. In particular, "X is n times faster than Y" will mean

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

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

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

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

Unfortunately, time is not always the metric quoted in comparing the performance of computers. Our position is that the only consistent and reliable measure of performance is the execution time of real programs, and that all proposed alternatives to time as the metric or to real programs as the items measured have eventually led to misleading claims or even mistakes in computer design.

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 processor 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 consider this activity. *CPU time* recognizes this distinction and means the time the processor is computing, *not* includ-

ing 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.)

Computer users who routinely run the same programs would be the perfect candidates to evaluate a new computer. To evaluate a new system the users would simply compare the execution time of their *workloads*—the mixture of programs and operating system commands that users run on a computer. Few are in this happy situation, however. Most must rely on other methods to evaluate computers, and often other evaluators, hoping that these methods will predict performance for their usage of the new computer.

Benchmarks

The best choice of benchmarks to measure performance are real applications, such as a compiler. Attempts at running programs that are much simpler than a real application have led to performance pitfalls. Examples include

- *kernels*, which are small, key pieces of real applications;
- *toy programs*, which are 100-line programs from beginning programming assignments, such as quicksort; and
- *synthetic benchmarks*, which are fake programs invented to try to match the profile and behavior of real applications, such as Dhrystone.

All three are discredited today, usually because the compiler writer and architect can conspire to make the computer appear faster on these stand-in programs than on real applications.

Another issue is the conditions under which the benchmarks are run. One way to improve the performance of a benchmark has been with benchmark-specific flags; these flags often caused transformations that would be illegal on many programs or would slow down performance on others. To restrict this process and increase the significance of the results, benchmark developers often require the vendor to use one compiler and one set of flags for all the programs in the same language (C or FORTRAN). In addition to the question of compiler flags, another question is whether source code modifications are allowed. There are three different approaches to addressing this question:

1. No source code modifications are allowed.
2. Source code modifications are allowed, but are essentially impossible. For example, database benchmarks rely on standard database programs that are tens of millions of lines of code. The database companies are highly unlikely to make changes to enhance the performance for one particular computer.
3. Source modifications are allowed, as long as the modified version produces the same output.

The key issue that benchmark designers face in deciding to allow modification of the source is whether such modifications will reflect real practice and provide

useful insight to users, or whether such modifications simply reduce the accuracy of the benchmarks as predictors of real performance.

To overcome the danger of placing too many eggs in one basket, collections of benchmark applications, called *benchmark suites*, are a popular measure of 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. The goal of a benchmark suite is that it will characterize the relative performance of two computers, particularly for programs not in the suite that customers are likely to run.

As a cautionary example, the EDN Embedded Microprocessor Benchmark Consortium (or EEMBC, pronounced "embassy") is a set of 41 kernels used to predict performance of different embedded applications: automotive/industrial, consumer, networking, office automation, and telecommunications. EEMBC reports unmodified performance and "full fury" performance, where almost anything goes. Because they use kernels, and because of the reporting options, EEMBC does not have the reputation of being a good predictor of relative performance of different embedded computers in the field. The synthetic program Dhrystone, which EEMBC was trying to replace, is still reported in some embedded circles.

One of the most successful attempts to create standardized benchmark application suites has been the SPEC (Standard Performance Evaluation Corporation), which had its roots in the late 1980s efforts to deliver better benchmarks for workstations. Just as the computer industry has evolved over time, so has the need for different benchmark suites, and there are now SPEC benchmarks to cover different application classes. All the SPEC benchmark suites and their reported results are found at *www.spec.org*.

Although we focus our discussion on the SPEC benchmarks in many of the following sections, there are also many benchmarks developed for PCs running the Windows operating system.

Desktop Benchmarks

Desktop benchmarks divide into two broad classes: processor-intensive benchmarks and graphics-intensive benchmarks, although many graphics benchmarks include intensive processor activity. SPEC originally created a benchmark set focusing on processor performance (initially called SPEC89), which has evolved into its fifth generation: SPEC CPU2006, which follows SPEC2000, SPEC95 SPEC92, and SPEC89. SPEC CPU2006 consists of a set of 12 integer benchmarks (CINT2006) and 17 floating-point benchmarks (CFP2006). Figure 1.13 describes the current SPEC benchmarks and their ancestry.

SPEC benchmarks are real programs modified to be portable and to minimize the effect of I/O on performance. The integer benchmarks vary from part of a C compiler to a chess program to a quantum computer simulation. The floating-point benchmarks include structured grid codes for finite element modeling, par-

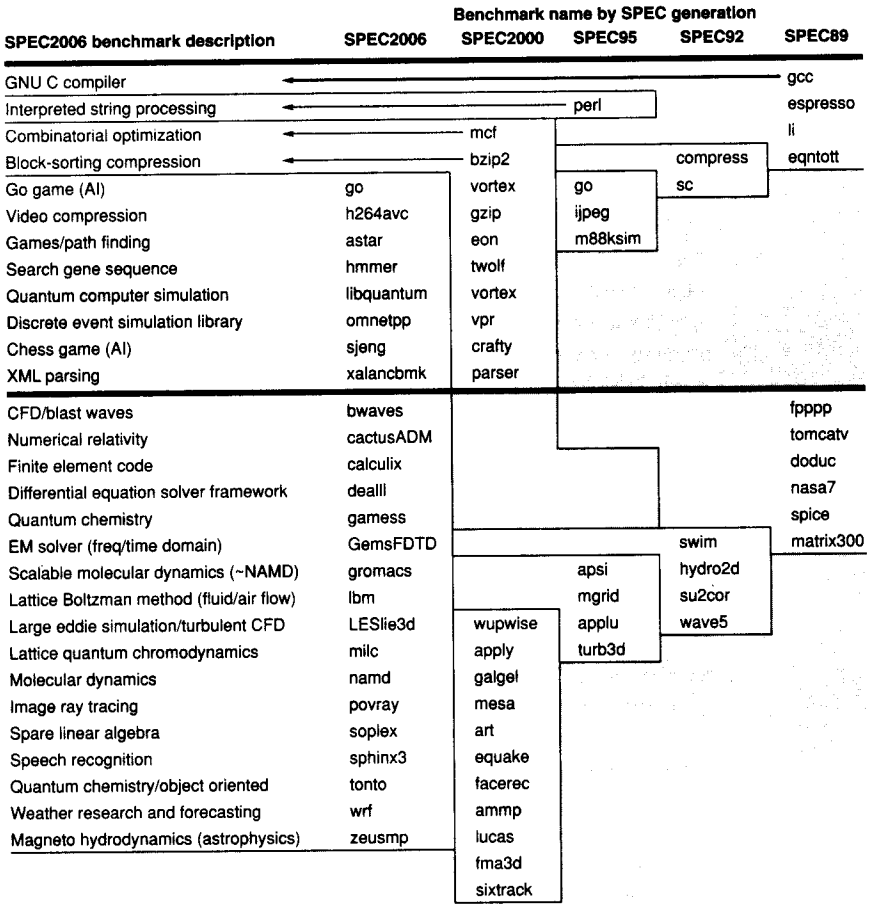

Figure 1.13 SPEC2006 programs and the evolution of the SPEC benchmarks over time, with integer programs above the line and floating-point programs below the line. Of the 12 SPEC2006 integer programs, 9 are written in C, and the rest in C++. For the floating-point programs the split is 6 in FORTRAN, 4 in C++, 3 in C, and 4 in mixed C and Fortran. The figure shows all 70 of the programs in the 1989, 1992, 1995, 2000, and 2006 releases. The benchmark descriptions on the left are for SPEC2006 only and do not apply to earlier ones. Programs in the same row from different generations of SPEC are generally not related; for example, fpppp is not a CFD code like bwaves. Gcc is the senior citizen of the group. Only 3 integer programs and 3 floating-point programs survived three or more generations. Note that all the floating-point programs are new for SPEC2006. Although a few are carried over from generation to generation, the version of the program changes and either the input or the size of the benchmark is often changed to increase its running time and to avoid perturbation in measurement or domination of the execution time by some factor other than CPU time.

ticle method codes for molecular dynamics, and sparse linear algebra codes for fluid dynamics. The SPEC CPU suite is useful for processor benchmarking for both desktop systems and single-processor servers. We will see data on many of these programs throughout this text.

In Section 1.11, we describe pitfalls that have occurred in developing the SPEC benchmark suite, as well as the challenges in maintaining a useful and predictive benchmark suite. Although SPEC CPU2006 is aimed at processor performance, SPEC also has benchmarks for graphics and Java.

Server Benchmarks

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

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

Transaction-processing (TP) benchmarks measure the ability of a system to handle transactions, which consist of database accesses and updates. Airline reservation systems and bank ATM systems are typical simple examples of TP; more sophisticated TP systems involve complex databases and decision-making. In the mid-1980s, a group of concerned engineers formed the vendor-independent Transaction Processing Council (TPC) to try to create realistic and fair benchmarks for TP. The TPC benchmarks are described at *www.tpc.org*.

The first TPC benchmark, TPC-A, was published in 1985 and has since been replaced and enhanced by several different benchmarks. TPC-C, initially created in 1992, simulates a complex query environment. TPC-H models ad hoc decision support—the queries are unrelated and knowledge of past queries cannot be used to optimize future queries. TPC-W is a transactional Web benchmark. The workload is performed in a controlled Internet commerce environment that simulates the activities of a business-oriented transactional Web server. The most recent is TPC-App, an application server and Web services benchmark. The workload simulates the activities of a business-to-business transactional application server operating in a 24x7 environment.

All the TPC benchmarks measure performance in transactions per second. In addition, they include a response time requirement, so that throughput perfor-

mance is measured only when the response time limit is met. To model real-world systems, higher transaction rates are also associated with larger systems, in terms of both users and the database to which the transactions are applied. Finally, the system cost for a benchmark system must also be included, allowing accurate comparisons of cost-performance.

Reporting Performance Results

The guiding principle of reporting performance measurements should be *reproducibility*—list everything another experimenter would need to duplicate the results. A SPEC benchmark report requires an extensive description of the computer and the compiler flags, as well as the publication of both the baseline and optimized results. In addition to hardware, software, and baseline tuning parameter descriptions, a SPEC report contains the actual performance times, shown both in tabular form and as a graph. A TPC benchmark report is even more complete, since it must include results of a benchmarking audit and cost information. These reports are excellent sources for finding the real cost of computing systems, since manufacturers compete on high performance and cost-performance.

Summarizing Performance Results

In practical computer design, you must evaluate myriads of design choices for their relative quantitative benefits across a suite of benchmarks believed to be relevant. Likewise, consumers trying to choose a computer will rely on performance measurements from benchmarks, which hopefully are similar to the user's applications. In both cases, it is useful to have measurements for a suite of benchmarks so that the performance of important applications is similar to that of one or more benchmarks in the suite and that variability in performance can be understood. In the ideal case, the suite resembles a statistically valid sample of the application space, but such a sample requires more benchmarks than are typically found in most suites and requires a randomized sampling, which essentially no benchmark suite uses.

Once we have chosen to measure performance with a benchmark suite, we would like to be able to summarize the performance results of the suite in a single number. A straightforward approach to computing a summary result would be to compare the arithmetic means of the execution times of the programs in the suite. Alas, some SPEC programs take four times longer than others, so those programs would be much more important if the arithmetic mean were the single number used to summarize performance. An alternative would be to add a weighting factor to each benchmark and use the weighted arithmetic mean as the single number to summarize performance. The problem would be then how to pick weights; since SPEC is a consortium of competing companies, each company might have their own favorite set of weights, which would make it hard to reach consensus. One approach is to use weights that make all programs execute an equal time on

some reference computer, but this biases the results to the performance characteristics of the reference computer.

Rather than pick weights, we could normalize execution times to a reference computer by dividing the time on the reference computer by the time on the computer being rated, yielding a ratio proportional to performance. SPEC uses this approach, calling the ratio the SPECRatio. It has a particularly useful property that it matches the way we compare computer performance throughout this text—namely, comparing performance ratios. For example, suppose that the SPECRatio of computer A on a benchmark was 1.25 times higher than computer B; then you would know

$$1.25 = \frac{\text{SPECRatio}_A}{\text{SPECRatio}_B} = \frac{\dfrac{\text{Execution time}_{\text{reference}}}{\text{Execution time}_A}}{\dfrac{\text{Execution time}_{\text{reference}}}{\text{Execution time}_B}} = \frac{\text{Execution time}_B}{\text{Execution time}_A} = \frac{\text{Performance}_A}{\text{Performance}_B}$$

Notice that the execution times on the reference computer drop out and the choice of the reference computer is irrelevant when the comparisons are made as a ratio, which is the approach we consistently use. Figure 1.14 gives an example.

Because a SPECRatio is a ratio rather than an absolute execution time, the mean must be computed using the *geometric* mean. (Since SPECRatios have no units, comparing SPECRatios arithmetically is meaningless.) The formula is

$$\text{Geometric mean} = \sqrt[n]{\prod_{i=1}^{n} sample_i}$$

In the case of SPEC, $sample_i$ is the SPECRatio for program i. Using the geometric mean ensures two important properties:

1. The geometric mean of the ratios is the same as the ratio of the geometric means.
2. The ratio of the geometric means is equal to the geometric mean of the performance ratios, which implies that the choice of the reference computer is irrelevant.

Hence, the motivations to use the geometric mean are substantial, especially when we use performance ratios to make comparisons.

Example Show that the ratio of the geometric means is equal to the geometric mean of the performance ratios, and that the reference computer of SPECRatio matters not.

Benchmarks	Ultra 5 Time (sec)	Opteron Time (sec)	SPECRatio	Itanium 2 Time (sec)	SPECRatio	Opteron/Itanium Times (sec)	Itanium/Opteron SPECRatios
wupwise	1600	51.5	31.06	56.1	28.53	0.92	0.92
swim	3100	125.0	24.73	70.7	43.85	1.77	1.77
mgrid	1800	98.0	18.37	65.8	27.36	1.49	1.49
applu	2100	94.0	22.34	50.9	41.25	1.85	1.85
mesa	1400	64.6	21.69	108.0	12.99	0.60	0.60
galgel	2900	86.4	33.57	40.0	72.47	2.16	2.16
art	2600	92.4	28.13	21.0	123.67	4.40	4.40
equake	1300	72.6	17.92	36.3	35.78	2.00	2.00
facerec	1900	73.6	25.80	86.9	21.86	0.85	0.85
ammp	2200	136.0	16.14	132.0	16.63	1.03	1.03
lucas	2000	88.8	22.52	107.0	18.76	0.83	0.83
fma3d	2100	120.0	17.48	131.0	16.09	0.92	0.92
sixtrack	1100	123.0	8.95	68.8	15.99	1.79	1.79
apsi	2600	150.0	17.36	231.0	11.27	0.65	0.65
Geometric mean			20.86		27.12	1.30	1.30

Figure 1.14 SPECfp2000 execution times (in seconds) for the Sun Ultra 5—the reference computer of SPEC2000—and execution times and SPECRatios for the AMD Opteron and Intel Itanium 2. (SPEC2000 multiplies the ratio of execution times by 100 to remove the decimal point from the result, so 20.86 is reported as 2086.) The final two columns show the ratios of execution times and SPECratios. This figure demonstrates the irrelevance of the reference computer in relative performance. The ratio of the execution times is identical to the ratio of the SPECRatios, and the ratio of the geometric means (27.12/20.86 = 1.30) is identical to the geometric mean of the ratios (1.30).

Answer Assume two computers A and B and a set of SPECRatios for each.

$$\frac{\text{Geometric mean}_A}{\text{Geometric mean}_B} = \frac{\sqrt[n]{\prod_{i=1}^{n} \text{SPECRatio A}_i}}{\sqrt[n]{\prod_{i=1}^{n} \text{SPECRatio B}_i}} = \sqrt[n]{\prod_{i=1}^{n} \frac{\text{SPECRatio A}_i}{\text{SPECRatio B}_i}}$$

$$= \sqrt[n]{\prod_{i=1}^{n} \frac{\dfrac{\text{Execution time}_{\text{reference}_i}}{\text{Execution time}_{A_i}}}{\dfrac{\text{Execution time}_{\text{reference}_i}}{\text{Execution time}_{B_i}}}} = \sqrt[n]{\prod_{i=1}^{n} \frac{\text{Execution time}_{B_i}}{\text{Execution time}_{A_i}}} = \sqrt[n]{\prod_{i=1}^{n} \frac{\text{Performance}_{A_i}}{\text{Performance}_{B_i}}}$$

That is, the ratio of the geometric means of the SPECRatios of A and B is the geometric mean of the performance ratios of A to B of all the benchmarks in the suite. Figure 1.14 demonstrates the validity using examples from SPEC.

A key question is whether a single mean summarizes the performance of the programs in the benchmark suite well. If we characterize the variability of the distribution, using the standard deviation, we can decide whether the mean is likely to be a good predictor. The standard deviation is more informative if we know the distribution has one of several standard forms.

One useful possibility is the well-known bell-shaped *normal distribution*, whose sample data are, of course, symmetric around the mean. Another is the *lognormal distribution*, where the logarithms of the data—not the data itself—are normally distributed on a logarithmic scale, and thus symmetric on that scale. (On a linear scale, a lognormal is not symmetric, but has a long tail to the right.)

For example, if each of two systems is 10X faster than the other on two different benchmarks, the relative performance is the set of ratios {.1, 10}. However, the performance summary should be equal performance. That is, the average should be 1.0, which in fact is true on a logarithmic scale.

To characterize variability about the arithmetic mean, we use the arithmetic standard deviation (stdev), often called σ. It is defined as:

$$\text{stdev} = \sqrt{\sum_{i=1}^{n} (sample_i - \text{Mean})^2}$$

Like the geometric mean, the geometric standard deviation is multiplicative rather than additive. For working with the geometric mean and the geometric standard deviation, we can simply take the natural logarithm of the samples, compute the standard mean and standard deviation, and then take the exponent to convert back. This insight allows us to describe the multiplicative versions of mean and standard deviation (gstdev), also often called σ, as

$$\text{Geometric mean} = \exp\left(\frac{1}{n} \times \sum_{i=1}^{n} \ln(sample_i)\right)$$

$$\text{gstdev} = \exp\left(\sqrt{\frac{\sum_{i=1}^{n} (\ln(sample_i) - \ln(\text{Geometric mean}))^2}{n}}\right)$$

Note that functions provided in a modern spreadsheet program, like EXP( ) and LN( ), make it easy to calculate the geometric mean and the geometric standard deviation.

For a lognormal distribution, we expect that 68% of the samples fall in the range [Mean / gstdev, Mean × gstdev], 95% within [Mean / gstdev^2, Mean × gstdev^2], and so on.

Example Using the data in Figure 1.14, calculate the geometric standard deviation and the percentage of the results that fall within a single standard deviation of the geometric mean. Are the results compatible with a lognormal distribution?

Answer The geometric means are 20.86 for Opteron and 27.12 for Itanium 2. As you might guess from the SPECRatios, the standard deviation for the Itanium 2 is much higher—1.93 versus 1.38—indicating that the results will differ more widely from the mean, and therefore are likely less predictable. The single standard deviation range is [27.12 / 1.93, 27.12 × 1.93] or [14.06, 52.30] for Itanium 2 and [20.86 / 1.38, 20.86 × 1.38] or [15.12, 28.76] for Opteron. For Itanium 2, 10 of 14 benchmarks (71%) fall within one standard deviation; for Opteron, it is 11 of 14 (78%). Thus, both results are quite compatible with a lognormal distribution.

1.9 Quantitative Principles of Computer Design

Now that we have seen how to define, measure, and summarize performance, cost, dependability, and power, we can explore guidelines and principles that are useful in the design and analysis of computers. This section introduces important observations about design, as well as two equations to evaluate alternatives.

Take Advantage of Parallelism

Taking advantage of parallelism is one of the most important methods for improving performance. Every chapter in this book has an example of how performance is enhanced through the exploitation of parallelism. We give three brief examples, which are expounded on in later chapters.

Our first example is the use of parallelism at the system level. To improve the throughput performance on a typical server benchmark, such as SPECWeb or TPC-C, multiple processors and multiple disks can be used. The workload of handling requests can then be spread among the processors and disks, resulting in improved throughput. Being able to expand memory and the number of processors and disks is called *scalability*, and it is a valuable asset for servers.

At the level of an individual processor, taking advantage of parallelism among instructions is critical to achieving high performance. One of the simplest ways to do this is through pipelining. The basic idea behind pipelining, which is explained in more detail in Appendix A and is a major focus of Chapter 2, is to overlap instruction execution to reduce the total time to complete an instruction sequence. A key insight that allows pipelining to work is that not every instruction depends on its immediate predecessor, and thus, executing the instructions completely or partially in parallel may be possible.

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

Principle of Locality

Important fundamental observations have come from properties of programs. The most important program property that we regularly exploit is the *principle of locality:* Programs tend to reuse data and instructions they have used recently. A widely held rule of thumb is that a program spends 90% of its execution time in only 10% of the code. An implication of locality is that we can predict with reasonable accuracy what instructions and data a program will use in the near future based on its accesses in the recent past. The principle of locality also applies to data accesses, though not as strongly as to code accesses.

Two different types of locality have been observed. *Temporal locality* states that recently accessed items are likely to be accessed in the near future. *Spatial locality* says that items whose addresses are near one another tend to be referenced close together in time. We will see these principles applied in Chapter 5.

Focus on the Common Case

Perhaps the most important and pervasive principle of computer design is to focus on the common case: In making a design trade-off, favor the frequent case over the infrequent case. This principle applies when determining how to spend resources, since the impact of the improvement is higher if the occurrence is frequent.

Focusing on the common case works for power as well as for resource allocation and performance. The instruction fetch and decode unit of a processor may be used much more frequently than a multiplier, so optimize it first. It works on dependability as well. If a database server has 50 disks for every processor, as in the next section, storage dependability will dominate system dependability.

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 processor, 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 computer 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 computer with the enhancement as opposed to the original computer.

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 computer 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, say, 2 seconds for a portion of the program, while it is 5 seconds in the original mode, 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 computer with the enhanced mode will be the time spent using the unenhanced portion of the computer 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 want to enhance the processor used for Web serving. The new processor is 10 times faster on computation in the Web serving application than the original processor. Assuming that the original processor is busy with computation 40% of the time and is waiting for I/O 60% of the time, what is the overall speedup gained by incorporating the enhancement?

Answer $\text{Fraction}_{\text{enhanced}} = 0.4$, $\text{Speedup}_{\text{enhanced}} = 10$, $\text{Speedup}_{\text{overall}} = \dfrac{1}{0.6 + \dfrac{0.4}{10}} = \dfrac{1}{0.64} \approx 1.56$

Amdahl's Law expresses the law of diminishing returns: The incremental improvement in speedup gained by an improvement 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!

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

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

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

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

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

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

Amdahl's Law is applicable beyond performance. Let's redo the reliability example from page 27 after improving the reliability of the power supply via redundancy from 200,000-hour to 830,000,000-hour MTTF, or 4150X better.

Example The calculation of the failure rates of the disk subsystem was

$$\text{Failure rate}_{\text{system}} = 10 \times \frac{1}{1{,}000{,}000} + \frac{1}{500{,}000} + \frac{1}{200{,}000} + \frac{1}{200{,}000} + \frac{1}{1{,}000{,}000}$$

$$= \frac{10 + 2 + 5 + 5 + 1}{1{,}000{,}000 \text{ hours}} = \frac{23}{1{,}000{,}000 \text{ hours}}$$

Therefore, the fraction of the failure rate that could be improved is 5 per million hours out of 23 for the whole system, or 0.22.

Answer The reliability improvement would be

$$\text{Improvement}_{\text{power supply pair}} = \frac{1}{(1 - 0.22) + \frac{0.22}{4150}} = \frac{1}{0.78} = 1.28$$

Despite an impressive 4150X improvement in reliability of one module, from the system's perspective, the change has a measurable but small benefit.

In the examples above we needed the fraction consumed by the new and improved version; 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. Furthermore, it is often possible to build simulators that measure these components before the hardware is actually designed.

The Processor Performance Equation

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

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

or

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

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

CPI is computed as

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

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

By transposing instruction count in the above formula, clock cycles can be defined as IC × CPI. This allows us to use CPI in the execution time formula:

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

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

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

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

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

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

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

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

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

and overall CPI as

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

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

Consider our performance example on page 40, here modified to use measurements of the frequency of the instructions and of the instruction CPI values, which, in practice, are obtained by simulation or by hardware instrumentation.

Example Suppose we have made the following measurements:

Frequency of FP operations = 25%

Average CPI of FP operations = 4.0

Average CPI of other instructions = 1.33

Frequency of FPSQR= 2%

CPI of FPSQR = 20

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

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

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

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

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

$$\text{CPI}_{\text{new FP}} = (75\% \times 1.33) + (25\% \times 2.5) = 1.62$$

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

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

Happily, we obtained this same speedup using Amdahl's Law on page 40.

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

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

1.10 Putting It All Together: Performance and Price-Performance

In the "Putting It All Together" sections that appear near the end of every chapter, we show real examples that use the principles in that chapter. In this section, we

look at measures of performance and price-performance, in desktop systems using the SPEC benchmark and then in servers using the TPC-C benchmark.

Performance and Price-Performance for Desktop and Rack-Mountable Systems

Although there are many benchmark suites for desktop systems, a majority of them are OS or architecture specific. In this section we examine the processor performance and price-performance of a variety of desktop systems using the SPEC CPU2000 integer and floating-point suites. As mentioned in Figure 1.14, SPEC CPU2000 summarizes processor performance using a geometric mean normalized to a Sun Ultra 5, with larger numbers indicating higher performance.

Figure 1.15 shows the five systems including the processors and price. Each system was configured with one processor, 1 GB of DDR DRAM (with ECC if available), approximately 80 GB of disk, and an Ethernet connection. The desktop systems come with a fast graphics card and a monitor, while the rack-mountable systems do not. The wide variation in price is driven by a number of factors, including the cost of the processor, software differences (Linux or a Microsoft OS versus a vendor-specific OS), system expandability, and the commoditization effect, which we discussed in Section 1.6.

Figure 1.16 shows the performance and the price-performance of these five systems using SPEC CINT2000base and CFP2000base as the metrics. The figure also plots price-performance on the right axis, showing CINT or CFP per $1000 of price. Note that in every case, floating-point performance exceeds integer performance relative to the base computer.

Vendor/model	Processor	Clock rate	L2 cache	Type	Price
Dell Precision Workstation 380	Intel Pentium 4 Xeon	3.8 GHz	2 MB	Desk	$3346
HP ProLiant BL25p	AMD Opteron 252	2.6 GHz	1 MB	Rack	$3099
HP ProLiant ML350 G4	Intel Pentium 4 Xeon	3.4 GHz	1 MB	Desk	$2907
HP Integrity rx2620-2	Itanium 2	1.6 GHz	3 MB	Rack	$5201
Sun Java Workstation W1100z	AMD Opteron 150	2.4 GHz	1 MB	Desk	$2145

Figure 1.15 Five different desktop and rack-mountable systems from three vendors using three different microprocessors showing the processor, its clock rate, L2 cache size, and the selling price. Figure 1.16 plots absolute performance and price performance. All these systems are configured with 1 GB of ECC SDRAM and approximately 80 GB of disk. (If software costs were not included, we added them.) Many factors are responsible for the wide variation in price despite these common elements. First, the systems offer different levels of expandability (with the Sun Java Workstation being the least expandable, the Dell systems being moderately expandable, and the HP BL25p blade server being the most expandable). Second, the cost of the processor varies by at least a factor of 2, with much of the reason for the higher costs being the size of the L2 cache and the larger die. In 2005, the Opteron sold for about $500 to $800 and Pentium 4 Xeon sold for about $400 to $700, depending on clock rates and cache size. The Itanium 2 die size is much larger than the others, so it's probably at least twice the cost. Third, software differences (Linux or a Microsoft OS versus a vendor-specific OS) probably affect the final price. These prices were as of August 2005.

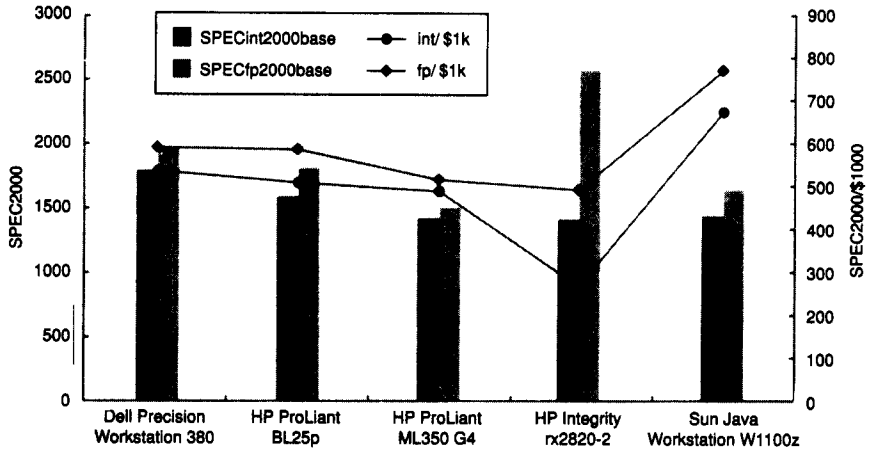

Figure 1.16 Performance and price-performance for five systems in Figure 1.15 measured using SPEC CINT2000 and CFP2000 as the benchmark. Price-performance is plotted as CINT2000 and CFP2000 performance per $1000 in system cost. These performance numbers were collected in January 2006 and prices were as of August 2005. The measurements are available online at *www.spec.org*.

The Itanium 2–based design has the highest floating-point performance but also the highest cost, and hence has the lowest performance per thousand dollars, being off a factor of 1.1–1.6 in floating-point and 1.8–2.5 in integer performance. While the Dell based on the 3.8 GHz Intel Xeon with a 2 MB L2 cache has the high performance for CINT and second highest for CFP, it also has a much higher cost than the Sun product based on the 2.4 GHz AMD Opteron with a 1 MB L2 cache, making the latter the price-performance leader for CINT and CFP.

Performance and Price-Performance for Transaction-Processing Servers

One of the largest server markets is online transaction processing (OLTP). The standard industry benchmark for OLTP is TPC-C, which relies on a database system to perform queries and updates. Five factors make the performance of TPC-C particularly interesting. First, TPC-C is a reasonable approximation to a real OLTP application. Although this is complex and time-consuming, it makes the results reasonably indicative of real performance for OLTP. Second, TPC-C measures total system performance, including the hardware, the operating system, the I/O system, and the database system, making the benchmark more predictive of real performance. Third, the rules for running the benchmark and reporting execution time are very complete, resulting in numbers that are more comparable. Fourth, because of the importance of the benchmark, computer system vendors devote significant effort to making TPC-C run well. Fifth, vendors are required to

report both performance and price-performance, enabling us to examine both. For TPC-C, performance is measured in transactions per minute (TPM), while price-performance is measured in dollars per TPM.

Figure 1.17 shows the characteristics of 10 systems whose performance or price-performance is near the top in one measure or the other. Figure 1.18 plots absolute performance on a log scale and price-performance on a linear scale. The number of disks is determined by the number of I/Os per second to match the performance target rather than the storage capacity need to run the benchmark.

The highest-performing system is a 64-node shared-memory multiprocessor from IBM, costing a whopping $17 million. It is about twice as expensive and twice as fast as the same model half its size, and almost three times faster than the third-place cluster from HP. These five computers average 35–50 disks per processor and 16–20 GB of DRAM per processor. Chapter 4 discusses the design of multiprocessor systems, and Chapter 6 and Appendix E describe clusters.

The computers with the best price-performance are all uniprocessors based on Pentium 4 Xeon processors, although the L2 cache size varies. Notice that these systems have about three to four times better price-performance than the

Vendor and system	Processors	Memory	Storage	Database/OS	Price
IBM eServer p5 595	64 IBM POWER 5 @1.9 GHz, 36 MB L3	64 cards, 2048 GB	6548 disks 243,236 GB	IBM DB2 UDB 8.2/ IBM AIX 5L V5.3	$16,669,230
IBM eServer p5 595	32 IBM POWER 5 @1.9 GHz, 36 MB L3	32 cards, 1024 GB	3298 disks 112,885 GB	Orcale 10g EE/ IBM AIX 5L V5.3	$8,428,470
HP Integrity rx5670 Cluster	64 Intel Itanium 2 @ 1.5 GHz, 6 MB L3	768 dimms, 768 GB	2195 disks, 93,184 GB	Orcale 10g EE/ Red Hat E Linux AS 3	$6,541,770
HP Integrity Superdome	64 Intel Itanium 2 @ 1.6 GHz, 9 MB L3	512 dimms, 1024 GB	1740 disks, 53,743 GB	MS SQL Server 2005 EE/MS Windows DE 64b	$5,820,285
IBM eServer pSeries 690	32 IBM POWER4+ @ 1.9 GHz, 128 MB L3	4 cards, 1024 GB	1995 disks, 74,098 GB	IBM DB2 UDB 8.1/ IBM AIX 5L V5.2	$5,571,349
Dell PowerEdge 2800	1 Intel Xeon @ 3.4 GHz, 2MB L2	2 dimms, 2.5 GB	76 disks, 2585 GB	MS SQL Server 2000 WE/ MS Windows 2003	$39,340
Dell PowerEdge 2850	1 Intel Xeon @ 3.4 GHz, 1MB L2	2 dimms, 2.5 GB	76 disks, 1400 GB	MS SQL Server 2000 SE/ MS Windows 2003	$40,170
HP ProLiant ML350	1 Intel Xeon @ 3.1 GHz, 0.5MB L2	3 dimms, 2.5 GB	34 disks, 696 GB	MS SQL Server 2000 SE/ MS Windows 2003 SE	$27,827
HP ProLiant ML350	1 Intel Xeon @ 3.1 GHz, 0.5MB L2	4 dimms, 4 GB	35 disks, 692 GB	IBM DB2 UDB EE V8.1/ SUSE Linux ES 9	$29,990
HP ProLiant ML350	1 Intel Xeon @ 2.8 GHz, 0.5MB L2	4 dimms, 3.25 GB	35 disks, 692 GB	IBM DB2 UDB EE V8.1/ MS Windows 2003 SE	$30,600

Figure 1.17 The characteristics of 10 OLTP systems, using TPC-C as the benchmark, with either high total performance (top half of the table, measured in transactions per minute) or superior price-performance (bottom half of the table, measured in U.S. dollars per transactions per minute). Figure 1.18 plots absolute performance and price performance, and Figure 1.19 splits the price between processors, memory, storage, and software.

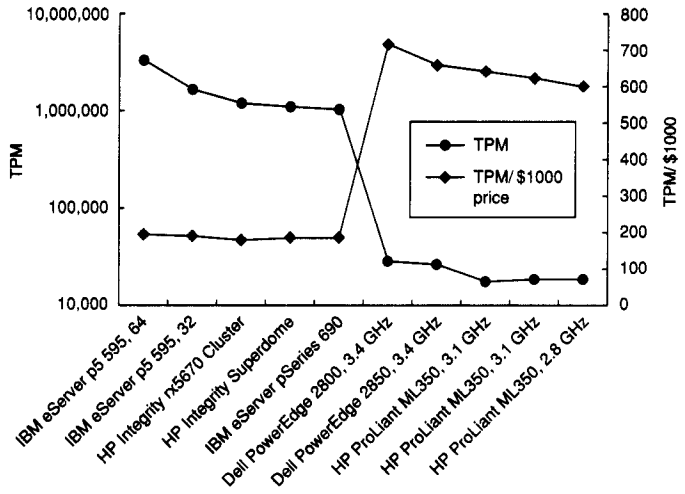

Figure 1.18 Performance and price-performance for the 10 systems in Figure 1.17 using TPC-C as the benchmark. Price-performance is plotted as TPM per $1000 in system cost, although the conventional TPC-C measure is $/TPM (715 TPM/$1000 = $1.40 $/TPM). These performance numbers and prices were as of July 2005. The measurements are available online at *www.tpc.org*.

high-performance systems. Although these five computers also average 35–50 disks per processor, they only use 2.5–3 GB of DRAM per processor. It is hard to tell whether this is the best choice or whether it simply reflects the 32-bit address space of these less expensive PC servers. Since doubling memory would only add about 4% to their price, it is likely the latter reason.

1.11 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 computers that you design.

Pitfall *Falling prey to Amdahl's Law.*

Virtually every practicing computer architect knows Amdahl's Law. Despite this, we almost all occasionally expend tremendous effort optimizing some feature before we measure its usage. Only when the overall speedup is disappointing do we recall that we should have measured first before we spent so much effort enhancing it!

Pitfall *A single point of failure.*

The calculations of reliability improvement using Amdahl's Law on page 41 show that dependability is no stronger than the weakest link in a chain. No matter how much more dependable we make the power supplies, as we did in our example, the single fan will limit the reliability of the disk subsystem. This Amdahl's Law observation led to a rule of thumb for fault-tolerant systems to make sure that every component was redundant so that no single component failure could bring down the whole system.

Fallacy *The cost of the processor dominates the cost of the system.*

Computer science is processor centric, perhaps because processors seem more intellectually interesting than memories or disks and perhaps because algorithms are traditionally measured in number of processor operations. This fascination leads us to think that processor utilization is the most important figure of merit. Indeed, the high-performance computing community often evaluates algorithms and architectures by what fraction of peak processor performance is achieved. This would make sense if most of the cost were in the processors.

Figure 1.19 shows the breakdown of costs for the computers in Figure 1.17 into the processor (including the cabinets, power supplies, and so on), DRAM

	Processor + cabinetry	Memory	Storage	Software
IBM eServer p5 595	28%	16%	51%	6%
IBM eServer p5 595	13%	31%	52%	4%
HP Integrity rx5670 Cluster	11%	22%	35%	33%
HP Integrity Superdome	33%	32%	15%	20%
IBM eServer pSeries 690	21%	24%	48%	7%
Median of high-performance computers	21%	24%	48%	7%
Dell PowerEdge 2800	6%	3%	80%	11%
Dell PowerEdge 2850	7%	3%	76%	14%
HP ProLiant ML350	5%	4%	70%	21%
HP ProLiant ML350	9%	8%	65%	19%
HP ProLiant ML350	8%	6%	65%	21%
Median of price-performance computers	7%	4%	70%	19%

Figure 1.19 Cost of purchase split between processor, memory, storage, and software for the top computers running the TPC-C benchmark in Figure 1.17. Memory is just the cost of the DRAM modules, so all the power and cooling for the computer is credited to the processor. TPC-C includes the cost of the clients to drive the TPC-C benchmark and the three-year cost of maintenance, which are not included here. Maintenance would add about 10% to the numbers here, with differences in software maintenance costs making the range be 5% to 22%. Including client hardware would add about 2% to the price of the high-performance servers and 7% to the PC servers.

memory, disk storage, and software. Even giving the processor category the credit for the sheet metal, power supplies, and cooling, it's only about 20% of the costs for the large-scale servers and less than 10% of the costs for the PC servers.

Fallacy *Benchmarks remain valid indefinitely.*

Several factors influence the usefulness of a benchmark as a predictor of real performance, and some change over time. A big factor influencing the usefulness of a benchmark is its ability 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]). When an IBM compiler optimized this inner loop (using an idea called blocking, discussed in Chapter 5), performance improved by a factor of 9 over a prior version of the compiler! This benchmark tested compiler tuning and was not, of course, a good indication of overall performance, nor of the typical value of this particular optimization.

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

Over a long period, these changes may make even a well-chosen benchmark obsolete; Gcc is the lone survivor from SPEC89. Figure 1.13 on page 31 lists the status of all 70 benchmarks from the various SPEC releases. Amazingly, almost 70% of all programs from SPEC2000 or earlier were dropped from the next release.

Fallacy *The rated mean time to failure of disks is 1,200,000 hours or almost 140 years, so disks practically never fail.*

The current marketing practices of disk manufacturers can mislead users. How is such an MTTF calculated? Early in the process, manufacturers will put thousands of disks in a room, run them for a few months, and count the number that fail. They compute MTTF as the total number of hours that the disks worked cumulatively divided by the number that failed.

One problem is that this number far exceeds the lifetime of a disk, which is commonly assumed to be 5 years or 43,800 hours. For this large MTTF to make some sense, disk manufacturers argue that the model corresponds to a user who buys a disk, and then keeps replacing the disk every 5 years—the planned lifetime of the disk. The claim is that if many customers (and their great-

grandchildren) did this for the next century, on average they would replace a disk 27 times before a failure, or about 140 years.

A more useful measure would be percentage of disks that fail. Assume 1000 disks with a 1,000,000-hour MTTF and that the disks are used 24 hours a day. If you replaced failed disks with a new one having the same reliability characteristics, the number that would fail in a year (8760 hours) is

$$\text{Failed disks} = \frac{\text{Number of disks} \times \text{Time period}}{\text{MTTF}} = \frac{1000 \text{ disks} \times 8760 \text{ hours/drive}}{1{,}000{,}000 \text{ hours/failure}} = 9$$

Stated alternatively, 0.9% would fail per year, or 4.4% over a 5-year lifetime.

Moreover, those high numbers are quoted assuming limited ranges of temperature and vibration; if they are exceeded, then all bets are off. A recent survey of disk drives in real environments [Gray and van Ingen 2005] claims about 3–6% of SCSI drives fail per year, or an MTTF of about 150,000–300,000 hours, and about 3–7% of ATA drives fail per year, or an MTTF of about 125,000–300,000 hours. The quoted MTTF of ATA disks is usually 500,000–600,000 hours. Hence, according to this report, real-world MTTF is about 2–4 times worse than manufacturer's MTTF for ATA disks and 4–8 times worse for SCSI disks.

Fallacy *Peak performance tracks observed performance.*

The only universally true definition of peak performance is "the performance level a computer is guaranteed not to exceed." Figure 1.20 shows the percentage of peak performance for four programs on four multiprocessors. It varies from 5% to 58%. Since the gap is so large and can vary significantly by benchmark, peak performance is not generally useful in predicting observed performance.

Pitfall *Fault detection can lower availability.*

This apparently ironic pitfall is because computer hardware has a fair amount of state that may not always be critical to proper operation. For example, it is not fatal if an error occurs in a branch predictor, as only performance may suffer.

In processors that try to aggressively exploit instruction-level parallelism, not all the operations are needed for correct execution of the program. Mukherjee et al. [2003] found that less than 30% of the operations were potentially on the critical path for the SPEC2000 benchmarks running on an Itanium 2.

The same observation is true about programs. If a register is "dead" in a program—that is, the program will write it before it is read again—then errors do not matter. If you were to crash the program upon detection of a transient fault in a dead register, it would lower availability unnecessarily.

Sun Microsystems lived this pitfall in 2000 with an L2 cache that included parity, but not error correction, in its Sun E3000 to Sun E10000 systems. The SRAMs they used to build the caches had intermittent faults, which parity detected. If the data in the cache was not modified, the processor simply reread the data from the cache. Since the designers did not protect the cache with ECC, the operating system had no choice but report an error to dirty data and crash the

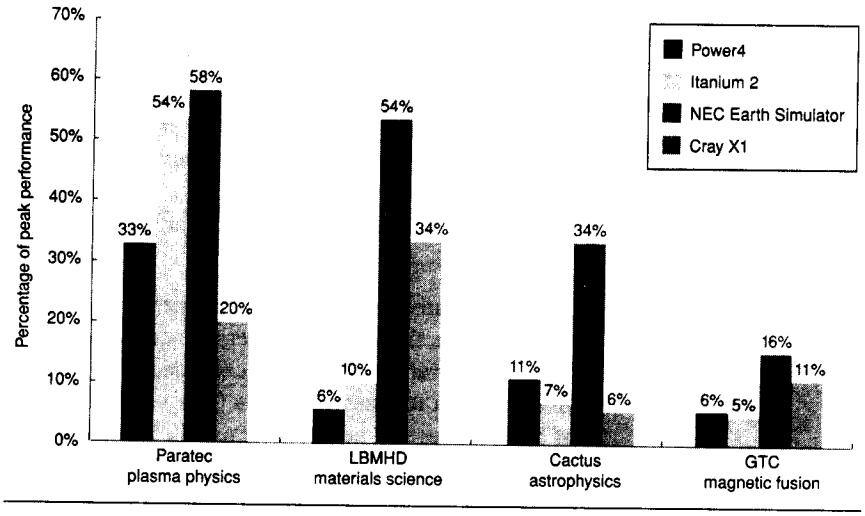

Figure 1.20 Percentage of peak performance for four programs on four multiprocessors scaled to 64 processors. The Earth Simulator and X1 are vector processors. (See Appendix F.) Not only did they deliver a higher fraction of peak performance, they had the highest peak performance and the lowest clock rates. Except for the Paratec program, the Power 4 and Itanium 2 systems deliver between 5% and 10% of their peak. From Oliker et al. [2004].

program. Field engineers found no problems on inspection in more than 90% of the cases.

To reduce the frequency of such errors, Sun modified the Solaris operating system to "scrub" the cache by having a process that proactively writes dirty data to memory. Since the processor chips did not have enough pins to add ECC, the only hardware option for dirty data was to duplicate the external cache, using the copy without the parity error to correct the error.

The pitfall is in detecting faults without providing a mechanism to correct them. Sun is unlikely to ship another computer without ECC on external caches.

1.12 Concluding Remarks

This chapter has introduced a number of concepts that we will expand upon as we go through this book.

In Chapters 2 and 3, we look at instruction-level parallelism (ILP), of which pipelining is the simplest and most common form. Exploiting ILP is one of the most important techniques for building high-speed uniprocessors. The presence of two chapters reflects the fact that there are several approaches to exploiting ILP and that it is an important and mature technology. Chapter 2 begins with an extensive discussion of basic concepts that will prepare you for the wide range of

ideas examined in both chapters. Chapter 2 uses examples that span about 35 years, drawing from one of the first supercomputers (IBM 360/91) to the fastest processors in the market in 2006. It emphasizes what is called the dynamic or run time approach to exploiting ILP. Chapter 3 focuses on limits and extensions to the ILP ideas presented in Chapter 2, including multithreading to get more from an out-of-order organization. Appendix A is introductory material on pipelining for readers without much experience and background in pipelining. (We expect it to be review for many readers, including those of our introductory text, *Computer Organization and Design: The Hardware/Software Interface.*)

Chapter 4 focuses on the issue of achieving higher performance using multiple processors, or multiprocessors. Instead of using parallelism to overlap individual instructions, multiprocessing uses parallelism to allow multiple instruction streams to be executed simultaneously on different processors. Our focus is on the dominant form of multiprocessors, shared-memory multiprocessors, though we introduce other types as well and discuss the broad issues that arise in any multiprocessor. Here again, we explore a variety of techniques, focusing on the important ideas first introduced in the 1980s and 1990s.

In Chapter 5, we turn to the all-important area of memory system design. We will examine a wide range of techniques that conspire to make memory look infinitely large while still being as fast as possible. As in Chapters 2 through 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. This chapter also covers virtual machines. Appendix C is introductory material on caches for readers without much experience and background in them.

In Chapter 6, we move away from a processor-centric view and discuss issues in storage systems. We apply a similar quantitative approach, but one based on observations of system behavior and using an end-to-end approach to performance analysis. It addresses the important issue of how to efficiently store and retrieve data using primarily lower-cost magnetic storage technologies. Such technologies offer better cost per bit by a factor of 50–100 over DRAM. In Chapter 6, our focus is on examining the performance of disk storage systems for typical I/O-intensive workloads, like the OLTP benchmarks we saw in this chapter. We extensively explore advanced topics in RAID-based systems, which use redundant disks to achieve both high performance and high availability. Finally, the chapter introduces queing theory, which gives a basis for trading off utilization and latency.

This book comes with a plethora of material on the companion CD, both to lower cost and to introduce readers to a variety of advanced topics. Figure 1.21 shows them all. Appendices A, B, and C, which appear in the book, will be review for many readers. Appendix D takes the embedded computing perspective on the ideas of each of the chapters and early appendices. Appendix E explores the topic of system interconnect broadly, including wide area and system area networks used to allow computers to communicate. It also describes clusters, which are growing in importance due to their suitability and efficiency for database and Web server applications.

Appendix	Title
A	Pipelining: Basic and Intermediate Concepts
B	Instruction Set Principles and Examples
C	Review of Memory Hierarchies
D	Embedded Systems (CD)
E	Interconnection Networks (CD)
F	Vector Processors (CD)
G	Hardware and Software for VLIW and EPIC (CD)
H	Large-Scale Multiprocessors and Scientific Applications (CD)
I	Computer Arithmetic (CD)
J	Survey of Instruction Set Architectures (CD)
K	Historical Perspectives and References (CD)
L	Solutions to Case Study Exercises (Online)

Figure 1.21 List of appendices.

Appendix F explores vector processors, which have become more popular since the last edition due in part to the NEC Global Climate Simulator being the world's fastest computer for several years. Appendix G reviews VLIW hardware and software, which in contrast, are less popular than when EPIC appeared on the scene just before the last edition. Appendix H describes large-scale multiprocessors for use in high performance computing. Appendix I is the only appendix that remains from the first edition, and it covers computer arithmetic. Appendix J is a survey of instruction architectures, including the 80x86, the IBM 360, the VAX, and many RISC architectures, including ARM, MIPS, Power, and SPARC. We describe Appendix K below. Appendix L has solutions to Case Study exercises.

1.13 Historical Perspectives and References

Appendix K on the companion CD includes historical perspectives on the key ideas presented in each of the chapters in this text. These historical perspective sections allow us to 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 each history. For this chapter, see Section K.2, The Early Development of Computers, for a discussion on the early development of digital computers and performance measurement methodologies.

As you read the historical material, you'll soon come to realize that one of the important benefits of the youth of computing, compared to many other engineering fields, is that many of the pioneers are still alive—we can learn the history by simply asking them!

Case Studies with Exercises by Diana Franklin

Case Study 1: Chip Fabrication Cost

Concepts illustrated by this case study

- Fabrication Cost
- Fabrication Yield
- Defect Tolerance through Redundancy

There are many factors involved in the price of a computer chip. New, smaller technology gives a boost in performance and a drop in required chip area. In the smaller technology, one can either keep the small area or place more hardware on the chip in order to get more functionality. In this case study, we explore how different design decisions involving fabrication technology, area, and redundancy affect the cost of chips.

1.1 [10/10/Discussion] <1.5, 1.5> Figure 1.22 gives the relevant chip statistics that influence the cost of several current chips. In the next few exercises, you will be exploring the trade-offs involved between the AMD Opteron, a single-chip processor, and the Sun Niagara, an 8-core chip.

a. [10] <1.5> What is the yield for the AMD Opteron?

b. [10] <1.5> What is the yield for an 8-core Sun Niagara processor?

c. [Discussion] <1.4, 1.6> Why does the Sun Niagara have a worse yield than the AMD Opteron, even though they have the same defect rate?

1.2 [20/20/20/20/20] <1.7> You are trying to figure out whether to build a new fabrication facility for your IBM Power5 chips. It costs $1 billion to build a new fabrication facility. The benefit of the new fabrication is that you predict that you will be able to sell 3 times as many chips at 2 times the price of the old chips. The new chip will have an area of 186 mm^2, with a defect rate of .7 defects per cm^2. Assume the wafer has a diameter of 300 mm. Assume it costs $500 to fabricate a wafer in either technology. You were previously selling the chips for 40% more than their cost.

Chip	Die size (mm^2)	Estimated defect rate (per cm^2)	Manufacturing size (nm)	Transistors (millions)
IBM Power5	389	.30	130	276
Sun Niagara	380	.75	90	279
AMD Opteron	199	.75	90	233

Figure 1.22 Manufacturing cost factors for several modern processors. $\alpha = 4$.

a. [20] <1.5> What is the cost of the old Power5 chip?

b. [20] <1.5> What is the cost of the new Power5 chip?

c. [20] <1.5> What was the profit on each old Power5 chip?

d. [20] <1.5> What is the profit on each new Power5 chip?

e. [20] <1.5> If you sold 500,000 old Power5 chips per month, how long will it take to recoup the costs of the new fabrication facility?

1.3 [20/20/10/10/20] <1.7> Your colleague at Sun suggests that, since the yield is so poor, it might make sense to sell two sets of chips, one with 8 working processors and one with 6 working processors. We will solve this exercise by viewing the yield as a probability of no defects occurring in a certain area given the defect rate. For the Niagara, calculate probabilities based on each Niagara core separately (this may not be entirely accurate, since the yield equation is based on empirical evidence rather than a mathematical calculation relating the probabilities of finding errors in different portions of the chip).

a. [20] <1.7> Using the yield equation for the defect rate above, what is the probability that a defect will occur on a single Niagara core (assuming the chip is divided evenly between the cores) in an 8-core chip?

b. [20] <1.7> What is the probability that a defect will occur on one or two cores (but not more than that)?

c. [10] <1.7> What is the probability that a defect will occur on none of the cores?

d. [10] <1.7> Given your answers to parts (b) and (c), what is the number of 6-core chips you will sell for every 8-core chip?

e. [20] <1.7> If you sell your 8-core chips for $150 each, the 6-core chips for $100 each, the cost per die sold is $80, your research and development budget was $200 million, and testing itself costs $1.50 per chip, how many processors would you need to sell in order to recoup costs?

Case Study 2: Power Consumption in Computer Systems

Concepts illustrated by this case study

- Amdahl's Law
- Redundancy
- MTTF
- Power Consumption

Power consumption in modern systems is dependent on a variety of factors, including the chip clock frequency, efficiency, the disk drive speed, disk drive utilization, and DRAM. The following exercises explore the impact on power that different design decisions and/or use scenarios have.

Component type	Product	Performance	Power
Processor	Sun Niagara 8-core	1.2 GHz	72-79W peak
	Intel Pentium 4	2 GHz	48.9-66W
DRAM	Kingston X64C3AD2 1 GB	184-pin	3.7W
	Kingston D2N3 1 GB	240-pin	2.3W
Hard drive	DiamondMax 16	5400 rpm	7.0W read/seek, 2.9 W idle
	DiamondMax Plus 9	7200 rpm	7.9W read/seek, 4.0 W idle

Figure 1.23 Power consumption of several computer components.

1.4 [20/10/20] <1.6> Figure 1.23 presents the power consumption of several computer system components. In this exercise, we will explore how the hard drive affects power consumption for the system.

a. [20] <1.6> Assuming the maximum load for each component, and a power supply efficiency of 70%, what wattage must the server's power supply deliver to a system with a Sun Niagara 8-core chip, 2 GB 184-pin Kingston DRAM, and two 7200 rpm hard drives?

b. [10] <1.6> How much power will the 7200 rpm disk drive consume if it is idle rougly 40% of the time?

c. [20] <1.6> Assume that rpm is the only factor in how long a disk is not idle (which is an oversimplification of disk performance). In other words, assume that for the same set of requests, a 5400 rpm disk will require twice as much time to read data as a 10,800 rpm disk. What percentage of the time would the 5400 rpm disk drive be idle to perform the same transactions as in part (b)?

1.5 [10/10/20] <1.6, 1.7> One critical factor in powering a server farm is cooling. If heat is not removed from the computer efficiently, the fans will blow hot air back onto the computer, not cold air. We will look at how different design decisions affect the necessary cooling, and thus the price, of a system. Use Figure 1.23 for your power calculations.

a. [10] <1.6> A cooling door for a rack costs $4000 and dissipates 14 KW (into the room; additional cost is required to get it out of the room). How many servers with a Sun Niagara 8-core processor, 1 GB 240-pin DRAM, and a single 5400 rpm hard drive can you cool with one cooling door?

b. [10] <1.6, 1.8> You are considering providing fault tolerance for your hard drive. RAID 1 doubles the number of disks (see Chapter 6). Now how many systems can you place on a single rack with a single cooler?

c. [20] <1.8> In a single rack, the MTTF of each processor is 4500 hours, of the hard drive is 9 million hours, and of the power supply is 30K hours. For a rack with 8 processors, what is the MTTF for the rack?

	Sun Fire T2000	IBM x346
Power (watts)	298	438
SPECjbb (op/s)	63,378	39,985
Power (watts)	330	438
SPECWeb (composite)	14,001	4,348

Figure 1.24 Sun power / performance comparison as selectively reported by Sun.

1.6 [10/10/Discussion] <1.2, 1.9> Figure 1.24 gives a comparison of power and performance for several benchmarks comparing two servers: Sun Fire T2000 (which uses Niagara) and IBM x346 (using Intel Xeon processors).

a. [10] <1.9> Calculate the performance/power ratio for each processor on each benchmark.

b. [10] <1.9> If power is your main concern, which would you choose?

c. [Discussion] <1.2> For the database benchmarks, the cheaper the system, the lower cost *per database operation* the system is. This is counterintuitive: larger systems have more throughput, so one might think that buying a larger system would be a larger absolute cost, but lower per operation cost. Since this is true, why do any larger server farms buy expensive servers? (*Hint:* Look at exercise 1.4 for some reasons.)

1.7 [10/20/20/20] <1.7, 1.10> Your company's internal studies show that a single-core system is sufficient for the demand on your processing power. You are exploring, however, whether you could save power by using two cores.

a. [10] <1.10> Assume your application is 100% parallelizable. By how much could you decrease the frequency and get the same performance?

b. [20] <1.7> Assume that the voltage may be decreased linearly with the frequency. Using the equation in Section 1.5, how much dynamic power would the dual-core system require as compared to the single-core system?

c. [20] <1.7, 1.10> Now assume the voltage may not decrease below 30% of the original voltage. This voltage is referred to as the "voltage floor," and any voltage lower than that will lose the state. What percent of parallelization gives you a voltage at the voltage floor?

d. [20] <1.7, 1.10> Using the equation in Section 1.5, how much dynamic power would the dual-core system require from part (a) compared to the single-core system when taking into account the voltage floor?

Case Study 3: The Cost of Reliability (and Failure) in Web Servers

Concepts illustrated by this case study

- TPCC
- Reliability of Web Servers
- MTTF

This set of exercises deals with the cost of not having reliable Web servers. The data is in two sets: one gives various statistics for Gap.com, which was down for maintenance for two weeks in 2005 [AP 2005]. The other is for Amazon.com, which was not down, but has better statistics on high-load sales days. The exercises combine the two data sets and require estimating the economic cost to the shutdown.

1.8 [10/10/20/20] <1.2, 1.9> On August 24, 2005, three Web sites managed by the Gap—Gap.com, OldNavy.com, and BananaRepublic.com—were taken down for improvements [AP 2005]. These sites were virtually inaccessible for the next two weeks. Using the statistics in Figure 1.25, answer the following questions, which are based in part on hypothetical assumptions.

a. [10] <1.2> In the third quarter of 2005, the Gap's revenue was $3.9 billion [Gap 2005]. The Web site returned live on September 7, 2005 [Internet Retailer 2005]. Assume that online sales total $1.4 million per day, and that everything else remains constant. What would the Gap's estimated revenue be third quarter 2005?

b. [10] <1.2> If this downtime occurred in the fourth quarter, what would you estimate the cost of the downtime to be?

Company	Time period	Amount	Type
Gap	3rd qtr 2004	$4 billion	Sales
	4th qtr 2004	$4.9 billion	Sales
	3rd qtr 2005	$3.9 billion	Sales
	4th qtr 2005	$4.8 billion	Sales
	3rd qtr 2004	$107 million	Online sales
	3rd qtr 2005	$106 million	Online sales
Amazon	3rd qtr 2005	$1.86 billion	Sales
	4th qtr 2005	$2.98 billion	Sales
	4th qtr 2005	108 million	Items sold
	Dec 12, 2005	3.6 million	Items sold

Figure 1.25 Statistics on sales for Gap and Amazon. Data compiled from AP [2005], Internet Retailer [2005], Gamasutra [2005], Seattle PI [2005], MSN Money [2005], Gap [2005], and Gap [2006].

c. [20] <1.2> When the site returned, the number of users allowed to visit the site at one time was limited. Imagine that it was limited to 50% of the customers who wanted to access the site. Assume that each server costs $7500 to purchase and set up. How many servers, per day, could they purchase and install with the money they are losing in sales?

d. [20] <1.2, 1.9> Gap.com had 2.6 million visitors in July 2004 [AP 2005]. On average, a user views 8.4 pages per day on Gap.com. Assume that the high-end servers at Gap.com are running SQLServer software, with a TPCC benchmark estimated cost of $5.38 per transaction. How much would it cost for them to support their online traffic at Gap.com.?

1.9 [10/10] <1.8> The main reliability measure is MTTF. We will now look at different systems and how design decisions affect their reliability. Refer to Figure 1.25 for company statistics.

a. [10] <1.8> We have a single processor with an FIT of 100. What is the MTTF for this system?

b. [10] <1.8> If it takes 1 day to get the system running again, what is the availability of the system?

1.10 [20] <1.8> Imagine that the government, to cut costs, is going to build a supercomputer out of the cheap processor system in Exercise 1.9 rather than a special-purpose reliable system. What is the MTTF for a system with 1000 processors? Assume that if one fails, they all fail.

1.11 [20/20] <1.2, 1.8> In a server farm such as that used by Amazon or the Gap, a single failure does not cause the whole system to crash. Instead, it will reduce the number of requests that can be satisfied at any one time.

a. [20] <1.8> If a company has 10,000 computers, and it experiences catastrophic failure only if 1/3 of the computers fail, what is the MTTF for the system?

b. [20] <1.2, 1.8> If it costs an extra $1000, per computer, to double the MTTF, would this be a good business decision? Show your work.

Case Study 4: Performance

Concepts illustrated by this case study

- Arithmetic Mean
- Geometric Mean
- Parallelism
- Amdahl's Law
- Weighted Averages

In this set of exercises, you are to make sense of Figure 1.26, which presents the performance of selected processors and a fictional one (Processor X), as reported by *www.tomshardware.com*. For each system, two benchmarks were run. One benchmark exercised the memory hierarchy, giving an indication of the speed of the memory for that system. The other benchmark, Dhrystone, is a CPU-intensive benchmark that does not exercise the memory system. Both benchmarks are displayed in order to distill the effects that different design decisions have on memory and CPU performance.

1.12 [10/10/Discussion/10/20/Discussion] <1.7> Make the following calculations on the raw data in order to explore how different measures color the conclusions one can make. (Doing these exercises will be much easier using a spreadsheet.)

a. [10] <1.8> Create a table similar to that shown in Figure 1.26, except express the results as normalized to the Pentium D for each benchmark.

b. [10] <1.9> Calculate the arithmetic mean of the performance of each processor. Use both the original performance and your normalized performance calculated in part (a).

c. [Discussion] <1.9> Given your answer from part (b), can you draw any conflicting conclusions about the relative performance of the different processors?

d. [10] <1.9> Calculate the geometric mean of the normalized performance of the dual processors and the geometric mean of the normalized performance of the single processors for the Dhrystone benchmark.

e. [20] <1.9> Plot a 2D scatter plot with the x-axis being Dhrystone and the y-axis being the memory benchmark.

f. [Discussion] <1.9> Given your plot in part (e), in what area does a dual-processor gain in performance? Explain, given your knowledge of parallel processing and architecture, why these results are as they are.

Chip	# of cores	Clock frequency (MHz)	Memory performance	Dhrystone performance
Athlon 64 X2 4800+	2	2,400	3,423	20,718
Pentium EE 840	2	2,200	3,228	18,893
Pentium D 820	2	3,000	3,000	15,220
Athlon 64 X2 3800+	2	3,200	2,941	17,129
Pentium 4	1	2,800	2,731	7,621
Athlon 64 3000+	1	1,800	2,953	7,628
Pentium 4 570	1	2,800	3,501	11,210
Processor X	1	3,000	7,000	5,000

Figure 1.26 Performance of several processors on two benchmarks.

1.13 [10/10/20] <1.9> Imagine that your company is trying to decide between a single-processor system and a dual-processor system. Figure 1.26 gives the performance on two sets of benchmarks—a memory benchmark and a processor benchmark. You know that your application will spend 40% of its time on memory-centric computations, and 60% of its time on processor-centric computations.

a. [10] <1.9> Calculate the weighted execution time of the benchmarks.

b. [10] <1.9> How much speedup do you anticipate getting if you move from using a Pentium 4 570 to an Athlon 64 X2 4800+ on a CPU-intensive application suite?

c. [20] <1.9> At what ratio of memory to processor computation would the performance of the Pentium 4 570 be equal to the Pentium D 820?

1.14 [10/10/20/20] <1.10> Your company has just bought a new dual Pentium processor, and you have been tasked with optimizing your software for this processor. You will run two applications on this dual Pentium, but the resource requirements are not equal. The first application needs 80% of the resources, and the other only 20% of the resources.

a. [10] <1.10> Given that 40% of the first application is parallelizable, how much speedup would you achieve with that application if run in isolation?

b. [10] <1.10> Given that 99% of the second application is parallelizable, how much speedup would this application observe if run in isolation?

c. [20] <1.10> Given that 40% of the first application is parallelizable, how much *overall system speedup* would you observe if you parallelized it?

d. [20] <1.10> Given that 99% of the second application is parallelizable, how much overall system speedup would you get?