CHAPTER

3 Parallel processing and distributed computing

This chapter overviews concepts in parallel and distributed systems important for understanding the basic challenges of design and use of large-scale collections of autonomous and heterogeneous distributed systems such as computer clouds. Cloud computing is the result of knowledge and wisdom accumulated over some 60 years of computing and is intimately tied to parallel and distributed processing.

Parallel processing has mesmerized computational science and engineering communities since early days of the computing era, resulting in fascination with high-performance computer systems and, ultimately, with supercomputers. It is hard to expose parallelism in many scientific applications, but, the harder the problem, the more satisfying it was to develop parallel algorithms, implement them, wait for the next generation of processors running at a higher clock rate, and enjoy the impressive speedup. The enterprise computing world seemed more skeptical and less involved in the parallel processing movement.

Distributed computing studies distributed systems, i.e., systems whose components are running on networked computers, communicating and coordinating their actions by message passing or shared memory. Message passing has its roots in the operating system of the 1960s and the widespread use of distributed systems can be traced back to the invention of the Ethernet in the 1970s.

More than a half century after the dawn of the computing era, an eternity in the age of the silicon, disruptive multicore technology forced the community to realize the need to understand and exploit parallelism. Rather than wait for faster clock rates, we should better design algorithms and applications able to use all the cores of a modern processor.

Things changed again when cloud computing showed that there are new applications that can effortlessly exploit parallelism and, in the process, generate huge revenues. A new era in parallel and distributed systems began, the era of Big Data hiding nuggets of useful information and requiring massive amounts of computing resources. The new challenge is to obtain the results faster by effectively harnessing the power of millions of multicore processors and systems on a chip.

To process massive amounts of data efficiently, cloud applications distribute data to large numbers of servers. Cloud applications use a number of instances running concurrently. Many cloud applications are based on the *client–server* paradigm. A relatively simple software, a *thin-client*, is often running on the user's mobile device with limited resources, while the computationally intensive tasks are carried out on the cloud.

Transaction processing systems, including web-based services, represent a large class of applications hosted by computing clouds. Such applications run multiple instances of the service and require reliable and in-order delivery of messages.

Early on scientists and engineers understood that parallel processing requires specialized hardware and system software. It was also clear that the interconnection fabric was critical for the performance of parallel processing systems. Building high-performance computing systems proved to be a major challenge. The list of companies aiming to support parallel processing and ending up as casualties of this effort is long and includes names such as: Ardent, Convex, Encore, Floating Point Systems, Inmos, Kendall Square Research, MasPar, nCube, Sequent, Tandem, Thinking Machines, and possibly others, now forgotten. The difficulties of developing new programming models and the effort to design programming environments for parallel applications added to the challenges faced by each of these companies.

Hardware parallelism is critical for the performance of a single-core, multicore processors, systems on a chip, multiprocessor systems, clusters, and warehouse-scale computers, the backbone of computer clouds. In this chapter, the first sections cover parallel and distributed system hardware stressing the quantitative rather than qualitative aspects of computer architecture. Basic architectural concepts of modern computer systems, optimizations of computer architecture including caching, out-of-order instruction execution, dynamic scheduling, branch predictions, and ARM architecture are discussed in Sections 3.1, 3.2, and 3.3. Sections 3.4, 3.5, 3.6, and 3.7 cover SIMD architectures, GPUs (Graphics Processing Units), TPUs (Tensor Processing Units), SOCs (Systems On a Chip), and edge cloud computing.

Section 3.8 covers data, task, and thread-level parallelism. Extracting parallelism depends on the application; Sections 3.9 and 3.10 analyze the speedup limits given by Amdahl's Law and Amdahl's Law for multicore processors. Section 3.11 reviews the evolution of the most powerful computing systems, from supercomputers to large-scale distributed systems.

Organization principles for distributed systems such as modularity and layering, presented in Sections 3.12 and 3.13, are applied to the design of peer-to-peer and large-scale systems discussed in Sections 3.14 and 3.15, respectively. Section 3.16 presents composability bounds and scalability, and Section 3.17 surveys fallacies in distributed computing. Finally, Section 3.18 discusses blockchain technologies and applications.

3.1 Computer architecture concepts

Architecture is the science of designing buildings, monuments, parks, cities, and even computers. Computer architecture describes the functionality, organization, and implementation of computer systems. A digital computer processes data and consists of three sub-systems with distinct functionality and performance: (i) CPU (Central Processing Unit)—transforms data; (ii) memory—stores data; and (iii) I/O (Input/Output)—supports communication with the outside world.

The three subsystems communicate with one another using computer busses, groups of wires transporting different information; for example, one set of wires transmits instructions and control information, a second transmits data address, and a third one moves data between communicating entities. Each unit is connected to the bus via an interface, which buffers data before and after any exchange, allowing its end points with different bandwidths to operate asynchronously. A computer bus is the most primitive type of communication device we encounter in computer clouds.

Latency hiding. This concise view of computer architecture is a first sign that *computing and communication are inseparable,* a theme reverberating throughout several chapters. The three subsystems of a processor have different *bandwidths,* i.e., numbers of operations per unit of time, and *latency,* i.e., the time between the start and the completion of an operation. CPU registers are the fastest, the main memory is slower by some two orders of magnitude, and I/O devices are orders of magnitude slower than memory. The impact of this discrepancy on processor performance, due to physical and technological limitations of the three subsystems, can be hidden to some extent through architectural complications and data buffering.

Amazingly, the gap between the bandwidth and the latency of the three subsystems has widened significantly during the past three decades, increasing the pressure on computer architects. Processor bandwidth has improved by a factor of 10 000 – 25 000 versus a factor of 300 – 1 200 for memory and disks. The corresponding latency improvements are 30 – 80 times for processors versus 6 – 8 times for memory and disks [232].

Hiding the latency of slower subsystems is the holy grail of computer architecture. Latency hiding leads to architectural complications, eventually creating the hardware vulnerabilities discovered a few years ago, a topic discussed in depth in Section 3.2.

ISA—Instruction Set Architecture. ISA is an abstract model of a processor defining the set of instructions executed by the CPU. ISA reflects compiler writer's and programmer's view of the processor. CISC (Complex Instruction Set Computer) and RISC (Reduced Instruction Set Computer) are two competing CPU architectures.

CISC architecture is characterized by a large number of machine instructions, some operating with data in registers and others directly with data in the system's memory. CISC leads to a smaller footprint of the application code, but complicates CPU implementation and limits the efficiency of pipelining and other CPU optimization techniques. CISC architecture was used by IBM System 360 and 370 in the late 1960s. x86-64 is a 64-bit architecture developed by Intel as a successor of its 32-bit architecture, x86-32. x86-64 is implemented by Intel and AMD processors. x86-64 CISC instructions are not executed by the hardware, but compiled into microcode; then, the CPU executes the microcode.[1]

RISC architecture, true to the name, limits the number of instructions. RISC is a *load-store architecture*. RISC ISA includes two instructions, one to fetch data into CPU registers and the other to store data from CPU registers in memory; all arithmetic and logic instructions operate only with data in registers. RISC architecture was introduced in the early 1980s. Power PC, MIPS, and SPAC are three ISA RISC architectures conceived independently by John Cocke at IBM Research, John Hennessy at Stanford, and David Patterson at UC Berkeley. The contribution of all three was recognized with Turing Awards, the computer science equivalent of a Nobel Prize, John Cocke in 1987 and the others in 2017. John Hennessy and David Patterson are the authors of a seminal text on computer architecture; they pioneered the transition from *qualitative* to *quantitative* methods in computer architecture emphasizing objective measurements. Qualitative methods provide only some insights and identify factors affecting processor performance.

Control flow versus data flow processor architecture. The dominant processor architecture is the *control flow* architecture pioneered by John von Neumann [79]. The implementation of the processor control flow is straightforward: The *program counter* (PC) determines the next instruction to be loaded into the *instruction register* (IR) and then executed. The execution is strictly sequential, until a branch is encountered.

There is an alternative to control flow architecture, the *data flow* architecture when an operation is executed as soon as the data required by the operation becomes available. Though only a few general

[1] x86 instructions are internally converted into simpler RISC-style micro-operations specific to a particular processor and stepping level.

purpose data flow systems are available today, this alternative computer architecture is widely used by network routers, digital signal processors, and other special-purpose systems. The fastest way to get the results of any computation is using a data flow model, i.e., carry out each operation as soon as the input data becomes available.

Still, the von Neumann architecture based on the control flow model, is implemented by the vast majority of computers today. Why? Lack of locality, inefficient use of cache, and ineffective pipelining are most likely some of the reasons why data flow general-purpose processors are not as popular as control flow processors. It should not be surprising that some of the systems discussed in Chapters 4, 9, and 11 apply the data flow model for task scheduling on large clusters. The power of this model for supporting optimal parallel execution is unquestionable, and we should probably expect soon the addition of general-purpose data flow systems to the cloud infrastructure.

Processor clock, CPI, and IPC. The basic subsystems of a CPU are a Control Unit (CU), an ALU (Arithmetic and Logic Unit), and a Register File (RF). All CPU activities are controlled by an internal clock. The CU implements a state machine and at each clock cycle dictates what latches should open along the control paths to different circuits and what actions should be carried out. R, the clock rate, is measured in GHz (Gigahertz) (a Hz is one cycle/second and one GHz is 10^9 Hz). CPU bandwidth is determined by IPC (Instructions per Clock Cycle) or CPI (Cycle per Instruction), IPC = 1/CPI. The larger the IPC, the more performant the processor architecture. T, the execution time of an application, is a function of: N, the number of instructions executed, CPI, and R:

$$T = N \times CPI \times R.$$

The only way to decrease the execution time of applications is to increase CPI, R, or both. Increasing the clock rate, R, was the preferred method to boost processor performance until 2003–2006, when the challenges of heat removal forced an abrupt halt to the frenetic pace of clock-rate increase; the clock rate increased from a high of 22 MHz in 1986 (VAX 8600) to 3.2 GHz in 2003 (Intel Xeon EE). The energy consumption of solid-state devices is proportional to the second or third power of the clock rate depending upon the technology, thus increasing the clock rate by two orders of magnitude led to a dramatic increase of energy dissipated and of heat generated by processors.

The implacable laws of physics show that energy dissipation and heat removal will eventually seal the fate of solid-state technology. The arguments are very simple: Transistors must be densely packed to minimize communication time; the speed of light cannot be exceeded. A sphere allows optimal transistor packing; then, the energy dissipated by the transistors is proportional to the volume of the sphere ($V = 4/3\pi r^3$); thus, the heat generated is proportional to the cube of the radius r of the sphere. The heat can only be removed though the surface of the sphere ($S = 4\pi r^2$), thus, the amount of heat removed is proportional to the square of the radius.

The only alternative to boost system performance is to increase the CPI of each CPU and to have multiple cores. This is feasible because the large number of transistors on a chip predicted by Moore's Law allows the implementation of various architectural optimizations and of multiple cores. The first step to increase the CPI was to exploit bit-level and instruction-level parallelism implemented since the early days of computing. Multicore processors were developed much later. In 2001, IBM introduced the world's first multicore processor, a chip with two 64-bit microprocessors comprising more than 170 million transistors. The total number of instructions executed by a multicore processor is the sum of instructions executed by individual cores.

Table 3.1 Basic superscalar processor pipeline. The pipeline has five stages: instruction fetch (IF), instruction decode (ID), instruction execution (EX), memory access (MEM), and write back (WB). In each clock cycle two instructions are executed; instructions i, $i+2$, $i+4$, $i+6$ and $i+8$ are executed by unit 1, and instructions $i+1$, $i+3$, $i+5$, $i+7$ and $i+9$ are executed by unit 2. Once the pipeline is full, two instructions complete execution of every clock cycle.

	1	2	3	4	5	6	7	8	9
i	IF	ID	EX	MEM	WB				
$i+1$	IF	ID	EX	MEM	WB				
$i+2$		IF	ID	EX	MEM	WB			
$i+3$		IF	ID	EX	MEM	WB			
$i+4$			IF	ID	EX	MEM	WB		
$i+5$			IF	ID	EX	MEM	WB		
$i+6$				IF	ID	EX	MEM	WB	
$i+7$				IF	ID	EX	MEM	WB	
$i+8$					IF	ID	EX	MEM	WB
$i+9$					IF	ID	EX	MEM	WB

Bit-level and instruction-level parallelism. Parallelism at different levels can be exploited by a von Neumann processor. The first two levels are:

1. *Bit-level parallelism.* A computer word is handled as a unit by the instruction set and the processor hardware. The number of bits in a word has increased gradually from 4-bit processors to 8-bit, 16-bit, and 32-bit processors. This has reduced the number of instructions required to process larger size operands and enabled a dramatic performance improvement. During this evolutionary process, the number of address bits have also increased from 32 bits to 64 bits in 2004 allowing instructions to reference a larger address space, from 2^{32} (about 4 GB), to 2^{64} (17 179 869 184 GB).
2. *Instruction-level parallelism* (ILP). Computers used multi-stage pipelines to speedup execution. *Once an n-stage pipeline is full, an instruction is completed at every clock cycle unless the pipeline is stalled.* This idea mimics assembly lines used in manufacturing as early as 1913 when Henry Ford introduced this innovation at the Highland Park assembly plant.

Pipelining means splitting an instruction into a sequence of steps that can be executed concurrently by multiple units on the chip. A basic pipeline of a RISC (Reduced Instruction Set Computing) architecture consists of five stages.[2]

A *superscalar processor* executes more than one instruction per clock cycle, see Table 3.1 [232]. A Complex Instruction Set Computer (CISC) architecture could have a much larger number of pipelines stages, e.g., an Intel Pentium 4 processor has a 35-stage pipeline.

The events in each pipeline stage involve multiple hardware components including: the PC (Program Counter), a register pointing to the next instruction to be executed, the IR (Instruction Register)

[2] The number of pipeline stages in different RISC processors varies. For example, ARM7 and earlier implementations of ARM processors have a three-stage pipeline, fetch, decode, and execute. Higher performance designs, such as the ARM9, have deeper pipelines: Cortex-A8 pipeline has 13 stages.

containing the current instruction, the register file, and the ALU. A generic descriptions of each stage of a five stage pipeline is:

1. IF—fetch the instruction and store it into IR, compute new program counter, store the incremented program counter in the PC register and into a pipeline register for later computing the branch target address if necessary.
2. ID—decode the instruction, fetch the data required by the instruction, and store it in the registers specified by the instruction in IR.
3. EX—perform an ALU operation or an address calculation and the condition code for the ALU operation; compute the target address if the instruction is a taken branch.
4. MEM—cycle the memory; write the PC if needed and pass along values needed in the final stage.
5. WB—update the register field from either the ALU output or the loaded value.

The execution flow is different for arithmetic and logic, load, store, and control and branch instructions. ALU instructions access two register operands; load and branch instructions access a register to obtain a base address; and store instructions access a register to obtain the register operand and another register for the base address.

Pipeline hazards occur when unchecked pipelining would produce incorrect results. Data, structural, and control hazards have to be handled carefully. *Data hazards* occur when the instructions in the pipeline are dependent upon one another. For example, a Read after Write (RAW) hazard occurs when an instruction operates with data in a register being modified by a previous instruction. A Write after Read (WAR) hazard occurs when an instruction modifies data in a register being used by a previous instruction. Finally, a Write after Write (WAW) hazard occurs when two instructions in a sequence attempt to modify data in the same register and a sequential execution order is violated.

Structural hazards occur when the circuits implementing different hardware functions are needed by two or more instructions at the same time. For example, a single memory unit is accessed during the instruction fetch stage when an instruction is retrieved from memory, and it is also accessed during the memory stage when data is written and/or read from memory. Structural hazards can be resolved by separating the component into orthogonal units (such as separate caches) or bubbling the pipeline. *Control hazard* are due to conditional branches. On some pipelined microarchitectures, a processor does not know the outcome of a branch when it needs to insert a new instruction into the pipeline during the fetch stage.

A *pipeline stall* is the delay in the execution of an instruction in the pipeline to resolve a hazard. Such stalls could drastically increase the CPI according to the expression:

$$PipelineCPI = IdealPipelineCPI + StructuralStalls + DataHazardStalls + ControlStalls$$

Pipeline scheduling separates the dependent instruction from the source instruction by the pipeline latency of the source instruction. Its effect is to reduce the number of stalls.

Computer architecture taxonomy. In 1966, Michael Flynn proposed a taxonomy of computer architectures based on the number of *concurrent instructions* and the number of *data streams*: SISD (Single Instruction, Single Data); SIMD (Single Instruction, Multiple Data); and MIMD (Multiple Instructions, Multiple Data).

SISD processors have been around since ENIAC, the computer built at the University of Pennsylvania's Moore School of Electrical Engineering between 1943 and 1946 by J. Presper Eckert and John Mauchly.

Individual cores of a multicore processor are SISD (unless multithreaded) and support the execution of one or more threads at any given time. Multiple threads may run concurrently but only one is active at any given time. A *superscalar* processor executes more than one instruction per clock cycle. A single-core superscalar is still a SISD processor.

SIMD supports vector processing. When a SIMD instruction is issued, the operations on individual vector components are carried out concurrently. For example, to add two vectors $(a_0, a_1 \ldots \ldots a_{63})$ and $(b_0, b_1 \ldots \ldots b_{63})$, all 64 pairs of vector elements are added concurrently, all sums $(a_i + b_i), 0 \le i \le 63$ are available at the same time.

The first use of SIMD instructions was in vector supercomputers, such as the CDC Star-100 and Texas Instruments ASC in the early 1970s. Vector processing was especially popularized by Cray in the 1970s and 1980s by attached vector processors such as those produced by Floating Point Systems and by supercomputers such as Thinking Machines CM-1 and CM-2. The first widely deployed SIMD instruction set for gaming was Intel's MMX extensions to the *x86* architecture. IBM and Motorola then added AltiVec to the POWER architecture. There have been several extensions to the SIMD instruction sets for both architectures as we shall see in Section 3.4.

MIMD refers to a system with several processors that function asynchronously and independently; at any one time, different processors may be executing different instructions on different data. Multicore processors support true MIMD execution. Each core has its own register file, ALU, and floating-point execution units. Each core has its private L1 instruction and data caches, as well as L2 cache. All processor cores share the L3 cache.

Several processors can share a common memory, and we distinguish several types of multiprocessor systems: UMA, NUMA, and COMA, uniform, non-uniform, and cache only memory access, respectively. A MIMD system could have a distributed memory. Processors and memory modules communicate with one another using an interconnection network, such as a hypercube, a *2D* torus, a *3D* torus, an omega network, or another network topology. Today, most supercomputers are MIMD machines, and some use GPUs instead of traditional processors. Multicore processors with multiple processing units are now ubiquitous. As more powerful processors are needed, *hyper-threading* was introduced for Xeon and later Pentium 4 processors by Intel in 2002. Hyper-threading takes advantage of unused processor resources and presents itself to the operating system as a two-core processor.

MISD (Multiple Instructions, Single Data) is a fourth possible architecture, but it is very rarely used, mostly for fault tolerance.

Performance metrics and benchmarks. MIPS (Million Instructions per Second) quantify processor performance for integer arithmetic, while MFLOPS (Million Floating Point Operations per Second) quantify processor performance for floating-point arithmetic. The information provided by MIPS and MFLOPS is of limited usefulness because processor performance is application-specific. An application can be CPU-, memory-, I/O-, or all-intensive; the application will run optimally on a processor where the corresponding subsystem, CPU, memory, and I/O is optimized. This is the reason why AWS offers various types of instances, e.g., compute-optimized, memory-optimized, storage-optimized.

Applications running on the same processor may run at very different MIPS or MFLOS rates depending on the type of resources intensively used. The only useful performance characterization of a processor is provided by *benchmarks*, suites of applications with similar characteristics. PARSEC benchmark suite developed at Princeton by Christian Bienia for his Ph.D. dissertation (http://parsec.cs.princeton.edu) is widely used. PARSEC compares favorably with other benchmarks, including SPEC CPU 2017, BioParallel, PhusicsBench, SPLAH-2, among others.

3.2 Grand architectural complications

In this section we assume a RISC architecture though there are only minor differences for CISC architecture. Two conditions guarantee computation completion in the shortest possible time: A—memory optimization: the instruction to be executed next should be in the IR register; after the instruction is decoded, the data referenced by instruction should be in the CPU registers before the instruction execution stage can begin; B—CPU optimization: once condition A is satisfied, we should keep the pipeline full at all times.

To address condition A, we have to consider memory latency, bandwidth, and cost. *Memory latency* is the time it takes to fetch one instruction, or a word or byte of data, measured in picoseconds (psec), nanoseconds (nsec), or milliseconds (msec); *memory bandwidth* is the amount of data delivered in one unit of time, measured in GB/sec or MB/sec. Ideally, the storage should have infinite capacity and bandwidth and zero latency and cost, but in reality, the storage has limited capacity, finite bandwidth, and non-zero latency. Memory cost is also a factor: The faster, the higher is the cost of physical memory. Computers need large memories because the code footprint, as well as the size of data, have been continually increasing. The cost of very large and very fast memory is prohibitive, and solutions balancing the memory performance and cost have been developed.

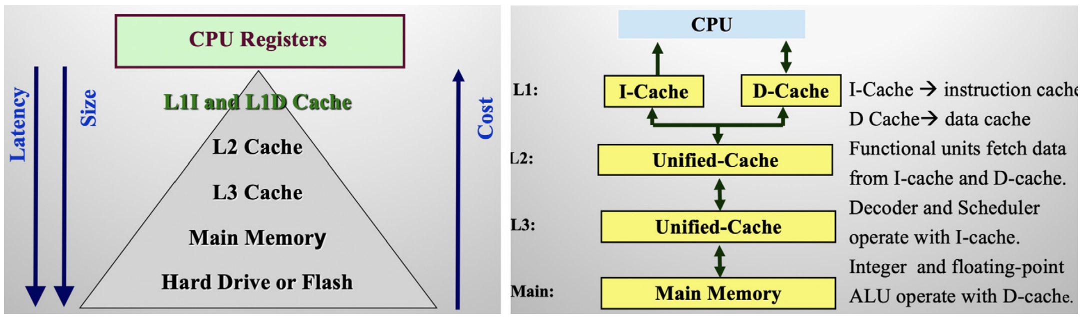
Keeping the pipeline full at all times has major implications:

- B1—avoid pipeline stalls;
- B2—deal with the fact that different arithmetic operations require a different number of clock cycles; for example, an integer addition my require two clock cycles, a multiplication 12 clock cycles, and a division 50–60 clock cycles;
- B3—avoid wasting clock cycles due to control flow instructions which alter the sequential execution pattern. A control flow instruction could be a condition generated by an arithmetic operation that modifies condition codes and may or may not force us to fetch a new instruction, or a subprogram call or an instruction that transfers control to a subprogram or elsewhere in the address space of the program.

A paramount complication: The architecture must ensure program correctness. This means that the architecture should: (i) *Preserve exception behavior*, any change in instruction execution order must not change the order in which exceptions are raised[3]; (ii) *preserve instruction flow*, the flow of data between instructions that produce results and consume them; and last but not least (iii) *preserve security*, i.e., it does not open side channels allowing an intruder to access data in the address space of the executable. Not even a data flow architecture could easily overcome the problems posed by B1–B3 and by the requirements for program correctness.

A. Memory hierarchy; data and instruction caches; cache organization and performance. The first challenge seems the easiest of the two, so we start with it. The goal is to decrease memory latency to match the CPU speed. The solution is to access blocks of code and data rather than individual items. Fortunately, code and data enjoy a very useful property, *locality*. There are two flavors of locality, *spatial* and *temporal*.

[3] An *exception* is an unexpected event within the processor, e.g., an arithmetic exception or an addressing exception, while an *interrupt* is an unexpected outside event, e.g., an I/O event or a timing interrupt. Whenever an exception or an interrupt occurs, the hardware stops the execution of the running process and executes the code performing an action in response to the exception.



(a) (b)

FIGURE 3.1

(a) Memory hierarchy; CPU registers have the shortest latency followed by several levels of cache. (b) Cache levels: the first level cache consists of L1I, instruction cache and L1D, and data cache; L2 and L3 level caches are unified, used for instruction and data.

Spatial locality means that, once a block of code is brought into a faster storage, there is a high probability that several, possibly many instructions in the block will be executed before another block of code is referenced. *Temporal locality* means that, once a block of code is brought into the faster storage, several, hopefully many, instructions in this block of code will be executed; this is a reasonable expectation because instructions are executed sequentially most of the time. Similar arguments hold for data locality: Both spatial and temporal data locality reflect the logical organization of data.

Handling blocks of code rather than individual instructions and blocks of data rather than words of data decreases the average latency per reference. Temporal locality has an important corollary, captured by the *working-set* concept. The working set of a program is a collection of blocks that have been recently referenced. A program with a small working set references a smaller number of data blocks in an interval of time Δt than one with a larger working set.

The first idea that comes to mind is to design a storage hierarchy consisting of small, very fast, and expensive memory working in concert with increasingly larger and less expensive memory. The fast memory, called *cache*, was introduced by Maurice Wilkes from the University of Cambridge, who received the Turing Award in 1967 for his lifelong contributions to computer architecture.

Fig. 3.1(a) displays the memory hierarchy and the bandwidth, latency, cost-per-bit, and the storage technology (SRAM—Static Random Access Memory, DRAM–Dynamic Random Access Memory, and magnetic) at each level of storage hierarchy. Memory technology, bandwidth, latency, and cost are:

CPU registers—multipoint SRAM, 200+ GB/sec, 300+ psec, \$ 0.01 per bit.
On-chip L1I or L1D cache—SRAM, 50+ GB/sec, 300+ psec, \$ 10^{-4} per bit.
On-chip L2 cache—SRAM, 10+ GB/sec, 2+ nsec, \$ 10^{-4} per bit.
Main memory—DRAM, 2+ GB/sec, 50+ nsec, < \$ 2×10^{-7} per bit.
Hard disk—magnetic, 10+ MB/sec, 10 msec, \$ 1×10^{-9} per bit.

L1 cache capacity is several hundred KB (384 KB for AMD Ryzen 5 5600X, 1 MB for Intel i9-9980XE); L2 cache capacity 1–10 MB, and L3 10–50 MB. Fig. 3.1(b) shows a further refinement of storage hierarchy with multiple cache levels. The fastest cache consists of on-chip instruction cache, L1I, and on-chip data cache, L1D. The other cache levels, L2 and L3, are unified instruction and data caches. Recently, L4-level cache was added to the cache hierarchy. Different cache levels are accessed by ALU, instruction decoder, and scheduler.

Cache implementation requires a number of important decisions: What should the block size be, where to place a block brought in to cache, what to do when the cache is full and a new block must be fetched from memory, and how to handle a modified cache block?

Cache organization elements are: (i) *cache block*—the unit transferred between memory and cache; (ii) *cache row*—consists of three elements for each cache block: *Tag:* contains part of the address of the item fetched from memory, the *Data block:* contains data fetched from memory, and it flags either valid or dirty; and (iii) *Cache line*—identifies the cache set for an n-way set associative cache. The mapping between memory address and cache address is done by splitting the address generated by CPU into Tag, Index, and Word offset in the block. The Index portion of the CPU-generated address identifies the cache line, and the Tag of CPU-generated address is compared with the Tag bits of the cache line. The valid bit indicates if there is a cache hit or a cash miss. In case of an *n-way set associative cache,* each cache block of the line includes all three fields, valid bit, tag, and data.

There are several options for block placement in cache: (i) *fully associative cache*—a block can be placed in any cache location; (ii) *direct map cache*—a block can be placed only in one cache location determined by the starting address of the block in memory and the cache capacity in number of blocks. For example, if cache capacity is 16 blocks, a memory block starting at address 128 should be placed in cache block 8 (128 modulo 16); (iii) *set-associative cache*—the cache is organized in sets consisting of multiple blocks. This cache organization reduces the miss rate as multiple alternatives for bloc placement in the set exist, e.g., a four-way set associative cache offers four-options for block placement in the cache, i.e., the first, second, third, or fourth block of the set.

For example, consider a 32-bit architecture and a processor with a 64 Kbyte cache, 32 byte cache block and one block per cache set. Then the number of blocks is $64\,000/32 = 2\,000$ and the 32 bits of the address generated by the CPU are used by the cache management as follows: bits $0 - 4$ give the offset of a word in a block of 32 bytes; bits $5 - 15$ give index of one of the 2 000 cache blocks; and bits $16 - 31$ are the block tag. Each cache line consists of one valid bit, the block tag, and 32 bytes of data. If another processor had the same cache and block size but a two-way set-associative cache there will be 1 000 cache lines, each line will have one valid bit, a 17 bit tag and 32 bytes of data for each one of the two blocks in the set.

When CPU generates an address in the address space of the process, the cache is accessed first. The address can be in a block already in cache, and we experience a *cache hit*, or a new block must be fetched from memory and a *cache miss* occurs. There are several types of cache misses: *compulsory* - first reference to a block; *capacity* - blocks discarded and later retrieved; and *conflict* - repeated references to multiple addresses from different blocks that map to the same location in the cache.

Cache capacity is limited: It is two to three orders of magnitude smaller, KB or MB as shown in the caption of Fig. 3.1, versus GB for storage capacity. Once the cache is full, the block to be evicted depends upon the placement strategy that dictates where the new block should reside. The block to be evicted can be dirty, i.e., modified, different from its copy left in memory, so the memory must be updated. A dirty bit will identify a modified cache block. There are two choices for handling the modified cache block: *Write through cache*—a modified cache block is immediately written to memory and *write back cache* when memory is updated when the block must be evicted from the cache.

The *cache miss rate* is the number of cache misses per unit of time. A cache miss incurs a *miss penalty* much larger than the time to access a block already in cache, the so-called *hit time*. The average memory access time (AMAT) is:

$$AMAT = HitTime + MissRate \times MissPenalty.$$

Several cache techniques are used to reduce AMAT:

1. Small and simple L1 caches—reduce the critical cache timing path consisting of the time to extract the Index and Tag from the CPU generated address to compare the tags of the bocks on the same cache line, and to select the block.
2. Fast hit times via way prediction—keep extra bits in cache to predict the "way," i.e., the block within the set, at the next cache access.
3. Cache pipelining—use multiple cycles to access the cache. For example, in stage 1, read the tag and valid bit, in stage 2, combine the result of stage 1 and start read if hit, and in stage 3, finish data read and transfer data to the CPU. Cache pipelining improves bandwidth, but it has higher latency and increases the branch miss-prediction penalty.
4. Increase cache bandwidth with non-blocking caches—allow data cache to continue to supply cache hits during a miss; processors can hide L1 miss penalty but not an L2 miss penalty.
5. Independent banks; interleaving—the cache is divided into a number of *cache banks* that can be accessed concurrently.
6. Early restart and critical word first—as soon as the requested word of the block arrives, send it to the CPU, and let the CPU continue execution while filling the rest of the words in the block.
7. Merging write buffers to reduce miss penalty—write buffers allow CPU to continue while waiting to write to memory.

Several other cache-like units are used to speed up instruction execution. For example, ROB (Reorder Buffer) is a cache for out-of-order instruction execution, and BTB (Branch Translation Buffer) is a cache used by the branch prediction hardware, discussed next.

TLB (Translation Look aside Buffer) is used by the virtual memory management to speed up dynamic address translation, the translation of virtual addresses generated by the CPU to physical memory addresses. Virtual memory applies ideas very similar to caching and extends the physical memory of a system allowing multiple processes, each with an address space (the universe of addresses visible to the process) larger than the processor's physical memory, to run concurrently. In this case, the units of data exchanged between the lower level of storage hierarchy, disks, and memory are called *pages*. Without virtual memory one would have to reorganize a program to fit in the physical memory of each processor, leading to an unthinkable debacle.

This concludes our analysis of challenge A for CPI optimization. Needless to say, this presentation omits many additional complications, e.g., the fact that the address generated by the CPU is a *virtual address* forcing the cache management to interact with virtual memory management.

B. CPU optimization. Modern von Neumann microprocessors attempt to emulate the data flow model. This means to execute an instruction as soon as the data it needs is available, while guaranteeing program-order execution. Multiple instructions issue, dynamic instructions scheduling, branch prediction, speculative execution, and multi-threading are some of the complications designed to speed up execution.

Recall that CPU performance is determined by the product of three terms: the number of program instructions, the CPI, and the clock rate. Performance optimization implies CPI minimization; thus, the maximization of three types of flows involved in instruction processing are: *instruction flow*—affected by branch instructions, *register data flow*—affected by ALU instructions, and *memory data flow*—affected by load and store instructions [232].

Sequential instruction execution is interrupted by control instructions. Handling unconditional and conditional branches generates empty instruction pipeline slots. In case of unconditional branches, the next instruction cannot be fetched until the target address of the branch is calculated. The number of delay slots needed for target address generation depends upon the addressing modes of the branch instruction. The three addressing modes are: (i) *PC-relative* addressing when the branch target address can be generated during the fetch stage; (ii) *register indirect* addressing when the branch instruction must traverse the decode stage to access the register; and (iii) *register indirect with an offset* addressing when the offset must be added after register access.

Conditional branches incur a larger penalty: the CPU must wait for the resolution of the branch condition. If the branch is taken, it must also wait until the target address is available; the delay involves several clock cycles for traversal of the decode, dispatch, and execute pipeline stages. The number of *delay slots* needed for condition evaluation of a conditional branch instruction is different when condition code registers are available or when the comparison of two general purpose registers generates the branch condition. The number of empty instruction pipeline slots is multiplied by the CPU width for a superscalar CPU.

Out-of-order instruction execution requires additional hardware, including: reservation stations, load-store buffers, a FIFO queue holding the instructions issued by functional units, a reorder buffer (ROB), and a common data bus (CDB). There are multiple reservation stations (RS) for each functional unit, e.g., floating point add/subtract, floating-point multiple/divide. A reservation station stores the instruction, buffered operand values (when available), and the ID of the reservation station providing the operand values. RS feed data to floating-point arithmetic units modified to accept ROB as a source.

ROB tracks the state of inflight instructions in the pipeline and provides the illusion of in-order program execution. After instructions are decoded and renamed, they are then dispatched to the ROB and the FIFO queue and marked as busy. When an instruction finishes execution, ROB is informed, and the status of the instruction becomes *not busy.* Once the "head" of the ROB is no longer busy, the instruction is *committed,* and its state is visible. If an exception occurs and the excepting instruction is at the head of the ROB, the pipeline is flushed, and no architectural changes that occurred after the excepting instruction are made visible. The ROB then redirects the PC to the appropriate exception handler. Register values and memory values are not written until an instruction commits.

The instruction execution steps and the actions carried in each steps are:

1. *Instruction issue:* (i) get next instruction from the FIFO queue; (ii) if a reservation station is available, issue the instruction to the reservation station including operand values, if available; and (iii) if operand values are not available, stall the instruction. When an operand value is unavailable, use a placeholder for the value. The placeholder is a tag indicating the reservation station to produce the value. The placeholder will be later replaced by the result produced by a functional which will broadcasts it on CDB.
2. *Execute instruction:* (i) when an operand becomes available, store it in reservation station(s) waiting for it; (ii) when all operands are ready, issue the instruction. To ensure correctness loads and store

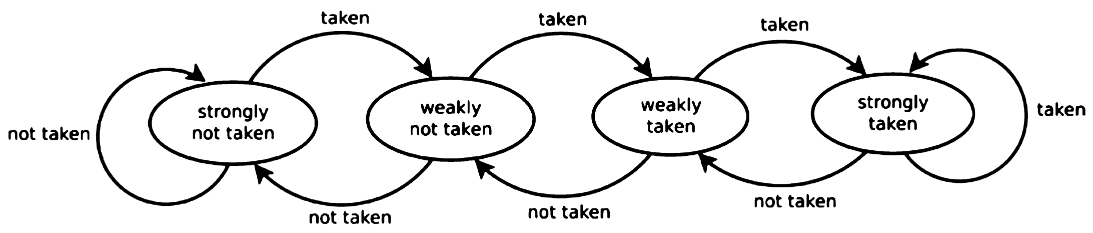


FIGURE 3.2

Two-bit saturation counter FSM states and transitions.

are maintained in program-order through effective address and no instruction is allowed to initiate execution until all branches that precede it in program-order have completed.

3. *Write instruction result:* broadcast result on CDB for waiting reservation stations or for store buffers.

Branch prediction. The penalty for interrupting the sequential instruction execution by branches increases when the number of pipeline stages increases. The behavior of branch instruction is predictable, and accurate predictions lead to significant performance improvements. Two functions are required for branch prediction: *branch target speculation* - to predict the address of the next instruction and *branch condition speculation* - to predict if the branch will be taken or not. Both require additional hardware.

Branch target speculation uses a cache accessed during the instruction fetch to store the target address of the previous branch instruction, called BTB (Branch Target Buffer). BTB is indexed by the current PC, and it is accessed concurrently with the L1I-cache. BTB has two fields: (i) branch instruction address containing the address of the current branch instruction; and (ii) branch target address containing the speculative target address to be used as the new PC.

If the address of the current branch instruction has an entry in BTB, more precisely in field (i) of BTB, then field (ii) of the same BTB entry contains the predicted address when the branch is taken. There are two possibilities: (1) the current branch is taken; thus, the prediction was accurate and there is no penalty for accessing the BTB; (2) the current branch is not taken, the BTB prediction did not help, and there is a penalty of several clock cycles needed to compute the branch address.

BTB is a cache with limited capacity, thus, the address of the current branch instruction may not be in BTB. When the branch is not taken, there is no penalty; if the branch is actually taken, there is the penalty for computing the branch address, which depends on the addressing modes supported by the ISA.

Branch condition speculation can use a *biased strategy*. In this case, the branch target offset, the difference between the address of the current instruction (ci), and the address of the target instruction (ti) are used. When $ci < ti$, it is likely to have a loop closing branch, and the prediction is that the branch is taken; if $ci > ti$, the prediction is that the branch is not taken. This is not a very effective predictor.

History-based prediction uses a finite state machine (FSM) to decide if T (taken) or N (not taken). The FSM for *two-bit* predictor has four states, each state labeled by the last two events (TT, NT, TN, NN) and the predicted state either T or N. The prediction is changed when the prediction is wrong two consecutive times. The *two-bit saturation counter* is a better alternative to the two-bit predictor. In this case a single iteration will not change the predicted direction. For example, if a branch has been taken many times in succession, the FSM in Fig. 3.2 will be in the *strongly taken* state; if next time this branch is not taken, the new state will be *weakly taken,* and, if the next branch is taken, it will switch back to the *strongly taken* state. Only if the branch in not taken two or more times consecutively will the prediction change to not taken. This hysteresis effect can significantly boost prediction, e.g., to 85%, for SPECint92.

Yeh and Patt algorithm for branch condition speculation looks at previous behavior when this same sequence has occurred, using historical data to generate a prediction. For each branch it maintains a storage register with the state of the last k branches, 1 if taken and 0 if not taken and a pattern table of 2^k entries each implementing a two-bit saturating counter that tracks the results of previous iterations that occurred when the history register was in a given state. When a branch is encountered, the contents of the history register are used as an index into the pattern table, selecting the entry that corresponds to the recent history of that branch. After the branch is resolved, the result (taken or not taken) is shifted into the history register and used to update the appropriate entry in the pattern table.

Correlating predictors use global history. To record what happened at the last m branches, we need a shift register with m bits, one for each branch; each bit is set to 1 if the branch was taken and to 0 if the branch was not taken. Multiple two-bit predictors for each branch are used, one predictor for each possible combination of outcomes of preceding n branches.

The concepts and ideas discussed in this section project the view of a very complex mechanism with many moving parts that have to work in concert, and they do. Billions and billions of processors used today run flawlessly on all continents and in space on satellites and space probes. Unfortunately, the interaction of cash and processor optimization discussed in this section has led to the surprising hardware vulnerabilities discussed next.

Critical vulnerabilities in modern processors. Two hardware vulnerabilities, Meltdown and Spectre, affect systems with Intel x86, IBM Power architectures, and some ARM microprocessors. *Meltdown vulnerability* exploits a race condition between memory access and privilege checking during instruction processing and affects x86-based systems. The mechanism used to create a side-channel attack is:

1. The CPU is instructed to read data at an address A that it is forbidden to access. The read is speculatively executed, and address A is not saved in the visible CPU state, but it would become visible only after access check confirms that access to A is permitted.
2. The CPU continues speculative execution and is instructed to read the contents of an address calculated using a base register B and A as displacement. The access to this address is permitted and causes caching of Base+A location.
3. The instruction in step 1 attempts to complete and save its result in visible CPU state architecture. The instruction retirement logic detects that access is forbidden, and the results produced by this and all subsequent instructions are discarded. However, the effect of caching data at the address given Base+A is not undone.
4. The attacker measures how fast elements of Base[x] array can be accessed. If the data at the address Base+A is cached, and all other elements Base+x are not; only instruction referencing Base+A execute faster. The measurement can detect the timing difference and determine the value of A obtained by the rejected instruction of step 1, thus allowing the attacker to access the contents of the forbidden memory address.

Spectre enables a whole class of potential vulnerabilities by exploiting branch prediction. The speculative execution resulting from a branch miss-prediction enables an attacker to trick a program into accessing arbitrary locations in the program's address space and potentially obtain sensitive data. The attacker trains the branch prediction logic to reliably hit or miss cache, accurately times the difference between cache hits and cache misses, and uses this information to open a covert channel.

The existing code in the address space of the victim process is searched for places where speculation touches upon otherwise inaccessible data. Then the attacker manipulates the processor into a state where speculative execution has to touch that data. Next, it times the side effect of the processor being faster and determines if its by-now-prepared prefetch machinery indeed did load a cache line. This attack can also be remotely conducted using the following sequence of actions: flush cache, mis-train the branch predictor, and time the read operations.

Software patches to limit the impact of these hardware vulnerabilities have been released making even harder the exploitation of the two vulnerabilities. Only very sophisticated attackers could exploit these vulnerabilities, but in the age of state-sponsored cyber terrorism such attacks could affect the critical infrastructure of a country.

The conclusion of this section is obvious: Advances in computer architecture have increased the complexity of processor design, sometimes with the unexpected and dangerous side effects just discussed. Solid-state technology will unquestionably reach its limits and will most likely be replaced by quantum technology, a super-disruptive set of concepts and ideas that requires a very different way of thinking. The wealth of ingenious solutions for CPU optimizations will then be forgotten.

3.3 ARM architecture

There is no better way to conclude the discussion of general-purpose processors and illustrate the dramatic evolution of computer architecture than to overview the ARM architecture. ARM—Advance RISC Machine—is a state-of-the-art microarchitecture for a broad range of applications and workloads. ARM microprocessors are used not only for mobile devices such as smartphones, tables, and laptops, but also by some cloud instances. ARM architecture has some notable features:

a. ARM and Thumb are two different instruction sets supported by ARM cores with a "T" in their name. For instance, ARM7 TDMI supports Thumb mode. ARM instructions are 32 bits wide, and Thumb instructions are 16 wide. Thumb mode allows for code to be smaller and potentially faster when the target has slow memory.
b. Uniform 16 × 32-bit register file including program counter, stack pointer, and link register. Fixed instruction width of 32 bits eases decoding and pipelining at the cost of decreased code density; the 16-bit Thumb instruction increases code density.
c. Mostly single clock-cycle execution.
d. Powerful indexed addressing modes.
e. Support unaligned accesses for halfword and single-word load/store instructions with some limitations, such as no guaranteed atomicity.
f. Fast leaf function calls use a link register. When calling the leaf function, the return address does not have to be pushed to the stack since it's stored in the link register. However, there are situations where the leaf function must save data to the stack.
g. Arithmetic instructions alter condition codes *only when desired.*
h. Conditional execution of most instructions reduces branch overhead and compensates for the lack of a branch predictor in early chips.

i. 32-bit barrel shifter that can be used with most arithmetic instructions and address calculations without performance penalty. A barrel shifter is used to shift and rotate n-bits typically within a single clock cycle.
j. Almost every ARM instruction has a conditional execution feature, the predication, implemented with a four-bit condition code selector (the predicate). To allow for unconditional execution, one of the four-bit codes causes the instruction to be always executed. Most other CPU architectures only have condition codes on branch instructions.
k. Integer add, subtract, and multiply arithmetic operations; ARMv7-M and ARMv7E-M versions of the architecture also support divide operations, while others, including ARMv7-A, only optionally support it.
l. Simple and fast, two-priority-level interrupt subsystem has switched register banks.

There are several CPU modes, depending on the implemented architecture features. At any moment in time, the CPU can be in only one mode, but the mode can be switched in response to external events or programmatically. These modes are:

1. User—the only non-privileged mode.
2. FIQ—privileged mode entered whenever the processor accepts a fast interrupt request.
3. IRQ—privileged mode entered whenever the processor accepts an interrupt.
4. Supervisor (SVC)—privileged mode entered when the CPU is reset or when an SVC instruction is executed.
5. Abort—privileged mode entered whenever a prefetch abort or data abort exception occurs.
6. Undefined—privileged mode entered whenever an undefined instruction exception occurs.
7. System (ARMv4 and above)—the only privileged mode not entered by an exception. It can only be entered by executing an instruction that explicitly writes to the mode bits of the Current Program Status Register (CPSR) from another privileged mode (not from user mode).
8. Monitor (ARMv6 and ARMv7 Security Extensions, ARMv8 EL3)—introduced to support Trust-Zone extension in ARM cores.
9. Hyp (ARMv7 Virtualization Extensions, ARMv8 EL2)—hypervisor mode supporting Popek and Goldberg virtualization requirements for non-secure CPU operation.
10. Thread (ARMv6-M, ARMv7-M, ARMv8-M)—can be specified as either privileged or unprivileged. Whether the Main Stack Pointer (MSP) or Process Stack Pointer (PSP) is used can also be specified in CONTROL register with privileged access. This mode is designed for user tasks in RTOS (Real Time OS) environment, but it's typically used in bare-metal for super-loop.
11. Handler (ARMv6-M, ARMv7-M, ARMv8-M)—mode dedicated for exception handling (except RESET handled in Thread mode).

The *state of a processor/core* is given by the contents of the register file. This state is saved at the time of a context switch and restored when the interrupted process or thread is scheduled to run again. The overhead of a context switch depends upon the number of registers and this overhead is low because the ARM register file consists of a small number of registers. Register functions are: registers R0 through R7 are the same across all CPU modes and are never banked; registers R8 through R12 are the same across all CPU modes except FIQ mode. FIQ mode has its own distinct R8 through R12 registers. R13 and R14 are banked across all privileged CPU modes except system mode, that is, each mode that can be entered because of an exception has its own R13 and R14. These registers generally contain the stack pointer and the return address from function calls, respectively. R13 is also referred to as SP, the Stack Pointer, R14 is also referred to as LR, the Link Register and R15 is also referred to as PC, the Program Counter.

The Current Program Status Register (CPSR) contains critical CPU state information, the CPU execution mode, whether the CPU can be interrupted or not, the condition code set by arithmetic and logic instruction, and so on. The 32 bits of CPSR are: processor mode bits, M (0 to 4); Thumb state bit, T (bit 5); FIQ disable bit, F (bit 6); IRQ disable bit, I (bit 7); imprecise data abort disable bit, A (bit 8); data endianness bit, E (bit 9); if–then state bits, T (bits 10 to 15 and 25, 26); greater-than-or-equal-to bits, GE (bits 16 to 19); do not modify bits, DNM (bits 20 to 23); Java state bit, J (bit 24); sticky overflow bit, Q (bit 27); overflow bit, V (bit 28); carry/borrow/extend bit, C (bit 29); zero bit, Z (bit 30); and negative/less than bit, N (bit 31).

Register banking refers to providing multiple copies of a register at the same address. FIQ stands for Fast Interrupt reQuest, and it is basically a higher priority interrupt. FIQ will always have precedence over regular interrupts; regular interrupts won't mask or interrupt an FIQ, while an FIQ will mask or interrupt any IRQ (Interrupt reQuest).

One cannot conclude even an overview of ARM architecture without mentioning the TrustZone, a technology to build a Trusted Execution Environment (TTE) in a cost-effective manner without complicating the development of different components of a System on a Chip (SoC), see also Section 3.7. Traditional isolation for a system with only one Operating System (OS) managing system resources and functionality is MMU-based. The Memory Management Unit (MMU) splits memory into regions isolated from each other, but this solution does not work well for mobile device with several cores.

TrustZone provides a hardware solution for separating a rich operating system (OS), supporting a massive set of features and consisting of a massive amount of code, from a considerably smaller but secure OS hosting the TTE. A firewall separates the TTE from a normal environment running general code. This solution allows independent bus masters, possibly running different OS on different cores, to support the isolation required by security as shown in Fig. 3.3 (from https://www.trustonic.com/technical-articles/what-is-trustzone). Moreover, the different operating systems do not need to be modified for the implementation of a security subsystem; at the same time, TrustZone encourages applications to separate security from general purpose aspects.

ARM implementation is hardwired without microcode. ARM architecture is cheap to produce and has low energy consumption and astounding performance. It is not surprising that more than 150 billion ARM microprocessors have been shipped since 2019, roughly 20 for every inhabitant of planet Earth [257]!! Once dominant, Intel's x86-64 architecture now lags behind ARM in popularity, as shown by Table 3.2.

ARM Ltd. founded in 1990 is located outside Cambridge, UK, and is a joint venture of Acorn Computing, Apple, and VLSI Technology. ARM licenses the architecture to companies who design and produce the microprocessors after paying the ARM licenses; in FY 2018, ARM received US $1.8 billion from royalties [257]. The 2020 acquisition of ARM by NVIDIA for US $40 billion is likely to have a significant impact on the computer-game industry and expanding AI computing in computer clouds and beyond.

This outline aims to offer convincing arguments for the elegance, power, and energy consumption frugality rather than an in-depth presentation of an architecture designed to be flexible, supporting virtualization, real-time execution, and security.

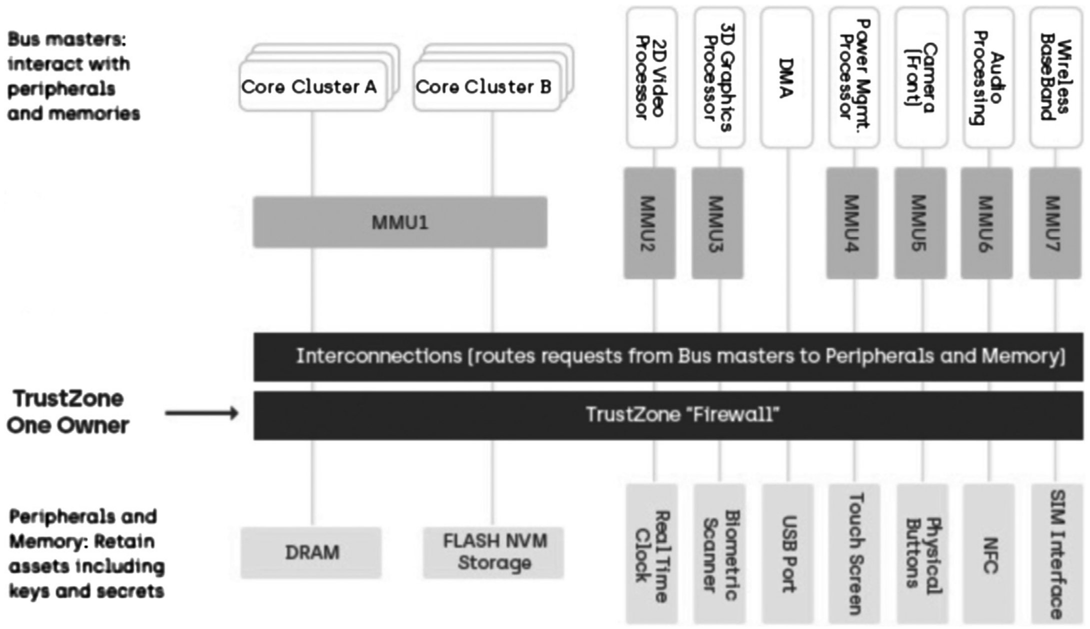


FIGURE 3.3

ARM hardware solution for security. The TrustZone Firewall allows independent bus masters to support isolation.

Table 3.2 Shipments of ARM and x86-64 units [408].

Year	2010	2011	2012	2013	2014	2015	2016
ARM (in billions)	6.1	7.9	8.7	10.4	12.0	14.8	17.1
x86-64 (in millions)	8.9	9.52	9.67	9.89	10.09	11.09	11.1

3.4 SIMD architectures

SIMD architectures have significant advantages over the other systems described by Flynn's taxonomy. Some of these advantages are:

a. Exploit a significant level of data-parallelism. Enterprise applications in data mining and multimedia applications, as well as the applications in computational science and engineering using linear algebra, benefit the most.
b. Allow mobile device to exploit parallelism for media-oriented image and sound processing using SIMD extensions of traditional ISA.
c. Are more energy efficient than MIMD architecture. Only one instruction is fetched for multiple data operations, rather than fetching one instruction per operation.
d. Have a higher potential speedup than MIMD architectures. SIMD potential speedup could be twice as large as that of MIMD.
e. Allow developers to continue thinking sequentially.

Three flavors of the SIMD architecture are encountered in modern processor design: vector architecture; SIMD extensions for mobile systems and multimedia applications; and Graphics Processing Units.

Vector architectures use vector registers holding 64, 128, 256, or more vector elements. Vector functional units carry out arithmetic and logic operations using data from vector registers as input and disperse the results back to memory. Vector load-store units are pipelined, hide memory latency, and leverage memory bandwidth. The memory system spreads access to multiple *memory banks*, which can be addressed independently.

Vector length registers allow handling of vectors when the number of elements is not a multiple of the physical vector registers size, e.g., a vector with 100 elements when the vector register can only contain 64 vector elements. *Vector mask registers* disable/select vector elements and are used by conditional statements that select specific vector elements.

Non-adjacent vector elements of a multidimensional array can be loaded into a vector register by specifying the *stride*, the distance between elements to be gathered in one register. Scatter-gather operations support processing of sparse vectors. A *gather* operation takes an *index vector* and fetches the vector elements at the addresses given by adding a base address to the offsets given by the index vector; as a result, a dense vector is loaded in a vector register. A *scatter* operation does the inverse, it scatters the elements of a vector register to addresses given by the index vector and the base address.

Chaining allows vector operations to start as soon as individual elements of vector source operands become available and operate on *convoys*, sets of vector instructions that can potentially be executed together. Multiple *lanes* process several vector elements per clock cycle. Each lane contains a subset of the vector register file and one execution pipeline from each functional unit.

SIMD extensions for multimedia applications. The name of this class of SIMD architectures reflects the basic architectural philosophy—augmenting an existing instruction set of a scalar processor with a set of vector instructions. SIMD extensions have obvious advantages over vector architecture:

1. Low cost to add circuitry to an existing ALU.
2. Little extra state is added, thus the extensions have little impact on context-switching.
3. Need little extra memory bandwidth.
4. Do not pose additional complications to the virtual memory management for cross-page access and page-fault handling.

Multimedia applications often run on mobile devices and operate on narrower data types than the native word size. For example, graphics applications use three 8-bit for colors and one 8-bit for transparency, audio applications use 8, 16, or 24-bit samples. To accommodate narrower data types, carry chains have to be disconnected. For example a 256-bit adder can be partitioned to perform simultaneously 32, 16, 8 or 4 additions on 8, 16, 32, or 64 bit, respectively. The instructions *opcode* now encode the data type and neither sophisticated addressing modes supported by vector architectures, such as stride-base addressing or scatter-gather, nor mask registers, are supported.

Intel extended its x86-64 instruction set architecture, and in 1996 introduced MMX (Multi-Media Extensions) which supports eight 8-bit, or four 16-bit integer operations. MMX was followed by multiple generations of streaming SIMD extensions (SSE) in 1999 and ending with SSE4 in 2007. The SSEs operate on eight 8-bit integers, four 32-bit or two 64-bit either integer or floating-point operations.

AVX (Advanced Vector Extensions) introduced by Intel in 2010 operates on four 64-bit either integer or floating-point operations. Several members of the AVX family of Intel processors are: Sandy

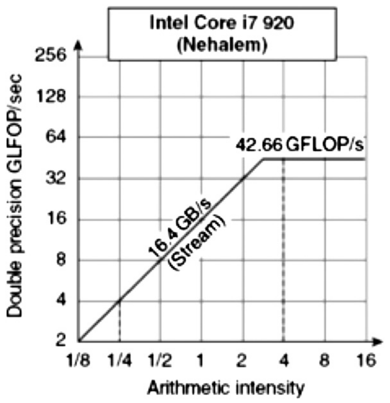


FIGURE 3.4

Roofline performance model for Intel i7 920. When $ArI < 3$, the memory bandwidth of 16.4 GB/sec is the bottleneck. The processor delivers 42.66 Gflops, and this limits the performance of applications with arithmetic intensity larger than three.

Bridge, Ivy Bridge, Haswell, Broadwell, Skylake, and its follower, the Baby Lake, announced in August 2016. AMD offers several processor families with multimedia extensions, including the Steamroller.

Floating-point performance models for SIMD architecture. The gap between the processor and the memory speed, though bridged by different level of caches, is still a major factor affecting the performance of many applications. Applications displaying low spatial and temporal locality are particularly affected by the gap. The effects of this gap are also most noticeable for SIMD architecture and floating-point operations. *Arithmetic intensity* (ArI), defined as the number of floating-point operations per byte of data read, is used to characterize application scalability and to quantify the performance of SIMD systems.

Arithmetic intensity of applications involving dense matrices is high, and this means that dense matrix operations scale with problem size, while sparse matrix applications have a low arithmetic intensity and do not scale well with the problem size. Applications involving spectral methods and FFT (Fast Fourier Transform) have an average arithmetic intensity.

The *roofline* model captures the fact that the performance of an application is limited by its arithmetic intensity and the memory bandwidth. A graph depicting the floating-point performance function of the arithmetic intensity is shown in Fig. 3.4. Memory bandwidth limits the performance at low arithmetic intensity, and this effect is captured by the sloped line of the graph. As arithmetic intensity increases, the floating-point performance of the processor is the limiting factor that is captured by the straight line of the graph.

3.5 Graphics processing units

The desire to support real-time graphics with vectors of two, three, or four dimensions led to the development of Graphics Processing Units, which are very efficient at manipulating computer graphics. GPUs produced by NVIDIA, and AMD/ATI are also used in embedded systems, mobile phones, personal computers, workstations, and game consoles. GPU processing is based on a heterogeneous execution model with a CPU acting as the *host* connected with a GPU called the *device*.

The highly parallel structures of GPUs are based on SIMD execution and support parallel processing of large blocks of data. A GPU has multiple *multithreaded SIMD* processors. The current generation of GPUs, e.g., Fermi of NVIDIA, have 7–15 multithreaded SIMD processors. Compared with vector processors, each multithreaded SIMD processor has several wide and shallow SISD *lanes*. For example, an NVIDIA GPU has 32 768 registers divided among the 16 physical SIMD lanes; each lane has 2 048 registers.

A typical execution includes several steps: (i) CPU copies the input data from the main memory to the GPU memory; (ii) CPU instructs the GPU to start processing using the executable in the GPU memory; (iii) GPU uses multiple cores to execute the parallel code; and (iv) when done, the GPU copies the result back to the main memory.

The GPU programming model is Single-Instruction-Multiple-Thread (SIMT). GPUs are often programmed in CUDA, a C-like programming language developed by NVIDIA, the pioneer of GPU-accelerated computing. All forms of GPU parallelism are unified as CUDA threads. In the SIMT model a *thread* is associated with each data element. Thousands of CUDA threads can run concurrently. Threads are organized in *blocks*, groups of 512 vector elements, and multiple blocks form a grid. Fig. 3.5 shows the grid representing a vector A with 8 192 components; there are 16 blocks, each block has 16 SIMD threads; each thread operates on 32 elements of the vector.

A two-level scheduling mechanisms assigns threads to the multiple *lanes* of a multithreaded SIMD processor. A *thread block scheduler* assigns thread blocks to multithreaded SIMD processors, and then a *tread scheduler* assigns threads to SIMD *lanes*. Fig. 3.6 illustrates scheduling for the grid in Fig. 3.5. The thread blocks must be able to run independently. NVIDIA GPU memory has the following organization:

a. Each SIMD lane has an off-chip private memory for stack frame, spilling registers, private variables, and GPU data in L1 and L2 caches.
b. Each multithread SIMD processor has on-chip local memory shared by all its lanes, thus, by the threads within the block scheduled on the processor.
c. GPU memory. The host can read from and write to this off-chip memory.

In 2018, Nvidia and AMD controlled nearly 100% of the GPU market. Nvidia launched RTX 20 series in 2018 with ray-tracing cores to improve performance on lighting effects. One of the most powerful GPUs produced by NVIDIA is GEFORCE RTX 3090 with a 1.7 GHz clock, 24 GB of memory with a 384-bit memory interface width and 10 496 cores. In 2019 AMD launched GCN (Graphics Core Next) ISA and the RX 6000 series.

It should not be surprising that cloud service providers now offer GPU instances. For example, AWS EC2 P4 instances feature up to eight NVIDIA A100 Tensor Core GPUs with 600 GB/s peer to peer GPU, and communication with NVIDIA NVSwitch and 400-Gbps instance networking with support for Elastic Fabric Adapter (EFA) and NVIDIA GPUDirect RDMA (remote direct memory access).

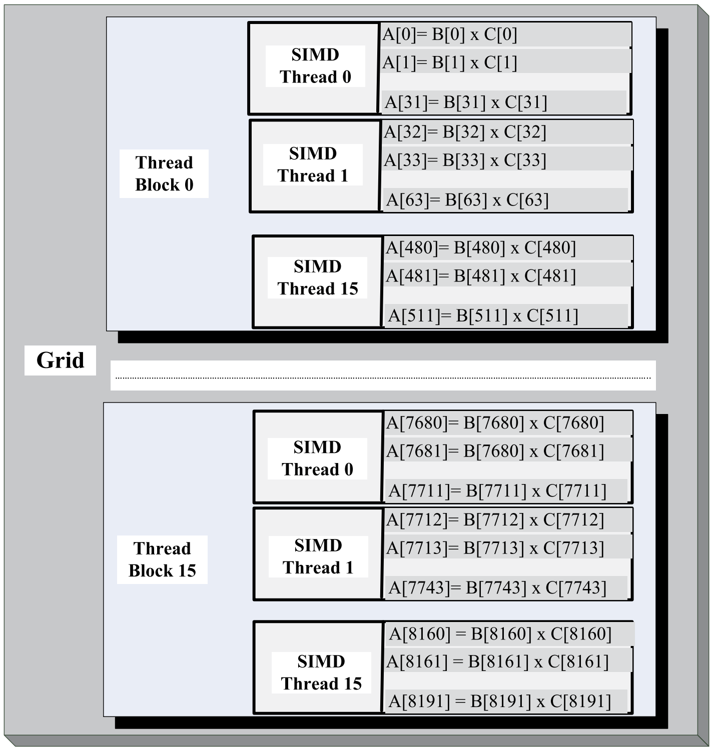


FIGURE 3.5

Grid, blocks, and threads. Grids with 8 192 components for vectors A, B, C. There are 16 blocks with 512 vector components each. Each bloc has 6 threads, and each thread operates on 32 vector elements.

3.6 Tensor processing units

For more than half a century, Moores's Law enabled major architectural enhancements including deep pipelines, branch prediction, out-of-order instruction execution, speculative prefetching, multithreading, deep memory hierarchies, and wide SIMD units. In the first decade of the new millennium, it became apparent that the end of an era was in sight and the age of Domain Specific Architectures (DSA) began.

With the end of Moore's Law is in sight, Dennard's Law has already limited processor clock rate. Moore's Law, formulated in 1965 by one of Intel's founders, Gordon Moore, states that the number of transistors per chip at constant cost doubles every two years. This empirical, rather than physical law, has often been confused as stating that processor performance doubles every two years because transistor size and speed are related, the smaller the transistor the higher its switching speed. In 1974, Dennard observed that voltage and current should be proportional to the linear dimensions of a transistor. Power is proportional to transistor area and, as transistors get smaller, power density increases. Since around 2006, Dennard's Law has limited processor clock rate to 4 GHz or less.

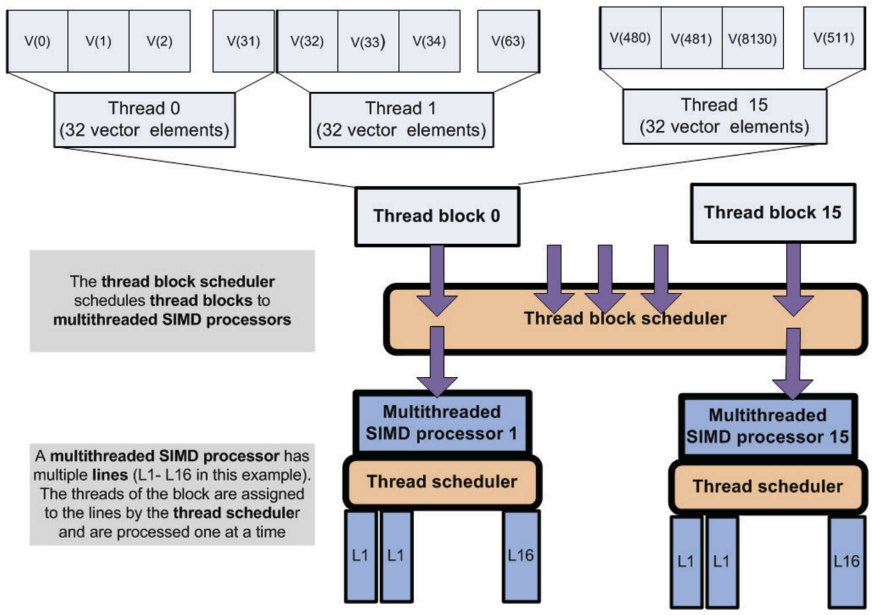

V(0)
V(1)
V(2)
V(31)
V(32)
V(33)
V(34)
V(63)
V(480)
V(481)
V(8130)
V(511)
Thread 0 (32 vector elements)
Thread 1 (32 vector elements)
Thread 15 (32 vector elements)
Thread block 0
Thread block 15
The thread block scheduler schedules thread blocks to multithreaded SIMD processors
Thread block scheduler
Multithreaded SIMD processor 1
Multithreaded SIMD processor 15
A multithreaded SIMD processor has multiple lines (L1- L16 in this example). The threads of the block are assigned to the lines by the thread scheduler and are processed one at a time
Thread scheduler
Thread scheduler
L1
L1
L16
L1
L1
L16

FIGURE 3.6

The execution for the grid in Fig. 3.5. The *thread block scheduler* assigns thread blocks to multithreaded SIMD processors. A *thread scheduler* running on each multithreaded SIMD processor assigns threads to the SIMD *lanes* of the processors.

The DSA objective is to extract performance from software oblivious to system architecture. In 2015, Google deployed the first custom Application Specific Integrated Circuit (ASIC) for its cloud computing infrastructure, a Tensor Processing Unit. TPUs were designed for Machine Learning (ML) applications, in particular, for inference stages of Deep Neural Networks (DNNs). TPUs were followed by Microsoft Catapult, a Field-Programable Gate Array (FPGS), Nervana's ASIC called CREST, and Google's own Pixel Visual Core.

TPU design aimed to achieve at least one-order-of-magnitude performance improvement versus a GPU. TPUs are single-threaded processors, based on a deterministic execution model, a good match to the 99th-percentile response-time requirement of a typical DNN inference application [262]. TPUs are designed to be plugged into existing servers as co-processors. The host CPU sends TPU instructions over the PCIe bus into an instruction buffer. The internal blocks are typically connected together by 256-byte-wide (2048-bits) paths. TPU organization is shown in Fig. 3.7; TPA ISA includes the following instructions:

Read_Host_Memory—reads data from the CPU host memory into Unified Buffer.
Write_Host_Memory—writes data from the Unified Buffer into CPU host memory.
Read_Weights—reads weights from Weight Memory into Weight FIFO as input to Matrix Unit.

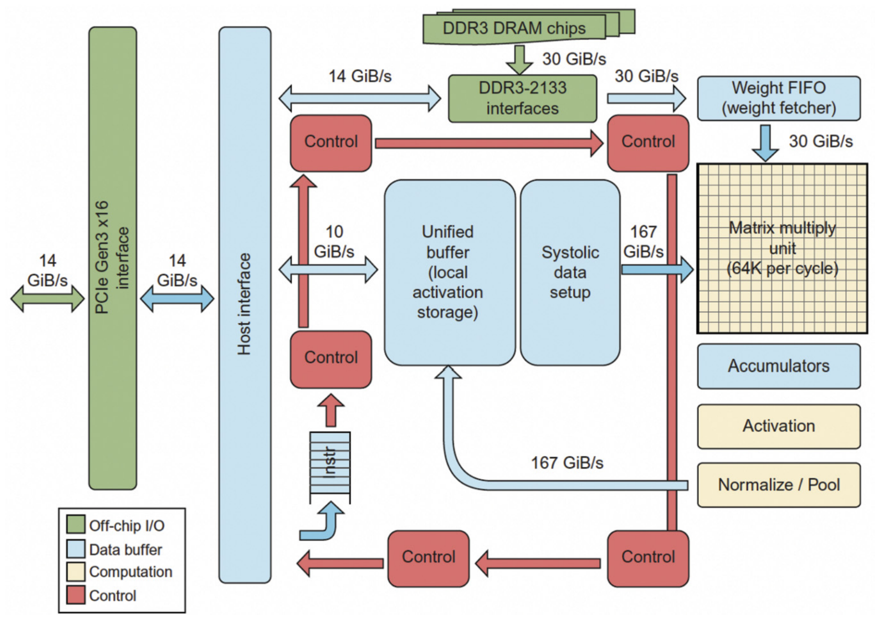


FIGURE 3.7

TPU organization and its connections to the CPU via a host interface and a 14 GiB/sec PCIe gen3 x 16 bus and with the DDR3 Dram memory via a 30 GiB/sec memory bus. The Matrix Multiply Unit gives the TPU its enormous computational power. The activation unit performs nonlinear functions, such as sigmoid on data supplied by accumulators, and transmits them to the unified buffer [262].

MatrixMultiply/Convolve—causes the Matrix Multiply Unit to perform a matrix-matrix multiply, a vector-matrix multiply, an element-wise matrix multiply, an element-wise vector multiply, or a convolution from the Unified Buffer into the Accumulators.
Activate—performs nonlinear function of the artificial neuron, with options for ReLU, Sigmoid, tanh, and so on. Its inputs are the Accumulators, and its output is the Unified Buffer.

Application code running on the TPU is typically written in TensorFlow and is compiled into an API that can run on GPUs or TPUs. A third-generation TPU, the Edge TPU, much smaller and using far less power compared to Cloud TPUs, was released in 2018. The Edge TPU is capable of four trillion operations per second while using 2W and runs a version of Tensor Flow called TensorFlow Lite.

A 2017 paper [262] reports on performance analysis of a datacenter TPU with a 65 536 8-bit MAC (multiply-accumulate unit) matrix multiply unit with a peak throughput of 92 TeraOps/second (TOPS) and a 28 MiB software-managed on-chip memory. It concludes that "TPU leverages the order-of-magnitude reduction in energy and area of 8-bit integer systolic matrix multipliers over 32-bit floating-point data paths of a K80 GPU to pack 25 times as many MACs (65 536 8-bit vs. 2 496 32-bit) and 3.5 times the on-chip memory (28 MiB vs. 8 MiB) while using less than half the power of the K80 in a relatively small die."

3.7 Systems on a chip

A system on a chip (SoC) is an integrated circuit hosting on the same substrate/chip multiples cores, including a mix of traditional CPUs, alongside GPUs, TPUs, other types of functional units, e.g., DSPs (Digital Signal Processors), memory, input/output ports, secondary storage, and sometimes WiFi and cellular modems. SoC organization improves performance and reduces power consumption, as well as semiconductor die area, compared with motherboard-based architecture with equivalent functionality but discrete components.

All this is possible due to cutting-edge technologies such as the five-nanometer process, which can populate the chip with billions of transistors. The trend to processor customization reflects the reality that Moore's Law is slowing down and power dissipation limits the clock rates [408]. The side effect of this organization is increased replacement cost. Once any component of an SoC becomes inoperative, the entire SoC must be replaced.

SoC processor cores typically use RISC instruction-set architectures and in particular ARM due to its advantages discussed in Section 3.3. SoC memory consists of a memory hierarchy and cache hierarchy including read-only memory (ROM), random-access memory (RAM), Electrically Erasable Programmable ROM (EEPROM), and flash memory. SRAM is used to implement processor registers and cores' L1 caches, whereas DRAM is used for low level(s) of memory hierarchy discussed in Section 3.2. SoC units communicate with one another using data bus architectures, often based on ARM's royalty-free Advanced Microcontroller Bus Architecture (AMBA) standard and, more recently, sparse intercommunication networks known as networks-on-a-chip (NoC).

SoCs are optimized to maximize power efficiency quantified by performance per watt. Applications, such as edge computing, distributed processing, and ambient intelligence, require high-performance computing SoCs with high-power efficiency, while power is limited for SoCs used for mobile device such as smartphones, watches, tablets, laptops, or GPS. The five-fold discrepancy between battery density improvements and Moore's law implies there is a widening gap between compute requirements and energy availability. Increasing power density leads to thermal challenges exceeding the inherent passive cooling capability of a mobile device. All SoCs minimize energy dissipation because the heat from high energy consumption of one unit can damage other circuit components. Mobile devices operate under strict thermal and energy budgets.

Apple A14 Bionic is a 64-bit ARMv8.5a SoC with 64-bit six-core CPU with two high-performance cores and four energy-efficient GPU cores for faster graphics. It is manufactured by TSMC (Taiwan Semiconductor Manufacturing) using their first-generation 5-nm fabrication process. The transistor density is 134 million transistors per m2, for a total of 11.8 billion transistors. This SoC includes a custom silicon block called the Neural Engine, hardware tailor-made to perform operations used in machine learning and AI. A14 has also a Neural Engine with 16 cores delivering 11 trillion operations per second.

Another processor, M1, inspired by A14, is the first ARM-based SoC designed as CPU for Apple computers. M1 has four high-performance and four energy-efficient cores, The high-performance cores have 192 KB of L1 instruction cache and 128 KB of L1 data cache and share a 12 MB L2 cache. The energy-efficient cores have a 128 KB L1 instruction cache, 64 KB L1 data cache, and shared 4 MB L2 cache and consume one tenth of the energy consumed by high-performance cores. M1 has a GPU with eight cores; each one contains eight *executions units* (Apple terminology for the multithreaded SIMD processors in Fig. 3.6), which in turn contain eight ALUs per execution unit. M1's GPU can execute nearly 25 000 threads simultaneously and has a maximum performance of 2.6 Tflops.

The computing power of today SoCs, even of those for mobile devices, is on par with, or higher than, that of supercomputers of two decades ago and can carry out complex computational tasks locally with lower power consumption and without the need to transfer data to the cloud. Computer clouds are in a symbiotic relationship not only with billions of mobile devices used to access data stored on clouds but also with systems outside the cloud infrastructure and carrying out computational tasks together with clouds as edge computing, also discussed in Section 12.12. Edge computing is a distributed computing framework that brings enterprise applications closer to data sources, such as IoT devices or local edge servers. This proximity to data at its source can deliver strong benefits: faster insights, improved response times, and better bandwidth availability.

3.8 Data, thread-level, and task-level parallelism

As demonstrated by nature, the ability to work as a group and carry out many tasks in parallel represents a very efficient way to reach a common goal. Thus, we should not be surprised that the thought that individual computer systems should work in parallel for solving complex applications was formulated early on, at the dawn of the computer age. Parallel processing allows us to solve large problems by splitting them into smaller ones and solving them concurrently. Parallel processing was considered for many years the holy grail for solving data-intensive problems encountered in many areas of science, engineering, and enterprise computing and required major advances in several areas, including algorithms, programming languages and environments, performance monitoring, computer architecture, interconnection networks, and, last but not least, solid-state technologies. Often discovering parallelism is quite challenging, and the development of parallel algorithms requires a considerable effort.

Fine-gained versus coarse-grained parallelism. We distinguish *fine-grained* from *coarse-grained* parallelism. In the former case only relatively small blocks of the code can be executed in parallel without the need to communicate or synchronize with other threads or processes, whereas in the latter case large blocks of code can be executed concurrently.

Numerical computations involving linear algebra operations exhibit fine-grained parallelism. For example, many numerical analysis problems, such as solving large systems of linear equations, or solving systems of Partial Differential Equations (PDEs) require algorithms based on domain decomposition methods. The speedup of applications displaying fine-grained parallelism is considerably lower that those of coarse-grained applications. Indeed, even systems with a fast interconnect processor speed are orders of magnitude higher than communication speed.

Concurrent processes or threads communicate using message-passing or shared-memory. Shared-memory is the defining attribute of distributed shared-memory multiprocessor systems. Shared-memory is also used by multicore processors where each core has private L1 instruction and data caches, as well as an L2 cache, and all cores share the L3 cache. Shared-memory is not scalable, thus, it is seldom used in supercomputers and large clusters. Message passing is used exclusively in large-scale distributed systems, and our discussion is restricted to this communication paradigm. Debugging a message-passing application is considerably easier than debugging a shared-memory one.

Shared-memory is extensively used by system software. The system stack is an example of shared-memory used to save the state of a process or thread at the time of a context switch. The kernel of an operating system uses control structures, such as processor and core tables for multiprocessor and multicore system management, process and thread tables for process and thread management, page tables for virtual memory management, among others. Multiple application threads running on a multicore processor often communicate via the shared-memory of the system.

Data-level parallelism. This is an extreme form of coarse-grained parallelism. It is based on partitioning data into large chunks/blocks/segments and running concurrently either multiple programs or copies of the same program, each on a different data block. In the latter case the paradigm is called *Same Program Multiple Data* (SPMD). There are the so-called *embarrassingly parallel* problems where little or no effort is needed to extract parallelism and to run a number of concurrent tasks with little or no communication among them.

Assume that we wish to search for the occurrence of an object in a set of n images or of a string of characters in n records. Such a search can be conducted in parallel. In all these instances the time required to carry out the computational task using N servers is reduced by a factor of N, and the speedup is almost linear to the number of servers used. This type of data-parallel applications is at the heart of enterprise computing on computer clouds and will be discussed in depth in Chapter 11. The MapReduce programming model will be presented in Section 11.5, followed by the discussion of Hadoop and Yarn in Section 11.7.

Decomposition of a large problem into a set of smaller problems that can be solved concurrently is sometimes trivial and can be implemented in hardware, a topic discussed in Section 3.1. For example, assume that we wish to manipulate the image of a three-dimensional object represented as a *3D* lattice of $n \times n \times n$ points. To rotate the image, we apply the same transformation to each one of the n^3 points. Such a transformation can be done by a *geometric engine*, a processor designed to carry out the transformation of a subset of n^3 points concurrently. Graphics Processing Units (GPUs) discussed in Section 3.5 initially designed as graphics engines are now widely used by data-intensive applications.

Thread-level and task-level parallelism. The term thread-level parallelism is overloaded. In the computer architecture literature it is used for data-parallel execution using a GPU. In this case a *thread* is a subset of vector elements processed by one of the lanes of a multithreaded processor, see Section 3.5. *Hyper-threading* is used to describe multiple execution threads possibly running concurrently, but on a single core, see Section 3.1. Threads are also used in a multicore processor to run multiple processes concurrently. Database applications are memory-intensive and I/O-intensive; and multiple *threads* are used to hide the latency of memory and I/O access.

In the context of scheduling, a job consists of multiple tasks scheduled either independently or co-scheduled when they need to communicate with one another. Often fine-grained execution units are given control of resources for a relatively short time to guarantee a low-latency response time.

Cloud computing is an appealing environment of running applications attempting to identify optimal model parameters that best fit experimental data. Such applications involve computationally intensive tasks. Multiple instances running concurrently test the fitness of different sets of parameters, and the results are then compared to determine the optimal set of model parameters.

There are also numerical simulations of complex systems that require an optimal design of a physical system. Multiple design alternatives are compared, and the optimal one is selected according to several optimization criteria. Consider for example the design of a circuit using Field Programmable Gate Arrays (FPGAs). An FPGA is an integrated circuit designed to be configured by the customer using a hardware description language (HDL), similar to that used for an application-specific integrated circuit (ASIC).

As multiple choices for the placement of components and for interconnecting them exist, the designer could run concurrently N versions of the design choices and choose the one with the best performance, e.g., minimum power consumption. Alternative optimization objectives could be to reduce cross-talk among the wires or to minimize the overall noise. Because each alternative configuration requires hours, or maybe days, of computing, running them concurrently reduces the design time considerably.

3.9 Speedup, Amdhal's law, and scaled speedup

Parallel hardware and software systems allow us to solve problems demanding more resources than those provided by a single system and, at the same time, to reduce the time required to obtain a solution. There are applications, the so-called *embarrassingly parallel applications*, when the *parallel execution time,* the time when running on N cores is $1/N$ of the *sequential execution time*, the time when running on a single core. There are other applications when the parallel execution time is only slightly smaller than the sequential execution.

This begs questions such as: Is there a way to quantify the amount of parallelism that can be extracted from an application?; what is the largest number of cores that can be used effectively to run an application?; how does one measure the parallelization effectiveness?; and does the amount of data processed by the application matter? These questions were addressed in 1960s by Gene Amdahl and more recently by Gustafson and are discussed in this and the next section. The effectiveness of parallelization is measured by the speedup. In the general case the *speedup* of the parallel computation is defined as

$$S(N) = \frac{T(1)}{T(N)}, \tag{3.1}$$

with $T(1)$ the execution time of the sequential computation and $T(N)$ the execution time when N parallel computations are carried out.

Amdahl's Law. Gene Myron Amdahl[4] was a theoretical physicist turned computer architect best known for Amdahl's Law. In a 1967 seminal paper [21] Amdhal argued that the fraction of a computation that is not parallelizable is significant enough to favor single-processor systems. He reasoned that large-scale computing capabilities can be achieved by enhancing the performance of single processors, rather than building multiprocessor systems.

Though this thesis was disproved, Amdahl's Law is a fundamental result used to predict the theoretical maximum speedup for a program using multiple processors. This law states that *the portion of the computation that cannot be parallelized determines the overall speedup.* If α is the fraction of running time a sequential program spends on non-parallelizable segments of the computation, then the maximum speedup achievable S is

$$S = \frac{1}{\alpha}. \tag{3.2}$$

[4] Gene Amdhal contributed significantly to the development of several IBM systems, including System/360, and in the 1970s, started his own company, Amdahl Corporation, to produce high-performance systems.

To prove this result, call σ the sequential time and π the parallel time and start from the definitions of $T(1)$, $T(N)$, and $\alpha = \sigma/(\pi + \sigma)$:

$$T(1) = \sigma + \pi, \quad T(N) = \sigma + \frac{\pi}{N}, \quad \text{and} \quad \alpha = \frac{\sigma}{\sigma + \pi}. \tag{3.3}$$

Then

$$S = \frac{T(1)}{T(N)} = \frac{\sigma + \pi}{\sigma + \pi/N} = \frac{1 + \pi/\sigma}{1 + (\pi/\sigma) \times (1/N)}. \tag{3.4}$$

But

$$\pi/\sigma = \frac{1 - \alpha}{\alpha}. \tag{3.5}$$

Thus, for large N,

$$S = \frac{1 + (1 - \alpha)/\alpha}{1 + (1 - \alpha)/(N\alpha)} = \frac{1}{\alpha + (1 - \alpha)/N} \approx \frac{1}{\alpha}. \tag{3.6}$$

An alternative formulation of Amdahl's Law is that, if a fraction f of a computation is enhanced by a speedup S, then the overall speedup is

$$S_{overall}(f, S) = \frac{f}{(1 - f) + \frac{f}{S}} \quad \text{or} \quad S_{overall}(f, S) = \frac{1}{\frac{1}{f} + \frac{1}{S} - 1}. \tag{3.7}$$

Scaled speedup. Amdahl's Law applies to a *fixed problem size*; in this case, the amount of work assigned to each one of the parallel processes decreases when the number of processes increases, and this affects the efficiency of the parallel execution. When the problem size is allowed to change, Gustafson's Law gives the *scaled speedup* with N parallel processes as

$$S(N) = N - \alpha(N - 1). \tag{3.8}$$

As before, we call σ the sequential time; now π is the *fixed parallel time per process*; α is given by Eq. (3.3). The sequential execution time, $T(1)$, and the parallel execution time with N parallel processes, $T(N)$, are

$$T(1) = \sigma + N\pi \quad \text{and} \quad T(N) = \sigma + \pi. \tag{3.9}$$

Then, the scaled speedup is

$$S(N) = \frac{T(1)}{T(N)} = \frac{\sigma + N\pi}{\sigma + \pi} = \frac{\sigma}{\sigma + \pi} + \frac{N\pi}{\sigma + \pi} = \alpha + N(1 - \alpha) = N - \alpha(N - 1). \tag{3.10}$$

Amdahl's Law expressed by Eq. (3.2) and the *scaled speedup* given by Eq. (3.8) assume that all processes are assigned the same amount of work. The scaled speedup assumes that the amount of work assigned to each process is the same regardless of the problem size. Then, to maintain the same execution time, the number of parallel processes must increase with the problem size. The scaled speedup captures the essence of efficiency, namely, that the limitations of the sequential part of a code can be balanced by increasing the problem size.

3.10 Multicore processor speedup

We now live in the age of multicore processors brought about by the limitations imposed on solid-state devices by the laws of physics. Increased power dissipation due to faster clock rates makes the heat removal more challenging, and this implies that in the future, we could expect only a modest increase of the clock rate. Multicore processors use the billions of transistors on a chip to deliver significantly higher computing power and process more data every second. Yet, the ability to get data in and out of the chip is limited by the number of pins. Increasing the number of cores on a chip faces its own physical limitations.

There are alternative designs of multicore processors, and the next question is to investigate chip configurations most useful for applications exhibiting a limited parallelism. The cores can be identical or different from one another, or there could be a few powerful cores or a larger number of less powerful cores. Theoretically, the cores could be configured automatically or be immutable.

More cores will lead to a large speedup of highly parallel applications; a powerful core will favor highly sequential applications. If feasible, a chameleonic system will adapt to the level of parallelism, though changing the core configuration will incur overhead and challenge application developers. Even considering re-configuration overhead, the speedup of automatic core configuration will be superior to either symmetric or asymmetric core design.

The design space of multicore processors should be driven by cost–performance considerations. A design will be cost-effective if the speedup achieved will exceed the *cost up* defined as the multicore processor cost divided by the single-core processor cost. The cost of a multicore processor depends on the number of cores and the complexity, ergo, the power of individual cores.

The *Basic Core Equivalent* (BCE) concept was introduced to quantify the resources of individual cores. A *symmetric core* processor may have n BCEs with r resources each. Alternatively, $n \times r$ resources may be distributed unevenly in an *asymmetric core* processor.

A quantitative analysis of design choices based on an extension of Amdahl's Law to multicore processors is presented in [238]. We expect this analysis to confirm the obvious, that is: *The larger the parallelizable fraction f of an application, the larger the speedup.* This analysis is based on a number of simplifying assumptions:

i. A number of factors such as the chip area, the power dissipation, or combinations of these two with other factors limit the number of cores to n BCE. These limitations consider only on-chip resources. Off-chip resources such as shared caches, memory controllers, or interconnection networks are assumed to be approximately identical for the alternative designs.

ii. The performance of a single BCE core is the unity. When the chip is limited to n BCEs, all cores are identical and the performance of each core is r BCE, then the total number of cores on a chip is $\lceil n/r \rceil$. Fig. 3.8 shows a symmetric 16-BCE chip with two configurations: one with sixteen 1-BCE cores and the other with one 16-BCE core.

iii. The sequential performance of r BCEs is denoted as $perf(r)$. When $perf(r) > r$, the speedup of both sequential and parallel execution increases; therefore, the designers should increase as much as feasible the core resources and implicitly the individual core performance. On the other hand, when $perf(r) < r$, increasing individual core performance increases the performance for sequential execution but lowers that of the parallel execution. Therefore, the following analysis is focused on the case when $perf(r) < r$. A good model is $perf(r) = \sqrt{r}$ when the performance doubles, triples, and quadruples for 4, 9, 16 cores, respectively.

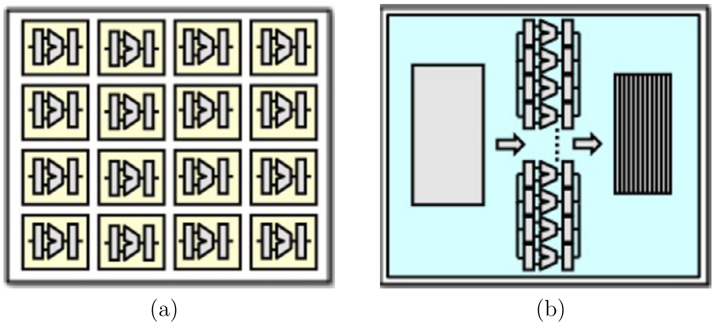


FIGURE 3.8

16-BCE chip. Symmetric core processor with two different configurations: (a) sixteen 1-BCE cores; (b) one 16-BCE core.

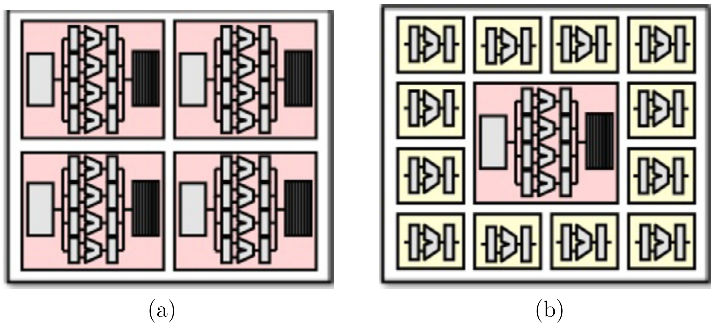


FIGURE 3.9

16-BCE chip. (a) Symmetric core processor with four 4-BCE cores; (b) Asymmetric core processor with 1 4-BCE core and 12 1-BCE cores.

The first case analyzed assumes a symmetric multicore chip. There are n/r cores on the chip; for example, when $n = 32$, the chip could have 16 cores of 2 BCE each, 8 cores of 4 BCE each, and so on. Call f the parallelizable fraction of a computation. One core will run the $(1 - f)$ sequential component of the computation, and n/r cores will run the parallel component of the computation, f, with a performance $perf(r)$. According to Eq. (3.7) the speedup will be

$$Speedup_{symcore}(f, n, r) = \frac{1}{\frac{1-f}{perf(r)} + \frac{f \cdot r}{n \cdot perf(r)}}. \tag{3.11}$$

In an asymmetric multicore processor, more powerful cores will coexist with less powerful ones. Fig. 3.9 illustrates the differences between symmetric and asymmetric cores; the asymmetric core processor has a 4-BCE core and 12 1-BCE cores. The sequential performance will benefit from the more powerful core running four times faster, while the parallel performance is $perf(r)$ from the 4-BCE core and *one* each from the remaining $(n-r)$, in our case, 12 1-BCE cores. The speedup with one powerful core and $(n-r)$ 1-BCE cores is then

$$Speedup_{asymcore}(f,n,r) = \frac{1}{\frac{1-f}{perf(r)} + \frac{f}{perf(r)+n-r}}. \tag{3.12}$$

A dynamic multicore chip could configure its n BCE depending on the fraction f of a particular application. If the sequential component of the application, $(1-f)$ is large, then configure the chip as one r-BCE core, while in parallel mode use all base cores in parallel. In this case

$$Speedup_{dyncore}(f,n,r) = \frac{1}{\frac{1-f}{perf(r)} + \frac{f}{n}}. \tag{3.13}$$

What non-obvious conclusions can be drawn from this analysis? First, the speedup of asymmetric multicore processors is always larger and, in some cases, could be significantly larger than the speedup of symmetric core processors. For example, the largest speedup when $f = 0.975$ and $n = 1\,024$ is achieved for a configuration with one 345-BCE core and 679 1-BCE cores. Second, increasing the power of individual cores is beneficial even for symmetric core processors. For example, when $f = 0.975$ and $n = 256$, the maximum speedup occurs for seven 1-BCE cores.

Not to be forgotten should be the fact that task scheduling on asymmetric and dynamic multicore processors will be fairly difficult. There are also other factors affecting the performance ignored by the simple model discussed in [238].

3.11 From supercomputers to distributed systems

Computational science and engineering applications require high-performance computing systems able to exploit fine-grained parallelism. The stringent requirements of such applications have motivated many of the architectural advancements we see in modern processors. For example, the supercomputers of the mid 1960s, CDC 6400, 6500, and 7600 manufactured by Control Data Corporation (CDC), pioneered pipelining and multiple functional units for addition, multiplication, and other arithmetic and logic operations. In 1969, IBM unsuccessfully entered the scientific computing market with IBM 360 model 91, which had novel features including an implementation of Tomasulo's algorithm for out-of-order instruction execution.

Modern supercomputers derive their power from architecture and parallelism rather than faster processors running at higher clock rates. The supercomputers of today consist of a very large number of processors and cores communicating through very fast and expensive *custom interconnects*. In mid 2012 the most powerful supercomputer was an IBM Sequoia-BlueGene/Q Linux-based system powered by Power BQC 16-core processors running at 1.6 GHz. The system, installed at Lawrence Livermore National Laboratory, which had a total of 1 572 864 cores and 1 572.864 TB of memory, achieved a sustainable speed of 16.32 PFlops, and consumed 7.89 MW of power. Later in 2012 a Cray XK7 system, Titan, installed at the Oak Ridge National Laboratory (ORNL) with 560 640 processors, including 261 632 Nvidia K20x accelerator cores, achieved a speed of 17.59 PFlops on the Linpack benchmark. In 2016, the most powerful supercomputer was Sunwai TaihuLight at National Supercomputer Center in Wixi, China, with 10 649 600 cores with a peak bandwidth of 125.436 PFlops. The system needed 15.371 MW of power. Its Linpack performance is 93.0146 PFlops, and it has 1 310.720 TB of memory.

As of June 2020, the Fugaku supercomputer at the RIKEN Center for Computational Science, Japan, powered by A64FX 2.2 GHz, an ARM architecture microprocessor designed by Fujitsu topped the list. It has 7 299 072 cores, a Linpack performance of 415.530 TFlop/sec, and needs 28.3345 MW. It is followed by the Summit at DOE/SC/Oak Ridge National Laboratory in the US using IBM Power System AC922, IBM POWER9 22C 3.07 GHz, with NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband. It has 2 414 592 cores, a Linpack performance of 148.600 TFlops/sec and needs 10 MW. Several of the most powerful systems listed in [478] are powered by Nvidia 2050 GPU. Some of the top 10 supercomputers use the Infiniband interconnect discussed in Chapter 6.

The next natural step to increase the computing power was possible due to advances in communication networks when low-latency and high-bandwidth Wide Area Networks (WANs) allowed individual systems, many of them multiprocessors, to be geographically separated. Large-scale distributed systems were first used for scientific and engineering applications and took advantage of the advancements in system software, programming models, tools, and algorithms developed for parallel processing.

A distributed system is a collection of autonomous and heterogeneous systems able to communicate effectively with each other. The question is: How could such a collection be organized to cooperate and compute efficiently? In spite of intensive research efforts spanning many years, an optimal organization of a large-scale system has eluded us.

The inability to conceive an optimal general-purpose system reflects the realization that a system cannot be looked upon in isolation, rather it should be analyzed in the context of the environment it is expected to operate in. The more complex and diverse this environment, the less likely it is to display an asymptotically optimal performance for all, or virtually all, applications. The organization and the system management may be optimal for a class of applications, yet have a sub-optimal performance for others.

Distributed systems have existed for several decades. For example, distributed file systems and network file systems have been used for user convenience and for improving reliability and functionality of file systems for many years, see Section 7.4. Modern operating systems allow a user to *mount* a remote file system and access it the same way a local file system is accessed, but with a performance penalty due to larger communication costs. *Remote Procedure Calls* (RPCs) allow a procedure on one system to invoke a procedure running in another address space, possibly on a remote system.

A *distributed system* is a collection of computers connected through a network and a distribution software called *middleware*, which enables computers to coordinate their activities and to share the resources of the system; the users perceive the system as a single, integrated computing facility. The middleware should support this set of desirable properties of the distributed system:

1. *Access transparency;* local and remote information objects should be accessed using identical operations.
2. *Location transparency;* information objects should be accessed without knowledge of their location.
3. *Concurrency transparency;* processes running concurrently should shared information objects without interference among them.
4. *Replication transparency;* multiple instances of information objects should be used to increase reliability without the knowledge of users or applications.
5. *Failure transparency;* the faults should be concealed.

6. *Migration transparency;* the information objects in the system should be moved without affecting the operation performed on them.
7. *Performance transparency;* the system should be reconfigured based on the load and quality of service requirements.
8. *Scaling transparency;* the system and the applications should scale without a change in the system structure and without affecting the applications.

A distributed system has several characteristics: Its components are autonomous, meaning scheduling and other resource management and security policies are implemented by each system. There are multiple points of control and multiple points of failure in a distributed system, and some sources may not be accessible at all times. Distributed systems can be scaled-up by adding more resources and can be designed to maintain availability even at low levels of hardware/software/network reliability. The availability is a system metric aiming to ensure an agreed-upon level of operational performance for an extended period of time.

3.12 Modularity. Soft modularity versus enforced modularity

Modularity is a basic concept in the design of man-made systems; a system is made out of components, or modules, with well-defined functions. Modularity supports the separation of concerns, encourages specialization, improves maintainability, reduces costs, and decreases the development time of a system.

Modularity has been used extensively since the Industrial Revolution to build every imaginable product, from weaving looms to steam engines, from watches to automobiles, from electronic devices to airplanes. Individual modules are often made of sub-assemblies. Modularity can reduce cost for the manufacturer and for the consumers. The same module may be used by a manufacturer in multiple products; to repair a defective product, a consumer only replaces the module causing the malfunction rather than the entire product. Modularity encourages specialization as individual modules can be developed by experts with deep understanding of a particular field. It also supports innovation since it allows a module to be replaced with a better one, without affecting the rest of the system.

It is no surprise that the hardware, as well as the software systems are composed of modules interacting with one another through well-defined interfaces. Software development for distributed systems is more challenging than for sequential systems, and these challenges are amplified by the scale of the system and the diversity of applications.

Modularity, layering, and hierarchy are some of the means to cope with the complexity of a distributed application software. Software modularity, the separation of a function into independent, interchangeable modules, requires well-defined interfaces specifying the elements provided and supplied to a module [389]. A modular software design is driven by several principles outlined in [139], namely:

a. *Information hiding;* the user of a module does not need to know anything about the internal mechanism of the module to make effective use of it.
b. *Invariant behavior;* the functional behavior of a module must be independent of the site or context from which it is invoked.

c. *Data generality;* the interface to a module must be capable of passing any data object an application may require.
d. *Secure arguments;* the interface to a module must not allow side-effects on arguments supplied to the interface.
e. *Recursive construction;* a program constructed from modules must be usable as a component in building larger programs or modules.
f. *System resource management;* resource management for program modules must be performed by the computer system and not by individual program modules.

Some of these principles are implicitly supported by the enforced modularity. A system should prevent modules to make private resource-allocation decisions and should support a global address space. The modularity concept is dissected, and its applications are reviewed in the next section.

The progress made in system design is notable not in the least due to a number of principles guiding the design of parallel and distributed systems. One of these principles is specialization; this means that a number of functions are identified and an adequate number of system components are configured to provide these functions. For example, data storage is an intrinsic function, and storage servers are a ubiquitous presence in most systems. This brings us to the modularity concept.

Modularity allows us to create a complex software system from a set of components built and tested independently. A requirement for modularity is to clearly define the interfaces between modules and enable the modules to work together. The steps involved in the transfer of the flow of control between the caller and the callee are: (i) the caller saves on the stack its state including registers, arguments, and return address; (ii) the callee loads the arguments from the stack, carries out the calculations, and then transfers control back to the caller; and (iii) the caller adjusts the stack, restores its registers, and continues its processing.

Soft modularity. We distinguish *soft modularity* from *enforced modularity*. The former implies dividing a program into modules that call each other and communicate using shared-memory or follow the procedure call convention.

Soft modularity hides the details of the implementation of a module and has many advantages: Once the interfaces of the modules are defined, the modules can be developed independently; a module can be replaced with a more elaborate, or with a more efficient one, as long as its interfaces with the other modules are not changed. The modules can be written using different programming languages and can be tested independently.

Soft modularity presents a number of challenges. It increases the difficulty of debugging, for example, a call to a module with an infinite loop it will never return. There could be naming conflicts and wrong context specifications. The caller and the callee are in the same address space and may misuse the stack, e.g., the callee may use registers that the caller has not saved on the stack, and so on.

Strongly-typed languages may enforce soft modularity by ensuring type safety at compile time or at run time, it may reject operations or function class that disregard the data types, or it may not allow class instances to have their class altered. Soft modularity may be affected by errors in the run-time system, errors in the compiler, or by the fact that different modules are written in different programming languages.

Enforced modularity. The ubiquitous client–server paradigm is based on enforced modularity; this means that the modules are forced to *interact only by sending and receiving messages.* This paradigm leads to a more robust design: The clients and the servers are independent modules and may fail separately.

Moreover, the servers are stateless, i.e., they do not have to maintain state information. A server may fail and then come back up without the clients being affected, or even noticing the failure of the server. The system is more robust because it does not allow errors to propagate. Enforced modularity makes an attack less likely because it is difficult for an intruder to guess the format of the messages or the sequence numbers of segments, when messages are transported by TCP.

Last, but not least, resources can be managed more efficiently. For example, a server typically consists of an ensemble of systems, a *front-end* system which dispatches the requests to multiple *back-end* systems which process the requests. Such an architecture exploits the elasticity of a computer cloud infrastructure: The larger the request rate, the larger is the number of back-end systems activated.

The client–server paradigm. This paradigm allows systems with different processor architecture, e.g., 32-bit or 64-bit, with different operating systems, e.g., multiple versions of operating systems, such as Linux, Mac OS, or Microsoft Windows, libraries and other system software, to cooperate. The client–server paradigm increases flexibility and choice; the same service could be available from multiple providers, a server may use services provided by other servers, a client may use multiple servers, and so on.

Heterogeneity of systems based on the client–server paradigm is less of a blessing, e.g., the problems it creates outweigh its appeal. Heterogeneity adds to the complexity of the interactions between a client and a server as it may require conversion from one data format to another, e.g., from little-endian to big-endian or vice-versa, or conversion to a canonical data representation. There is also uncertainty in terms of response time because some servers may be more performant than others or may have a lower workload.

A major difference between the basic models of grid and cloud computing is that the former does not impose any restrictions regarding heterogeneity of the computing platforms, whereas homogeneity used to be a basic tenet of computer clouds' infrastructure. Originally, a computer cloud was a collection of homogeneous systems, systems with the same architecture and running under the same or very similar system software. We have already seen in Chapter 2 that nowadays, computer clouds exhibit some heterogeneity. For example, AWS instances use now x86 64-bit and ARM 64-bit processors; accelerated computing instances have attached co-processors with NVIDIA cores and TPUs.

The clients and the servers communicate through a network that can be congested. Transferring large volumes of data through the network can be time-consuming; this is a major concern for data-intensive applications in cloud computing. Communication through the network further lengthens the response time. Security becomes a major concern as the traffic between a client and a server can be intercepted.

Remote Procedure Call (RPC). RPCs were introduced in the early 1970s by Bruce Nelson and used for the first time at PARC (Palo Alto Research Park), a place credited with many innovative ideas in distributed systems including the development of the Ethernet, the GUI interfaces, bitmap displays, and the Alto system. The Network File System (NFS) introduced in 1984 was based on Sun's RPC. Many programming languages support RPCs; for example, Java Remote Method Invocation (Java RMI) provides a functionality similar to the one of UNIX RPC methods, and XML-RPC uses XML to encode HTML-based calls.

RPC is often used for implementation of client–server systems' interactions. To use an RPC, a process may use special services *PORTMAP* or *RPCBIND* available at port 111 to register and for service lookup. RPC messages must be well-structured; they identify the RPC and are addressed to an RPC demon listening at an RPC port. *XDP* is a machine-independent representation standard for RPC. The RPC standard is described in RFC 1831.

RPCs reduce the *fate sharing* between caller and the callee. RPC's take longer than local calls due to communication delays. Several RPC semantics are used to overcome potential communication problems:

- **i.** *At least once:* a message is resent several times, and an answer is expected. The server may end up executing a request more than once, but an answer may never be received. This semantics is suitable for operation free of side effects.
- **ii.** *At most once:* a message is acted upon at most once. The sender sets up a timeout for receiving the response. When the timeout expires, an error code is delivered to the caller. This semantics requires the sender to keep a history of the time stamps of all messages because messages may arrive out-of-order. This semantics is suitable for operations which have side effects.
- **iii.** *Exactly once:* it implements the *at most once* semantics and requests an acknowledgment from the server.

Applications of the client–server paradigm. The large spectrum of applications attests to the role played by the client–server paradigm in the modern computing landscape. Examples of popular applications of the client–server paradigm are numerous and include: the World Wide Web, the Domain Name System (DNS), the X-windows, electronic mail, see Fig. 3.10(a), event services, see Fig. 3.10(b), and so on.

The World Wide Web illustrates the power of the client–server paradigm and its effects on society. The web enables users to access *resources* such as text, images, digital music, and any imaginable type of information previously stored in a digital format. A *web page* is created using a description language called HTML (Hypertext Description Language). Information in each web page is encoded and formatted according to some standard, e.g., GIF or JPEG for images, MPEG for videos, MP3 or MP4 for audio, and so on.

The web is based upon a "pull" paradigm; the resources are stored at the server's site and the client pulls them from the server. Some web pages are created "on the fly"; others are fetched from the disk. The client, called a *web browser,* and the server communicate using an application-level protocol called HTTP (HyperText Transfer Protocol) built on top of the TCP transport protocol.

The web server also called an *HTTP server* listens for connections from clients at port 80. The sequence of events when a client browser sends an HTTP request to a server to retrieve some information and the server constructs the page on the fly and then the browser sends another HTTP request for an image stored on the disk is discussed next. First, a TCP connection between the client and the server is established using a *three-way handshake.* The client provides an arbitrary initial sequence number in a special segment with the *SYN* control bit on; then, the server acknowledges the segment and adds its own arbitrarily chosen initial sequence number; finally, the client sends its own acknowledgment *ACK,* as well as the HTTP request, and the connection is established.

The time elapsed from the initial request until the server's acknowledgment reaches the client is called the RTT (Round-Trip Time). Once the TCP connection is established, the HTTP server takes its time to construct the page to respond to the first request; to satisfy the second request, the HTTP server must retrieve an image from the disk. The *response time* includes the RTT, the server residence time, and the data transmission time.

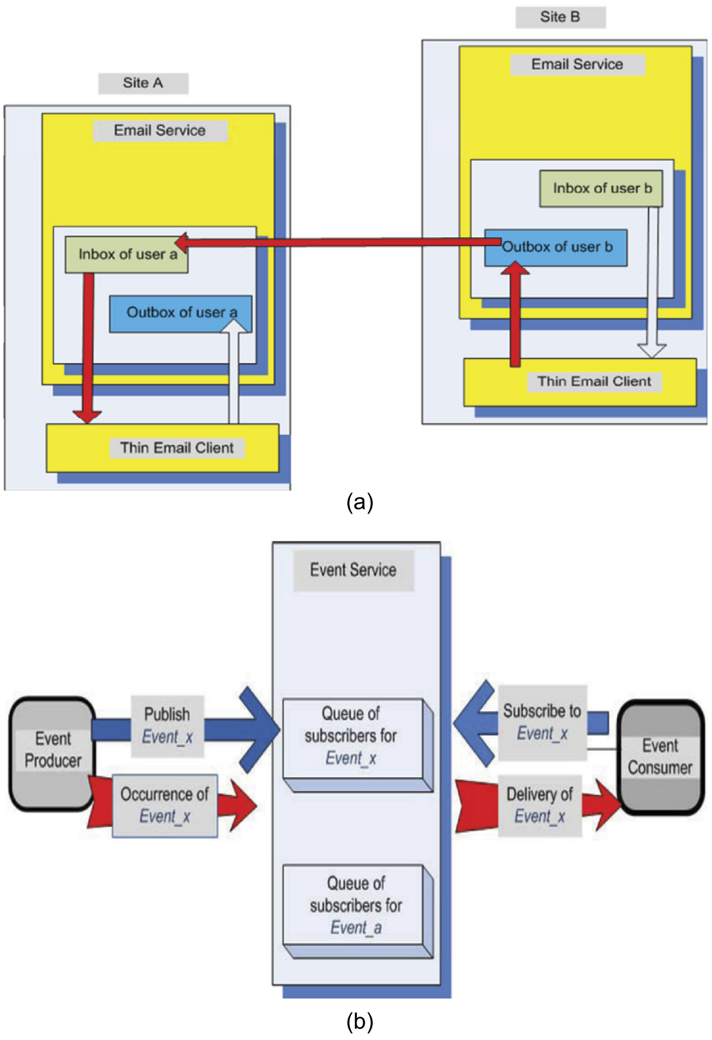


FIGURE 3.10

(a) Email service; sender and receiver communicate asynchronously using inboxes and outboxes. Mail daemons run at each site. (b) An event service supports coordination in a distributed system environment. The service is based on the *publish–subscribe* paradigm; an event producer publishes events, and an event consumer subscribes to events. The server maintains queues for each event and delivers notifications to clients when an event occurs.

The *response time*, defined as the time from the instance the first bit of the request is sent until the last bit of the response is received, consists of several components: the RTT, the *server residence time*, the time it takes the server to construct the response, and the data transmission time. RTT depends on the network latency, the time it takes a packet to cross the network from the sender to the receiver.

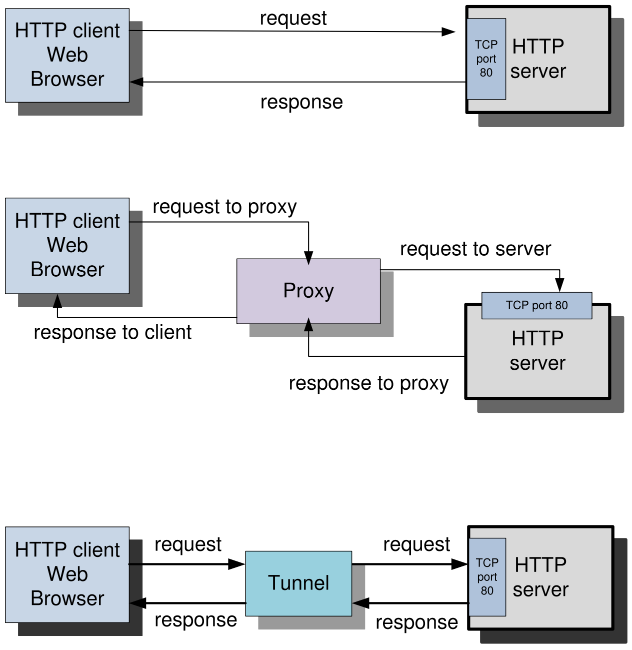


FIGURE 3.11

A client can communicate directly with the server, can communicate through a proxy, or may use tunneling to cross the network.

The data transmission time is determined by the network bandwidth. In turn, the server residence time depends on the server load.

Often, the client and the server do not communicate directly, but through a proxy server as shown in Fig. 3.11. Proxy servers could provide multiple functions; for example, they may filter client requests and decide whether or not to forward the request based on some filtering rules. A proxy server may redirect a request to a server in close proximity of the client or to a less-loaded server. A proxy can also act as a cache and provide a local copy of a resource, rather than forward the request to the server.

Another type of client–server communication is *HTTP-tunneling,* used most often as a means of communication from network locations with restricted connectivity. Tunneling means encapsulation of a network protocol; in our case HTTP acts as a wrapper for the communication channel between the client and the server, see Fig. 3.11.

3.13 Layering and hierarchy

Layering and hierarchy have been present in social systems since ancient times. For example, the Spartan Constitution, called Politeia, describes a Dorian society based on a rigidly layered social system and a strong military. Nowadays, in a modern society, we are surrounded by organizations structured hierarchically. We have to recognize that layering and hierarchical organization have their own problems, could negatively affect the society, impose a rigid structure and affect social interactions, increase the overhead of activities, and prevent the system from acting promptly when such actions are necessary.

Layering demands modularity because each layer fulfills a well-defined function. The communication patterns in the case of layering are more restrictive; a layer is expected to communicate only with the adjacent layers. This restriction, the limitation of communication patterns, clearly reduces the complexity of the system and makes it easier to understand its behavior.

Modularity, layering, and hierarchy are critical for computer and communication systems. Large programs have been split into modules, each with a well-defined functionality since the early days of computing. Modules with related functionalities have then been grouped together into numerical, graphics, statistical, and many other libraries. Layering helps us dealing with complicated problems when we have to separate concerns that prevent us from making optimal design decisions. To do so, we define layers that address each concern and design clear interfaces between the layers.

Probably the best example is layering of communication protocols. Early on, the need of accommodating a variety of physical communication channels that carry electromagnetic, optical, or acoustic signals, thus, the need for a *physical* layer, was recognized. The next concern is how to transport bits, not signals, between two systems directly connected to one another by a communication channel, i.e., the need for a *data link* layer.

Communication requires networks with multiple intermediate nodes. When bits have to traverse a chain of intermediate nodes from a source to the destination, the concern is how to forward the bits from one intermediate node to the next, so the *network* layer was introduced. Then, it was recognized that the source and the recipient of information are in fact outside the network and only want the data to reach the destination unaltered. Therefore, the *transport* layer was deemed necessary. Finally, the data sent and received has a meaning only in the context of an application, thus, the need for the *application* layer.

Strictly enforced layering can prevent optimizations. For example, cross-layer communication in networking was proposed to enable wireless applications to take advantage of information available at the Media Access Control (MAC) sub-layer of the data link layer. This example shows that layering gives us insight into where to place the basic mechanisms for error control, flow control, and congestion control of the network protocol stack.

An interesting question is whether a layered cloud architecture could be designed that has practical implications for the future development of computing clouds. One could argue that it may be too early for such an endeavor, that we need time to fully understand how to better organize a cloud infrastructure, and that we need to gather data to support the advantages of one approach over another. On the other hand, there are systems where it is difficult to envision a layered organization because of the complexity of the interaction between the individual modules. Consider for example an operating system that has a set of well-defined functional components:

1. The processor management subsystem, responsible for processor virtualization, scheduling, interrupt handling, and execution of privileged operations and system calls.
2. The virtual memory management subsystem, responsible for translating virtual addresses to physical addresses.
3. The multi-level memory management subsystem, responsible for transferring storage blocks between different memory levels, most commonly between primary and secondary storage.

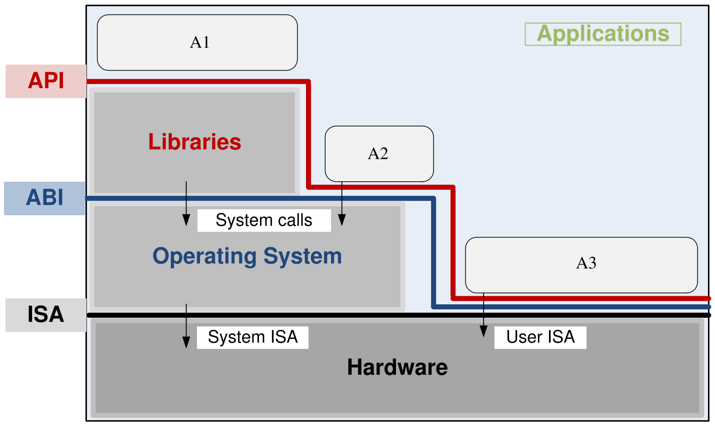


FIGURE 3.12

Layering and interfaces between layers in a computer system. The software components including applications, libraries, and operating systems interact with the hardware via several interfaces: the Application Program Interface (API), the Application Binary Interface (ABI), and the Instruction Set Architecture (ISA). An application uses library functions (A1), makes system calls (A2), and executes machine instructions (A3).

4. The I/O subsystem, responsible for transferring data between the primary memory and the I/O devices.
5. The networking subsystem responsible for network communication.

The processor management interacts with all other subsystems, and there are also multiple interactions between the other subsystems; therefore, it seems unlikely that a layered organization would be feasible in this case.

Layers and interfaces between layers. A common approach to managing system complexity is to identify a set of *layers* with well-defined *interfaces* among them. The interfaces separate different levels of abstraction. Layering minimizes the interactions among the subsystems and simplifies the description of the subsystems. Each subsystem is abstracted through its interfaces with the other subsystems, thus, we are able to design, implement, and modify the individual subsystems independently.

The ISA (Instruction Set Architecture) defines the set of instructions of a processor; for example, the Intel architecture is represented by the *x86*-32 and *x86*-64 instruction sets for systems supporting 32-bit addressing and 64-bit addressing, respectively. The hardware supports two execution modes, a *privileged*, or *kernel* mode, and a *user* mode.

The instruction set consists of two sets of instructions, *privileged* instructions that can only be executed in kernel mode and the *non-privileged* instructions that can be executed in user mode. There are also *sensitive instructions* that can be executed in kernel and in user mode, but behave differently, see Section 5.4.

Computer systems are fairly complex and their operation is best understood when we consider a model similar with the one in Fig. 3.12, which shows the interfaces between the software components and the hardware [450]. The hardware consists of one or more multicore processors, a system interconnect (e.g., one or more busses), a memory translation unit, the main memory, and I/O devices, including one or more networking interfaces.

Applications written mostly in High-Level Languages (HLL) often call library modules and are compiled into *object code*. Privileged operations, such as I/O requests, cannot be executed in user mode; instead, application and library modules issue *system calls,* and the operating system determines if the privileged operations required by the application do not violate system security or integrity and, if so, executes them on behalf of the user. Binaries resulting from translation of HLL programs target a specific hardware architecture.

The first interface, at the boundary of hardware and software, is the *Instruction Set Architecture* (ISA). The next interface is the *Application Binary Interface* (ABI), which allows the ensemble consisting of the application and the library modules to access the hardware. ABI does not include privileged system instructions; rather, it invokes system calls.

Finally, the *Application Program Interface* (API) defines the set of instructions the hardware was designed to execute and gives the application access to the ISA. API includes HLL library calls, which often invoke system calls. Recall that a *process* is the abstraction for the code of an application at execution time; a *thread* is a light-weight process. ABI is the projection of the computer system seen by the process, and API is the projection of the same system from the perspective of the HLL program.

Binaries created by a compiler for a specific ISA and a specific operating systems are not portable. Such code cannot run on a computer with a different ISA, or on the computer with the same ISA but a different OS. But it is possible to compile an HLL program for a VM environment. In this case, portable code is produced and distributed and then converted by binary translators to the ISA of the host system. A *dynamic binary translation* converts blocks of guest instructions from the portable code to the host instructions and leads to a significant performance improvement because such blocks are cached and reused.

3.14 Peer-to-peer systems

The distributed systems discussed in this chapter allow access to resources in a tightly controlled environment. System administrators enforce security rules and control the allocation of physical, rather than virtual resources. In all models of network-centric computing prior to utility computing, a user maintained direct control of the software and the data residing on remote systems.

This user-centric model, in place since the early 1960s, was challenged in the 1990s by the peer-to-peer (P2P) model. P2P systems share some ideas with computer clouds. The new distributed computing model promoted the idea of low-cost access to storage and CPU cycles provided by participant systems. In this case, the resources are located in different administrative domains. The P2P systems are self-organizing and decentralized, while the servers in a cloud are in a single administrative domain and have a central management.

P2P systems exploit the network infrastructure to provide access to distributed computing resources. Decentralized applications developed in the 1980s, such as SMTP (Simple Mail Transfer Protocol), a protocol for Email distribution, and NNTP (Network News Transfer Protocol), an application protocol for dissemination of news articles, are early examples of P2P systems. Systems developed in the late 1990s, such as the music-sharing system Napster, gave participants access to storage distributed over the network. The first volunteer-based scientific computing, SETI@home, used free cycles of participating systems to carry out compute-intensive tasks.

P2P model represents a significant departure from the client–server model, the cornerstone of distributed applications for several decades. P2P systems have several desirable properties [420]:

- **i.** Require a minimally dedicated infrastructure; participating systems provide resources.
- **ii.** Are highly decentralized.
- **iii.** Are scalable; individual nodes are not required to be aware of the global state.
- **iv.** Are resilient to faults and attacks because few of their elements are critical for the delivery of service and the abundance of resources can support a high degree of replication.
- **v.** Individual nodes do not require excessive network bandwidth.
- **vi.** The dynamic unstructured system architecture shields P2P systems from censorship.

The undesirable properties of peer-to-peer systems are also notable. Decentralization raises the question if P2P systems can be managed effectively and provide the security required by various applications. The fact that they are shielded from censorship makes them a fertile ground for illegal activities, including the distribution of copyrighted content.

In spite of its problems, the new paradigm was embraced by applications other than file sharing. Since 1999, new P2P applications, such as the ubiquitous Skype, a voice over IP telephony service,[5] data-streaming applications such as Cool Streaming [537] and BBC's online video service, content distribution networks such as CoDeeN [503], and volunteer computing applications based on the BOINC (Berkeley Open Infrastructure for Networking Computing) platform [26], have proved their appeal to users.

Skype reported in 2008 that 276 million registered users have used more than 100 billion minutes for voice and video calls. The site www.boinc.berkeley.edu reports that at the end of June 2012, volunteer computing involved more than 275 000 individuals and more than 430 000 computers providing a monthly average of almost 6.3×10^9 MFlops. It is also reported that the P2P traffic accounts for a very large fraction of the Internet traffic, with estimates ranging from 40% to more than 70%.

Many groups from industry and academia have rushed to develop and test new ideas taking advantage of the fact that P2P applications do not require a dedicated infrastructure. Applications such as Chord [457] and Credence [502] address issues critical for the effective operation of decentralized systems. Chord is a distributed lookup protocol to identify the node where a particular data item is stored. The routing tables are distributed and, while other algorithms for locating an object require the nodes to be aware of most of the nodes of the network, Chord maps a key related to an object to a node of the network using routing information about a few nodes only.

Credence is an object reputation and ranking scheme for large-scale P2P file sharing systems. Reputation is of paramount importance for systems that often include many unreliable and malicious nodes. In the decentralized algorithm used by *Credence,* each client uses local information to evaluate the reputation of other nodes and shares its own assessment with its neighbors. The credibility of a node depends only on the votes it casts.

[5] Skype allows close to 700 million registered users from many countries around the globe to communicate using a proprietary voice-over-IP protocol. The system developed in 2003 by Niklas Zennström and Julius Friis was acquired by Microsoft in 2011 and nowadays is a hybrid P2P and client–server system.

Each node computes the reputation of another node based solely on the degree of matching with its own votes and relies on like-minded peers. Overcite [461] is a P2P application to aggregate documents based on a three-tier design. The web front-ends accept queries and display the results, while servers crawl through the web to generate indexes and to perform keyword searches; the web back-ends store documents, metadata, and coordination state on the participating systems.

The rapid acceptance of the new paradigm has triggered the development of a new communication protocol allowing hosts at the network periphery to cope with the limited network bandwidth available to them. BitTorrent is a peer-to-peer file sharing protocol enabling a node to download/upload large files from/to several hosts simultaneously.

The P2P systems differ in their architecture. Some do not have any centralized infrastructure, while others have a dedicated controller, but this controller is not involved in resource-intensive operations. For example, Skype has a central site to maintain the user accounts; the users sign in and pay for specific activities on this site. The controller for a BOINC platform maintains membership and is involved in task distribution to participating systems. The nodes with abundant resources in systems without any centralized infrastructure often act as *supernodes* and maintain information useful to increasing the system efficiency, e.g., indexes of the available content.

Regardless of the architecture, P2P systems are built around an *overlay network*, a virtual network superimposed over the real network. Each node maintains a table of *overlay links* connecting it with other nodes of this virtual network, each node being identified by its IP addresses. Two types of overlay networks, *unstructured* and *structured*, are used by P2P systems. Random walks starting from a few bootstrap nodes are usually used by systems desiring to join an unstructured overlay.

Each node of a structured overlay has a unique key that determines its position in the structure; the keys are selected to guarantee a uniform distribution in a very large name space. Structured overlay networks use *key-based routing* (KBR); given a starting node v_0 and a key k, the function $KBR(v_0, k)$ returns the path in the graph from v_0 to the vertex with key k. Epidemic algorithms are often used by unstructured overlays to disseminate the network topology.

3.15 Large-scale systems

The developments in computer architecture, storage technology, networking, and software during the last several decades of the 20th century, coupled with the need to access and process information, led to several large-scale distributed system developments:

The web and the semantic web expected to support composition of services (not necessarily computational services) available on the web. The web is dominated by unstructured or semi-structured data, while the semantic web advocates inclusion of semantic content in web pages.
The Grid, initiated in the early 1990s by National Laboratories and universities primarily for applications in science and engineering.

The need to share data from high energy physics experiments motivated Sir Tim Berners-Lee, who worked at CERN at Geneva in the late 1980s, to put together the two major components of the World Wide Web: HTML (Hypertext Markup Language) for data description and HTTP (Hypertext Transfer Protocol) for data transfer. The web opened a new era in data sharing and ultimately led to the concept of network-centric content.

The *Semantic Web* is an effort to enable lay people to find, share, and combine information available on the web more easily. The name was coined by Berners-Lee to describe "a web of data that can be processed directly and indirectly by machines." It is a framework for data sharing among applications based on the Resource Description Framework (RDF). In this vision, information can be readily interpreted by machines, so machines can perform the tedious work involved in finding, combining, and acting upon information on the web.

The semantic web is "largely unrealized" according to Berners-Lee. Several technologies are necessary to provide a formal description of concepts, terms, and relationships within a given knowledge domain; they include the Resource Description Framework (RDF), a variety of data interchange formats, and notations such as RDF Schema (RDFS) and the Web Ontology Language (OWL).

Gradually, the need to make computing more affordable and to liberate the users from the concerns regarding system and software maintenance reinforced the idea of concentrating computing resources in data centers. Initially, these centers were specialized, each running a limited palette of software systems, as well as applications developed by the users of these systems. In the early 1980s major research organizations, such as the National Laboratories and large companies, had powerful computing centers supporting large user populations scattered throughout wide geographic areas. Then the idea to link such centers in an infrastructure resembling the power grid was born; the model known as network-centric computing was taking shape.

A *computing grid* is a distributed system consisting of a large number of loosely coupled, heterogeneous, and geographically dispersed systems in various administrative domains. The term *computing grid* is a metaphor for accessing computer power with similar ease as we access power provided by the electric grid. Software libraries known as *middleware* have been furiously developed since the early 1990s to facilitate access to grid services.

The vision of the grid movement was to give a user the illusion of a very large virtual supercomputer. The autonomy of the individual systems and the fact that these systems were connected by wide-area networks with latency higher than the latency of the interconnection network of a supercomputer posed serious challenges to this vision. Nevertheless, several "Grand Challenge" problems, such as protein folding, financial modeling, earthquake simulation, and climate/weather modeling, run successfully on specialized grids. The Enabling Grids for Escience project is arguably the largest computing grid; along with the LHC Computing Grid (LCG), the Escience project aims to support the experiments using the Large Hadron Collider (LHC) at CERN which generates several gigabytes of data per second, or 10 PB (petabytes) per year.

In retrospect, two basic assumptions about the infrastructure prevented the grid movement from having the impact its supporters were hoping for. The first is the heterogeneity of the individual systems interconnected by the grid. The second is that systems in different administrative domain are expected to cooperate seamlessly. Indeed, the heterogeneity of the hardware and of the system software poses significant challenges for application development and for application mobility.

At the same time, critical areas of system management including scheduling, optimization of resource allocation, load balancing, and fault-tolerance are extremely difficult in a heterogeneous system. The fact that resources are in different administrative domains further complicates many, already difficult, problems related to security and resource management. While very popular in the science and the engineering communities, the grid movement did not address the major concerns of enterprise computing community and did not make a noticeable impact on the IT industry.

Cloud computing is largely viewed as the next big step in the development and deployment of an increasing number of distributed applications. The companies promoting cloud computing seem to have learned the most important lessons from the grid movement. Computer clouds are typically homogeneous. An entire cloud shares the same security, resource management, cost, and other policies, and last, but not least, it targets enterprise computing. These are some of the reasons why several agencies of the US government, including Health and Human Services, the Center for Disease Control (CDC), NASA, the Navy's Next Generation Enterprise Network (NGEN), and Defense Information Systems Agency (DISA), have launched cloud computing initiatives and conduct actual system developments intended to improve the efficiency and effectiveness of their information processing needs.

3.16 Composability bounds and scalability (R)

Nature creates complex systems from simple components. For example, a vast variety of proteins are linear chains assembled from 20 amino acids, the building blocks of proteins found in human body. These amino acids are naturally incorporated into polypeptides and are encoded by the genetic code. Imitating nature, manmade systems are assembled from sub-assemblies; in turn, a sub-assembly is made from several modules, each module could consist of sub-modules, and so on. Composability has natural bounds imposed by the laws of physics as we have seen when discussing heat dissipation of solid-state devices. As the number of components increases, the complexity of a system also increases.

The limits of composability can be reached when the physical size of individual components changes. A recent paper with the suggestive title "When every atom counts" [350] shows that even the most modern solid-state fabrication facilities cannot produce chips with consistent properties. The percentage of defective or substandard chips has been constantly increasing as components have become smaller and smaller.

The lack of consistency in the manufacturing process of solid-state devices is attributed to the increasingly smaller size of the physical components of a chip. This problem is identified by the International Technology Roadmap for Semiconductors as "a red brick wall," a problem without a clear solution, a wall that could prevent further progress. Chip consistency is no longer feasible because the transistors and the "wires" on a chip are so small that random differences in the placement of an atom can have a devastating effect, e.g., it can increase the power consumption by an order of magnitude and slowdown the chip by as much as 30%.

As features become smaller and smaller, the range of the *threshold voltage,* the voltage needed to turn a transistor on and off, has been widening; since now many transistors have this threshold voltage at or near zero, they cannot operate as switches. While the range for 28-nm technology was approximately between $+0.01$ and $+0.4$ V, the range for 20-nm technology is between -0.05 and $+0.45$ V, and the range becomes even wider, from -0.18 to $+0.55$ for 14-nm technology.

There are physical bounds for the composition of analog systems: Noise accumulation, heat dissipation, cross-talk, the interference of signals on multiple communication channels, and several other factors limit the number of components of an analog system. Digital systems have more distant bounds, but composability is still limited by physical laws.

There are virtually no bounds on the composition of digital computing and communication systems controlled by software. The Internet is a network of networks and a prime example of composability with distant bounds. Computer clouds are another example; a cloud is composed of a very large number of servers and interconnects, each server is made up of multiple processors, and each processor has multiple cores. Software is the ingredient that pushes the composability bounds and liberates computer and communication system from the limits imposed by physical laws.

In the physical world the laws valid at one scale break down at a different scale, e.g., the laws of classical mechanics are replaced at the atomic and subatomic scales by quantum mechanics. Thus, we should not be surprised that scale really matters in the design of computing and communication systems. Indeed, architectures, algorithms, and policies that work well for systems with a small number of components very seldom scale up.

For example, many computer clusters have a front-end that acts as the nerve center of the system, manages communication with the outside world, monitors the entire system, and supports system administration and software maintenance. A computer cloud has multiple such nerve centers, and new algorithms to support collaboration among these centers must be developed. Scheduling algorithms that work well within the confines of a single system cannot be extended to collections of autonomous systems when each system manages local resources; in this case, as in the previous example, entities must collaborate with one another, and this requires communication and consensus.

Another manifestation of this phenomenon is in the vulnerabilities of large-scale distributed systems. The implementation of Google's Bigtable revealed that many distributed protocols designed to protect against network partitions and fail–stop are unable to cope with failures due to scale [94]. Memory and network corruption, extended and asymmetric network partitions, systems that fail to respond, and large clock skews occur with increasing frequency in a large-scale system, and they interact with one another in a manner that greatly affects the overall system availability.

Scaling has other dimensions than just the number of components, and space plays an important role. The communication latency is small when the component systems are clustered together within a small area, and this allows us to implement efficient algorithms for global decision making, e.g., consensus algorithms. When, for the reasons discussed in Section 1.4, the data centers of a cloud provider are distributed over a large geographic area, transactional database systems are of little use for most online transaction-oriented systems, and a new type of data store has to be introduced in the computational ecosystem.

Societal scaling means that a service is used by a very large segment of the population and/or is a critical element of the infrastructure. There is no better example to illustrate how societal scaling affects the system complexity than communication supported by the Internet. The infrastructure supporting the service must be highly available. A consequence of redundancy and of the measures to maintain consistency is increased system complexity.

At the same time, the popularity of the service demands simple and intuitive means to access the infrastructure. Again, the system complexity increases due to the need to hide the intricate mechanisms from a lay person with little understanding of the technology. The vulnerability of wireless systems has increased due to the desire to design wireless devices that: (a) operate efficiently in terms of power consumption; (b) present the user with a simple interface and few choices; and (c) satisfy a host of other popular functions. This is happening at a time when not many smartphone and tablet users understand the security risks of wireless communication.

3.17 Distributed computing fallacies and the CAP theorem

Distributed processing became a reality after Berkeley sockets were released in 1983 as a programming interface in 4.2 BSD Unix operating system. A *socket* is an abstract representation of the local endpoint of a network communication path. A socket is like a file descriptor in the Unix philosophy; it provides a common interface for input and output streams of data. Sockets became the standard interface for applications running in the Internet. All operating systems implement a version of Berkeley socket interface.

The group at UC Berkeley, credited with the developments of sockets, founded Sun Microsystems in 1982. Sun is credited with a number of remarkable contributions to distributed computing, including the Java programming language, the Solaris operating system, and the Network File System (NFS), as well as SPARC microprocessors. At Sun, Bill Joy and Dave Lyon formulated a list of flawed assumptions about distributed computing. James Gosling, best known as the founder and lead designer of the Java programming language, codified four of these assumptions as "The Fallacies of Networked Computing."

According to the Merriam–Webster dictionary, a fallacy is a false or mistaken idea or an often plausible argument using false or invalid inferences. In the early days of distributed processing, and even later, many developers of distributed applications made a number of false assumptions. The list of fallacies was then extended: Peter Deutsch added another three, and James Gosling contributed the most recent one, see http://java.sys-con.com/read/38665.htm. Ignoring the problems caused by each one of these fallacies has severe consequences:

1. *The network is reliable.* This assumption causes applications to stall or infinitely wait for an answer; memory and other resources may fail to retry stalled operations or require a manual restart in case of network outage.
2. *The latency is zero and no packets are lost.* This causes application- and transport-layer developers to assume unbounded traffic, greatly increasing the number of dropped packets and wasting bandwidth.
3. *The bandwidth is infinite.* Assuming infinite bandwidth and repeated packet retransmissions can cause bottlenecks.
4. *The network is secure.* Ignoring malicious users is a capital sin.
5. *The topology does not change.* Topology changes affect both bandwidth and latency.
6. *There is one administrator.* There may be conflicting policies preventing packets to reach their destination, and these may limit user access to some resources.
7. *The transport cost is zero.* The costs of building and maintaining networks cannot be ignored.
8. *The network is homogeneous.* The Internet is a network of networks with various different latencies and bandwidths.

Grid computing movement initiated in late 1980s captured the interest of the distributed research community for more than two decades, in spite of ignoring the consequences of the eight fallacies. This movement aimed to create a "super virtual computer" composed of many loosely coupled computers acting together to perform large computational tasks. Grid computers tended to be heterogeneous with systems in different administrative domains; moreover, the computers were mostly connected by networks with limited communication bandwidth and high latency. The Grid movement faded away for these reasons and because it did not attract applications other than those in computational science and engineering, it was eventually replaced by cloud computing.

Networking has dramatically changed since 1991. The effects of some of these fallacies have been amplified; in fact, malicious users are now able to threaten the critical infrastructure of the society. The consequences of other fallacies have been attenuated; the communication technology has evolved, bandwidth of terabytes per second are not uncommon, error rates are low, and today's routers have substantial resources to support various types of traffic. Application developers have now access to cloud computing services that mostly shield them from the effects of the eight fallacies.

The CAP theorem due to Eric Brewer from UC Berkeley states that it is impossible for a distributed data store to simultaneously provide more than two out of the three guarantees: (i) consistency—every read receives either the most recent write or an error; (ii) availability—every request receives a (non-error) response, without the guarantee that it contains the most recent write; and (iii) partition tolerance—the system continues to operate despite network errors when an arbitrary number of messages are dropped or delayed [198].

Tradeoffs are common, for example, consistency can be traded for availability. For example, when running of a cluster, a subset of nodes being consistent is sufficient for practical consistency. Write Quorum requires that the number of nodes writing successfully should be more than half the number of nodes involved in replication.

In 2010, Daniel Abadi from Yale University argued that "Ignoring the consistency/latency trade-off of replicated systems is a major oversight [in CAP], as it is present at all times during system operation, whereas CAP is only relevant in the arguably rare case of a network partition" [2]. The PACELC theorem due to Abadi is an extension of CAP stating that "in case of network partitioning (P) in a distributed computer system, one has to choose between availability (A) and consistency (C) (as per the CAP theorem), but else (E), even when the system is running normally in the absence of partitions, one has to choose between latency (L) and consistency (C)." Section 7.9 includes an in-depth discussion of PACELC implications for cloud storage systems.

Note that the blockchain technology gives the impression that CAP is invalid since it sacrifices consistency for availability and partition tolerance, but availability and partition tolerance is achieved through validation among the nodes over time.

3.18 Blockchain technology and applications

A blockchain is a decentralized, distributed, digital ledger consisting of records called *blocks* recording transactions so that no block can be altered retroactively without the alteration of all subsequent blocks; it was invented in 2008 by an individual of unknown identity. A blockchain is managed by a P2P network collectively adhering to a protocol for inter-node communication and for new block validation.

A block contains batches of valid transactions hashed and encoded into a Merkle tree, see also Section 7.13. Every node of the system has a copy of the blockchain, and data quality is maintained by massive database replication and computational trust. Fig. 3.13 illustrates the flow of blockchain information. Cryptocurrencies such as Bitcoin made blockchain technology ubiquitous.

Bitcoin mining is the process by which new Bitcoins are entered into circulation and it is also a critical component of the maintenance and development of the blockchain ledger. Powerful ASICs (Application Specific Integrated Circuits) are used for bitcoin mining; for example, WhatsMiner M30S++ delivers 112 TH/s (Tera Hashing operations per second). Aside from the short-term Bitcoin payoff, being a coin miner gives "voting" power when changes in the Bitcoin network protocol are proposed.

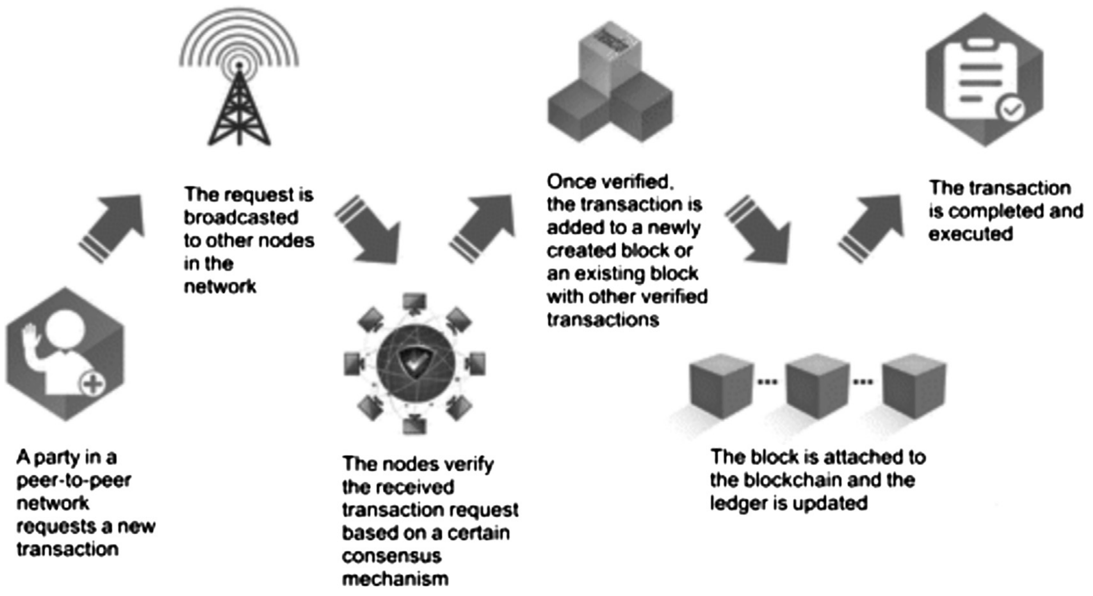

A party in a peer-to-peer network requests a new transaction
The request is broadcasted to other nodes in the network
The nodes verify the received transaction request based on a certain consensus mechanism
Once verified, the transaction is added to a newly created block or an existing block with other verified transactions
The block is attached to the blockchain and the ledger is updated
The transaction is completed and executed

FIGURE 3.13

Blockchain flow of information [318].

While blockchain is widely used by cryptocurrencies, there are many other potential applications of this technology with immense economic and social implications, [318]. In recent years, we have seen an explosion of interest in blockchain that "could someday underlie everything from how we vote to who we connect with online to what we buy" (Wall Street Journal 2018, p. B4). For example, GSA (General Service Administration) with contracts of some $55 billions per year is working with a technology company to develop a new procurement blockchain for its vendors network. Blockchain could be used in detecting counterfeits by associating unique identifiers to individual items and storing records associated with transactions involving the item that cannot be forged or altered. A non-fungible token (NFT) is a unit of data stored on a blockchain that certifies a digital asset to be unique. NFTs are now widely used to sign digital art and other collectibles.

Blockchain exhibits fundamental properties unreachable by other technologies. *Decentralized consensus* is a hard problem; achieving consensus in a decentralized network requires careful design of the algorithms. Blockchain enables data integrity because the complete copy of identical information can be held by anyone who has access to it and wants to keep it. *Machine-based automation* gives blockchain the freedom to bypass human actors' unpredictability and inability to process massive amounts of information.

The rise of blockchain transactions has led to increased environmental criticism. The proof-of-work blockchain process is computationally intensive, requiring high energy consumption contributing to global warming. According to the University of Cambridge's Bitcoin electricity consumption index, Bitcoin miners are expected to consume roughly 130 Terawatt-hours of energy (TWh), roughly 0.6% of global electricity consumption in 2022 [125].

3.19 History notes and further readings

Ex nihilo nihil, the philosophical dictum first appearing in Aristotle's Physics[6] translated as "nothing comes from nothing," applies to every human endeavor and means to search for the origins of everything to grasp the intellectual challenges faced along the way by the great minds of the past. This motivates us to take a brief excursion into the history of computing.

Two theoretical developments in 1930s were critical for the development of modern computers. The first was the publication of Alan Turing's 1936 paper [481]. The paper provided a definition of a universal computer, called a Turing Machine, which executes a program stored on tape; the paper also proved that there were problems, such as the halting problem, that could not be solved by any sequential process. The second major development was the publication in 1937 of Claude Shannon's master's thesis at MIT "A Symbolic Analysis of Relay and Switching Circuits" in which he showed that any Boolean logic expression can be implemented using logic gates. This should remind us how much the digital era owes to George Boole, a humble professor at University College Cork, in Ireland!

The first Turing complete[7] computing device was Z3, an electro-mechanical device built by Konrad Zuse in Germany in May 1941; Z3 used a binary floating-point representation of numbers and was program-controlled by a film-stock. The first programmable electronic computer ENIAC, built at the Moore School of Electrical Engineering at the University of Pennsylvania by a team led by John Prosper Eckart and John Mauchly, became operational in July 1946 [341]; ENIAC, unlike Z3, used a decimal number system and was program-controlled by patch cables and switches.

John von Neumann, the famous mathematician and theoretical physicist, contributed fundamental ideas for modern computers [79,497,498]. He was one of the most brilliant minds of the 20th century, with an uncanny ability to map fuzzy ideas and garbled thoughts to crystal clear and scientifically sound concepts. John von Neumann drew the insight for the stored-program computer from Alan Turing's work[8] and from his visit to the University of Pennsylvania; he thought that ENIAC was an engineering marvel, but was less impressed with the awkward manner to "program" it by manually connecting cables and setting switches. He introduced the so-called "von Neumann architecture" in a report published in the 1940s; to this day, he is faulted by some because he failed to mention in this report the sources of his insight.

John von Neumann led the development at the Institute of Advanced Studies at Princeton of MANIAC, an acronym for "mathematical and numerical integrator and computer." MANIAC was closer to the modern computers than any of its predecessors; it was used for sophisticated calculations required by the development of the hydrogen bomb nicknamed "Ivy Mike" secretly detonated on November 1, 1952, over an island that no longer exists in the South Pacific. In a recent book [157], the historian of science George Dyson writes: "The history of digital computing can be divided into an Old Testament whose prophets, led by Leibnitz, supplied the logic, and a New Testament whose prophets led by von Neumann built the machines. Alan Turing arrived between them."

[6] This dictum comes from ancient Greek cosmology and states that everything has its origin in something.

[7] A Turing complete computer is equivalent to a universal Turing machine except for memory limitations.

[8] Alan Turing came to the Institute of Advanced Studies at Princeton in 1936 and got his Ph.D. there in 1938; von Neumann offered him a position at the Institute, but, as the dark clouds signaling the approaching war were gathering over Europe, Turing decided to go back to England.

In 1951, Sir Maurice Vincent Wilkes developed the concept of microprogramming first implemented on the EDSAC 2 computer. Wilkes is also credited with the idea of symbolic labels, macros and subroutine libraries, and caching.

Third-generation computers were built during the period 1964–1971; they made extensive use of integrated circuits (ICs) and ran under the control of an operating systems. MULTIX (Multiplexed Information and Computing Service) was an early time-sharing operating system developed in 1983 by MIT, GE, and Bell Labs for the GE 645 mainframe. MULTIX had a lasting impact on the design and implementation of computer systems, had numerous novel features, and implemented a fair number of interesting concepts such as: a hierarchical file system, access control lists for file information sharing, dynamic linking, and online reconfiguration [115,116].

In his address "A Career in Computer System Architecture", MIT Professor Jack Dennis wrote: "In 1960 Professor John McCarthy, now at Stanford University and known for his contributions to artificial intelligence, led the *Long Range Computer Study Group* (LRCSG) which proposed objectives for MIT's future computer systems. I had the privilege of participating in the work of the LRCSG, which led to Project MAC and the Multics computer and operating system, under the organizational leadership of Prof. Robert Fano and the technical guidance of Prof. Fernando Corbató. At this time, Prof. Fano had a vision of the Computer Utility—the concept of the computer system as a repository for the knowledge of a community—data and procedures in a form that could be readily shared—a repository that could be built upon to create ever more powerful procedures, services, and active knowledge from those already in place. Prof. Corbató's goal was to provide the kind of central computer installation and operating system that could make this vision a reality. With funding from DARPA, the Defense Advanced Research Projects Agency, the result was Multics...in the 1970s I found it easy to get government funding. The agencies were willing to fund pretty wild ideas, and I was supported to do research on *data flow* architecture, first by NSF and later by the DOE" (http://csg.csail.mit.edu/Users/dennis/essay.htm).

The development of the UNIX system was a consequence of the withdrawal of Bell Labs from the MULTIX project in 1968. UNIX was developed in 1969 for a DEC PDP minicomputer by a group led by Kenneth Thompson and Dennis Ritchie [417]. According to [416], "the most important job of *UNIX* is to provide a file-system;" the same reference discusses another concept introduced by the system: "For most users, communication with UNIX is carried on with the aid of a program called the Shell. The Shell is a command line interpreter: it reads lines typed by the user and interprets them as requests to execute other programs."

The first microprocessor, Intel 4004, was announced in 1971; it performed binary-coded decimal (BCD) arithmetic using 4-bit words. Intel 4004 was followed in by Intel 8080, the first 8-bit microprocessor, and by its competitor, Motorola 6800, released in 1974. The first 16-bit multi-chip microprocessor, IMP-16, was announced in 1973 by National Semiconductors. The 32-bit microprocessors appeared in 1979; it widely used Motorola MC68000, had 32-bit registers, and supported 24-bit addressing. Intel's 80286 was introduced in 1982. The 64-bit processor era was inaugurated by AMD64, an architecture called x86-64, backward compatible with Intel x86 architecture. Dual-core processors appeared in 2005; multicore processors are ubiquitous in today's servers, PCs, tablets, and even smartphones.

The development of distributed systems was only possible after major advances in communication technology. In early 1960s Leonard Kleinrock from UCLA developed theoretical foundations for packet switching networks and in early 1970s for hierarchical routing in packet switching networks. Kleinrock published the first paper on packet switching theory in 1961 and the first book in 1964.

The Advanced Research Projects Agency (ARPA), created in 1958, funded research at multiple universities and businesses sites and the ARPANET project to connect them all within a network was initiated. The Internet is a global network based on the Internet Protocol Suite (TCP/IP); its origins can be traced back to 1965 when Ivan Sutherland, the Head of the Information Processing Technology Office (IPTO) at ARPA, encouraged Lawrence Roberts, who had worked previously at MITs Lincoln laboratories, to become the Chief Scientist at IPTO and to initiate a networking project based on packet switching rather than circuit switching.

In August 1968, DARPA (Defence Advanced Research Projects Agency), the successor of ARPA, released a request for quotation (RFQ) for the development of packet switches called Interface Message Processors (IMPs). A group from Bolt Beranek and Newman (BBN) won the contract. Several researchers including Robert Kahn from BBN, Lawrence Roberts from DARPA, Howard Frank from Network Analysis Corporation, and Leonard Kleinrock from UCLA and their teams played a major role in the overall ARPANET architectural design. The idea of open-architecture networking was first introduced by Kahn in 1972, and his collaboration with Vincent Cerf from Stanford led to the design of TCP/IP. Three groups, one at Stanford, one at BBN, and one at UCLA, won the DARPA contract to implement TCP/IP.

In 1969, BBN installed the first IMP at UCLA. The first two ARPANET nodes interconnected were the Network Measurement Center at the UCLA and SRI International in Menlo Park, California. Two more nodes were added at UC Santa Barbara and the University of Utah. By the end of 1971, there were 15 sites interconnected by ARPANET.

Ethernet technology, developed by Bob Metcalfe at Xerox PARC in 1973 and other local area network technologies, such as token passing rings, allowed PCs and workstations to be connected to the Internet in the 1980s. The Domain Name System (DNS) was invented by Paul Mockapetris of USC/ISI. The DNS permitted a scalable distributed mechanism for resolving hierarchical host names into an Internet address.

UC Berkeley with support from DARPA rewrote the TCP/IP code developed at BBN and incorporated it into the Unix BSD system. In 1985, Dennis Jennings started the NSFNET program at NSF to support the general research and academic communities.

The first distributed computing programs were a pair of worm programs called Creeper and Reaper. In the 1970s, Creeper was using the idle CPU cycles of processors in the ARPANET to copy itself onto the next system and then delete itself from the previous one. Then it was modified to remain on all previous computers. Reaper deleted all copies of the Creeper.

Further readings. Several texts are highly recommended. “Computer Architecture: A Quantitative Approach” [232] by John Hennessy and David Patterson is the authoritative reference for computer architecture. The text “Principles of Computer Systems Design” co-authored by Jerome Saltzer and Frans Kaashoek [430] covers basic concepts in computer system design. “Computer Networks: A Top-Down Approach Featuring the Internet” by James Kurose and Keith Ross is a good introduction to networking.

Amdahl’s paper [21] is a classic, and [416] and [417] are the references for UNIX. [238] covers multicore processors. Load balancing in distributed system is analyzed in [158]. A comprehensive survey of peer-to-peer systems was published in 2010 [420]. *Chord* [457] and *Credence* [502] are important references in the area of peer-to-peer systems.

3.20 Exercises and problems

Problem 1. Do you believe that the homogeneity of a large-scale distributed systems is an advantage? Discuss the reasons for your answer. What aspects of hardware homogeneity are the most relevant in your view and why? What aspects of software homogeneity do you believe are the most relevant and why?

Problem 2. Peer-to-peer systems and clouds share a few goals, but not the means to accomplish them. Compare the two classes of systems in terms of architecture, resource management, scope, and security.

Problem 3. Explain briefly how the *publish–subscribe* paradigm works, and discuss its application to services such as bulletin boards, mailing lists, etc. Outline the design of an event service based on this paradigm, see Fig. 3.10(b). Can you identify a cloud service that emulates an event service?

Problem 4. Tuple spaces can be thought of as an implementation of a distributed shared-memory. Tuple spaces have been developed for many programming languages, including Java, Lisp, Python, Prolog, Smalltalk, and Tcl (Tool Command Language). Explain briefly how tuple spaces work. How secure and scalable are the tuple spaces, e.g., JavaSpaces, you are familiar with?

Problem 5. Arithmetic intensity is defined as the number of floating-point operations divided by the number of bytes in the main memory accessed for running a program. (1) Give examples of computations with low, medium, and high arithmetic intensity. (2) Derive a formula relating the attainable performance to the peak performance of a processor, the peak memory bandwidth, and the arithmetic intensity. (3) How is this formula related to the roofline model?

Problem 6. Read Section 3.5 of [232] regarding dynamic instruction scheduling and answer the following questions: (1) What is the role of the reservation stations used by Tomasulo's approach for dynamic instruction scheduling? (2) What are instruction execution steps for dynamic instruction scheduling? (3) Draw a diagram of a system with two reservation stations.

Problem 7. There are several approaches to hardware multithreading, fine-grain, coarse-grain, and simultaneous (SMT). What are the benefits and the disadvantages of each one of them, and if and where are they used.

Problem 8. Amazon offers EC2 instances with GPU coprocessors for data-intensive applications. (1) Is it beneficial to have the GPUs attached as coprocessors, or is it a disadvantage? (2) Is the thread scheduling inside a GPU done by hardware or controlled by the application? (3) Is it beneficial to add more CUDA threads to an application?

Problem 9. Reference [52] analyzes the power consumption of computing, storage, and networking infrastructure in a cloud data center. Find the percentage of the total power consumed by the CPUs, the DRAM, the disks, the networking, and the other consumers, and then suggest the most effective means to reduce the power consumption.

Problem 10. Given the 32-bit processor with 64 Kbyte cache, 32 byte block and one block per cache set from Section 3.2 draw a figure showing the cache lines and the 32 bit address generated by the CPU. Show how the bits used for the offset of a word in a block, the index, and the tag are used to identify the block in cache. Repeat the task for a processor with the same cache and block size, but with a two-way set associative cache.