The Memory Hierarchy -Chapter 6

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.

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**.

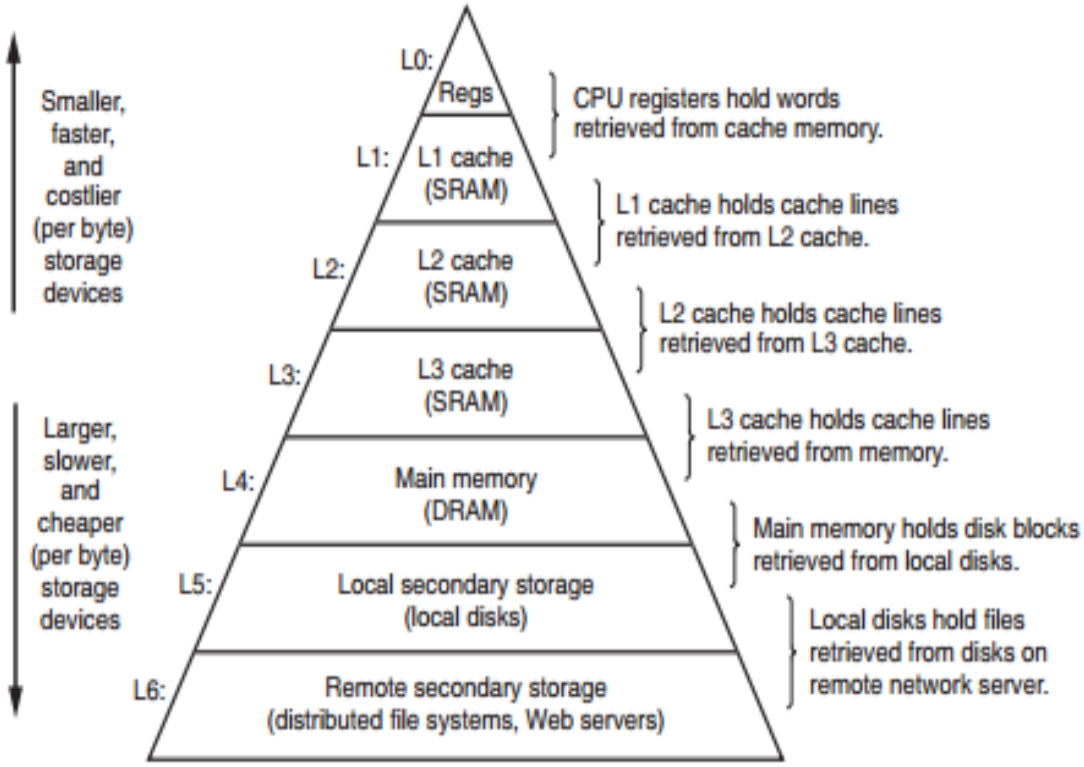


Figure 1.9 **An example of a memory hierarchy.**

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 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.

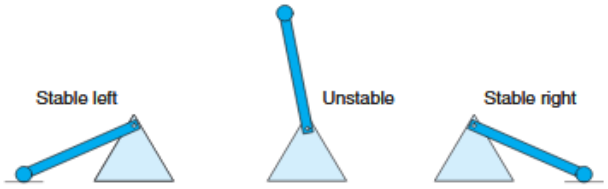


Figure 6.1
Inverted pendulum. Like an SRAM cell, the pendulum has only two stable configurations, or *states*.

	Transistors per bit	Relative access time	Persistent?	Sensitive?	Relative cost	Applications
SRAM	6	1×	Yes	No	100×	Cache memory
DRAM	1	10×	No	Yes	1×	Main mem, 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 a few more bits (e.g., a 32-bit word might be encoded using 38 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

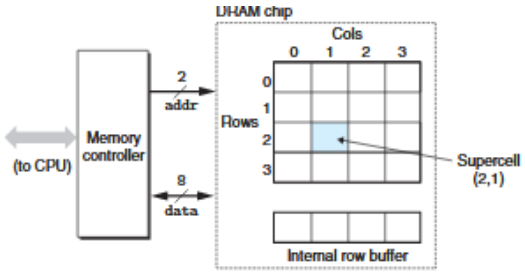


Figure 6.3
High-level view of a 128-bit 16 × 8 DRAM chip.

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.

Aside A note on terminology

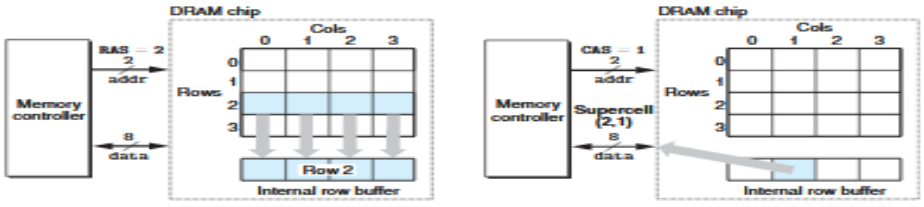
The storage community has never settled on a standard name for a DRAM array element. Computer architects tend to refer to it as a "cell," overloading the term with the DRAM storage cell. Circuit designers tend to refer to it as a "word," overloading the term with a word of main memory. To avoid confusion,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

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). Common packages include the 168-pin *dual inline memory module* (DIMM), which transfers data to and from the memory controller in 64-bit chunks, and the 72-pin *single inline memory module* (SIMM), which transfers data in 32-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 doubleword[1] 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 a 64-bit doubleword 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 doubleword, 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.

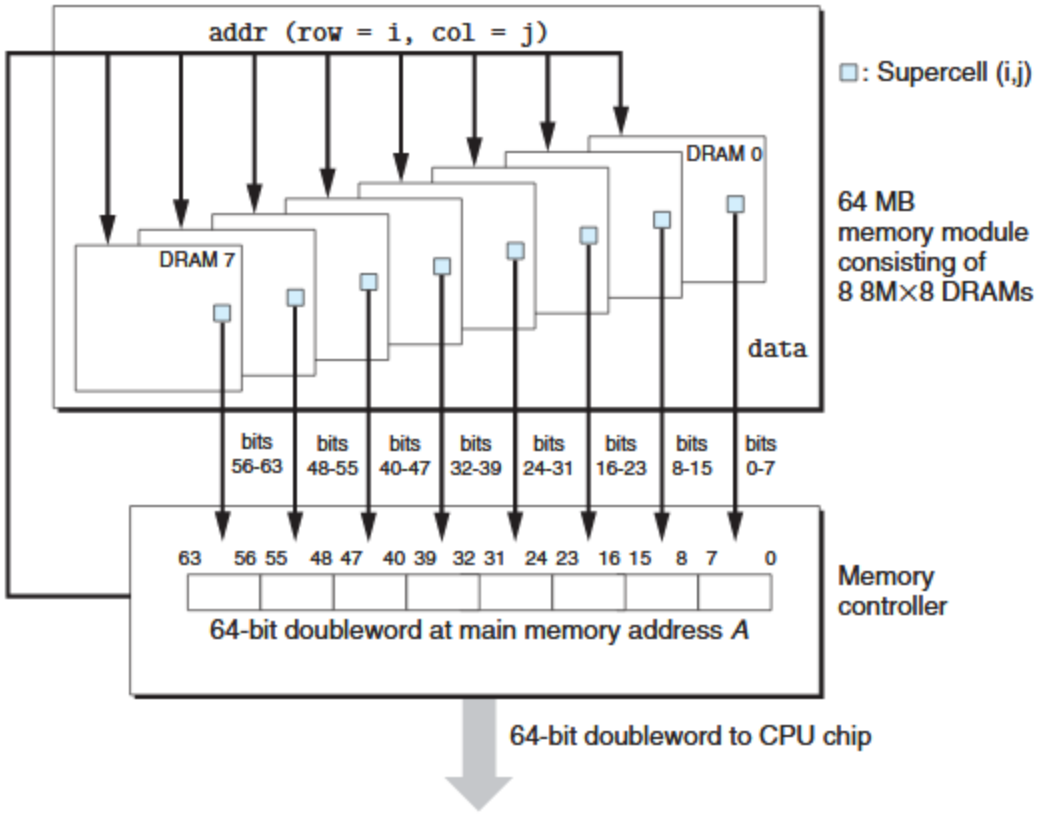


Figure 6.5 **Reading the contents of a memory module.**

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.

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 1000 times. An *electrically erasable PROM* (EEPROM) is akin to an EPROM, but 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, music players, PDAs, and 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

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 they 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,

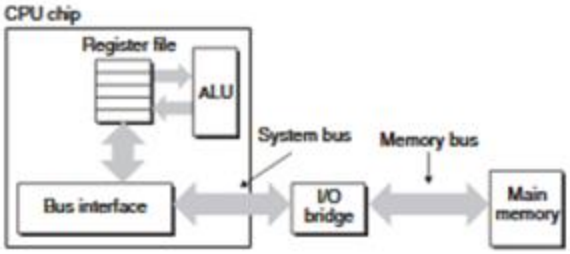
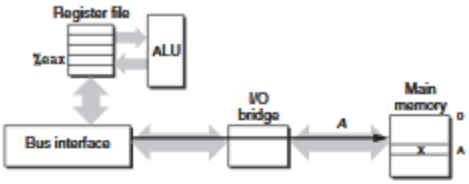


Figure 6.6
Example bus structure that connects the CPU and main memory.

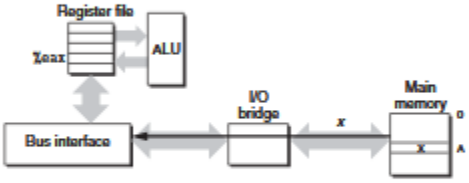
is this transaction of interest to the main memory, or to some other I/O device such as a disk controller? Is the transaction a read or a write? Is the information on the bus an address or a data item?

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 bridge to the main memory.

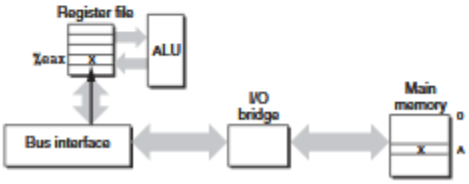
The I/O bridge translates the electrical signals of the system bus into the electrical signals of the memory bus. As we will see, the I/O bridge also connects the system bus and memory bus to an I/O bus that is shared by I/O devices such as disks and graphics cards. For now, though, we will focus on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



(a) CPU places address *A* on the memory bus.

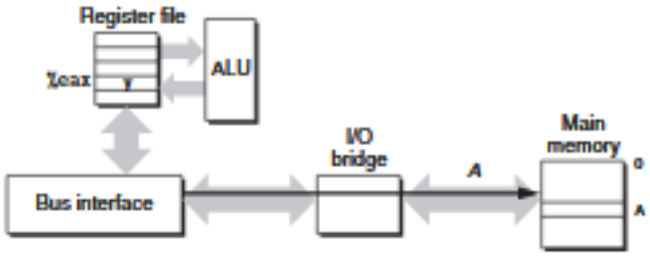


(b) Main memory reads *A* from the bus, retrieves word *x*, and places it on the bus.

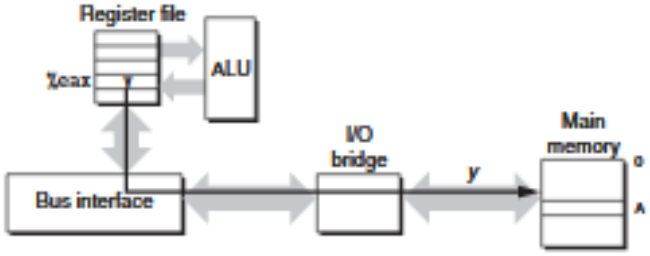


(c) CPU reads word *x* from the bus, and copies it into register %eax.

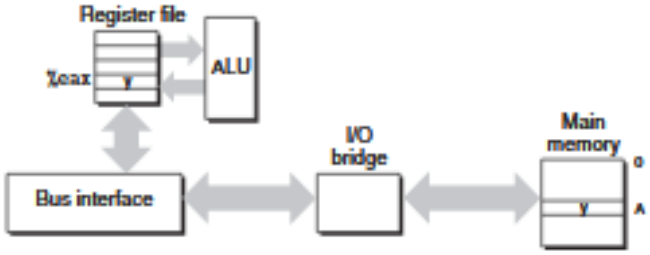
Figure 6.7 **Memory read transaction for a load operation: `movl A,%eax`.**



(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*.

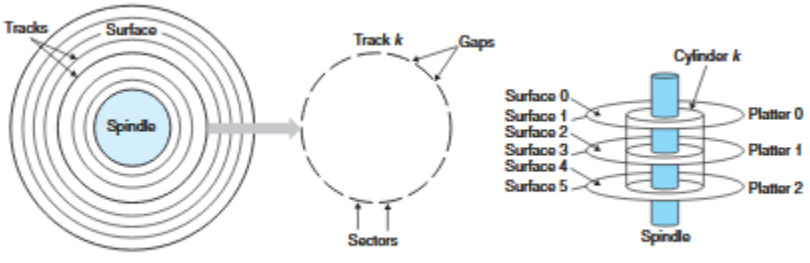
Figure 6.8 **Memory write transaction for a store operation: `movl %eax,A`.**

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.

Disk Geometry

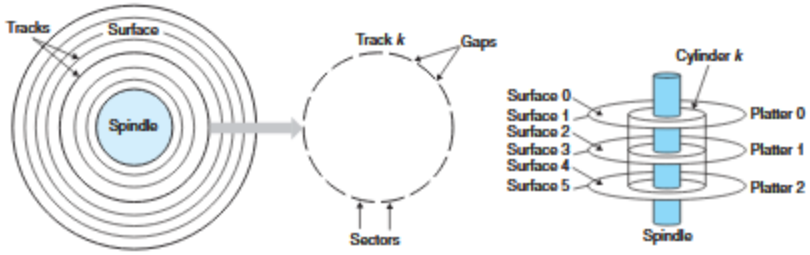
Disks are constructed from *platters*. Each platter consists of two sides, or *surfaces*, that are coated with magnetic recording material. A rotating *spindle* in the center of the platter spins the platter at a fixed *rotational rate*, typically between 5400 and



(a) Single-platter view

(b) Multiple-platter view

Figure 6.9 **Disk geometry.**



(a) Single-platter view

(b) Multiple-platter view

Figure 6.9 **Disk geometry.**

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 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 sometime 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.

Disk manufacturers work tirelessly to increase areal density (and thus capacity), and this is doubling every few 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 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. Note that diskettes (floppy disks) still use the old-fashioned approach, with a constant number of sectors per track.

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

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

For example, suppose we have a disk with 5 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:

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

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

- **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, $T_{avg\ seek}$, 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, $T_{max\ seek}$, 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 sector 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

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

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

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

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	7200 RPM
$T_{avg\ seek}$	9 ms
Average # sectors/track	400

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

$$\begin{aligned} T_{avg\ rotation} &= 1/2 \times T_{max\ rotation} \\ &= 1/2 \times (60\ \text{secs} / 7200\ \text{RPM}) \times 1000\ \text{ms/sec} \\ &\approx 4\ \text{ms} \end{aligned}$$

The average transfer time is

$$\begin{aligned} T_{avg\ transfer} &= 60 / 7200\ \text{RPM} \times 1 / 400\ \text{sectors/track} \times 1000\ \text{ms/sec} \\ &\approx 0.02\ \text{ms} \end{aligned}$$

Putting it all together, the total estimated access time is

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

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 doubleword stored in SRAM is roughly 4 ns, and 60 ns for DRAM. Thus, the time to read a 512-byte sector-sized block from memory is roughly 256 ns for SRAM and 4000 ns for DRAM. The disk access time, roughly 10 ms, is about 40,000 times greater than SRAM, and about 2500 times greater than DRAM. The difference in access times is even more dramatic if we compare the times to access a single word.

Practice Problem 6.3

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

Parameter	Value
Rotational rate	15,000 RPM
$T_{avg\ seek}$	8 ms
Average # sectors/track	500

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 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.

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.

Caches

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 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.

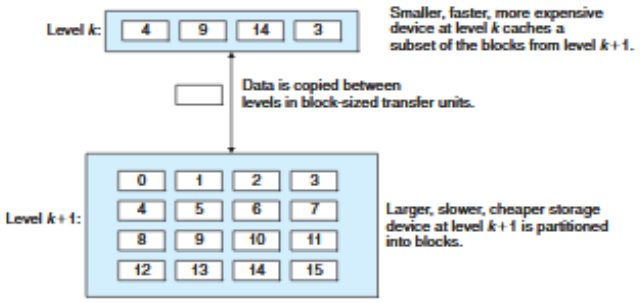


Figure 6.24 **The basic principle of caching in a memory hierarchy.**

Figure 6.24 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-sized (the usual case) or variable-sized (e.g., the remote HTML files stored on Web servers). For example, the level $k+1$ storage in Figure 6.24 is partitioned into 16 fixed-sized 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

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 6.24, 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.