1

Fundamentals of Quantitative Design and Analysis

An iPod, a phone, an Internet mobile communicator… these are NOT three separate devices! And we are calling it iPhone! Today Apple is going to reinvent the phone. And here it is.

Steve Jobs, January 9, 2007

New information and communications technologies, in particular high-speed Internet, are changing the way companies do business, transforming public service delivery, and democratizing innovation. With 10 percent increase in high speed Internet connections, economic growth increases by 1.3 percent.

The World Bank, July 28, 2009

Computer Architecture. https://doi.org/10.1016/B978-0-443-15406-5.00001-8

1.1 Introduction

Computer technology has made incredible progress in the roughly 75 years since the first general-purpose electronic computer was created. Today, $800 will purchase a cell phone that has as much performance as the world's fastest computer bought in 2000 for more than $100 million. This rapid improvement has come from both advances in the technology used to build computers and innovations in computer design.

Although technological improvements historically 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 performance improvement—roughly 35% growth per year.

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 to succeed commercially with a new architecture. First, the virtual elimination of assembly language programming reduced the need for object-code compatibility. Second, the creation of standardized, vendor-independent operating systems, such as UNIX and its clone, Linux, lowered the cost and risk of bringing out a new architecture.

These changes made it possible to successfully develop a new set of architectures 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* (ILP; 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 80x86 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 80x86 architecture became negligible. In low-end applications, such as cell phones, the cost in power and silicon area of the 80x86-translation overhead helped lead to a RISC architecture, ARM, becoming dominant.

Figure 1.1 shows that the combination of architectural and organizational enhancements led to 17 years of sustained growth in performance at an annual rate of over 50%—a rate that is unprecedented in the computer industry.

The effects of this dramatic growth rate during the 20th century was fourfold. First, it has significantly enhanced the capability available to computer users. For

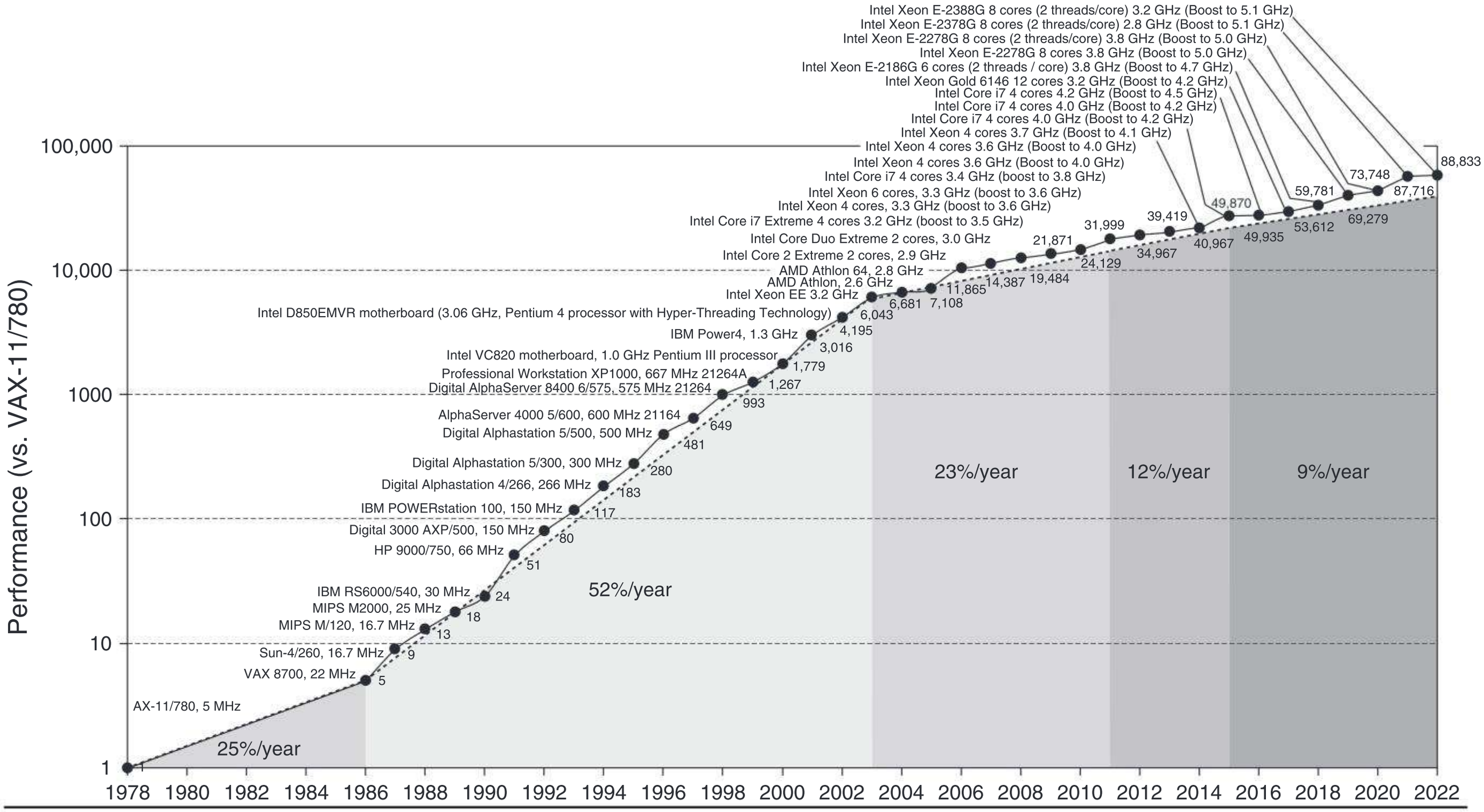

Figure 1.1 Growth in processor performance over 45 years. This chart plots program performance relative to the VAX 11/780 as measured by the SPEC integer base benchmarks (see Section 1.8). Prior to the mid-1980s, growth in processor performance was largely technology driven and averaged about 25% per year, or doubling performance every 3.5 years. The increase in growth to about 52% starting in 1986, or doubling every 2 years, is attributable to more advanced architectural and organizational ideas typified in RISC architectures plus the switch to the microprocessor as the CPU building block. By 2003, this growth led to a difference in performance of 25× versus the performance that would have occurred if it had continued at the 25% rate. If we assume a 35% annual increase due to improvements in semiconductor technology, the difference would be 7.5×. In 2003, the limits of power due to the end of Dennard scaling and the available instruction-level parallelism slowed uniprocessor performance to 23% per year until 2011, or doubling every 3.5 years. These results are limited to single-chip systems with up to 8 cores per chip typically and 1 or 2 hyperthreads per core. The fastest SPECintbase performance since 2007 has had automatic parallelization turned on, so uniprocessor speed is harder to gauge. From 2011 to 2015, the annual improvement was less than 12% (doubling every 6 years) in part due to the limits of parallelism of Amdahl's Law. Since 2015, with the slowing of Moore's Law, improvement has been 9% per year, or doubling every 8 years. Performance for floating-point—oriented calculations follows the same trends but typically has 1% to 2% higher annual growth in each shaded region. (SPEC had six versions from 1989 to 2017, so performance over time is estimated using scaling factors calculated from computers with two SPEC ratings across two generations.)

many applications, the highest-performance microprocessors outperformed the supercomputer of less than 20 years earlier.

Second, this dramatic improvement in cost-performance led to new classes of computers. Personal computers and workstations emerged in the 1980s with the availability of the microprocessor. The past 15 years saw the rise of smart cell phones and tablet computers, which many people are using as their primary computing platforms instead of PCs. These mobile client devices are increasingly using the Internet to access warehouses containing 50,000 servers, which are being designed as if they were a single gigantic computer. Embedded computers with wireless connections, called collectively Internet of Things (IoT) devices, exceed 10 billion units shipped per year today and are projected to grow dramatically.

Third, improvement of semiconductor manufacturing as predicted by Moore's Law has led to the dominance of microprocessor-based computers across the entire range of computer design. Minicomputers, which were traditionally made from off-the-shelf logic or from gate arrays, were replaced by servers made by using microprocessors. Even mainframes and high-performance supercomputers today are all collections of microprocessors.

The preceding hardware innovations led to a renaissance in computer design, which emphasized both architectural innovation and efficient use of technology improvements. This rate of growth compounded so that by 2003, high-performance microprocessors were 7.5 times as fast as what would have been obtained by relying solely on technology, including improved circuit design, that is, 52% per year CAGR versus 35% per year.

This hardware renaissance led to the fourth impact, which was on software development. This 90,000-fold performance improvement since 1978 (see Figure 1.1) allowed modern programmers to trade performance for productivity. In place of performance-oriented languages like C and C++, much more programming today is done in managed programming languages like Java. Moreover, scripting languages like JavaScript and Python, which are even more productive, are gaining in popularity along with programming frameworks like AngularJS and Django. To maintain productivity and try to close the performance gap, interpreters with just-in-time compilers and trace-based compiling are replacing the traditional compiler and linker of the past. Software deployment is changing as well, with Software as a Service (SaaS) used over the Internet replacing shrink-wrapped software that must be installed and run on a local computer.

The nature of applications is also changing. Speech, sound, images, and video are becoming increasingly important, along with predictable response time that is so critical to the user experience. An inspiring example is Google Translate. This application lets you hold up your cell phone to point its camera at an object, and the image is sent wirelessly over the Internet to a warehouse-scale computer (WSC) that recognizes the text in the photo and translates it into your native language. You can also speak into it, and it will translate what you said into audio output

in another language. It translates text into 133 languages, proposes translations via photo in 37 languages, and 32 languages via voice in conversation mode.

Alas, Figure 1.1 also shows that this 17-year hardware renaissance of rapid improvement in general-purpose processors is over. The fundamental reason is that two characteristics of semiconductor processes that were true for decades no longer hold.

In 1974, Robert Dennard observed that power density was constant for a given area of silicon even as you increased the number of transistors because of smaller dimensions of each transistor. Remarkably, transistors could go faster but use less power. *Dennard scaling* ended around 2004 because current and voltage couldn't keep dropping and still maintain the dependability of integrated circuits.

This change forced the microprocessor industry to use multiple efficient processors or cores instead of a single inefficient processor. Indeed, in 2004 Intel canceled its high-performance uniprocessor projects and joined others in declaring that the road to higher performance would be via multiple processors per chip rather than via faster uniprocessors. This milestone signaled a historic switch from relying solely on ILP, the primary focus of the first three editions of this book, to *data-level parallelism* (DLP) and *thread-level parallelism* (TLP), which were featured in the fourth edition and expanded in the fifth edition. The fifth edition also added WSCs and *request-level parallelism* (RLP), which is expanded in this edition. Whereas the compiler and hardware conspire to exploit ILP implicitly without the programmer's attention, DLP, TLP, and RLP are explicitly parallel, requiring the restructuring of the application so that it can exploit explicit parallelism. In some instances this is easy; in many it is a major new burden for programmers.

Amdahl's Law (Section 1.10) prescribes practical limits to the number of useful cores per chip. If 10% of the task is serial, then the maximum performance benefit from parallelism is 10, no matter how many cores you put on the chip.

The second observation that has changed recently is *Moore's Law.* In 1965 Gordon Moore famously predicted that the number of transistors per chip would double every year, which he amended in 1975 to every 2 years. That prediction lasted about 50 years but no longer holds. For example, in the 2010 edition of this book, the newest Intel microprocessor had 1,170,000,000 transistors. If Moore's Law had continued, we could have expected microprocessors in 2022 to have 105,900,000,000 transistors. Instead, the Apple M2 microprocessor has 20,000,000,000 transistors, or off by a factor of 5 from what Moore's Law would have predicted.

The combination of

- transistors no longer getting much better because of the slowing of Moore's Law and the end of Dennard scaling,
- the unchanging power budgets for microprocessors,

- the replacement of the single power-hungry processor with several energy-efficient processors, and
- the limits to multiprocessing given Amdahl's Law

caused improvements in processor performance to slow down, that is, to *double every 8 years* rather than every 1.5 years, as it did between 1986 and 2003 (see Figure 1.1).

The only path left to improve energy-performance-cost significantly is specialization. Future microprocessors will include several domain-specific cores that perform only one class of computations well, but they do so remarkably better than general-purpose cores. Chapter 7 introduces *domain-specific architectures*. The release of ChatGPT in 2023 has dramatically accelerated the deployment of accelerators designed solely for the domain of AI in everything from IoT devices to WSCs.

This book focuses on synchronous digital computers, which have driven the computer industry for the past 75 years. As improvements start to slow, some are investigating alternatives that have not yet had significant commercial impact, e.g., analog, biological, quantum, neuromorphic, or asynchronous computing. As it's still unclear when or if these different designs will become economically viable, synchronous digital hardware remains our subject.

This text is about the architectural ideas and accompanying compiler improvements that made the incredible growth rate possible over the past century, the reasons for the dramatic change, and the challenges and initial promising approaches to architectural ideas, compilers, and interpreters 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. The purpose of this chapter is to lay the quantitative foundation on which the following chapters and appendices are based.

This book was written not only to explain this design style but also to stimulate you to contribute to this progress. We believe this approach will serve the computers of the future just as it worked for the implicitly parallel computers of the past.

1.2 Classes of Computers

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 have we seen such striking changes in the way computers appear and in how they are used. These changes in computer use have led to five diverse 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.

Feature	Internet of things/ embedded	Personal mobile device (PMD)	Desktop	Server	Clusters/warehouse-scale computer
Price of system	$10–$100,000	$100–$1000	$300–$2500	$5000–$10,000,000	$100,000–$200,000,000
Price of microprocessor	$0.01–$100	$10–$100	$50–$500	$200–$2000	$50–$250
Critical system design issues	Price, energy, application-specific performance	Price, energy, media performance, responsiveness	Price-performance, energy, graphics performance	Throughput, availability, scalability, energy	Price-performance, throughput, energy proportionality

Figure 1.2 A summary of the five mainstream computing classes and their system characteristics. Sales in 2021 included about 2.0 billion PMDs (90% cell phones), 357 million PCs (80% laptops), and 13 million servers. The total number of embedded processors sold was about 30 billion. In total, 23 billion ARM technology–based chips were shipped in 2021. Note the wide range in system price for servers and embedded systems, which go from USB keys to network routers. For servers, this range arises from the need for very large-scale multiprocessor systems for high-end transaction processing.

Internet of Things/Embedded Computers

Embedded computers are found in everyday machines: microwaves, washing machines, most printers, networking switches, and all automobiles. The phrase *Internet of Things* (IoT) refers to embedded computers that are connected to the Internet, typically wirelessly. When augmented with sensors and actuators, IoT devices collect useful data and interact with the physical world, leading to a wide variety of "smart" applications, such as smart watches, smart thermostats, smart speakers, smart cars, smart homes, smart grids, and smart cities.

Embedded computers have the widest spread of processing power and cost. They include 8-bit to 32-bit processors that may cost 1 penny and high-end 64-bit processors for cars and network switches that cost $100. 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 often meets the performance need at a minimum price rather than achieving more performance at a higher price. The projections for the number of IoT devices shipped in 2025 range from 30 to 40 billion.

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

Unfortunately, the data that drive the quantitative design and evaluation of other classes of computers are in the early stages of being extended to embedded computing (see the challenges with EEMBC, for example, in Section 1.9). Hence we are left for now with qualitative descriptions, which do not fit well with the rest of the book. As a result, the embedded material is concentrated in Appendix E. We

believe a separate appendix improves the flow of ideas in the text while allowing readers to see how the differing requirements affect embedded computing.

Personal Mobile Device

Personal mobile device (PMD) is the term we apply to a collection of wireless devices with multimedia user interfaces such as smartphones, feature phones, tablet computers, smartwatches, and so on. In 2021, PMDs had the largest annual revenue at roughly half a trillion dollars. Cost is a prime concern given the consumer price for the whole product is a few hundred dollars. Although the emphasis on energy efficiency 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 energy and power in more detail in Section 1.5. Applications on PMDs are often web based and media oriented, like the previously mentioned Google Translate example. Energy and size requirements lead to the use of Flash memory for storage (Chapter 2) instead of magnetic disks.

The processors in a PMD are often considered embedded computers, but we are keeping them as a separate category because PMDs are platforms that can run externally developed software, and they share many of the characteristics of desktop computers. Other embedded devices are more limited in hardware and software sophistication. We use the ability to run third-party software as the dividing line between nonembedded and embedded computers.

Responsiveness and predictability are key characteristics of media applications. A *real-time performance* requirement means a segment of the application has an absolute maximum execution time. For example, in playing a video on a PMD, 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 miss the time constraint on an event occasionally, as long as not too many are missed. Real-time performance tends to be highly application dependent.

Other key characteristics in many PMD applications are the need to minimize memory and the need to use energy efficiently. Energy efficiency is driven by both battery power and heat dissipation. The memory can be a substantial portion of the system cost, and it is important to optimize memory size in such cases. The importance of memory size translates to an emphasis on code size, since data size is dictated by the application.

Desktop Computing

In 2021 desktop computing was about half the size of the smartphone market in dollar terms. Desktop computing spans from low-end netbooks that sell for under $300 to high-end, heavily configured workstations that may sell for $2500. Since

2008, more than half of the desktop computers made each year have been battery-operated laptop computers. In 2021, laptop computers comprised three-fourths of desktop sales.

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 in the 1980s, the role of servers grew to provide larger-scale and more reliable file and computing services. Such servers have become the backbone of large-scale enterprise computing, replacing the traditional mainframe.

For servers, different characteristics are important. First, availability is critical. (We discuss it in Section 1.7.) Consider the servers running ATMs for banks or airline reservation systems. Failure of such server systems is far more catastrophic than failure of a single desktop, since these servers must operate 7 days a week, 24 hours a day. Figure 1.3 estimates revenue costs of downtime for server applications.

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 expansion 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.

A third feature is that the cost target for servers is total cost of ownership (TCO) rather than purchase price. TCO is the purchase price plus the cost of operation over the lifetime of the computer, which includes the cost of electricity, cooling, maintenance, and space. Reasons for the TCO focus are:

- servers have a much longer lifetime than desktops, typically 5 to 8 years,
- they use much more power than PCs, and
- the purchase price of the server processor can be a quarter of the total TCO.

Finally, servers are designed for efficient throughput. That is, the overall performance of the server—in terms of transactions per minute or web pages served per second—is what is crucial. Responsiveness to an individual request remains important, but overall efficiency and cost-effectiveness, as determined by how many requests can be handled in a unit time, are the key metrics for most servers.

Application	Cost of downtime per hour	Annual losses with downtime of		
		1% (87.6 h/year)	0.5% (43.8 h/year)	0.1% (8.8 h/year)
Brokerage service	$4,000,000	$350,400,000	$175,200,000	$35,000,000
Energy	$1,750,000	$153,300,000	$76,700,000	$15,300,000
Telecom	$1,250,000	$109,500,000	$54,800,000	$11,000,000
Manufacturing	$1,000,000	$87,600,000	$43,800,000	$8,800,000
Retail	$650,000	$56,900,000	$28,500,000	$5,700,000
Health care	$400,000	$35,000,000	$17,500,000	$3,500,000
Media	$50,000	$4,400,000	$2,200,000	$400,000

Figure 1.3 Costs rounded to the nearest $100,000 of an unavailable system are 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 Landstrom (2014) and were collected and analyzed by Contingency Planning Research.

We return to the issue of assessing performance for different types of computing environments in Section 1.9.

Clusters/Warehouse-Scale Computers

The growth of SaaS for applications like search, social networking, video viewing and sharing, multiplayer games, online shopping, and so on has led to the growth of a class of computers called *clusters*. Clusters are collections of typically inexpensive servers connected by local area networks to act as a single larger computer. Each node runs its own operating system, and nodes communicate using a networking protocol. WSCs are the largest of the clusters, in that they are designed so that tens of thousands of servers can act as one. Chapter 6 describes this class of extremely large computers.

Price-performance and power are critical to WSCs since they are so large. As Chapter 6 explains, the majority of the cost of a warehouse is associated with power and cooling of the computers inside the warehouse. The annual amortized computers themselves and the networking gear cost for a WSC is ~$50 million, because they are usually replaced every few years. When you are buying that much computing, you need to buy wisely, because a 10% improvement in price-performance means an annual savings of $5 million (10% of $50 million) per WSC; a company like Amazon might have more than 100 WSCs!

WSCs are related to servers in that availability is critical. For example, Amazon.com had $575 billion in sales in 2023. As there are about 8760 hours in a year, the average revenue per hour was about $66 million. During a peak hour for Christmas shopping, the potential loss would be many times higher. As Chapter 6 explains, the difference between WSCs and servers is that WSCs use

redundant, inexpensive components as the building blocks, relying on a software layer to catch and isolate the many failures that will happen with computing at this scale to deliver the availability needed for such applications. Note that scalability for a WSC is handled by the local area network connecting the computers and not by integrated computer hardware, as in the case of servers.

Supercomputers are related to WSCs in that they are equally expensive, costing hundreds of millions of dollars. However, supercomputers differ by emphasizing floating-point performance and by running large, communication-intensive batch programs that can run for weeks at a time. They also typically use high bandwidth, low latency interconnects. In contrast, WSCs emphasize interactive applications, large-scale storage, standard local area networks, dependability, and high Internet bandwidth.

Classes of Parallelism and Parallel Architectures

Parallelism at multiple levels is now the driving force of computer design across all four classes of computers, with energy and cost being the primary constraints. There are basically two kinds of parallelism in applications:

1. *Data-level parallelism (DLP)* arises because there are many data items that can be operated on at the same time.
2. *Task-level parallelism (TLP)* arises because tasks of work are created that can operate independently and largely in parallel.

Computer hardware in turn can exploit these two kinds of application parallelism in four major ways:

1. *Instruction Level Parallelism (ILP)* exploits DLP at modest levels with compiler help using ideas like pipelining and at medium levels using ideas like speculative execution.
2. *Vector architectures, graphic processor units (GPUs), and multimedia instruction sets* exploit DLP by applying a single instruction to a collection of data in parallel.
3. *Thread Level Parallelism (TLP)* exploits either DLP or task-level parallelism in a tightly coupled hardware model that allows for interaction between parallel threads.
4. *Request Level Parallelism (RLP)* exploits parallelism among largely decoupled tasks specified by the programmer or the operating system.

When Flynn (1966) studied the parallel computing efforts in the 1960s, he found a simple classification whose abbreviations we still use today. They target DLP and TLP. He looked at the parallelism in the instruction and data streams

called for by the instructions at the most constrained component of the multiprocessor and placed all computers in one of four categories:

1. *Single instruction stream, single data stream* (SISD)—This category is the uniprocessor. The programmer thinks of it as the standard sequential computer, but it can exploit ILP. Chapter 3 covers SISD architectures that use ILP techniques such as superscalar and speculative execution.
2. *Single instruction stream, multiple data streams* (SIMD)—The same instruction is executed by multiple processors using different data streams. SIMD computers exploit *DLP* by applying the same operations to multiple items of data in parallel. In the early SIMD computers each processor had its own data memory (hence the MD of SIMD), but there is a single instruction memory and control processor, which fetches and dispatches instructions. (Recent SIMD computers fetch multiple data items at a time from a single shared memory.) Chapter 4 covers DLP and three different architectures that exploit it: vector architectures, multimedia extensions to standard instruction sets, and GPUs.
3. *Multiple instruction streams, single data stream* (MISD)—No successful commercial multiprocessor of this type has been built to date, but it rounds out this simple classification.
4. *Multiple instruction streams, multiple data streams* (MIMD)—Each processor fetches its own instructions and operates on its own data, and it targets TLP. In general, MIMD is more flexible than SIMD and thus more generally applicable, but it is inherently more expensive than SIMD. For example, MIMD computers can also exploit DLP, although the overhead is likely to be higher than would be seen in an SIMD computer. This overhead means that grain size must be sufficiently large to exploit the parallelism efficiently. Chapter 5 covers tightly coupled MIMD architectures, which exploit *TLP* because multiple cooperating threads operate in parallel. Chapter 6 covers loosely coupled MIMD architectures—specifically, *clusters* and *warehouse-scale computers*—that exploit *RLP*, where many independent tasks can proceed in parallel naturally with little need for communication or synchronization.

This taxonomy is a coarse model, as many parallel processors are hybrids of the SISD, SIMD, and MIMD classes. Nonetheless, it is useful to put a framework on the design space for the computers we will see in this book.

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 and energy efficiency 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.

A few decades ago, the term *computer architecture* generally referred to only 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: The Myopic View of Computer Architecture

We use the term *instruction set architecture* (ISA) to refer to the actual programmer-visible instruction set in this book. The ISA is the most important computer interface since it serves as the boundary between the software and hardware. This quick review of ISA will use examples from 80x86, ARMv8, and RISC-V to illustrate the seven dimensions of an ISA. The most popular RISC processors come from ARM (Advanced RISC Machine), which were in 30 billion chips shipped in 2023, or more than 100 times as many chips that shipped with 80x86 processors (often abbreviated x86). Appendices A and K give more details on the three ISAs.

RISC-V ("RISC Five") is a modern RISC instruction set developed at the University of California, Berkeley, in 2010, which was made free and openly adoptable in response to requests from industry. In addition to a full software stack (compilers, operating systems, and simulators), there are several RISC-V implementations freely available for use in custom chips or in field-programmable gate arrays. Developed 30 years after the first RISC instruction sets, RISC-V inherits its ancestors' good ideas—a large set of registers, easy-to-pipeline instructions, and a lean set of operations—while avoiding their omissions or mistakes. It is a free and open, elegant example of the RISC architectures mentioned earlier, which is why over 200 companies have joined the RISC-V International foundation, including Alibaba, AMD, Google, Intel, MIPS, Nokia, NVIDIA, Qualcomm, Samsung, Seagate, Siemens, Sony, and Western Digital. Two billion RISC-V systems on a chip (see Section 1.6) were shipped in 2024, and they are projected to grow to 30B per year in 2031. We use the integer core ISA of RISC-V as the example ISA in this book.

Class of ISA—Nearly all ISAs today are classified as general-purpose register architectures, where the operands are either registers or memory locations. The x86 has 16 general-purpose registers and 16 that can hold floating-point data, while RISC-V 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 x86, which can access memory as part of many instructions, and

Register	Name	Use	Saver
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	–
x4	tp	Thread pointer	–
x5-x7	t0-t2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Function arguments/return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller
f0-f7	ft0-ft7	FP temporaries	Caller
f8-f9	fs0-fs1	FP saved registers	Callee
f10-f11	fa0-fa1	FP function arguments/return values	Caller
f12-f17	fa2-fa7	FP function arguments	Caller
f18-f27	fs2-fs11	FP saved registers	Callee
f28-f31	ft8-ft11	FP temporaries	Caller

Figure 1.4 RISC-V registers, names, usage, and calling conventions. In addition to the 32 general-purpose registers (x0—x31), RISC-V has 32 floating-point registers (f0—f31) that can hold either a 32-bit single-precision number or a 64-bit double-precision number. The registers that are preserved across a procedure call are labeled "Callee" saved.

load-store ISAs, such as ARMv8 and RISC-V, which can access memory only with load or store instructions. All ISAs announced since 1985 are load-store.

1. *Memory addressing*—Virtually all desktop and server computers, including the x86, ARMv8, and RISC-V, use byte addressing to access memory operands. Some architectures, like ARMv8, require that objects must be *aligned.* An access to an object of size s bytes at byte address A is aligned if A mod $s = 0$. (See Figure A.5 on page A-8.) The 80x86 and RISC-V do not require alignment, but accesses are generally faster if operands are aligned.
2. *Addressing modes*—In addition to specifying registers and constant operands, addressing modes specify the address of a memory object. RISC-V addressing modes are Register, Immediate (for constants), and Displacement, where a constant offset is added to a register to form the memory address. The x86 supports those three modes, plus three variations of displacement: no register (absolute), two registers (based indexed with displacement), and two registers where 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 modes, minus the displacement field, plus register indirect, indexed, and based with scaled index. ARMv8 has the three RISC-V addressing modes plus PC-relative addressing, the sum of two registers, and the sum of two registers where one register is multiplied by the size of the operand in bytes. It also has autoincrement and autodecrement addressing, where the calculated address replaces the contents of one of the registers used in forming the address.

3. *Types and sizes of operands*—Like most ISAs, x86, ARMv8, and RISC-V 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 x86 also supports 80-bit floating point (extended double precision). With the recent excitement about AI, recent processors have made 16-bit (half-precision) and even smaller floating-point formats an optional feature (see Chapter 7).

4. *Operations*—The general categories of operations are data transfer, arithmetic logical, control (discussed next), and floating point. RISC-V is a simple and easy-to-pipeline ISA, and it is representative of the RISC architectures being used today. Figure 1.5 summarizes the integer RISC-V ISA, and Figure 1.6 lists the floating-point ISA. The x86 has a much richer and larger set of operations (see Appendix K).

5. *Control flow instructions*—Virtually all ISAs, including these three, support conditional branches, unconditional jumps, procedure calls, and returns. All three 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. RISC-V conditional branches (`BE`, `BNE`, etc.) test the contents of registers, and the x86 and ARMv8 branches test condition code bits set as side effects of arithmetic/logic operations. The ARMv8 and RISC-V procedure call places the return address in a register, whereas the x86 call (`CALLF`) places the return address on a stack in memory.

Encoding an ISA—There are two basic choices for encoding: *fixed length* and *variable length*. All ARMv8 and RISC-V instructions are 32 bits long, which simplifies instruction decoding. Figure 1.7 shows the RISC-V instruction formats. The x86 encoding is variable length, 1 to 18 bytes. Variable-length instructions can take less space than fixed-length instructions, so a program compiled for the x86 is usually smaller than the same program compiled for RISC-V. Note that choices mentioned previously 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, because the register field and addressing mode field can appear many times in a single instruction. (ARMv7 and RISC-V later offered extensions, called Thumb-2 and RV64IC, that provide a mix of 16-bit and 32-bit length instructions, respectively, to reduce program size. ARMv8 remains 32-bit exclusively. Code

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 12-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)
`flw, fld, fsw, fsd`	Load SP float, load DP float, store SP float, store DP float
`fmv._.x, fmv.x._`	Copy from/to integer register to/from floating-point register; "__" = S for single-precision, D for double-precision
`csrrw, csrrwi, csrrs, csrrsi, csrrc, csrrci`	Read counters and write status registers, which include counters: clock cycles, time, instructions retired
Arithmetic/logical	*Operations on integer or logical data in GPRs*
`add, addi, addw, addiw`	Add, add immediate (all immediates are 12 bits), add 32-bits only & sign-extend to 64 bits, add immediate 32-bits only
`sub, subw`	Subtract, subtract 32-bits only
`mul, mulw, mulh, mulhsu, mulhu`	Multiply, multiply 32-bits only, multiply upper half, multiply upper half signed-unsigned, multiply upper half unsigned
`div, divu, rem, remu`	Divide, divide unsigned, remainder, remainder unsigned
`divw, divuw, remw, remuw`	Divide and remainder: as previously, but divide only lower 32-bits, producing 32-bit sign-extended result
`and, andi`	And, and immediate
`or, ori, xor, xori`	Or, or immediate, exclusive or, exclusive or immediate
`lui`	Load upper immediate; loads bits 31-12 of register with immediate, then sign-extends
`auipc`	Adds immediate in bits 31–12 with zeros in lower bits to PC; used with `JALR` to transfer control to any 32-bit address
`sll, slli, srl, srli, sra, srai`	Shifts: shift left logical, right logical, right arithmetic; both variable and immediate forms
`sllw, slliw, srlw, srliw, sraw, sraiw`	Shifts: as previously, but shift lower 32-bits, producing 32-bit sign-extended result
`slt, slti, sltu, sltiu`	Set less than, set less than immediate, signed and unsigned
Control	*Conditional branches and jumps; PC-relative or through register*
`beq, bne, blt, bge, bltu, bgeu`	Branch GPR equal/not equal; less than; greater than or equal, signed and unsigned
`jal, jalr`	Jump and link: save PC + 4, target is PC-relative (`JAL`) or a register (`JALR`); if specify `x0` as destination register, then acts as a simple jump
`ecall`	Make a request to the supporting execution environment, which is usually an OS
`ebreak`	Debuggers used to cause control to be transferred back to a debugging environment
`fence, fence.i`	Synchronize threads to guarantee ordering of memory accesses; synchronize instructions and data for stores to instruction memory

Figure 1.5 Subset of the instructions in RISC-V. RISC-V has a base set of instructions (R64I) and offers optional extensions: multiply-divide (RVM), single-precision floating point (RVF), double-precision floating point (RVD). This figure includes RVM and the next one shows RVF and RVD. Appendix A details RISC-V.

Instruction type/opcode	Instruction meaning
Floating point	*FP operations on DP and SP formats*
`fadd.d`, `fadd.s`	Add DP, SP numbers
`fsub.d`, `fsub.s`	Subtract DP, SP numbers
`fmul.d`, `fmul.s`	Multiply DP, SP floating point
`fmadd.d`, `fmadd.s`, `fnmadd.d`, `fnmadd.s`	Multiply-add DP, SP numbers; negative multiply-add DP, SP numbers
`fmsub.d`, `fmsub.s`, `fnmsub.d`, `fnmsub.s`	Multiply-sub DP, SP numbers; negative multiply-sub DP, SP numbers
`fdiv.d`, `fdiv.s`	Divide DP, SP floating point
`fsqrt.d`, `fsqrt.s`	Square root DP, SP floating point
`fmax.d`, `fmax.s`, `fmin.d`, `fmin.s`	Maximum and minimum DP, SP floating point
`fcvt._._`, `fcvt._._u`, `fcvt._u._`	Convert instructions: `FCVT.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). Integers can be unsigned (`U`)
`feq._`, `flt._`,`fle._`	Floating-point compare between floating-point registers and record the Boolean result in integer register; "`__`" = `S` for single-precision, `D` for double-precision
`fclass.d`, `fclass.s`	Writes to integer register a 10-bit mask that indicates the class of the floating-point number ($-\infty$, $+\infty$, -0, $+0$, NaN, …)
`fsgnj._`, `fsgnjn._`, `fsgnjx._`	Sign-injection instructions that changes only the sign bit: copy sign bit from other source, the oppositive of sign bit of other source, XOR of the 2 sign bits

Figure 1.6 Floating-point instructions for RISC-V. RISC-V has a base set of instructions (R64I) and offers optional extensions for single-precision floating point (RVF) and double-precision floating point (RVD). *DP*, Double precision; *SP*, single precision.

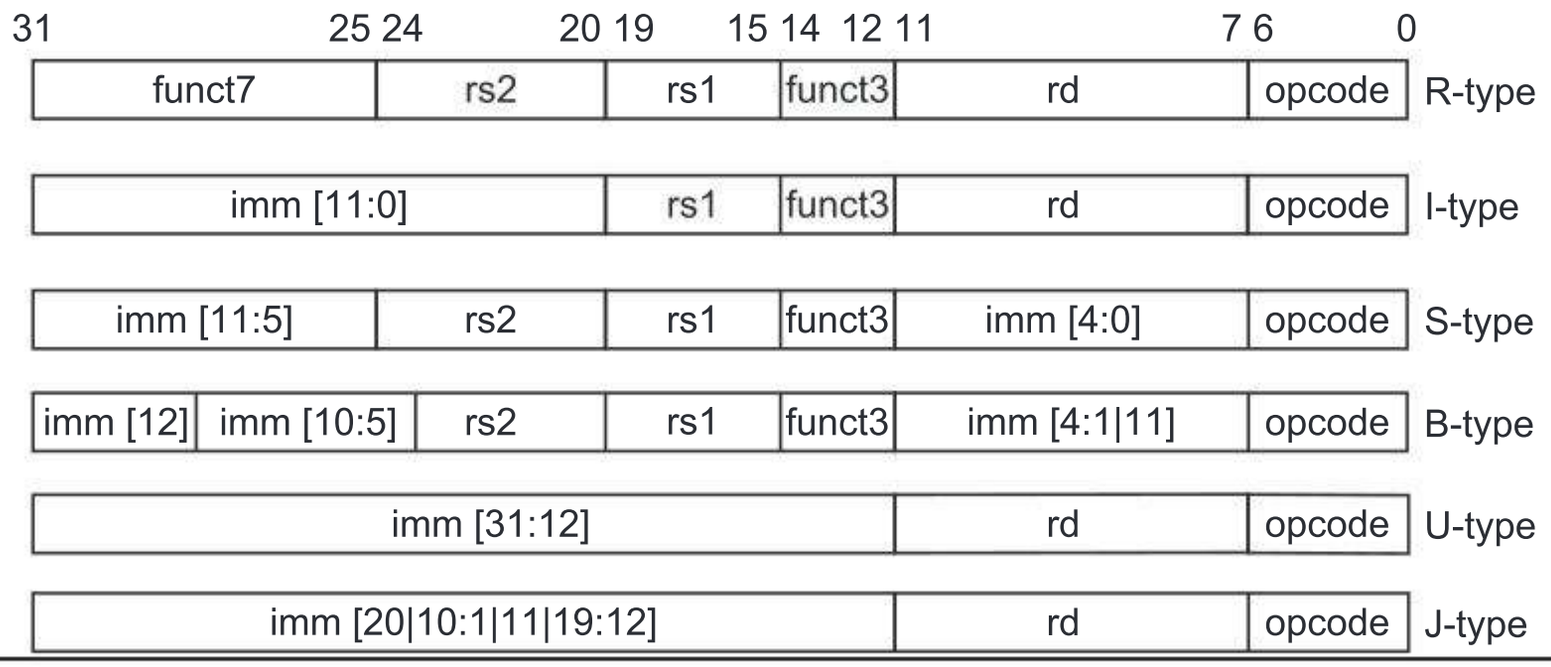

Figure 1.7 The base RISC-V instruction set architecture formats. All instructions are 32 bits long. The R format is for integer register-to-register operations, such as ADD, SUB, and so on. The I format is for loads and immediate operations, such as LD and ADDI. The B format is for branches and the J format is for jumps and link. The S format is for stores. Having a separate format for stores allows the three register specifiers (rd, rs1, rs2) to always be in the same location in all formats. The U format is for the wide immediate instructions (LUI, AUIPC).

sizes of these compact versions of RISC architectures are smaller than that of the x86. See Appendix K.)

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

Genuine Computer Architecture: Designing the Organization and Hardware to Meet Goals and Functional Requirements

A computer implementation 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). The term *microarchitecture* is also used instead of organization. For example, two processors with the same ISA but different organizations are the AMD EPYC and the Intel Xeon. Both processors implement the x86 instruction set, but they have very different pipeline and cache organizations.

The switch to multiple processors per microprocessor led to the term *core* also being used for processors. Instead of saying multiprocessor microprocessor, the term *multicore* caught on. The term central processing unit, or CPU, generally refers to one of the cores of a microprocessor, since virtually all microprocessors in PMDs, PCs, and servers are multicore.

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 ISAs and very similar organizations, but they differ in the detailed hardware implementation. For example, the Intel Core i7 (see Chapter 3) and the Intel Xeon E7 (see Chapter 5) are nearly identical but offer different clock rates and different memory systems, making the Xeon E7 more effective for server computers.

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

Computer architects must design a computer to meet functional requirements as well as price, power, performance, and availability goals. Figure 1.8 summarizes requirements to consider in designing a new computer. Often, architects 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 typically drives the choice of certain functional requirements by determining how the computer will be used. If a large body of software exists for a particular ISA, the architect may decide that a new computer should

Functional requirements	Typical features required or supported
Application area	*Target of computer*
Internet of things/embedded computing	Often requires special support for graphics or video (or other application-specific extension); power limitations and power control may be required; real-time constraints (Chapters 2, 3, 5, and 7; Appendices A and E)
Personal mobile device	Real-time performance for a range of tasks, including interactive performance for graphics, video, and audio; energy efficiency (Chapters 2–5 and 7; Appendix A)
General-purpose desktop	Balanced performance for a range of tasks, including interactive performance for graphics, video, and audio (Chapters 2–5; Appendix A)
Servers	Support for databases and transaction processing; enhancements for reliability and availability; support for scalability and security (Chapters 2, 5, and 7; Appendices A, D, and F)
Clusters/warehouse-scale computers	Throughput performance for many independent tasks; error correction for memory; energy proportionality; security (Chapters 2, 6, and 7; Appendix F)
Level of software compatibility	*Determines amount of existing software for computer*
At programming language	Most flexible for designer; need new compiler (Chapters 3, 5, and 7; Appendix A)
Object code or binary compatible	Instruction set architecture is completely defined—little flexibility—but no investment needed in software or porting programs (Appendix A)
Operating system requirements	*Necessary features to support chosen OS* (Chapter 2; Appendix B)
Size of address space	Very important feature (Chapter 2); may limit applications
Memory management	Required for modern OS; may be paged or segmented (Chapter 2)
Protection	Different OS and application needs: page versus segment; virtual machines (Chapter 2)
Standards	*Certain standards may be required by marketplace*
Floating point	Format and arithmetic: IEEE 754 standard (Appendix J), special arithmetic for graphics or signal processing or AI
I/O interfaces	For I/O devices: Serial ATA, Serial Attached SCSI, PCI Express (Appendices D and F)
Operating systems	UNIX, Windows, Linux, iOS Android
Networks	Support required for different networks: Ethernet, Infiniband (Appendix F)
Programming languages	Languages (ANSI C, C++, Java, Fortran) affect instruction set (Appendix A)

Figure 1.8 Summary of some of the key functional requirements facing an architect. 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.

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. Later chapters examine many of these requirements and features in depth.

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

1.4 Trends in Technology

If an ISA is to prevail, it must be designed to survive rapid changes in computer technology. After all, a successful new ISA may last decades—for example, the core of the IBM mainframe has been in use for 60 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. Five implementation technologies, which change at a dramatic pace, are critical to modern implementations:

- *Integrated circuit logic technology*—Historically, transistor density increased by about 35% per year, quadrupling somewhat over 4 years. Die size increases are less predictable and slower, from 10% to 20% per year. The combined effect was a traditional growth rate in transistor count on a chip of about 40% to 55% per year, or doubling every 18–24 months. As previously mentioned, this trend is called Moore's Law. Device speed scales more slowly, as we will discuss. Shockingly, Moore's Law is slowing. The number of devices per chip is still increasing, but at a decelerating rate. Unlike in the Moore's Law era, we expect the doubling time to be stretched with each new technology generation (see Figure 1.24 in Section 1.12).
- *Semiconductor DRAM* (dynamic random-access memory)—This technology is the foundation of main memory, and we discuss it in Chapter 2. The growth of DRAM has slowed dramatically, from quadrupling every 3 years as in the past. The 8-gigabit DRAM was shipping in 2014, but the 16-gigabit DRAM didn't ship until 2019. The next step was 24-gigabit in 2023, which was the first DRAM capacity that is not a power of 2. The following target is 32-gigabit DRAM. That means it will take at least 10 years for DRAM capacity to quadruple from 8 to 32 gigabits. There are significant challenges to future growth (Kim, 2005). Chapter 2 mentions several other technologies that hope to replace DRAM, but it's become more clear that a DRAM replacement is unlikely.
- *Semiconductor Flash* (electrically erasable programmable read-only memory)—This nonvolatile semiconductor memory is the standard storage device in PMDs, and its rapidly increasing popularity has fueled its rapid growth rate in capacity. In the server space, solid state disks (SSDs) based on flash are more than half of disk sales in 2024. In recent years the capacity per Flash chip increased by about 50%–60% per year, doubling roughly every 3 years. Currently, Flash memory is 8–10 times cheaper per bit than DRAM. Chapter 2 describes Flash memory. Future capacity increases will continue to be in the third dimension, by adding many more layers to the processing of a wafer as well as shrinking the bit cell size.
- *Magnetic disk technology*—Prior to 1990, density increased by about 30% per year, doubling in 3 years. It rose to 60% per year thereafter and increased to

100% per year in 1996. Between 2004 and 2011, it dropped to about 40% per year, or doubled every 2 years. For the past decade disk improvement has slowed to less than 5% per year, doubling in 15–20 years. One way to increase disk capacity is to add more platters at the same areal density, but there are already 10 platters within the 1-inch depth of the 3.5-inch form factor disks. Hard disk drives are now 3–5 times cheaper per bit than Flash and 100–200 times cheaper per bit than DRAM. In the future, the depth of the package will likely need to increase to accommodate more platters, which could reduce the number of disks per unit volume. This technology is central to server- and warehouse-scale storage, and we discuss the trends in detail in Appendix D.

- *Network technology*—Network performance depends both on the performance of switches and on the performance of the transmission system. We discuss networking trends in Appendix F.

These technologies shape the design of a computer that, with speed and technology enhancements, may have a lifetime of 3–7 years. Engineers often design for the next technology, knowing that, when a product begins shipping in volume, the following technology may be the most cost-effective or may have performance advantages.

Although technology generally improves continuously, the impact of these increases can be in discrete leaps, as a threshold that allows a new capability to be 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 a chip. Later gains incorporated second and even third levels of cache. By eliminating chip crossings within the processor and between the processor and levels of caches, a dramatic improvement in cost-performance and energy-performance was possible. This design was simply unfeasible 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 1.9 plots the relative improvement in bandwidth and latency for technology milestones for microprocessors, memory, networks, and disks. Figure 1.10 describes the examples and milestones in more detail.

Performance is the primary differentiator for microprocessors and networks, so they have seen the greatest gains: 45,000–80,000× in bandwidth and 50–90× in latency. Capacity is generally more important than performance for

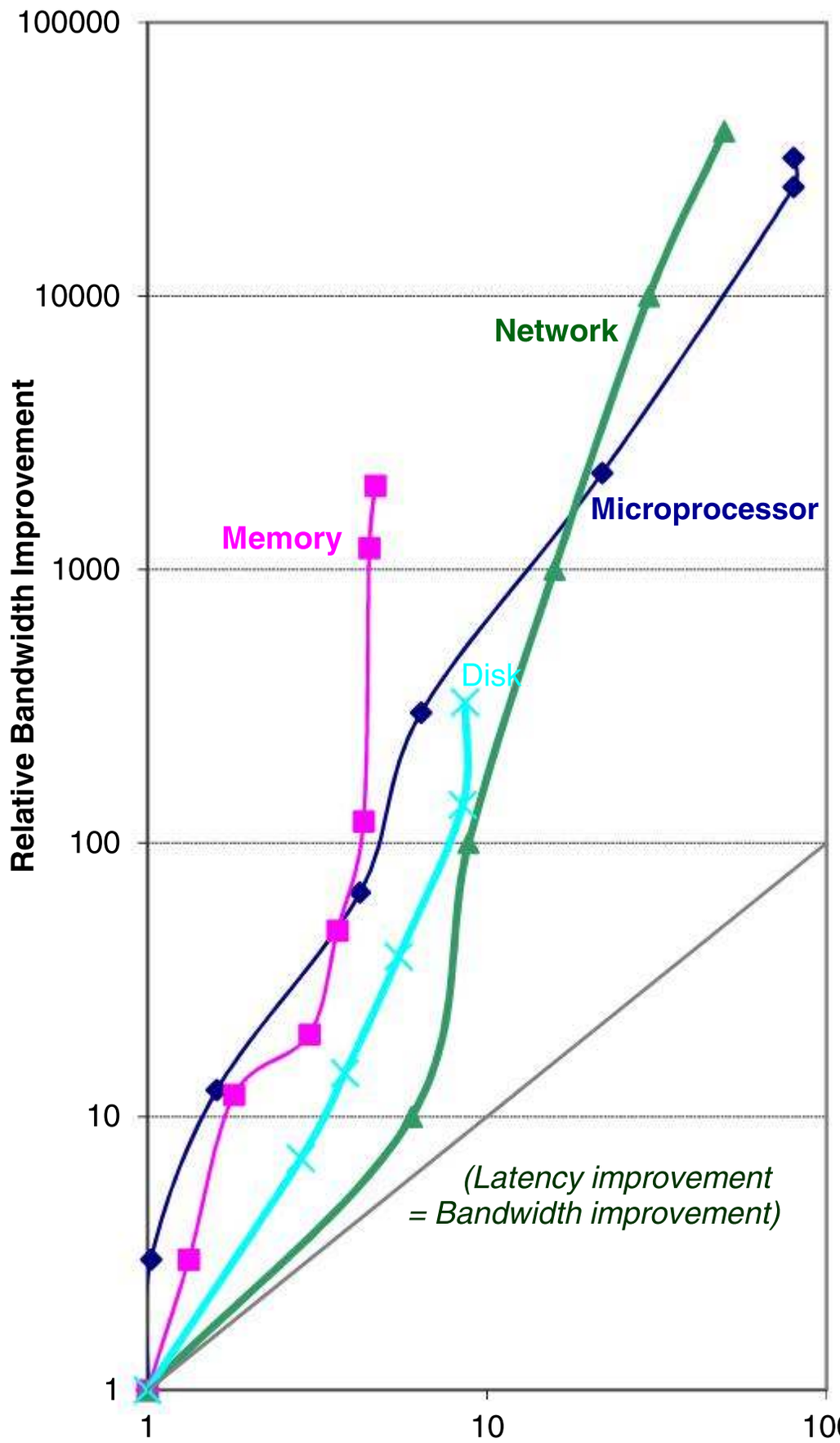

Figure 1.9 Log-log plot of bandwidth and latency milestones in Figure 1.10 relative to the first milestone. Note that latency improved by 5× to 64×, while bandwidth improved by about 450× to 80,000×. In the 5 years since the last edition, bandwidth improved for all technologies from 14% to 100%, but latency did not improve. In fact, latency *increased* for disks and microprocessors. For disks, the same 15,000 RPM disk at the same 0.6 TB capacity as in 2016 is still available, but we felt it made more sense to include the 20 TB 7200 RPM disk, which has longer latency. The highest-performing SPEC2017 microprocessor in 2021 had a lower clock rate of 2.8 GHz (albeit with a higher 5.1 GHz Turbo mode rate) than the 2016 microprocessor with a 4.0 GHz clock and a 4.2 GHz Turbo mode. The latency is longer for the slower standard clock. Note that the network standard is not expected to complete until 2026, so this chart is a bit optimistic for networks. Updated from Patterson, D., 2004. Latency lags bandwidth. Commun. ACM 47 (10), 71—75.

Micrprocessor	16-bit address/ bus, microcode	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 L2 cache	Multicore OOO 16-way on chip 8MB L3 cache, Turbo	Multicore OOO 16-way on chip 40MB L3 cache, Turbo	Multicore OOO 16-way on chip 40MB L3 cache, Turbo
Product	Intel 80286	Intel 80386	Intel 80486	Intel Penitum	Intel Pentium Pro	Intel Pentium 4	Intel Core i7	Intel Core i7	Intel Xeon
Year	1982	1985	1989	1993	1997	2001	2010	2015	2021
Die size (mm^2)	47	43	81	90	308	217	240	122	400
Transistors	134,000	275,000	1,200,000	3,100,000	5,500,000	42,000,000	1,170,000,000	1,750,000,000	
Processors/Chip	1	1	1	1	1	1	4	4	8
Latency (clocks)	6	5	5	5	10	22	14	14	14
Bus width (bits)	16 bits	32 bits	32 bits	64 bits	64 bits	64 bits	196	196	196
Clock rate (MHz)	13	16	25	66	200	1500	3333	4000	2800
Turbo mode (MHz)	--	--	--	--	--	--	3600	4200	5100
Bandwidth (Peak MIPS)	2	6	25	132	600	4500	50000	64000	89600
Lantecy (nanoseconds)	320	313	200	76	50	15	4	4	5
Memory Module	DRAM	Page Mode DRAM	Fast Page Mode DRAM	Fast Page Mode DRAM	SDRAM	DDR DRAM	DDR3 SDRAM	DDR4 SDRAM	DDR5 SDRAM
Module width (bits)	16 bits	16 bits	32 bits	64 bits	64 bits	64 bits	64 bits	64 bits	64 bits
Year	1980	1983	1986	1993	1997	2000	2010	2016	2021
Mbits/DRAM chip	0.06	0.25	1.00	16.00	64.00	256.00	2048	4,096	16,384
Pins/DRAM chip	16	16	18	20	54	66	134	134	288
Bandwidth (MBytes/sec)	13	40	160	267	640	1,600	16,000	27,000	51,000
Lantecy (nanoseconds)	225	170	125	75	62	52	50	48	46
Local Area Network	Ethernet	Fast Ethernet	Gigabit Ethernet	10 Gigabit Thernet	100 Gigabit Ethernet	400 Gigabit Ethernet	800 Gigabit Ethernet		
Year	1978	1995	1999	2003	2010	2017	2024?		
IEEE Standard	802.3	802.3u	802.3ab	802.3ac	802.3ba	802.3bs	802.3df		
Bandwidth (Mbits/sec)	10	100	1000	10000	100,000	400,000	800,000		
Latency (usecs)	3000	500	340	190	100	60	60		
Hard Disk Product	3600 RPM	5400 RPM	7200 RPM	10000 RPM	15000 RPM	15,000 RPM	15,000 RPM	7,200 RPM	
Year	1983	1990	1994	1998	2003	2010	2016	2021	
Capacity (GBytes)	0.03	1.4	4.3	9.1	73.4	600	600	20,000	
Disk form factor	5.25 in.	5.25 in.	3.5 in.	3.5 in.	3.5 in.	3.5 in.	3.5 in.	3.5 in.	
Media diameter	5.25 in.	5.25 in.	3.5 in.	3.0 in.	2.5 in.	2.5 in.	2.5 in.	2.5 in.	
Interface							SAS	SATA 6 Gbps	
Bandwidth (MBytes/sec)	1	4	9	24	86	204	250	285	
Lantecy (ms)	48.3	17.1	12.7	8.8	5.7	5.6	5.6	12.0	

Figure 1.10 Performance milestones over 25—45 years for microprocessors, memory, networks, and disks. The microprocessor milestones are several generations of IA-32 processors, going from a 16-bit bus, microcoded 80286 to a 64-bit bus, multicore, out-of-order execution, superpipelined Xeon with three levels of on-chip cache. Memory module milestones go from 16-bit-wide, plain DRAM to 64-bit-wide double data rate version 5 synchronous DRAM. Ethernet advanced from 10 Mbits/s to 800 Gbits/s in about 2026. Disk milestones are based on rotation speed, improving from 3600 to 15,000 RPM through 2016 and then reverting to 7200 RPM in 2021. Each case is best-case bandwidth, and latency is the time for a simple operation assuming no contention. Updated from Patterson, D., 2004. Latency lags bandwidth. Commun. ACM 47 (10), 71—75.

memory and disks, so capacity has improved more, yet bandwidth advances of 450—4000× are still much greater than gains in latency of 5—9×.

Clearly, bandwidth has outpaced latency across these technologies and will likely continue to do so. Computer designers should plan accordingly.

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 decreased from 10 μm in 1971 to 0.007 μm in 2022; in fact, we have switched units, so production in 2022 is referred to as "7 nm" and 3 nm chips are underway. Intel announced plans for 1.8 nm chips, which they relabeled as 18Å (Angstrom), to set the stage for ever-smaller geometrics. 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.

Increasing 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, in the past, the transistor performance improved linearly with decreasing feature size.

The fact that transistor count improves quadratically with a linear increase 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, to 64-bit microprocessors. More recently, density improvements have supported the introduction of multiple processors per chip, wider SIMD units, and many of the innovations in speculative execution and caches found in Chapters 2–5.

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 addition to the power dissipation limit, wire delay has become a major design obstacle 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, but power now plays an even greater role than wire delay.

1.5 Trends in Power and Energy in Integrated Circuits

Today, energy is the biggest challenge facing the computer designer for nearly every class of computer. First, power must be brought in and distributed around the chip, and modern microprocessors use hundreds of pins and multiple interconnect layers just for power and ground. Second, power is dissipated as heat and must be removed.

Power and Energy: A Systems Perspective

How should a system architect or a user think about performance, power, and energy? From the viewpoint of a system designer, there are three primary concerns.

First, what is the maximum power a processor ever requires? Meeting this demand can be important to ensuring correct operation. For example, if a processor attempts to draw more power than a power-supply system can provide (by drawing more current than the system can supply), the result is typically a voltage drop, which can cause devices to malfunction. Modern processors can vary widely in power consumption with high peak currents; hence they provide voltage indexing methods that allow the processor to slow down and regulate voltage within a wider margin. Obviously, doing so decreases performance.

Second, what is the sustained power consumption? This metric is widely called the *thermal design power* (TDP) because it determines the cooling requirement. TDP is neither peak power, which is often 1.5 times higher, nor is it the actual average power that will be consumed during a given computation, which is likely to be lower still. A typical power supply for a system is typically sized to exceed the TDP, and a cooling system is usually designed to match or exceed TDP. Failure to provide adequate cooling allows the junction temperature in the processor to exceed its maximum value, resulting in device failure and possibly permanent damage. Modern processors provide two features to assist in managing heat, since the highest power (and hence heat and temperature rise) can exceed the long-term average specified by the TDP. First, as the thermal temperature approaches the junction temperature limit, circuitry lowers the clock rate, thereby reducing power. Should this technique not be successful, a second thermal overload trap is activated to power down the chip.

The third factor designers and users should consider is energy and energy efficiency. Recall that power is simply energy per unit time: 1 W = 1 joule per second. Which metric is right for comparing processors: energy or power? In general, energy is always a better metric because it is tied to a specific task and the time required for that task. In particular, the energy to complete a workload is equal to the average power times the execution time for the workload.

Thus, if we want to know which of two processors is more efficient for a given task, we need to compare energy consumption (not power) for executing the task. For example, processor A may have a 20% higher average power consumption than processor B, but if A executes the task in only 70% of the time needed by B, its energy consumption will be $1.2 \times 0.7 = 0.84$, which is clearly better.

One might argue that in a large server or in the cloud, it is sufficient to consider the average power, since the workload is often assumed to be infinite, but this is misleading. If our cloud were populated with processor Bs rather than As, then the cloud would do less work for the same amount of energy expended. Using energy to compare the alternatives avoids this pitfall. Whenever we have a fixed workload, whether for a warehouse-size cloud or a smartphone, comparing energy will be the right way to compare computer alternatives,

because the electricity bill for the cloud and the battery lifetime for the smartphone are both determined by the energy consumed.

When is power consumption a useful measure? The primary legitimate use is as a constraint: for example, an air-cooled chip might be limited to 300 W. It can be used as a metric if the workload is fixed, but then it's just a variation of the true metric of energy per task.

Energy and Power Within a Microprocessor

For CMOS chips, the traditional primary energy consumption has been in switching transistors, also called *dynamic energy.* The energy required per transistor is proportional to the product of the capacitive load driven by the transistor and the square of the voltage:

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

This equation is the energy of the pulse of the logic transition of $0 \rightarrow 1 \rightarrow 0$ or $1 \rightarrow 0 \rightarrow 1$. The energy of a single transition ($0 \rightarrow 1$ or $1 \rightarrow 0$) is then:

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

The power required per transistor is just the product of the energy of a transition multiplied by the frequency of transitions:

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

For a fixed task, slowing clock rate reduces power, but not energy.

Clearly, dynamic power and energy are greatly reduced by lowering the voltage, so voltages have dropped from 5 V to just under 1 V 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.

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

Answer Because the capacitance is unchanged, the answer for energy is the ratio of the voltages

$$\frac{\text{Energy}_{\text{new}}}{\text{Energy}_{\text{old}}} = \frac{(\text{Voltage} \times 0.85)^2}{\text{Voltage}^2} = 0.85^2 = 0.72$$

which reduces dynamic energy to about 72% of the original. For power, we add the ratio of the frequencies

$$\frac{\text{Power}_{\text{new}}}{\text{Power}_{\text{old}}} = 0.72 \times \frac{(\text{Frequency switched} \times 0.85)}{\text{Frequency switched}} = 0.61$$

shrinking dynamic power to about 61% 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 change dominate the decrease in load capacitance and voltage, leading to an overall growth in power consumption and energy. The first microprocessors consumed less than a watt, and the first 32-bit microprocessors (such as the Intel 80386) used about 2 W, whereas a 2.4 GHz Intel Xeon Platinum 8380 consumes 270 W. Given that this heat must be dissipated from a chip that is about 2.5 cm on a side, we are near the limit of what can be cooled by air, and this is where we have been stuck for nearly a decade. (Recent announcements from Intel and NVIDIA include chips with TDPs near 1000 W, which require liquid cooling.)

Given the preceding equation, you would expect clock frequency growth to slow down if we can't reduce voltage or increase power per chip. Figure 1.11 shows that this has indeed been the case since 2003, even for the microprocessors in Figure 1.1 that were the highest performers each year. Note that this period of flatter clock rates corresponds to periods of slow performance improvement range in Figure 1.1.

Distributing the power, removing the heat, and preventing hot spots have become increasingly difficult challenges. Energy is now the major constraint to using transistors; in the past, it was the raw silicon area. Therefore modern microprocessors offer many techniques to try to improve energy efficiency despite flat clock rates and constant supply voltages:

1. *Do nothing well.* 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. If some cores are idle, their clocks are stopped.
2. *Dynamic voltage-frequency scaling (DVFS).* The second technique comes directly from the preceding formulas. PMDs, laptops, and even servers have periods of low activity where there is no need to operate at the highest clock frequency and voltages. Modern microprocessors typically offer a few clock frequencies and voltages in which to operate that use lower power and energy. Figure 1.12 plots the potential power savings via DVFS for a server as the workload shrinks for three different clock rates: 2.4, 1.8, and 1 GHz. The overall server power savings is about 10% to 15% for each of the two steps.
3. *Design for the common case.* Given that PMDs and laptops are often idle, memory and storage offer low-power modes to save energy. For example, DRAMs have a series of increasingly lower power modes to extend battery life in PMDs and laptops, and there have been proposals for disks that have

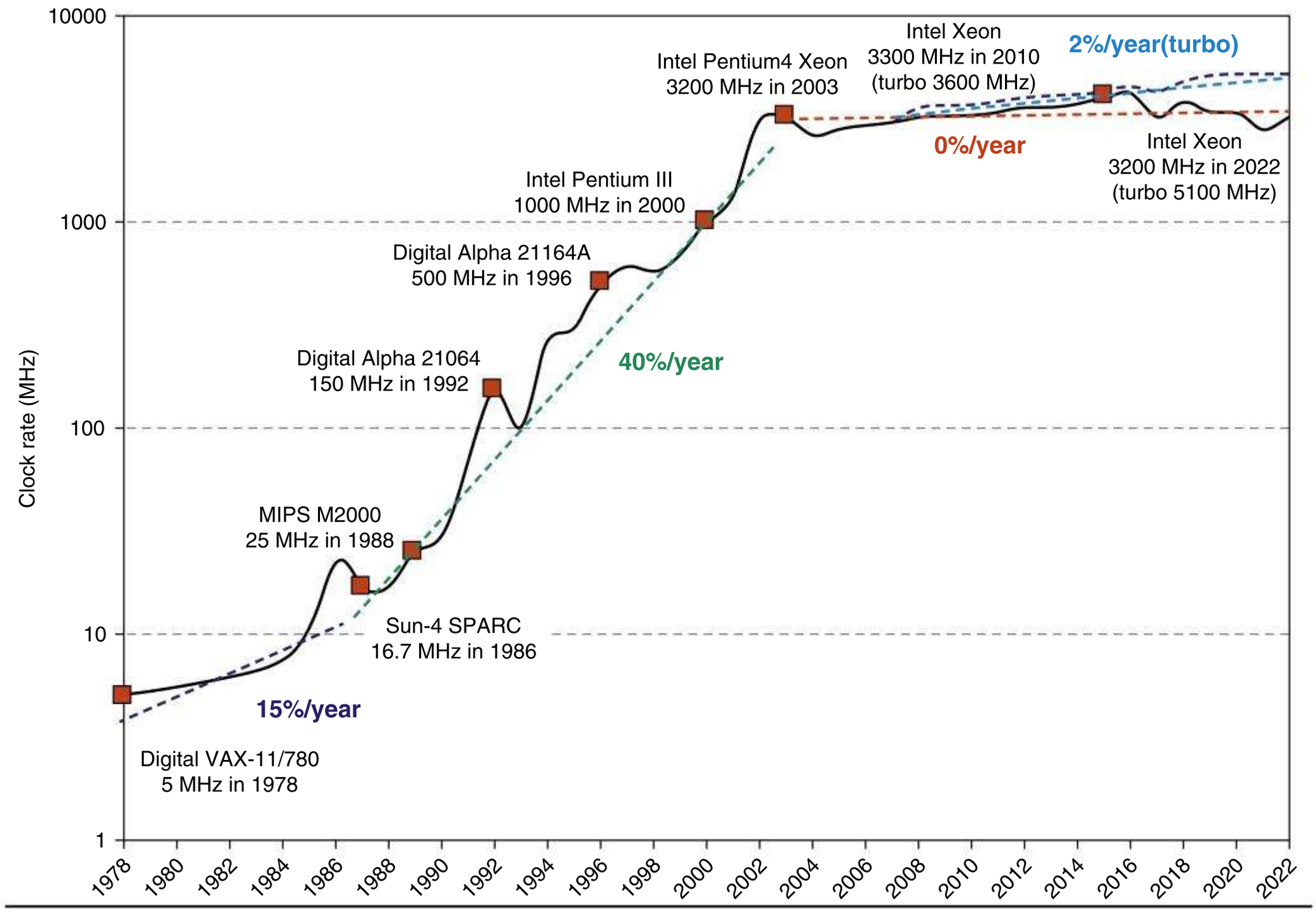

Figure 1.11 Growth in clock rate of microprocessors in Figure 1.1. Between 1978 and 1986, the clock rate improved less than 15% per year while performance improved by 22% per year. During the "renaissance period" of 52% performance improvement per year between 1986 and 2003, clock rates shot up almost 40% per year. Since then, the clock rate has been nearly flat. Since 2008 Intel has offered Turbo mode, where the chip can run temporarily at a higher clock rate until it gets too warm. Even Turbo mode has only increased about 2% per year from 2011 to 2022.

a mode that spins more slowly when unused to save power. However, you cannot access DRAMs or disks in these modes, so you must return to fully active mode to read or write, no matter how low the access rate. As mentioned, microprocessors for PCs have been designed instead for heavy use at high operating temperatures, relying on on-chip temperature sensors to detect when activity should be reduced automatically to avoid overheating. This "emergency slowdown" allows manufacturers to design for a more typical case and then rely on this safety mechanism if someone really does run programs that consume much more power than is typical.

4. *Overclocking.* Intel started offering *Turbo mode* in 2008, where the chip decides that it is safe to run at a higher clock rate for a short time, possibly on just a few cores, until temperature starts to rise. For example, the 2.8 GHz

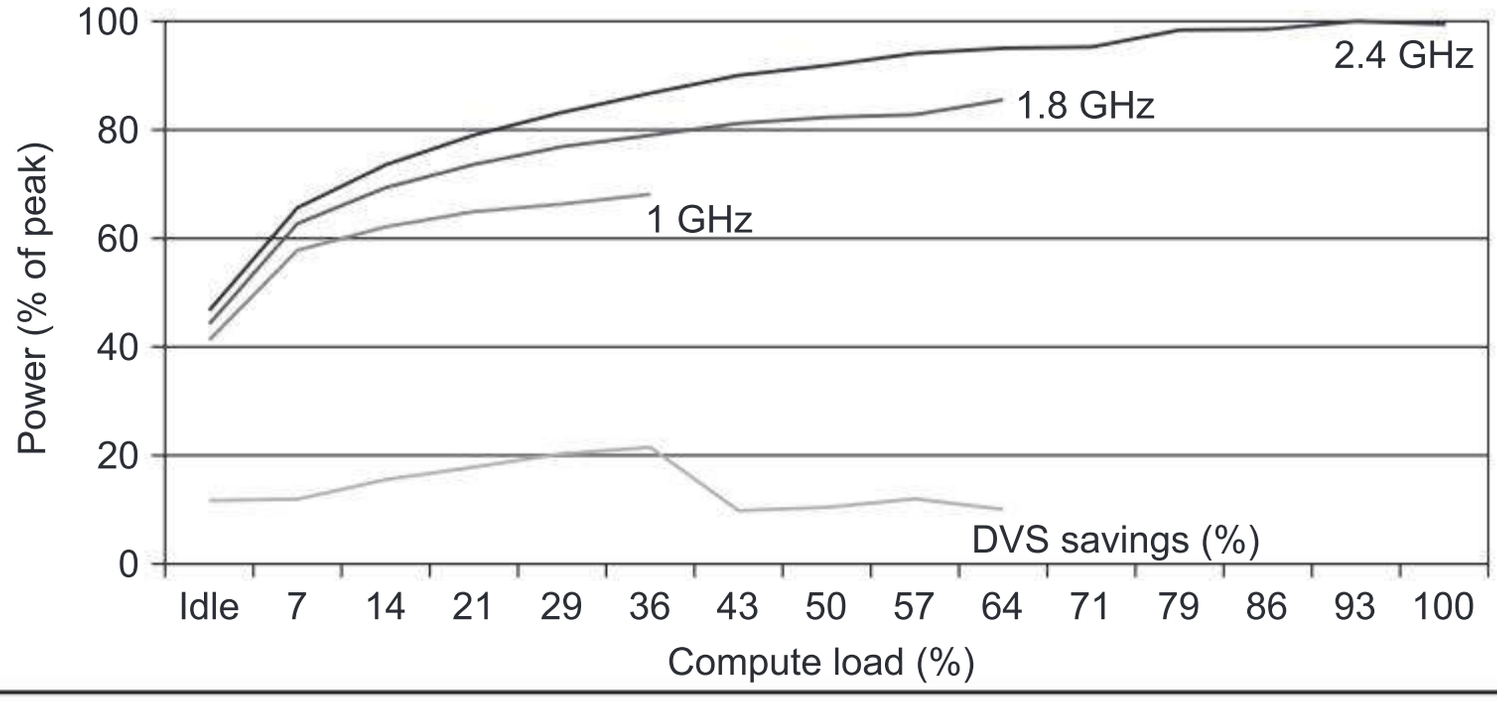

Figure 1.12 Power vs. compute load for a server at three voltage-frequency levels and corresponding energy savings. At 1.8 GHz, the server can handle at most up to two-thirds of the workload without causing service-level violations, and at 1 GHz, it can safely handle only one-third of the workload (Figure 5.11 in Barroso, Hölzle, and Ranganathan, 2019).

Xeon E-2378G can run in short bursts for 5.1 GHz (Figure 1.11). Indeed, the highest-performing microprocessors each year since 2008, shown in Figure 1.1, have all offered temporary overclocking of about 10% over the nominal clock rate through 2016, but since then Turbo mode has been 30% to 80% higher. For single-threaded code, these microprocessors can turn off all cores but one and run it faster. Note that although the operating system can turn off Turbo mode, there is no notification once it is enabled, so the programmers may be surprised to see their programs vary in performance because of room temperature!

5. *Heterogenous cores.* Cell phones have long had both fast, high-powered cores and slow, energy-efficient cores. For example, the chip in the Apple iPhone 14 Pro smartphone has two high-performance cores and four high-efficiency cores. The goal is to save energy by using the most efficient processor to run an application that maintains the responsiveness that the user expects. Smartphones even have processors dedicated to narrow domains to further improve energy efficiency for those tasks.

Although dynamic power is traditionally thought of as 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:

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

That is, static power is proportional to the number of devices.

Thus increasing the number of transistors increases power even if they are idle, and current leakage increases in processors with smaller transistor sizes. As a result, very low-power systems are even turning off the power supply

(*power gating*) to inactive modules to control loss because of leakage. In 2011, the goal for leakage was 25% of the total power consumption, with leakage in high-performance designs sometimes far exceeding that goal. Leakage can be as high as 50% for such chips, in part because of the large SRAM caches that need power to maintain the storage values. (The S in SRAM is for static.) The only hope to stop leakage is to turn off power to the chips' subsets.

Finally, because the processor is just a portion of the whole energy cost of a system, it can make sense to use a faster, less energy-efficient processor to allow the rest of the system to go into sleep mode. This strategy is known as *race-to-halt*.

The importance of power and energy has increased the scrutiny on the efficiency of an innovation, so the primary evaluation now is tasks per joule or performance per watt, contrary to performance per square millimeter of silicon as in the past. This new metric affects approaches to parallelism, as we will see in Chapters 4 and 5.

The Shift in Computer Architecture Because of Limits of Energy

As transistor improvement decelerates, computer architects must look elsewhere for improved energy efficiency. Indeed, given the energy budget, it is easy today to design a microprocessor with so many transistors that they cannot all be turned on at the same time. This phenomenon has been called *dark silicon*, in that much of a chip cannot be used ("dark") at any moment in time because of thermal constraints. This observation has led architects to reexamine the fundamentals of processors' design in the search for a greater energy-cost performance.

Figure 1.13, which lists the energy cost of the building blocks of a modern computer, reveals surprisingly large ratios. For example, a 32-bit floating-point addition uses 50 times as much energy as an 8-bit integer add. However, the

Operation	Energy (pJ)	Relative Energy
Int 8 add	0.007	1
Int 32 add	0.030	4
BFloat 16 add	0.110	16
IEEE FP 16 add	0.160	23
IEEE FP 32 add	0.380	54
Int 8 multiply	0.070	10
Int 32 multiply	1.480	211
BFloat 16 multiply	0.210	30
IEEE FP 16 multiply	0.340	49
IEEE FP 32 multiply	1.310	187
32b read 8 KB SRAM	7.500	1,071
32b read 1 MB SRAM	14.000	2,000
32b read DDR4 DRAM	1,300.000	185,714

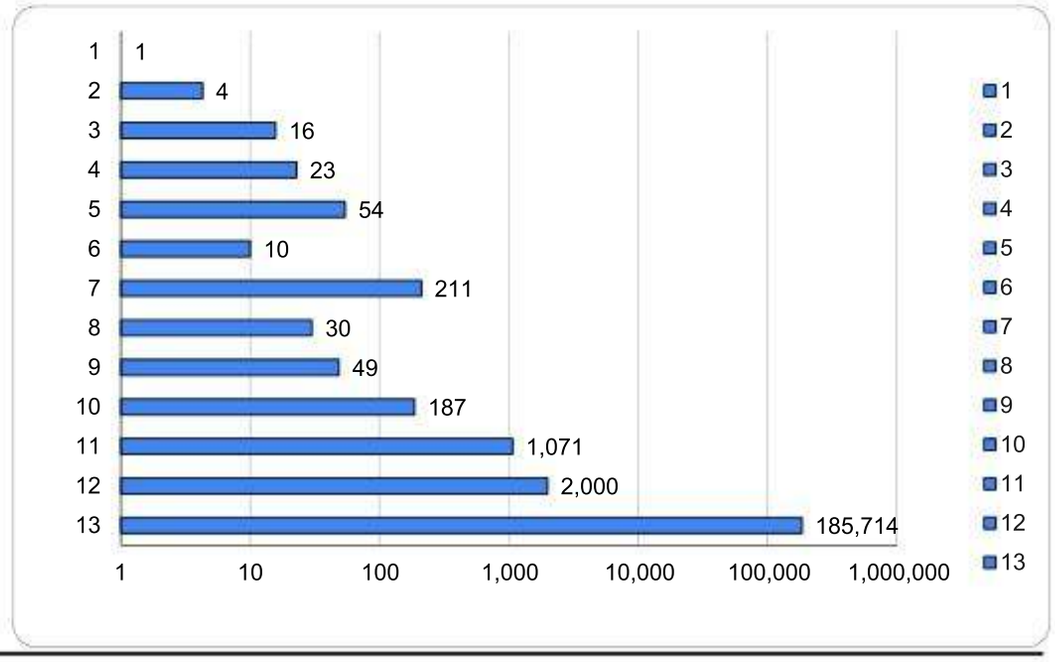

Figure 1.13 Comparison: energy cost of arithmetic operations and accesses to SRAM and DRAM for TSMC 7 nm technology node (Jouppi et al., 2021).

biggest difference is in memory; a 32-bit DRAM access takes nearly 200,000 times as much energy as an 8-bit addition. A small SRAM is 175 times more energy efficient than DRAM, which demonstrates the importance of careful use of caches and memory buffers.

The new design principle of minimizing energy per task combined with the relative energy in Figure 1.13 has inspired a new direction for computer architecture, which we describe in Chapter 7. Domain-specific processors save energy by reducing wide floating-point operations and deploying special-purpose memories to reduce accesses to DRAM. They use those savings to provide 10—1000 more (narrower) arithmetic units than a traditional processor. Although such processors perform only a limited set of tasks, they perform them remarkably faster and more energy efficiently than a general-purpose processor.

Like a hospital with general practitioners and medical specialists, computers in this energy-aware world have become combinations of general-purpose cores that can perform any task and special-purpose cores that do a few things extremely well and even more efficiently.

The Impact of Energy Use by Computers on Global Climate Change

Global climate change is a huge challenge facing society today. The rapid growth of computing has rightfully raised concerns about its carbon footprint.

Emissions can be classified as

- *Operational*, the energy cost of operating the hardware including datacenter overheads for servers, or
- *Embodied,* which additionally includes the embedded carbon emitted during the manufacturing of all components involved, from chips to datacenter buildings.

Embodied costs are much harder to calculate, as they require an accounting of emissions going recursively back to the original manufacturing sources and then accumulating the emissions from the intermediary stages through the final product.

Operational energy is

$$\text{Energy} = \text{execution time} \times \text{number of computers} \times \text{average power per computer}$$

(For data centers, there is another factor accounting for the efficiency of energy delivery to the computers inside called Power Usage Effectiveness, which is described in Chapter 6.)

Energy is measured in kilowatt-hours and emissions as kilograms of *carbon dioxide equivalent emissions* (CO_2e), which includes greenhouse gasses like methane. The emissions formula is

$$CO_2e = \text{Energy in KWh} \times CO_2e \text{ per KWh}$$

While the energy is relatively easy to calculate, the last term is hard to discover as it varies by factors of 5—10 between geographic regions (Patterson et al., 2022) depending on the primary energy source. Coal is the dirtiest energy provider, natural gas is cleaner but still includes carbon, and solar, wind, nuclear, and hydroelectric are examples of carbon-free energy sources. Using Google's energy sources as an example, in 2020 only 4% of energy used in Singapore data centers was carbon free, while data centers in Finland used 94% carbon-free energy. Even within a single country it can vary significantly: in the United States, 19% of the Nevada data center energy use was carbon free, while it was 93% in Iowa. Google gets such high numbers by contracting from renewable energy sources to reduce the carbon emissions of electricity from the local grid. In Iowa's case there is ample wind energy available day and night.

Computer architects can help reduce the carbon footprint of information technology by increasing the operational energy efficiency of the computers they design and by learning how to reduce the embodied carbon footprint during the manufacturing of their designs.

1.6 Trends in Cost

Although costs tend to be less important in some computer designs—specifically supercomputers—cost-sensitive designs are of growing significance. Indeed, in the past 40 years, the use of technology improvements to lower cost, as well as to 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. Nevertheless, it's essential for computer architects to have an understanding of cost and its factors to make intelligent decisions about whether a new feature should be included in designs where cost is an issue. (Imagine building 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 Commoditization

The cost of a manufactured computer component decreases over time, even without significant 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 gigabyte of DRAM has dropped over the long term. Since DRAMs tend to be priced in close relationship to cost—except for 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.

Volume is a second key factor in determining cost. Increasing volumes affect cost in several ways. First, they decrease the time needed to get through the learning curve, which is partly proportional to the number of systems (or chips) manufactured. Second, volume decreases cost because it increases purchasing and manufacturing efficiency. As a rule of thumb, some designers estimate that costs decrease about 10% for each doubling of volume. Moreover, volume decreases the amount of development costs that must be amortized by each computer, thus allowing cost and selling price to be closer and still make a profit.

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, Flash memory, monitors, and keyboards. In the past 40 years much of the personal computer industry has become a commodity business focused on building desktop and laptop computers often running Microsoft Windows.

Because many commodity vendors ship virtually identical products, the market 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 rivalry has led to the low end of the computer business being able to achieve better price-performance than other sectors and has 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, Flash memory, 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. Indeed, with PMDs' reliance on *systems on a chip* (SOCs), the cost of the integrated circuits is much of the cost of the PMD. 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 1.14–1.16). Therefore 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 initial term of the chip cost equation is its sensitivity to die size, shown below.

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 and for a die that is 1.0 cm on a side.

Answer When die area is 2.25 cm^2:

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

Because the area of the larger die is 2.25 times bigger, there are roughly 2.25 as many smaller dies per wafer (640/270 = 2.37):

$$\text{Dies per wafer} = \frac{\pi \times (30/2)^2}{1.00} - \frac{\pi \times 30}{\sqrt{2 \times 1.00}} = \frac{706.9}{1.00} - \frac{94.2}{1.41} = 640$$

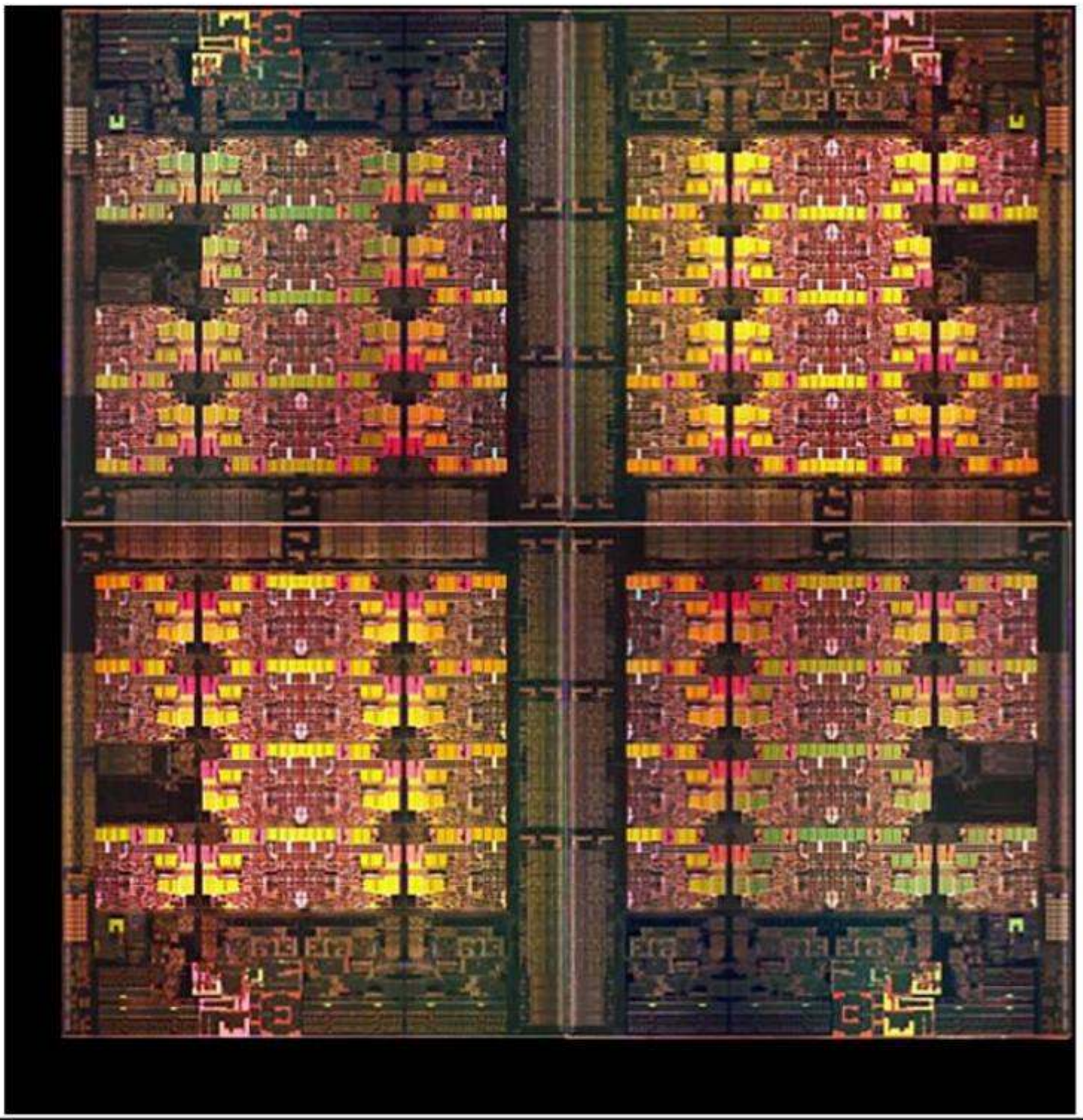

Figure 1.14 Photograph of an Intel Sapphire Rapids microprocessor die, which is evaluated in Chapter 4. It is about 400 mm^2.

However, this formula gives only the maximum number of dies per wafer. The critical question is: What is the fraction of *good* dies on a wafer, 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:

$$\text{Die yield} = \text{Wafer yield} \times 1/(1 + \text{Defects per unit area} \times \text{Die area})^{N}$$

This Bose-Einstein formula is an empirical model developed by looking at the yield of many manufacturing lines (Sydow, 2006), and it still applies today. *Wafer yield* accounts for wafers that are completely bad and so individual dies 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 2022

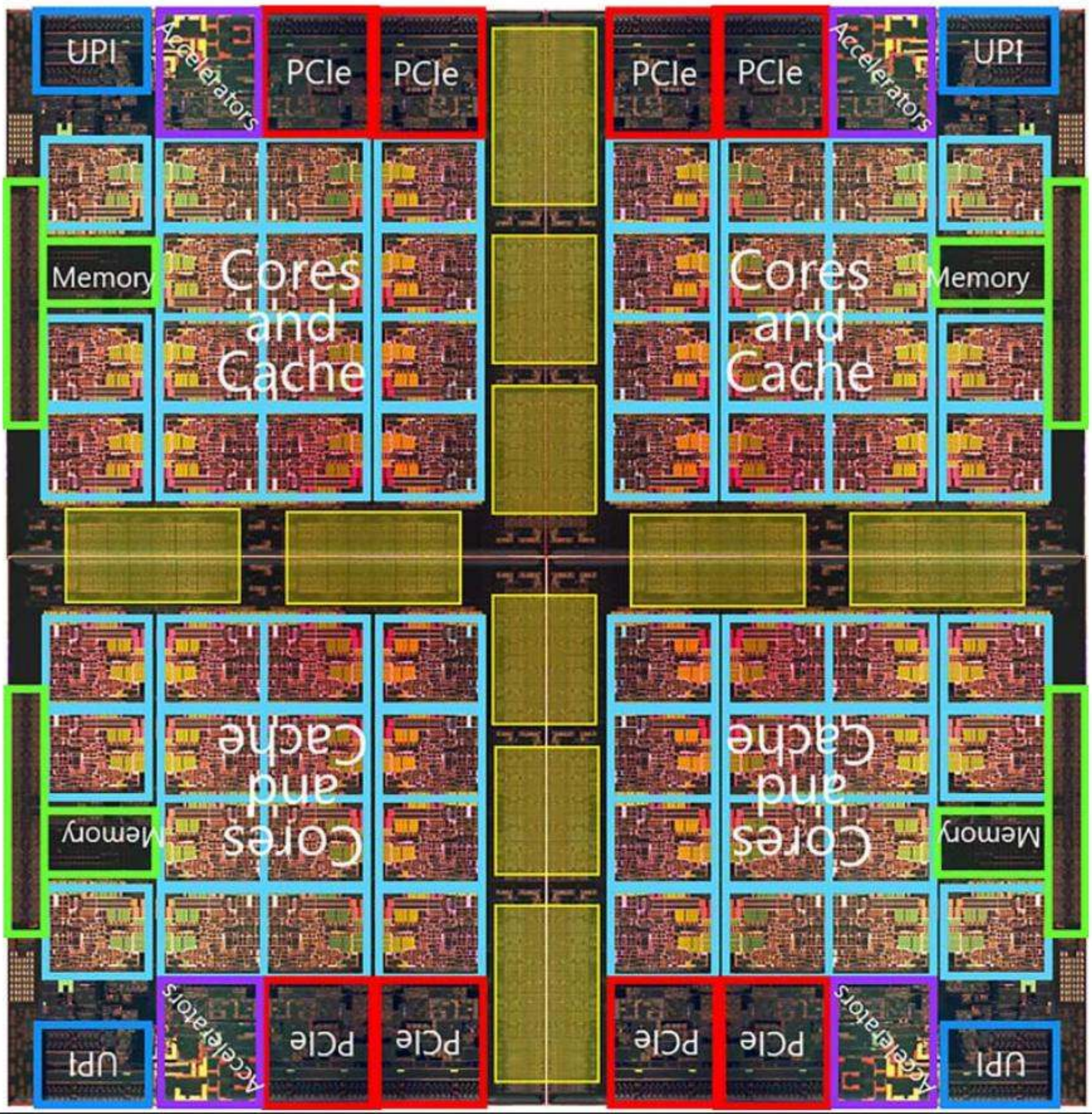

Figure 1.15 Microprocessor die components in Figure 1.14 are labeled with their functions. *UPI* (Ultra Path Interconnect) is an Intel point-to-point interconnect between sockets. *PCIe* (Peripheral Component Interconnect Express) is a standard, high-speed serial expansion bus to GPUs, network interfaces, and storage devices. The *Memory* block is the memory controller for DDR5 DRAM. The *Accelerators* block includes hardware support for the Advance Matrix Extension (AMX), the Advance Vector Extension (AVX), an engine to accelerate cryptography and data compression/decompressions (QuickAssist Technology), and the aptly named Data Streaming Accelerator (DSA). (There is also a version of Sapphire Rapids that is packaged with High Bandwidth Memory, and the HBM interface is also in the Accelerator block.)

the value was typically 0.075 defects per square inch for both 7 nm and 5 nm. The metric version is 0.012 defects per square centimeter. Finally, N is a parameter called the process-complexity factor, a measure of manufacturing difficulty. For 7 nm processes in 2022, N is 33. For a 5 nm process, N is 35.

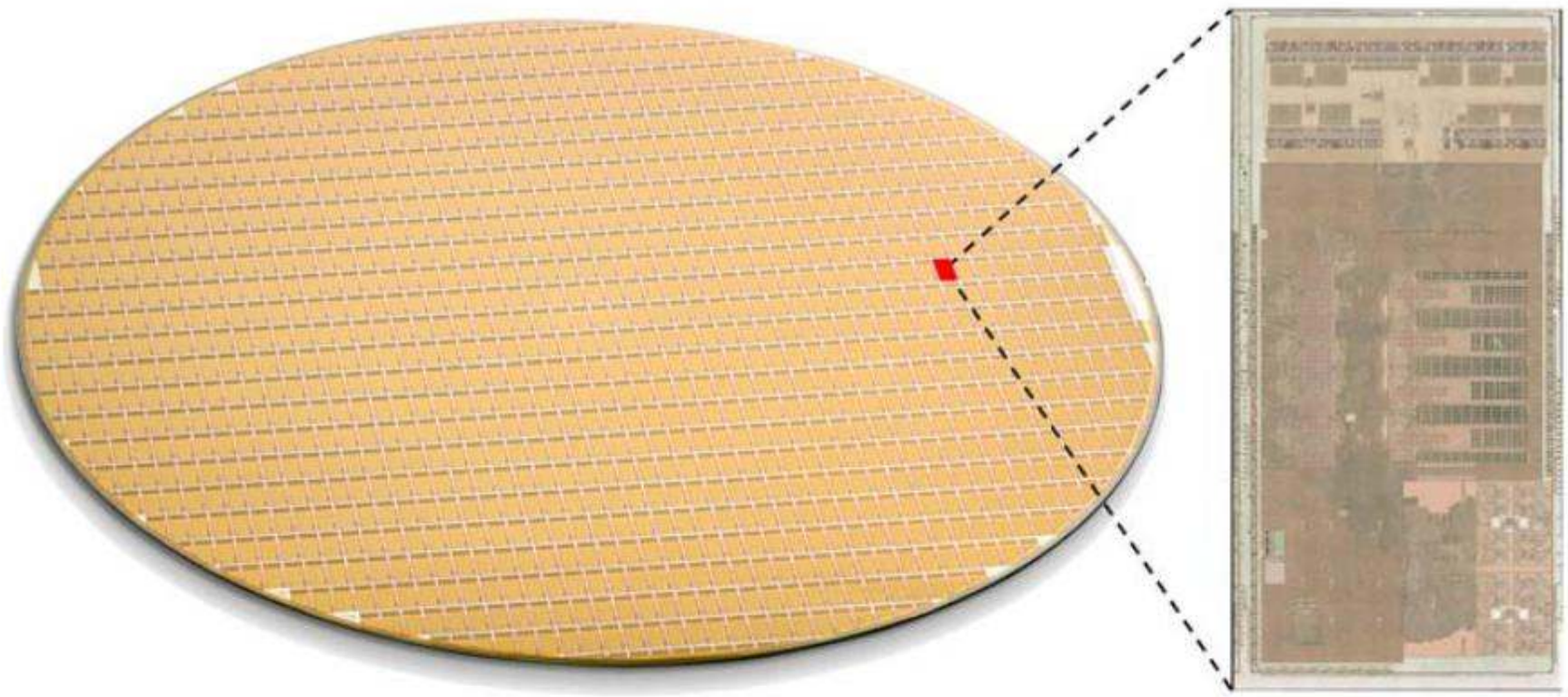

Figure 1.16 This 300 mm diameter wafer of RISC-V dies was designed by SiFive. A SiFive FU740 die is 5.58 mm × 12.17 mm. The wafer contains 1040 dies.

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 0.012 per cm^2 and N is 35.

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

$$\text{Die yield} = 1/(1 + 0.012 \times 2.25)^{35} \times 270 = 109$$

For the smaller die, the yield is

$$\text{Die yield} = 1/(1 + 0.012 \times 1.00)^{35} \times 640 = 427$$

The bottom line is the number of good dies per wafer. Only 40% of the large dies are good, but two-thirds of the small dies are good. There are about four times as many good dies for the smaller chip that is a little under half the area.

Although many microprocessors fall between 1.00 and 2.25 cm^2, low-end embedded 32-bit processors are sometimes as small as 0.05 cm^2, processors used for embedded control (for inexpensive IoT devices) are often less than 0.01 cm^2, and high-end server and GPU chips can be as large as 8 cm^2. The maximum size that a chip can be replicated is called the *reticle limit*—the area that can be exposed by the lithography equipment in a single mask step—which is around 8.5 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 large SRAM arrays used for caches within microprocessors. GPUs have 4 redundant processors out of ~100 on chip for the same reason. Obviously, the presence of redundant entries can be used to boost the yield significantly.

In 2022 processing of a 300 mm (12-inch) diameter wafer in a 7-nm technology costs about $10,000, and a 5-nm wafer costs about $16,000. Assuming a processed wafer cost of $16,000, the cost of the 1.00 cm^2 die would be around $35, but the cost per die of the 2.25 cm^2 die would be about $135.

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 control of the designer is die area and redundancy. In practice, because the number of defects per unit area is small, the number of good dies per wafer, and therefore the cost per die, grows roughly as the square of the die area. The computer designer affects die size, and thus cost, both by what functions are included on or excluded from the die and by the number of I/O pins. From an environmental perspective, the embodied carbon cost of a wafer is independent of the yield of the number of ideas on a wafer, so a great way to reduce the embodied carbon cost of dies is to increase the yield of the number of dies per wafer.

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. This testing adds significantly to the time from tape out to working chips and to the cost of building chips.

The preceding analysis 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 costs 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. Therefore, for modern high-density fabrication processes with up to 10 metal layers, mask costs are about $10 million for 7 nm and $20 million for 5 nm.

Figure 1.17 shows a breakdown of the total cost of commercial chip development and how the costs are rising significantly with recent technology nodes. These are collectively called nonrecurring engineering (NRE) costs, which are the one-time costs of new product development.

The good news is that semiconductor companies offer “shuttle runs” to dramatically lower the costs of tiny test chips. They lower costs by putting many small designs onto a single die to amortize the mask costs and then later split the dies into smaller pieces for each project. Thus TSMC delivers 80−100 untested dies that are 1 × 1 mm in a 28 nm process for $12,000 in 2022. Although these dies are tiny, they offer the architect millions of transistors to play with. For example, several RISC-V processors would fit on such a die.

Although shuttle runs help with prototyping and debugging runs, they don’t address small-volume production of tens to hundreds of thousands of parts. Because mask costs are likely to continue to increase, a new approach is gaining popularity, discussed next.

Chiplets

The prior editions of this book document the expense of going to finer geometries. In the sixth edition we listed the cost of processing a 300 mm wafer in

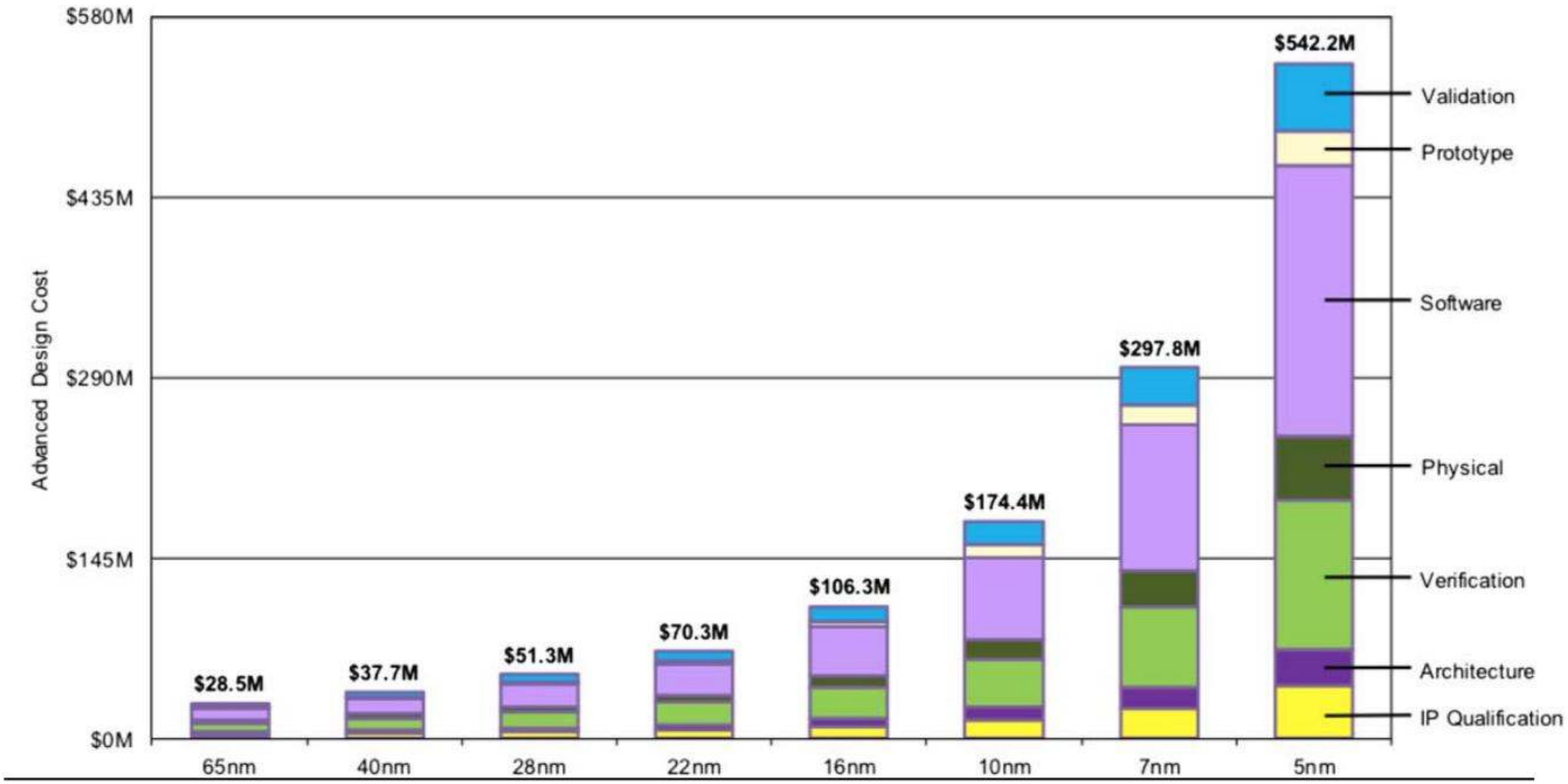

Figure 1.17 Nonrecurring engineering (NRE) costs for different technology nodes over time. Reviewing the legend bottom up, IP qualification is the cost validating the IP blocks licensed for the chip; architecture is the logical design of the chip; verification is checking the chip against the design specification; physical is the physical design of the layout of the chip; software is the low-level software that is unique to the design (drivers, firmware, diagnostics) that must be ported or reoptimized for that chip; prototype includes the boards and software to evaluate a chip; and validation is higher level than verification as it checks the hardware-software system against the actual needs of the user's end application requirements. Mask costs are included in the physical design. The 5 nm node result is a projection, while the rest are based on experience. From "As Chip Design Costs Skyrocket, 3nm Process Node Is in Jeopardy," by Joel Hruska, June 22, 2018.

28 nm as $4000 to $5000, which was similar to the price for 90 nm wafer in the fifth edition. It is ~4× more expensive at $16,000 for 5 nm. Mask costs were $1,500,000 for 28 nm in the sixth edition, which again were similar to mask costs for 90 nm in the fifth edition. At $20,000,000 for 5 nm, the costs rose 13×. Indeed, AMD found that the cost of a yielded 250 mm^2 die rose 5× from 45 nm to 5 nm (Naffziger, 2021).

The increasing mask costs have a relatively smaller impact on the cost of high-volume chips (e.g., >10M) but set a significant lower bound on costs of lower-volume chips (e.g., <1M). As we shall see in Chapter 7, the inability of general-purpose processors to make the significant gains in cost-performance that special-purpose processors can deliver led to building many instances of lower-volume chips rather than just higher volumes of a smaller number of chips. The architecture trend toward specialization conflicts with the current semiconductor economic trends favoring even higher volume manufacturing.

Moreover, while gates are improving with the latest and more expensive technologies, the other chip components are not keeping pace (see Figure 1.13). Energy per unit length of wire improves much more slowly, and SRAM density is scaling slower than in the past. Comparing 65 nm to 3 nm, Figure 1.18 shows SRAM capacity per mm^2 is ~20× less dense than ideal scaling would suggest, which earlier geometries delivered. Analog scaling is also decelerating, which means I/O gains will improve slowly past 7 nm.

As the technology trends are in the opposite direction of architecture trends, one solution is *chiplets*. Smaller dies, some in older and less expensive technology nodes, are integrated into a single package instead of the traditional solution

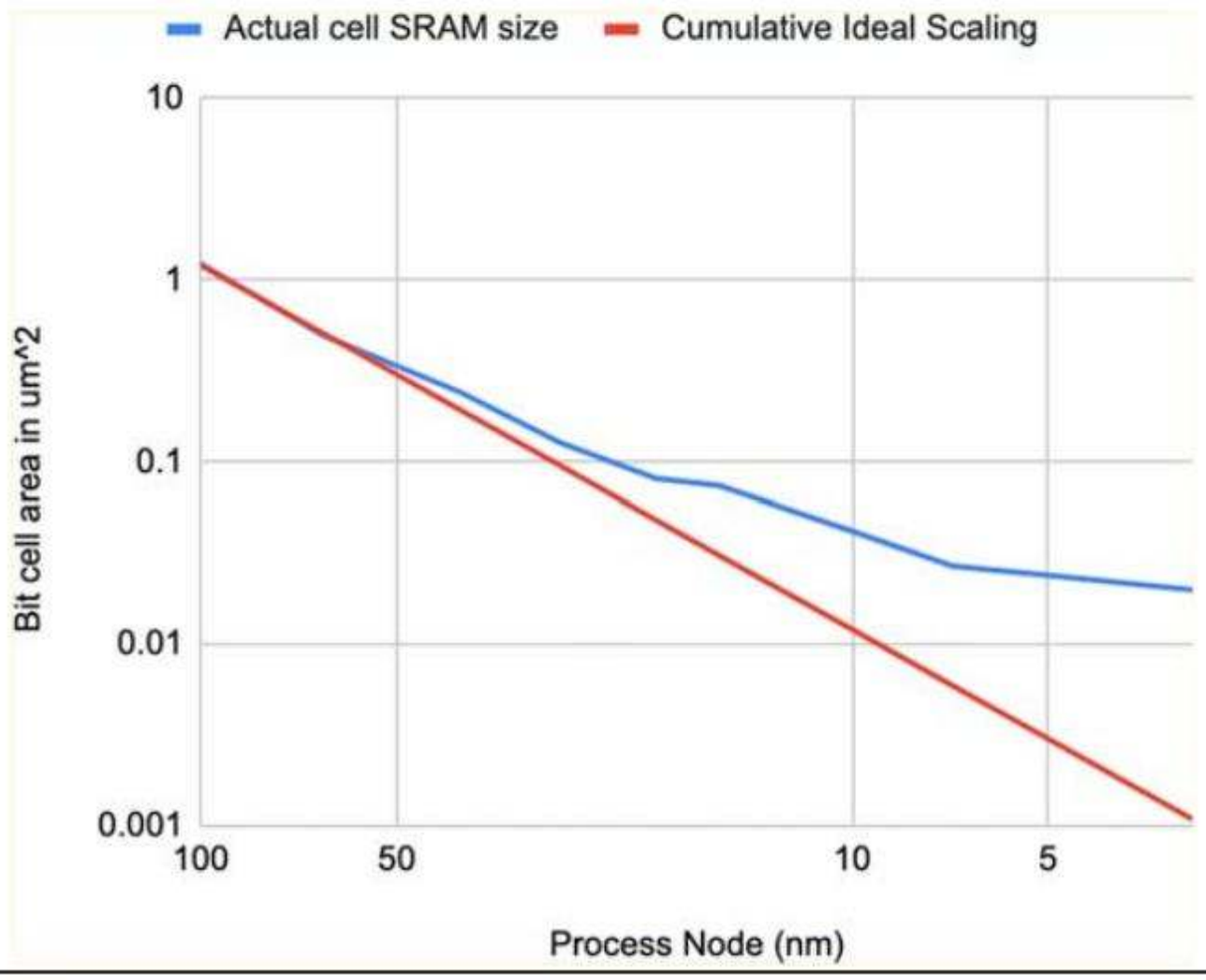

Figure 1.18 SRAM Scaling versus processing technology. SRAM is no longer scaling at the historical trends and is projected to be off from ideal scaling 5× at 7 nm, 8× at 5 nm, and 18× at 3 nm. (Norrie et al., 2021).

projected by Moore's Law of increasingly larger dies. This novel packaging solution introduces new challenges:

- Chip design is more complicated since it requires partitioning a single logical design into separate, smaller components.
- They require new chiplet-to-chiplet communication paths. Compared to on-chip communication, they are longer, have higher power or latency, and have lower bandwidth.
- Some functions may need to be replicated per chiplet that appeared only once in a single large die.

Nevertheless, a judicious choice of chiplets can lower costs and increase effective total silicon area as compared to single large chip designs.

For example, the initial AMD EPYC CPU used four chiplets in 14 nm (Naffziger, 2021). AMD projected that a single 32-core chip would need 777 mm^2 in a 14 nm process. Instead, each chiplet used 213 mm^2, so the four chiplets needed about 10% more area in total ($4 \times 213 = 852$). After factoring in yield, AMD projected a cost savings of 1.7× for chiplets. Moreover, chiplets let AMD offer both a 16-core product and a 32-core product from a single design. For the second generation EPYC Rome CPU, AMD found that since the I/O didn't improve much in 7 nm, it made more sense to consolidate those functions into a single I/O chiplet fabricated in a cheaper technology (416 mm^2 in 12 nm), while the 8-core chiplet could shrink to 74 mm^2 in 7 nm. Once again, chiplets allow AMD to offer many products from a single design: 16 cores, 24 cores, 48 cores, and 64 cores. Compared to single custom designs, AMD estimated the costs would be roughly 1.8× to 2× higher for 16- to 48-core designs, but a 64-core design would not be manufacturable since at over 1000 mm^2 it would exceed the mask reticle limit. While communicating between chiplets uses more power than communicating inside a single larger chip, that is a small amount of the power of the whole system and can be accounted for as part of a system design.

AMD, Intel, NVIDIA, and many other companies are using bespoke chiplets designed for specific products. Beyond bespoke chiplets, a potential boon for the whole industry would be to standardize on chiplets so that they could be reused across many product lines, thereby increasing their volumes and lowering relative mask costs. For those who know their chip history, the Texas Instruments TTL Data Book—with its 74xx chips during the small-scale integration phase of Moore's Law in the 1970s—is a successful example of a universal building block standard. There are many efforts underway today to try to kickstart a universal chiplet industry. If successful, it could dramatically decrease the cost and development time of custom hardware designs. Thus far, it is more a theoretical attractive possibility than a likely probability.

To avoid the confusion between chips and chiplets in systems, subsequent chapters will use the term *socket* to refer to a CPU, whether it is a single large chip or a package of chiplets.

Cost Versus Price

With the commoditization of computers, the margin between the cost to 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 pays for all engineering.

Cost of Manufacturing Versus Cost of Operation

For the first four editions of this book, cost meant the cost to build a computer and price meant price to purchase a computer. With the advent of WSCs, which contain tens of thousands of servers, the cost to operate the computers is significant in addition to the cost of purchase. Economists refer to these two costs as capital expenses (CAPEX) and operational expenses (OPEX).

As Chapter 6 shows, the amortized purchase price of servers and networks is about half of the monthly cost to operate a WSC, assuming a short lifetime of the IT equipment of 3–4 years. TCO is then CAPEX plus 3–4 times OPEX. About 40% of the monthly operational costs are for power use and the amortized infrastructure to distribute power and to cool the IT equipment, despite this infrastructure being amortized over 10–15 years. Thus, to lower operational costs in a WSC, computer architects need to use energy efficiently. Note that as Moore's Law slows, some companies are starting to stretch the lifetimes of IT equipment to 6 to 8 years.

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 failure rate inside the chip was very low. That conventional wisdom is changing for feature sizes of 5 nm and smaller, because both transient faults and permanent faults are becoming more commonplace, so architects must design systems to cope with these challenges. Memory systems have long had to deal with *transient* errors (a flipped bit value) and *permanent* errors (a bad memory cell), but we are now seeing problems in the logic inside a chip.

Indeed, researchers found a few of what they called *mercurial* cores per thousand servers (Dixit, 2021; Hochschild, 2021; Trippel, 2022). This small subset malfunctioned repeatedly when executing some instructions that could only be detected by checking the results of these instructions against the expected results. These *silent data errors* or *silent corrupt execution errors*

- are detected much more frequently than programmers expect;
- are not just incremental increases in the background rate of hardware errors;

- are in the functional logic as opposed to memory structures that are easier to discover with error detection codes;
- can appear long after the hardware is installed;
- typically affect specific cores rather than the entire multicore chip; and
- can appear suddenly and unpredictably for several reasons, including minor software changes that make heavier use of otherwise rarely used instructions.

Researchers recommend software mitigations to help cope with silent corruption errors and ask hardware manufacturers to build more robust hardware, perhaps by borrowing error-checking techniques employed by mainframe computers. One suggests improving postmanufacturing testing by enhancing scan tests and system-level tests.

This section gives a quick overview of the issues in dependability, leaving the official definition of the terms and approaches to Section D.3 in Appendix D.

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 theoretical point became concrete with the popularity of Internet services. Infrastructure providers started offering *service-level agreements* (SLAs) or *service-level objectives* (SLOs) 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 of 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. Therefore the *mean time to failure* (MTTF) is a reliability measure. Service interruption is measured as *mean time to repair* (MTTR). *Mean time between failures* (MTBF) is simply the sum of MTTF and MTTR.

The reciprocal of MTBF is a rate of failures. The relationship between *annual failure rate* (*AFR*) and MTBF (in hours) is

$$AFR = 1 - e^{-365 \times 24/MTBF}$$

If the MTBF is large, or equivalently if AFR is small, then this ratio can be approximated as

$$AFR = 365 \times 24/MTBF$$

AFR is generally reported as failures per billion hours of operation, or *FIT* (for *failures in time*). Thus, a large MTTF of 1,000,000 hours is about $10^9/10^6$ or 1000 FIT. If a collection of modules has 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 alternating 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 the 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 ATA controller, 500,000-hour MTTF
- 1 power supply, 200,000-hour MTTF
- 1 fan, 200,000-hour MTTF
- 1 ATA 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 is 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 preceding components and MTTFs, calculate the reliability of redundant power supplies. Assume that 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 the MTTF of the pair is large.

Since we have two power supplies and independent failures, the mean time until one supply 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. Therefore 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 preceding MTTF numbers, if we assume it takes on average 24 hours for a human operator to notice that a power supply has failed and to 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.

1.8 Security

While the numerous software bugs in operating systems and other systems software with privileged access (like virtual machines) are the primary vehicle of attackers of computer systems, in 2015, Google demonstrated that a user program could subvert virtual memory protection by exploiting a weakness in DDR3 DRAM chips (Seaborn, 2015). Given the two-dimensional nature of DRAM internals and the very small memory cells of DDR3 DRAMs, researchers observed that "hammering" one row of a DDR3 DRAM by writing it repeatedly could cause disturbance errors in an adjacent row, which would flip bits in the victim row. Later efforts showed that attacking two nearby rows was even more effective.

Row hammer is attacking physical memory, so it's much easier to destroy some data than to hit a particular target in virtual memory. The exact outcome of hammering was unpredictable, but multiple tries could lead to catastrophic results. A clever attacker could use the "row hammer" technique to change the protection bits of page table entries and thereby grant the program access to memory regions that the operating system is trying to protect. Later microprocessors and DRAMs include mechanisms to detect row hammer attacks so as to defeat them. Techniques for preventing directed row hammer attacks both in DRAM and in the OS are discussed in Chapter 2.

The attack stunned many security researchers who, up until then, had thought hardware was invulnerable to security issues. Although one can argue that row hammer is really a dependability problem—in that memory contains the wrong information—row hammer was just the opening volley of this new attack vector where hardware weaknesses were the target.

Timing channels have been known as a vulnerability since at least the 1970s, but most architects incorrectly considered them practically unimportant. Prior efforts exploited OS flaws or required physical presence to perform the attack or they leaked information at a very low bandwidth. However, implementation properties, such as timing, can affect function.

This pitfall was prominently exhibited by the 2018 exposure of Spectre that used microarchitecture speculation to leak private information to user-level

attacker code from user-level sandboxes, the kernel, or the hypervisor (Kocher, 2019). Spectre exploited three microarchitecture techniques:

1. **Instruction speculation:** A processor core seeks to execute dozens of instructions concurrently by speculating past branches, committing ISA changes if speculation is correct, and rolling them back when speculation is wrong. Perversely, Spectre speculatively executes instructions whose ISA changes it knows will be rolled back. Its subtle goal is to leave microarchitectural "breadcrumbs" of what the programmer thinks are hidden secrets.
2. **Caching:** Caches are invisible to an ISA. In particular, according to conventional computer architecture wisdom, which block is least recently used in a set associative cache did not matter for proper execution, so its status need not be restored on mispeculation. Spectre exploits this surprising vulnerability to place and later finds "breadcrumbs" that reveal a secret. It thus uses the contents of a cache as a "side channel" to transmit a (secret) data value.
3. **Hardware multithreading:** It is much easier to notice such subtle timing changes if the attacking program can run in close proximity to the target program. Hardware multithreading (see Chapter 3), where instructions from one program can intermix with others, simplifies this task. Hardware attacks are worrisome enough that cloud providers now offer the option to prevent sharing your server with programs of other customers. For example, AWS offers "Dedicated Instances," which cost about 5% more than the traditional shared instances.

Since the announcement of Spectre, numerous other attacks have been developed exploiting similar microarchitecture vulnerabilities.

We revisit security in more depth in Chapters 2 and 3 to describe information leaking via timing attacks in the memory hierarchy and in the processors.

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

1.9 Measuring, Reporting, and Summarizing Performance

When we say one computer is faster than another one is, what do we mean? The user of a cell phone 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 cell phone user wants to reduce *response time*—the time between the start and the completion of an event—also referred to as *execution time*. The operator of a cloud data center wants to increase *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 as fast as 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 as fast as 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 differently 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 storage accesses, memory accesses, I/O 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. Thus we need a term to consider this activity. *CPU time* recognizes this distinction and means the time the processor is computing, *not* including the time waiting for I/O or running other programs. (Clearly, the response time seen by the user is the elapsed time of the program, not the CPU time.)

Computer users who routinely run the same programs would be the perfect candidates to evaluate a new computer. To evaluate a new system, these 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. One approach is benchmark programs, which are programs that many companies use to establish the relative performance of their computers.

Benchmarks

The best choice of benchmarks to measure performance is real applications, such as Google Translate mentioned in Section 1.1. 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.
- *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. Regrettably for your authors—who dropped the fallacy about using synthetic benchmarks to characterize performance in the fourth edition of this book since we thought all computer architects agreed it was disreputable—the synthetic program Dhrystone is still one of the most widely quoted benchmarks for embedded processors today!

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 compiler 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 typically require the vendor to use one compiler and one set of flags for all the programs in the same language (such as C++ or C). The goal is to try to reduce the effect of "benchmarketing" (see Section 1.11)

Besides 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 altered version produces the same result.

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 these changes simply reduce the accuracy of the benchmarks as predictors of real performance. As we will see in Chapter 7, domain-specific architects often follow the third option when creating processors for well-defined tasks.

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 collections 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 real relative performance of two computers, particularly for programs not in the suite that customers are likely to run.

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 efforts in the late 1980s to deliver better benchmarks for workstations and servers. It was inspired at that time in part by a distaste of the inaccuracy of the increasingly popular Dhrystone benchmark. 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 many application classes. All the SPEC benchmark suites and their reported results are found at http://www.spec.org.

Although we focus our discussion on the SPEC benchmarks in many of the following sections, many benchmarks have also been developed for PCs running the Windows operating system, such as Geekbench.

A cautionary example is the Electronic Design News Embedded Microprocessor Benchmark Consortium (or EEMBC, pronounced "embassy") benchmarks.

It 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. In part because these benchmarks use small kernels, and because of the reporting options, EEMBC results are not widely quoted. This lack of success is why Dhrystone, which EEMBC was trying to replace, is sadly still used. EEMBC did develop CoreMark in 2009, which is a single synthetic program composed of four kernels, and it is now quoted as frequently as Dhrystone. Embench is a recent effort (Patterson et al., 2025) inspired by SPEC to offer a free and open benchmark suite for embedded computing whose goal is to displace Dhrystone and CoreMark as the standard embedded benchmark (see www.embench.org).

Desktop Benchmarks

Desktop benchmarks are divided 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 sixth generation: SPEC CPU2017, which follows SPEC2006, SPEC2000, SPEC95, SPEC92, and SPEC89. SPEC CPU2017 consists of a set of 10 integer benchmarks (CPU2017 Integer Speed) and 17 floating-point benchmarks (CPU2017 Floating Point Speed). Figure 1.19 describes the current SPEC CPU benchmarks and their ancestry. SPEC resolves the prior discussion about compiler flags by offering both *base* and *peak* versions of the results. Base requires

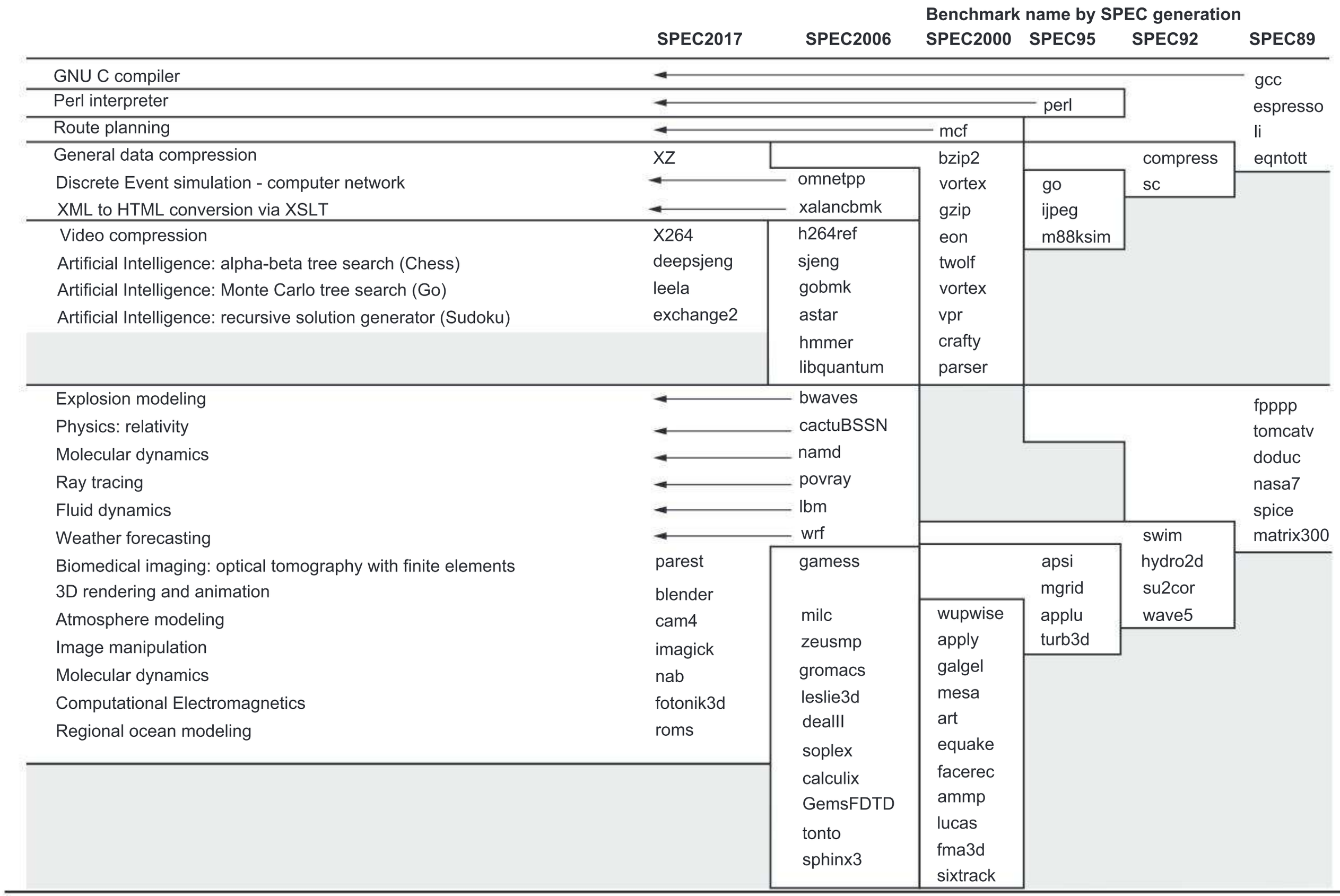

Figure 1.19 SPEC2017 programs and the evolution of the SPEC benchmarks over time, with integer programs above the line and floating-point programs below the line. Of the 10 SPEC2017 integer programs, 5 are written in C, 4 in C++, and 1 in Fortran. For the floating-point programs, the split is three in Fortran, two in C++, two in C, and six in mixed C, C++, and Fortran. The figure shows all 82 of the programs in the 1989, 1992, 1995, 2000, 2006, and 2017 releases. `gcc` is the senior citizen of the group. Only three integer programs and three floating-point programs survived three or more generations. 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 expanded to increase its running time and to avoid perturbation in measurement or domination of the execution time by some factor other than CPU time. The benchmark descriptions on the left are for SPEC2017 only and do not apply to earlier versions. Programs in the same row from different generations of SPEC are generally not related; for example, `fpppp` is not a CFD code like `bwaves`.

that all benchmarks for a given language use the same compiler flags, while peak allows the programmer to pick the best possible flags for each benchmark.

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 Go program to video compression. The floating-point benchmarks include molecular dynamics, image manipulation, and weather forecasting. The SPEC CPU suite is useful for processor benchmarking for both desktop systems and servers. We will see data on many of these programs throughout this book. However, these programs share little with modern programming languages and environments and the Google Translate application that Section 1.1 describes. A third of them are written at least partially in Fortran! They are even statically linked instead of being dynamically linked like most real programs. Alas, the SPEC2017 applications themselves may be real, but they are not inspiring. It's not clear that SPECINT2017 and SPECFP2017 capture what is exciting about computing in the 21st century.

In Section 1.11 we describe pitfalls that have occurred in developing the SPEC CPU benchmark suite and the challenges in maintaining a useful and predictive benchmark suite.

SPEC CPU2017 is aimed at processor performance, but SPEC offers many other benchmarks. Figure 1.20 lists the 17 SPEC benchmarks that are active in 2023.

Server Benchmarks

Just as servers have multiple functions, there are multiple types of benchmarks. The simplest benchmark is perhaps a processor throughput-oriented benchmark. SPEC CPU2017 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 rates called the SPEC CPU2017 Integer Rates and CPU2017 Floating Point Rates. If a measurement ran 8 copies in half the time of the reference machine, the SPECrate for that benchmark would be 16. These benchmarks are a measure of RLP from Section 1.2. To measure TLP, SPEC offers what they call high-performance computing benchmarks around OpenMP and MPI (Message Passing Interface) and for accelerators such as GPUs (see Figure 1.20).

Other than SPECrate, most server applications and benchmarks have significant I/O activity arising from either storage or network traffic, including benchmarks for file server systems, for web servers, and for database and transaction-processing (TP) systems. SPEC offers both a file server benchmark (SPECstorage) and a Java server benchmark. (Appendix D discusses some file and I/O system benchmarks in detail.) SPEC VIRT_SC 2013 and SPECvirt Datacenter 2021 evaluate end-to-end performance of virtualized data center servers. Another SPEC benchmark measures power, which we examine in Section 1.11.

TP benchmarks measure the ability of a system to handle transactions that consist of database accesses and updates. Airline reservation systems and bank

Category	Name	Measures performance of
Cloud	Cloud_IaaS 2018	Cloud using NoSQL database transaction and K-Means clustering using map/reduce
CPU	CPU2017	Compute-intensive integer and floating-point workloads
Graphics and workstation performance	SPECviewperf® 2020	3D graphics in systems running OpenGL and Direct X
	SPECworkstation 3.1	Workstations running professional apps under the Windows OS
	SPECapcSM for Maya® 2017	3D graphics running the proprietary Autodesk 3ds Max 2017 app
	SPECapcSM for PTC Creo 3.0	3D graphics running the proprietary PTC Creo 3.0 app
	SPECapcSM for Siemens NX 9.0 and 10.0	3D graphics running the proprietary Siemens NX 9.0 or 10.0 app
	SPECapcSM for SolidWorks 2021	3D graphics of systems running the proprietary SolidWorks 2021 CAD/CAM app
High performance computing	SPEC ACCEL	Accelerator and host CPU running parallel applications using OpenCL and OpenACC
	SPEChpc 2021	Four suites of HPC applications or miniapplications of increasing size, ranging from single nodes to hundreds of nodes.
	SPEC MPI 2007	MPI-parallel, floating-point, compute-intensive programs running on clusters and SMPs
	SPEC OMP 2012	Parallel apps running OpenMP
Java client/server	SPECjbb2015	Java servers
	SPECjEnterprise 2018 Web Profile	Java EE Web Profile V7 or later application servers, databases and supporting infrastructure.
	SPECJEnterprise 2010	Java EE Web Profile V5 or later application servers, databases and supporting infrastructure.
	SPECjvm 2008	Java Runtime Environment real applications and benchmarks focusing on coreJava functionality.
Power	SPECpower_ssj2008	Power of volume server class computers running SPECjbb2015. Several other SPEC suites also measure power: ACCEL, CPU 2017, OMP, 2012, and VIRT_SC 2013.
Storage	SPECstorage Solution 2020	File server throughput and response time
Virtualization	SPEC VIRT_SC 2013	Datacenter servers used in virtualized server consolidation
	SPECvirt Datacenter 2021	Third generation of benchmark of virtualized environments on multiple hosts in data centers.

Figure 1.20 Active benchmarks from SPEC as of 2022.

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 http://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. The TPC-DI benchmark, a new data integration (DI) task also known as ETL (standing for Extract, Transform, Load), is an important part of data warehousing. TPC-DS is a decision support benchmark. TPC-E is an online transaction processing (OLTP) workload that simulates a brokerage firm's customer accounts.

Recognizing the controversy between traditional relational databases and "NoSQL" storage solutions, TPCx-HS measures systems using the Hadoop file system running MapReduce programs, and TPC-DS measures a decision support system that uses either a relational database or a Hadoop-based system. TPC-VMS and TPCx-V measure database performance for virtualized systems, and TPC-Energy adds energy metrics to all the existing TPC benchmarks. The TPC has recently branched out to also offer benchmarks on Internet of Things and Artificial Intelligence.

All the TPC benchmarks measure performance in transactions per second. In addition, they include a response time requirement so that throughput performance is measured only when the response time limit is met. To model real-world systems, higher transaction rates are also associated with larger systems, in terms of both users and the database to which the transactions are applied. Finally, the system cost for a benchmark system must be included as well to allow accurate comparisons of cost-performance. TPC modified its pricing policy so that there is a single specification for all the TPC benchmarks and to allow verification of the prices that TPC publishes.

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 the 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, because it must include results of a benchmarking audit and cost information. These reports are excellent sources for finding the real costs of computing systems, since manufacturers compete on high performance and cost-performance.

Summarizing Performance Results

In practical computer design, one must evaluate myriad design choices for their relative quantitative benefits across a suite of benchmarks believed to be relevant. Likewise, consumers trying to choose a computer may rely on performance measurements from benchmarks, which ideally are similar to the users' 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 so that variability in performance can be understood. In the best 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 want to be able to summarize the suite's performance results in a single number. A simple approach to computing a summary result would be to compare the arithmetic means of the execution times of the programs in the suite. 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. One approach is to use weights that make all programs execute an equal time on some reference computer, but this biases the results toward 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 matches the way we benchmark computer performance throughout this text—namely, comparing performance ratios. For example, suppose that the SPECRatio of computer A on a benchmark is 1.25 times as fast as computer B; then we 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 1.21 gives an example.

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

Benchmarks	Sun UltraSPARC-IV+ time (seconds)	AMD EPYC 7763 time (seconds)	SPEC CPU2017 Ratio	Intel Xeon Platinum 8380 time (seconds)	SPEC CPU2017 Ratio	AMD/Intel times (seconds)	Intel/AMD SPEC ratios
perlbench	1,774	257	6.90	253	7.01	1.02	1.02
gcc	3,976	310	12.83	371	10.72	0.84	0.84
mcf	4,721	236	20.00	240	19.67	0.98	0.98
omnetpp	1,630	199	8.19	138	11.81	1.44	1.44
xalancbmk	1,417	103	13.76	108	13.12	0.95	0.95
x264	1,763	107	16.48	105	16.79	1.02	1.02
deepsjeng	1,432	231	6.20	248	5.77	0.93	0.93
leela	1,703	310	5.49	362	4.70	0.86	0.86
exchange2	2,939	131	22.44	156	18.84	0.84	0.84
xz	6,182	253	24.43	255	24.24	0.99	0.99
Geometric mean			11.99		11.69	0.98	0.98

Figure 1.21 SPEC CPU2017 Integer Speed execution times (in seconds) for the Sun UltraSPARC-IV+—the reference computer of SPEC CPU2017—and execution times and SPECRatios for the AMD EPYC 7763 and Intel Xeon Platinum 8380. The final two columns show the ratios of execution times and SPEC ratios. 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 SPEC ratios, and the ratio of the geometric means (11.69/11.99 = 0.98) is identical to the geometric mean of the ratios (0.98).

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

For 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.

Therefore 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 does not matter.

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 1.21 demonstrates this validity using examples from SPEC2017.

1.10 Quantitative Principles of Computer Design

Now that we have seen how to define, measure, and summarize performance, cost, dependability, energy, 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 and two equations to evaluate alternatives.

Take Advantage of Parallelism

Using 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 here, which are expounded on in later chapters.

Our first example is using system-level parallelism. To improve the throughput performance on a typical server benchmark, such as SPECStorage or TPC-C, multiple processors and multiple storage devices can be used. The workload of handling requests can then be spread among the processors and storage devices, resulting in improved throughput. Being able to expand memory and the number of processors and storage devices is called *scalability*, and it is a valuable asset for servers. Spreading of data across many storage devices for parallel reads and writes enables DLP. SPECstorage also relies on RLP to use many processors, whereas TPC-C uses TLP for faster processing of database queries.

Parallelism can also be exploited at the level of detailed digital design. For example, memory systems use multiple banks that allow multiple accesses to occur in parallel to different banks to increase memory bandwidth. Arithmetic-logical units use carry-lookahead, which uses parallelism to speed the process of computing sums from linear to logarithmic in the number of bits per operand. These are more examples of *DLP*.

Take Advantage of Pipelining

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. (Pipelining is explained in more detail in Appendix C and is a major focus of Chapter 3.) The basic idea behind pipelining is to overlap instruction execution to reduce the total time to complete an instruction sequence. A key insight into pipelining is that not every instruction depends on its immediate predecessor, so executing the instructions completely or partially in parallel may be possible. Pipelining is the best-known example of ILP.

Take Advantage of Prediction

Following the saying that it can be better to ask for forgiveness than ask for permission, the next opportunity is prediction. In some cases it can be faster on average to guess and start working rather than wait until you know for sure, if the mechanism to recover from a misprediction is not too expensive and your prediction is relatively accurate. Prediction is used in memory hierarchies (see Chapter 2) and extensively in modern pipelined processors (see Chapter 3), but it can also be a source of security problems (see Section 1.8).

Take Advantage of the 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 soon 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 soon. *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 2.

Deliver Dependability via Redundancy

Computers not only need to be fast, but they also need to be dependable. Since any physical device can fail, we make systems dependable by including redundant components that can take over when a failure occurs *and* to help detect failures, like the redundant power supply example above. Memory and storage have long depended upon redundancy to deliver dependability, but silent data corruption (Section 1.7) suggests that these techniques will become important for other components of the computer.

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, because the impact of the improvement is higher if the occurrence is commonplace.

Focusing on the common case works for energy 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 storage devices for every processor, storage dependability will likely dominate system dependability.

In addition, the common 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 emphasis 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 following 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 compared 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 40 seconds of the execution time of a program that takes 100 seconds in total can use an enhancement, the fraction is 40/100. This value, which we 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, 4 seconds for a portion of the program, while it is 40 seconds in the original mode, the improvement is 40/4 or 10. We 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 old 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

$$\text{Fraction}_{\text{enhanced}} = 0.4;\ \text{Speedup}_{\text{enhanced}} = 10;\ \text{Speedup}_{\text{overall}} = \frac{1}{0.6 + \frac{0.4}{10}}$$

$$= \frac{1}{0.64} \approx 1.56$$

Amdahl's Law expresses the law of diminishing returns: The incremental improvement in speedup gained by an 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 usable only for a fraction of a task, then 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 square root (FSQRT) vary significantly in performance, especially among processors designed for graphics. Suppose FSQRT is responsible for 20% of the execution time of a critical graphics benchmark. One proposal is to enhance the FSQRT 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; assume floating-point instructions are responsible for half of the execution time for the application. The design team believes that they can make all floating-point 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:

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

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

Improving the performance of the floating-point 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 39 after improving the reliability of the power supply via redundancy from 200,000-hour to 830,000,000-hour MTTF, or 4150× 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 4150× improvement in reliability of one module, from the system's perspective, the change has a measurable but small benefit.

In the preceding examples 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 *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 in 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 IC, 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 the IC in the preceding formula, clock cycles can be defined as IC $\times$ 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 on three characteristics: clock cycle (or rate), CPI, and IC. Furthermore, CPU time is

equally dependent on these three characteristics; for example, 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 ISA
- *Instruction count*—ISA and compiler technology

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

In designing the processor, sometimes it is useful 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 the 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

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

and overall CPI as

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

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

Consider our performance example on page 52, 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 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 FSQRT = 2%

CPI of FSQRT = 20

Assume that the two design alternatives are to decrease the CPI of FSQRT to 2 or to decrease the average CPI of all floating-point 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 IC remain identical. We start by finding the original CPI with neither enhancement:

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

$$= (4 \times 25\%) + (1.33 \times 75\%) = 2.0$$

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

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

$$= 2.0 - 2\% \times (20 - 2) = 1.64$$

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

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

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

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

$$= \frac{\text{CPI}_{\text{original}}}{\text{CPI}_{\text{new FP}}} = \frac{2.00}{1.625} = 1.23$$

Happily, we obtained this same speedup using Amdahl's Law on page 51.

It is often possible to measure components of the processor performance equation. Such isolated measurements are 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 IC and the CPI for each of the instructions in the set. Since the starting point is often individual IC 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 we know the default clock speed. The challenge lies in discovering the IC or the CPI. Most processors include counters for both instructions executed and clock cycles. By periodically monitoring these counters, it is also possible to attach execution time and IC to segments of the code, which can be helpful to programmers trying to understand and tune the performance of an application. Often designers or programmers 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, the simulation techniques used are like those for processors that are being designed.

Techniques that help with energy efficiency, such as dynamic voltage frequency scaling and overclocking (see Section 1.5), make this equation harder to use, because the clock speed may vary while we measure the program. A simple approach is to turn off those features to make the results reproducible. Fortunately, as performance and energy efficiency are often highly correlated—taking less time to run a program generally saves energy—it's probably safe to consider performance without worrying about the impact of DVFS or overclocking on the results.

1.11 Putting It All Together: Performance, Price, and Power

In the "Putting It All Together" sections that appear near the end of every chapter, we provide real examples that use the principles in that chapter. In this section, we look at measures of performance and power-performance in small servers using the SPECpower and SPECCPU benchmarks.

Figure 1.22 shows the two multiprocessor servers we are evaluating along with their price. To keep the price comparison fair, all are Dell PowerEdge servers. The first is the PowerEdge R650, which is based on the Intel Xeon Platinum 8380 microprocessor with a clock rate of 2.30 GHz and a Turbo mode of 3.4 GHz. Unlike the microprocessor in Chapters 2–5, this Intel chip has 40 cores and a 256 MB L3 cache. We selected a two-socket system—so 80 cores total—with 256 GB of 3200 MHz DDR4 DRAM. The next server is the PowerEdge R7525, which is based on the AMD EPYC 7763 microprocessor with a 2.45 GHz clock and a 3.5 GHz Turbo mode. It has the same number of sockets and same size and type of DRAM. The AMD microprocessor uses chiplets, which allows it to have 64 cores per socket—128 cores total—and a 256 MB L3 cache. For the SPECpower benchmark, both are running the Oracle Java HotSpot version 18.9 Java Virtual Machine

		price	% Total		price	% Total	R7525/R650
Base server	Dell PowerEdge R650	$5,049	21%	Dell PowerEdge R7525	$3,989	11%	0.79
Power supply	850 W	--		750 W	--		0.88
Ethernet	1 Gb/sec	--		1 Gb/sec	--		1.00
Processor	Intel Xeon Platinum 8380	$10,726	44%	AMD EPYC 7763	$22,455	64%	2.09
Clock rate	2.30 GHz	--		2.45 GHz	--		1.07
Turbo rate	3.4 GHz	--		3.5 GHz	--		1.03
Sockets	2	--		2	--		1.00
Cores/socket	40	--		64	--		1.60
DRAM	256	$8,277	34%	256	$8,277	23%	1.00
Disk	480GB SSD	$509	2%	480GB SSD	$509	1%	1.00
SUES Linux OS		$3,619	15%		$3,715	11%	1.03
Total		$24,561	100%		$35,230	100%	1.43
CPU2017 Integer Speed	11.7			12.0			1.03
CPU2017 Floating Point Speed	228			248			1.09
CPU2017 Integer Rates	533			822			1.54
CPU2017 Floating Point Rates	449			634			1.41
SPEC Power Max ssj_ops	7,923,820			12,339,147			1.56

Figure 1.22 Two Dell PowerEdge servers being measured, their prices as of June 2022, and their performance. We calculated the cost of the processors by subtracting the cost of a second processor. Hence the base cost of the server is adjusted by removing the estimated cost of the default processor and memory. Chapter 5 describes how these multisocket systems are connected together. Since Dell did not submit SPECPower results for these servers, we used SPECPower ratings for the servers from Fujitsu (for Intel) and Lenovo (for AMD) that used identical microprocessors, memory systems, and clock rates.

(JVM) and the SUSE Linux Enterprise Server version 5.3 operating system. Dell charges about twice as much for the 64-core AMD microprocessor than the 40-core Intel microprocessor, which results in the R7525 price being ~1.4 times the R650 price.

Note that due to the forces of benchmarking (see Section 1.9), these are unusually configured servers. The systems in Figure 1.22 have little memory relative to the amount of computation, and just a tiny 480 GB solid-state disk. It is inexpensive to add cores if you don't need to add commensurate increases in memory and storage!

Rather than run statically linked C programs of SPEC CPU, SPECpower uses a more modern software stack written in Java. It is based on SPECjbb, and it represents the server side of business applications, with performance measured as the number of transactions per second, called *ssj_ops* for *server-side Java operations per second*. It exercises not only the processor of the server, as does SPEC CPU, but also the caches, memory system, and even the multiprocessor interconnection system. In addition, it exercises the JVM, including the JIT runtime compiler and garbage collector, and portions of the underlying operating system.

The last five rows of Figure 1.22 show performance using SPEC CPU and SPECpower. The latency-oriented CPU2017 Integer Speed and Floating-Point Speed results are close, with the AMD system 1.03 and 1.09 times as fast,

respectively, but the throughput-oriented SPEC CPU2017 Integer Rates, Floating Point Rates, and maximum SPECpower ssj_ops are ~1.4, ~1.5, and ~1.6 times as fast, respectively, which is similar to the ~1.6 ratio of the number of cores.

While most benchmarks (and most computer architects) care only about performance of systems at peak load, computers rarely run at peak load. Indeed, Figure 6.35 in Chapter 6 shows the results of measuring the utilization of tens of thousands of servers over 6 months at Google, and less than 1% operate at an average utilization of 100%. The majority have an average utilization of 10% to 50%. Thus, the SPECpower benchmark captures power as the target workload varies from its peak in 10% intervals all the way to 0%, which is called Active Idle.

Figure 1.23 plots the ssj_ops (SSJ operations/second) per watt and the average power as the target load varies from 100% to 0%. The R7525 with the AMD sockets always has the lowest power and the best ssj_ops per watt across each target workload level. Since watts = joules/second, this metric is proportional to SSJ operations per joule:

$$\frac{\text{ssj\_operations/second}}{\text{Watt}} = \frac{\text{ssj\_operations/second}}{\text{Joule/second}} = \frac{\text{ssj\_operations}}{\text{Joule}}$$

To calculate a single number to use to compare the power efficiency of systems, SPECpower uses

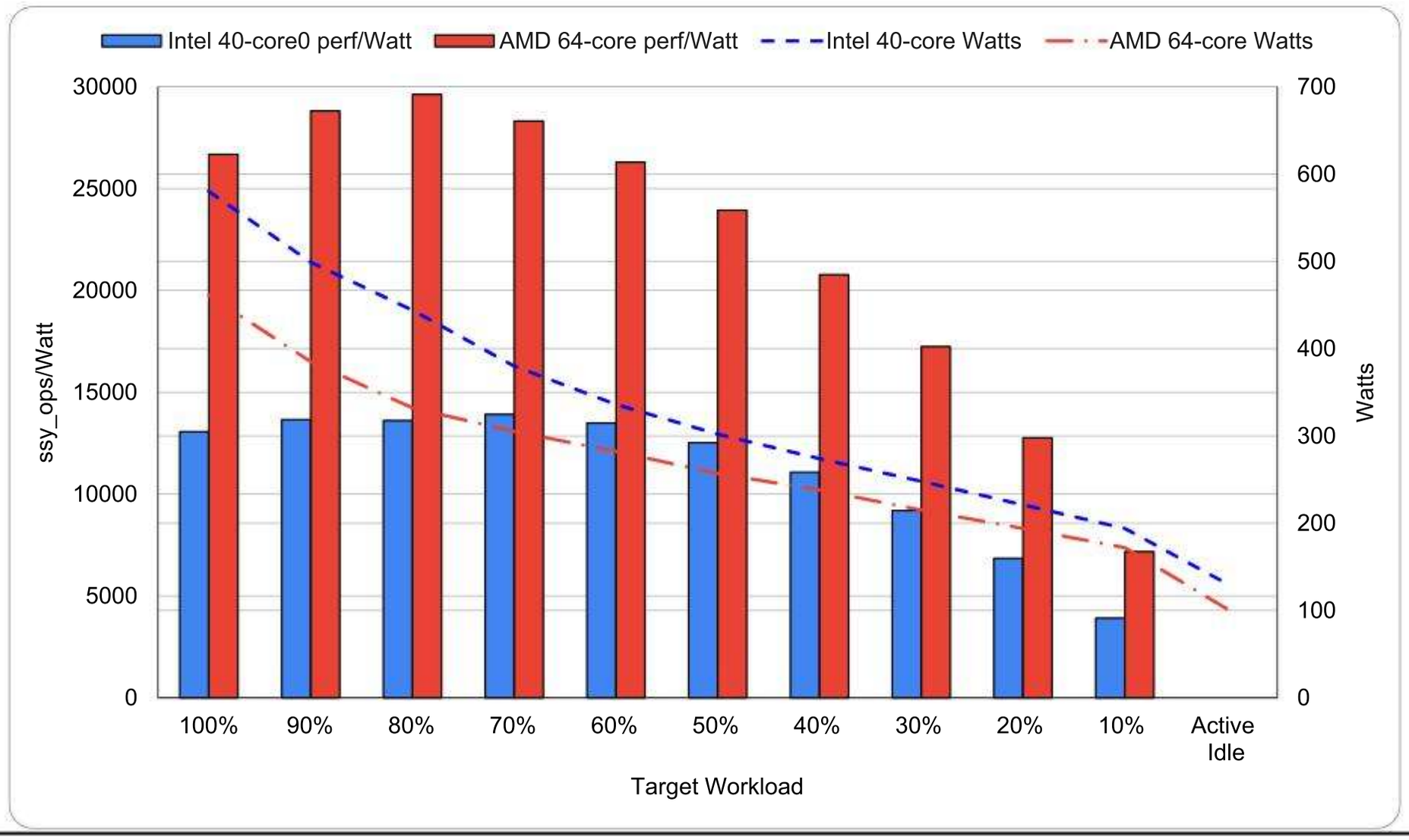

Figure 1.23 Power-performance of the two servers in Figure 1.20. Ssj_ops/watt values are on the left axis, with the columns associated with it. Watts are on the right axis, with the lines associated with it. The horizontal axis shows the target workload, as it varies from 100% to Active Idle.

$$\text{Overall ssj\_ops/watt} = \frac{\sum \text{ssj\_ops}}{\sum \text{power}}$$

The overall ssj_ops/watt is 11,533 for the R650 with the Intel socket and 23,036 for the R7525, which is twice the performance per watt at 1.4 times the price.

1.12 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 these errors in computers that you design. The topics of this chapter are rife with examples of misbeliefs and easy-to-repeat mistakes.

Pitfall *All exponential laws must slow down.*

The first to go was Dennard scaling. Dennard's 1974 observation was that voltage and current should be proportional to the linear dimensions of a transistor. With dynamic power consumption proportional to CV^2f, his observation meant that, as CMOS technology scaled, circuits could use more transistors and operate at higher frequencies while keeping power consumption constant. Dennard scaling ended 30 years later because threshold voltage and leakage currents set a nonscaling baseline for the power consumption of each transistor. As a result, modern chips are power limited and frequency scaling has essentially stopped, making it increasingly expensive to exploit greater transistor density.

The next slowdown was hard disk drives. Although there was no law for disks, in the past 30 years the maximum areal density of hard drives—which determines disk capacity—improved by 30% to 100% per year. For the past decade, it has been less than 5% per year. Increasing capacity per drive has come primarily from adding more platters to a hard disk drive.

Next up was the venerable Moore's Law. It's been a while since the number of transistors per chip doubled every 1 to 2 years. For example, the DRAM chip introduced in 2014 contained 8B transistors, and we didn't have a 16B transistor DRAM chip in mass production until 2019, when Moore's Law predicted it would be at least a 64B transistor DRAM chip.

Figure 1.24 shows Intel estimates for new process nodes since 1999. The rate was steady at every 2 years from 180 nm to 22 nm, which stretched to 3 years for 14 nm and even longer for the next node.

Figure 1.25 shows the changes in increases in bandwidth over time for microprocessors and DRAM—which are affected by the end of Dennard scaling and Moore's Law—as well as for networks and disks. The slowing of technology

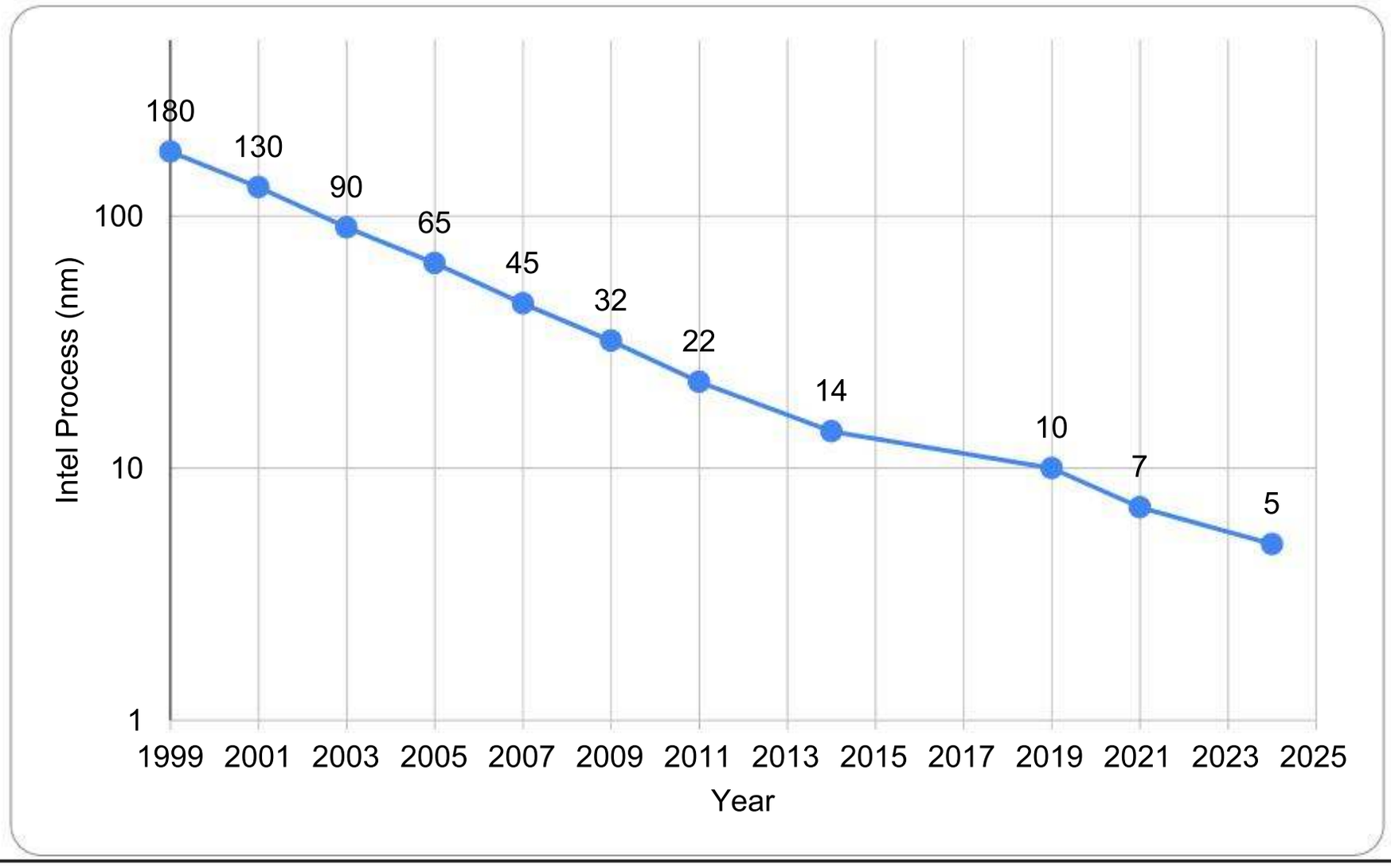

Figure 1.24 The slowing of Moore's Law. Time before new Intel semiconductor process technology measured in nm. The *y*-axis is log scale. Note that the time stretched previously from about 2 years per new process step to about 3—5 years since 2011. Source: https://en.wikichip.org/wiki/intel/process

improvements is apparent in the bending of the curves. Since around 1980 the bandwidths of microprocessors and networks have improved more than 10-fold over the improvement in memory bandwidth, which in turn is nearly another factor of 10 higher gain than the improvement in disk bandwidth.

Pitfall *Technology line widths are no longer tied to the measured width of a transistor.*

Until about 2005, the nanometer process number was a real physical measurement inside a chip. Later the traditional planar transistor offered disappointing performance at 20 nm, which resulted in the switch to the FinFET transistor at the same geometry. To indicate that the new FinFET-based transistors were better, for marketing reasons they relabeled it as 16 nm technology despite it still being in the same technology. Once the node number is no longer tied to a measurement, then anything goes. The pitch between two metal wires is 40 nm for TSMC 7 nm but 36 for Intel 10 nm, so Intel's 10 nm might yield smaller chips than TSMC's 7 nm. It's hard to predict the benefits of what are called 5 nm and smaller nodes given they are now marketing names, and they may be inconsistent between the three companies left making leading-edge logic chips: Intel, Samsung, and TSMC.

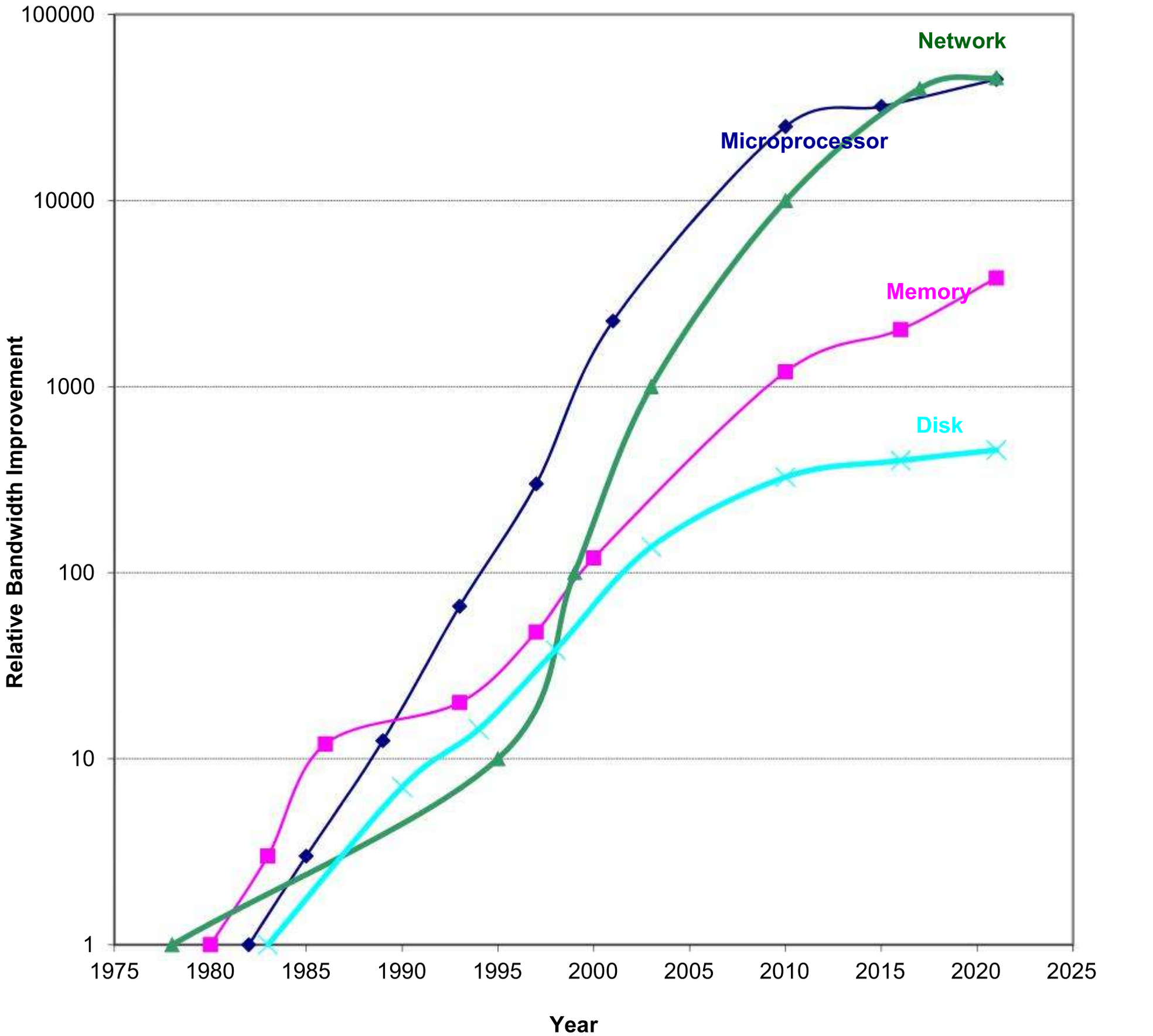

Figure 1.25 Relative bandwidth for microprocessors, networks, memory, and disks over time, based on data in Figure 1.10. Note the 10× gap between network and microprocessor versus memory bandwidth in 2021 and another nearly 10× gap between memory and disk bandwidth.

Fallacy *Multiprocessors are a silver bullet.*

The switch to multiple processors per chip around 2005 did not come from some breakthrough that dramatically simplified parallel programming or made it easy to build multicore computers. The change occurred because there was no other option due to the ILP walls and power walls. Multiple processors per chip do not guarantee lower power; it's certainly feasible to design a multicore chip that uses more power. The potential was just that it was possible to continue to improve performance by replacing a high-clock-rate, inefficient core with several lower-clock-rate, efficient cores.

As we will see in Chapters 4 and 5, performance is now a programmer's burden. The programmers' La-Z-Boy era of relying on a hardware designer to make their programs go faster without lifting a finger is officially over. If programmers want their programs to go faster with each generation, they must make their programs more parallel.

The popular version of Moore's law—increasing performance with each generation of technology—is now partly up to programmers.

Pitfall *Falling prey to Amdahl's heartbreaking 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 prior 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 53 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. Chapter 6 shows how a software layer avoids single points of failure inside WSCs.

Fallacy *Hardware enhancements that increase performance also improve energy efficiency or are, at worst, energy neutral.*

Esmaeilzadeh et al., (2011) measured SPEC2006 on just one core of a 2.67 GHz Intel Core i7 using Turbo mode (Section 1.5). Performance increased by a factor of 1.07 when the clock rate increased to 2.94 GHz (or a factor of 1.10), but the i7 used a factor of 1.37 more joules and a factor of 1.47 more watt hours! Perhaps such energy inefficiency is the reason that engineers turned off Turbo mode when they ran the SPECPower benchmarks for the two servers in Figure 1.23.

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 "benchmark engineering" or "benchmarketing." 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. Short kernels or programs that spend their time in a small amount 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 a good idea called blocking, discussed in Chapters 2 and 4), 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.

If we ignore history, we may be forced to repeat it. SPEC Cint2006 had not been updated for a decade, giving compiler writers substantial time to hone their optimizers to this suite. SPEC Cint2006 ratios of all benchmarks but libquantum fall within the range of 22 to 78 for an Intel microprocessor. Libquantum ran 7300 times faster! This "miracle" is a result of optimizations by the Intel compiler that automatically parallelizes the code across all cores and optimizes memory by using bit packing, which packs together multiple narrow-range integers to save memory space and thus memory bandwidth.

To illustrate the short lives of benchmarks, Figure 1.19 on page 43 lists the status of all 82 benchmarks from the various SPEC releases; gcc is the lone survivor from SPEC89. Amazingly, about 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.*

Current disk manufacturers' marketing practices 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 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 is the percentage of disks that fail or the annual failure rate (see Section 1.7). 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 4.5% over a 5-year lifetime. Moreover, those high numbers are quoted assuming limited ranges of temperature and vibration. If they are exceeded, all bets are off. A survey of disk drives in real environments (Gray and van Ingen, 2005) found that 3% to 7% of drives failed per year, for an MTTF of about 125,000–300,000 hours. An even larger study found annual disk failure rates of 2% to 10% (Pinheiro et al., 2007). Therefore the real-world MTTF is about 2–10 times worse than the manufacturer's MTTF.

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 1.26 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, because only performance may suffer. In processors that try to exploit ILP aggressively, 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.

The same observation is true about programs. If a register is "dead" in a program—that is, the program will write the register before it is read again—then errors in that register 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. The Sun Microsystems Division of Oracle experienced 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 were not modified, the processor would simply reread the data from the cache. Because the designers did not protect the cache with ECC (error-correcting code), the operating system had no choice but to report an error to dirty data and crash the program. Field engineers found no problems on inspection in more than 90% of the cases.

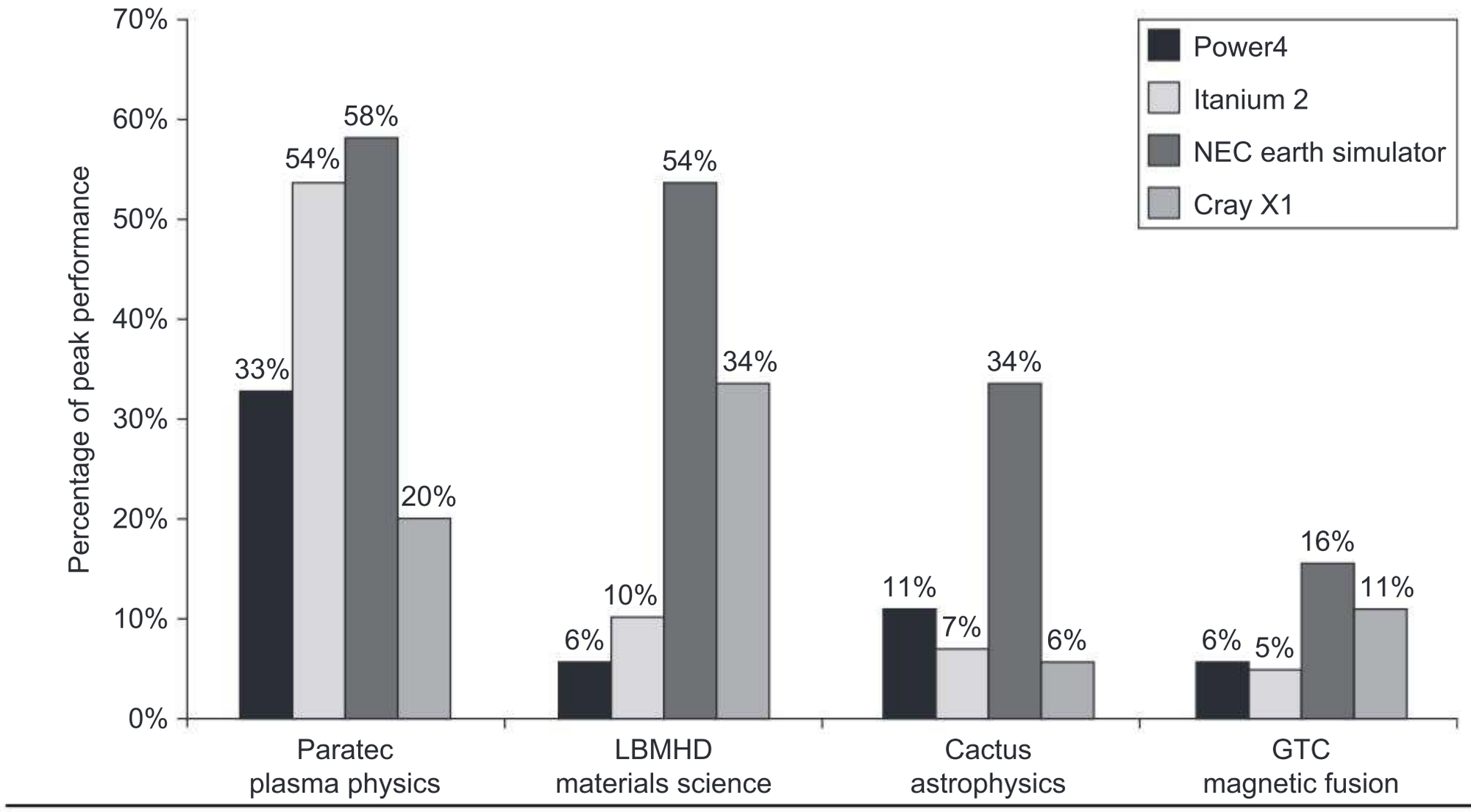

Figure 1.26 Percentage of peak performance for 4 programs on 4 multiprocessors scaled to 64 processors. The Earth Simulator and X1 are vector processors (see Chapter 4 and Appendix G). Not only did they deliver a higher fraction of peak performance, but they also had the highest peak performance and the lowest clock rates. Except for the Paratec program, the Power 4 and Itanium 2 systems delivered between 5% and 10% of their peak, respectively. From Oliker, L., Canning, A., Carter, J., Shalf, J., Ethier, S., 2004. Scientific computations on modern parallel vector systems. In Proceedings of the ACM/IEEE Conference on Supercomputing, November 6–12, 2004, Pittsburgh, PA, p. 10.

To reduce the frequency of such errors, Sun modified the Solaris operating system to "scrub" the cache by having a process that proactively wrote dirty data to memory. Because 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. These engineers are unlikely to design another computer without ECC on external caches.

Pitfall *Overestimating the impact of computer technology on carbon emissions.*

Given the importance of global climate change, it is undeniably important to consider the role of information technology. However, we must get the numbers right both to ensure that we address the most important problems and to maintain the credibility of our recommendations. Ironically, if mistakes are made in the calculations, they can become even more newsworthy and so can receive a great deal of attention. Even if subsequent publications correct the record, they do not get the same attention, so the flawed "factoids" have a life of their own on the Internet.

This irony is captured humorously by "Brandolini's Law," which states that the amount of energy needed to *refute* misinformation is 10 times greater than the amount of energy needed to *create* misinformation.

Koomey and Masanet point out four common mistakes when making emissions estimates (Koomey and Masanet, 2021):

1. Ignoring, misunderstanding, or mischaracterizing changes in key parameters over time.
2. Assuming that *short-term* changes in computing must lead to proportional and immediate changes in electricity use.
3. Given that information technology changes so quickly that even projecting a few years is highly uncertain, trying to make long-term projections.
4. Drawing broad conclusions based on trends of only one part of the information technology ecosystem.

Here are three examples of falling prey to those pitfalls:

- Between 2010 and 2018, the workloads in cloud data centers increased by a factor of 26× and estimated electricity use increased by a factor of 5×. Some used this information to predict dire consequences due to the cloud. Despite the rapid growth of the cloud, the actual global energy use of *all* data centers grew less than 10%. The reason that both observations are true is that between 2010 and 2018, a great deal of computing in the relatively inefficient local data centers was shifted to the much more efficient cloud data centers (Masanet et al., 2020). This estimate ran afoul of the fourth mistake above.
- Newspapers claimed that "the emissions generated by watching 30 minutes of Netflix [1.6 kg of CO_2e] is the same as driving almost 4 miles." A more accurate estimate is 36 grams of CO_2e, or a factor of 45 less (Kamiya, 2020). The first mistake about ignoring changes in key parameters largely explains this overestimate.
- Newspapers reported that "emissions from training a deep neural network model are five times the lifetime emissions of a car [284,019 kg]." The actual number was only 2.4 kg of CO_2e, a factor of 120,000 less (Patterson et al., 2022). The fallacies and pitfalls section of Chapter 7 explains the derivation of this flawed overestimate, largely due to being caught by the first of the four common mistakes.

1.13 Concluding Remarks

This chapter has introduced a number of concepts and provided a quantitative framework that we will expand on throughout the book. Starting with the last edition, energy efficiency is the constant companion to performance.

In Chapter 2 we start with 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. (Appendix B provides introductory material on caches for readers without much experience and background with them.) As in later chapters, we will see that hardware-software cooperation has become a key to high-performance memory systems, just as it has to high-performance pipelines.

In Chapter 3 we look at 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. Chapter 3 begins with an extensive discussion of basic concepts that will prepare you for the wide range of ideas examined in both chapters. Chapter 3 uses examples that span about nearly 50 years, drawing from one of the first supercomputers (IBM 360/91) to the fastest processors on the market today. It emphasizes what is called the *dynamic* or *runtime approach* to exploiting ILP. It also talks about the limits to ILP ideas and introduces multithreading, which is further developed in both Chapters 4 and 5. Appendix C provides introductory material on pipelining for readers without much experience and background in pipelining. (We expect it to be a review for many readers, including those of our introductory text, *Computer Organization and Design: The Hardware/Software Interface*.)

Chapter 4 explains three ways to exploit DLP. The classic and oldest approach is vector architecture, and we start there to lay down the principles of SIMD design. (Appendix G goes into greater depth on vector architectures.) We next explain the SIMD instruction set extensions found in most desktop microprocessors today. The third piece is an in-depth explanation of how modern GPUs work. Most GPU descriptions are written from the programmer's perspective, which usually hides how the computer really works. This section explains GPUs from an insider's perspective, including a mapping between GPU jargon and more traditional architecture terms.

Chapter 5 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.

Chapter 6 introduces clusters and then goes into depth on warehouse-scale computers (WSCs), which power the modern cloud computing era. The designers of WSCs are the professional descendants of supercomputer pioneers like Seymour Cray, in that they are designing extreme computers. WSCs contain tens of thousands of servers, and the equipment and the building that holds them cost nearly $200 million. The concerns of price-performance and energy efficiency of the earlier chapters apply to WSCs, as does the quantitative approach to making decisions. This chapter also covers virtual machines, which are a critical technology for secure cloud computing.

Chapter 7 introduces domain-specific architectures as the only path forward for improved performance and energy efficiency given the end of Moore's Law and Dennard scaling. It offers guidelines on how to build effective domain-specific architectures, introduces the exciting domain of deep neural networks, describes four recent examples that take very different approaches to accelerating neural networks, and then compares their cost-performance. The extraordinary excitement about AI as we complete this edition makes one wonder if general-purpose CPUs or AI accelerators will receive the most investment from now until the next edition.

This book comes with an abundance of material online (see Preface for more details), both to reduce cost and to introduce readers to a variety of advanced topics. Figure 1.27 shows them all. Appendices A to C, which appear in the book, will be a review for many readers.

In Appendix D 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. This appendix addresses the important issue of how to store and retrieve data efficiently using primarily lower-cost magnetic storage technologies. Our focus is on examining the performance of disk storage systems for typical I/O-intensive workloads, such as the OLTP benchmarks mentioned 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, Appendix D introduces queuing theory, which gives a basis for trading off utilization and latency.

Appendix E applies an embedded computing perspective to the ideas of each of the chapters and early appendices.

Appendix F explores the topic of system interconnect broadly, including wide area and system area networks that allow computers to communicate.

Appendix	Title
A	Instruction Set Principles
B	Review of Memory Hierarchies
C	Pipelining: Basic and Intermediate Concepts
D	Storage Systems
E	Embedded Systems
F	Interconnection Networks
G	Vector Processors in More Depth
H	Hardware and Software for VLIW and EPIC
I	Large-Scale Multiprocessors and Scientific Applications
J	Computer Arithmetic
K	Survey of Instruction Set Architectures
L	Advanced Concepts on Address Translation
M	Historical Perspectives and References

Figure 1.27 List of appendices.

Appendix H reviews very long instruction word (VLIW) hardware and software, which, in contrast, are less popular for general-purpose computing than when EPIC from Hewlett-Packard and Intel appeared on the scene in 1997. VLIW eventually found a home in digital signal processors and domain-specific architectures.

Appendix I describes large-scale multiprocessors for use in high-performance computing.

Appendix J is the only appendix left from the first edition, and it covers computer arithmetic.

Appendix K provides a survey of instruction architectures, including the x86, the IBM 360, the VAX, and many RISC architectures, including ARM, MIPS, Power, RISC-V, and SPARC.

Appendix L discusses advanced techniques for memory management, focusing on support for virtual machines and design of address translation for very large address spaces. With the growth in cloud processors, these architectural enhancements are becoming more important.

We describe Appendix M next.

1.14 Historical Perspectives and References

Appendix M (available online) 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 to describe significant projects. If you're interested in examining the initial development of an idea or processor or want further reading, references are provided at the end of each history. For this chapter, see Section M.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 some developers of pioneering ideas are still alive—we can learn the history by simply asking them!

Case Studies and Exercises by Gregory D. Peterson

Case Study 1: Chip Fabrication Trends

Concepts illustrated by this case study

- Fabrication Cost
- Fabrication Yield
- Defect Tolerance Through Redundancy

Even as semiconductor manufacturing pushes fundamental limits of physics, companies continue to deploy fabrication facilities to keep pace with Moore's Law. Each new fab costs tens of billions of dollars, requiring advancements in a host of related technologies to enable the manufacturing of new computer chips. These challenges directly impact the ability to create higher-performance computer chips and their price. In this case study we explore some of the design decisions in designing computer chips that will impact their performance, yield, redundancy, and cost.

1.1. [10/10/10/10/10/10] <1.6> Figure 1.28 gives hypothetical relevant data that influence the cost of GPU and CPU chips. In the next few exercises you will be exploring the effect of different possible design decisions for these chips.

a. [10] <1.6> What is the yield for the GPU A chip per good wafer?
b. [10] <1.6> What is the yield for the CPU A chip per good wafer?
c. [10] <1.6> GPU B is made by combining two large chiplets into a single packaged system. What is the yield for each GPU B chiplet per good wafer?
d. [10] <1.6> CPU B is made by combining a number of different chiplets into a packaged system. In this case there are 12 chiplet copies for compute tasks fabricated using a smaller manufacturing size and one chiplet for I/O tasks using a larger manufacturing size. What is the yield for each of the compute chiplets as well as the yield for the I/O chiplet that comprises CPU B, assuming good wafers?
e. [10] <1.6> Consider the possibility of manufacturing a single, larger chip for GPU B instead of using chiplets. How would the yield compare in this case? Can you think of other reasons that using chiplets would be a better option?
f. [10] <1.6> Consider the possibility of manufacturing a single, larger chip for CPU B instead of using chiplets using the smaller technology node (assume the I/O functionality uses the same area with the smaller technology node). How would the yield compare in this case?

1.2. [20/20/20/25/25] <1.6> Consider a manufacturing facility for each of the types of chips just discussed. We would like to evaluate how much capacity to dedicate to each type of chip. Imagine that GPU A will make a profit of $1000 per defect-free

Chip	Die size (mm^2)	Estimated defect rate (per cm^2)	N	Manufacturing size (nm)	Transistors (billions)	Cores or SMs
GPU A	800	0.011	36	4	80	132
GPU B	2 × 850	0.011	37	4	210	160
CPU A	270	0.011	36	4	16	8
CPU B	12 × 75 1 × 400	0.010 0.010	35 34	5 6	90	96

Figure 1.28 Manufacturing cost factors for hypothetical GPUs and CPUs.

chip, and CPU A will make a profit of $100 per defect-free chip. For each type of chip, assume that each wafer has a 300 mm diameter.

a. [20] <1.6> How much profit do you make on each wafer of GPU A chips?
b. [20] <1.6> How much profit do you make on each wafer of CPU A chips?
c. [20] <1.6> If your demand is up to 5000 GPU A chips per month and 3000 CPU A chips per month, and your facility can fabricate 90 wafers a month, how many wafers should you make of each chip?
d. [25] <1.6> If we assume that each GPU B will make a profit of $2500 per defect-free product (requiring two chiplets), how much profit do you make on each wafer of GPU B chips?
e. [25] <1.6> If we assume that each CPU B will make a profit of $75, per defect-free product (requiring 12 of the chiplets for compute along with one of the chiplets for I/O functions), how many CPU B products can be sold for each wafer of CPU B compute chiplets? How many wafers of CPU B I/O chiplets are needed for each wafer of CPU B compute chiplets? If your facility can fabricate 30 wafers of these chiplets, how many wafers should be for compute chiplets and how many for I/O chiplets to maximize profit? What is this profit?

1.3. [20/5] <1.6> Consider the possibility of increasing the economic viability of large processor chips with poor yields by selling partially functional chips for less instead of discarding them. For some products, different versions of chips (also known as stock keeping units, or SKUs) can be sold that include different numbers of cores. For example, you could sell the CPU A chip with versions that contain 8, 4, 2, and 1 cores on each chip. If all eight cores are defect-free, then it is sold as CPU A_8. Chips with four to seven defect-free cores are sold as CPU A_4, and those with two or three defect-free cores are sold as CPU A_2. For simplification, calculate the yield for a single core as the yield for a chip that is 1/8 the area of the original CPU A chip. Then view that yield as an independent probability of a single core being defect-free. Calculate the yield for each configuration as the probability of the corresponding number of cores being defect-free.

a. [20] <1.6> What is the yield for a single core being defect-free as well as the yield for CPU A_4, CPU A_2, and CPU A_1?
b. [5] <1.6> Using your results from (a), determine which chips you think would be worthwhile to package and sell, and explain why.

1.4. [20/5/25] <1.6> Now we consider a similar approach to improving yields for GPUs by using redundancy. In the case of GPU B assume the product is comprised of two chiplets, each of which contains 85 symmetric multiprocessors (SMs), the equivalent of cores for GPUs. Further assume that to ship products with 160 SMs, 2 functional GPU chiplets are required, each of which can have up to 5 of their SMs with defects. As in the case of problem 1.3 and CPU A, calculate the yield for a single SM as the yield for a chiplet that is 1/85 the area of the original

GPU B chiplet. Once again, view that yield as an independent probability of a single SM being defect-free. Calculate the yield for the chiplet as the probability of at least 80 SMs being defect-free.

a. [20] <1.6> What is the yield for a single SM being defect-free as well as the yield for each GPU B chiplet?
b. [5] <1.6> Using your results from (a), what is the profit from each wafer of GPU B chiplets? How does this compare to the results from 1.2(d)?
c. [25] <1.6> A similar analysis could be done for each GPU A chip, assuming there are 144 SMs on each die, and that chips with at least 132 functional SMs could be sold. How would this redundancy affect the profit for each wafer? How would this affect the answer for 1.2(c)?

Case Study 2: Scalable Computer Systems

Concepts illustrated by this case study

- Amdahl's Law
- Redundancy
- MTTF

Modern systems scale from tiny IoT devices to data centers and supercomputers. The following exercises address practical issues impacting systems such as cost, power and energy, and scalable performance.

Large, scalable computer systems typically seek to obtain high performance by addressing job throughput (handling many concurrent jobs), latency (completing tasks as quickly as possible), or both. Employing parallelism represents a key approach for improving performance spanning system scales from additional cores within a processor to increased numbers of processors/sockets within a server/node to expanded servers/nodes within a data center or supercomputer.

1.5. [10/20/20/20/25] <1.10> When parallelizing an application, the ideal speedup would be speeding the application up by the number of processors. In practice, this is limited by two things: the percentage of the application that can be parallelized and the impact of overheads such as the cost of communication. Amdahl's Law takes into account the former but not the latter.

a. [10] <1.10> What is the speedup with N processors if [80%, 90%, 95%, 99%] of the application is parallelizable, ignoring the cost of communication?
b. [20] <1.10> What is the speedup with 32 processors if, for every processor added, the communication overhead is [0.5%, 1%, 2%, 5%] of the original execution time?

c. [20] <1.10> What is the speedup with 32 processors if, for every time the number of processors is doubled, the communication overhead is increased by [0.5%, 1%, 2%, 5%] of the original execution time?
d. [20] <1.10> What is the speedup with *N* processors if, for every time the number of processors is doubled, the communication overhead is increased by [0.5%, 1%, 2%, 5%] of the original execution time?
e. [25] <1.10> Write the general equation that solves this question: What is the number of processors with the highest speedup in an application in which P% of the original execution time is parallelizable, and, for every time the number of processors is doubled, the communication is increased by [0.5%, 1%, 2%, 5%] of the original execution time?

1.6. [10/20/20/20] <1.10> In problem 1.5, the problem size being run in parallel was the same in each case, known as strong scaling. In weak scaling (sometimes referred to as *Gustafson's Law*) the problem size increases with the number of processors. Consider the performance, or scaled speedup, of a weakly scaled application considering the percentage of the serial application (baseline) that is parallelized and the cost of communication.

a. [10] <1.10> What is the scaled speedup with *N* processors if [80%, 90%, 95%, 99%] of the serial application is parallelizable, ignoring the cost of communication and otherwise assuming perfect scaling with additional processors?
b. [20] <1.10> What is the scaled speedup with 32 processors if, for every processor added, the communication overhead is [0.5%, 1%, 2%, 5%] of the original execution time?
c. [20] <1.10> What is the scaled speedup with 32 processors if, for every time the number of processors is doubled, the communication overhead is increased by [0.5%, 1%, 2%, 5%] of the original execution time?
d. [20] <1.10> What is the scaled speedup with *N* processors if, for every time the number of processors is doubled, the communication overhead is increased by [0.5%, 1%, 2%, 5%] of the original execution time?

1.7. [Discussion] <1.9, 1.10> As a benchmark for the largest supercomputers in the world, the Top 500 list (top500.org) employs High Performance LINPACK (HPL), a dense linear algebra code, to report the obtained floating-point operations per second (FLOPS) achieved along with theoretical peak FLOPS for the system. Similarly, they also use the High Performance Conjugate Gradient (HPCG), a sparse linear algebra code, to report the obtained FLOPS as well. From the June 2024 lists, the top machines averaged about 70% of the peak FLOPS performance for HPL, but less than 2% of the peak FLOPS for HPCG.

a. Can you translate the Top 500 HPL or HPCG results into speedup or scaled speedup? Explain.
b. Compare these Top 500 results to those reported for scientific codes on supercomputers as shown in Figure 1.26.
c. In a 2024 study exploring machine learning training on Frontier, then the top machine on the Top 500 list, researchers reported obtaining 32% to 38% of theoretical operations per second.[11] They also reported scaled speedups (weak scaling) of nearly 100% and speedups (strong scaling) of 87% to 89% for the training codes. How do these results compare to the cases from (a) and (b)?

1.8. [10/10/20/20] <1.10> You have been asked to optimize the performance of a mix of applications on a new processor that includes 64 cores. You will run four applications on this processor, but the resource requirements are not equal. Assume the system and application characteristics listed in Figure 1.29. This problem can be solved for any or all of the independent scenarios A to D in the figure. The percentage of resources in the figure indicates what proportion of the total runtime would be associated with each application (1−4), assuming they are all run serially. Assume that when you parallelize a portion of the program by X, the speedup for that portion is X.

a. [10] <1.10> How much speedup would result from running application 1 on the entire processor, as compared to running it serially?

Application	Application 1	Application 2	Application 3	Application 4
Scenario A				
% resources needed	35	31	20	14
% parallelizable	50	80	60	90
Scenario B				
% resources needed	41	28	17	14
% parallelizable	90	60	80	50
Scenario C				
% resources needed	37.5	25	22	15.5
% parallelizable	80	90	50	60
Scenario D				
% resources needed	50	25	12.5	12.5
% parallelizable	60	50	90	80

Figure 1.29 Four applications.

[1] See https://arxiv.org/pdf/2312.12705.pdf

b. [10] <1.10> How much speedup would result from running application 4 on the entire processor, as compared to running it serially?
c. [20] <1.10> Given the percentage of resources that application 1 requires, if we statically assign it that percentage of the cores, what is the overall speedup if application 1 is run parallelized but everything else is run serially?
d. [20] <1.10> What is the overall speedup if all four applications are statically assigned some of the cores, proportional to their percentage of resource needs, and all run parallelized?

1.9. [10/10/20/20] <1.7> Availability is the most important consideration for designing servers, followed closely by scalability and throughput.

a. [10] <1.7> We have a single processor with a failure in time (FIT) of 100. What is the mean time to failure (MTTF) for this system?
b. [10] <1.7> If it takes one day to get the system running again, what is the availability of the system?
c. [20] <1.7> Imagine that the government, to cut costs, is going to build a supercomputer out of inexpensive computers rather than expensive, reliable computers. What is the MTTF for a system with 1000 processors? Assume that if one fails, they all fail.
d. [20] <1.7> The largest supercomputers can have 10,000 or more nodes, each with multiple processors or accelerators. What is the MTTF for a system with 50,000 processors with the same reliability characteristics as above? In this case what is the longest runtime for a job that I can expect to complete before a failure?

1.10. [20/20/20] <1.2, 1.7> In a server farm such as those used by cloud vendors, a single failure does not cause the entire system to crash. Instead, it will reduce the number of requests that can be satisfied at any one time.

a. [20] <1.7> If a company has 10,000 computers, each with an MTTF of 35 days, and it experiences catastrophic failure only if 1/3 of the computers fail, what is the MTTF for the system?
b. [20] <1.2, 1.7> Assume that downtimes cost $100,000 an hour and it takes 1 day for the computers to recover after the system crashes. What would the annual losses from downtime cost?
c. [20] <1.2, 1.7> Assume that downtimes cost $100,000 an hour and it takes 1 day for the computers to recover after the system crashes. If it costs an extra $1000, per computer, to double the MTTF, would this be a good business decision? Assume the computers have a life span of 3 years.

Case Study 3: Energy and Power Considerations for Computer Systems

Concepts illustrated by this case study

- Dynamic Energy
- Power Consumption

Power consumption in modern systems depends on a variety of factors, including the chip clock frequency, efficiency, and voltage. The following exercises explore the impact on power and energy that different design decisions and use scenarios have.

1.11. [10/10/10/10] <1.5, 1.9> Cell phones perform very different tasks, including streaming music or video, reading email, and increasingly performing machine learning tasks such as image or speech recognition. Users demand long battery life for cell phones, so reductions in power and energy consumption are critical. In this problem we consider an unrealistic scenario in which the cell phone has no specialized processing units. Instead, it has a quad-core, general-purpose processing unit. Each core uses 0.5 W at full use. For its tasks, the quad-core is 8× as fast as necessary.

a. [10] <1.5> How much dynamic energy and power are required compared to running at full power? First, suppose that the quad-core operates for 1/8 of the time and is idle for the rest of the time. That is, the clock is disabled for 7/8 of the time, with no leakage occurring during that time. Compare total dynamic energy as well as dynamic power while the core is running.
b. [10] <1.5> How much dynamic energy and power are required using frequency and voltage scaling? Assume frequency and voltage are both reduced to 1/8 the entire time.
c. [10] <1.5, 1.9> Now assume the voltage may not decrease below 50% of the original voltage. This voltage is referred to as the *voltage floor*, and any voltage lower than that will lose the state. Therefore, while the frequency can keep decreasing, the voltage cannot. What are the dynamic energy and power savings in this case?
d. [10] <1.5> How much energy is used with a dark silicon approach? This involves creating specialized ASIC hardware for each major task and power gating those elements when not in use. Only one general-purpose core would be provided, and the rest of the chip would be filled with specialized units. For tasks such as email, the one core would operate for 25% of the time and be turned completely off with power gating for the other 75% of the time. During the other 75% of the time, a specialized ASIC unit that requires 20% of the energy of a core would be running for tasks such as speech recognition.

1.12. [10/10/10] <1.5> As mentioned in Exercise 1.11, cell phones run a wide variety of applications. We'll make the same assumptions as before, that it is 0.5 W per core.

a. [10] <1.5> Imagine that 80% of the code is parallelizable. By how much would the frequency and voltage on a single core need to be increased in order to execute at the same speed as the four-way parallelized code?
b. [10] <1.5> What is the reduction in dynamic energy from using frequency and voltage scaling in (a)?
c. [10] <1.5> How much energy is used with a dark silicon approach? In this approach all hardware units are power gated, allowing them to turn off entirely (causing no leakage). Specialized ASICs are provided that perform the same computation for 20% of the power as the general-purpose processor. Imagine that each core is power gated. Assume a task such as image recognition requires two ASICs and two cores. How much dynamic energy is required compared to the baseline of running in parallel on four cores?

1.13. [10/10/10/10/10/20] <1.5,1.9> General-purpose processors are optimized for general-purpose computing, so they perform well across a large number of applications. However, once the domain is restricted somewhat, optimized functional units or specialized processors can perform more efficiently. Many machine language operations can be performed concurrently, using specialized hardware like ASICs or graphical processing units (GPUs). This problem explores the trade-offs between a general-purpose processor and a GPU, in terms of performance and cooling. If heat is not removed from the computer efficiently, the fans will blow hot air back onto the computer, not cold air. Note: The differences involve more than just a processor—on-chip memory and DRAM also come into play. Therefore statistics are at a system level, not a chip level.

a. [10] <1.9> If a data center spends 70% of its time on workload A and 30% of its time on workload B when running CPUs, what is the speedup of the GPU system over the CPU system?
b. [10] <1.9> If a data center spends 70% of its time on workload A and 30% of its time on workload B when running CPUs, what percentage of Max IPS does it achieve for each of the three systems?
c. [15] <1.5, 1.9> Building on (b), assuming that the power scales linearly from idle to busy power as IPS grows from 0% to 100%, what is the performance per watt of the CPU system compared to the GPU system?
d. [10] <1.9> If another data center spends 40% of its time on workload A, 10% of its time on workload B, and 50% of its time on workload C, what is the speedup of the GPU system over the general-purpose system?
e. [10] <1.5> Assume 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 CPU- or GPU-based servers can you cool with one cooling door, assuming TDP in Figures 1.30 and 1.31?

System	TDP	Idle power	Busy power
CPU-based	504 W	159 W	455 W
GPU-based	1838 W	357 W	991 W

Figure 1.30 Hardware characteristics for general-purpose processor or graphical processing unit–based system, including measured power.

System	Throughput			% Max IPS		
	A	B	C	A	B	C
CPU-based	5482	13,194	12,000	42%	100%	90%
GPU-based	13,461	36,465	15,000	37%	100%	40%

Figure 1.31 Performance characteristics for general-purpose processor or graphical processing unit–based system on two neural-net workloads. Workloads A and B are from published results. Workload C is a fictional, more general-purpose application.

f. [20] <1.5> Typical server farms can dissipate a maximum of 200 W per square foot. Given that a server rack requires 11 square feet (including front and back clearance), how many servers from (e) can be placed on a single rack, and how many cooling doors are required?

Exercises

1.14. [15/10] <1.9> Assume that we make an enhancement to a computer that improves some mode of execution by a factor of 10. Enhanced mode is used 50% of the time, measured as a percentage of the execution time when the enhanced mode is in use. Recall that Amdahl's Law depends on the fraction of the original, unenhanced execution time that could make use of enhanced mode. Thus we cannot directly use this 50% measurement to compute speedup with Amdahl's Law.

a. [15] <1.9> What is the speedup we have obtained from fast mode?
b. [10] <1.9> What percentage of the original execution time has been converted to fast mode?

1.15. [20/10/10/15] <1.9> In this exercise assume that we are considering enhancing a chiplet-based multicore processor by adding encryption hardware to it as a substitute chiplet for a computing core chiplet. When computing encryption operations, it is 20 times faster than the normal mode of execution. We will define the percentage of encryption as the percentage of time in the original execution that is spent performing encryption operations.

a. [20] <1.9> Draw a graph that plots the speedup as a percentage of the computation spent performing encryption. Label the y-axis "Net speedup" and label the x-axis "Percent encryption."

b. [10] <1.9> With what percentage of encryption will adding encryption hardware result in a speedup of 2?
c. [10] <1.9> What percentage of time in the new execution will be spent on encryption operations if a speedup of 2 is achieved?
d. [15] <1.9> Consider the case that the loss of the computing core chiplet reduces the performance of other computations by 10%. How often must the encryption unit be used to make it worthwhile?

1.16. [20/20/15] <1.9> When making changes to optimize part of a processor, it is often the case that speeding up one type of instruction comes at the cost of slowing down something else. For example, if we put in a complicated fast floating-point unit, that takes space, and something might have to be moved farther away from the middle to accommodate it, adding an extra cycle in delay to reach that unit. The basic Amdahl's Law equation does not take into account this trade-off.

a. [20] <1.9> If the new fast floating-point unit speeds up floating-point operations by, on average, 2×, and floating-point operations take 20% of the original program's execution time, what is the overall speedup (ignoring the penalty to any other instructions)?
b. [20] <1.9> Now assume that speeding up the floating-point unit slowed down data cache accesses, resulting in a 1.5× slowdown (or 2/3 speedup). Data cache accesses consume 10% of the execution time. What is the overall speedup now?
c. [15] <1.9> After implementing the new floating-point operations, what percentage of execution time is spent on floating-point operations? What percentage is spent on data cache accesses?

1.17. [10/10/20/20] <1.5> Server farms such as those of cloud computing providers include enough compute capacity for the highest request rate of the day. Imagine that most of the time these servers operate at only 60% capacity. Assume further that the power does not scale linearly with the load; that is, when the servers are operating at 60% capacity, they consume 90% of maximum power. The servers could be turned off, but they would take too long to restart in response to more load. A new system has been proposed that allows for a quick restart but requires 20% of the maximum power while in this "barely alive" state.

a. [10] <1.5> How much power savings would be achieved by turning off 40% of the servers?
b. [10] <1.5> How much power savings would be achieved by placing 40% of the servers in the "barely alive" state?
c. [20] <1.5> How much power savings would be achieved by reducing the voltage by 20% and frequency by 20%?
d. [20] <1.5> How much power savings would be achieved by placing 20% of the servers in the "barely alive" state and 20% off?

1.18. [10/10/10/10/5] <1.5> When designing systems for real-time applications in which specific deadlines must be met, finishing the computation faster gains nothing. Consider the case of an IoT system that can execute the necessary code, in the worst case, twice as fast as necessary.

a. [10] <1.5> How much energy do you save if you execute at the current speed and turn off the system when the computation is complete?
b. [10] <1.5> How much energy do you save if you set the voltage and frequency to be half as much?
c. [10] <1.5> Consider the case in which a different IoT system has 50% more cores and requires 30% more power. How does the energy compare to (a) if you execute at the current speed and turn off the system when the computation is complete?
d. [10] <1.5> Once again considering the case of an IoT system with 50% more cores and requires 30% more power. How does the energy compare to (c) if you set the voltage and frequency to be one-third as much?
e. [5] <1.5> Is it worthwhile to use the IoT system with additional cores? Why?

1.19. [10/15/15/10/10] <1.4, 1.5> One challenge for architects is that the design created today will require several years of implementation, verification, and testing before appearing on the market. This delay means that the architect must project what the technology will be like several years in advance. Sometimes, this is difficult to do.

a. [10] <1.4> According to the trend in device scaling historically observed by Moore's Law, the number of transistors on a chip in 2030 should be how many times the number in 2020?
b. [15] <1.5> The increase in performance once mirrored this trend. Had performance continued to climb at the same rate as in the 1990s, approximately what performance would chips have over the VAX-11/780 in 2030?
c. [15] <1.5> At the current rate of increase of the late 2010s to early 2020s, what is a more updated projection of performance in 2030?
d. [10] <1.4> What has limited the rate of growth of the clock rate, and what are architects doing with the extra transistors now to increase performance?
e. [10] <1.4> The rate of growth for DRAM capacity has also slowed down. For 20 years, DRAM capacity improved by 60% each year. If the 8-gigabit DRAM was first available in 2014, and 16 gigabit is not available until 2019, what is the current DRAM growth rate?

1.20. [10/5/10/5/10] <1.2> This problem explores the total cost of ownership (TCO) associated with computer systems by exploring two similar server configurations as shown in Figure 1.22.

a. [10] <1.2> For each of the servers, assuming each has a workload that is 70% integer and 30% floating point. Using the SPEC integer and floating-point rates given in Figure 1.22 for each server, which provides better performance?
b. [5] <1.2> Using the results from (a), how do the servers compare in their price/performance for this workload?
c. [10] <1.2> Assume each of the servers is busy 60% of the time and idle the rest of the time. When each is busy, assume it uses 90% of the power supply limit and only 30% when it is idle. How much power will each server consume in a year?
d. [5] <1.2> Assuming that 1 W of power per year costs $1, how much will the power usage for each server cost per year using the results from (c)?
e. [10] <1.2> Assuming the servers have a lifetime of 5 years, how much do they cost per year, taking into account their initial cost and annual power costs? How does this change for 6- or 7-year lifetimes? How much does each cost in total lifetime costs?

Reference

Barroso, L., Hölzle, U., Ranganathan, P., 2019. The Datacenter as a Computer: Designing Warehouse-Scale Machines. Springer Nature.

Jouppi, N., Kurian, G., Li, S., Ma, P., Nagarajan, R., Nai, L., Patil, N., Subramanian, S., Swing, A., Towles, B. Young, C., Patterson, D. "TPU v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings." In *Proceedings of the 50th Annual International Symposium on Computer Architecture.* June 2023.

Norrie, T., Patil, N., Yoon, D.H., Kurian, G., Li, S., Laudon, J., Young, C., Jouppi, N., Patterson, D., 2021. The design process for Google's training chips: TPU v2 and TPU v3. IEEE Micro 41 (2), 56−63.

Patterson, D., Gonzalez, J., Holzle, U., Le, Q., Liang, C., Munguia, L.-M., Rothchild, D., So, D., Texier, M., Dean, J., 2022. The carbon footprint of machine learning training will plateau, then shrink. Computer 55 (7), 18−28.

Patterson, D., Bennett, J., Bennett, M., Chelin, H., Harris, D., Hellar, J., Jones, W., Moron, K., Savani, P., Shepherd, R., Simar, R., Suskind, Z., Wallentowitz, S., 2025. Emebnch IOT 2.0 and DSP 1.0: Modern Embedded Computing Benchmarks. IEEE Computer. In press.