6

The Memory Hierarchy

To this point in our study of systems, we have relied on a simple model of a computer system as a CPU that executes instructions and a memory system that holds instructions and data for the CPU. In our simple model, the memory system is a linear array of bytes, and the CPU can access each memory location in a constant amount of time. While this is an effective model up to a point, it does not reflect the way that modern systems really work.

In practice, a *memory system* is a hierarchy of storage devices with different capacities, costs, and access times. CPU registers hold the most frequently used data. Small, fast *cache memories* nearby the CPU act as staging areas for a subset of the data and instructions stored in the relatively slow main memory. The main memory stages data stored on large, slow disks, which in turn often serve as staging areas for data stored on the disks or tapes of other machines connected by networks.

Memory hierarchies work because well-written programs tend to access the storage at any particular level more frequently than they access the storage at the next lower level. So the storage at the next level can be slower, and thus larger and cheaper per bit. The overall effect is a large pool of memory that costs as much as the cheap storage near the bottom of the hierarchy but that serves data to programs at the rate of the fast storage near the top of the hierarchy.

As a programmer, you need to understand the memory hierarchy because it has a big impact on the performance of your applications. If the data your program needs are stored in a CPU register, then they can be accessed in 0 cycles during the execution of the instruction. If stored in a cache, 4 to 75 cycles. If stored in main memory, hundreds of cycles. And if stored in disk, tens of millions of cycles!

Here, then, is a fundamental and enduring idea in computer systems: if you understand how the system moves data up and down the memory hierarchy, then you can write your application programs so that their data items are stored higher in the hierarchy, where the CPU can access them more quickly.

This idea centers around a fundamental property of computer programs known as *locality*. Programs with good locality tend to access the same set of data items over and over again, or they tend to access sets of nearby data items. Programs with good locality tend to access more data items from the upper levels of the memory hierarchy than programs with poor locality, and thus run faster. For example, on our Core i7 system, the running times of different matrix multiplication kernels that perform the same number of arithmetic operations, but have different degrees of locality, can vary by a factor of almost 40!

In this chapter, we will look at the basic storage technologies—SRAM memory, DRAM memory, ROM memory, and rotating and solid state disks—and describe how they are organized into hierarchies. In particular, we focus on the cache memories that act as staging areas between the CPU and main memory, because they have the most impact on application program performance. We show you how to analyze your C programs for locality, and we introduce techniques for improving the locality in your programs. You will also learn an interesting way to characterize the performance of the memory hierarchy on a particular machine as a "memory mountain" that shows read access times as a function of locality.

6.1 Storage Technologies

Much of the success of computer technology stems from the tremendous progress in storage technology. Early computers had a few kilobytes of random access memory. The earliest IBM PCs didn't even have a hard disk. That changed with the introduction of the IBM PC-XT in 1982, with its 10-megabyte disk. By the year 2015, typical machines had 300,000 times as much disk storage, and the amount of storage was increasing by a factor of 2 every couple of years.

6.1.1 Random Access Memory

Random access memory (RAM) comes in two varieties—static and dynamic. *Static RAM (SRAM)* is faster and significantly more expensive than *dynamic RAM (DRAM)*. SRAM is used for cache memories, both on and off the CPU chip. DRAM is used for the main memory plus the frame buffer of a graphics system. Typically, a desktop system will have no more than a few tens of megabytes of SRAM, but hundreds or thousands of megabytes of DRAM.

Static RAM

SRAM stores each bit in a *bistable* memory cell. Each cell is implemented with a six-transistor circuit. This circuit has the property that it can stay indefinitely in either of two different voltage configurations, or *states*. Any other state will be unstable—starting from there, the circuit will quickly move toward one of the stable states. Such a memory cell is analogous to the inverted pendulum illustrated in Figure 6.1.

The pendulum is stable when it is tilted either all the way to the left or all the way to the right. From any other position, the pendulum will fall to one side or the other. In principle, the pendulum could also remain balanced in a vertical position indefinitely, but this state is *metastable*—the smallest disturbance would make it start to fall, and once it fell it would never return to the vertical position.

Due to its bistable nature, an SRAM memory cell will retain its value indefinitely, as long as it is kept powered. Even when a disturbance, such as electrical noise, perturbs the voltages, the circuit will return to the stable value when the disturbance is removed.

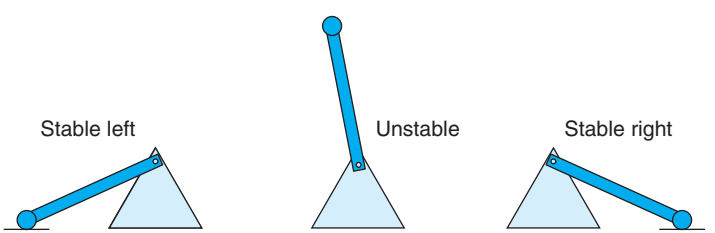


Figure 6.1
Inverted pendulum. Like an SRAM cell, the pendulum has only two stable configurations, or *states*.

	Transistors per bit	Relative access time	Persistent?	Sensitive?	Relative cost	Applications
SRAM	6	1×	Yes	No	1,000×	Cache memory
DRAM	1	10×	No	Yes	1×	Main memory, frame buffers

Figure 6.2 **Characteristics of DRAM and SRAM memory.**

Dynamic RAM

DRAM stores each bit as charge on a capacitor. This capacitor is very small—typically around 30 femtofarads—that is, 30×10^{-15} farads. Recall, however, that a farad is a very large unit of measure. DRAM storage can be made very dense—each cell consists of a capacitor and a single access transistor. Unlike SRAM, however, a DRAM memory cell is very sensitive to any disturbance. When the capacitor voltage is disturbed, it will never recover. Exposure to light rays will cause the capacitor voltages to change. In fact, the sensors in digital cameras and camcorders are essentially arrays of DRAM cells.

Various sources of leakage current cause a DRAM cell to lose its charge within a time period of around 10 to 100 milliseconds. Fortunately, for computers operating with clock cycle times measured in nanoseconds, this retention time is quite long. The memory system must periodically refresh every bit of memory by reading it out and then rewriting it. Some systems also use error-correcting codes, where the computer words are encoded using a few more bits (e.g., a 64-bit word might be encoded using 72 bits), such that circuitry can detect and correct any single erroneous bit within a word.

Figure 6.2 summarizes the characteristics of SRAM and DRAM memory. SRAM is persistent as long as power is applied. Unlike DRAM, no refresh is necessary. SRAM can be accessed faster than DRAM. SRAM is not sensitive to disturbances such as light and electrical noise. The trade-off is that SRAM cells use more transistors than DRAM cells and thus have lower densities, are more expensive, and consume more power.

Conventional DRAMs

The cells (bits) in a DRAM chip are partitioned into d *supercells*, each consisting of w DRAM cells. A $d \times w$ DRAM stores a total of dw bits of information. The supercells are organized as a rectangular array with r rows and c columns, where $rc = d$. Each supercell has an address of the form (i, j), where i denotes the row and j denotes the column.

For example, Figure 6.3 shows the organization of a 16×8 DRAM chip with $d = 16$ supercells, $w = 8$ bits per supercell, $r = 4$ rows, and $c = 4$ columns. The shaded box denotes the supercell at address (2, 1). Information flows in and out of the chip via external connectors called *pins*. Each pin carries a 1-bit signal. Figure 6.3 shows two of these sets of pins: eight `data` pins that can transfer 1 byte

Aside A note on terminology

The storage community has never settled on a standard name for a DRAM array element. Computer architects tend to refer to it as a "cell," overloading the term with the DRAM storage cell. Circuit designers tend to refer to it as a "word," overloading the term with a word of main memory. To avoid confusion, we have adopted the unambiguous term "supercell."

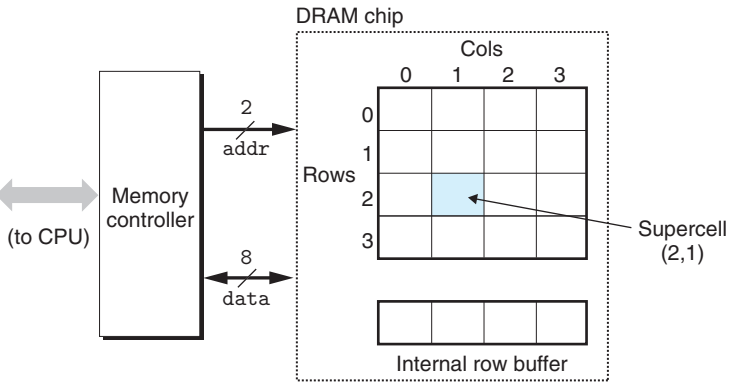


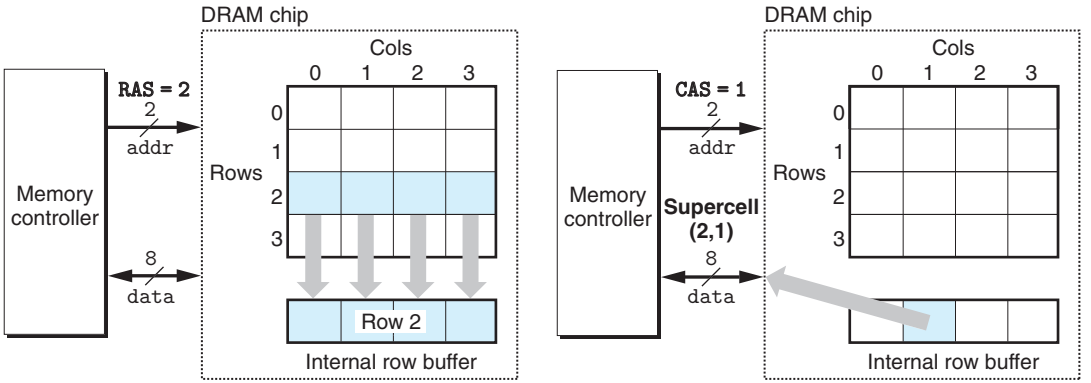
Figure 6.3
High-level view of a 128-bit 16 × 8 DRAM chip.

in or out of the chip, and two `addr` pins that carry two-bit row and column supercell addresses. Other pins that carry control information are not shown.

Each DRAM chip is connected to some circuitry, known as the *memory controller*, that can transfer w bits at a time to and from each DRAM chip. To read the contents of supercell (i, j), the memory controller sends the row address i to the DRAM, followed by the column address j. The DRAM responds by sending the contents of supercell (i, j) back to the controller. The row address i is called a *RAS (row access strobe) request*. The column address j is called a *CAS (column access strobe) request*. Notice that the RAS and CAS requests share the same DRAM address pins.

For example, to read supercell (2, 1) from the 16 × 8 DRAM in Figure 6.3, the memory controller sends row address 2, as shown in Figure 6.4(a). The DRAM responds by copying the entire contents of row 2 into an internal row buffer. Next, the memory controller sends column address 1, as shown in Figure 6.4(b). The DRAM responds by copying the 8 bits in supercell (2, 1) from the row buffer and sending them to the memory controller.

One reason circuit designers organize DRAMs as two-dimensional arrays instead of linear arrays is to reduce the number of address pins on the chip. For example, if our example 128-bit DRAM were organized as a linear array of 16 supercells with addresses 0 to 15, then the chip would need four address pins instead of two. The disadvantage of the two-dimensional array organization is that addresses must be sent in two distinct steps, which increases the access time.



(a) Select row 2 (RAS request).

(b) Select column 1 (CAS request).

Figure 6.4 **Reading the contents of a DRAM supercell.**

Memory Modules

DRAM chips are packaged in *memory modules* that plug into expansion slots on the main system board (motherboard). Core i7 systems use the 240-pin *dual inline memory module (DIMM)*, which transfers data to and from the memory controller in 64-bit chunks.

Figure 6.5 shows the basic idea of a memory module. The example module stores a total of 64 MB (megabytes) using eight 64-Mbit 8M $\times$ 8 DRAM chips, numbered 0 to 7. Each supercell stores 1 byte of *main memory*, and each 64-bit word at byte address A in main memory is represented by the eight supercells whose corresponding supercell address is (i, j). In the example in Figure 6.5, DRAM 0 stores the first (lower-order) byte, DRAM 1 stores the next byte, and so on.

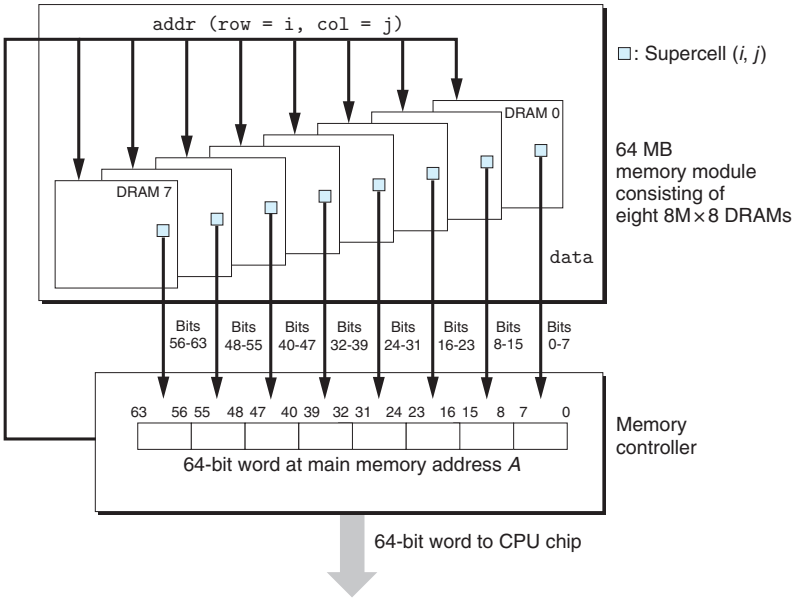
To retrieve the word at memory address A, the memory controller converts A to a supercell address (i, j) and sends it to the memory module, which then broadcasts i and j to each DRAM. In response, each DRAM outputs the 8-bit contents of its (i, j) supercell. Circuitry in the module collects these outputs and forms them into a 64-bit word, which it returns to the memory controller.

Main memory can be aggregated by connecting multiple memory modules to the memory controller. In this case, when the controller receives an address A, the controller selects the module k that contains A, converts A to its (i, j) form, and sends (i, j) to module k.

Practice Problem 6.1 (solution page 696)

In the following, let r be the number of rows in a DRAM array, c the number of columns, b_r the number of bits needed to address the rows, and b_c the number of bits needed to address the columns. For each of the following DRAMs, determine the power-of-2 array dimensions that minimize $\max(b_r, b_c)$, the maximum number of bits needed to address the rows or columns of the array.

Figure 6.5
Reading the contents of a memory module.



Organization	r	c	b_r	b_c	$\max(b_r, b_c)$
16×1					
16×4					
128×8					
512×4					
$1{,}024 \times 4$					

Enhanced DRAMs

There are many kinds of DRAM memories, and new kinds appear on the market with regularity as manufacturers attempt to keep up with rapidly increasing processor speeds. Each is based on the conventional DRAM cell, with optimizations that improve the speed with which the basic DRAM cells can be accessed.

Fast page mode DRAM (FPM DRAM). A conventional DRAM copies an entire row of supercells into its internal row buffer, uses one, and then discards the rest. FPM DRAM improves on this by allowing consecutive accesses to the same row to be served directly from the row buffer. For example, to read four supercells from row i of a conventional DRAM, the memory controller must send four RAS/CAS requests, even though the row address i is identical in each case. To read supercells from the same row of an FPM DRAM, the memory controller sends an initial RAS/CAS request, followed by three CAS requests. The initial RAS/CAS request copies row i into the row buffer and returns the supercell addressed by the

CAS. The next three supercells are served directly from the row buffer, and thus are returned more quickly than the initial supercell.

Extended data out DRAM (EDO DRAM). An enhanced form of FPM DRAM that allows the individual CAS signals to be spaced closer together in time.

Synchronous DRAM (SDRAM). Conventional, FPM, and EDO DRAMs are asynchronous in the sense that they communicate with the memory controller using a set of explicit control signals. SDRAM replaces many of these control signals with the rising edges of the same external clock signal that drives the memory controller. Without going into detail, the net effect is that an SDRAM can output the contents of its supercells at a faster rate than its asynchronous counterparts.

Double Data-Rate Synchronous DRAM (DDR SDRAM). DDR SDRAM is an enhancement of SDRAM that doubles the speed of the DRAM by using both clock edges as control signals. Different types of DDR SDRAMs are characterized by the size of a small prefetch buffer that increases the effective bandwidth: DDR (2 bits), DDR2 (4 bits), and DDR3 (8 bits).

Video RAM (VRAM). Used in the frame buffers of graphics systems. VRAM is similar in spirit to FPM DRAM. Two major differences are that (1) VRAM output is produced by shifting the entire contents of the internal buffer in sequence and (2) VRAM allows concurrent reads and writes to the memory. Thus, the system can be painting the screen with the pixels in the frame buffer (reads) while concurrently writing new values for the next update (writes).

Nonvolatile Memory

DRAMs and SRAMs are *volatile* in the sense that they lose their information if the supply voltage is turned off. *Nonvolatile memories*, on the other hand, retain their information even when they are powered off. There are a variety of nonvolatile memories. For historical reasons, they are referred to collectively as *read-only memories* (ROMs), even though some types of ROMs can be written to as well as read. ROMs are distinguished by the number of times they can be reprogrammed (written to) and by the mechanism for reprogramming them.

Aside Historical popularity of DRAM technologies

Until 1995, most PCs were built with FPM DRAMs. From 1996 to 1999, EDO DRAMs dominated the market, while FPM DRAMs all but disappeared. SDRAMs first appeared in 1995 in high-end systems, and by 2002 most PCs were built with SDRAMs and DDR SDRAMs. By 2010, most server and desktop systems were built with DDR3 SDRAMs. In fact, the Intel Core i7 supports only DDR3 SDRAM.

A *programmable ROM (PROM)* can be programmed exactly once. PROMs include a sort of fuse with each memory cell that can be blown once by zapping it with a high current.

An *erasable programmable ROM (EPROM)* has a transparent quartz window that permits light to reach the storage cells. The EPROM cells are cleared to zeros by shining ultraviolet light through the window. Programming an EPROM is done by using a special device to write ones into the EPROM. An EPROM can be erased and reprogrammed on the order of 1,000 times. An *electrically erasable PROM (EEPROM)* is akin to an EPROM, but it does not require a physically separate programming device, and thus can be reprogrammed in-place on printed circuit cards. An EEPROM can be reprogrammed on the order of 10^5 times before it wears out.

Flash memory is a type of nonvolatile memory, based on EEPROMs, that has become an important storage technology. Flash memories are everywhere, providing fast and durable nonvolatile storage for a slew of electronic devices, including digital cameras, cell phones, and music players, as well as laptop, desktop, and server computer systems. In Section 6.1.3, we will look in detail at a new form of flash-based disk drive, known as a *solid state disk (SSD)*, that provides a faster, sturdier, and less power-hungry alternative to conventional rotating disks.

Programs stored in ROM devices are often referred to as *firmware*. When a computer system is powered up, it runs firmware stored in a ROM. Some systems provide a small set of primitive input and output functions in firmware—for example, a PC's BIOS (basic input/output system) routines. Complicated devices such as graphics cards and disk drive controllers also rely on firmware to translate I/O (input/output) requests from the CPU.

Accessing Main Memory

Data flows back and forth between the processor and the DRAM main memory over shared electrical conduits called *buses*. Each transfer of data between the CPU and memory is accomplished with a series of steps called a *bus transaction*. A *read transaction* transfers data from the main memory to the CPU. A *write transaction* transfers data from the CPU to the main memory.

A *bus* is a collection of parallel wires that carry address, data, and control signals. Depending on the particular bus design, data and address signals can share the same set of wires or can use different sets. Also, more than two devices can share the same bus. The control wires carry signals that synchronize the transaction and identify what kind of transaction is currently being performed. For example, is this transaction of interest to the main memory, or to some other I/O device such as a disk controller? Is the transaction a read or a write? Is the information on the bus an address or a data item?

Figure 6.6 shows the configuration of an example computer system. The main components are the CPU chip, a chipset that we will call an *I/O bridge* (which includes the memory controller), and the DRAM memory modules that make up main memory. These components are connected by a pair of buses: a *system bus* that connects the CPU to the I/O bridge, and a *memory bus* that connects the I/O

Aside A note on bus designs

Bus design is a complex and rapidly changing aspect of computer systems. Different vendors develop different bus architectures as a way to differentiate their products. For example, some Intel systems use chipsets known as the *northbridge* and the *southbridge* to connect the CPU to memory and I/O devices, respectively. In older Pentium and Core 2 systems, a *front side bus* (FSB) connects the CPU to the northbridge. Systems from AMD replace the FSB with the *HyperTransport* interconnect, while newer Intel Core i7 systems use the *QuickPath* interconnect. The details of these different bus architectures are beyond the scope of this text. Instead, we will use the high-level bus architecture from Figure 6.6 as a running example throughout. It is a simple but useful abstraction that allows us to be concrete. It captures the main ideas without being tied too closely to the detail of any proprietary designs.

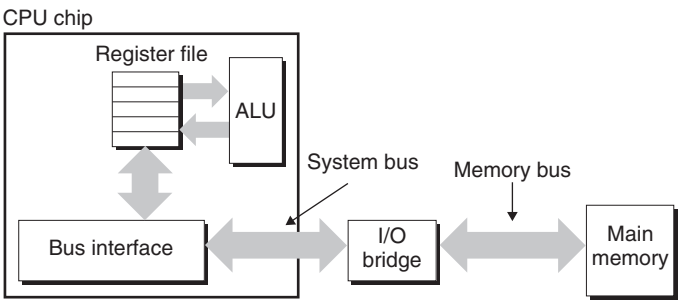


Figure 6.6
Example bus structure that connects the CPU and main memory.

bridge to the main memory. The I/O bridge translates the electrical signals of the system bus into the electrical signals of the memory bus. As we will see, the I/O bridge also connects the system bus and memory bus to an *I/O bus* that is shared by I/O devices such as disks and graphics cards. For now, though, we will focus on the memory bus.

Consider what happens when the CPU performs a load operation such as

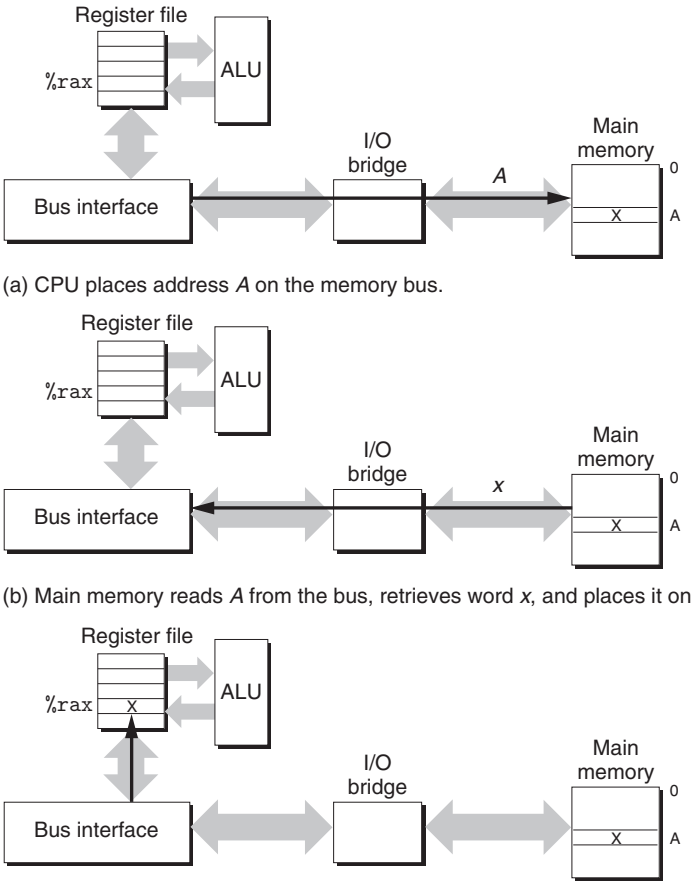
```
movq A,%rax
```

where the contents of address *A* are loaded into register `%rax`. Circuitry on the CPU chip called the *bus interface* initiates a read transaction on the bus. The read transaction consists of three steps. First, the CPU places the address *A* on the system bus. The I/O bridge passes the signal along to the memory bus (Figure 6.7(a)). Next, the main memory senses the address signal on the memory bus, reads the address from the memory bus, fetches the data from the DRAM, and writes the data to the memory bus. The I/O bridge translates the memory bus signal into a system bus signal and passes it along to the system bus (Figure 6.7(b)). Finally, the CPU senses the data on the system bus, reads the data from the bus, and copies the data to register `%rax` (Figure 6.7(c)).

Conversely, when the CPU performs a store operation such as

```
movq %rax,A
```

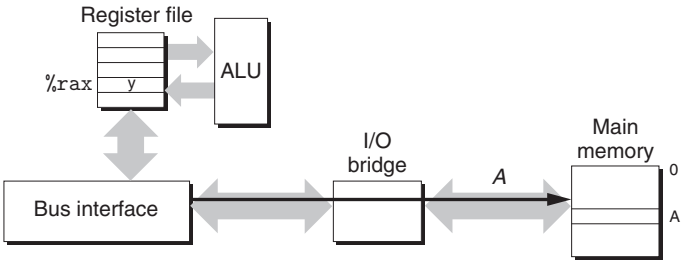
Figure 6.7
Memory read transaction for a load operation: `movq A,%rax`.



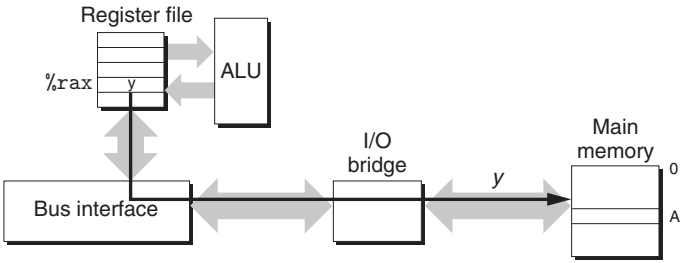
where the contents of register `%rax` are written to address A, the CPU initiates a write transaction. Again, there are three basic steps. First, the CPU places the address on the system bus. The memory reads the address from the memory bus and waits for the data to arrive (Figure 6.8(a)). Next, the CPU copies the data in `%rax` to the system bus (Figure 6.8(b)). Finally, the main memory reads the data from the memory bus and stores the bits in the DRAM (Figure 6.8(c)).

6.1.2 Disk Storage

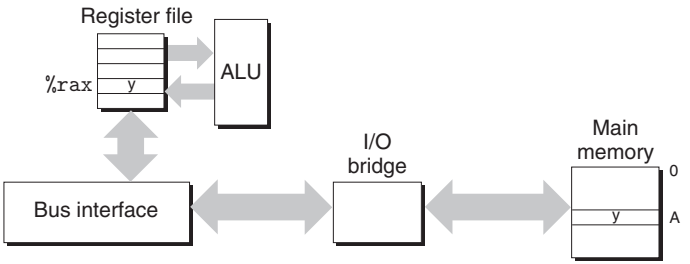
Disks are workhorse storage devices that hold enormous amounts of data, on the order of hundreds to thousands of gigabytes, as opposed to the hundreds or thousands of megabytes in a RAM-based memory. However, it takes on the order of milliseconds to read information from a disk, a hundred thousand times longer than from DRAM and a million times longer than from SRAM.



(a) CPU places address *A* on the memory bus. Main memory reads it and waits for the data word.



(b) CPU places data word *y* on the bus.



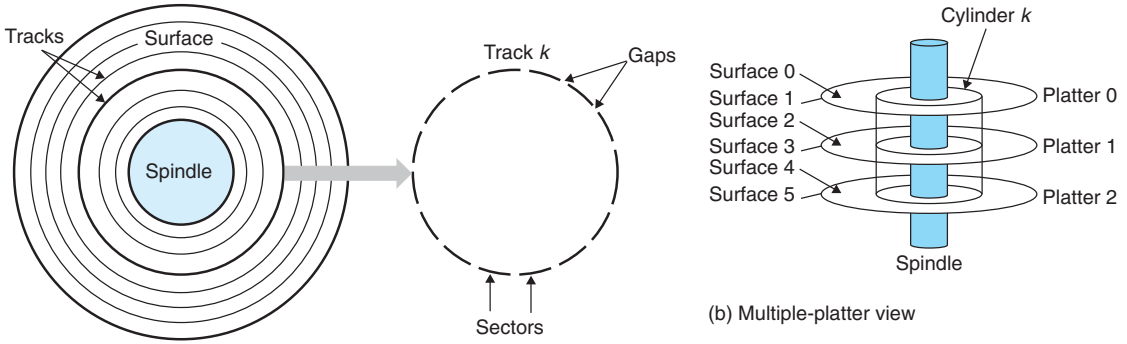
(c) Main memory reads data word *y* from the bus and stores it at address *A*.

Figure 6.8 **Memory write transaction for a store operation:** `movq %rax,A`.

Disk Geometry

Disks are constructed from *platters*. Each platter consists of two sides, or *surfaces*, that are coated with magnetic recording material. A rotating *spindle* in the center of the platter spins the platter at a fixed *rotational rate*, typically between 5,400 and 15,000 *revolutions per minute (RPM)*. A disk will typically contain one or more of these platters encased in a sealed container.

Figure 6.9(a) shows the geometry of a typical disk surface. Each surface consists of a collection of concentric rings called *tracks*. Each track is partitioned into a collection of *sectors*. Each sector contains an equal number of data bits (typically 512 bytes) encoded in the magnetic material on the sector. Sectors are separated by *gaps* where no data bits are stored. Gaps store formatting bits that identify sectors.



(a) Single-platter view

(b) Multiple-platter view

Figure 6.9 **Disk geometry.**

A disk consists of one or more platters stacked on top of each other and encased in a sealed package, as shown in Figure 6.9(b). The entire assembly is often referred to as a *disk drive*, although we will usually refer to it as simply a *disk*. We will sometimes refer to disks as *rotating disks* to distinguish them from flash-based solid state disks (SSDs), which have no moving parts.

Disk manufacturers describe the geometry of multiple-platter drives in terms of *cylinders*, where a cylinder is the collection of tracks on all the surfaces that are equidistant from the center of the spindle. For example, if a drive has three platters and six surfaces, and the tracks on each surface are numbered consistently, then cylinder k is the collection of the six instances of track k.

Disk Capacity

The maximum number of bits that can be recorded by a disk is known as its *maximum capacity*, or simply *capacity*. Disk capacity is determined by the following technology factors:

Recording density (bits/in). The number of bits that can be squeezed into a 1-inch segment of a track.

Track density (tracks/in). The number of tracks that can be squeezed into a 1-inch segment of the radius extending from the center of the platter.

Areal density (bits/in^2). The product of the recording density and the track density.

Disk manufacturers work tirelessly to increase areal density (and thus capacity), and this is doubling every couple of years. The original disks, designed in an age of low areal density, partitioned every track into the same number of sectors, which was determined by the number of sectors that could be recorded on the innermost track. To maintain a fixed number of sectors per track, the sectors were spaced farther apart on the outer tracks. This was a reasonable approach

Aside How much is a gigabyte?

Unfortunately, the meanings of prefixes such as kilo (K), mega (M), giga (G), and tera (T) depend on the context. For measures that relate to the capacity of DRAMs and SRAMs, typically $K = 2^{10}$, $M = 2^{20}$, $G = 2^{30}$, and $T = 2^{40}$. For measures related to the capacity of I/O devices such as disks and networks, typically $K = 10^3$, $M = 10^6$, $G = 10^9$, and $T = 10^{12}$. Rates and throughputs usually use these prefix values as well.

Fortunately, for the back-of-the-envelope estimates that we typically rely on, either assumption works fine in practice. For example, the relative difference between 2^{30} and 10^9 is not that large: $(2^{30} - 10^9)/10^9 \approx 7\%$. Similarly, $(2^{40} - 10^{12})/10^{12} \approx 10\%$.

when areal densities were relatively low. However, as areal densities increased, the gaps between sectors (where no data bits were stored) became unacceptably large. Thus, modern high-capacity disks use a technique known as *multiple zone recording*, where the set of cylinders is partitioned into disjoint subsets known as *recording zones*. Each zone consists of a contiguous collection of cylinders. Each track in each cylinder in a zone has the same number of sectors, which is determined by the number of sectors that can be packed into the innermost track of the zone.

The capacity of a disk is given by the following formula:

$$\text{Capacity} = \frac{\text{\# bytes}}{\text{sector}} \times \frac{\text{average \# sectors}}{\text{track}} \times \frac{\text{\# tracks}}{\text{surface}} \times \frac{\text{\# surfaces}}{\text{platter}} \times \frac{\text{\# platters}}{\text{disk}}$$

For example, suppose we have a disk with five platters, 512 bytes per sector, 20,000 tracks per surface, and an average of 300 sectors per track. Then the capacity of the disk is

$$\begin{aligned}\text{Capacity} &= \frac{512 \text{ bytes}}{\text{sector}} \times \frac{300 \text{ sectors}}{\text{track}} \times \frac{20{,}000 \text{ tracks}}{\text{surface}} \times \frac{2 \text{ surfaces}}{\text{platter}} \times \frac{5 \text{ platters}}{\text{disk}} \\ &= 30{,}720{,}000{,}000 \text{ bytes} \\ &= 30.72 \text{ GB}\end{aligned}$$

Notice that manufacturers express disk capacity in units of gigabytes (GB) or terabytes (TB), where $1\text{ GB} = 10^9$ bytes and $1\text{ TB} = 10^{12}$ bytes.

Practice Problem 6.2 **(solution page 697)**

What is the capacity of a disk with 3 platters, 15,000 cylinders, an average of 500 sectors per track, and 1,024 bytes per sector?

Disk Operation

Disks read and write bits stored on the magnetic surface using a *read/write head* connected to the end of an *actuator arm*, as shown in Figure 6.10(a). By moving

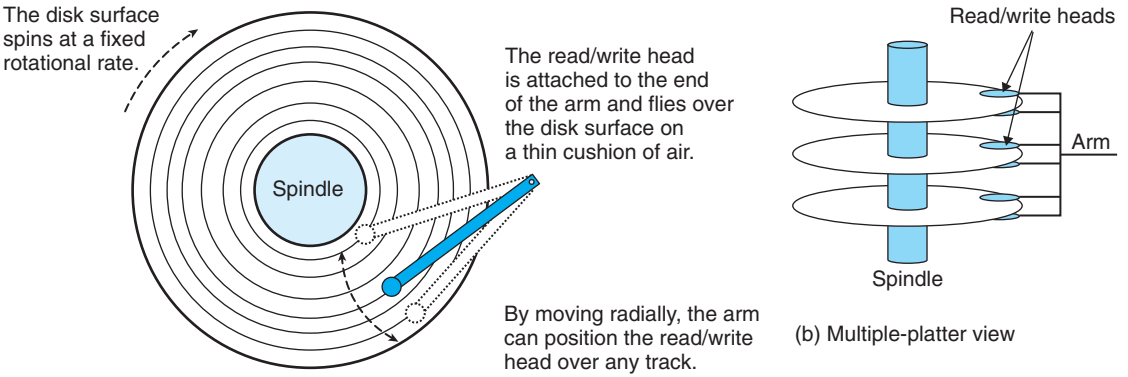


Figure 6.10 **Disk dynamics.**

the arm back and forth along its radial axis, the drive can position the head over any track on the surface. This mechanical motion is known as a *seek*. Once the head is positioned over the desired track, then, as each bit on the track passes underneath, the head can either sense the value of the bit (read the bit) or alter the value of the bit (write the bit). Disks with multiple platters have a separate read/write head for each surface, as shown in Figure 6.10(b). The heads are lined up vertically and move in unison. At any point in time, all heads are positioned on the same cylinder.

The read/write head at the end of the arm flies (literally) on a thin cushion of air over the disk surface at a height of about 0.1 microns and a speed of about 80 km/h. This is analogous to placing a skyscraper on its side and flying it around the world at a height of 2.5 cm (1 inch) above the ground, with each orbit of the earth taking only 8 seconds! At these tolerances, a tiny piece of dust on the surface is like a huge boulder. If the head were to strike one of these boulders, the head would cease flying and crash into the surface (a so-called head crash). For this reason, disks are always sealed in airtight packages.

Disks read and write data in sector-size blocks. The *access time* for a sector has three main components: *seek time*, *rotational latency*, and *transfer time:*

Seek time. To read the contents of some target sector, the arm first positions the head over the track that contains the target sector. The time required to move the arm is called the *seek time*. The seek time, T_{seek}, depends on the previous position of the head and the speed that the arm moves across the surface. The average seek time in modern drives, $T_{\text{avg seek}}$, measured by taking the mean of several thousand seeks to random sectors, is typically on the order of 3 to 9 ms. The maximum time for a single seek, $T_{\text{max seek}}$, can be as high as 20 ms.

Rotational latency. Once the head is in position over the track, the drive waits for the first bit of the target sector to pass under the head. The performance of this step depends on both the position of the surface when the head arrives at the target track and the rotational speed of the disk. In the worst case, the head just misses the target sector and waits for the disk to make a full rotation. Thus, the maximum rotational latency, in seconds, is given by

$$T_{\text{max rotation}} = \frac{1}{\text{RPM}} \times \frac{60 \text{ secs}}{1 \text{ min}}$$

The average rotational latency, $T_{\text{avg rotation}}$, is simply half of $T_{\text{max rotation}}$.

Transfer time. When the first bit of the target sector is under the head, the drive can begin to read or write the contents of the sector. The transfer time for one sector depends on the rotational speed and the number of sectors per track. Thus, we can roughly estimate the average transfer time for one sector in seconds as

$$T_{\text{avg transfer}} = \frac{1}{\text{RPM}} \times \frac{1}{(\text{average \# sectors/track})} \times \frac{60 \text{ secs}}{1 \text{ min}}$$

We can estimate the average time to access the contents of a disk sector as the sum of the average seek time, the average rotational latency, and the average transfer time. For example, consider a disk with the following parameters:

Parameter	Value
Rotational rate	7,200 RPM
$T_{\text{avg seek}}$	9 ms
Average number of sectors/track	400

For this disk, the average rotational latency (in ms) is

$$\begin{aligned} T_{\text{avg rotation}} &= 1/2 \times T_{\text{max rotation}} \\ &= 1/2 \times (60 \text{ secs}/7{,}200 \text{ RPM}) \times 1{,}000 \text{ ms/sec} \\ &\approx 4 \text{ ms} \end{aligned}$$

The average transfer time is

$$\begin{aligned} T_{\text{avg transfer}} &= 60/7{,}200 \text{ RPM} \times 1/400 \text{ sectors/track} \times 1{,}000 \text{ ms/sec} \\ &\approx 0.02 \text{ ms} \end{aligned}$$

Putting it all together, the total estimated access time is

$$\begin{aligned} T_{\text{access}} &= T_{\text{avg seek}} + T_{\text{avg rotation}} + T_{\text{avg transfer}} \\ &= 9 \text{ ms} + 4 \text{ ms} + 0.02 \text{ ms} \\ &= 13.02 \text{ ms} \end{aligned}$$

This example illustrates some important points:

- The time to access the 512 bytes in a disk sector is dominated by the seek time and the rotational latency. Accessing the first byte in the sector takes a long time, but the remaining bytes are essentially free.
- Since the seek time and rotational latency are roughly the same, twice the seek time is a simple and reasonable rule for estimating disk access time.
- The access time for a 64-bit word stored in SRAM is roughly 4 ns, and 60 ns for DRAM. Thus, the time to read a 512-byte sector-size block from memory is roughly 256 ns for SRAM and 4,000 ns for DRAM. The disk access time, roughly 10 ms, is about 40,000 times greater than SRAM, and about 2,500 times greater than DRAM.

Practice Problem 6.3 **(solution page 697)**

Estimate the average time (in ms) to access a sector on the following disk:

Parameter	Value
Rotational rate	12,000 RPM
$T_{\text{avg seek}}$	5 ms
Average number of sectors/track	300

Logical Disk Blocks

As we have seen, modern disks have complex geometries, with multiple surfaces and different recording zones on those surfaces. To hide this complexity from the operating system, modern disks present a simpler view of their geometry as a sequence of B sector-size *logical blocks*, numbered $0, 1, \ldots, B-1$. A small hardware/firmware device in the disk package, called the *disk controller*, maintains the mapping between logical block numbers and actual (physical) disk sectors.

When the operating system wants to perform an I/O operation such as reading a disk sector into main memory, it sends a command to the disk controller asking it to read a particular logical block number. Firmware on the controller performs a fast table lookup that translates the logical block number into a *(surface, track, sector)* triple that uniquely identifies the corresponding physical sector. Hardware on the controller interprets this triple to move the heads to the appropriate cylinder, waits for the sector to pass under the head, gathers up the bits sensed by the head into a small memory buffer on the controller, and copies them into main memory.

Practice Problem 6.4 **(solution page 697)**

Suppose that a 1 MB file consisting of 512-byte logical blocks is stored on a disk drive with the following characteristics:

Aside Formatted disk capacity

Before a disk can be used to store data, it must be *formatted* by the disk controller. This involves filling in the gaps between sectors with information that identifies the sectors, identifying any cylinders with surface defects and taking them out of action, and setting aside a set of cylinders in each zone as spares that can be called into action if one or more cylinders in the zone goes bad during the lifetime of the disk. The *formatted capacity* quoted by disk manufacturers is less than the maximum capacity because of the existence of these spare cylinders.

Parameter	Value
Rotational rate	13,000 RPM
$T_{\text{avg seek}}$	6 ms
Average number of sectors/track	5,000
Surfaces	4
Sector size	512 bytes

For each case below, suppose that a program reads the logical blocks of the file sequentially, one after the other, and that the time to position the head over the first block is $T_{\text{avg seek}} + T_{\text{avg rotation}}$.

A. *Best case:* Estimate the optimal time (in ms) required to read the file given the best possible mapping of logical blocks to disk sectors (i.e., sequential).

B. *Random case:* Estimate the time (in ms) required to read the file if blocks are mapped randomly to disk sectors.

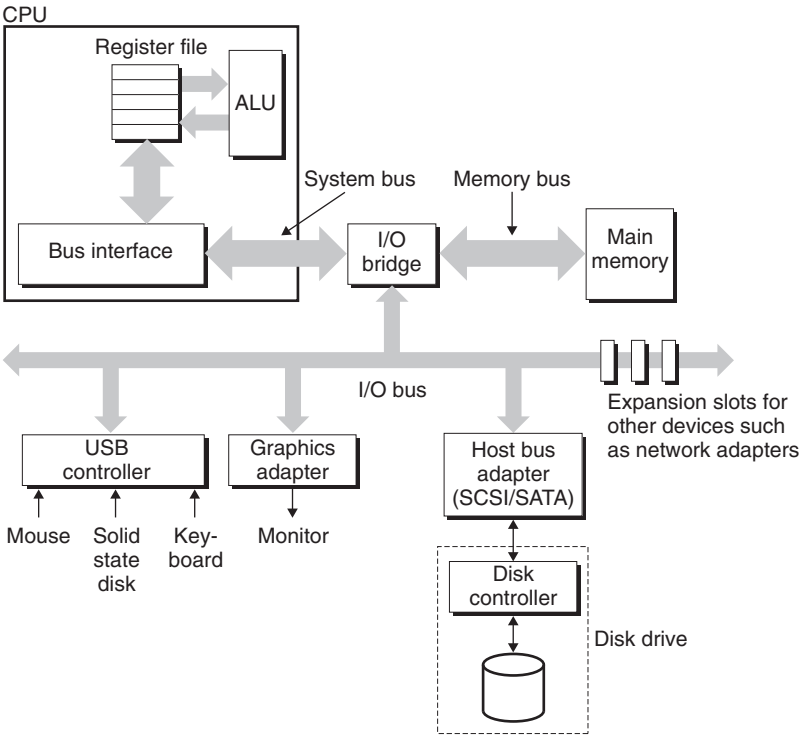
Connecting I/O Devices

Input/output (I/O) devices such as graphics cards, monitors, mice, keyboards, and disks are connected to the CPU and main memory using an *I/O bus*. Unlike the system bus and memory buses, which are CPU-specific, I/O buses are designed to be independent of the underlying CPU. Figure 6.11 shows a representative I/O bus structure that connects the CPU, main memory, and I/O devices.

Although the I/O bus is slower than the system and memory buses, it can accommodate a wide variety of third-party I/O devices. For example, the bus in Figure 6.11 has three different types of devices attached to it.

- A *Universal Serial Bus (USB)* controller is a conduit for devices attached to a USB bus, which is a wildly popular standard for connecting a variety of peripheral I/O devices, including keyboards, mice, modems, digital cameras, game controllers, printers, external disk drives, and solid state disks. USB 3.0 buses have a maximum bandwidth of 625 MB/s. USB 3.1 buses have a maximum bandwidth of 1,250 MB/s.

Figure 6.11
Example bus structure that connects the CPU, main memory, and I/O devices.



- A *graphics card* (or *adapter*) contains hardware and software logic that is responsible for painting the pixels on the display monitor on behalf of the CPU.
- A *host bus adapter* that connects one or more disks to the I/O bus using a communication protocol defined by a particular *host bus interface*. The two most popular such interfaces for disks are *SCSI* (pronounced "scuzzy") and *SATA* (pronounced "sat-uh"). SCSI disks are typically faster and more expensive than SATA drives. A SCSI host bus adapter (often called a *SCSI controller*) can support multiple disk drives, as opposed to SATA adapters, which can only support one drive.

Additional devices such as *network adapters* can be attached to the I/O bus by plugging the adapter into empty *expansion slots* on the motherboard that provide a direct electrical connection to the bus.

Accessing Disks

While a detailed description of how I/O devices work and how they are programmed is outside our scope here, we can give you a general idea. For example, Figure 6.12 summarizes the steps that take place when a CPU reads data from a disk.

Aside Advances in I/O bus designs

The I/O bus in Figure 6.11 is a simple abstraction that allows us to be concrete, without being tied too closely to the details of any specific system. It is based on the *peripheral component interconnect (PCI)* bus, which was popular until around 2010. In the PCI model, each device in the system shares the bus, and only one device at a time can access these wires. In modern systems, the shared PCI bus has been replaced by a *PCI express* (PCIe) bus, which is a set of high-speed serial, point-to-point links connected by switches, akin to the switched Ethernets that you will learn about in Chapter 11. A PCIe bus, with a maximum throughput of 16 GB/s, is an order of magnitude faster than a PCI bus, which has a maximum throughput of 533 MB/s. Except for measured I/O performance, the differences between the different bus designs are not visible to application programs, so we will use the simple shared bus abstraction throughout the text.

The CPU issues commands to I/O devices using a technique called *memory-mapped I/O* (Figure 6.12(a)). In a system with memory-mapped I/O, a block of addresses in the address space is reserved for communicating with I/O devices. Each of these addresses is known as an *I/O port*. Each device is associated with (or mapped to) one or more ports when it is attached to the bus.

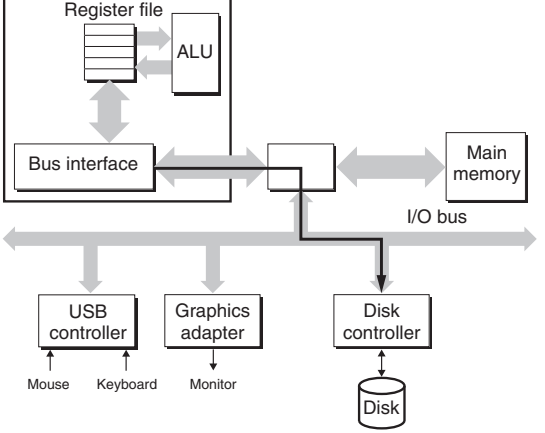
As a simple example, suppose that the disk controller is mapped to port `0xa0`. Then the CPU might initiate a disk read by executing three store instructions to address `0xa0`: The first of these instructions sends a command word that tells the disk to initiate a read, along with other parameters such as whether to interrupt the CPU when the read is finished. (We will discuss interrupts in Section 8.1.) The second instruction indicates the logical block number that should be read. The third instruction indicates the main memory address where the contents of the disk sector should be stored.

After it issues the request, the CPU will typically do other work while the disk is performing the read. Recall that a 1 GHz processor with a 1 ns clock cycle can potentially execute 16 million instructions in the 16 ms it takes to read the disk. Simply waiting and doing nothing while the transfer is taking place would be enormously wasteful.

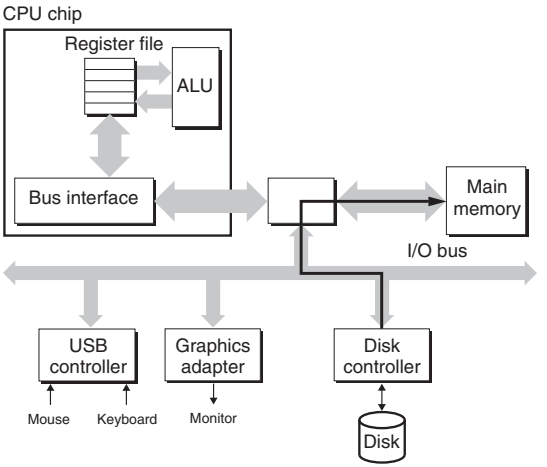
After the disk controller receives the read command from the CPU, it translates the logical block number to a sector address, reads the contents of the sector, and transfers the contents directly to main memory, without any intervention from the CPU (Figure 6.12(b)). This process, whereby a device performs a read or write bus transaction on its own, without any involvement of the CPU, is known as *direct memory access* (DMA). The transfer of data is known as a *DMA transfer*.

After the DMA transfer is complete and the contents of the disk sector are safely stored in main memory, the disk controller notifies the CPU by sending an interrupt signal to the CPU (Figure 6.12(c)). The basic idea is that an interrupt signals an external pin on the CPU chip. This causes the CPU to stop what it is currently working on and jump to an operating system routine. The routine records the fact that the I/O has finished and then returns control to the point where the CPU was interrupted.

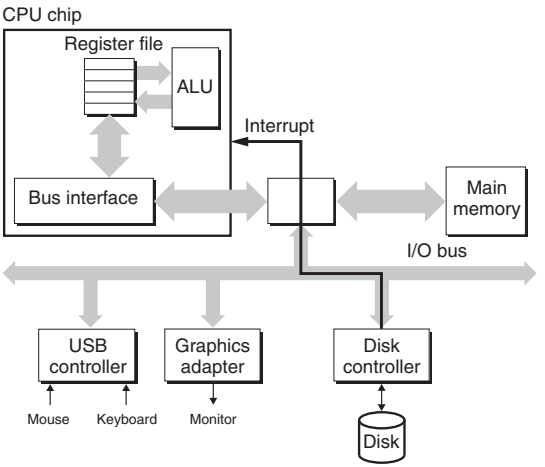
Reading a disk sector.



(a) The CPU initiates a disk read by writing a command, logical block number, and destination memory address to the memory-mapped address associated with the disk.



(b) The disk controller reads the sector and performs a DMA transfer into main memory.



(c) When the DMA transfer is complete, the disk controller notifies the CPU with an interrupt.

Aside Characteristics of a commercial disk drive

Disk manufacturers publish a lot of useful high-level technical information on their Web sites. For example, the Seagate Web site contains the following information (and much more!) about one of their popular drives, the Barracuda 7400. (Seagate.com)

Geometry characteristic	Value	Geometry characteristic	Value
Surface diameter	3.5 in	Rotational rate	7,200 RPM
Formatted capacity	3 TB	Average rotational latency	4.16 ms
Platters	3	Average seek time	8.5 ms
Surfaces	6	Track-to-track seek time	1.0 ms
Logical blocks	5,860,533,168	Average transfer rate	156 MB/s
Logical block size	512 bytes	Maximum sustained transfer rate	210 MB/s

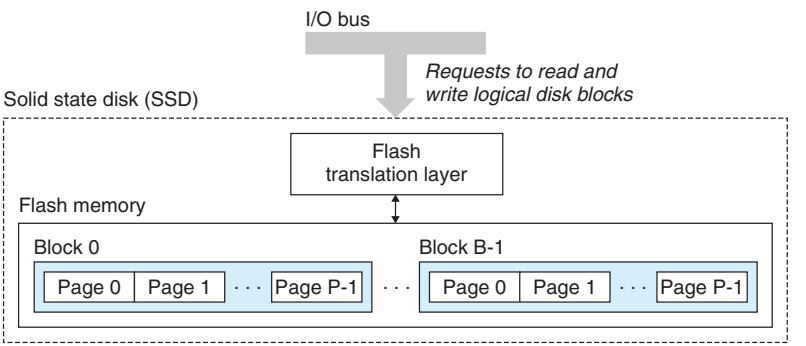


Figure 6.13
Solid state disk (SSD).

6.1.3 Solid State Disks

A solid state disk (SSD) is a storage technology, based on flash memory (Section 6.1.1), that in some situations is an attractive alternative to the conventional rotating disk. Figure 6.13 shows the basic idea. An SSD package plugs into a standard disk slot on the I/O bus (typically USB or SATA) and behaves like any other disk, processing requests from the CPU to read and write logical disk blocks. An SSD package consists of one or more flash memory chips, which replace the mechanical drive in a conventional rotating disk, and a *flash translation layer*, which is a hardware/firmware device that plays the same role as a disk controller, translating requests for logical blocks into accesses of the underlying physical device.

Figure 6.14 shows the performance characteristics of a typical SSD. Notice that reading from SSDs is faster than writing. The difference between random reading and writing performance is caused by a fundamental property of the underlying flash memory. As shown in Figure 6.13, a flash memory consists of a sequence of B *blocks*, where each block consists of P pages. Typically, pages are 512 bytes to 4 KB in size, and a block consists of 32–128 pages, with total block sizes ranging from 16

Reads		Writes	
Sequential read throughput	550 MB/s	Sequential write throughput	470 MB/s
Random read throughput (IOPS)	89,000 IOPS	Random write throughput (IOPS)	74,000 IOPS
Random read throughput (MB/s)	365 MB/s	Random write throughput (MB/s)	303 MB/s
Avg. sequential read access time	50 μs	Avg. sequential write access time	60 μs

Figure 6.14 Performance characteristics of a commercial solid state disk. Source: Intel SSD 730 product specification [53]. *IOPS* is I/O operations per second. Throughput numbers are based on reads and writes of 4 KB blocks. (Intel SSD 730 product specification. Intel Corporation. 52.)

KB to 512 KB. Data are read and written in units of pages. A page can be written only after the entire block to which it belongs has been *erased* (typically, this means that all bits in the block are set to 1). However, once a block is erased, each page in the block can be written once with no further erasing. A block wears out after roughly 100,000 repeated writes. Once a block wears out, it can no longer be used.

Random writes are slower for two reasons. First, erasing a block takes a relatively long time, on the order of 1 ms, which is more than an order of magnitude longer than it takes to access a page. Second, if a write operation attempts to modify a page p that contains existing data (i.e., not all ones), then any pages in the same block with useful data must be copied to a new (erased) block before the write to page p can occur. Manufacturers have developed sophisticated logic in the flash translation layer that attempts to amortize the high cost of erasing blocks and to minimize the number of internal copies on writes, but it is unlikely that random writing will ever perform as well as reading.

SSDs have a number of advantages over rotating disks. They are built of semiconductor memory, with no moving parts, and thus have much faster random access times than rotating disks, use less power, and are more rugged. However, there are some disadvantages. First, because flash blocks wear out after repeated writes, SSDs have the potential to wear out as well. *Wear-leveling* logic in the flash translation layer attempts to maximize the lifetime of each block by spreading erasures evenly across all blocks. In practice, the wear-leveling logic is so good that it takes many years for SSDs to wear out (see Practice Problem 6.5). Second, SSDs are about 30 times more expensive per byte than rotating disks, and thus the typical storage capacities are significantly less than rotating disks. However, SSD prices are decreasing rapidly as they become more popular, and the gap between the two is decreasing.

SSDs have completely replaced rotating disks in portable music devices, are popular as disk replacements in laptops, and have even begun to appear in desktops and servers. While rotating disks are here to stay, it is clear that SSDs are an important alternative.

Practice Problem 6.5 (solution page 698)

As we have seen, a potential drawback of SSDs is that the underlying flash memory can wear out. For example, for the SSD in Figure 6.14, Intel guarantees about

128 petabytes (128×10^{15} bytes) of writes before the drive wears out. Given this assumption, estimate the lifetime (in years) of this SSD for the following workloads:

A. *Worst case for sequential writes:* The SSD is written to continuously at a rate of 470 MB/s (the average sequential write throughput of the device).

B. *Worst case for random writes:* The SSD is written to continuously at a rate of 303 MB/s (the average random write throughput of the device).

C. *Average case:* The SSD is written to at a rate of 20 GB/day (the average daily write rate assumed by some computer manufacturers in their mobile computer workload simulations).

6.1.4 Storage Technology Trends

There are several important concepts to take away from our discussion of storage technologies.

Different storage technologies have different price and performance trade-offs. SRAM is somewhat faster than DRAM, and DRAM is much faster than disk. On the other hand, fast storage is always more expensive than slower storage. SRAM costs more per byte than DRAM. DRAM costs much more than disk. SSDs split the difference between DRAM and rotating disk.

The price and performance properties of different storage technologies are changing at dramatically different rates. Figure 6.15 summarizes the price and performance properties of storage technologies since 1985, shortly after the first PCs were introduced. The numbers were culled from back issues of trade magazines and the Web. Although they were collected in an informal survey, the numbers reveal some interesting trends.

Since 1985, both the cost and performance of SRAM technology have improved at roughly the same rate. Access times and cost per megabyte have decreased by a factor of about 100 (Figure 6.15(a)). However, the trends for DRAM and disk are much more dramatic and divergent. While the cost per megabyte of DRAM has decreased by a factor of 44,000 (more than four orders of magnitude!), DRAM access times have decreased by only a factor of 10 (Figure 6.15(b)). Disk technology has followed the same trend as DRAM and in even more dramatic fashion. While the cost of a megabyte of disk storage has plummeted by a factor of more than 3,000,000 (more than six orders of magnitude!) since 1980, access times have improved much more slowly, by only a factor of 25 (Figure 6.15(c)). These startling long-term trends highlight a basic truth of memory and disk technology: it is much easier to increase density (and thereby reduce cost) than to decrease access time.

DRAM and disk performance are lagging behind CPU performance. As we see in Figure 6.15(d), CPU cycle times improved by a factor of 500 between 1985 and 2010. If we look at the *effective cycle time* —which we define to be the cycle time of an individual CPU (processor) divided by the number of its processor cores— then the improvement between 1985 and 2010 is even greater, a factor of 2,000.

Metric	1985	1990	1995	2000	2005	2010	2015	2015:1985
$/MB	2,900	320	256	100	75	60	25	116
Access (ns)	150	35	15	3	2	1.5	1.3	115

(a) SRAM trends

Metric	1985	1990	1995	2000	2005	2010	2015	2015:1985
$/MB	880	100	30	1	0.1	0.06	0.02	44,000
Access (ns)	200	100	70	60	50	40	20	10
Typical size (MB)	0.256	4	16	64	2,000	8,000	16,000	62,500

(b) DRAM trends

Metric	1985	1990	1995	2000	2005	2010	2015	2015:1985
$/GB	100,000	8,000	300	10	5	0.3	0.03	3,333,333
Min. seek time (ms)	75	28	10	8	5	3	3	25
Typical size (GB)	0.01	0.16	1	20	160	1,500	3,000	300,000

(c) Rotating disk trends

Metric	1985	1990	1995	2000	2003	2005	2010	2015	2015:1985
Intel CPU	80286	80386	Pent.	P-III	Pent. 4	Core 2	Core i7 (n)	Core i7 (h)	—
Clock rate (MHz)	6	20	150	600	3,300	2,000	2,500	3,000	500
Cycle time (ns)	166	50	6	1.6	0.3	0.5	0.4	0.33	500
Cores	1	1	1	1	1	2	4	4	4
Effective cycle time (ns)	166	50	6	1.6	0.30	0.25	0.10	0.08	2,075

(d) CPU trends

Figure 6.15 **Storage and processing technology trends.** The Core i7 circa 2010 uses the Nehalem processor core. The Core i7 circa 2015 uses the Haswell core.

The split in the CPU performance curve around 2003 reflects the introduction of multi-core processors (see aside on page 641). After this split, cycle times of individual cores actually increased a bit before starting to decrease again, albeit at a slower rate than before.

Note that while SRAM performance lags, it is roughly keeping up. However, the gap between DRAM and disk performance and CPU performance is actually widening. Until the advent of multi-core processors around 2003, this performance gap was a function of latency, with DRAM and disk access times decreasing more slowly than the cycle time of an individual processor. However, with the introduction of multiple cores, this performance gap is increasingly a function of

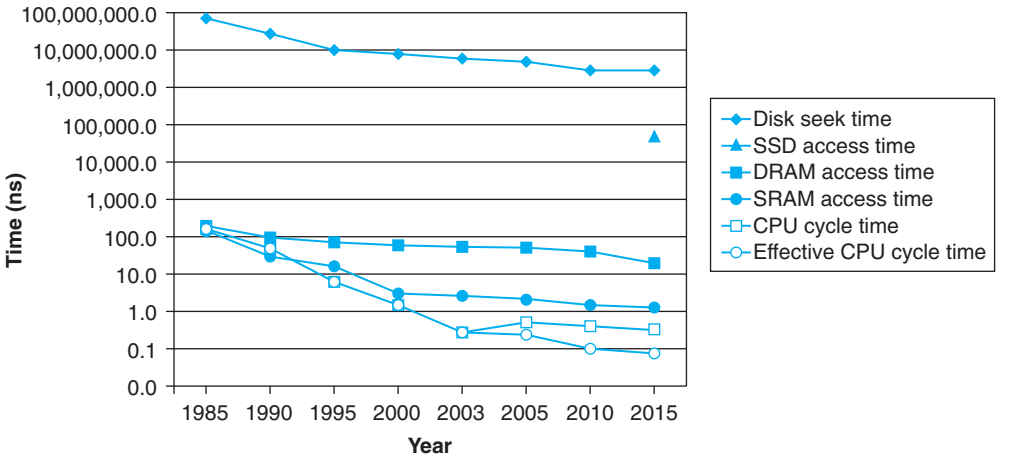


Figure 6.16 **The gap between disk, DRAM, and CPU speeds.**

throughput, with multiple processor cores issuing requests to the DRAM and disk in parallel.

The various trends are shown quite clearly in Figure 6.16, which plots the access and cycle times from Figure 6.15 on a semi-log scale.

As we will see in Section 6.4, modern computers make heavy use of SRAM-based caches to try to bridge the processor–memory gap. This approach works because of a fundamental property of application programs known as *locality*, which we discuss next.

Practice Problem 6.6 (solution page 698)

Using the data from the years 2005 to 2015 in Figure 6.15(c), estimate the year when you will be able to buy a petabyte (10^{15} bytes) of rotating disk storage for $200. Assume actual dollars (no inflation).

6.2 Locality

Well-written computer programs tend to exhibit good *locality*. That is, they tend to reference data items that are near other recently referenced data items or that were recently referenced themselves. This tendency, known as the *principle of locality*, is an enduring concept that has enormous impact on the design and performance of hardware and software systems.

Locality is typically described as having two distinct forms: *temporal locality* and *spatial locality*. In a program with good temporal locality, a memory location that is referenced once is likely to be referenced again multiple times in the near future. In a program with good spatial locality, if a memory location is referenced

Aside When cycle time stood still: The advent of multi-core processors

The history of computers is marked by some singular events that caused profound changes in the industry and the world. Interestingly, these inflection points tend to occur about once per decade: the development of Fortran in the 1950s, the introduction of the IBM 360 in the early 1960s, the dawn of the Internet (then called ARPANET) in the early 1970s, the introduction of the IBM PC in the early 1980s, and the creation of the World Wide Web in the early 1990s.

The most recent such event occurred early in the 21st century, when computer manufacturers ran headlong into the so-called power wall, discovering that they could no longer increase CPU clock frequencies as quickly because the chips would then consume too much power. The solution was to improve performance by replacing a single large processor with multiple smaller processor *cores*, each a complete processor capable of executing programs independently and in parallel with the other cores. This *multi-core* approach works in part because the power consumed by a processor is proportional to $P = fCV^2$, where f is the clock frequency, C is the capacitance, and V is the voltage. The capacitance C is roughly proportional to the area, so the power drawn by multiple cores can be held constant as long as the total area of the cores is constant. As long as feature sizes continue to shrink at the exponential Moore's Law rate, the number of cores in each processor, and thus its effective performance, will continue to increase.

From this point forward, computers will get faster not because the clock frequency increases but because the number of cores in each processor increases, and because architectural innovations increase the efficiency of programs running on those cores. We can see this trend clearly in Figure 6.16. CPU cycle time reached its lowest point in 2003 and then actually started to rise before leveling off and starting to decline again at a slower rate than before. However, because of the advent of multi-core processors (dual-core in 2004 and quad-core in 2007), the effective cycle time continues to decrease at close to its previous rate.

once, then the program is likely to reference a nearby memory location in the near future.

Programmers should understand the principle of locality because, in general, *programs with good locality run faster than programs with poor locality*. All levels of modern computer systems, from the hardware, to the operating system, to application programs, are designed to exploit locality. At the hardware level, the principle of locality allows computer designers to speed up main memory accesses by introducing small fast memories known as *cache memories* that hold blocks of the most recently referenced instructions and data items. At the operating system level, the principle of locality allows the system to use the main memory as a cache of the most recently referenced chunks of the virtual address space. Similarly, the operating system uses main memory to cache the most recently used disk blocks in the disk file system. The principle of locality also plays a crucial role in the design of application programs. For example, Web browsers exploit temporal locality by caching recently referenced documents on a local disk. High-volume Web servers hold recently requested documents in front-end disk caches that satisfy requests for these documents without requiring any intervention from the server.

```
1  int sumvec(int v[N])
2  {
3      int i, sum = 0;
4
5      for (i = 0; i < N; i++)
6          sum += v[i];
7      return sum;
8  }
```

(a)

Address	0	4	8	12	16	20	24	28
Contents	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7
Access order	1	2	3	4	5	6	7	8

(b)

Figure 6.17 (a) A function with good locality. (b) Reference pattern for vector v **($N = 8$).** Notice how the vector elements are accessed in the same order that they are stored in memory.

6.2.1 Locality of References to Program Data

Consider the simple function in Figure 6.17(a) that sums the elements of a vector. Does this function have good locality? To answer this question, we look at the reference pattern for each variable. In this example, the `sum` variable is referenced once in each loop iteration, and thus there is good temporal locality with respect to `sum`. On the other hand, since `sum` is a scalar, there is no spatial locality with respect to `sum`.

As we see in Figure 6.17(b), the elements of vector `v` are read sequentially, one after the other, in the order they are stored in memory (we assume for convenience that the array starts at address 0). Thus, with respect to variable `v`, the function has good spatial locality but poor temporal locality since each vector element is accessed exactly once. Since the function has either good spatial or temporal locality with respect to each variable in the loop body, we can conclude that the `sumvec` function enjoys good locality.

A function such as `sumvec` that visits each element of a vector sequentially is said to have a *stride-1 reference pattern* (with respect to the element size). We will sometimes refer to stride-1 reference patterns as *sequential reference patterns*. Visiting every kth element of a contiguous vector is called a *stride-k reference pattern*. Stride-1 reference patterns are a common and important source of spatial locality in programs. In general, as the stride increases, the spatial locality decreases.

Stride is also an important issue for programs that reference multidimensional arrays. For example, consider the `sumarrayrows` function in Figure 6.18(a) that sums the elements of a two-dimensional array.

The doubly nested loop reads the elements of the array in *row-major order*. That is, the inner loop reads the elements of the first row, then the second row, and so on. The `sumarrayrows` function enjoys good spatial locality because it references the array in the same row-major order that the array is stored (Figure 6.18(b)). The result is a nice stride-1 reference pattern with excellent spatial locality.

```
1   int sumarrayrows(int a[M][N])
2   {
3       int i, j, sum = 0;
4
5       for (i = 0; i < M; i++)
6           for (j = 0; j < N; j++)
7               sum += a[i][j];
8       return sum;
9   }
```

(a)

Address	0	4	8	12	16	20
Contents	a_{00}	a_{01}	a_{02}	a_{10}	a_{11}	a_{12}
Access order	1	2	3	4	5	6

(b)

Figure 6.18 **(a) Another function with good locality. (b) Reference pattern for array** a **($M = 2$, $N = 3$).** There is good spatial locality because the array is accessed in the same row-major order in which it is stored in memory.

```
1   int sumarraycols(int a[M][N])
2   {
3       int i, j, sum = 0;
4
5       for (j = 0; j < N; j++)
6           for (i = 0; i < M; i++)
7               sum += a[i][j];
8       return sum;
9   }
```

(a)

Address	0	4	8	12	16	20
Contents	a_{00}	a_{01}	a_{02}	a_{10}	a_{11}	a_{12}
Access order	1	3	5	2	4	6

(b)

Figure 6.19 **(a) A function with poor spatial locality. (b) Reference pattern for array** a **($M = 2$, $N = 3$).** The function has poor spatial locality because it scans memory with a stride-N reference pattern.

Seemingly trivial changes to a program can have a big impact on its locality. For example, the `sumarraycols` function in Figure 6.19(a) computes the same result as the `sumarrayrows` function in Figure 6.18(a). The only difference is that we have interchanged the i and j loops. What impact does interchanging the loops have on its locality?

The `sumarraycols` function suffers from poor spatial locality because it scans the array column-wise instead of row-wise. Since C arrays are laid out in memory row-wise, the result is a stride-N reference pattern, as shown in Figure 6.19(b).

6.2.2 Locality of Instruction Fetches

Since program instructions are stored in memory and must be fetched (read) by the CPU, we can also evaluate the locality of a program with respect to its instruction fetches. For example, in Figure 6.17 the instructions in the body of the

for loop are executed in sequential memory order, and thus the loop enjoys good spatial locality. Since the loop body is executed multiple times, it also enjoys good temporal locality.

An important property of code that distinguishes it from program data is that it is rarely modified at run time. While a program is executing, the CPU reads its instructions from memory. The CPU rarely overwrites or modifies these instructions.

6.2.3 Summary of Locality

In this section, we have introduced the fundamental idea of locality and have identified some simple rules for qualitatively evaluating the locality in a program:

- Programs that repeatedly reference the same variables enjoy good temporal locality.
- For programs with stride-k reference patterns, the smaller the stride, the better the spatial locality. Programs with stride-1 reference patterns have good spatial locality. Programs that hop around memory with large strides have poor spatial locality.
- Loops have good temporal and spatial locality with respect to instruction fetches. The smaller the loop body and the greater the number of loop iterations, the better the locality.

Later in this chapter, after we have learned about cache memories and how they work, we will show you how to quantify the idea of locality in terms of cache hits and misses. It will also become clear to you why programs with good locality typically run faster than programs with poor locality. Nonetheless, knowing how to glance at a source code and getting a high-level feel for the locality in the program is a useful and important skill for a programmer to master.

Practice Problem 6.7 **(solution page 698)**

Permute the loops in the following function so that it scans the three-dimensional array a with a stride-1 reference pattern.

```
1   int productarray3d(int a[N][N][N])
2   {
3       int i, j, k, product = 1;
4
5       for (i = N-1; i >= 0; i--) {
6           for (j = N-1; j >= 0; j--) {
7               for (k = N-1; k >= 0; k--) {
8                   product *= a[j][k][i];
9               }
10          }
11      }
12      return product;
13  }
```

(a) An array of `structs`

```
#define N 1000

typedef struct {
    int vel[3];
    int acc[3];
} point;

point p[N];
```

(b) The `clear1` function

```
void clear1(point *p, int n)
{
    int i, j;

    for (i = 0; i < n; i++) {
        for (j = 0; j < 3; j++)
            p[i].vel[j] = 0;
        for (j = 0; j < 3; j++)
            p[i].acc[j] = 0;
    }
}
```

(c) The `clear2` function

```
void clear2(point *p, int n)
{
    int i, j;

    for (i = 0; i < n; i++) {
        for (j = 0; j < 3; j++) {
            p[i].vel[j] = 0;
            p[i].acc[j] = 0;
        }
    }
}
```

(d) The `clear3` function

```
void clear3(point *p, int n)
{
    int i, j;

    for (j = 0; j < 3; j++) {
        for (i = 0; i < n; i++)
            p[i].vel[j] = 0;
        for (i = 0; i < n; i++)
            p[i].acc[j] = 0;
    }
}
```

Figure 6.20 **Code examples for Practice Problem 6.8.**

Practice Problem 6.8 **(solution page 699)**

The three functions in Figure 6.20 perform the same operation with varying degrees of spatial locality. Rank-order the functions with respect to the spatial locality enjoyed by each. Explain how you arrived at your ranking.

6.3 The Memory Hierarchy

Sections 6.1 and 6.2 described some fundamental and enduring properties of storage technology and computer software:

Storage technology. Different storage technologies have widely different access times. Faster technologies cost more per byte than slower ones and have less capacity. The gap between CPU and main memory speed is widening.

Computer software. Well-written programs tend to exhibit good locality.

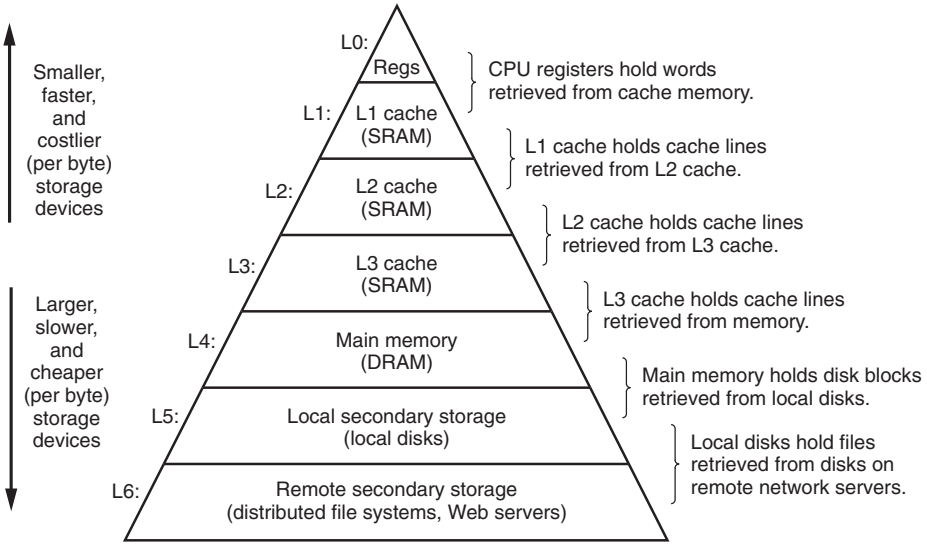


Figure 6.21 **The memory hierarchy.**

In one of the happier coincidences of computing, these fundamental properties of hardware and software complement each other beautifully. Their complementary nature suggests an approach for organizing memory systems, known as the *memory hierarchy*, that is used in all modern computer systems. Figure 6.21 shows a typical memory hierarchy.

In general, the storage devices get slower, cheaper, and larger as we move from higher to lower *levels*. At the highest level (L0) are a small number of fast CPU registers that the CPU can access in a single clock cycle. Next are one or more small to moderate-size SRAM-based cache memories that can be accessed in a few CPU clock cycles. These are followed by a large DRAM-based main memory that can be accessed in tens to hundreds of clock cycles. Next are slow but enormous local disks. Finally, some systems even include an additional level of disks on remote servers that can be accessed over a network. For example, distributed file systems such as the Andrew File System (AFS) or the Network File System (NFS) allow a program to access files that are stored on remote network-connected servers. Similarly, the World Wide Web allows programs to access remote files stored on Web servers anywhere in the world.

6.3.1 Caching in the Memory Hierarchy

In general, a *cache* (pronounced "cash") is a small, fast storage device that acts as a staging area for the data objects stored in a larger, slower device. The process of using a cache is known as *caching* (pronounced "cashing").

The central idea of a memory hierarchy is that for each k, the faster and smaller storage device at level k serves as a cache for the larger and slower storage device

Aside Other memory hierarchies

We have shown you one example of a memory hierarchy, but other combinations are possible, and indeed common. For example, many sites, including Google datacenters, back up local disks onto archival magnetic tapes. At some of these sites, human operators manually mount the tapes onto tape drives as needed. At other sites, tape robots handle this task automatically. In either case, the collection of tapes represents a level in the memory hierarchy, below the local disk level, and the same general principles apply. Tapes are cheaper per byte than disks, which allows sites to archive multiple snapshots of their local disks. The trade-off is that tapes take longer to access than disks. As another example, solid state disks are playing an increasingly important role in the memory hierarchy, bridging the gulf between DRAM and rotating disk.

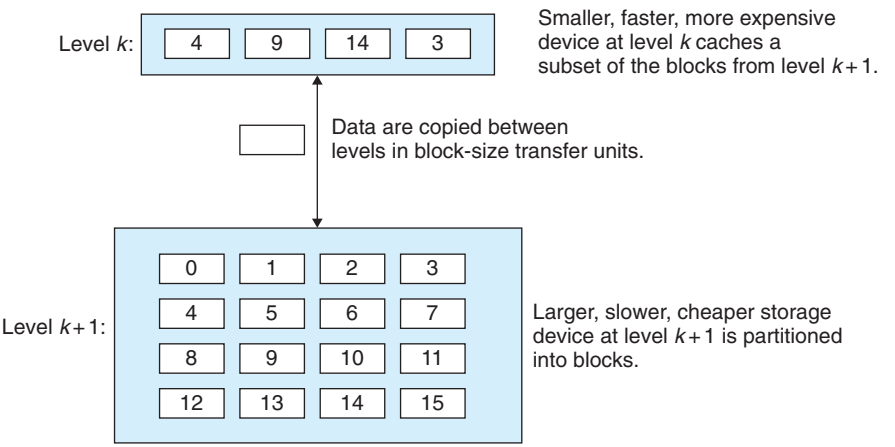


Figure 6.22 **The basic principle of caching in a memory hierarchy.**

at level $k + 1$. In other words, each level in the hierarchy caches data objects from the next lower level. For example, the local disk serves as a cache for files (such as Web pages) retrieved from remote disks over the network, the main memory serves as a cache for data on the local disks, and so on, until we get to the smallest cache of all, the set of CPU registers.

Figure 6.22 shows the general concept of caching in a memory hierarchy. The storage at level $k + 1$ is partitioned into contiguous chunks of data objects called *blocks*. Each block has a unique address or name that distinguishes it from other blocks. Blocks can be either fixed size (the usual case) or variable size (e.g., the remote HTML files stored on Web servers). For example, the level $k + 1$ storage in Figure 6.22 is partitioned into 16 fixed-size blocks, numbered 0 to 15.

Similarly, the storage at level k is partitioned into a smaller set of blocks that are the same size as the blocks at level $k + 1$. At any point in time, the cache at level k contains copies of a subset of the blocks from level $k + 1$. For example, in

Figure 6.22, the cache at level k has room for four blocks and currently contains copies of blocks 4, 9, 14, and 3.

Data are always copied back and forth between level k and level $k+1$ in block-size *transfer units*. It is important to realize that while the block size is fixed between any particular pair of adjacent levels in the hierarchy, other pairs of levels can have different block sizes. For example, in Figure 6.21, transfers between L1 and L0 typically use word-size blocks. Transfers between L2 and L1 (and L3 and L2, and L4 and L3) typically use blocks of tens of bytes. And transfers between L5 and L4 use blocks with hundreds or thousands of bytes. In general, devices lower in the hierarchy (further from the CPU) have longer access times, and thus tend to use larger block sizes in order to amortize these longer access times.

Cache Hits

When a program needs a particular data object d from level $k+1$, it first looks for d in one of the blocks currently stored at level k. If d happens to be cached at level k, then we have what is called a *cache hit*. The program reads d directly from level k, which by the nature of the memory hierarchy is faster than reading d from level $k+1$. For example, a program with good temporal locality might read a data object from block 14, resulting in a cache hit from level k.

Cache Misses

If, on the other hand, the data object d is not cached at level k, then we have what is called a *cache miss*. When there is a miss, the cache at level k fetches the block containing d from the cache at level $k+1$, possibly overwriting an existing block if the level k cache is already full.

This process of overwriting an existing block is known as *replacing* or *evicting* the block. The block that is evicted is sometimes referred to as a *victim block*. The decision about which block to replace is governed by the cache's *replacement policy*. For example, a cache with a *random replacement policy* would choose a random victim block. A cache with a *least recently used (LRU)* replacement policy would choose the block that was last accessed the furthest in the past.

After the cache at level k has fetched the block from level $k+1$, the program can read d from level k as before. For example, in Figure 6.22, reading a data object from block 12 in the level k cache would result in a cache miss because block 12 is not currently stored in the level k cache. Once it has been copied from level $k+1$ to level k, block 12 will remain there in expectation of later accesses.

Kinds of Cache Misses

It is sometimes helpful to distinguish between different kinds of cache misses. If the cache at level k is empty, then any access of any data object will miss. An empty cache is sometimes referred to as a *cold cache*, and misses of this kind are called *compulsory misses* or *cold misses*. Cold misses are important because they are often transient events that might not occur in steady state, after the cache has been *warmed up* by repeated memory accesses.

Whenever there is a miss, the cache at level k must implement some *placement policy* that determines where to place the block it has retrieved from level $k+1$. The most flexible placement policy is to allow any block from level $k+1$ to be stored in any block at level k. For caches high in the memory hierarchy (close to the CPU) that are implemented in hardware and where speed is at a premium, this policy is usually too expensive to implement because randomly placed blocks are expensive to locate.

Thus, hardware caches typically implement a simpler placement policy that restricts a particular block at level $k+1$ to a small subset (sometimes a singleton) of the blocks at level k. For example, in Figure 6.22, we might decide that a block i at level $k+1$ must be placed in block (i mod 4) at level k. For example, blocks 0, 4, 8, and 12 at level $k+1$ would map to block 0 at level k; blocks 1, 5, 9, and 13 would map to block 1; and so on. Notice that our example cache in Figure 6.22 uses this policy.

Restrictive placement policies of this kind lead to a type of miss known as a *conflict miss*, in which the cache is large enough to hold the referenced data objects, but because they map to the same cache block, the cache keeps missing. For example, in Figure 6.22, if the program requests block 0, then block 8, then block 0, then block 8, and so on, each of the references to these two blocks would miss in the cache at level k, even though this cache can hold a total of four blocks.

Programs often run as a sequence of phases (e.g., loops) where each phase accesses some reasonably constant set of cache blocks. For example, a nested loop might access the elements of the same array over and over again. This set of blocks is called the *working set* of the phase. When the size of the working set exceeds the size of the cache, the cache will experience what are known as *capacity misses*. In other words, the cache is just too small to handle this particular working set.

Cache Management

As we have noted, the essence of the memory hierarchy is that the storage device at each level is a cache for the next lower level. At each level, some form of logic must *manage* the cache. By this we mean that something has to partition the cache storage into blocks, transfer blocks between different levels, decide when there are hits and misses, and then deal with them. The logic that manages the cache can be hardware, software, or a combination of the two.

For example, the compiler manages the register file, the highest level of the cache hierarchy. It decides when to issue loads when there are misses, and determines which register to store the data in. The caches at levels L1, L2, and L3 are managed entirely by hardware logic built into the caches. In a system with virtual memory, the DRAM main memory serves as a cache for data blocks stored on disk, and is managed by a combination of operating system software and address translation hardware on the CPU. For a machine with a distributed file system such as AFS, the local disk serves as a cache that is managed by the AFS client process running on the local machine. In most cases, caches operate automatically and do not require any specific or explicit actions from the program.

Type	What cached	Where cached	Latency (cycles)	Managed by
CPU registers	4-byte or 8-byte words	On-chip CPU registers	0	Compiler
TLB	Address translations	On-chip TLB	0	Hardware MMU
L1 cache	64-byte blocks	On-chip L1 cache	4	Hardware
L2 cache	64-byte blocks	On-chip L2 cache	10	Hardware
L3 cache	64-byte blocks	On-chip L3 cache	50	Hardware
Virtual memory	4-KB pages	Main memory	200	Hardware + OS
Buffer cache	Parts of files	Main memory	200	OS
Disk cache	Disk sectors	Disk controller	100,000	Controller firmware
Network cache	Parts of files	Local disk	10,000,000	NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

Figure 6.23 The ubiquity of caching in modern computer systems. Acronyms: TLB: translation lookaside buffer; MMU: memory management unit; OS: operating system; NFS: network file system.

6.3.2 Summary of Memory Hierarchy Concepts

To summarize, memory hierarchies based on caching work because slower storage is cheaper than faster storage and because programs tend to exhibit locality:

Exploiting temporal locality. Because of temporal locality, the same data objects are likely to be reused multiple times. Once a data object has been copied into the cache on the first miss, we can expect a number of subsequent hits on that object. Since the cache is faster than the storage at the next lower level, these subsequent hits can be served much faster than the original miss.

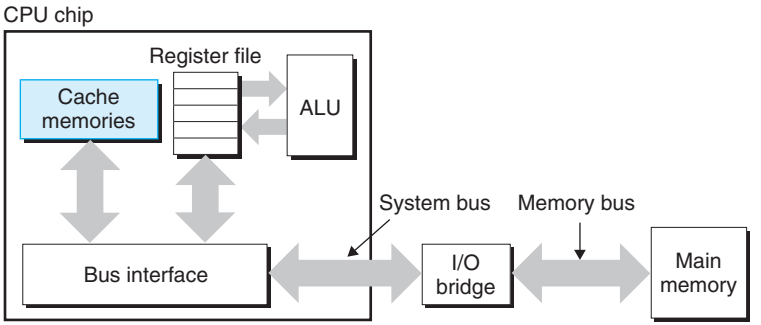
Exploiting spatial locality. Blocks usually contain multiple data objects. Because of spatial locality, we can expect that the cost of copying a block after a miss will be amortized by subsequent references to other objects within that block.

Caches are used everywhere in modern systems. As you can see from Figure 6.23, caches are used in CPU chips, operating systems, distributed file systems, and on the World Wide Web. They are built from and managed by various combinations of hardware and software. Note that there are a number of terms and acronyms in Figure 6.23 that we haven't covered yet. We include them here to demonstrate how common caches are.

6.4 Cache Memories

The memory hierarchies of early computer systems consisted of only three levels: CPU registers, main memory, and disk storage. However, because of the increasing gap between CPU and main memory, system designers were compelled to insert

Figure 6.24
Typical bus structure for cache memories.



a small SRAM *cache memory*, called an *L1 cache* (level 1 cache) between the CPU register file and main memory, as shown in Figure 6.24. The L1 cache can be accessed nearly as fast as the registers, typically in about 4 clock cycles.

As the performance gap between the CPU and main memory continued to increase, system designers responded by inserting an additional larger cache, called an *L2 cache*, between the L1 cache and main memory, that can be accessed in about 10 clock cycles. Many modern systems include an even larger cache, called an *L3 cache*, which sits between the L2 cache and main memory in the memory hierarchy and can be accessed in about 50 cycles. While there is considerable variety in the arrangements, the general principles are the same. For our discussion in the next section, we will assume a simple memory hierarchy with a single L1 cache between the CPU and main memory.

6.4.1 Generic Cache Memory Organization

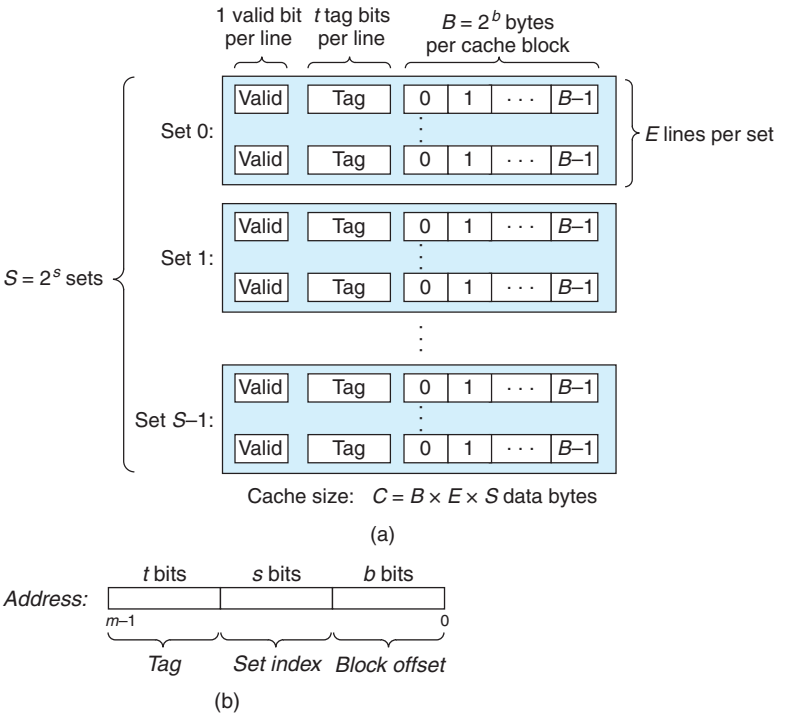
Consider a computer system where each memory address has m bits that form $M = 2^m$ unique addresses. As illustrated in Figure 6.25(a), a cache for such a machine is organized as an array of $S = 2^s$ *cache sets*. Each set consists of E *cache lines*. Each line consists of a data *block* of $B = 2^b$ bytes, a *valid bit* that indicates whether or not the line contains meaningful information, and $t = m - (b + s)$ *tag bits* (a subset of the bits from the current block's memory address) that uniquely identify the block stored in the cache line.

In general, a cache's organization can be characterized by the tuple (S, E, B, m). The size (or capacity) of a cache, C, is stated in terms of the aggregate size of all the blocks. The tag bits and valid bit are not included. Thus, $C = S \times E \times B$.

When the CPU is instructed by a load instruction to read a word from address A of main memory, it sends address A to the cache. If the cache is holding a copy of the word at address A, it sends the word immediately back to the CPU. So how does the cache know whether it contains a copy of the word at address A? The cache is organized so that it can find the requested word by simply inspecting the bits of the address, similar to a hash table with an extremely simple hash function. Here is how it works:

The parameters S and B induce a partitioning of the m address bits into the three fields shown in Figure 6.25(b). The s *set index bits* in A form an index into

Figure 6.25
General organization of cache (S, E, B, m). (a) A cache is an array of sets. Each set contains one or more lines. Each line contains a valid bit, some tag bits, and a block of data. (b) The cache organization induces a partition of the m address bits into t tag bits, s set index bits, and b block offset bits.



the array of S sets. The first set is set 0, the second set is set 1, and so on. When interpreted as an unsigned integer, the set index bits tell us which set the word must be stored in. Once we know which set the word must be contained in, the t tag bits in A tell us which line (if any) in the set contains the word. A line in the set contains the word if and only if the valid bit is set and the tag bits in the line match the tag bits in the address A. Once we have located the line identified by the tag in the set identified by the set index, then the b *block offset bits* give us the offset of the word in the B-byte data block.

As you may have noticed, descriptions of caches use a lot of symbols. Figure 6.26 summarizes these symbols for your reference.

Practice Problem 6.9 (solution page 699)

The following table gives the parameters for a number of different caches. For each cache, determine the number of cache sets (S), tag bits (t), set index bits (s), and block offset bits (b).

Cache	m	C	B	E	S	t	s	b
1.	32	1,024	4	1	_____	_____	_____	_____
2.	32	1,024	8	4	_____	_____	_____	_____
3.	32	1,024	32	32	_____	_____	_____	_____

Parameter	Description
Fundamental parameters	
$S = 2^s$	Number of sets
E	Number of lines per set
$B = 2^b$	Block size (bytes)
$m = \log_2(M)$	Number of physical (main memory) address bits
Derived quantities	
$M = 2^m$	Maximum number of unique memory addresses
$s = \log_2(S)$	Number of *set index bits*
$b = \log_2(B)$	Number of *block offset bits*
$t = m - (s + b)$	Number of *tag bits*
$C = B \times E \times S$	Cache size (bytes), not including overhead such as the valid and tag bits

Figure 6.26 **Summary of cache parameters.**

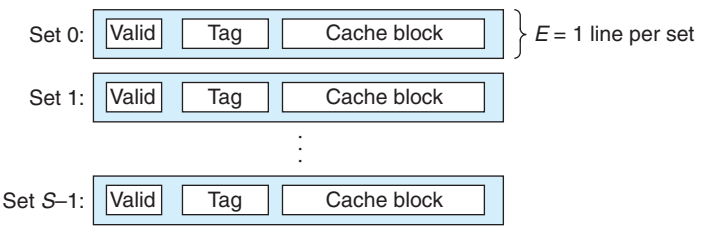


Figure 6.27 **Direct-mapped cache ($E = 1$).** There is exactly one line per set.

6.4.2 Direct-Mapped Caches

Caches are grouped into different classes based on E, the number of cache lines per set. A cache with exactly one line per set ($E = 1$) is known as a *direct-mapped* cache (see Figure 6.27). Direct-mapped caches are the simplest both to implement and to understand, so we will use them to illustrate some general concepts about how caches work.

Suppose we have a system with a CPU, a register file, an L1 cache, and a main memory. When the CPU executes an instruction that reads a memory word w, it requests the word from the L1 cache. If the L1 cache has a cached copy of w, then we have an L1 cache hit, and the cache quickly extracts w and returns it to the CPU. Otherwise, we have a cache miss, and the CPU must wait while the L1 cache requests a copy of the block containing w from the main memory. When the requested block finally arrives from memory, the L1 cache stores the block in one of its cache lines, extracts word w from the stored block, and returns it to the CPU. The process that a cache goes through of determining whether a request is a hit or a miss and then extracting the requested word consists of three steps: (1) *set selection*, (2) *line matching*, and (3) *word extraction*.

Figure 6.28
Set selection in a direct-mapped cache.

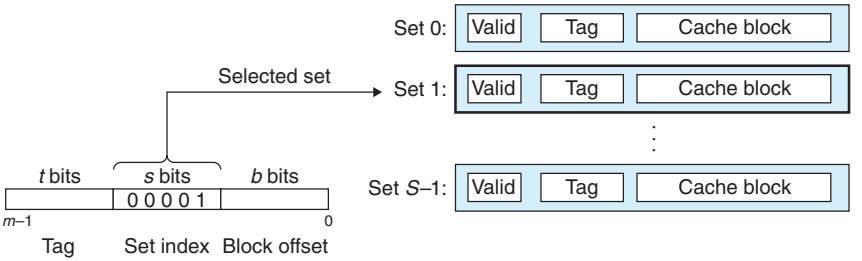
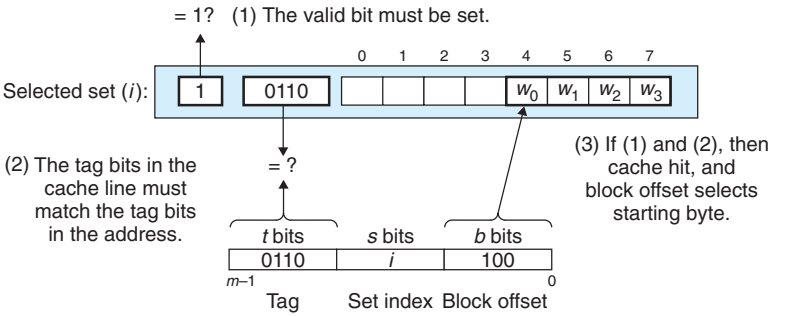


Figure 6.29
Line matching and word selection in a direct-mapped cache. Within the cache block, w_0 denotes the low-order byte of the word w, w_1 the next byte, and so on.



Set Selection in Direct-Mapped Caches

In this step, the cache extracts the s set index bits from the middle of the address for w. These bits are interpreted as an unsigned integer that corresponds to a set number. In other words, if we think of the cache as a one-dimensional array of sets, then the set index bits form an index into this array. Figure 6.28 shows how set selection works for a direct-mapped cache. In this example, the set index bits 00001_2 are interpreted as an integer index that selects set 1.

Line Matching in Direct-Mapped Caches

Now that we have selected some set i in the previous step, the next step is to determine if a copy of the word w is stored in one of the cache lines contained in set i. In a direct-mapped cache, this is easy and fast because there is exactly one line per set. A copy of w is contained in the line if and only if the valid bit is set and the tag in the cache line matches the tag in the address of w.

Figure 6.29 shows how line matching works in a direct-mapped cache. In this example, there is exactly one cache line in the selected set. The valid bit for this line is set, so we know that the bits in the tag and block are meaningful. Since the tag bits in the cache line match the tag bits in the address, we know that a copy of the word we want is indeed stored in the line. In other words, we have a cache hit. On the other hand, if either the valid bit were not set or the tags did not match, then we would have had a cache miss.

Word Selection in Direct-Mapped Caches

Once we have a hit, we know that w is somewhere in the block. This last step determines where the desired word starts in the block. As shown in Figure 6.29, the block offset bits provide us with the offset of the first byte in the desired word. Similar to our view of a cache as an array of lines, we can think of a block as an array of bytes, and the byte offset as an index into that array. In the example, the block offset bits of 100_2 indicate that the copy of w starts at byte 4 in the block. (We are assuming that words are 4 bytes long.)

Line Replacement on Misses in Direct-Mapped Caches

If the cache misses, then it needs to retrieve the requested block from the next level in the memory hierarchy and store the new block in one of the cache lines of the set indicated by the set index bits. In general, if the set is full of valid cache lines, then one of the existing lines must be evicted. For a direct-mapped cache, where each set contains exactly one line, the replacement policy is trivial: the current line is replaced by the newly fetched line.

Putting It Together: A Direct-Mapped Cache in Action

The mechanisms that a cache uses to select sets and identify lines are extremely simple. They have to be, because the hardware must perform them in a few nanoseconds. However, manipulating bits in this way can be confusing to us humans. A concrete example will help clarify the process. Suppose we have a direct-mapped cache described by

$$(S, E, B, m) = (4, 1, 2, 4)$$

In other words, the cache has four sets, one line per set, 2 bytes per block, and 4-bit addresses. We will also assume that each word is a single byte. Of course, these assumptions are totally unrealistic, but they will help us keep the example simple.

When you are first learning about caches, it can be very instructive to enumerate the entire address space and partition the bits, as we've done in Figure 6.30 for our 4-bit example. There are some interesting things to notice about this enumerated space:

- The concatenation of the tag and index bits uniquely identifies each block in memory. For example, block 0 consists of addresses 0 and 1, block 1 consists of addresses 2 and 3, block 2 consists of addresses 4 and 5, and so on.
- Since there are eight memory blocks but only four cache sets, multiple blocks map to the same cache set (i.e., they have the same set index). For example, blocks 0 and 4 both map to set 0, blocks 1 and 5 both map to set 1, and so on.
- Blocks that map to the same cache set are uniquely identified by the tag. For example, block 0 has a tag bit of 0 while block 4 has a tag bit of 1, block 1 has a tag bit of 0 while block 5 has a tag bit of 1, and so on.

Address (decimal)	Address bits: Tag bits ($t = 1$)	Index bits ($s = 2$)	Offset bits ($b = 1$)	Block number (decimal)
0	0	00	0	0
1	0	00	1	0
2	0	01	0	1
3	0	01	1	1
4	0	10	0	2
5	0	10	1	2
6	0	11	0	3
7	0	11	1	3
8	1	00	0	4
9	1	00	1	4
10	1	01	0	5
11	1	01	1	5
12	1	10	0	6
13	1	10	1	6
14	1	11	0	7
15	1	11	1	7

Figure 6.30 **4-bit address space for example direct-mapped cache.**

Let us simulate the cache in action as the CPU performs a sequence of reads. Remember that for this example we are assuming that the CPU reads 1-byte words. While this kind of manual simulation is tedious and you may be tempted to skip it, in our experience students do not really understand how caches work until they work their way through a few of them.

Initially, the cache is empty (i.e., each valid bit is 0):

Set	Valid	Tag	block[0]	block[1]
0	0			
1	0			
2	0			
3	0			

Each row in the table represents a cache line. The first column indicates the set that the line belongs to, but keep in mind that this is provided for convenience and is not really part of the cache. The next four columns represent the actual bits in each cache line. Now, let's see what happens when the CPU performs a sequence of reads:

1. **Read word at address 0.** Since the valid bit for set 0 is 0, this is a cache miss. The cache fetches block 0 from memory (or a lower-level cache) and stores the

block in set 0. Then the cache returns m[0] (the contents of memory location 0) from block[0] of the newly fetched cache line.

Set	Valid	Tag	block[0]	block[1]
0	1	0	m[0]	m[1]
1	0			
2	0			
3	0			

2. **Read word at address 1.** This is a cache hit. The cache immediately returns m[1] from block[1] of the cache line. The state of the cache does not change.

3. **Read word at address 13.** Since the cache line in set 2 is not valid, this is a cache miss. The cache loads block 6 into set 2 and returns m[13] from block[1] of the new cache line.

Set	Valid	Tag	block[0]	block[1]
0	1	0	m[0]	m[1]
1	0			
2	1	1	m[12]	m[13]
3	0			

4. **Read word at address 8.** This is a miss. The cache line in set 0 is indeed valid, but the tags do not match. The cache loads block 4 into set 0 (replacing the line that was there from the read of address 0) and returns m[8] from block[0] of the new cache line.

Set	Valid	Tag	block[0]	block[1]
0	1	1	m[8]	m[9]
1	0			
2	1	1	m[12]	m[13]
3	0			

5. **Read word at address 0.** This is another miss, due to the unfortunate fact that we just replaced block 0 during the previous reference to address 8. This kind of miss, where we have plenty of room in the cache but keep alternating references to blocks that map to the same set, is an example of a conflict miss.

Set	Valid	Tag	block[0]	block[1]
0	1	0	m[0]	m[1]
1	0			
2	1	1	m[12]	m[13]
3	0			

Conflict Misses in Direct-Mapped Caches

Conflict misses are common in real programs and can cause baffling performance problems. Conflict misses in direct-mapped caches typically occur when programs access arrays whose sizes are a power of 2. For example, consider a function that computes the dot product of two vectors:

```
1  float dotprod(float x[8], float y[8])
2  {
3      float sum = 0.0;
4      int i;
5
6      for (i = 0; i < 8; i++)
7          sum += x[i] * y[i];
8      return sum;
9  }
```

This function has good spatial locality with respect to `x` and `y`, and so we might expect it to enjoy a good number of cache hits. Unfortunately, this is not always true.

Suppose that floats are 4 bytes, that `x` is loaded into the 32 bytes of contiguous memory starting at address 0, and that `y` starts immediately after `x` at address 32. For simplicity, suppose that a block is 16 bytes (big enough to hold four floats) and that the cache consists of two sets, for a total cache size of 32 bytes. We will assume that the variable `sum` is actually stored in a CPU register and thus does not require a memory reference. Given these assumptions, each `x[i]` and `y[i]` will map to the identical cache set:

Element	Address	Set index	Element	Address	Set index
`x[0]`	0	0	`y[0]`	32	0
`x[1]`	4	0	`y[1]`	36	0
`x[2]`	8	0	`y[2]`	40	0
`x[3]`	12	0	`y[3]`	44	0
`x[4]`	16	1	`y[4]`	48	1
`x[5]`	20	1	`y[5]`	52	1
`x[6]`	24	1	`y[6]`	56	1
`x[7]`	28	1	`y[7]`	60	1

At run time, the first iteration of the loop references `x[0]`, a miss that causes the block containing `x[0]`–`x[3]` to be loaded into set 0. The next reference is to `y[0]`, another miss that causes the block containing `y[0]`–`y[3]` to be copied into set 0, overwriting the values of `x` that were copied in by the previous reference. During the next iteration, the reference to `x[1]` misses, which causes the `x[0]`–`x[3]` block to be loaded back into set 0, overwriting the `y[0]`–`y[3]` block. So now we have a conflict miss, and in fact each subsequent reference to `x` and `y` will result in a conflict miss as we thrash back and forth between blocks of `x` and `y`. The term *thrashing* describes any situation where a cache is repeatedly loading and evicting the same sets of cache blocks.

Aside Why index with the middle bits?

You may be wondering why caches use the middle bits for the set index instead of the high-order bits. There is a good reason why the middle bits are better. Figure 6.31 shows why. If the high-order bits are used as an index, then some contiguous memory blocks will map to the same cache set. For example, in the figure, the first four blocks map to the first cache set, the second four blocks map to the second set, and so on. If a program has good spatial locality and scans the elements of an array sequentially, then the cache can only hold a block-size chunk of the array at any point in time. This is an inefficient use of the cache. Contrast this with middle-bit indexing, where adjacent blocks always map to different cache sets. In this case, the cache can hold an entire C-size chunk of the array, where C is the cache size.

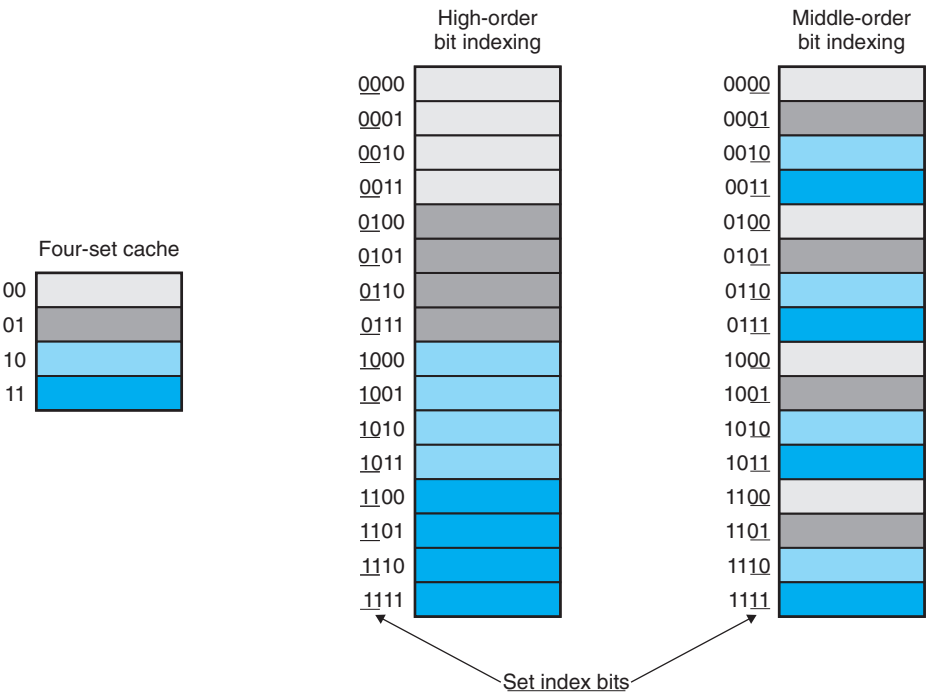


Figure 6.31 **Why caches index with the middle bits.**

The bottom line is that even though the program has good spatial locality and we have room in the cache to hold the blocks for both `x[i]` and `y[i]`, each reference results in a conflict miss because the blocks map to the same cache set. It is not unusual for this kind of thrashing to result in a slowdown by a factor of 2 or 3. Also, be aware that even though our example is extremely simple, the problem is real for larger and more realistic direct-mapped caches.

Luckily, thrashing is easy for programmers to fix once they recognize what is going on. One easy solution is to put B bytes of padding at the end of each array.

For example, instead of defining `x` to be `float x[8]`, we define it to be `float x[12]`. Assuming `y` starts immediately after `x` in memory, we have the following mapping of array elements to sets:

Element	Address	Set index	Element	Address	Set index
`x[0]`	0	0	`y[0]`	48	1
`x[1]`	4	0	`y[1]`	52	1
`x[2]`	8	0	`y[2]`	56	1
`x[3]`	12	0	`y[3]`	60	1
`x[4]`	16	1	`y[4]`	64	0
`x[5]`	20	1	`y[5]`	68	0
`x[6]`	24	1	`y[6]`	72	0
`x[7]`	28	1	`y[7]`	76	0

With the padding at the end of `x`, `x[i]` and `y[i]` now map to different sets, which eliminates the thrashing conflict misses.

Practice Problem 6.10 **(solution page 699)**

In the previous `dotprod` example, what fraction of the total references to `x` and `y` will be hits once we have padded array `x`?

Practice Problem 6.11 **(solution page 699)**

Imagine a hypothetical cache that uses the high-order s bits of an address as the set index. For such a cache, contiguous chunks of memory blocks are mapped to the same cache set.

A. How many blocks are in each of these contiguous array chunks?

B. Consider the following code that runs on a system with a cache of the form $(S, E, B, m) = (512, 1, 32, 32)$:

```
int array[4096];

for (i = 0; i < 4096; i++)
    sum += array[i];
```

What is the maximum number of array blocks that are stored in the cache at any point in time?

6.4.3 Set Associative Caches

The problem with conflict misses in direct-mapped caches stems from the constraint that each set has exactly one line (or in our terminology, $E = 1$). A *set associative cache* relaxes this constraint so that each set holds more than one cache line. A cache with $1 < E < C/B$ is often called an E-way set associative cache. We

Figure 6.32
Set associative cache ($1 < E < C/B$). In a set associative cache, each set contains more than one line. This particular example shows a two-way set associative cache.

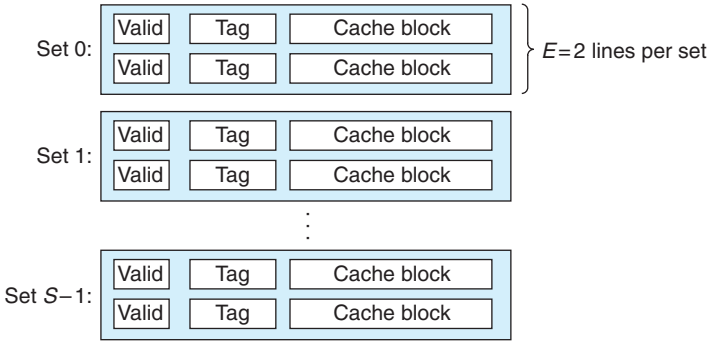
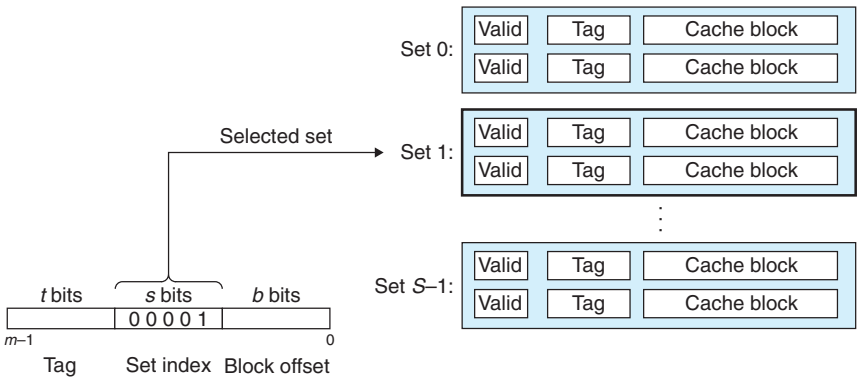


Figure 6.33
Set selection in a set associative cache.



will discuss the special case, where $E = C/B$, in the next section. Figure 6.32 shows the organization of a two-way set associative cache.

Set Selection in Set Associative Caches

Set selection is identical to a direct-mapped cache, with the set index bits identifying the set. Figure 6.33 summarizes this principle.

Line Matching and Word Selection in Set Associative Caches

Line matching is more involved in a set associative cache than in a direct-mapped cache because it must check the tags and valid bits of multiple lines in order to determine if the requested word is in the set. A conventional memory is an array of values that takes an address as input and returns the value stored at that address. An *associative memory*, on the other hand, is an array of (key, value) pairs that takes as input the key and returns a value from one of the (key, value) pairs that matches the input key. Thus, we can think of each set in a set associative cache as a small associative memory where the keys are the concatenation of the tag and valid bits, and the values are the contents of a block.

Figure 6.34
Line matching and word selection in a set associative cache.

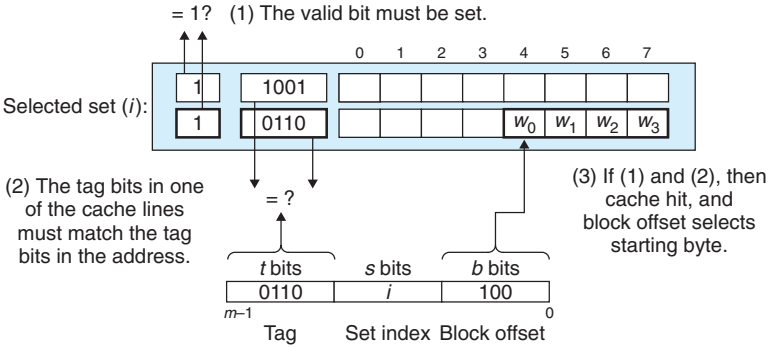


Figure 6.34 shows the basic idea of line matching in an associative cache. An important idea here is that any line in the set can contain any of the memory blocks that map to that set. So the cache must search each line in the set for a valid line whose tag matches the tag in the address. If the cache finds such a line, then we have a hit and the block offset selects a word from the block, as before.

Line Replacement on Misses in Set Associative Caches

If the word requested by the CPU is not stored in any of the lines in the set, then we have a cache miss, and the cache must fetch the block that contains the word from memory. However, once the cache has retrieved the block, which line should it replace? Of course, if there is an empty line, then it would be a good candidate. But if there are no empty lines in the set, then we must choose one of the nonempty lines and hope that the CPU does not reference the replaced line anytime soon.

It is very difficult for programmers to exploit knowledge of the cache replacement policy in their codes, so we will not go into much detail about it here. The simplest replacement policy is to choose the line to replace at random. Other more sophisticated policies draw on the principle of locality to try to minimize the probability that the replaced line will be referenced in the near future. For example, a *least frequently used (LFU)* policy will replace the line that has been referenced the fewest times over some past time window. A *least recently used (LRU)* policy will replace the line that was last accessed the furthest in the past. All of these policies require additional time and hardware. But as we move further down the memory hierarchy, away from the CPU, the cost of a miss becomes more expensive and it becomes more worthwhile to minimize misses with good replacement policies.

6.4.4 Fully Associative Caches

A *fully associative cache* consists of a single set (i.e., $E = C/B$) that contains all of the cache lines. Figure 6.35 shows the basic organization.

Figure 6.35
Fully associative cache ($E = C/B$). In a fully associative cache, a single set contains all of the lines.

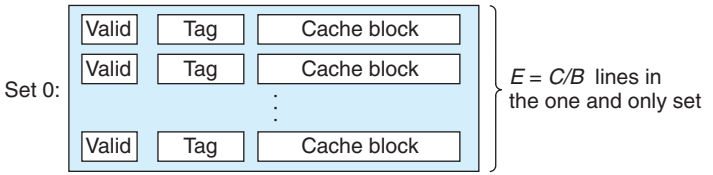


Figure 6.36
Set selection in a fully associative cache. Notice that there are no set index bits.

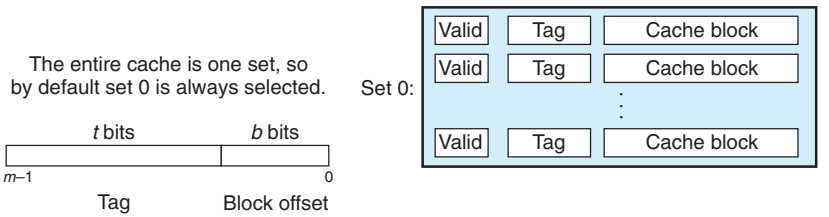
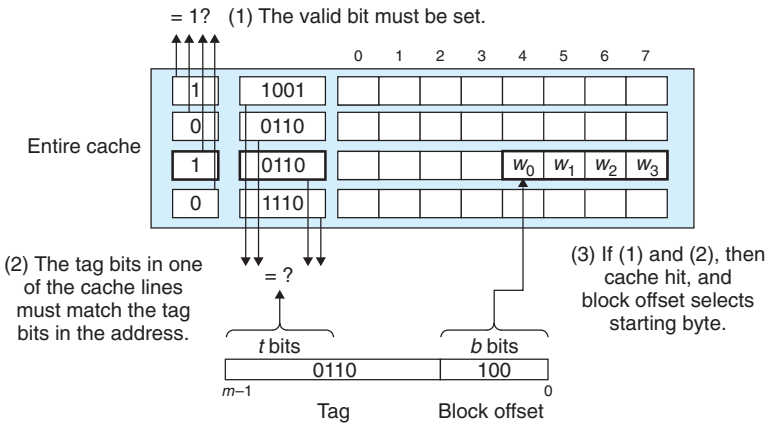


Figure 6.37
Line matching and word selection in a fully associative cache.



Set Selection in Fully Associative Caches

Set selection in a fully associative cache is trivial because there is only one set, summarized in Figure 6.36. Notice that there are no set index bits in the address, which is partitioned into only a tag and a block offset.

Line Matching and Word Selection in Fully Associative Caches

Line matching and word selection in a fully associative cache work the same as with a set associative cache, as we show in Figure 6.37. The difference is mainly a question of scale.

Because the cache circuitry must search for many matching tags in parallel, it is difficult and expensive to build an associative cache that is both large and fast. As a result, fully associative caches are only appropriate for small caches, such

as the translation lookaside buffers (TLBs) in virtual memory systems that cache page table entries (Section 9.6.2).

Practice Problem 6.12 (solution page 699)

The problems that follow will help reinforce your understanding of how caches work. Assume the following:

- The memory is byte addressable.
- Memory accesses are to 1-byte words (not to 4-byte words).
- Addresses are 13 bits wide.
- The cache is two-way set associative ($E = 2$), with a 4-byte block size ($B = 4$) and eight sets ($S = 8$).

The contents of the cache are as follows, with all numbers given in hexadecimal notation.

2-way set associative cache

	Line 0						Line 1					
Set index	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3
0	09	1	86	30	3F	10	00	0	—	—	—	—
1	45	1	60	4F	E0	23	38	1	00	BC	0B	37
2	EB	0	—	—	—	—	0B	0	—	—	—	—
3	06	0	—	—	—	—	32	1	12	08	7B	AD
4	C7	1	06	78	07	C5	05	1	40	67	C2	3B
5	71	1	0B	DE	18	4B	6E	0	—	—	—	—
6	91	1	A0	B7	26	2D	F0	0	—	—	—	—
7	46	0	—	—	—	—	DE	1	12	C0	88	37

The following figure shows the format of an address (1 bit per box). Indicate (by labeling the diagram) the fields that would be used to determine the following:

CO. The cache block offset

CI. The cache set index

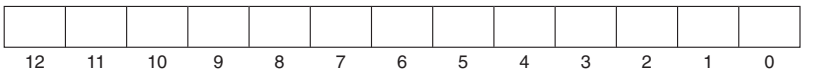
CT. The cache tag

Practice Problem 6.13 (solution page 700)

Suppose a program running on the machine in Problem 6.12 references the 1-byte word at address `0x0D53`. Indicate the cache entry accessed and the cache byte

value returned in hexadecimal notation. Indicate whether a cache miss occurs. If there is a cache miss, enter "—" for "Cache byte returned."

A. Address format (1 bit per box):



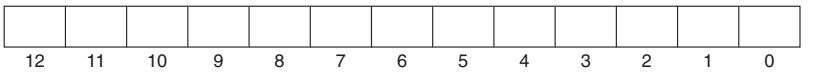
B. Memory reference:

Parameter	Value
Cache block offset (CO)	0x_____
Cache set index (CI)	0x_____
Cache tag (CT)	0x_____
Cache hit? (Y/N)	_____
Cache byte returned	0x_____

Practice Problem 6.14 (solution page 700)

Repeat Problem 6.13 for memory address `0x0CB4`.

A. Address format (1 bit per box):



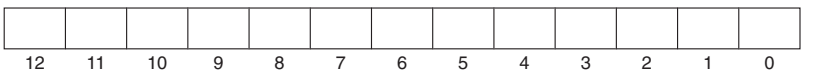
B. Memory reference:

Parameter	Value
Cache block offset (CO)	0x_____
Cache set index (CI)	0x_____
Cache tag (CT)	0x_____
Cache hit? (Y/N)	_____
Cache byte returned	0x_____

Practice Problem 6.15 (solution page 700)

Repeat Problem 6.13 for memory address `0x0A31`.

A. Address format (1 bit per box):



B. Memory reference:

Parameter	Value
Cache block offset (CO)	0x_____
Cache set index (CI)	0x_____
Cache tag (CT)	0x_____
Cache hit? (Y/N)	_____
Cache byte returned	0x_____

Practice Problem 6.16 **(solution page 701)**

For the cache in Problem 6.12, list all of the hexadecimal memory addresses that will hit in set 3.

6.4.5 Issues with Writes

As we have seen, the operation of a cache with respect to reads is straightforward. First, look for a copy of the desired word w in the cache. If there is a hit, return w immediately. If there is a miss, fetch the block that contains w from the next lower level of the memory hierarchy, store the block in some cache line (possibly evicting a valid line), and then return w.

The situation for writes is a little more complicated. Suppose we write a word w that is already cached (a *write hit*). After the cache updates its copy of w, what does it do about updating the copy of w in the next lower level of the hierarchy? The simplest approach, known as *write-through*, is to immediately write w's cache block to the next lower level. While simple, write-through has the disadvantage of causing bus traffic with every write. Another approach, known as *write-back*, defers the update as long as possible by writing the updated block to the next lower level only when it is evicted from the cache by the replacement algorithm. Because of locality, write-back can significantly reduce the amount of bus traffic, but it has the disadvantage of additional complexity. The cache must maintain an additional *dirty bit* for each cache line that indicates whether or not the cache block has been modified.

Another issue is how to deal with write misses. One approach, known as *write-allocate*, loads the corresponding block from the next lower level into the cache and then updates the cache block. Write-allocate tries to exploit spatial locality of writes, but it has the disadvantage that every miss results in a block transfer from the next lower level to the cache. The alternative, known as *no-write-allocate*, bypasses the cache and writes the word directly to the next lower level. Write-through caches are typically no-write-allocate. Write-back caches are typically write-allocate.

Optimizing caches for writes is a subtle and difficult issue, and we are only scratching the surface here. The details vary from system to system and are often proprietary and poorly documented. To the programmer trying to write reason-

ably cache-friendly programs, we suggest adopting a mental model that assumes write-back, write-allocate caches. There are several reasons for this suggestion: As a rule, caches at lower levels of the memory hierarchy are more likely to use write-back instead of write-through because of the larger transfer times. For example, virtual memory systems (which use main memory as a cache for the blocks stored on disk) use write-back exclusively. But as logic densities increase, the increased complexity of write-back is becoming less of an impediment and we are seeing write-back caches at all levels of modern systems. So this assumption matches current trends. Another reason for assuming a write-back, write-allocate approach is that it is symmetric to the way reads are handled, in that write-back write-allocate tries to exploit locality. Thus, we can develop our programs at a high level to exhibit good spatial and temporal locality rather than trying to optimize for a particular memory system.

6.4.6 Anatomy of a Real Cache Hierarchy

So far, we have assumed that caches hold only program data. But, in fact, caches can hold instructions as well as data. A cache that holds instructions only is called an *i-cache*. A cache that holds program data only is called a *d-cache*. A cache that holds both instructions and data is known as a *unified cache*. Modern processors include separate i-caches and d-caches. There are a number of reasons for this. With two separate caches, the processor can read an instruction word and a data word at the same time. I-caches are typically read-only, and thus simpler. The two caches are often optimized to different access patterns and can have different block sizes, associativities, and capacities. Also, having separate caches ensures that data accesses do not create conflict misses with instruction accesses, and vice versa, at the cost of a potential increase in capacity misses.

Figure 6.38 shows the cache hierarchy for the Intel Core i7 processor. Each CPU chip has four cores. Each core has its own private L1 i-cache, L1 d-cache, and L2 unified cache. All of the cores share an on-chip L3 unified cache. An interesting feature of this hierarchy is that all of the SRAM cache memories are contained in the CPU chip.

Figure 6.39 summarizes the basic characteristics of the Core i7 caches.

6.4.7 Performance Impact of Cache Parameters

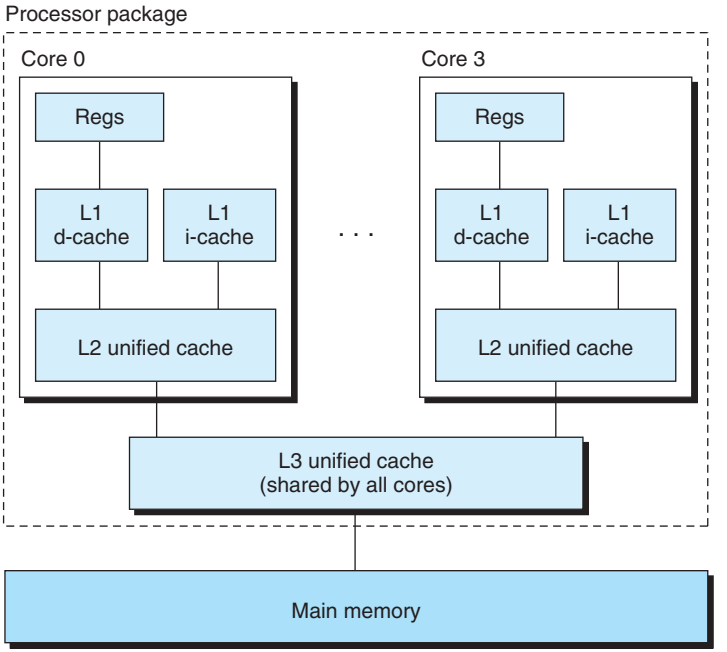
Cache performance is evaluated with a number of metrics:

Miss rate. The fraction of memory references during the execution of a program, or a part of a program, that miss. It is computed as # *misses*/ # *references*.

Hit rate. The fraction of memory references that hit. It is computed as $1 - \textit{miss rate}$.

Hit time. The time to deliver a word in the cache to the CPU, including the time for set selection, line identification, and word selection. Hit time is on the order of several clock cycles for L1 caches.

Figure 6.38
Intel Core i7 cache hierarchy.



Cache type	Access time (cycles)	Cache size (C)	Assoc. (E)	Block size (B)	Sets (S)
L1 i-cache	4	32 KB	8	64 B	64
L1 d-cache	4	32 KB	8	64 B	64
L2 unified cache	10	256 KB	8	64 B	512
L3 unified cache	40–75	8 MB	16	64 B	8,192

Figure 6.39 **Characteristics of the Intel Core i7 cache hierarchy.**

Miss penalty. Any additional time required because of a miss. The penalty for L1 misses served from L2 is on the order of 10 cycles; from L3, 50 cycles; and from main memory, 200 cycles.

Optimizing the cost and performance trade-offs of cache memories is a subtle exercise that requires extensive simulation on realistic benchmark codes and thus is beyond our scope. However, it is possible to identify some of the qualitative trade-offs.

Impact of Cache Size

On the one hand, a larger cache will tend to increase the hit rate. On the other hand, it is always harder to make large memories run faster. As a result, larger caches tend to increase the hit time. This explains why an L1 cache is smaller than an L2 cache, and an L2 cache is smaller than an L3 cache.

Impact of Block Size

Large blocks are a mixed blessing. On the one hand, larger blocks can help increase the hit rate by exploiting any spatial locality that might exist in a program. However, for a given cache size, larger blocks imply a smaller number of cache lines, which can hurt the hit rate in programs with more temporal locality than spatial locality. Larger blocks also have a negative impact on the miss penalty, since larger blocks cause larger transfer times. Modern systems such as the Core i7 compromise with cache blocks that contain 64 bytes.

Impact of Associativity

The issue here is the impact of the choice of the parameter E, the number of cache lines per set. The advantage of higher associativity (i.e., larger values of E) is that it decreases the vulnerability of the cache to thrashing due to conflict misses. However, higher associativity comes at a significant cost. Higher associativity is expensive to implement and hard to make fast. It requires more tag bits per line, additional LRU state bits per line, and additional control logic. Higher associativity can increase hit time, because of the increased complexity, and it can also increase the miss penalty because of the increased complexity of choosing a victim line.

The choice of associativity ultimately boils down to a trade-off between the hit time and the miss penalty. Traditionally, high-performance systems that pushed the clock rates would opt for smaller associativity for L1 caches (where the miss penalty is only a few cycles) and a higher degree of associativity for the lower levels, where the miss penalty is higher. For example, in Intel Core i7 systems, the L1 and L2 caches are 8-way associative, and the L3 cache is 16-way.

Impact of Write Strategy

Write-through caches are simpler to implement and can use a *write buffer* that works independently of the cache to update memory. Furthermore, read misses are less expensive because they do not trigger a memory write. On the other hand, write-back caches result in fewer transfers, which allows more bandwidth to memory for I/O devices that perform DMA. Further, reducing the number of transfers becomes increasingly important as we move down the hierarchy and the transfer times increase. In general, caches further down the hierarchy are more likely to use write-back than write-through.

6.5 Writing Cache-Friendly Code

In Section 6.2, we introduced the idea of locality and talked in qualitative terms about what constitutes good locality. Now that we understand how cache memories work, we can be more precise. Programs with better locality will tend to have lower miss rates, and programs with lower miss rates will tend to run faster than programs with higher miss rates. Thus, good programmers should always try to

Aside Cache lines, sets, and blocks: What's the difference?

It is easy to confuse the distinction between cache lines, sets, and blocks. Let's review these ideas and make sure they are clear:

- A *block* is a fixed-size packet of information that moves back and forth between a cache and main memory (or a lower-level cache).
- A *line* is a container in a cache that stores a block, as well as other information such as the valid bit and the tag bits.
- A *set* is a collection of one or more lines. Sets in direct-mapped caches consist of a single line. Sets in set associative and fully associative caches consist of multiple lines.

In direct-mapped caches, sets and lines are indeed equivalent. However, in associative caches, sets and lines are very different things and the terms cannot be used interchangeably.

Since a line always stores a single block, the terms "line" and "block" are often used interchangeably. For example, systems professionals usually refer to the "line size" of a cache, when what they really mean is the block size. This usage is very common and shouldn't cause any confusion as long as you understand the distinction between blocks and lines.

write code that is *cache friendly*, in the sense that it has good locality. Here is the basic approach we use to try to ensure that our code is cache friendly.

1. *Make the common case go fast.* Programs often spend most of their time in a few core functions. These functions often spend most of their time in a few loops. So focus on the inner loops of the core functions and ignore the rest.
2. *Minimize the number of cache misses in each inner loop.* All other things being equal, such as the total number of loads and stores, loops with better miss rates will run faster.

To see how this works in practice, consider the `sumvec` function from Section 6.2:

```
1  int sumvec(int v[N])
2  {
3      int i, sum = 0;
4
5      for (i = 0; i < N; i++)
6          sum += v[i];
7      return sum;
8  }
```

Is this function cache friendly? First, notice that there is good temporal locality in the loop body with respect to the local variables `i` and `sum`. In fact, because these are local variables, any reasonable optimizing compiler will cache them in the register file, the highest level of the memory hierarchy. Now consider the stride-1 references to vector `v`. In general, if a cache has a block size of B bytes, then a

stride-k reference pattern (where k is expressed in words) results in an average of min (1, (*word size* $\times$ k)/B) misses per loop iteration. This is minimized for $k = 1$, so the stride-1 references to `v` are indeed cache friendly. For example, suppose that `v` is block aligned, words are 4 bytes, cache blocks are 4 words, and the cache is initially empty (a cold cache). Then, regardless of the cache organization, the references to `v` will result in the following pattern of hits and misses:

`v[i]`	$i = 0$	$i = 1$	$i = 2$	$i = 3$	$i = 4$	$i = 5$	$i = 6$	$i = 7$
Access order, [h]it or [m]iss	1 **[m]**	2 [h]	3 [h]	4 [h]	5 **[m]**	6 [h]	7 [h]	8 [h]

In this example, the reference to `v[0]` misses and the corresponding block, which contains `v[0]`–`v[3]`, is loaded into the cache from memory. Thus, the next three references are all hits. The reference to `v[4]` causes another miss as a new block is loaded into the cache, the next three references are hits, and so on. In general, three out of four references will hit, which is the best we can do in this case with a cold cache.

To summarize, our simple `sumvec` example illustrates two important points about writing cache-friendly code:

- Repeated references to local variables are good because the compiler can cache them in the register file (temporal locality).
- Stride-1 reference patterns are good because caches at all levels of the memory hierarchy store data as contiguous blocks (spatial locality).

Spatial locality is especially important in programs that operate on multi-dimensional arrays. For example, consider the `sumarrayrows` function from Section 6.2, which sums the elements of a two-dimensional array in row-major order:

```
1   int sumarrayrows(int a[M][N])
2   {
3       int i, j, sum = 0;
4
5       for (i = 0; i < M; i++)
6           for (j = 0; j < N; j++)
7               sum += a[i][j];
8       return sum;
9   }
```

Since C stores arrays in row-major order, the inner loop of this function has the same desirable stride-1 access pattern as `sumvec`. For example, suppose we make the same assumptions about the cache as for `sumvec`. Then the references to the array `a` will result in the following pattern of hits and misses:

`a[i][j]`	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = 5$	$j = 6$	$j = 7$
$i = 0$	1 **[m]**	2 [h]	3 [h]	4 [h]	5 **[m]**	6 [h]	7 [h]	8 [h]
$i = 1$	9 **[m]**	10 [h]	11 [h]	12 [h]	13 **[m]**	14 [h]	15 [h]	16 [h]
$i = 2$	17 **[m]**	18 [h]	19 [h]	20 [h]	21 **[m]**	22 [h]	23 [h]	24 [h]
$i = 3$	25 **[m]**	26 [h]	27 [h]	28 [h]	29 **[m]**	30 [h]	31 [h]	32 [h]

But consider what happens if we make the seemingly innocuous change of permuting the loops:

```
1  int sumarraycols(int a[M][N])
2  {
3      int i, j, sum = 0;
4
5      for (j = 0; j < N; j++)
6          for (i = 0; i < M; i++)
7              sum += a[i][j];
8      return sum;
9  }
```

In this case, we are scanning the array column by column instead of row by row. If we are lucky and the entire array fits in the cache, then we will enjoy the same miss rate of 1/4. However, if the array is larger than the cache (the more likely case), then each and every access of `a[i][j]` will miss!

`a[i][j]`	$j=0$	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$	$j=6$	$j=7$
$i=0$	1 **[m]**	5 **[m]**	9 **[m]**	13 **[m]**	17 **[m]**	21 **[m]**	25 **[m]**	29 **[m]**
$i=1$	2 **[m]**	6 **[m]**	10 **[m]**	14 **[m]**	18 **[m]**	22 **[m]**	26 **[m]**	30 **[m]**
$i=2$	3 **[m]**	7 **[m]**	11 **[m]**	15 **[m]**	19 **[m]**	23 **[m]**	27 **[m]**	31 **[m]**
$i=3$	4 **[m]**	8 **[m]**	12 **[m]**	16 **[m]**	20 **[m]**	24 **[m]**	28 **[m]**	32 **[m]**

Higher miss rates can have a significant impact on running time. For example, on our desktop machine, `sumarrayrows` runs 25 times faster than `sumarraycols` for large array sizes. To summarize, programmers should be aware of locality in their programs and try to write programs that exploit it.

Practice Problem 6.17 (solution page 701)

Transposing the rows and columns of a matrix is an important problem in signal processing and scientific computing applications. It is also interesting from a locality point of view because its reference pattern is both row-wise and column-wise. For example, consider the following transpose routine:

```
1  typedef int array[2][2];
2
3  void transpose1(array dst, array src)
4  {
5      int i, j;
6
7      for (i = 0; i < 2; i++) {
8          for (j = 0; j < 2; j++) {
9              dst[j][i] = src[i][j];
10         }
11     }
12 }
```

Assume this code runs on a machine with the following properties:

- `sizeof(int)` = 4.
- The `src` array starts at address 0 and the `dst` array starts at address 16 (decimal).
- There is a single L1 data cache that is direct-mapped, write-through, and write-allocate, with a block size of 8 bytes.
- The cache has a total size of 16 data bytes and the cache is initially empty.
- Accesses to the `src` and `dst` arrays are the only sources of read and write misses, respectively.

A. For each `row` and `col`, indicate whether the access to `src[row][col]` and `dst[row][col]` is a hit (h) or a miss (m). For example, reading `src[0][0]` is a miss and writing `dst[0][0]` is also a miss.

	`dst` array			`src` array	
	Col. 0	Col. 1		Col. 0	Col. 1
Row 0	m	_____	Row0	m	_____
Row 1	_____	_____	Row 1	_____	_____

B. Repeat the problem for a cache with 32 data bytes.

Practice Problem 6.18 (solution page 702)

The heart of the recent hit game *SimAquarium* is a tight loop that calculates the average position of 512 algae. You are evaluating its cache performance on a machine with a 2,048-byte direct-mapped data cache with 32-byte blocks ($B = 32$). You are given the following definitions:

```
1  struct algae_position {
2      int x;
3      int y;
4  };
5
6  struct algae_position grid[32][32];
7  int total_x = 0, total_y = 0;
8  int i, j;
```

You should also assume the following:

- `sizeof(int)` = 4.
- `grid` begins at memory address 0.
- The cache is initially empty.
- The only memory accesses are to the entries of the array `grid`. Variables `i`, `j`, `total_x`, and `total_y` are stored in registers.

Determine the cache performance for the following code:

```
1       for (i = 31; i >= 0; i--) {
2           for (j = 31; j >= 0; j--) {
3               total_x += grid[i][j].x;
4           }
5       }
6
7       for (i = 31; i >= 0; i--) {
8           for (j = 31; j >= 0; j--) {
9               total_y += grid[i][j].y;
10          }
11      }
```

A. What is the total number of reads?

B. What is the total number of reads that miss in the cache?

C. What is the miss rate?

Practice Problem 6.19 (solution page 702)

Given the assumptions of Practice Problem 6.18, determine the cache performance of the following code:

```
1       for (i = 31; i >= 0; i--){
2           for (j = 31; j >= 0; j--) {
3               total_x += grid[j][i].x;
4               total_y += grid[j][i].y;
5           }
6       }
```

A. What is the total number of reads?

B. What is the total number of reads that hit in the cache?

C. What is the hit rate?

D. What would the miss hit be if the cache were twice as big?

Practice Problem 6.20 (solution page 702)

Given the assumptions of Practice Problem 6.18, determine the cache performance of the following code:

```
1       for (i = 31; i >= 0; i--){
2           for (j = 31; j >= 0; j--) {
3               total_x += grid[i][j].x;
4               total_y += grid[i][j].y;
5           }
6       }
```

A. What is the total number of reads?

B. What is the total number of reads that hit in the cache?

C. What is the hit rate?

D. What would the hit rate be if the cache were twice as big?

6.6 Putting It Together: The Impact of Caches on Program Performance

This section wraps up our discussion of the memory hierarchy by studying the impact that caches have on the performance of programs running on real machines.

6.6.1 The Memory Mountain

The rate that a program reads data from the memory system is called the *read throughput*, or sometimes the *read bandwidth*. If a program reads n bytes over a period of s seconds, then the read throughput over that period is n/s, typically expressed in units of megabytes per second (MB/s).

If we were to write a program that issued a sequence of read requests from a tight program loop, then the measured read throughput would give us some insight into the performance of the memory system for that particular sequence of reads. Figure 6.40 shows a pair of functions that measure the read throughput for a particular read sequence.

The `test` function generates the read sequence by scanning the first `elems` elements of an array with a stride of `stride`. To increase the available parallelism in the inner loop, it uses 4×4 unrolling (Section 5.9). The `run` function is a wrapper that calls the `test` function and returns the measured read throughput. The call to the `test` function in line 37 warms the cache. The `fcyc2` function in line 38 calls the `test` function with arguments `elems` and estimates the running time of the `test` function in CPU cycles. Notice that the `size` argument to the `run` function is in units of bytes, while the corresponding `elems` argument to the `test` function is in units of array elements. Also, notice that line 39 computes MB/s as 10^6 bytes/s, as opposed to 2^{20} bytes/s.

The `size` and `stride` arguments to the `run` function allow us to control the degree of temporal and spatial locality in the resulting read sequence. Smaller values of `size` result in a smaller working set size, and thus better temporal locality. Smaller values of `stride` result in better spatial locality. If we call the `run` function repeatedly with different values of `size` and `stride`, then we can recover a fascinating two-dimensional function of read throughput versus temporal and spatial locality. This function is called a *memory mountain* [112].

Every computer has a unique memory mountain that characterizes the capabilities of its memory system. For example, Figure 6.41 shows the memory mountain for an Intel Core i7 Haswell system. In this example, the `size` varies from 16 KB to 128 MB, and the `stride` varies from 1 to 12 elements, where each element is an 8-byte `long int`.

code/mem/mountain/mountain.c

```
long data[MAXELEMS];      /* The global array we'll be traversing */

/* test - Iterate over first "elems" elements of array "data" with
 *        stride of "stride", using 4 x 4 loop unrolling.
 */
int test(int elems, int stride)
{
    long i, sx2 = stride*2, sx3 = stride*3, sx4 = stride*4;
    long acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0;
    long length = elems;
    long limit = length - sx4;

    /* Combine 4 elements at a time */
    for (i = 0; i < limit; i += sx4) {
        acc0 = acc0 + data[i];
        acc1 = acc1 + data[i+stride];
        acc2 = acc2 + data[i+sx2];
        acc3 = acc3 + data[i+sx3];
    }

    /* Finish any remaining elements */
    for (; i < length; i++) {
        acc0 = acc0 + data[i];
    }
    return ((acc0 + acc1) + (acc2 + acc3));
}

/* run - Run test(elems, stride) and return read throughput (MB/s).
 *       "size" is in bytes, "stride" is in array elements, and Mhz is
 *       CPU clock frequency in Mhz.
 */
double run(int size, int stride, double Mhz)
{
    double cycles;
    int elems = size / sizeof(double);

    test(elems, stride);                    /* Warm up the cache */
    cycles = fcyc2(test, elems, stride, 0);  /* Call test(elems,stride) */
    return (size / stride) / (cycles / Mhz); /* Convert cycles to MB/s */
}
```

code/mem/mountain/mountain.c

Figure 6.40 Functions that measure and compute read throughput. We can generate a memory mountain for a particular computer by calling the `run` function with different values of `size` (which corresponds to temporal locality) and `stride` (which corresponds to spatial locality).

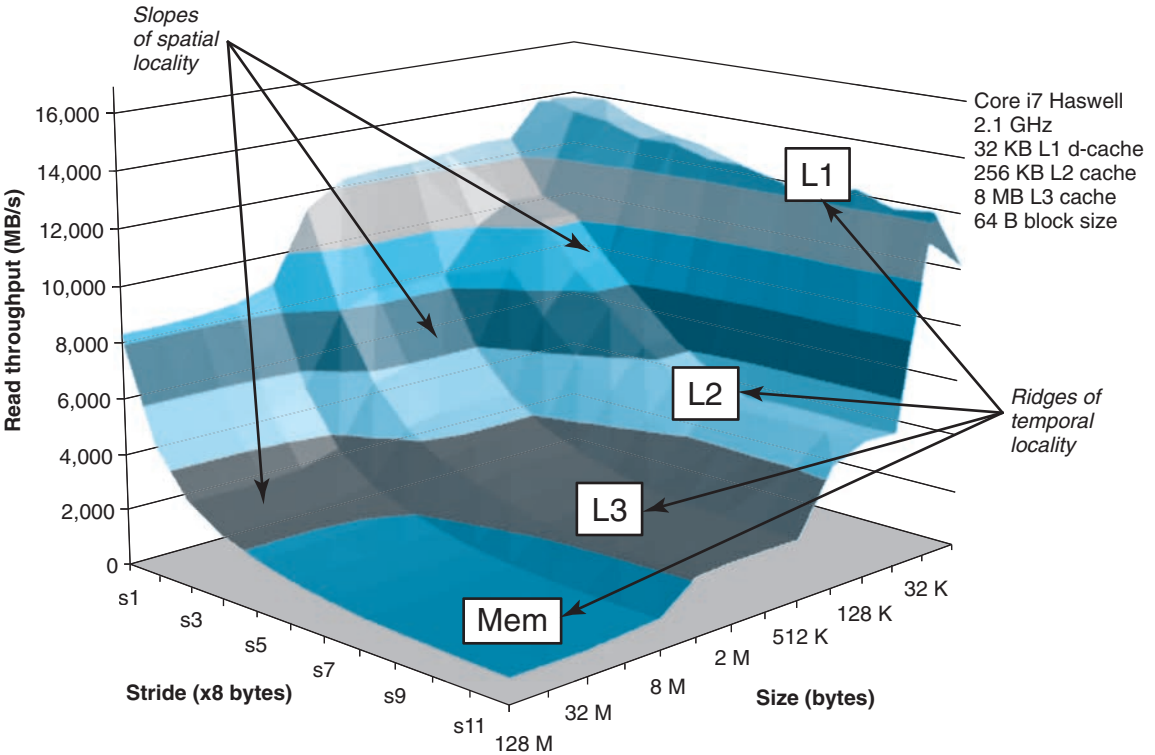


Figure 6.41 **A memory mountain.** Shows read throughput as a function of temporal and spatial locality.

The geography of the Core i7 mountain reveals a rich structure. Perpendicular to the `size` axis are four *ridges* that correspond to the regions of temporal locality where the working set fits entirely in the L1 cache, L2 cache, L3 cache, and main memory, respectively. Notice that there is more than an order of magnitude difference between the highest peak of the L1 ridge, where the CPU reads at a rate of over 14 GB/s, and the lowest point of the main memory ridge, where the CPU reads at a rate of 900 MB/s.

On each of the L2, L3, and main memory ridges, there is a slope of spatial locality that falls downhill as the stride increases and spatial locality decreases. Notice that even when the working set is too large to fit in any of the caches, the highest point on the main memory ridge is a factor of 8 higher than its lowest point. So even when a program has poor temporal locality, spatial locality can still come to the rescue and make a significant difference.

There is a particularly interesting flat ridge line that extends perpendicular to the stride axis for a stride of 1, where the read throughput is a relatively flat 12 GB/s, even though the working set exceeds the capacities of L1 and L2. This is apparently due to a hardware *prefetching* mechanism in the Core i7 memory system that automatically identifies sequential stride-1 reference patterns and attempts to fetch those blocks into the cache before they are accessed. While the

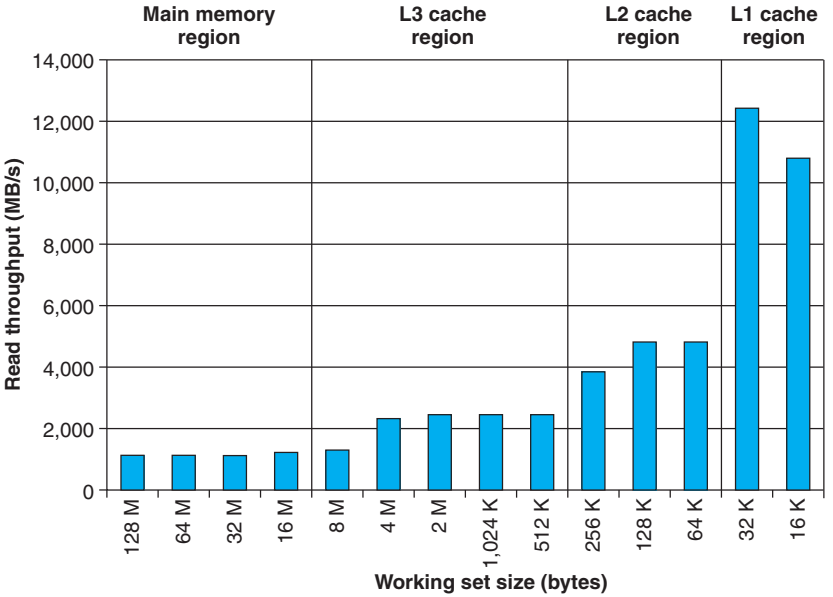


Figure 6.42 Ridges of temporal locality in the memory mountain. The graph shows a slice through Figure 6.41 with `stride` = 8.

details of the particular prefetching algorithm are not documented, it is clear from the memory mountain that the algorithm works best for small strides—yet another reason to favor sequential stride-1 accesses in your code.

If we take a slice through the mountain, holding the stride constant as in Figure 6.42, we can see the impact of cache size and temporal locality on performance. For sizes up to 32 KB, the working set fits entirely in the L1 d-cache, and thus reads are served from L1 at throughput of about 12 GB/s. For sizes up to 256 KB, the working set fits entirely in the unified L2 cache, and for sizes up to 8 MB, the working set fits entirely in the unified L3 cache. Larger working set sizes are served primarily from main memory.

The dips in read throughputs at the leftmost edges of the L2 and L3 cache regions—where the working set sizes of 256 KB and 8 MB are equal to their respective cache sizes—are interesting. It is not entirely clear why these dips occur. The only way to be sure is to perform a detailed cache simulation, but it is likely that the drops are caused by conflicts with other code and data lines.

Slicing through the memory mountain in the opposite direction, holding the working set size constant, gives us some insight into the impact of spatial locality on the read throughput. For example, Figure 6.43 shows the slice for a fixed working set size of 4 MB. This slice cuts along the L3 ridge in Figure 6.41, where the working set fits entirely in the L3 cache but is too large for the L2 cache.

Notice how the read throughput decreases steadily as the stride increases from one to eight words. In this region of the mountain, a read miss in L2 causes a block to be transferred from L3 to L2. This is followed by some number of hits

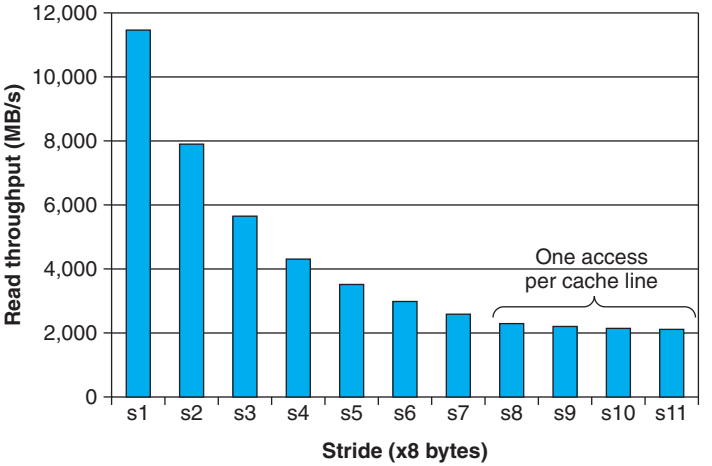


Figure 6.43 A slope of spatial locality. The graph shows a slice through Figure 6.41 with `size` = 4 MB.

on the block in L2, depending on the stride. As the stride increases, the ratio of L2 misses to L2 hits increases. Since misses are served more slowly than hits, the read throughput decreases. Once the stride reaches eight 8-byte words, which on this system equals the block size of 64 bytes, every read request misses in L2 and must be served from L3. Thus, the read throughput for strides of at least eight is a constant rate determined by the rate that cache blocks can be transferred from L3 into L2.

To summarize our discussion of the memory mountain, the performance of the memory system is not characterized by a single number. Instead, it is a mountain of temporal and spatial locality whose elevations can vary by over an order of magnitude. Wise programmers try to structure their programs so that they run in the peaks instead of the valleys. The aim is to exploit temporal locality so that heavily used words are fetched from the L1 cache, and to exploit spatial locality so that as many words as possible are accessed from a single L1 cache line.

Practice Problem 6.21 **(solution page 702)**

Use the memory mountain in Figure 6.41 to estimate the time, in CPU cycles, to read a 16-byte word from the L1 d-cache.

6.6.2 Rearranging Loops to Increase Spatial Locality

Consider the problem of multiplying a pair of $n \times n$ matrices: $C = AB$. For example, if $n = 2$, then

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

where

$$c_{11} = a_{11}b_{11} + a_{12}b_{21}$$
$$c_{12} = a_{11}b_{12} + a_{12}b_{22}$$
$$c_{21} = a_{21}b_{11} + a_{22}b_{21}$$
$$c_{22} = a_{21}b_{12} + a_{22}b_{22}$$

A matrix multiply function is usually implemented using three nested loops, which are identified by their indices i, j, and k. If we permute the loops and make some other minor code changes, we can create the six functionally equivalent versions of matrix multiply shown in Figure 6.44. Each version is uniquely identified by the ordering of its loops.

At a high level, the six versions are quite similar. If addition is associative, then each version computes an identical result.[1] Each version performs $O(n^3)$ total operations and an identical number of adds and multiplies. Each of the n^2 elements of A and B is read n times. Each of the n^2 elements of C is computed by summing n values. However, if we analyze the behavior of the innermost loop iterations, we find that there are differences in the number of accesses and the locality. For the purposes of this analysis, we make the following assumptions:

- Each array is an $n \times n$ array of `double`, with `sizeof(double)` $= 8$.
- There is a single cache with a 32-byte block size ($B = 32$).
- The array size n is so large that a single matrix row does not fit in the L1 cache.
- The compiler stores local variables in registers, and thus references to local variables inside loops do not require any load or store instructions.

Figure 6.45 summarizes the results of our inner-loop analysis. Notice that the six versions pair up into three equivalence classes, which we denote by the pair of matrices that are accessed in the inner loop. For example, versions ijk and jik are members of class AB because they reference arrays A and B (but not C) in their innermost loop. For each class, we have counted the number of loads (reads) and stores (writes) in each inner-loop iteration, the number of references to A, B, and C that will miss in the cache in each loop iteration, and the total number of cache misses per iteration.

The inner loops of the class AB routines (Figure 6.44(a) and (b)) scan a row of array A with a stride of 1. Since each cache block holds four 8-byte words, the miss rate for A is 0.25 misses per iteration. On the other hand, the inner loop scans a column of B with a stride of n. Since n is large, each access of array B results in a miss, for a total of 1.25 misses per iteration.

The inner loops in the class AC routines (Figure 6.44(c) and (d)) have some problems. Each iteration performs two loads and a store (as opposed to the

1. As we learned in Chapter 2, floating-point addition is commutative, but in general not associative. In practice, if the matrices do not mix extremely large values with extremely small ones, as often is true when the matrices store physical properties, then the assumption of associativity is reasonable.

(a) Version ijk

code/mem/matmult/mm.c

```
1   for (i = 0; i < n; i++)
2       for (j = 0; j < n; j++) {
3           sum = 0.0;
4           for (k = 0; k < n; k++)
5               sum += A[i][k]*B[k][j];
6           C[i][j] += sum;
7       }
```

code/mem/matmult/mm.c

(b) Version jik

code/mem/matmult/mm.c

```
1   for (j = 0; j < n; j++)
2       for (i = 0; i < n; i++) {
3           sum = 0.0;
4           for (k = 0; k < n; k++)
5               sum += A[i][k]*B[k][j];
6           C[i][j] += sum;
7       }
```

code/mem/matmult/mm.c

(c) Version jki

code/mem/matmult/mm.c

```
1   for (j = 0; j < n; j++)
2       for (k = 0; k < n; k++) {
3           r = B[k][j];
4           for (i = 0; i < n; i++)
5               C[i][j] += A[i][k]*r;
6       }
```

code/mem/matmult/mm.c

(d) Version kji

code/mem/matmult/mm.c

```
1   for (k = 0; k < n; k++)
2       for (j = 0; j < n; j++) {
3           r = B[k][j];
4           for (i = 0; i < n; i++)
5               C[i][j] += A[i][k]*r;
6       }
```

code/mem/matmult/mm.c

(e) Version kij

code/mem/matmult/mm.c

```
1   for (k = 0; k < n; k++)
2       for (i = 0; i < n; i++) {
3           r = A[i][k];
4           for (j = 0; j < n; j++)
5               C[i][j] += r*B[k][j];
6       }
```

code/mem/matmult/mm.c

(f) Version ikj

code/mem/matmult/mm.c

```
1   for (i = 0; i < n; i++)
2       for (k = 0; k < n; k++) {
3           r = A[i][k];
4           for (j = 0; j < n; j++)
5               C[i][j] += r*B[k][j];
6       }
```

code/mem/matmult/mm.c

Figure 6.44 **Six versions of matrix multiply.** Each version is uniquely identified by the ordering of its loops.

Matrix multiply version (class)	Per iteration					
	Loads	Stores	A misses	B misses	C misses	Total misses
ijk & jik (AB)	2	0	0.25	1.00	0.00	1.25
jki & kji (AC)	2	1	1.00	0.00	1.00	2.00
kij & ikj (BC)	2	1	0.00	0.25	0.25	0.50

Figure 6.45 **Analysis of matrix multiply inner loops.** The six versions partition into three equivalence classes, denoted by the pair of arrays that are accessed in the inner loop.

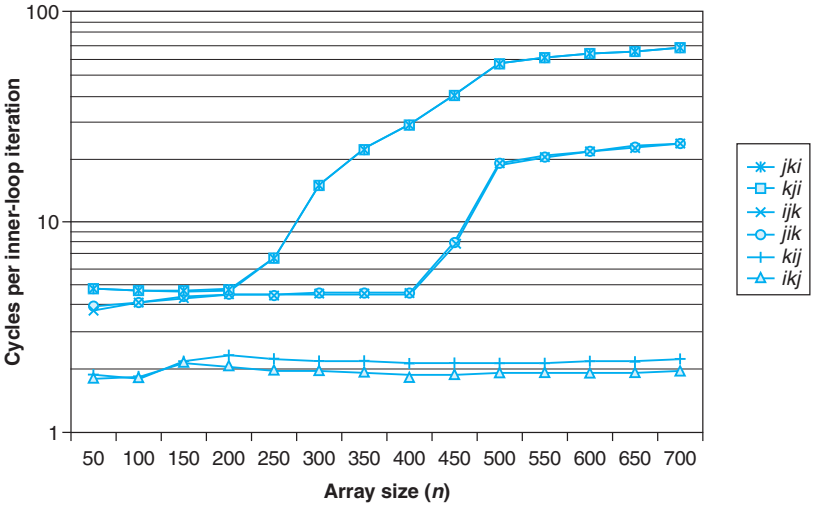


Figure 6.46 **Core i7 matrix multiply performance.**

class AB routines, which perform two loads and no stores). Second, the inner loop scans the columns of A and C with a stride of n. The result is a miss on each load, for a total of two misses per iteration. Notice that interchanging the loops has decreased the amount of spatial locality compared to the class AB routines.

The BC routines (Figure 6.44(e) and (f)) present an interesting trade-off: With two loads and a store, they require one more memory operation than the AB routines. On the other hand, since the inner loop scans both B and C row-wise with a stride-1 access pattern, the miss rate on each array is only 0.25 misses per iteration, for a total of 0.50 misses per iteration.

Figure 6.46 summarizes the performance of different versions of matrix multiply on a Core i7 system. The graph plots the measured number of CPU cycles per inner-loop iteration as a function of array size (n).

There are a number of interesting points to notice about this graph:

- For large values of n, the fastest version runs almost 40 times faster than the slowest version, even though each performs the same number of floating-point arithmetic operations.
- Pairs of versions with the same number of memory references and misses per iteration have almost identical measured performance.
- The two versions with the worst memory behavior, in terms of the number of accesses and misses per iteration, run significantly slower than the other four versions, which have fewer misses or fewer accesses, or both.
- Miss rate, in this case, is a better predictor of performance than the total number of memory accesses. For example, the class BC routines, with 0.5 misses per iteration, perform much better than the class AB routines, with 1.25 misses per iteration, even though the class BC routines perform more

Web Aside MEM:BLOCKING Using blocking to increase temporal locality

There is an interesting technique called *blocking* that can improve the temporal locality of inner loops. The general idea of blocking is to organize the data structures in a program into large chunks called *blocks*. (In this context, "block" refers to an application-level chunk of data, *not* to a cache block.) The program is structured so that it loads a chunk into the L1 cache, does all the reads and writes that it needs to on that chunk, then discards the chunk, loads in the next chunk, and so on.

Unlike the simple loop transformations for improving spatial locality, blocking makes the code harder to read and understand. For this reason, it is best suited for optimizing compilers or frequently executed library routines. Blocking does not improve the performance of matrix multiply on the Core i7, because of its sophisticated prefetching hardware. Still, the technique is interesting to study and understand because it is a general concept that can produce big performance gains on systems that don't prefetch.

memory references in the inner loop (two loads and one store) than the class AB routines (two loads).

- For large values of n, the performance of the fastest pair of versions (kij and ikj) is constant. Even though the array is much larger than any of the SRAM cache memories, the prefetching hardware is smart enough to recognize the stride-1 access pattern, and fast enough to keep up with memory accesses in the tight inner loop. This is a stunning accomplishment by the Intel engineers who designed this memory system, providing even more incentive for programmers to develop programs with good spatial locality.

6.6.3 Exploiting Locality in Your Programs

As we have seen, the memory system is organized as a hierarchy of storage devices, with smaller, faster devices toward the top and larger, slower devices toward the bottom. Because of this hierarchy, the effective rate that a program can access memory locations is not characterized by a single number. Rather, it is a wildly varying function of program locality (what we have dubbed the memory mountain) that can vary by orders of magnitude. Programs with good locality access most of their data from fast cache memories. Programs with poor locality access most of their data from the relatively slow DRAM main memory.

Programmers who understand the nature of the memory hierarchy can exploit this understanding to write more efficient programs, regardless of the specific memory system organization. In particular, we recommend the following techniques:

- Focus your attention on the inner loops, where the bulk of the computations and memory accesses occur.
- Try to maximize the spatial locality in your programs by reading data objects sequentially, with stride 1, in the order they are stored in memory.
- Try to maximize the temporal locality in your programs by using a data object as often as possible once it has been read from memory.

6.7 Summary

The basic storage technologies are random access memories (RAMs), nonvolatile memories (ROMs), and disks. RAM comes in two basic forms. Static RAM (SRAM) is faster and more expensive and is used for cache memories. Dynamic RAM (DRAM) is slower and less expensive and is used for the main memory and graphics frame buffers. ROMs retain their information even if the supply voltage is turned off. They are used to store firmware. Rotating disks are mechanical nonvolatile storage devices that hold enormous amounts of data at a low cost per bit, but with much longer access times than DRAM. Solid state disks (SSDs) based on nonvolatile flash memory are becoming increasingly attractive alternatives to rotating disks for some applications.

In general, faster storage technologies are more expensive per bit and have smaller capacities. The price and performance properties of these technologies are changing at dramatically different rates. In particular, DRAM and disk access times are much larger than CPU cycle times. Systems bridge these gaps by organizing memory as a hierarchy of storage devices, with smaller, faster devices at the top and larger, slower devices at the bottom. Because well-written programs have good locality, most data are served from the higher levels, and the effect is a memory system that runs at the rate of the higher levels, but at the cost and capacity of the lower levels.

Programmers can dramatically improve the running times of their programs by writing programs with good spatial and temporal locality. Exploiting SRAM-based cache memories is especially important. Programs that fetch data primarily from cache memories can run much faster than programs that fetch data primarily from memory.

Bibliographic Notes

Memory and disk technologies change rapidly. In our experience, the best sources of technical information are the Web pages maintained by the manufacturers. Companies such as Micron, Toshiba, and Samsung provide a wealth of current technical information on memory devices. The pages for Seagate and Western Digital provide similarly useful information about disks.

Textbooks on circuit and logic design provide detailed information about memory technology [58, 89]. *IEEE Spectrum* published a series of survey articles on DRAM [55]. The International Symposiums on Computer Architecture (ISCA) and High Performance Computer Architecture (HPCA) are common forums for characterizations of DRAM memory performance [28, 29, 18].

Wilkes wrote the first paper on cache memories [117]. Smith wrote a classic survey [104]. Przybylski wrote an authoritative book on cache design [86]. Hennessy and Patterson provide a comprehensive discussion of cache design issues [46]. Levinthal wrote a comprehensive performance guide for the Intel Core i7 [70].

Stricker introduced the idea of the memory mountain as a comprehensive characterization of the memory system in [112] and suggested the term “memory mountain” informally in later presentations of the work. Compiler researchers

work to increase locality by automatically performing the kinds of manual code transformations we discussed in Section 6.6 [22, 32, 66, 72, 79, 87, 119]. Carter and colleagues have proposed a cache-aware memory controller [17]. Other researchers have developed *cache-oblivious* algorithms that are designed to run well without any explicit knowledge of the structure of the underlying cache memory [30, 38, 39, 9].

There is a large body of literature on building and using disk storage. Many storage researchers look for ways to aggregate individual disks into larger, more robust, and more secure storage pools [20, 40, 41, 83, 121]. Others look for ways to use caches and locality to improve the performance of disk accesses [12, 21]. Systems such as Exokernel provide increased user-level control of disk and memory resources [57]. Systems such as the Andrew File System [78] and Coda [94] extend the memory hierarchy across computer networks and mobile notebook computers. Schindler and Ganger developed an interesting tool that automatically characterizes the geometry and performance of SCSI disk drives [95]. Researchers have investigated techniques for building and using flash-based SSDs [8, 81].

Homework Problems

6.22 ◆◆

Suppose you are asked to design a rotating disk where the number of bits per track is constant. You know that the number of bits per track is determined by the circumference of the innermost track, which you can assume is also the circumference of the hole. Thus, if you make the hole in the center of the disk larger, the number of bits per track increases, but the total number of tracks decreases. If you let r denote the radius of the platter, and $x \cdot r$ the radius of the hole, what value of x maximizes the capacity of the disk?

6.23 ◆

Estimate the average time (in ms) to access a sector on the following disk:

Parameter	Value
Rotational rate	12,000 RPM
$T_{\text{avg seek}}$	3 ms
Average number of sectors/track	500

6.24 ◆◆

Suppose that a 2 MB file consisting of 512-byte logical blocks is stored on a disk drive with the following characteristics:

Parameter	Value
Rotational rate	18,000 RPM
$T_{\text{avg seek}}$	8 ms
Average number of sectors/track	2,000
Surfaces	4
Sector size	512 bytes