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. Registers in the CPU hold the most frequently used data. Small, fast cache memories near 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 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.

In contrast to the uniform access times in our simple system model, memory access times on a real system can vary by factors of ten, or one hundred, or even one million. Unwary programmers who assume a flat, uniform memory risk significant and inexplicable performance slowdowns in their programs. On the other hand, wise programmers who understand the hierarchical nature of memory can use relatively simple techniques to produce efficient programs with fast average memory access times.

In this chapter, we look at the most basic storage technologies of SRAM memory, DRAM memory, and disks. We also introduce a fundamental property of programs known as *locality* and show how locality motivates the organization of memory as a hierarchy of devices. Finally, we focus on the design and performance impact of the cache memories that act as staging areas between the CPU and main memory, and show you how to use your understanding of locality and caching to make your programs run faster.

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 1000 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.

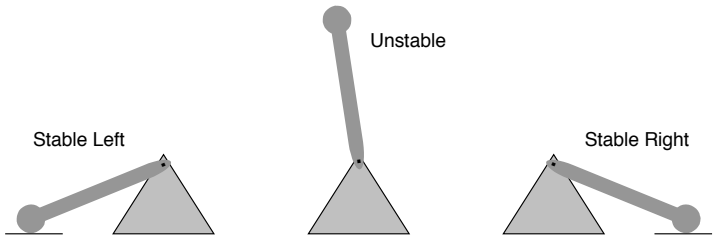


Figure 6.1: **Inverted pendulum.** Like an SRAM cell, the pendulum has only two stable configurations, or *states*.

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 tradeoff 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 2-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

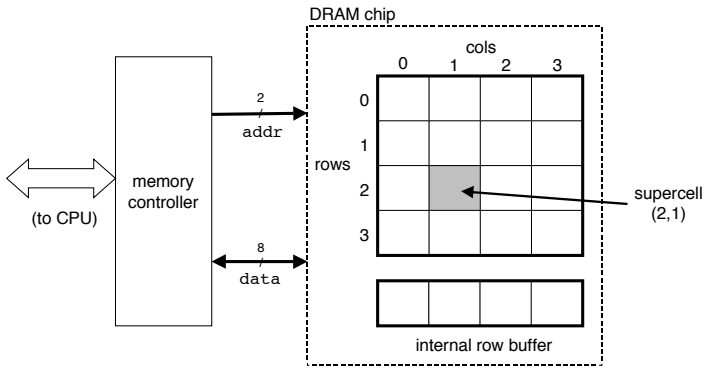
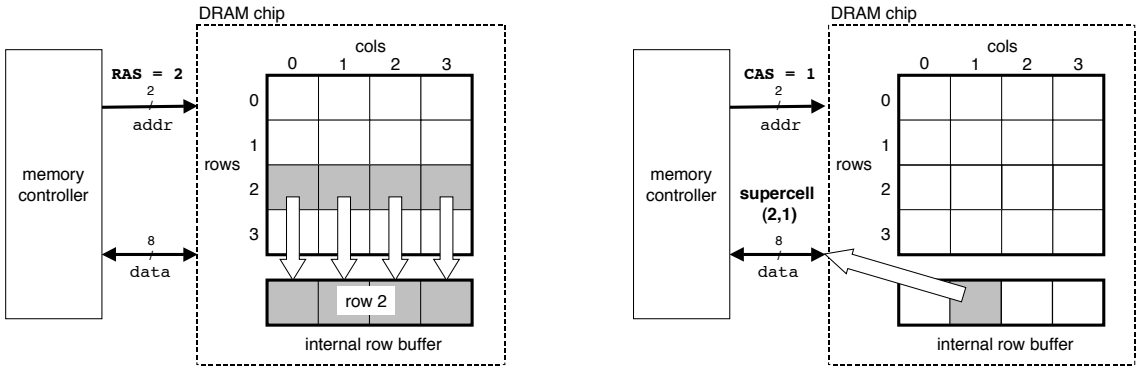


Figure 6.3: **High level view of a 128-bit 16×8 DRAM chip.**

"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.



(a) Select row 2 (RAS request). (b) Select column 1 (CAS request).

Figure 6.4: **Reading the contents of a DRAM cell.**

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.

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 our example in Figure 6.5, DRAM 0 stores the first (lower-order) byte, DRAM 1 stores the next byte, and so on.

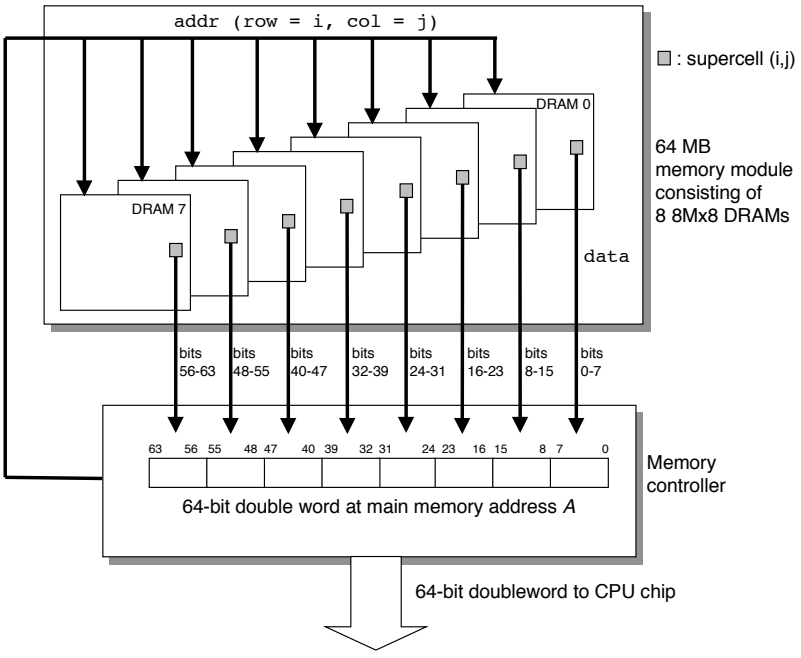


Figure 6.5: **Reading the contents of a memory module.**

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.

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

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.

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.

- *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 PC's were built with FPM DRAMs. From 1996-1999, EDO DRAMs dominated the market while FPM DRAMs all but disappeared. SDRAMs first appeared in 1995 in high-end systems, and by 2001 most PC's were built with 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 small transparent window on the outside of the chip that exposes the memory cells to outside light. The EPROM is reprogrammed by placing it in a special device that shines ultraviolet light onto the storage cells. An EPROM can be reprogrammed on the order of 1,000 times. An *electrically-erasable PROM (EEPROM)* is akin to an EPROM, but it has an internal structure that allows it to be reprogrammed electrically. Unlike EPROMs, EEPROMs do 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 comprise 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.

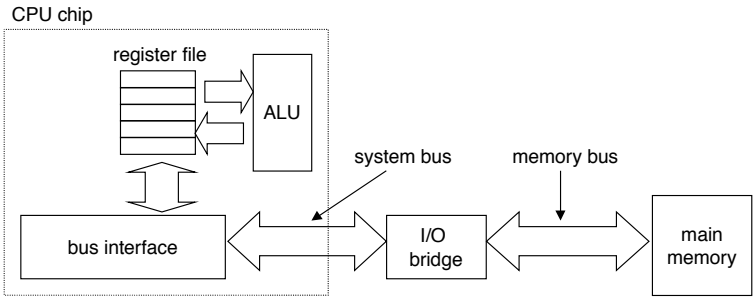


Figure 6.6: **Typical bus structure that connects the CPU and main memory.**

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

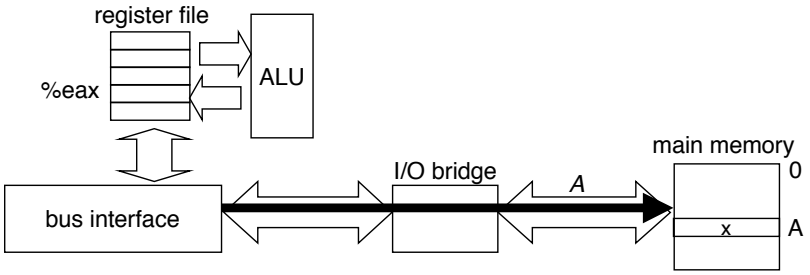
```
movl A,%eax
```

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)).

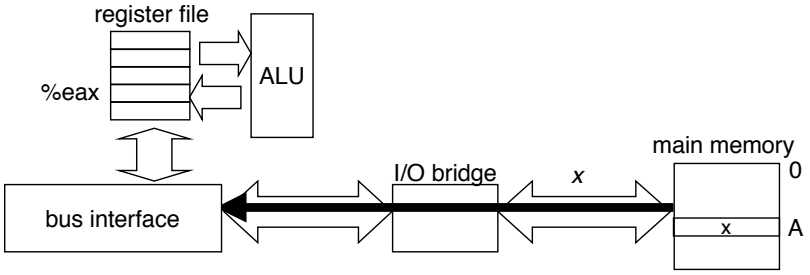
Conversely, when the CPU performs a store instruction such as

```
movl %eax,A
```

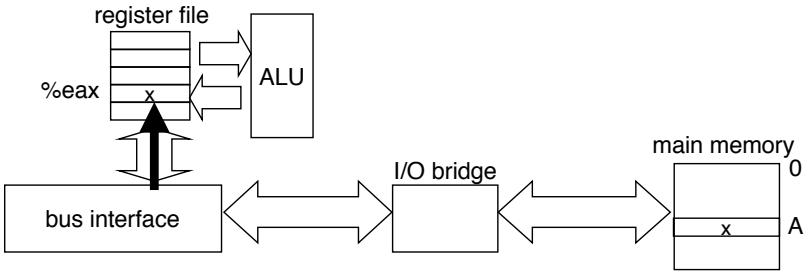
where the contents of register `%eax` are written to address A, the CPU initiates a write transaction. Again, there are three basic steps. First, the CPU places the address on the system bus. The memory reads the address from the memory bus and waits for the data to arrive (Figure 6.8(a)). Next, the 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)).



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

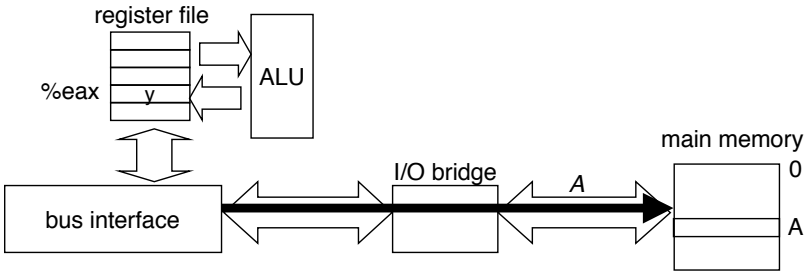


(b) Main memory reads A from the bus, retrieves word x, and places it on the bus.

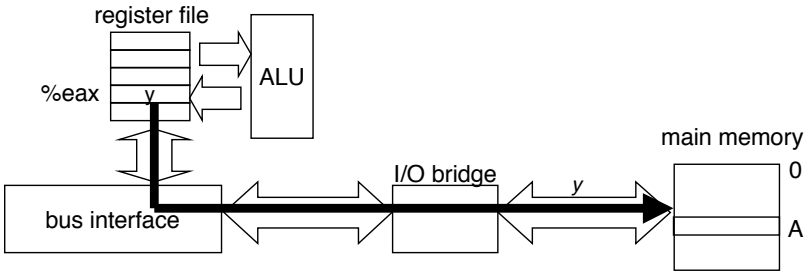


(c) CPU reads word x from the bus, and copies it into register `%eax`.

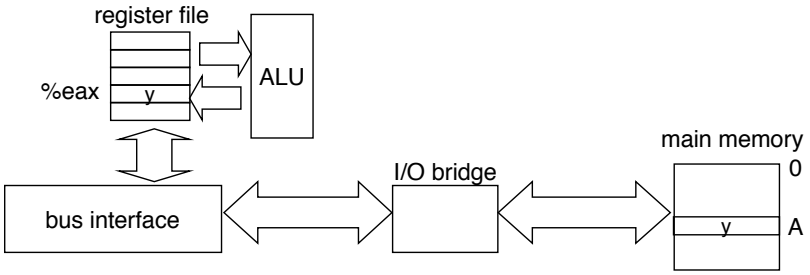
Figure 6.7: **Memory read transaction for a load operation:** `movl A,%eax`.



(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`**.**

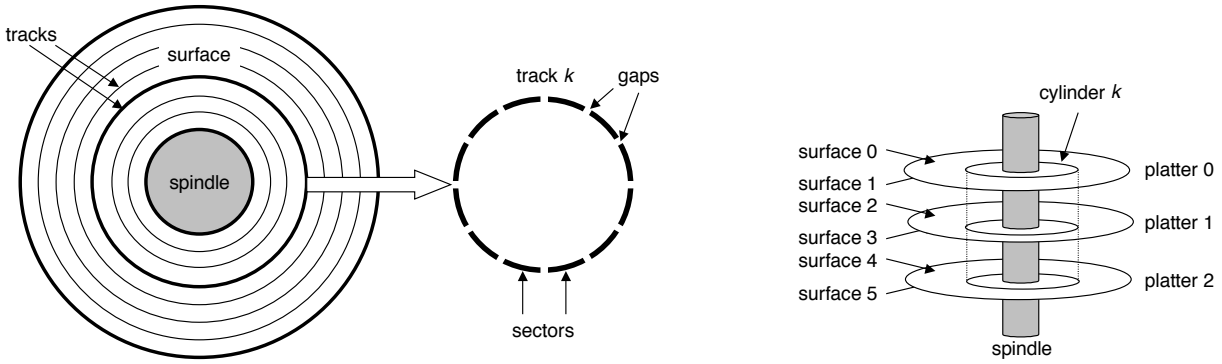
6.1.2 Disk Storage

Disks are workhorse storage devices that hold enormous amounts of data, on the order of 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) Single-platter view. (b) Multiple-platter view.

Figure 6.9: **Disk geometry.**

A disk consists of one or more platters stacked on top of each other and encased in a sealed package, as shown in Figure 6.9(b). The entire assembly is often referred to as a *disk drive*, although we will usually refer to it as simply a *disk*.

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.

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:

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

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.

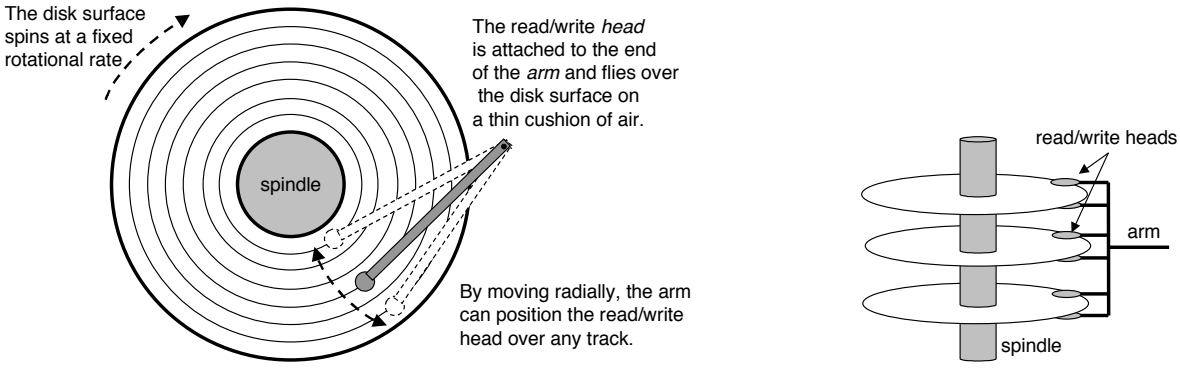
Fortunately, for the back-of-the-envelope estimates that we typically rely on, either assumption works fine in practice. For example, the relative difference between $2^{20} = 1,048,576$ and $10^6 = 1,000,000$ is small: $(2^{20} - 10^6)/10^6 \approx 5\%$. Similarly for $2^{30} = 1,073,741,824$ and $10^9 = 1,000,000,000$: $(2^{30} - 10^9)/10^9 \approx 7\%$. **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.



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

Figure 6.10: **Disk dynamics.**

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 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 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. So the maximum rotational latency in seconds is:

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

  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 a 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.

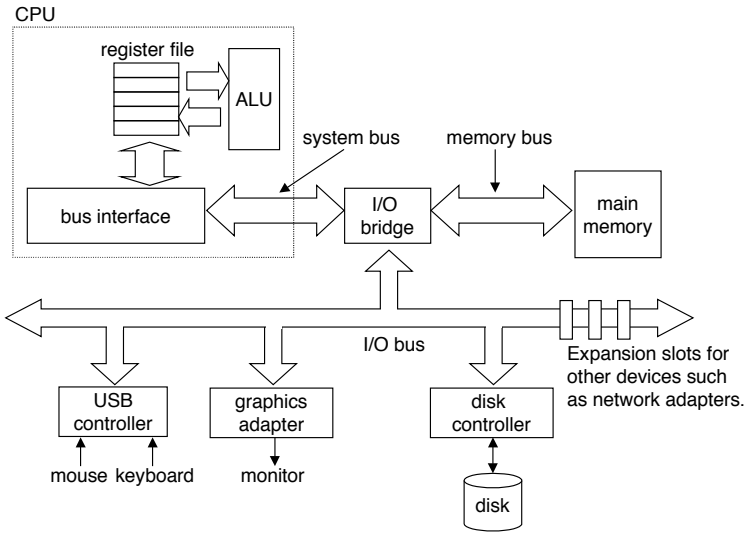


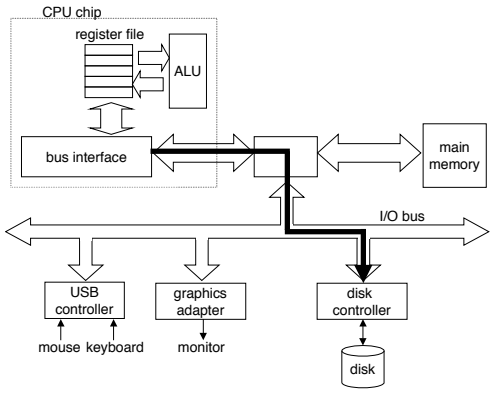
Figure 6.11: **Typical bus structure that connects the CPU, main memory, and I/O devices.**

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

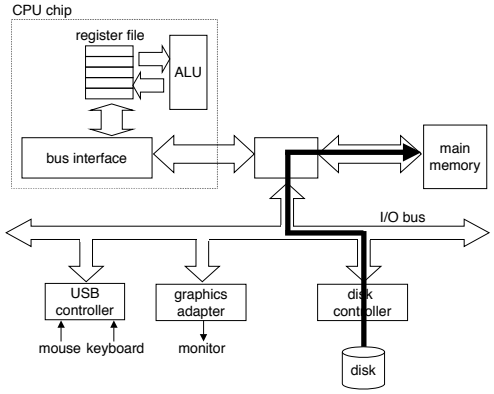
- A *Universal Serial Bus* (USB) controller is a conduit for devices attached to the USB. A USB has a throughput of 12 Mbits/s and is designed for slow to moderate speed serial devices such as keyboards, mice, modems, digital cameras, joysticks, CD-ROM drives, and printers.
- A *graphics card* (or *adapter*) 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.

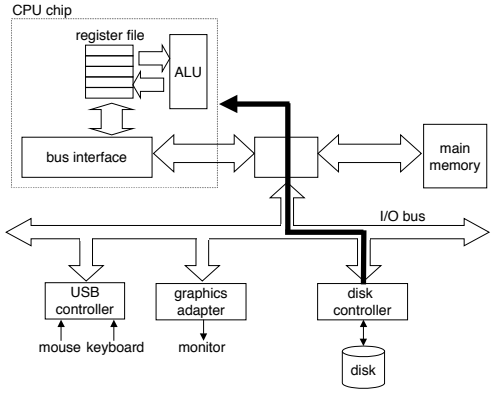
While a detailed description of how I/O devices work and how they are programmed is outside our scope, we can give you a general idea. For example, Figure 6.12 summarizes the steps that take place when a CPU reads data from a disk.



(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.**

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 where a device performs a read or write bus transaction on its own, without any involvement of the CPU, is known as *direct memory access (DMA)*. The transfer of data is known as a *DMA transfer*.

After the DMA transfer is complete and the contents of the disk sector are safely stored in main memory, the disk controller notifies the CPU by sending an interrupt signal to the CPU (Figure 6.12(c)). The basic idea is that an interrupt signals an external pin on the CPU chip. This causes the CPU to stop what it is currently working on and to 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 GBbytes

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 [64]. 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 [64].

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 and transparently 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.

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.**

- *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.

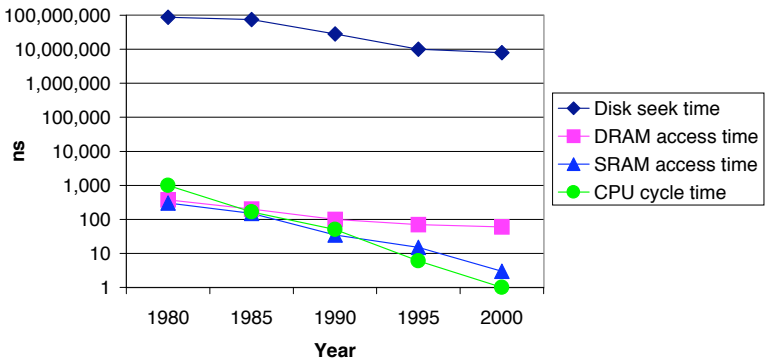


Figure 6.16: **The increasing gap between DRAM, disk, and CPU speeds.**

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.

6.2.1 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 that 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 runtime. 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++) {
8                 sum += a[k][i][j];
9             }
10        }
11    }
12    return sum;
13 }
```

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.

```
1 #define N 1000
2
3 typedef struct {
4     int vel[3];
5     int acc[3];
6 } point;
7
8 point p[N];
```

(a) An array of `structs`.

```
1 void clear1(point *p, int n)
2 {
3     int i, j;
4
5     for (i = 0; i < n; i++) {
6         for (j = 0; j < 3; j++)
7             p[i].vel[j] = 0;
8         for (j = 0; j < 3; j++)
9             p[i].acc[j] = 0;
10    }
11 }
```

(b) The `clear1` function.

```
1 void clear2(point *p, int n)
2 {
3     int i, j;
4
5     for (i = 0; i < n; i++) {
6         for (j = 0; j < 3; j++) {
7             p[i].vel[j] = 0;
8             p[i].acc[j] = 0;
9         }
10    }
11 }
```

(a) The `clear2` function.

```
1 void clear3(point *p, int n)
2 {
3     int i, j;
4
5     for (j = 0; j < 3; j++) {
6         for (i = 0; i < n; i++)
7             p[i].vel[j] = 0;
8         for (i = 0; i < n; i++)
9             p[i].acc[j] = 0;
10    }
11 }
```

(b) The `clear3` function.

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

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

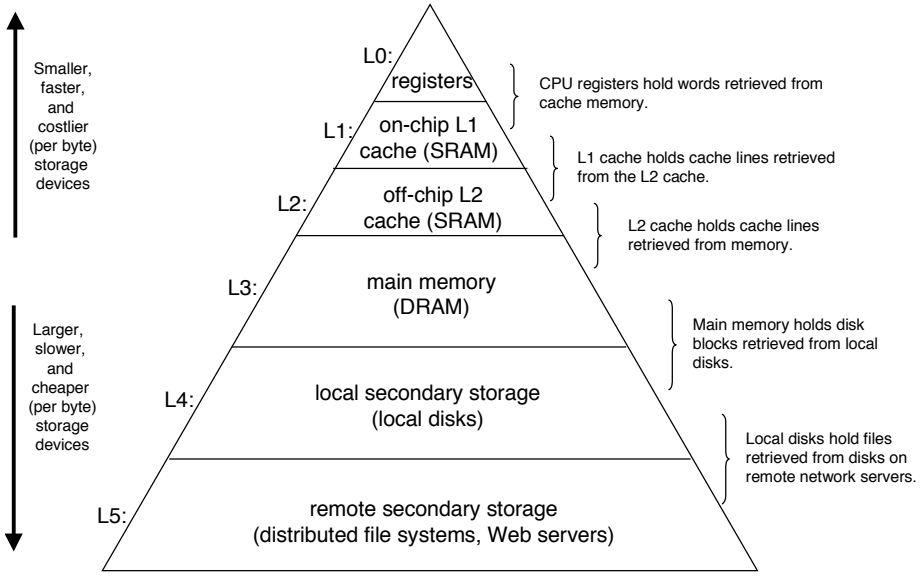


Figure 6.21: **The memory hierarchy.**

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 tradeoff 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.

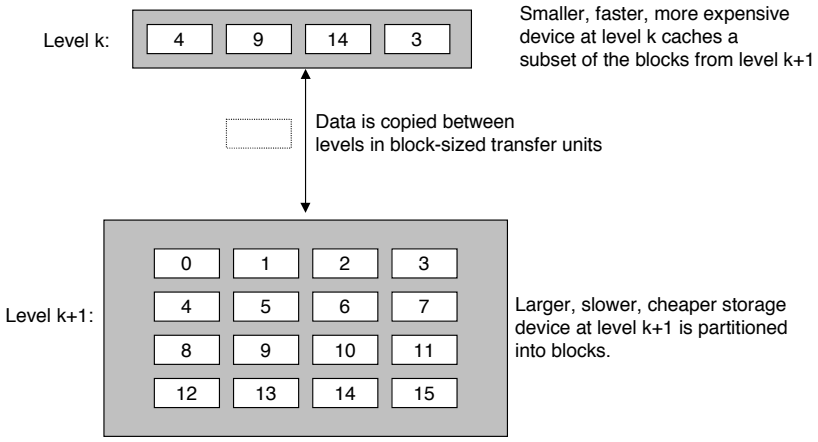


Figure 6.22: **The basic principle of caching in a memory hierarchy.**

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*, where 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
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.

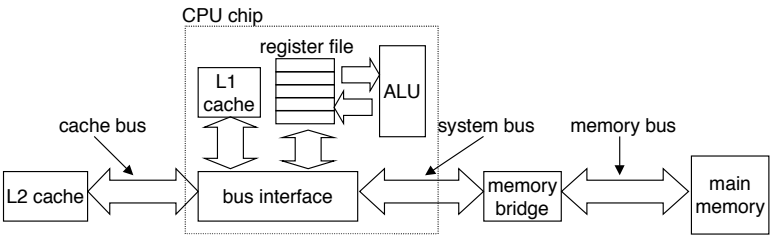
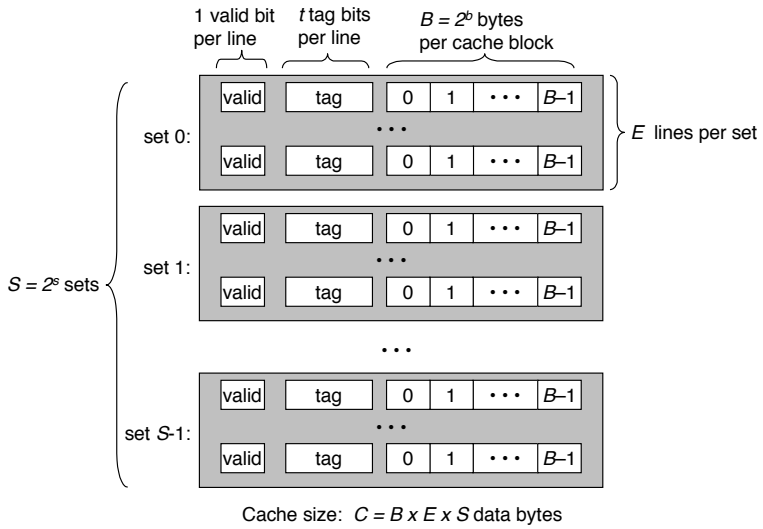


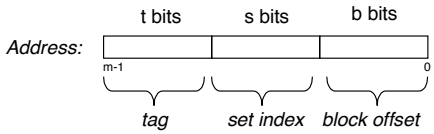
Figure 6.24: **Typical bus structure for L1 and L2 caches.**

6.4.1 Generic Cache Memory Organization

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



(a)



(b)

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

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

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

The parameters S and B induce a partitioning of the m address bits into the three fields shown in Figure 6.25(b). The s *set index bits* in A form an index into the array of S sets. The first set is set 0, the second set is set 1, and so on. When interpreted as an unsigned integer, the set index bits tell us which set the word must be stored in. Once we know which set the word must be contained in, the t tag bits in A tell us which line (if any) in the set contains the word. A line in the set contains the word if and only if the valid bit is set and the tag bits in the line match the tag bits in the address A. Once we have located the line identified by the tag in the set identified by the set index, then the b *block offset bits* give us the offset of the word in the B-byte data block.

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

Fundamental parameters	
Parameter	Description
$S = 2^s$	Number of sets
E	Number of lines per set
$B = 2^b$	Block size (bytes)
$m = \log_2(M)$	Number of physical (main memory) address bits

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

Figure 6.26: **Summary of cache parameters.**

Practice Problem 6.6:

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

Cache	m	C	B	E	S	t	s	b
1.	32	1024	4	1				
2.	32	1024	8	4				
3.	32	1024	32	32				

6.4.2 Direct-Mapped Caches

Caches are grouped into different classes based on E, the number of cache lines per set. A cache with exactly one line per set ($E = 1$) is known as a *direct-mapped* cache (see Figure 6.27). Direct-mapped

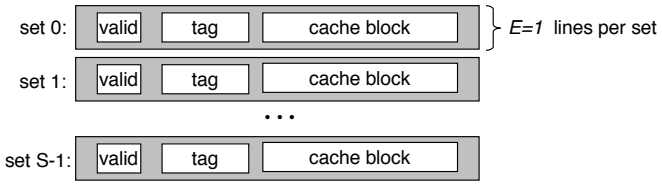


Figure 6.27: **Direct-mapped cache ($E = 1$).** There is exactly one line per set.

caches are the simplest both to implement and to understand, so we will use them to illustrate some general concepts about how caches work.

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

Set Selection in Direct-Mapped Caches

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

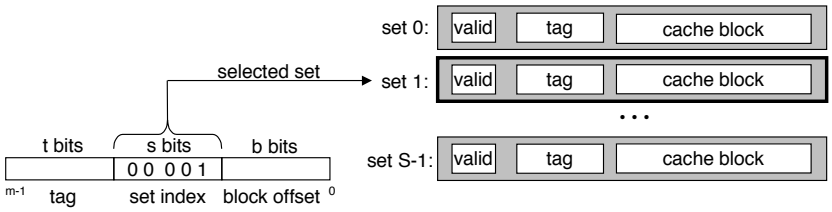


Figure 6.28: **Set selection in a direct-mapped cache.**

Line Matching in Direct-Mapped Caches

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

set and the tag in the cache line matches the tag in the address of w.

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

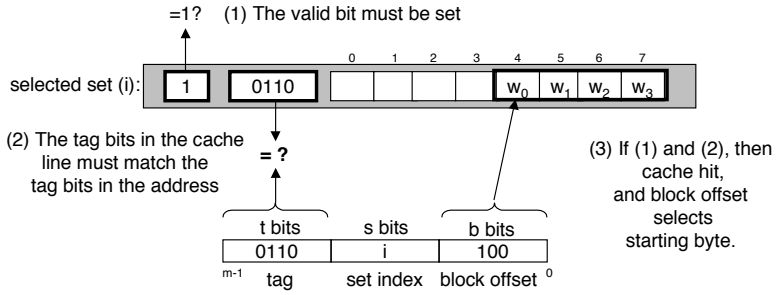


Figure 6.29: **Line matching and word selection in a direct-mapped cache.** Within the cache block, w_0 denotes the low-order byte of the word w, w_1 the next byte, and so on.

Word Selection in Direct-Mapped Caches

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

Line Replacement on Misses in Direct-Mapped Caches

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

Putting it Together: A Direct-Mapped Cache in Action

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

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

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

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

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

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

to notice about this enumerated space.

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

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

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

set	valid	tag	block[0]	block[1]
0	0			
1	0			
2	0			
3	0			

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

1. **Read word at address 0.** Since the valid bit for set 0 is zero, this is a cache miss. The cache fetches block 0 from memory (or a lower-level cache) and stores the block in set 0. Then the cache returns m[0] (the contents of memory location 0) from block[0] of the newly fetched cache line.

set	valid	tag	block[0]	block[1]
0	1	0	m[0]	m[1]
1	0			
2	0			
3	0			

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

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

set	valid	tag	block[0]	block[1]
0	1	0	m[0]	m[1]
1	0			
2	1	1	m[12]	m[13]
3	0			

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

set	valid	tag	block[0]	block[1]
0	1	1	m[8]	m[9]
1	0			
2	1	1	m[12]	m[13]
3	0			

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

set	valid	tag	block[0]	block[1]
0	1	0	m[0]	m[1]
1	0			
2	1	1	m[12]	m[13]
3	0			

Conflict Misses in Direct-Mapped Caches

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

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

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

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

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

At runtime, the first iteration of the loop references `x[0]`, a miss that causes the block containing `x[0]`–`x[3]` to be loaded into set 0. The next reference is to `y[0]`, another miss that causes the block containing `y[0]`–`y[3]` to be copied into set 0, overwriting the values of `x` that were copied in by the previous reference. During the next iteration, the reference to `x[1]` misses, which causes the `x[0]`–`x[3]` block to be

loaded back into set 0, overwriting the `y[0]`–`y[3]` block. So now we have a conflict miss, and in fact each subsequent reference to `x` and `y` will result in a conflict miss as we *thrash* back and forth between blocks of `x` and `y`. The term *thrashing* describes any situation where a cache is repeatedly loading and evicting the same sets of cache blocks.

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

Luckily, thrashing is easy for programmers to fix once they recognize what is going on. One easy solution is to put B bytes of padding at the end of each array. For example, instead of defining `x` to be `float x[8]`, we define it to be `float x[12]`. Assuming `y` starts immediately after `x` in memory, we have the following mapping of array elements to sets:

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

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

Practice Problem 6.7:

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

Why Index With the Middle Bits?

You may be wondering why caches use the middle bits for the set index instead of the high order bits. There is a good reason why the middle bits are better. Figure 6.31 shows why.

If the high-order bits are used as an index, then some contiguous memory blocks will map to the same cache set. For example, in the figure, the first four blocks map to the first cache set, the second four blocks map to the second set, and so on. If a program has good spatial locality and scans the elements of an array sequentially, then the cache can only hold a block-sized chunk of the array at any point in time. This is an inefficient use of the cache.

Contrast this with middle-bit indexing, where adjacent blocks always map to different cache lines. In this case, the cache can hold an entire C-sized chunk of the array, where C is the cache size.

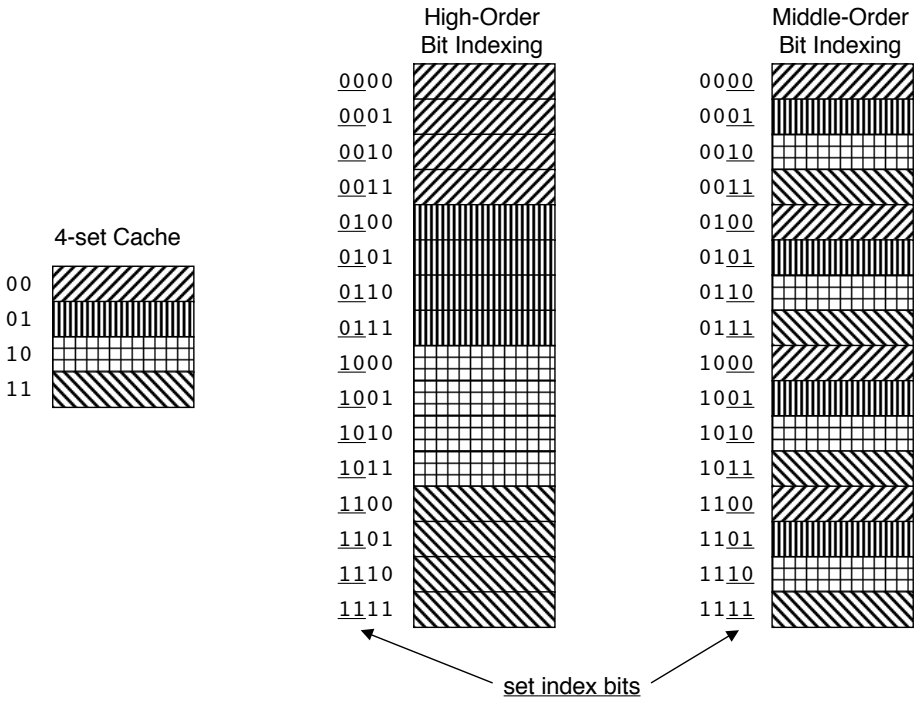


Figure 6.31: **Why caches index with the middle bits.**

Practice Problem 6.8:

In general, if the high-order s bits of an address are used as the set index, contiguous chunks of memory blocks are mapped to the same cache set.

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

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

```
int array[4096];

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

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

6.4.3 Set Associative Caches

The problem with conflict misses in direct-mapped caches stems from the constraint that each set has exactly one line (or in our terminology, $E = 1$). A *set associative cache* relaxes this constraint so each set holds more than one cache line. A cache with $1 < E < C/B$ is often called an E-way set associative cache. We will discuss the special case, where $E = C/B$, in the next section. Figure 6.32 shows the organization of a two-way set associative cache.

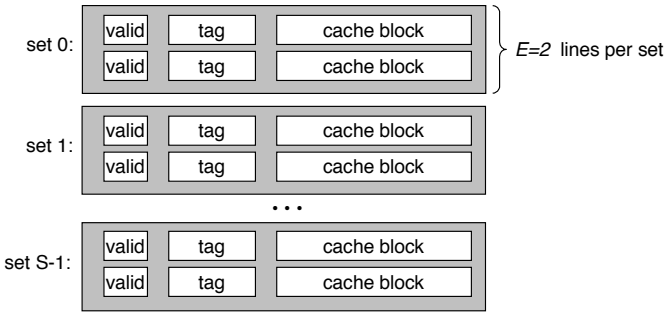


Figure 6.32: **Set associative cache ($1 < E < C/B$).** In a set associative cache, each set contains more than one line. This particular example shows a 2-way set associative cache.

Set Selection in Set Associative Caches

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

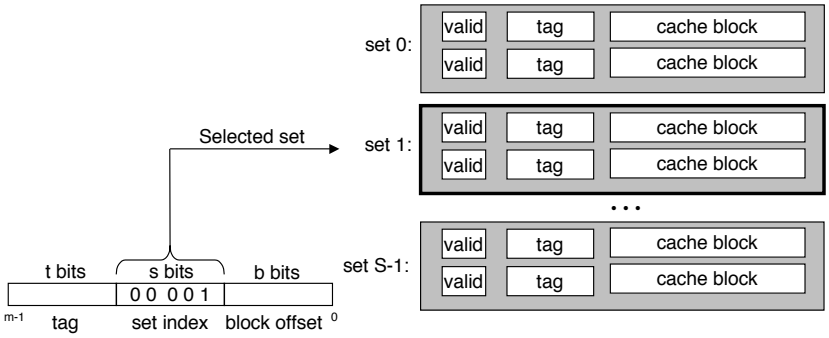


Figure 6.33: **Set selection in a set associative cache.**

Line Matching and Word Selection in Set Associative Caches

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

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

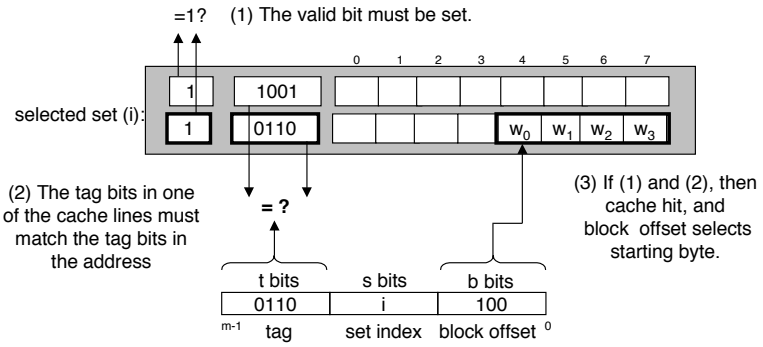


Figure 6.34: **Line matching and word selection in a set associative cache.**

Line Replacement on Misses in Set Associative Caches

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

It is very difficult for programmers to exploit knowledge of the cache replacement policy in their codes, so we will not go into much detail. The simplest replacement policy is to choose the line to replace at random. Other more sophisticated policies draw on the principle of locality to try to minimize the probability that the replaced line will be referenced in the near future. For example, a

least-frequently-used (LFU) policy will replace the line that has been referenced the fewest times over some past time window. A *least-recently-used (LRU)* policy will replace the line that was last accessed the furthest in the past. All of these policies require additional time and hardware. But as we move further down the memory hierarchy, away from the CPU, the cost of a miss becomes more expensive and it becomes more worthwhile to minimize misses with good replacement policies.

6.4.4 Fully Associative Caches

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

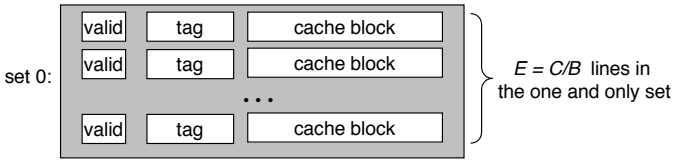


Figure 6.35: **Fully set associative cache ($E = C/B$).** In a fully associative cache, a single set contains all of the lines.

Set Selection in Fully Associative Caches

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

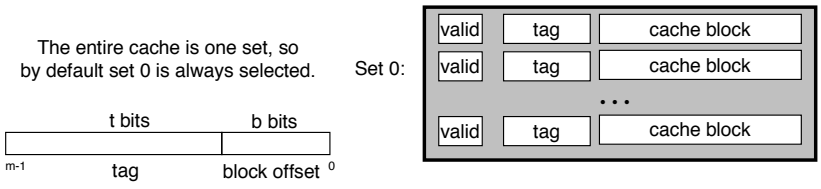


Figure 6.36: **Set selection in a fully associative cache.** Notice that there are no set index bits

Line Matching and Word Selection in Fully Associative Caches

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

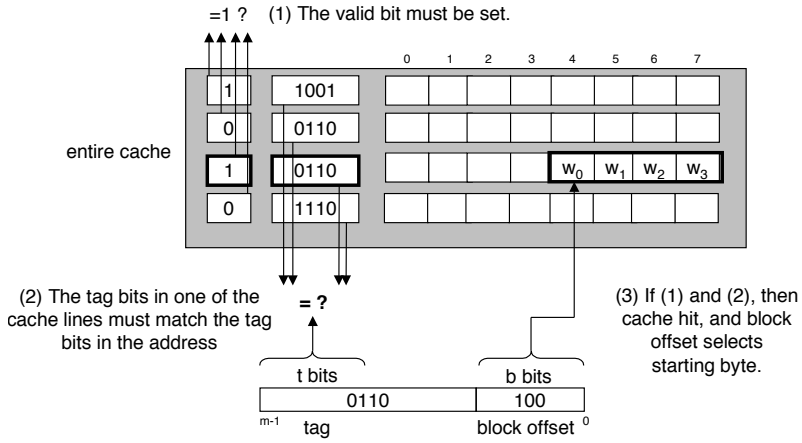


Figure 6.37: **Line matching and word selection in a fully associative cache.**

must search for many matching tags in parallel, it is difficult and expensive to build an associative cache that is both large and fast. As a result, fully associative caches are only appropriate for small caches, such as the translation lookaside buffers (TLBs) in virtual memory systems that cache page table entries (Section 10.6.2).

Practice Problem 6.9:

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

- The memory is byte addressable.
- Memory accesses are to **1-byte words** (not 4-byte words).

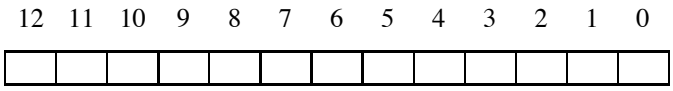
- Addresses are 13 bits wide.
- The cache is 2-way set associative ($E = 2$), with a 4-byte block size ($B = 4$) and 8 sets ($S = 8$).

The contents of the cache are as follows. All numbers are given in hexadecimal notation.

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

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

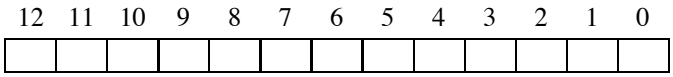
CO The cache block offset
CI The cache set index
CT The cache tag



Practice Problem 6.10:

Suppose a program running on the machine in Problem 6.9 references the 1-byte word at address `0x0E34`. Indicate the cache entry accessed and the cache byte value returned **in hex**. Indicate whether a cache miss occurs. If there is a cache miss, enter "–" for "Cache byte returned".

A. Address format (one bit per box):



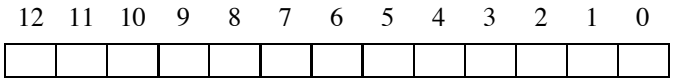
B. Memory reference:

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

Practice Problem 6.11:

Repeat Problem 6.10 for memory address `0x0DD5`.

A. Address format (one bit per box):



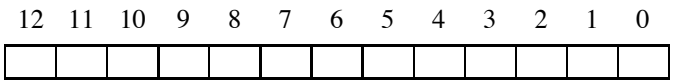
B. Memory reference:

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

Practice Problem 6.12:

Repeat Problem 6.10 for memory address `0x1FE4`.

A. Address format (one bit per box):



B. Memory reference:

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

Practice Problem 6.13:

For the cache in Problem 6.9, list all of the hex memory addresses that will hit in Set 3.

6.4.5 Issues with Writes

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

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

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

Optimizing caches for writes is a subtle and difficult issue, and we are only touching the surface here. The details vary from system to system and are often proprietary and poorly documented. To the programmer trying to write reasonably cache-friendly programs, we suggest adopting a mental model that assumes write-back write-allocate caches. There are several reasons for this suggestion.

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

6.4.6 Instruction Caches and Unified Caches

So far, we have assumed that caches hold only program data. But in fact, caches can hold instructions as well as data. A cache that holds instructions only is known as an *i-cache*. A cache that holds program data only is known as a *d-cache*. A cache that holds both instructions and data is known as a *unified cache*. A typical desktop systems includes an L1 i-cache and an L1 d-cache on the CPU chip itself, and a separate off-chip L2 unified cache. Figure 6.38 summarizes the basic setup.

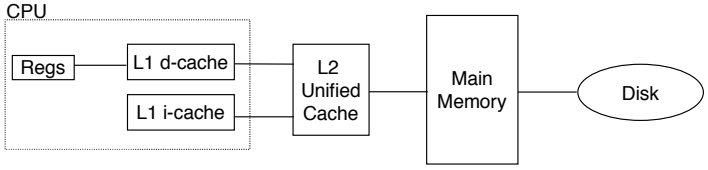


Figure 6.38: **A typical multi-level cache organization.**

Some higher-end systems, such as those based on the Alpha 21164, put the L1 and L2 caches on the CPU chip and have an additional off-chip L3 cache. Modern processors include separate on-chip i-caches and d-caches in order to improve performance. With two separate caches, the processor can read an instruction word and a data word during the same cycle. To our knowledge, no system incorporates an L4 cache, although as processor and memory speeds continue to diverge, it is likely to happen.

Aside: What kind of cache organization does a real system have?
Intel Pentium systems use the cache organization shown in Figure 6.38, with an on-chip L1 i-cache, an on-chip L1 d-cache, and an off-chip unified L2 cache. Figure 6.39 summarizes the basic parameters of these caches. **End Aside.**

Cache type	Associativity (E)	Block size (B)	Sets (S)	Cache size (C)
on-chip L1 i-cache	4	32 B	128	16 KB
on-chip L1 d-cache	4	32 B	128	16 KB
off-chip L2 unified cache	4	32 B	1024–16384	128 KB–2 MB

Figure 6.39: **Intel Pentium cache organization.**

6.4.7 Performance Impact of Cache Parameters

Cache performance is evaluated with a number of metrics:

- *Miss rate*. The fraction of memory references during the execution of a program, or a part of a program, that miss. It is computed as *# misses/# references*.
- *Hit rate*. The fraction of memory references that hit. It is computed as $1 - $ *miss rate*.
- *Hit time*. The time to deliver a word in the cache to the CPU, including the time for set selection, line identification, and word selection. Hit time is typically 1 to 2 clock cycle for L1 caches.
- *Miss penalty.* Any additional time required because of a miss. The penalty for L1 misses served from L2 is typically 5 to 10 cycles. The penalty for L1 misses served from main memory is typically 25 to 100 cycles.

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

Impact of Cache Size

On the one hand, a larger cache will tend to increase the hit rate. On the other hand, it is always harder to make big memories run faster. So larger caches tend to decrease the hit time. This is especially important for on-chip L1 caches that must have a hit time of one clock cycle.

Impact of Block Size

Large blocks are a mixed blessing. On the one hand, larger blocks can help increase the hit rate by exploiting any spatial locality that might exist in a program. However, for a given cache size, larger blocks imply a smaller number of cache lines, which can hurt the hit rate in programs with more temporal locality than spatial locality. Larger blocks also have a negative impact on the miss penalty, since larger blocks cause larger transfer times. Modern systems usually compromise with cache blocks that contain 4 to 8 words.

Impact of Associativity

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

The choice of associativity ultimately boils down to a trade-off between the hit time and the miss penalty. Traditionally, high-performance systems that pushed the clock rates would opt for direct-mapped L1 caches (where the miss penalty is only a few cycles) and a small degree of associativity (say 2 to 4) for the lower levels. But there are no hard and fast rules. In Intel Pentium systems, the L1 and L2 caches are all four-way set associative. In Alpha 21164 systems, the L1 instruction and data caches are direct-mapped, the L2 cache is three-way set associative, and the L3 cache is direct-mapped.

Impact of Write Strategy

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

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

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

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

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

6.5 Writing Cache-friendly Code

In Section 6.2 we introduced the idea of locality and talked in general terms about what constitutes good locality. But now that we understand how cache memories work, we can be more precise. Programs with better locality will tend to have lower miss rates, and programs with lower miss rates will tend to run faster than programs with higher miss rates. Thus, good programmers should always try to write code that is *cache-friendly*, in the sense that it has good locality. Here is the basic approach we use to try to ensure that our code is cache-friendly.

1. *Make the common case go fast*. Programs often spend most of their time in a few core functions. These functions often spend most of their time in a few loops. So focus on the inner loops of the core functions and ignore the rest.

2. *Minimize the number of cache misses in each inner loop*. All other things being equal, such as the total number of loads and stores, loops with better miss rates will run faster.

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

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

Is this function cache-friendly? First, notice that there is good temporal locality in the loop body with respect to the local variables `i` and `sum`. In fact, because these are local variables, any reasonable optimizing compiler will cache them in the register file, the highest level of the memory hierarchy. Now consider the stride-1 references to vector `v`. In general, if a cache has a block size of B bytes, then a stride-k reference pattern (where k is in expressed in words) results in an average of $\min(1, (wordsize \times k)/B)$ misses per loop iteration. This is minimized for $k = 1$, so the stride-1 references to `v` are indeed cache-friendly. For example, suppose that `v` is block-aligned, words are 4-bytes, cache blocks are 4 words, and the cache is initially empty (a cold cache). Then regardless of the cache organization, the references to `v` will result in the following pattern of hits and misses:

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

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

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

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

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

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

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

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

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

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

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

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

Higher miss rates can have a significant impact on running time. For example, on our desktop machine, `sumarraycols` runs in about 20 clock cycles per iteration, while `sumarrayrows` runs in about 10 cycles per iteration. To summarize, programmers should be aware of locality in their programs and try to write programs that exploit it.

Practice Problem 6.14:

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

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

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

- `sizeof(int) == 4`.

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

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

`dst` array		
	col 0	col 1
row 0	m	
row 1		

`src` array		
	col 0	col 1
row 0	m	
row 1		

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

Practice Problem 6.15:

The heart of the recent hit game *SimAquarium* is a tight loop that calculates the average position of 256 algae. You are evaluating its cache performance on a machine with a 1024-byte direct-mapped data cache with 16-byte blocks ($B = 16$). You are given the following definitions:

```
1 struct algae_position {
2     int x;
3     int y;
4 };
5
6 struct algae_position grid[16][16];
7 int total_x = 0, total_y = 0;
8 int i, j;
```

You should also assume:

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

Determine the cache performance for the following code:

```
1     for (i = 0; i < 16; i++) {
2         for (j = 0; j < 16; j++) {
3             total_x += grid[i][j].x;
4         }
5     }
```

```
6
7       for (i = 0; i < 16; i++) {
8           for (j = 0; j < 16; j++) {
9               total_y += grid[i][j].y;
10          }
11      }
```

A. What is the total number of reads? _____.

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

C. What is the miss rate? _____.

Practice Problem 6.16:

Given the assumptions of Problem 6.15, determine the cache performance of the following code:

```
1       for (i = 0; i < 16; i++){
2           for (j = 0; j < 16; j++) {
3               total_x += grid[j][i].x;
4               total_y += grid[j][i].y;
5           }
6       }
```

A. What is the total number of reads? _____.

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

C. What is the miss rate? _____.

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

Practice Problem 6.17:

Given the assumptions of Problem 6.15, determine the cache performance of the following code:

```
1       for (i = 0; i < 16; i++){
2           for (j = 0; j < 16; j++) {
3               total_x += grid[i][j].x;
4               total_y += grid[i][j].y;
5           }
6       }
```

A. What is the total number of reads? _____.

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

C. What is the miss rate? _____.

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

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

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

6.6.1 The Memory Mountain

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

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

code/mem/mountain/mountain.c

```
1  void test(int elems, int stride) /* The test function */
2  {
3      int i, result = 0;
4      volatile int sink;
5
6      for (i = 0; i < elems; i += stride)
7          result += data[i];
8      sink = result; /* So compiler doesn't optimize away the loop */
9  }
10
11 /* Run test(elems, stride) and return read throughput (MB/s) */
12 double run(int size, int stride, double Mhz)
13 {
14     double cycles;
15     int elems = size / sizeof(int);
16
17     test(elems, stride);                      /* warm up the cache */
18     cycles = fcyc2(test, elems, stride, 0);  /* call test(elems,stride) */
19     return (size / stride) / (cycles / Mhz); /* convert cycles to MB/s */
20 }
```

code/mem/mountain/mountain.c

Figure 6.40: **Functions that measure and compute read throughput.**

The `test` function generates the read sequence by scanning the first `elems` elements of an integer array with a stride of `stride`. The `run` function is a wrapper that calls the `test` function and returns the measured read throughput. The `fcyc2` function in line 18 (not shown) estimates the running time of the `test` function, in CPU cycles, using the K-best measurement scheme described in Chapter 9. Notice that the `size` argument to the `run` function is in units of bytes, while the corresponding `elems` argument to

the `test` function is in units of words. Also, notice that line 19 computes MB/s as 10^6 bytes/s, as opposed to 2^{20} bytes/s.

The `size` and `stride` arguments to the `run` function allow us to control the degree of locality in the resulting read sequence. Smaller values of `size` result in a smaller working set size, and thus more temporal locality. Smaller values of `stride` result in more spatial locality. If we call the `run` function repeatedly with different values of `size` and `stride`, then we can recover a two-dimensional function of read bandwidth versus temporal and spatial locality called the *memory mountain*. Figure 6.41 shows a program, called `mountain`, that generates the memory mountain.

code/mem/mountain/mountain.c

```
1  #include <stdio.h>
2  #include "fcyc2.h" /* K-best measurement timing routines */
3  #include "clock.h" /* routines to access the cycle counter */
4
5  #define MINBYTES (1 << 10)  /* Working set size ranges from 1 KB */
6  #define MAXBYTES (1 << 23)  /* ... up to 8 MB */
7  #define MAXSTRIDE 16        /* Strides range from 1 to 16 */
8  #define MAXELEMS MAXBYTES/sizeof(int)
9
10 int data[MAXELEMS];         /* The array we'll be traversing */
11
12 int main()
13 {
14     int size;        /* Working set size (in bytes) */
15     int stride;      /* Stride (in array elements) */
16     double Mhz;      /* Clock frequency */
17
18     init_data(data, MAXELEMS); /* Initialize each element in data to 1 */
19     Mhz = mhz(0);              /* Estimate the clock frequency */
20     for (size = MAXBYTES; size >= MINBYTES; size >>= 1) {
21         for (stride = 1; stride <= MAXSTRIDE; stride++) {
22             printf("%.1f\t", run(size, stride, Mhz));
23         }
24         printf("\n");
25     }
26     exit(0);
27 }
```

code/mem/mountain/mountain.c

Figure 6.41: `mountain`**: A program that generates the memory mountain.**

The `mountain` program calls the `run` function with different working set sizes and strides. Working set sizes start at 1 KB, increasing by a factor of two, to a maximum of 8 MB. Strides range from 1 to 16. For each combination of working set size and stride, `mountain` prints the read throughout, in units of MB/s. The `mhz` function in line 19 (not shown) is a system-dependent routine that estimates the CPU clock frequency, using techniques described in Chapter 9.

Every computer has a unique memory mountain that characterizes the capabilities of its memory system. For example, Figure 6.42 shows the memory mountain for an Intel Pentium III Xeon system.

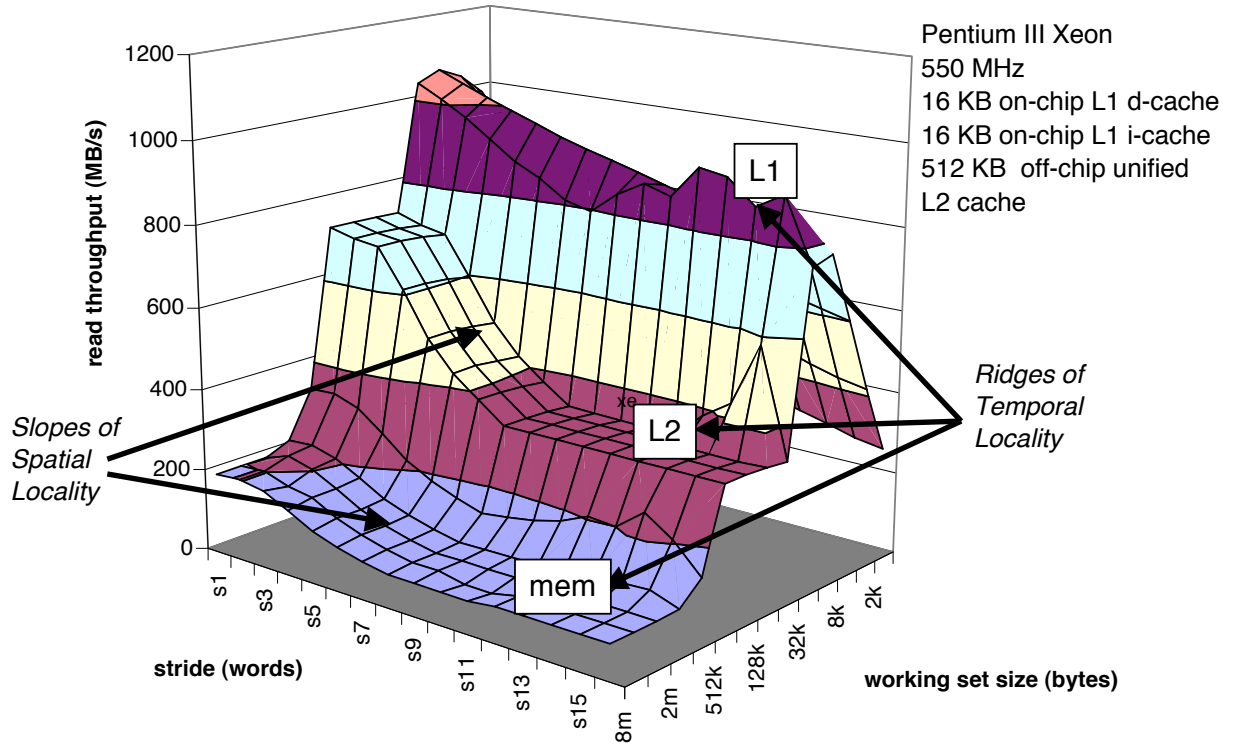


Figure 6.42: **The memory mountain.**

The geography of the Xeon mountain reveals a rich structure. Perpendicular to the `size` axis are three ridges that correspond to the regions of temporal locality where the working set fits entirely in the L1 cache, the L2 cache, and main memory respectively. Notice that there is an order of magnitude difference between the highest peak of the L1 ridge, where the CPU reads at a rate of 1 GB/s, and the lowest point of the main memory ridge, where the CPU reads at a rate of 80 MB/s.

There are two features of the L1 ridge that should be pointed out. First, for a constant stride, notice how the read throughput plummets as the working set size decreases from 16 KB to 1 KB (falling off the back side of the ridge). Second, for a working set size of 16 KB, the peak of the L1 ridge line decreases with increasing stride. Since the L1 cache holds the entire working set, these features do not reflect the true L1 cache performance. They are artifacts of overheads of calling the `test` function and setting up to execute the loop. For the small working set sizes along the L1 ridge, these overheads are not amortized, as they are with the larger working set sizes.

On the L2 and main memory ridges, there is a slope of spatial locality that falls downhill as the stride increases. This slope is steepest on the L2 ridge because of the large absolute miss penalty that the L2 cache suffers when it has to transfer blocks from main memory. Notice that even when the working set is too large to fit in either of the L1 or L2 caches, the highest point on the main memory ridge is a factor of two higher than its lowest point. So even when a program has poor temporal locality, spatial locality can still come to

the rescue and make a significant difference.

If we take a slice through the mountain, holding the stride constant as in Figure 6.43, we can see quite clearly the impact of cache size and temporal locality on performance. For sizes up to and including 16 KB, the working set fits entirely in the L1 d-cache, and thus reads are served from L1 at the peak throughput of about 1 GB/s. For sizes up to and including 256 KB, the working set fits entirely in the unified L2 cache.

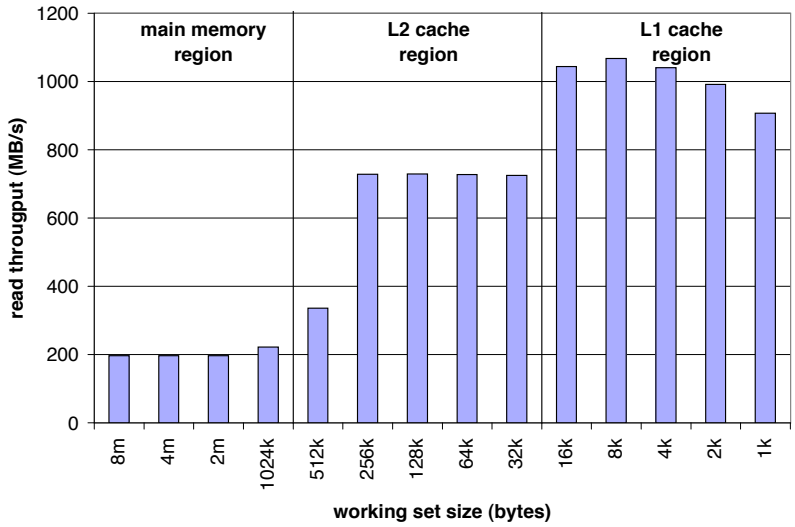


Figure 6.43: **Ridges of temporal locality in the memory mountain.** The graph shows a slice through Figure 6.42 with `stride=1`.

Larger working set sizes are served primarily from main memory. The drop in read throughput between 256 KB and 512 KB is interesting. Since the L2 cache is 512 KB, we might expect the drop to occur at 512 KB instead of 256 KB. The only way to be sure is to perform a detailed cache simulation, but we suspect the reason lies in the fact that the Pentium III L2 cache is a unified cache that holds both instructions and data. What we might be seeing is the effect of conflict misses between instructions and data in L2 that make it impossible for the entire array to fit in the L2 cache.

Slicing through the mountain in the opposite direction, holding the working set size constant, gives us some insight into the impact of spatial locality on the read throughput. For example, Figure 6.44 shows the slice for a fixed working set size of 256 KB. This slice cuts along the L2 ridge in Figure 6.42, where the working set fits entirely in the L2 cache, but is too large for the L1 cache. Notice how the read throughput decreases steadily as the stride increases from 1 to 8 words. In this region of the mountain, a read miss in L1 causes a block to be transferred from L2 to L1. This is followed by some number of hits on the block in L1, depending on the stride. As the stride increases, the ratio of L1 misses to L1 hits increases. Since misses are served slower than hits, the read throughput decreases. Once the stride reaches 8 words, which on this system equals the block size, every read request misses in L1 and must be served from L2. Thus the read throughput for strides of at least 8 words is a constant rate determined by the rate that cache blocks can be transferred from L2 into L1.

To summarize our discussion of the memory mountain: The performance of the memory system is not characterized by a single number. Instead, it is a mountain of temporal and spatial locality whose elevations

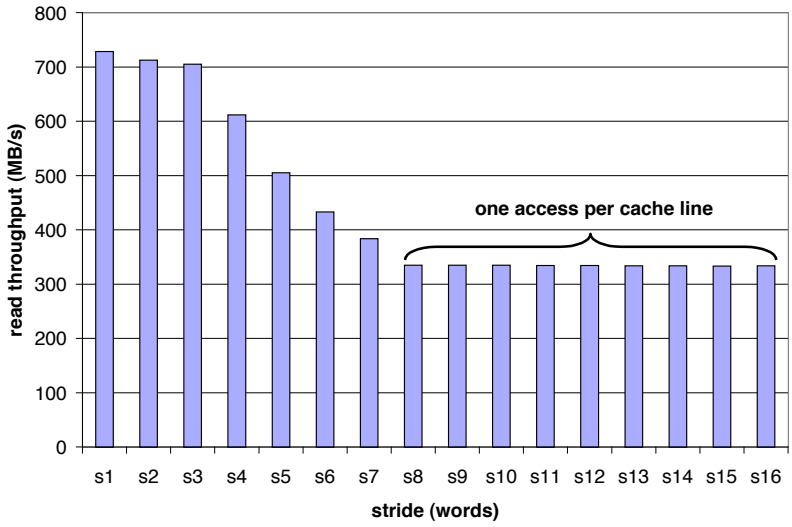


Figure 6.44: **A slope of spatial locality.** The graph shows a slice through Figure 6.42 with `size=256 KB`.

can vary by over an order of magnitude. Wise programmers try to structure their programs so that they run in the peaks instead of the valleys. The aim is to exploit temporal locality so that heavily used words are fetched from the L1 cache, and to exploit spatial locality so that as many words as possible are accessed from a single L1 cache line.

Practice Problem 6.18:

The memory mountain in Figure 6.42 has two axes: stride and working set size. Which axis corresponds to spatial locality? Which axis corresponds to temporal locality?

Practice Problem 6.19:

As programmers who care about performance, it is important for us to know rough estimates of the access times to different parts of the memory hierarchy. Using the memory mountain in Figure 6.42, estimate the time, in CPU cycles, to read a 4-byte word from:

A. The on-chip L1 d-cache.

B. The off-chip L2 cache.

C. Main memory.

Assume that the read throughput at (size=16M, stride=16) is 80 MB/s.

6.6.2 Rearranging Loops to Increase Spatial Locality

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

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

where

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

Matrix multiply is usually implemented using three nested loops, which are identified by their indexes i, j, and k. If we permute the loops and make some other minor code changes, we can create the six functionally equivalent versions of matrix multiply shown in Figure 6.45. Each version is uniquely identified by the ordering of its loops.

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

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

Figure 6.46 summarizes the results of our inner loop analysis. Notice that the six versions pair up into three equivalence classes, which we denote by the pair of matrices that are accessed in the inner loop. For example, versions ijk and jik are members of Class AB because they reference arrays A and B (but not C) in their innermost loop. For each class, we have counted the number of loads (reads) and stores (writes) in each inner loop iteration, the number of references to A, B, and C that will miss in the cache in each loop iteration, and the total number of cache misses per iteration.

The inner loops of the Class AB routines (Figure 6.45(a) and (b)) scan a row of array A with a stride of 1. Since each cache block holds four doublewords, the miss rate for A is 0.25 misses per iteration. On the other hand, the inner loop scans a column of B with a stride of n. Since n is large, each access of array B results in a miss, for a total of 1.25 misses per iteration.

The inner loops in the Class AC routines (Figure 6.45(c) and (d)) have some problems. Each iteration performs two loads and a store (as opposed to the Class AB routines, which perform 2 loads and no stores). Second, the inner loop scans the columns of A and C with a stride of n. The result is a miss on each load, for

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

code/mem/matmult/mm.c

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

code/mem/matmult/mm.c

(a) Version ijk.

code/mem/matmult/mm.c

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

code/mem/matmult/mm.c

(b) Version jik.

code/mem/matmult/mm.c

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

code/mem/matmult/mm.c

(c) Version jki.

code/mem/matmult/mm.c

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

code/mem/matmult/mm.c

(d) Version kji.

code/mem/matmult/mm.c

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

code/mem/matmult/mm.c

(e) Version kij.

code/mem/matmult/mm.c

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

code/mem/matmult/mm.c

(f) Version ikj.

Figure 6.45: **Six versions of matrix multiply.**

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

Figure 6.46: **Analysis of matrix multiply inner loops.** The six versions partition into three equivalence classes, denoted by the pair of arrays that are accessed in the inner loop.

a total of two misses per iteration. Notice that interchanging the loops has decreased the amount of spatial locality compared to the Class AB routines.

The BC routines (Figure 6.45(e) and (f)) present an interesting tradeoff. With two loads and a store, they require one more memory operation than the AB routines. On the other hand, since the inner loop scans both B and C row-wise with a stride-1 access pattern, the miss rate on each array is only 0.25 misses per iteration, for a total of 0.50 misses per iteration.

Figure 6.47 summarizes the performance of different versions of matrix multiply on a Pentium III Xeon system. The graph plots the measured number of CPU cycles per inner loop iteration as a function of array size (n).

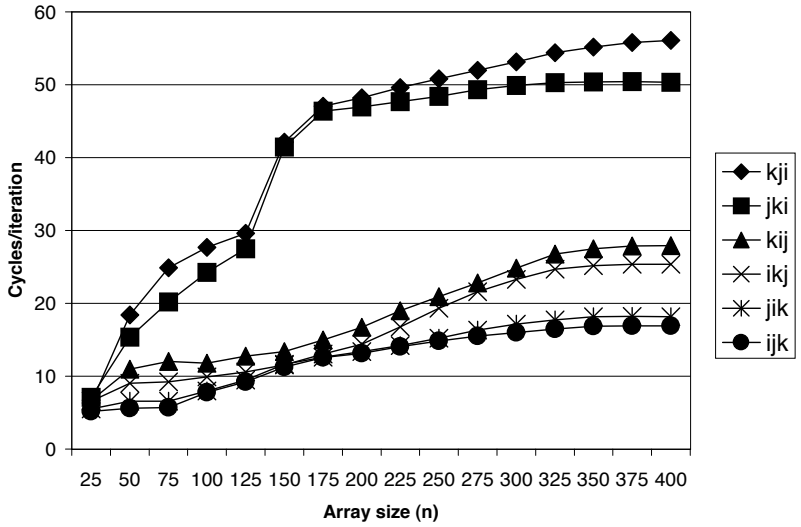


Figure 6.47: **Pentium III Xeon matrix multiply performance.** Legend: kji and jki: Class AC; kij and ikj: Class BC; ijk and jik: Class AB

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

- For large n, the fastest version runs three times faster than the slowest version, even though each performs the same number of floating-point arithmetic operations.
- Versions with the same number and locality of memory accesses have roughly the same measured performance.
- The two versions with the worst memory behavior, in terms of the number of accesses and misses per iteration, run significantly slower than the other four versions, which have fewer misses or fewer accesses, or both.
- The Class AB routines — 2 memory accesses and 1.25 misses per iteration — perform somewhat better on this particular machine than the Class BC routines — 3 memory accesses and 0.5 misses per iteration — which trade off an additional memory reference for a lower miss rate. The point is that cache misses are not the whole story when it comes to performance. The number of memory accesses

is also important, and in many cases, finding the best performance involves a tradeoff between the two. Problems 6.32 and 6.33 delve into this issue more deeply.

6.6.3 Using Blocking to Increase Temporal Locality

In the last section we saw how some simple rearrangements of the loops could increase spatial locality. But observe that even with good loop nestings, the time per loop iteration increases with increasing array size. What is happening is that as the array size increases, the temporal locality decreases, and the cache experiences an increasing number of capacity misses. To fix this, we can use a general technique called *blocking*. However, we must point out that, unlike the simple loop transformations for improving spatial locality, blocking makes the code harder to read and understand. For this reason it is best suited for optimizing compilers or frequently executed library routines. Still, the technique is interesting to study and understand because it is a general concept that can produce big performance gains.

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

Blocking a matrix multiply routine works by partitioning the matrices into submatrices and then exploiting the mathematical fact that these submatrices can be manipulated just like scalars. For example, if $n = 8$, then we could partition each matrix into four 4×4 submatrices:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

where

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

Figure 6.48 shows one version of blocked matrix multiplication, which we call the $bijk$ version. The basic idea behind this code is to partition A and C into $1 \times$ *bsize row slivers* and to partition B into *bsize* $\times$ *bsize* blocks. The innermost (j, k) loop pair multiplies a sliver of A by a block of B and accumulates the result into a sliver of C. The i loop iterates through n row slivers of A and C, using the same block in B.

Figure 6.49 gives a graphical interpretation of the blocked code from Figure 6.48. The key idea is that it loads a block of B into the cache, uses it up, and then discards it. References to A enjoy good spatial locality because each sliver is accessed with a stride of 1. There is also good temporal locality because the entire sliver is referenced $bsize$ times in succession. References to B enjoy good temporal locality because the entire $bsize \times bsize$ block is accessed n times in succession. Finally, the references to C have good spatial locality because each element of the sliver is written in succession. Notice that references to C do not have good temporal locality because each sliver is only accessed one time.

code/mem/matmult/bmm.c

```
1 void bijk(array A, array B, array C, int n, int bsize)
2 {
3     int i, j, k, kk, jj;
4     double sum;
5     int en = bsize * (n/bsize); /* Amount that fits evenly into blocks */
6
7     for (i = 0; i < n; i++)
8         for (j = 0; j < n; j++)
9             C[i][j] = 0.0;
10
11    for (kk = 0; kk < en; kk += bsize) {
12        for (jj = 0; jj < en; jj += bsize) {
13            for (i = 0; i < n; i++) {
14                for (j = jj; j < jj + bsize; j++) {
15                    sum = C[i][j];
16                    for (k = kk; k < kk + bsize; k++) {
17                        sum += A[i][k]*B[k][j];
18                    }
19                    C[i][j] = sum;
20                }
21            }
22        }
23    }
24 }
```

code/mem/matmult/bmm.c

Figure 6.48: **Blocked matrix multiply.** A simple version that assumes that the array size (`n`) is an integral multiple of the block size (`bsize`).

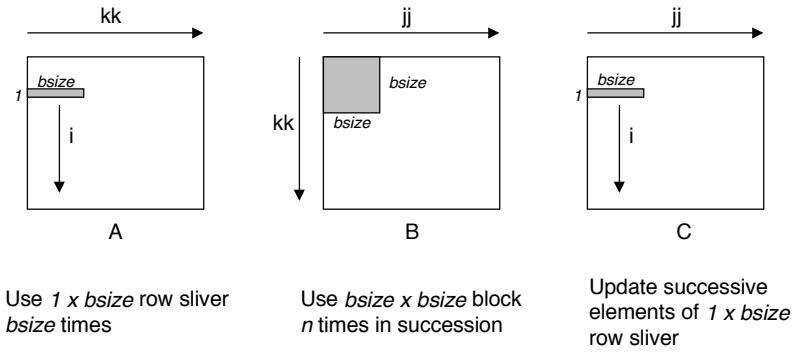


Figure 6.49: **Graphical interpretation of blocked matrix multiply** The innermost (j, k) loop pair multiplies a $1 \times bsize$ sliver of `A` by a $bsize \times bsize$ block of `B` and accumulates into a $1 \times bsize$ sliver of `C`.

Blocking can make code harder to read, but it can also pay big performance dividends. Figure 6.50 shows the performance of two versions of blocked matrix multiply on a Pentium III Xeon system ($bsize = 25$). Notice that blocking improves the running time by a factor of two over the best non-blocked version, from about 20 cycles per iteration down to about 10 cycles per iteration. The other interesting impact of blocking is that the time per iteration remains nearly constant with increasing array size. For small array sizes, the additional overhead in the blocked version causes it to run slower than the non-blocked versions. There is a crossover point, at about $n = 100$, after which the blocked version runs faster.

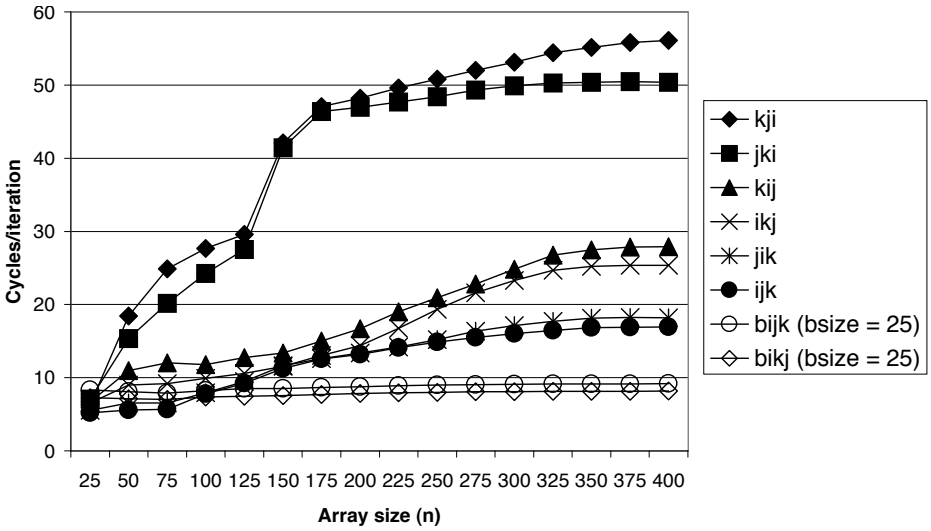


Figure 6.50: **Pentium III Xeon matrix multiply performance.** Legend: $bijk$ and $bikj$: two different versions of blocked matrix multiply. Performance of the unblocked versions from Figure 6.47 is shown for reference.

Aside: Caches and streaming media workloads

Applications that process network video and audio data in real time are becoming increasingly important. In these applications, the data arrive at the machine in a steady stream from some input device such as a microphone, a camera, or a network connection (see Chapter 12). As the data arrive, they are processed, sent to an output device, and eventually discarded to make room for newly arriving data.

How well suited is the memory hierarchy for these *streaming media* workloads? Since the data are processed sequentially as they arrive, we able to derive some benefit from spatial locality, as with our matrix multiply example from Section 6.6. However, since the data are processed once and then discarded, the amount of temporal locality is limited.

To address this problem, system designers and compiler writers have pursued a strategy known as *prefetching*. The idea is to hide the latency of cache misses by anticipating which blocks will be accessed in the near future, and then fetching these blocks into the cache beforehand using special machine instructions. If the prefetching is done perfectly, then each block is copied into the cache just before the program references it, and thus every load instruction results in a cache hit. Prefetching entails risks, though. Since prefetching traffic shares the bus with the DMA traffic that is streaming from an I/O device to main memory, too much prefetching might interfere with the DMA traffic and slow down overall system performance. Another potential problem is that every prefetched cache block must evict an existing block. If we do too much prefetching, we run the risk of *polluting the cache* by evicting a previously prefetched block that the program has not referenced yet, but will in the near future. **End Aside.**

6.7 Summary

The memory system is organized as a hierarchy of storage devices, with smaller, faster devices towards the top and larger, slower devices towards the bottom. Because of this hierarchy, the effective rate that a program can access memory locations is not characterized by a single number. Rather, it is a wildly varying function of program locality (what we have dubbed the memory mountain) that can vary by orders of magnitude. Programs with good locality access most of their data from fast L1 and L2 cache memories. Programs with poor locality access most of their data from the relatively slow DRAM main memory.

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

- Focus your attention on the inner loops where the bulk of the computations and memory accesses occur.
- Try to maximize the spatial locality in your programs by reading data objects sequentially, in the order they are stored in memory.
- Try to maximize the temporal locality in your programs by using a data object as often as possible once it has been read from memory.
- Remember that miss rates are only one (albeit important) factor that determines the performance of your code. The number of memory accesses also plays an important role, and sometimes it is necessary to trade off between the two.

Bibliographic Notes

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

Textbooks on circuit and logic design provide detailed information about memory technology [36, 58]. IEEE Spectrum published a series of survey articles on DRAM [33]. The International Symposium on Computer Architecture (ISCA) is a common forum for characterizations of DRAM memory performance [20, 21].

Wilkes wrote the first paper on cache memories [83]. Smith wrote a classic survey [68]. Przybylski wrote an authoritative book on cache design [56]. Hennessy and Patterson provide a comprehensive discussion of cache design issues [31].

Stricker introduced the idea of the memory mountain as a comprehensive characterization of the memory system in [78], and suggested the term "memory mountain" in later presentations of the work. Compiler researchers work to increase locality by automatically performing the kinds manual code transformations we discussed in Section 6.6 [13, 23, 42, 45, 51, 57, 85]. Carter and colleagues have proposed a cache-aware memory controller [10]. Seward developed an open-source cache profiler, called `cacheprof`, that characterizes the miss behavior of C programs on an arbitrary simulated cache (`www.cacheprof.org`).

There is a large body of literature on building and using disk storage. Many storage researchers look for ways to aggregate individual disks into larger, more robust, and more secure storage pools [11, 26, 27, 54, 86]. Others look for ways to use caches and locality to improve the performance of disk accesses [6, 12]. Systems such as Exokernel provide increased user-level control of disk and memory resources [35]. Systems such as the Andrew File System [50] and Coda [63] extend the memory hierarchy across computer networks and mobile notebook computers. Schindler and Ganger have developed an interesting tool that automatically characterizes the geometry and performance of SCSI disk drives [64].

Homework Problems

Homework Problem 6.20 [Category 2]:

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

Homework Problem 6.21 [Category 1]:

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

Cache	m	C	B	E	S	t	s	b
1.	32	1024	4	4				
2.	32	1024	4	256				
3.	32	1024	8	1				
4.	32	1024	8	128				
5.	32	1024	32	1				
6.	32	1024	32	4				

Homework Problem 6.22 [Category 1]:

This problem concerns the cache in Problem 6.9.

A. List all of the hex memory addresses that will hit in Set 1.

B. List all of the hex memory addresses that will hit in Set 6.

Homework Problem 6.23 [Category 2]:

Consider the following matrix transpose routine:

```
1 typedef int array[4][4];
2
```

```
 3 void transpose2(array dst, array src)
 4 {
 5     int i, j;
 6
 7     for (i = 0; i < 4; i++) {
 8         for (j = 0; j < 4; j++) {
 9             dst[j][i] = src[i][j];
10         }
11     }
12 }
```

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

- `sizeof(int) == 4`.
- The *src* array starts at address 0 and the *dst* array starts at address 64 (decimal).
- There is a single L1 data cache that is direct-mapped, write-through, write-allocate,with a block size of 16 bytes.
- The cache has a total size of 32 data bytes and the cache is initially empty.
- Accesses to the *src* and *dst* arrays are the only sources of read and write misses, respectively.

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

dst array				
	col 0	col 1	col 2	col 3
row 0	m			
row 1				
row 2				
row 3				

src array				
	col 0	col 1	col 2	col 3
row 0	m			
row 1				
row 2				
row 3				

Homework Problem 6.24 [Category 2]:

Repeat Problem 6.23 for a cache with a total size of 128 data bytes.

dst array				
	col 0	col 1	col 2	col 3
row 0				
row 1				
row 2				
row 3				

src array				
	col 0	col 1	col 2	col 3
row 0				
row 1				
row 2				
row 3				

Homework Problem 6.25 [Category 1]:

3M decides to make Post-Its by printing yellow squares on white pieces of paper. As part of the printing process, they need to set the CMYK (cyan, magenta, yellow, black) value for every point in the square. 3M hires you to determine the efficiency of the following algorithms on a machine with a 2048-byte direct-mapped data cache with 32-byte blocks. You are given the following definitions:

```
1 struct point_color {
2     int c;
3     int m;
4     int y;
5     int k;
6 };
7
8 struct point_color square[16][16];
9 int i, j;
```

Assume:

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

Determine the cache performance of the following code:

```
1     for (i = 0; i < 16; i++){
2         for (j = 0; j < 16; j++) {
3             square[i][j].c = 0;
4             square[i][j].m = 0;
5             square[i][j].y = 1;
6             square[i][j].k = 0;
7         }
8     }
```

A. What is the total number of writes? _____.

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

C. What is the miss rate? _____.

Homework Problem 6.26 [Category 1]:

Given the assumptions in Problem 6.25, determine the cache performance of the following code:

```
1     for (i = 0; i < 16; i++){
2         for (j = 0; j < 16; j++) {
3             square[j][i].c = 0;
4             square[j][i].m = 0;
5             square[j][i].y = 1;
6             square[j][i].k = 0;
7         }
8     }
```

A. What is the total number of writes? _____.

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

C. What is the miss rate? _____.

Homework Problem 6.27 [Category 1]:

Given the assumptions in Problem 6.25, determine the cache performance of the following code:

```
1     for (i = 0; i < 16; i++) {
2         for (j = 0; j < 16; j++) {
3             square[i][j].y = 1;
4         }
5     }
6     for (i = 0; i < 16; i++) {
7         for (j = 0; j < 16; j++) {
8             square[i][j].c = 0;
9             square[i][j].m = 0;
10            square[i][j].k = 0;
11        }
12    }
```

A. What is the total number of writes? _____.

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

C. What is the miss rate? _____.

Homework Problem 6.28 [Category 2]:

You are writing a new 3D game that you hope will earn you fame and fortune. You are currently working on a function to blank the screen buffer before drawing the next frame. The screen you are working with is a 640×480 array of pixels. The machine you are working on has a 64 KB direct-mapped cache with 4-byte lines. The C structures you are using are:

```
1  struct pixel {
2      char r;
3      char g;
4      char b;
5      char a;
6  };
7
8  struct pixel buffer[480][640];
9  int i, j;
10 char *cptr;
11 int *iptr;
```

Assume:

- `sizeof(char) == 1` and `sizeof(int) == 4`
- `buffer` begins at memory address 0.
- The cache is initially empty.
- The only memory accesses are to the entries of the array `buffer`. Variables `i`, `j`, `cptr`, and `iptr` are stored in registers.

What percentage of writes in the following code will miss in the cache?

```
1      for (j = 0; j < 640; j++) {
2          for (i = 0; i < 480; i++){
3              buffer[i][j].r = 0;
4              buffer[i][j].g = 0;
5              buffer[i][j].b = 0;
6              buffer[i][j].a = 0;
7          }
8      }
```

Homework Problem 6.29 [Category 2]:

Given the assumptions in Problem 6.28, what percentage of writes in the following code will miss in the cache?

```
1      char *cptr = (char *) buffer;
2      for (; cptr < (((char *) buffer) + 640 * 480 * 4); cptr++)
3          *cptr = 0;
```

Homework Problem 6.30 [Category 2]:

Given the assumptions in Problem 6.28, what percentage of writes in the following code will miss in the cache?

```
    int *iptr = (int *)buffer;
    for (; iptr < ((int *)buffer + 640*480); iptr++)
        *iptr = 0;
```

Homework Problem 6.31 [Category 3]:

Download the `mountain` program from the CS:APP Web site and run it on your favorite PC/Linux system. Use the results to estimate the sizes of the L1 and L2 caches on your system.

Homework Problem 6.32 [Category 4]:

In this assignment you will apply the concepts you learned in Chapters 5 and 6 to the problem of optimizing code for a memory intensive application. Consider a procedure to copy and transpose the elements of an $N \times N$ matrix of type `int`. That is, for source matrix S and destination matrix D, we want to copy each element $s_{i,j}$ to $d_{j,i}$. This code can be written with a simple loop:

```
void transpose(int *dst, int *src, int dim)
{
    int i, j;

    for (i = 0; i < dim; i++)
        for (j = 0; j < dim; j++)
            dst[j*dim + i] = src[i*dim + j];
}
```

where the arguments to the procedure are pointers to the destination (`dst`) and source (`src`) matrices, as well as the matrix size N (`dim`). Making this code run fast requires two types of optimizations. First, although the routine does a good job exploiting the spatial locality of the source matrix, it does a poor job for large values of N with the destination matrix. Second, the code generated by GCC is not very efficient. Looking at the assembly code, one sees that the inner loop requires 10 instructions, 5 of which reference memory—one for the source, one for the destination, and three to read local variables from the stack. Your job is to address these problems and devise a transpose routine that runs as fast as possible.

Homework Problem 6.33 [Category 4]:

This assignment is an intriguing variation of Problem 6.32. Consider the problem of converting a directed graph g into its undirected counterpart g'. The graph g' has an edge from vertex u to vertex v iff there is an edge from u to v or from v to u in the original graph g. The graph g is represented by its *adjacency matrix* G as follows. If N is the number of vertices in g then G is an $N \times N$ matrix and its entries are all either 0 or 1. Suppose the vertices of g are named $v_0, v_1, v_2, ..., v_{N-1}$. Then $G[i][j]$ is 1 if there is an edge from v_i to v_j and 0 otherwise. Observe, that the elements on the diagonal of an adjacency matrix are always 1 and that the adjacency matrix of an undirected graph is symmetric. This code can be written with a simple loop:

```
void col_convert(int *G, int dim) {
     int i, j;

     for (i = 0; i < dim; i++)
         for (j = 0; j < dim; j++)
             G[j*dim + i] = G[j*dim + i] || G[i*dim + j];
}
```

Your job is to devise a conversion routine that runs as fast as possible. As before, you will need to apply concepts you learned in Chapters 5 and 6 to come up with a good solution.