Chapter 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 as far as it goes, 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 zero cycles during the execution of the instruction. If stored in a cache, one to ten cycles. If stored in main memory, 50 to 100 cycles. And if stored in disk about 20,000,000 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.

The 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, 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 six!

In this chapter, we will look at the basic storage technologies — SRAM memory, DRAM memory, ROM memory, and 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 2000, typical machines had 1,000 times as much disk storage, and the ratio was increasing by a factor of 10 every two or three 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 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.

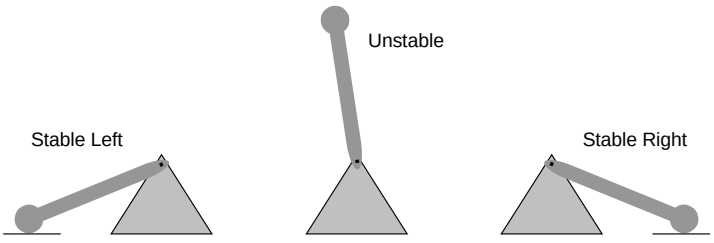


Figure 6.1: **Inverted pendulum.** Like an SRAM cell, the pendulum has only two stable configurations, or *states*.

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.

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 cycles 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 to them. 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.

	Transistors per bit	Relative access time	Persistent?	Sensitive?	Relative cost	Applications
SRAM	6	1X	Yes	No	100X	Cache memory
DRAM	1	10X	No	Yes	1X	Main mem, frame buffers

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

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: 8 `data` pins that can transfer one byte in or out of the chip, and 2 `addr` pins that carry two-bit row and column supercell addresses. Other pins that carry control information are not shown.

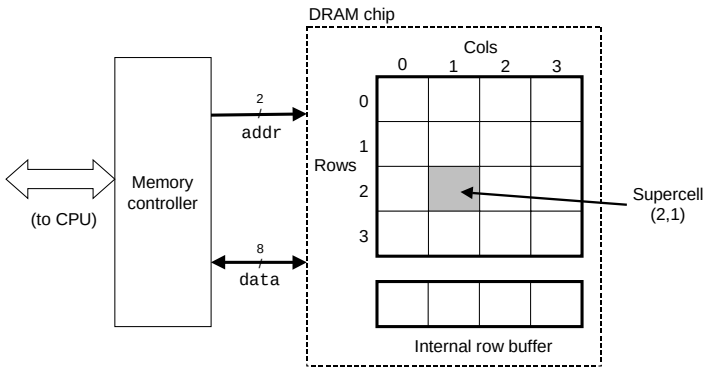


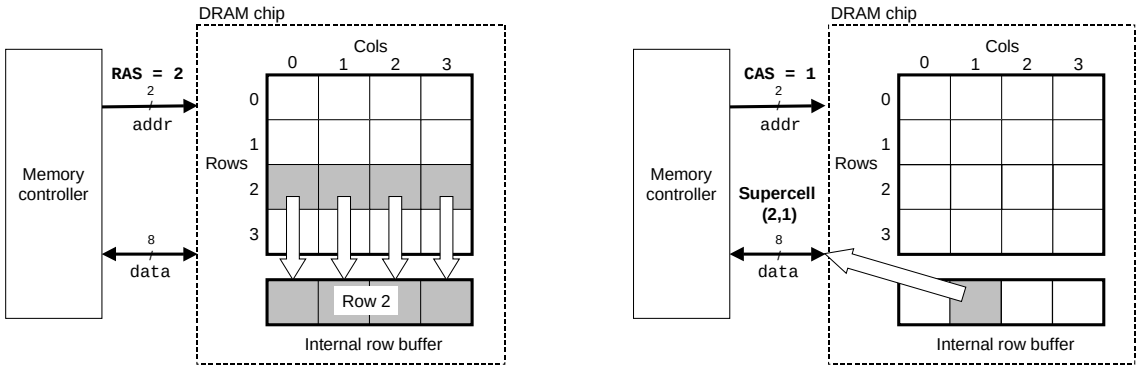
Figure 6.3: **High level view of a 128-bit 16×8 DRAM chip.**

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". **End Aside.**

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 one 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.

Practice Problem 6.1:

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-two array dimensions that minimize $\max(b_r, b_c)$, the maximum number of bits needed to address the rows or columns of the array.

[1] IA32 would call this 64-bit quantity a "quadword."

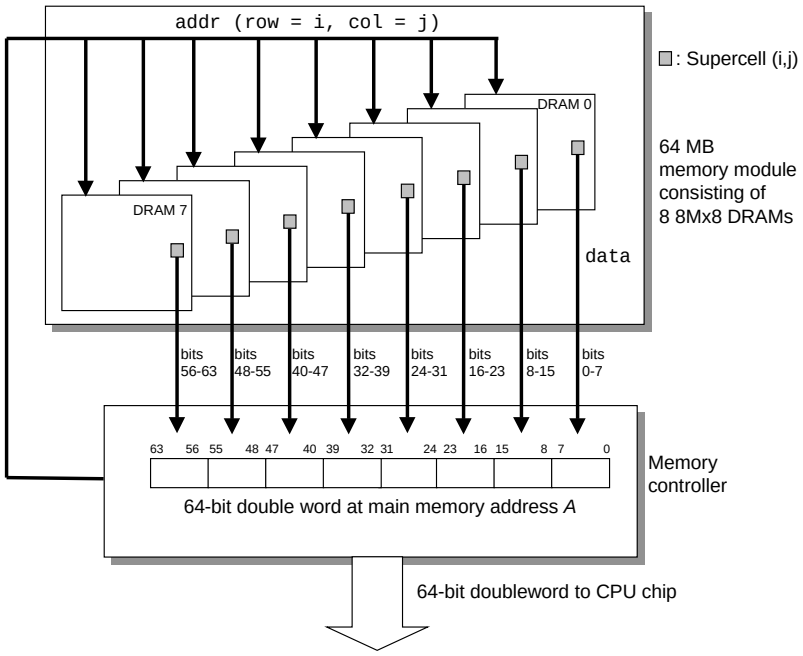


Figure 6.5: **Reading the contents of a memory module.**

Organization	r	c	b_r	b_c	$\max(b_r, b_c)$
16×1					
16×4					
128×8					
512×4					
1024×4					

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 first supercell. The next three supercells are served directly from the row buffer, and thus 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.
- *Rambus DRAM (RDRAM)*. This is an alternative proprietary technology with a higher maximum bandwidth than DDR SDRAM.
- *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).

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. **End Aside.**

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. *Flash memory* is a family of small nonvolatile memory cards, based on EEPROMs, that can be plugged in and out of a desktop machine, handheld device, or video game console.

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 drives 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 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, 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 a typical desktop 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.

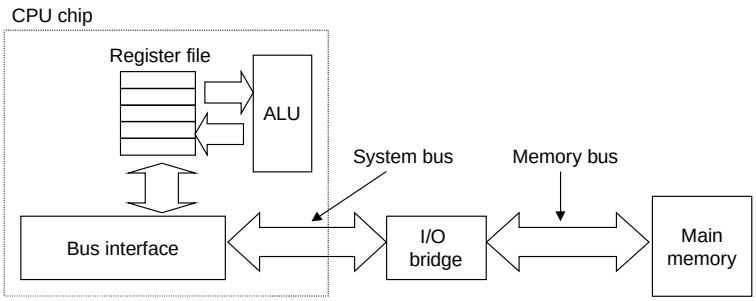


Figure 6.6: **Typical bus structure that connects the CPU and main memory.**

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

```
movl A,%eax
```

where the contents of address A are loaded into register `%eax`. 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 word 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 it from the bus, and copies it to register `%eax` (Figure 6.7(c)).

Conversely, when the CPU performs a store instruction such as

```
movl %eax,A
```

where the contents of register `%eax` 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 word in `%eax` to the system bus (Figure 6.8(b)). Finally, the main memory reads the data word 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 tens to hundreds 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.

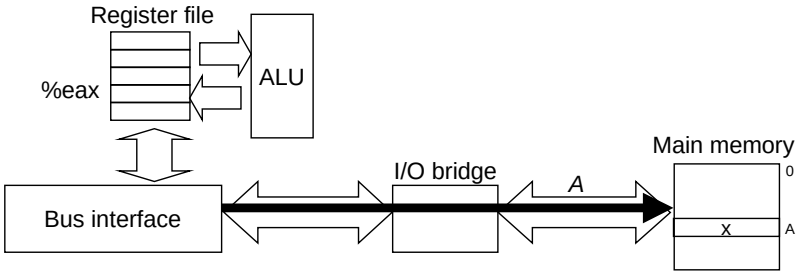
Disk Geometry

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 15,000 *revolutions per minute (RPM)*. A disk will typically contain one or more of these platters encased in a sealed container.

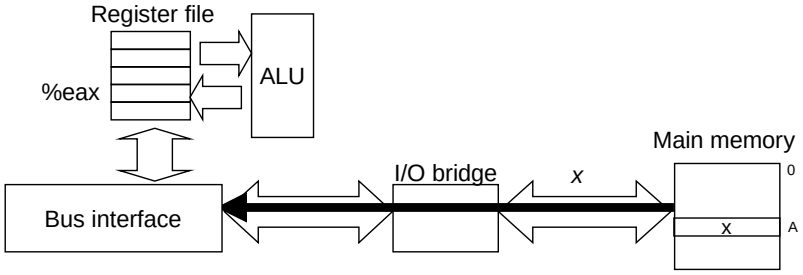
Figure 6.9(a) shows the geometry of a typical disk surface. Each surface consists of a collection of concentric rings called *tracks*. Each track is partitioned into a collection of *sectors*. Each sector contains an equal number of data bits (typically 512 bytes) encoded in the magnetic material on 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*.

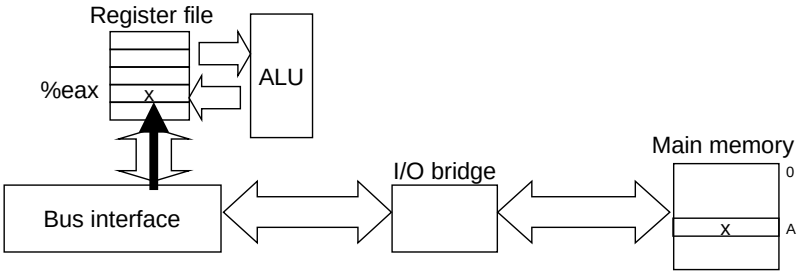
Disk manufacturers often describe the geometry of multiple-platter drives in terms of *cylinders*, where a cylinder is the collection of tracks on all the surfaces that are equidistant from the center of the spindle. For example, if a drive has three platters and six surfaces, and the tracks on each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



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

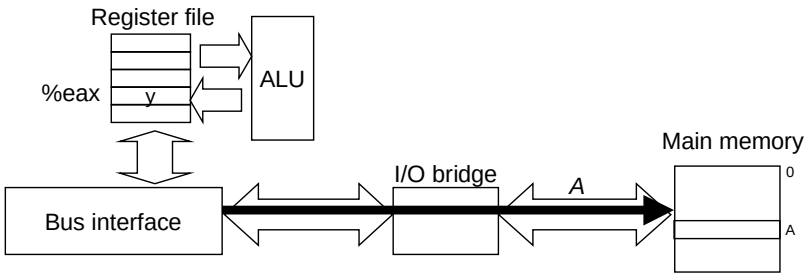


(b) Main memory reads A from the bus, retrieves word x, and places it on the bus.

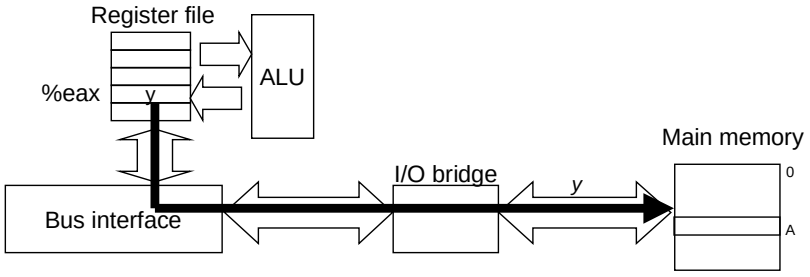


(c) CPU reads word x from the bus, and copies it into register `%eax`.

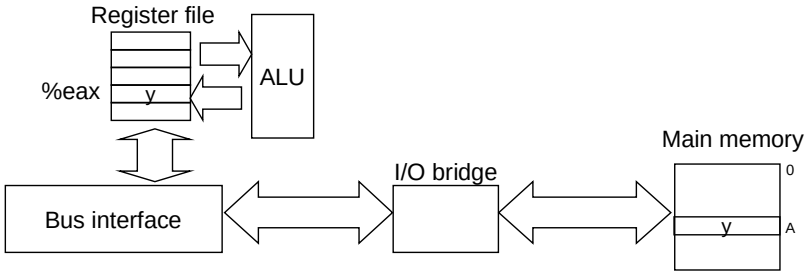
Figure 6.7: **Memory read transaction for a load operation: `movl A,%eax`.**



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

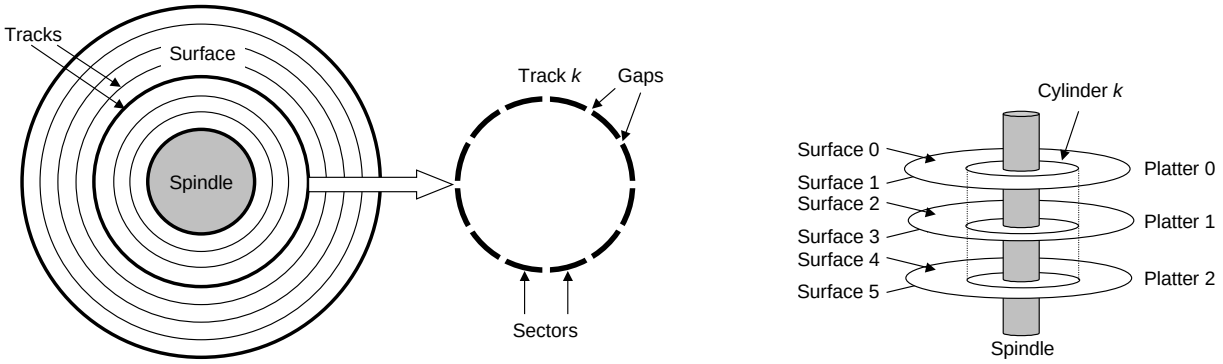


(b) CPU places data word y on the bus.



(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`.**



(a) Single-platter view.

(b) Multiple-platter view.

Figure 6.9: **Disk geometry.**

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 one-inch segment of a track.
- *Track density* (*tracks*/*in*): The number of tracks that can be squeezed into a one-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 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 further 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 tracks is partitioned into disjoint subsets known as *recording zones*. Each zone contains a contiguous collection of tracks. Each track 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:

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

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:

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

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

Aside: How much is a gigabyte?
Unfortunately, the meanings of prefixes such as kilo (K), mega (M) and giga (G) depend on the context. For measures that relate to the capacity of DRAMs and SRAMs, typically $K = 2^{10}$, $M = 2^{20}$ and $G = 2^{30}$. For measures related to the capacity of I/O devices such as disks and networks, typically $K = 10^3$, $M = 10^6$ and $G = 10^9$. 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 $2^{20} = 1,048,576$ and $10^6 = 1,000,000$ is small: $(2^{20} - 10^6)/10^6 \approx 5\%$. Similarly for $2^{30} = 1,073,741,824$ and $10^9 = 1,000,000,000$: $(2^{30} - 10^9)/10^9 \approx 7\%$. **End Aside.**

Practice Problem 6.2:

What is the capacity of a disk with 2 platters, 10,000 cylinders, an average of 400 sectors per track, and 512 bytes per sector?

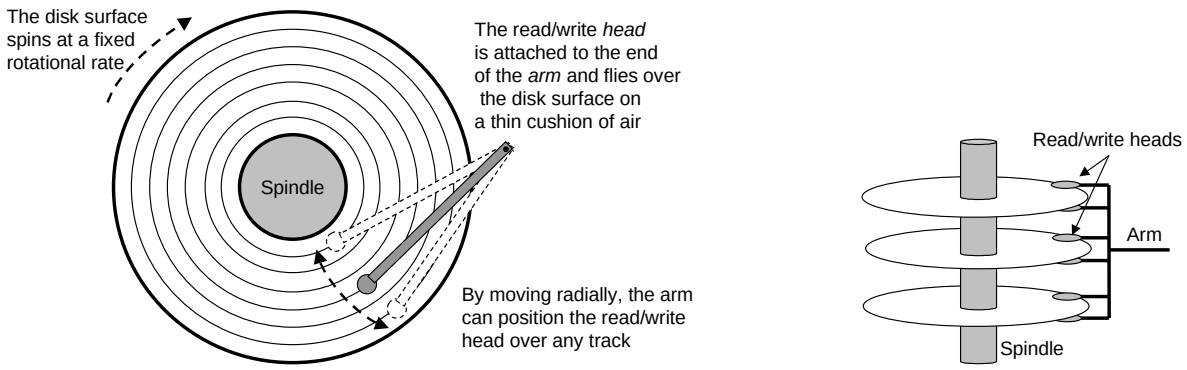
Disk Operation

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 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 the Sears Tower 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 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-sized 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



(a) Single-platter view (b) Multiple-platter view

Figure 6.10: **Disk dynamics.**

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 6 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

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

  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

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

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

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

$$\begin{aligned} T_{avg\,rotation} &= 1/2 \times T_{max\,rotation} \\ &= 1/2 \times (60\ secs\ /\ 7{,}200\ RPM) \times 1000\ ms/sec \\ &\approx 4\ ms. \end{aligned}$$

The average transfer time is

$$\begin{aligned} T_{avg\,transfer} &= 60\ /\ 7{,}200\ RPM \times 1\ /\ 400\ sectors/track \times 1000\ ms/sec \\ &\approx 0.02\ ms. \end{aligned}$$

Putting it all together, the total estimated access time is

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

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 2,500 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

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-sized *logical blocks*, numbered $0, 1, \ldots, b-1$. A small hardware/firmware device in the disk, 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 buffer on the controller, and copies them into main memory.

> **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 of 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. **End Aside.**

Accessing Disks

Devices such as graphics cards, monitors, mice, keyboards, and disks are connected to the CPU and main memory using an *I/O bus* such as Intel's *Peripheral Component Interconnect* (PCI) bus. Unlike the system bus and memory buses, which are CPU-specific, I/O buses such as PCI are designed to be independent of the underlying CPU. For example, PCs and Macintosh's both incorporate the PCI bus. Figure 6.11 shows a typical I/O bus structure (modeled on PCI) 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 the USB. A USB has a throughput of 12 Mbits/s and is designed for slow to moderate speed serial devices such as keyboards, mice, modems, digital cameras, joysticks, CD-ROM drives, and printers.
- 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 disk controller contains the hardware and software logic for reading and writing disk data on behalf of the CPU.

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.

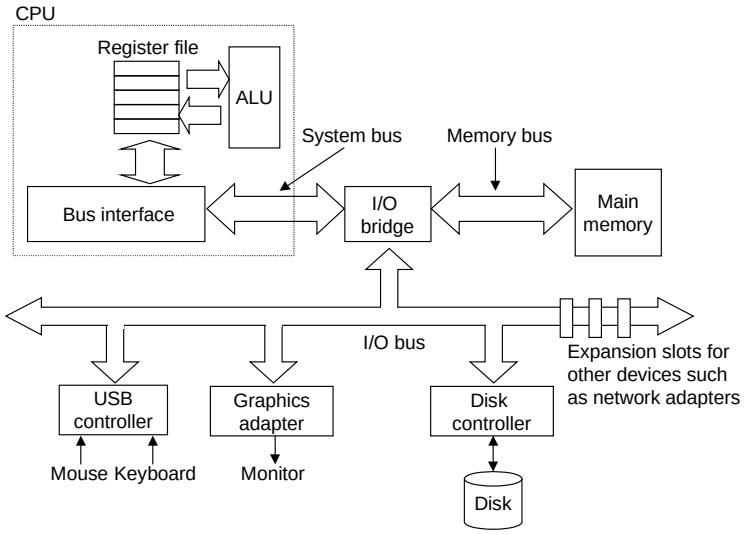


Figure 6.11: **Typical bus structure that connects the CPU, main memory, and I/O devices.**

While a detailed description of how I/O devices work and how they are programmed is outside our scope, 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.

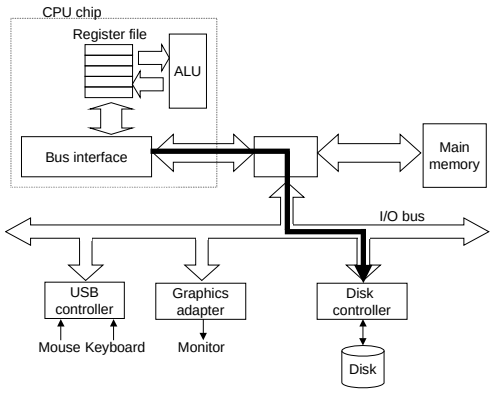
The CPU issues commands to I/O devices using a technique called *memory-mapped I/O* (Figure 6.12(a)). In a system with memory-mapped I/O, a block of addresses in the address space is reserved for communicating with I/O devices. Each of these addresses is known as an *I/O port*. Each device is associated 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 `0xa`: 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 number of the logical block that should be read. The third instruction indicates the main memory address where the contents of the disk sector should be stored.

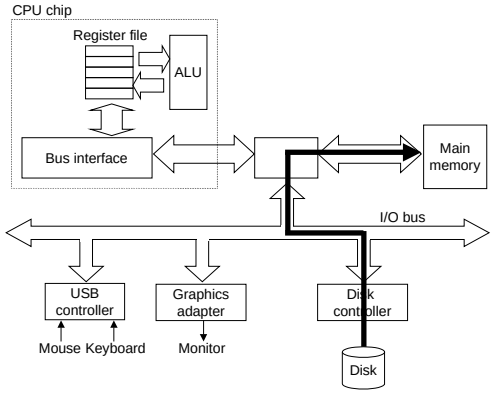
After it issues the request, the CPU will typically do other work while the disk is performing the read. Recall that a 1 GHz processor with a 1 ns clock cycle can potentially execute 16 million instructions in the 16 ms it takes to read the disk. Simply waiting and doing nothing while the transfer is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

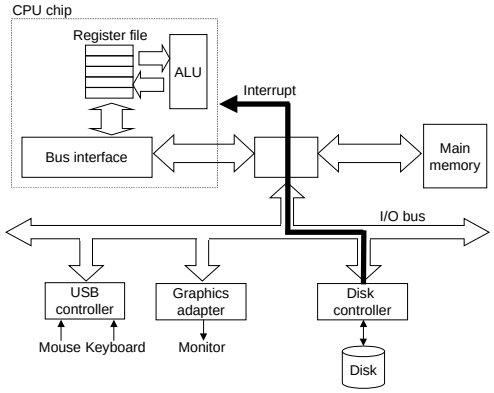
After the DMA transfer is complete and the contents of the disk sector are safely stored in main memory, the disk controller notifies the CPU by sending an interrupt signal to the CPU (Figure 6.12(c)). The basic



(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.

Figure 6.12: **Reading a disk sector.**

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.

Aside: Anatomy of a commercial disk.
Disk manufacturers publish a lot of high-level technical information on their Web pages. For example, if we visit the Web page for the IBM Ultrastar 36LZX disk, we can glean the geometry and performance information shown in Figure 6.13.

Geometry attribute	Value
Platters	6
Surfaces (heads)	12
Sector size	512 bytes
Zones	11
Cylinders	15,110
Recording density (max)	352,000 bits/in.
Track density	20,000 tracks/in.
Areal density (max)	7040 Mbits/sq. in.
Formatted capacity	36 GBytes

Performance attribute	Value
Rotational rate	10,000 RPM
Avg. rotational latency	2.99 ms
Avg. seek time	4.9 ms
Sustained transfer rate	21–36 MBytes/s

Figure 6.13: **IBM Ultrastar 36LZX geometry and performance.** Source: `www.storage.ibm.com`

Disk manufacturers often neglect to publish detailed technical information about the geometry of the individual recording zones. However, storage researchers have developed a useful tool, called DIXtrac, that automatically discovers a wealth of low-level information about the geometry and performance of SCSI disks [68]. For example, DIXtrac is able to discover the detailed zone geometry of our example IBM disk, which we've shown in Figure 6.14. Each row in the table characterizes one of the 11 zones on the disk surface, in terms of the number of sectors in the zone, the range of logical blocks mapped to the sectors in the zone, and the range and number of cylinders in the zone.

Zone number	Sectors per track	Starting logical block	Ending logical block	Starting cylinder	Ending cylinder	Cylinders per zone
(outer) 0	504	0	2,292,096	1	380	380
1	476	2,292,097	11,949,751	381	2,078	1,698
2	462	11,949,752	19,416,566	2,079	3,430	1,352
3	420	19,416,567	36,409,689	3,431	6,815	3,385
4	406	36,409,690	39,844,151	6,816	7,523	708
5	392	39,844,152	46,287,903	7,524	8,898	1,375
6	378	46,287,904	52,201,829	8,899	10,207	1,309
7	364	52,201,830	56,691,915	10,208	11,239	1,032
8	352	56,691,916	60,087,818	11,240	12,046	807
9	336	60,087,819	67,001,919	12,047	13,768	1,722
(inner) 10	308	67,001,920	71,687,339	13,769	15,042	1,274

Figure 6.14: **IBM Ultrastar 36LZX zone map.** Source: DIXtrac automatic disk drive characterization tool [68].

The zone map confirms some interesting facts about the IBM disk. First, more tracks are packed into the outer zones (which have a larger circumference) than the inner zones. Second, each zone has more sectors than logical blocks (check this yourself). The unused sectors form a pool of spare cylinders. If the recording material on a sector goes bad, the disk controller will automatically remap the logical blocks on that cylinder to an available spare. So we see

that the notion of a logical block not only provides a simpler interface to the operating system, it also provides a level of indirection that enables the disk to be more robust. This general idea of indirection is very powerful, as we will see when we study virtual memory in Chapter 10. **End Aside.**

6.1.3 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 tradeoffs.* 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.

- *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 1980, when the first PCs were introduced. The numbers were culled from back issues of trade magazines. Although they were collected in an informal survey, the numbers reveal some interesting trends.

  Since 1980, both the cost and performance of SRAM technology have improved at roughly the same rate. Access times have decreased by a factor of about 100 and cost per megabyte by a factor of 200 (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 8000 (almost four orders of magnitude!), DRAM access times have decreased by only a factor of 6 or so (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 50,000 since 1980, access times have improved much more slowly, by only a factor of 10 or so (Figure 6.15(c)). These startling long-term trends highlight a basic truth of memory and disk technology: it is easier to increase density (and thereby reduce cost) than to decrease access time.

- *DRAM and disk access times are lagging behind CPU cycle times.* As we see in Figure 6.15(d), CPU cycle times improved by a factor of 600 between 1980 and 2000. While SRAM performance lags, it is roughly keeping up. However, the gap between DRAM and disk performance and CPU performance is actually widening. 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.

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,

Metric	1980	1985	1990	1995	2000	2000:1980
$/MB	19,200	2,900	320	256	100	190
Access (ns)	300	150	35	15	3	100

(a) SRAM trends

Metric	1980	1985	1990	1995	2000	2000:1980
$/MB	8,000	880	100	30	1	8,000
Access (ns)	375	200	100	70	60	6
Typical size (MB)	0.064	0.256	4	16	64	1,000

(b) DRAM trends

Metric	1980	1985	1990	1995	2000	2000:1980
$/MB	500	100	8	0.30	0.01	50,000
Seek time (ms)	87	75	28	10	8	11
Typical size (MB)	1	10	160	1,000	20,000	20,000

(c) Disk trends

Metric	1980	1985	1990	1995	2000	2000:1980
Intel CPU	8080	80286	80386	Pentium	P-III	—
CPU clock rate (MHz)	1	6	20	150	600	600
CPU cycle time (ns)	1,000	166	50	6	1.6	600

(d) CPU trends

Figure 6.15: **Storage and processing technology trends.**

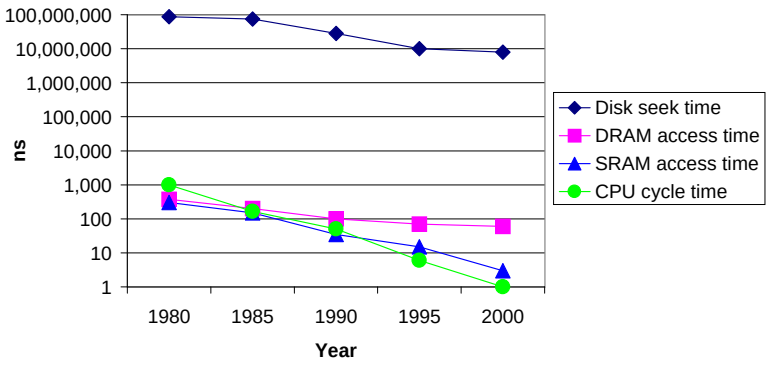


Figure 6.16: **The increasing gap between DRAM, disk, and CPU speeds.**

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.

6.2.1 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`.

```
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.

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). 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. 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

```
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.

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.

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

```
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-$(N \times sizeof(int))$ reference pattern.

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-($N \times sizeof(int)$) 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 can not be modified at run time. While a program is executing, the CPU only reads its instructions from memory. The CPU never overwrites or modifies these instructions.

6.2.3 Summary of Locality

In this section, we have introduced the fundamental idea of locality and we 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 6.4:

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

```
1 int sumarray3d(int a[N][N][N])
2 {
3     int i, j, k, sum = 0;
4
5     for (i = 0; i < N; i++) {
6         for (j = 0; j < N; j++) {
7             for (k = 0; k < N; k++) {
```

```
                sum += a[k][i][j];
            }
        }
    }
    return sum;
}
```

```
#define N 1000

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

point p[N];
```

(a) An array of `structs`.

```
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;
    }
}
```

(b) The `clear1` 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;
        }
    }
}
```

(a) The `clear2` 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;
    }
}
```

(b) The `clear3` function.

Figure 6.20: **Code examples for Practice Problem 6.5.**

Practice Problem 6.5:

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:

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

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 faster, cheaper, and larger as we move

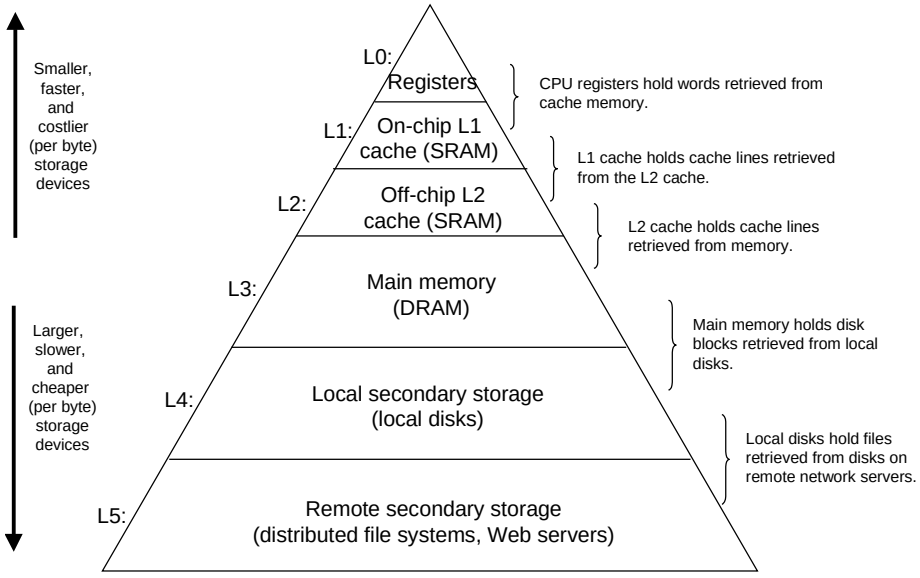


Figure 6.21: **The memory hierarchy.**

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-sized 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.

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 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. **End Aside.**

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 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-sized (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-sized blocks, numbered 0 to 15.

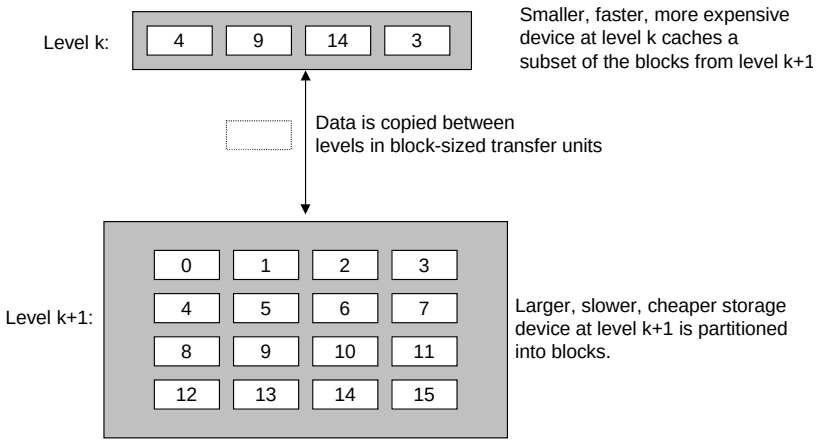


Figure 6.22: **The basic principle of caching in a memory hierarchy.**

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 is always copied back and forth between level k and level $k+1$ in block-sized *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 1-word blocks. Transfers between L2 and L1 (and L3 and L2) typically use blocks of 4 to 8 words. And transfers between L4 and L3 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 more restricted 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 4 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 and L2 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.

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.

Type	What cached	Where cached	Latency (cycles)	Managed by
CPU registers	4-byte word	On-chip CPU registers	0	Compiler
TLB	Address translations	On-chip TLB	0	Hardware MMU
L1 cache	32-byte block	On-chip L1 cache	1	Hardware
L2 cache	32-byte block	Off-chip L2 cache	10	Hardware
Virtual memory	4-KB page	Main memory	100	Hardware + OS
Buffer cache	Parts of files	Main memory	100	OS
Network buffer cache	Parts of files	Local disk	10,000,000	AFS/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, AFS: Andrew File System, NFS: Network File System.

6.4 Cache Memories

The memory hierarchies of early computer systems consisted of only three levels: CPU registers, main DRAM memory, and disk storage. However, because of the increasing gap between CPU and main memory, system designers were compelled to insert a small SRAM memory, called an *L1 cache* (Level 1 cache), between the CPU register file and main memory. In modern systems, the L1 cache is located on the CPU chip (i.e., it is an *on-chip cache*), as shown in Figure 6.24. The L1 cache can be accessed nearly as fast as the registers, typically in one or two clock cycles.

As the performance gap between the CPU and main memory continued to increase, system designers responded by inserting an additional cache, called an *L2 cache*, between the L1 cache and the main memory, that can be accessed in a few clock cycles. The L2 cache can be attached to the memory bus, or it can be attached to its own *cache bus*, as shown in Figure 6.24. Some high-performance systems, such as those based on the Alpha 21164, will even include an additional level of cache on the memory bus, called an *L3 cache*, which sits between the L2 cache and main memory in the hierarchy. While there is considerable variety in the arrangements, the general principles are the same.