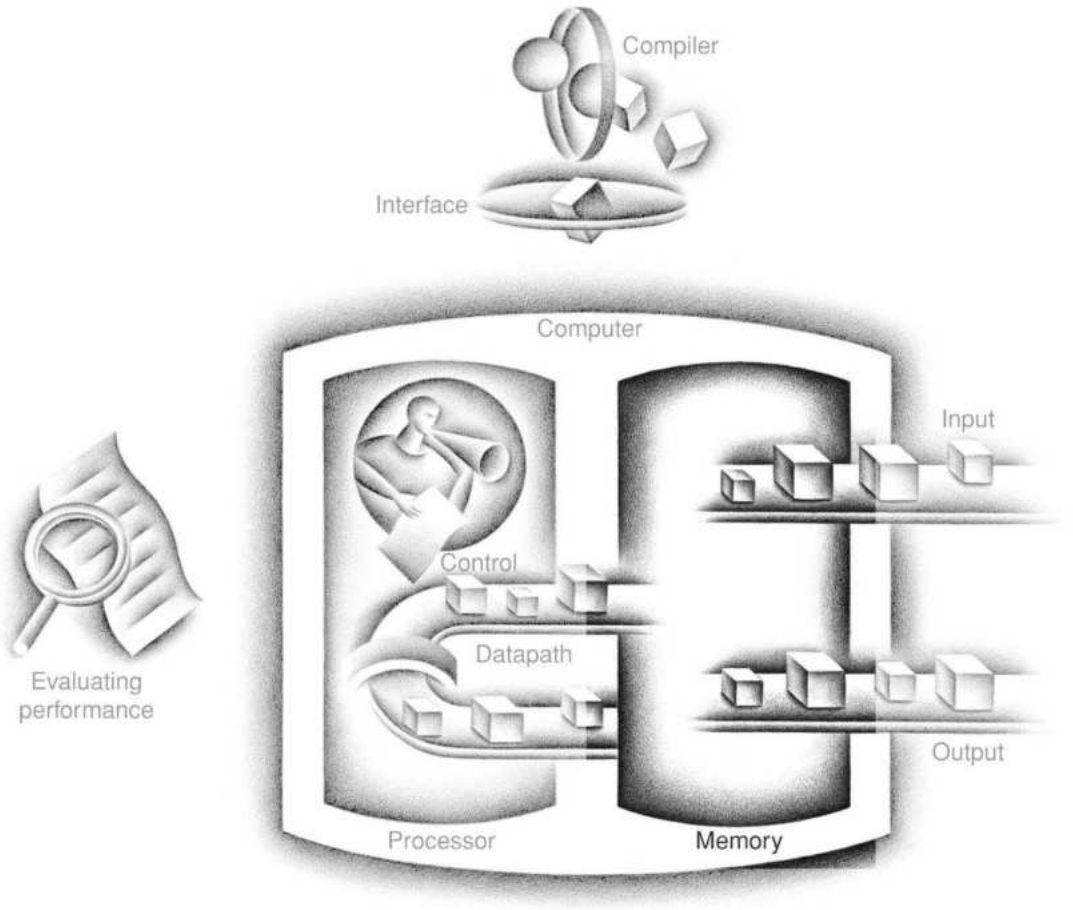


The Five Classic Components of a Computer

5.1 Introduction

From the earliest days of computing, programmers have wanted unlimited amounts of fast memory. The topics in this chapter aid programmers by creating that illusion. Before we look at creating the illusion, let's consider a simple analogy that illustrates the key principles and mechanisms that we use.

Suppose you were a student writing a term paper on important historical developments in computer hardware. You are sitting at a desk in a library with a collection of books that you have pulled from the shelves and are examining. You find that several of the important computers that you need to write about are described in the books you have, but there is nothing about the EDSAC. Therefore, you go back to the shelves and look for an additional book. You find a book on early British computers that covers the

EDSAC. Once you have a good selection of books on the desk in front of you, there is a high probability that many of the topics you need can be found in them, and you may spend most of your time just using the books on the desk without returning to the shelves. Having several books on the desk in front of you saves time compared to having only one book there and constantly having to go back to the shelves to return it and take out another.

The same principle allows us to create the illusion of a large memory that we can access as fast as a very small memory. Just as you did not need to access all the books in the library at once with equal probability, a program does not access all of its code or data at once with equal probability. Otherwise, it would be impossible to make most memory accesses fast and still have large memory in computers, just as it would be impossible for you to fit all the library books on your desk and still find what you wanted quickly.

This *principle of locality* underlies both the way in which you did your work in the library and the way that programs operate. The principle of locality states that programs access a relatively small portion of their address space at any instant of time, just as you accessed a very small portion of the library's collection. There are two different types of locality:

- **Temporal locality** (locality in time): if an item is referenced, it will tend to be referenced again soon. If you recently brought a book to your desk to look at, you will probably need to look at it again soon.
- **Spatial locality** (locality in space): if an item is referenced, items whose addresses are close by will tend to be referenced soon. For example, when you brought out the book on early English computers to learn about the EDSAC, you also noticed that there was another book shelved next to it about early mechanical computers, so you likewise brought back that book and, later on, found something useful in that book. Libraries put books on the same topic together on the same shelves to increase spatial locality. We'll see how memory hierarchies use spatial locality a little later in this chapter.

temporal locality

The locality principle stating that if a data location is referenced

then it will tend to be referenced again soon.

spatial locality

The locality principle stating that if a data location is referenced, data locations with nearby addresses will tend to be referenced soon.

Just as accesses to books on the desk naturally exhibit locality, locality in programs arises from simple and natural program structures. For example, most programs contain loops, so instructions and data are likely to be accessed repeatedly, showing large temporal locality. Since instructions are normally accessed sequentially, programs also show high spatial locality. Accesses to data also exhibit a natural spatial locality. For example, sequential accesses to elements of an array or a record will naturally have high degrees of spatial locality.

We take advantage of the principle of locality by implementing the memory of a computer as a **memory hierarchy**. A memory hierarchy consists of multiple levels of memory with different speeds and sizes. The faster memories are more expensive per bit than the slower memories and thus are smaller.

memory hierarchy

A structure that uses multiple levels of memories; as the distance from the processor increases, the size of the memories and the access time both increase.

Figure 5.1 shows the faster memory is close to the processor and the slower, less expensive memory is below it. The goal is to present the user with as much memory as is available in the cheapest technology, while providing access at the speed offered by the fastest memory.

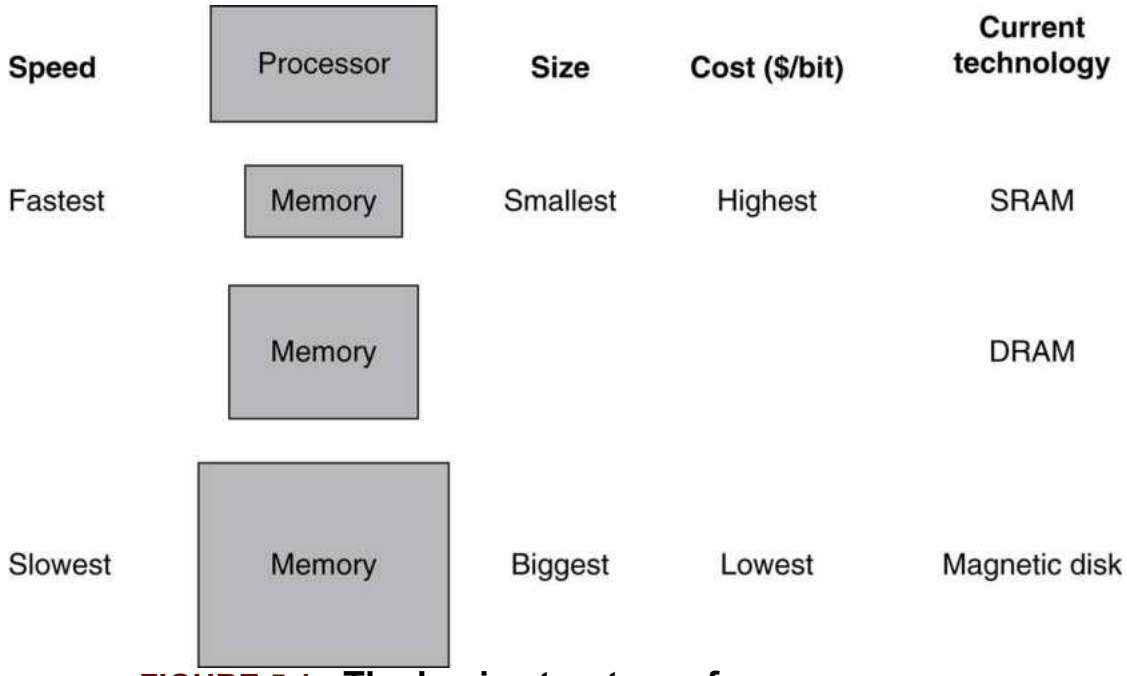


FIGURE 5.1 The basic structure of a memory hierarchy.
By implementing the memory system as a hierarchy, the user has the illusion of a memory that is as large as the largest level of the hierarchy, but can be accessed as if it were all built from the fastest memory. Flash memory has replaced disks in many personal mobile devices, and may lead to a new level in the storage hierarchy for desktop and server computers; see Section 5.2.

The data are similarly hierarchical: a level closer to the processor is generally a subset of any level further away, and all the data are stored at the lowest level. By analogy, the books on your desk form a subset of the library you are working in, which is in turn a subset of all the libraries on campus. Furthermore, as we move away from the processor, the levels take progressively longer to access, just as we might encounter in a hierarchy of campus libraries.

A memory hierarchy can consist of multiple levels, but data are copied between only two adjacent levels at a time, so we can focus our attention on just two levels. The upper level—the one closer to the processor—is smaller and faster than the lower level, since the upper level uses technology that is more expensive. Figure 5.2 shows that the minimum unit of information that can be either present or not present in the two-level hierarchy is called a **block** or a **line**; in our library analogy, a block of information is one book.

block (or line)

The minimum unit of information that can be either present or not present in a cache.

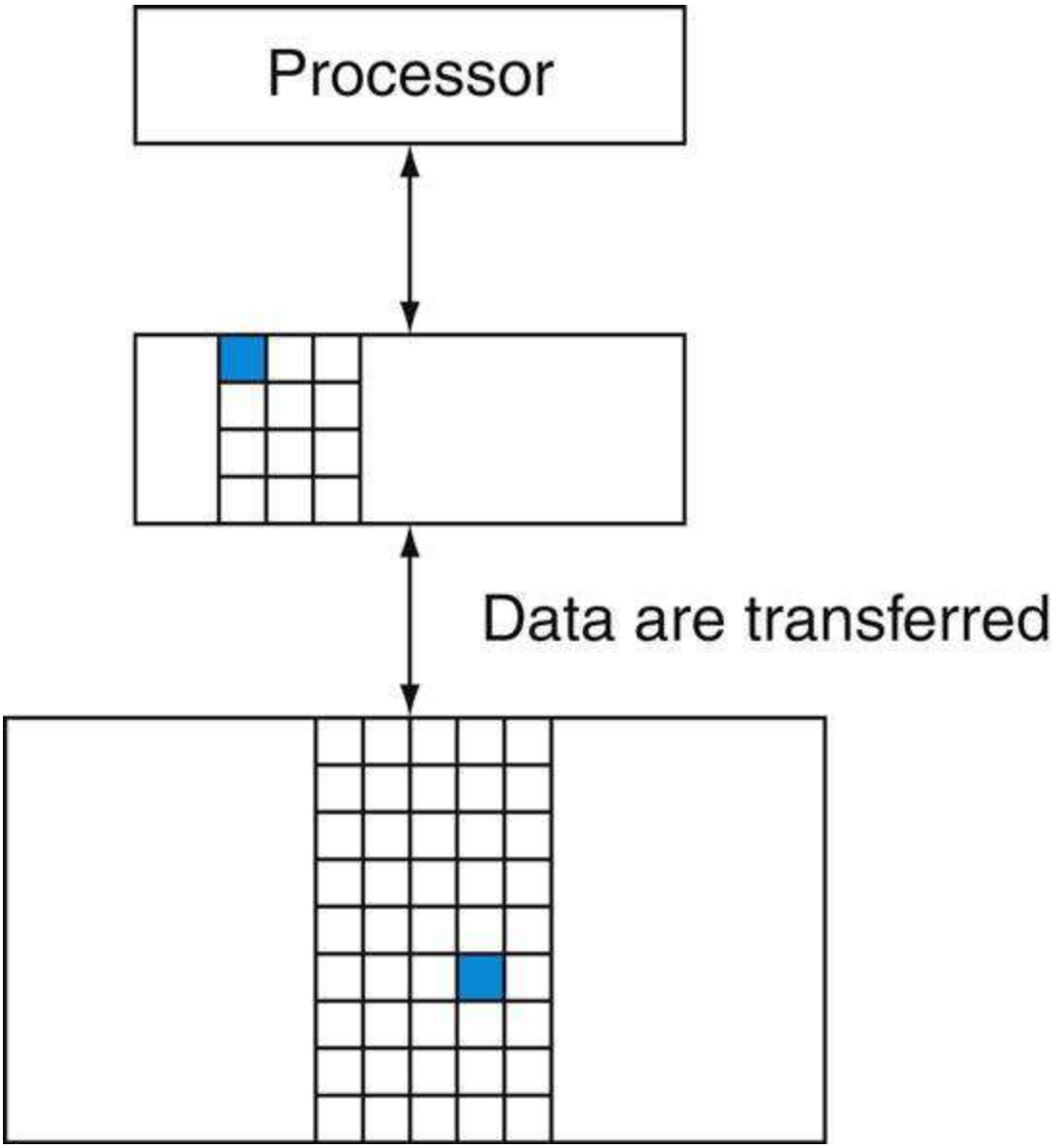


FIGURE 5.2 Every pair of levels in the memory hierarchy can be thought of as having an upper and lower level.
Within each level, the unit of information that is present or not is called a *block* or a *line*. Usually we transfer an entire block when we copy something between levels.

If the data requested by the processor appear in some block in the upper level, this is called a *hit* (analogous to your finding the

information in one of the books on your desk). If the data are not found in the upper level, the request is called a *miss*. The lower level in the hierarchy is then accessed to retrieve the block containing the requested data. (Continuing our analogy, you go from your desk to the shelves to find the desired book.) The **hit rate**, or *hit ratio*, is the fraction of memory accesses found in the upper level; it is often used as a measure of the performance of the memory hierarchy. The **miss rate** (1−hit rate) is the fraction of memory accesses not found in the upper level.

hit rate

The fraction of memory accesses found in a level of the memory hierarchy.

miss rate

The fraction of memory accesses not found in a level of the memory hierarchy.

Since performance is the major reason for having a memory hierarchy, the time to service hits and misses is important. **Hit time** is the time to access the upper level of the memory hierarchy, which includes the time needed to determine whether the access is a hit or a miss (that is, the time needed to look through the books on the desk). The **miss penalty** is the time to replace a block in the upper level with the corresponding block from the lower level, plus the time to deliver this block to the processor (or the time to get another book from the shelves and place it on the desk). Because the upper level is smaller and built using faster memory parts, the hit time will be much smaller than the time to access the next level in the hierarchy, which is the major component of the miss penalty. (The time to examine the books on the desk is much smaller than the time to get up and get a new book from the shelves.)

hit time

The time required to access a level of the memory hierarchy, including the time needed to determine whether the access is a hit or a miss.

miss penalty

The time required to fetch a block into a level of the memory hierarchy from the lower level, including the time to access the block, transmit it from one level to the other, insert it in the level that experienced the miss, and then pass the block to the requestor.

As we will see in this chapter, the concepts used to build memory systems affect many other aspects of a computer, including how the operating system manages memory and I/O, how compilers generate code, and even how applications use the computer. Of course, because all programs spend much of their time accessing memory, the memory system is necessarily a major factor in determining performance. The reliance on memory hierarchies to achieve performance has meant that programmers, who used to be able to think of memory as a flat, random access storage device, now need to understand that memory is a hierarchy to get good performance. We show how important this understanding is in later examples, such as Figure 5.18 on page 400, and Section 5.14, which shows how to double matrix multiply performance.

Since memory systems are critical to performance, computer designers devote a great deal of attention to these systems and develop sophisticated mechanisms for improving the performance of the memory system. In this chapter, we discuss the major conceptual ideas, although we use many simplifications and abstractions to keep the material manageable in length and complexity.

The BIG Picture

Programs exhibit both temporal locality, the tendency to reuse recently accessed data items, and spatial locality, the tendency to reference data items that are close to other recently accessed items. Memory hierarchies take advantage of temporal locality by keeping more recently accessed data items closer to the processor. Memory hierarchies take advantage of spatial locality by moving blocks consisting of multiple contiguous words in memory to upper levels of the hierarchy.

Figure 5.3 shows that a memory hierarchy uses smaller and faster memory technologies close to the processor. Thus, accesses

that hit in the highest level of the hierarchy can be processed quickly. Accesses that miss go to lower levels of the hierarchy, which are larger but slower. If the hit rate is high enough, the memory hierarchy has an effective access time close to that of the highest (and fastest) level and a size equal to that of the lowest (and largest) level.

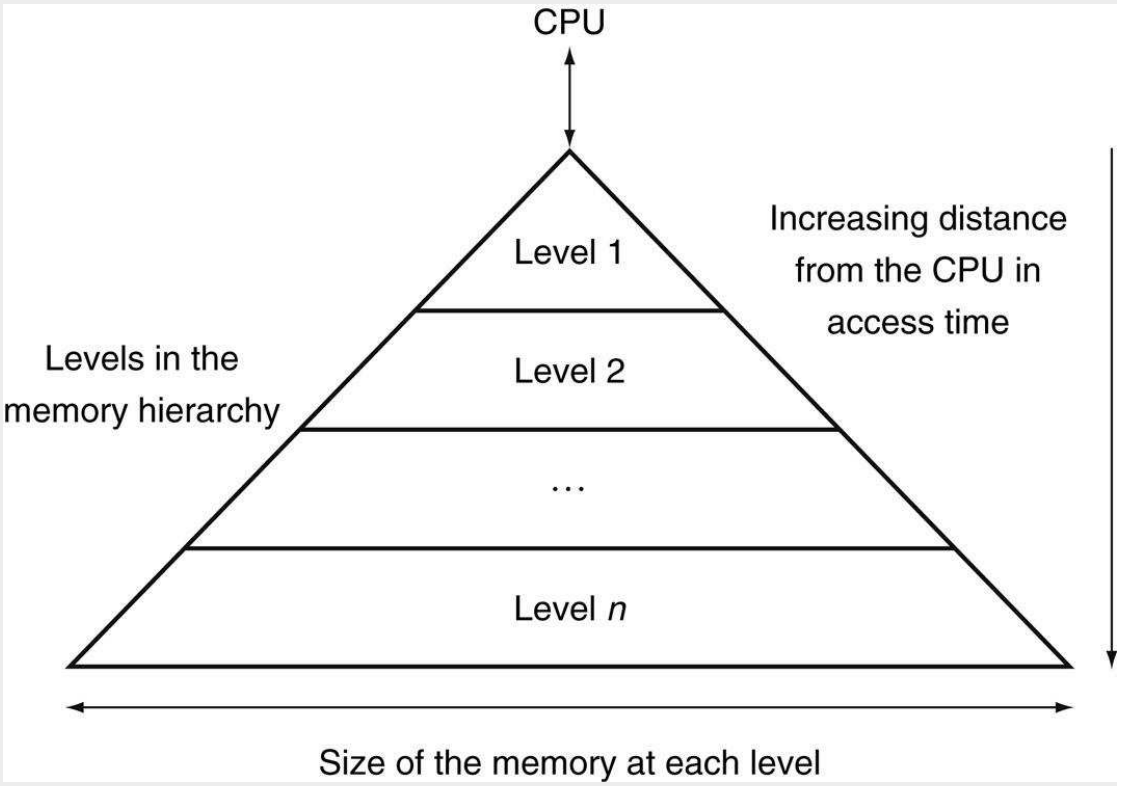


FIGURE 5.3 This diagram shows the structure of a memory hierarchy: as the distance from the processor increases, so does the size.
This structure, with the appropriate operating mechanisms, allows the processor to have an access time that is determined primarily by level 1 of the hierarchy and yet have a memory as large as level *n*. Maintaining this illusion is the subject of this chapter. Although the local disk is normally the bottom of the hierarchy, some systems use tape or a file server over a local area network as the next levels of the hierarchy.

In most systems, the memory is a true hierarchy, meaning that data cannot be present in level i unless they are also present in level $i + 1$.

Check Yourself

Which of the following statements are generally true?

1. Memory hierarchies take advantage of temporal locality.
2. On a read, the value returned depends on which blocks are in the cache.
3. Most of the cost of the memory hierarchy is at the highest level.
4. Most of the capacity of the memory hierarchy is at the lowest level.

5.2 Memory Technologies

There are four primary technologies used today in memory hierarchies. Main memory is implemented from DRAM (*dynamic random access memory*), while levels closer to the processor (caches) use SRAM (*static random access memory*). DRAM is less costly per bit than SRAM, although it is substantially slower. The price difference arises because DRAM uses significantly less area per bit of memory, and DRAMs thus have larger capacity for the same amount of silicon; the speed difference arises from several factors described in Section A.9 of Appendix A. The third technology is flash memory. This nonvolatile memory is the secondary memory in Personal Mobile Devices. The fourth technology, used to implement the largest and slowest level in the hierarchy in servers, is magnetic disk. The access time and price per bit vary widely among these technologies, as the table below shows, using typical values for 2012.

Memory technology	Typical access time	$ per GiB in 2012
SRAM semiconductor memory	0.5–2.5 ns	$500–$1000
DRAM semiconductor memory	50–70 ns	$10–$20
Flash semiconductor memory	5,000–50,000 ns	$0.75–$1.00
Magnetic disk	5,000,000–20,000,000 ns	$0.05–$0.10

We describe each memory technology in the remainder of this section.

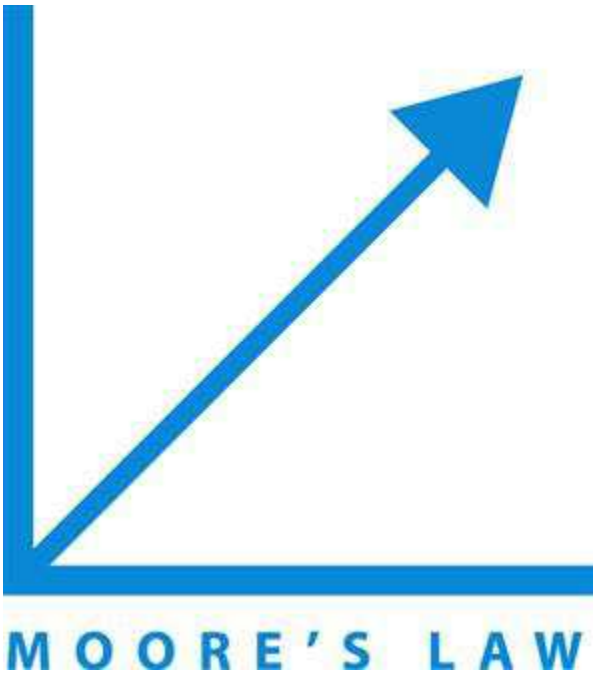
SRAM Technology

SRAMs are simply integrated circuits that are memory arrays with (usually) a single access port that can provide either a read or a write. SRAMs have a fixed access time to any datum, though the

read and write access times may differ.

SRAMs don't need to refresh and so the access time is very close to the cycle time. SRAMs typically use six to eight transistors per bit to prevent the information from being disturbed when read. SRAM needs only minimal power to retain the charge in standby mode.

In the past, most PCs and server systems used separate SRAM chips for either their primary, secondary, or even tertiary caches. Today, thanks to **Moore's Law**, all levels of caches are integrated onto the processor chip, so the market for independent SRAM chips has nearly evaporated.



DRAM Technology

In a SRAM, as long as power is applied, the value can be kept indefinitely. In a *dynamic RAM* (DRAM), the value kept in a cell is stored as a charge in a capacitor. A single transistor is then used to access this stored charge, either to read the value or to overwrite the charge stored there. Because DRAMs use only one transistor per bit of storage, they are much denser and cheaper per bit than SRAM. As DRAMs store the charge on a capacitor, it cannot be kept indefinitely and must periodically be refreshed. That is why this memory structure is called dynamic, in contrast to the static storage

in an SRAM cell.

To refresh the cell, we merely read its contents and write it back. The charge can be kept for several milliseconds. If every bit had to be read out of the DRAM and then written back individually, we would constantly be refreshing the DRAM, leaving no time for accessing it. Fortunately, DRAMs use a two-level decoding structure, and this allows us to refresh an entire *row* (which shares a word line) with a read cycle followed immediately by a write cycle.

Figure 5.4 shows the internal organization of a DRAM, and Figure 5.5 shows how the density, cost, and access time of DRAMs have changed over the years.

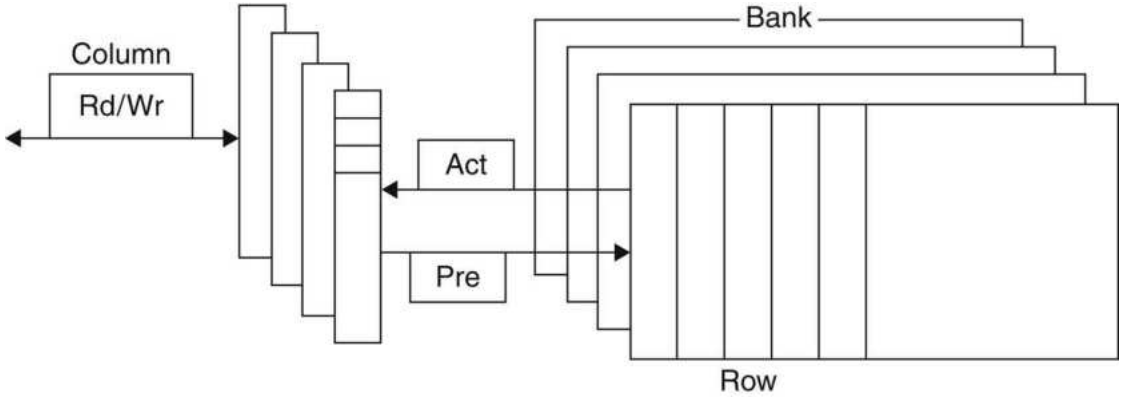


FIGURE 5.4 Internal organization of a DRAM. Modern DRAMs are organized in banks, typically four for DDR3. Each bank consists of a series of rows. Sending a PRE (precharge) command opens or closes a bank. A row address is sent with an Act (activate), which causes the row to transfer to a buffer. When the row is in the buffer, it can be transferred by successive column addresses at whatever the width of the DRAM is (typically 4, 8, or 16 bits in DDR3) or by specifying a block transfer and the starting address. Each command, as well as block transfers, is synchronized with a clock.

Year introduced	Chip size	$ per GiB	Total access time to a new row/column	Average column access time to existing row
1980	64 Kibibit	$1,500,000	250 ns	150 ns
1983	256 Kibibit	$500,000	185 ns	100 ns
1985	1 Mebibit	$200,000	135 ns	40 ns
1989	4 Mebibit	$50,000	110 ns	40 ns
1992	16 Mebibit	$15,000	90 ns	30 ns
1996	64 Mebibit	$10,000	60 ns	12 ns
1998	128 Mebibit	$4,000	60 ns	10 ns
2000	256 Mebibit	$1,000	55 ns	7 ns
2004	512 Mebibit	$250	50 ns	5 ns
2007	1 Gibibit	$50	45 ns	1.25 ns
2010	2 Gibibit	$30	40 ns	1 ns
2012	4 Gibibit	$1	35 ns	0.8 ns

FIGURE 5.5 DRAM size increased by multiples of four approximately once every 3 years until 1996, and thereafter considerably slower.
The improvements in access time have been slower but continuous, and cost roughly tracks density improvements, although cost is often affected by other issues, such as availability and demand. The cost per gibibyte is not adjusted for inflation.

The row organization that helps with refresh also helps with performance. To improve performance, DRAMs buffer rows for repeated access. The buffer acts like an SRAM; by changing the address, random bits can be accessed in the buffer until the next row access. This capability improves the access time significantly, since the access time to bits in the row is much lower. Making the chip wider also improves the memory bandwidth of the chip. When the row is in the buffer, it can be transferred by successive addresses at whatever the width of the DRAM is (typically 4, 8, or 16 bits), or by specifying a block transfer and the starting address within the buffer.

To improve the interface to processors further, DRAMs added clocks and are properly called synchronous DRAMs or SDRAMs. The advantage of SDRAMs is that the use of a clock eliminates the time for the memory and processor to synchronize. The speed advantage of synchronous DRAMs comes from the ability to transfer the bits in the burst without having to specify additional address bits. Instead, the clock transfers the successive bits in a burst. The fastest version is called *Double Data Rate (DDR)* SDRAM.

The name means data transfers on both the rising *and* falling edge of the clock, thereby getting twice as much bandwidth as you might expect based on the clock rate and the data width. The latest version of this technology is called DDR4. A DDR4-3200 DRAM can do 3200 million transfers per second, which means it has a 1600- MHz clock.

Sustaining that much bandwidth requires clever organization *inside* the DRAM. Instead of just a faster row buffer, the DRAM can be internally organized to read or write from multiple *banks*, with each having its own row buffer. Sending an address to several banks permits them all to read or write simultaneously. For example, with four banks, there is just one access time and then accesses rotate between the four banks to supply four times the bandwidth. This rotating access scheme is called *address interleaving*.

Although personal mobile devices like the iPad (see Chapter 1) use individual DRAMs, memory for servers is commonly sold on small boards called *dual inline memory modules* (DIMMs). DIMMs typically contain 4–16 DRAMs, and they are normally organized to be 8 bytes wide for server systems. A DIMM using DDR4-3200 SDRAMs could transfer at 8 ×3200 =25,600 megabytes per second. Such DIMMs are named after their bandwidth: PC25600. Since a DIMM can have so many DRAM chips that only a portion of them are used for a particular transfer, we need a term to refer to the subset of chips in a DIMM that share common address lines. To avoid confusion with the internal DRAM names of row and banks, we use the term *memory rank* for such a subset of chips in a DIMM.

Elaboration

One way to measure the performance of the memory system behind the caches is the Stream benchmark [McCalpin, 1995]. It measures the performance of long vector operations. They have no temporal locality and they access arrays that are larger than the cache of the computer being tested.

Flash Memory

Flash memory is a type of *electrically erasable programmable read-only memory* (EEPROM).

Unlike disks and DRAM, but like other EEPROM technologies,

writes can wear out flash memory bits. To cope with such limits, most flash products include a controller to spread the writes by remapping blocks that have been written many times to less trodden blocks. This technique is called *wear leveling*. With wear leveling, personal mobile devices are very unlikely to exceed the write limits in the flash. Such wear leveling lowers the potential performance of flash, but it is needed unless higher-level software monitors block wear. Flash controllers that perform wear leveling can also improve yield by mapping out memory cells that were manufactured incorrectly.

Disk Memory

As Figure 5.6 shows, a magnetic hard disk consists of a collection of platters, which rotate on a spindle at 5400 to 15,000 revolutions per minute. The metal platters are covered with magnetic recording material on both sides, similar to the material found on a cassette or videotape. To read and write information on a hard disk, a movable *arm* containing a small electromagnetic coil called a *read-write head* is located just above each surface. The entire drive is permanently sealed to control the environment inside the drive, which, in turn, allows the disk heads to be much closer to the drive surface.

FIGURE 5.6 A disk showing 10 disk platters and the read/write heads.
The diameter of today's disks is 2.5 or 3.5 inches, and there are typically one or two platters per drive today.

Each disk surface is divided into concentric circles, called **tracks**. There are typically tens of thousands of tracks per surface. Each track is in turn divided into **sectors** that contain the information; each track may have thousands of sectors. Sectors are typically 512

to 4096 bytes in size. The sequence recorded on the magnetic media is a sector number, a gap, the information for that sector including error correction code (see Section 5.5), a gap, the sector number of the next sector, and so on.

track

One of thousands of concentric circles that make up the surface of a magnetic disk.

sector

One of the segments that make up a track on a magnetic disk; a sector is the smallest amount of information that is read or written on a disk.

The disk heads for each surface are connected together and move in conjunction, so that every head is over the same track of every surface. The term *cylinder* is used to refer to all the tracks under the heads at a given point on all surfaces.

To access data, the operating system must direct the disk through a three-stage process. The first step is to position the head over the proper track. This operation is called a **seek**, and the time to move the head to the desired track is called the *seek time*.

seek

The process of positioning a read/write head over the proper track on a disk.

Disk manufacturers report minimum seek time, maximum seek time, and average seek time in their manuals. The first two are easy to measure, but the average is open to wide interpretation because it depends on the seek distance. The industry calculates average seek time as the sum of the time for all possible seeks divided by the number of possible seeks. Average seek times are usually advertised as 3 ms to 13 ms, but, depending on the application and scheduling of disk requests, the actual average seek time may be only 25% to 33% of the advertised number because of the locality of disk references. This locality arises both because of successive

accesses to the same file and because the operating system tries to schedule such accesses together.

Once the head has reached the correct track, we must wait for the desired sector to rotate under the read/write head. This time is called the **rotational latency** or **rotational delay**. The average latency to the desired information is halfway around the disk. Disks rotate at 5400 RPM to 15,000 RPM. The average rotational latency at 5400 RPM is

rotational latency

Also called **rotational delay**. The time required for the desired sector of a disk to rotate under the read/write head; usually assumed to be half the rotation time.

$$\text{Average rotational latency} = \frac{0.5 \text{ rotation}}{5400 \text{ RPM}} = \frac{0.5 \text{ rotation}}{5400 \text{ RPM} / \left(60 \frac{\text{seconds}}{\text{minute}}\right)}$$
$$= 0.0056 \text{ seconds} = 5.6 \text{ ms}$$

The last component of a disk access, *transfer time*, is the time to transfer a block of bits. The transfer time is a function of the sector size, the rotation speed, and the recording density of a track. Transfer rates in 2012 were between 100 and 200 MB/sec.

One complication is that most disk controllers have a built-in cache that stores sectors as they are passed over; transfer rates from the cache are typically higher, and were up to 750 MB/sec (6 Gbit/sec) in 2012.

Alas, where block numbers are located is no longer intuitive. The assumptions of the sector-track-cylinder model above are that nearby blocks are on the same track, blocks in the same cylinder take less time to access since there is no seek time, and some tracks are closer than others. The reason for the change was the raising of the level of the disk interfaces. To speed-up sequential transfers, these higher-level interfaces organize disks more like tapes than like random access devices. The logical blocks are ordered in serpentine fashion across a single surface, trying to capture all the sectors that are recorded at the same bit density to try to get best performance. Hence, sequential blocks may be on different tracks.

In summary, the two primary differences between magnetic disks and semiconductor memory technologies are that disks have a slower access time because they are mechanical devices—flash is 1000 times as fast and DRAM is 100,000 times as fast—yet they are cheaper per bit because they have very high storage capacity at a modest cost—disks are 10 to 100 times cheaper. Magnetic disks are nonvolatile like flash, but unlike flash there is no write wear-out problem. However, flash is much more rugged and hence a better match to the jostling inherent in personal mobile devices.

5.3 The Basics of Caches

Cache: a safe place for hiding or storing things.

Webster's New World Dictionary of the American Language, Third College Edition, 1988

In our library example, the desk acted as a cache—a safe place to store things (books) that we needed to examine. *Cache* was the name chosen to represent the level of the memory hierarchy between the processor and main memory in the first commercial computer to have this extra level. The memories in the datapath in Chapter 4 are simply replaced by caches. Today, although this remains the dominant use of the word *cache*, the term is also used to refer to any storage managed to take advantage of locality of access. Caches first appeared in research computers in the early 1960s and in production computers later in that same decade; every general-purpose computer built now from servers to low-power embedded processors, includes caches.

In this section, we begin by looking at a very simple cache in which the processor requests are each one word, and the blocks also consist of a single word. (Readers already familiar with cache basics may want to skip to Section 5.4.) Figure 5.7 shows such a simple cache, before and after requesting a data item that is not initially in the cache. Before the request, the cache contains a collection of recent references X_1, X_2, …, X_{n-1}, and the processor requests a word X_n that is not in the cache. This request results in a miss, and the word X_n is brought from memory into the cache.

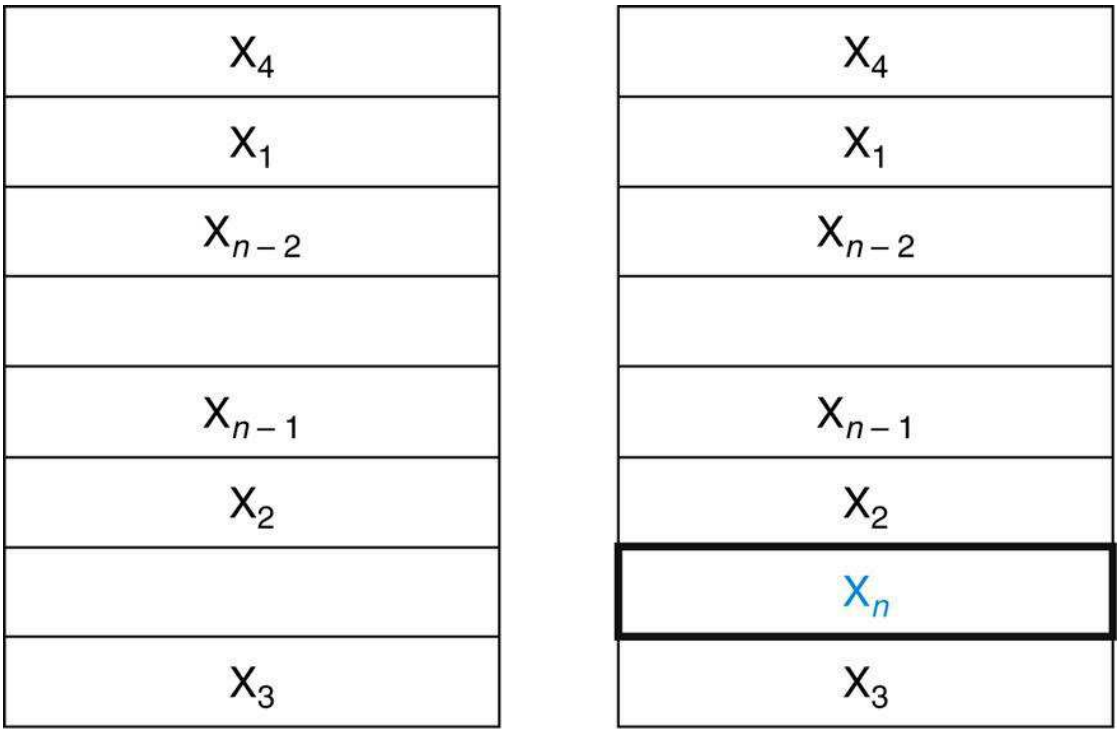


FIGURE 5.7 The cache just before and just after a reference to a word X_n that is not initially in the cache.
This reference causes a miss that forces the cache to fetch X_n from memory and insert it into the cache.

In looking at the scenario in Figure 5.7, there are two questions to answer: How do we know if a data item is in the cache? Moreover, if it is, how do we find it? The answers are related. If each word can go in exactly one place in the cache, then it is straightforward to find the word if it is in the cache. The simplest way to assign a location in the cache for each word in memory is to assign the cache location based on the *address* of the word in memory. This cache structure is called **direct mapped**, since each memory location is mapped directly to exactly one location in the cache. The typical mapping between addresses and cache locations for a direct-mapped cache is usually simple. For example, almost all direct-mapped caches use this mapping to find a block:

direct-mapped cache
A cache structure in which each memory location is mapped to

exactly one location in the cache.

(Block address) modulo (Number of blocks in the cache)

If the number of entries in the cache is a power of 2, then modulo can be computed simply by using the low-order $\log_2$ (cache size in blocks) bits of the address. Thus, an 8-block cache uses the three lowest bits ($8=2^3$) of the block address. For example, Figure 5.8 shows how the memory addresses between 1_{ten} (00001_{two}) and 29_{ten} (11101_{two}) map to locations 1_{ten} (001_{two}) and 5_{ten} (101_{two}) in a direct-mapped cache of eight words.

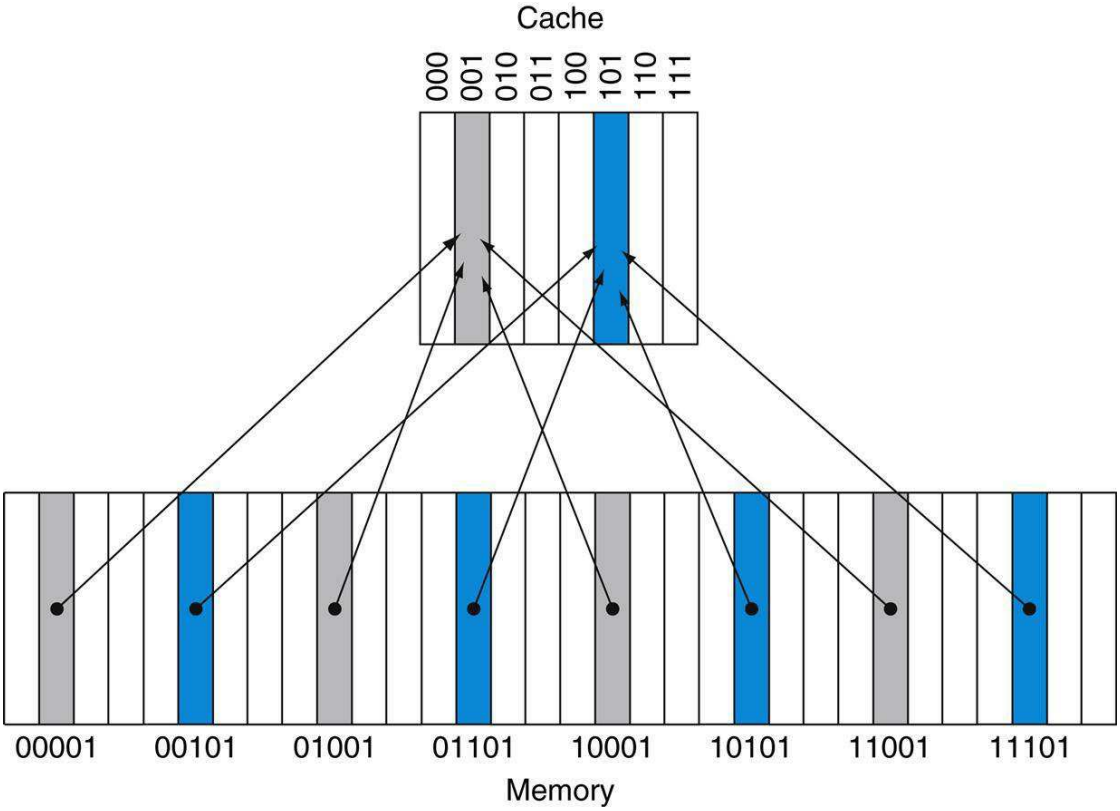


FIGURE 5.8 A direct-mapped cache with eight entries showing the addresses of memory words between 0 and 31 that map to the same cache locations.

Because there are eight words in the cache, an address X maps to the direct-mapped cache word X modulo 8. That is, the low-order $\log_2(8)=3$ bits are used as the cache index. Thus, addresses 00001_{two}, 01001_{two}, 10001_{two}, and 11001_{two} all map to entry 001_{two}

of the cache, while addresses 00101_{two}, 01101_{two}, 10101_{two}, and 11101_{two} all map to entry 101_{two} of the cache.

Because each cache location can contain the contents of a number of different memory locations, how do we know whether the data in the cache corresponds to a requested word? That is, how do we know whether a requested word is in the cache or not? We answer this question by adding a set of **tags** to the cache. The tags contain the address information required to identify whether a word in the cache corresponds to the requested word. The tag needs just to contain the upper portion of the address, corresponding to the bits that are not used as an index into the cache. For example, in Figure 5.8 we need only have the upper two of the five address bits in the tag, since the lower 3-bit index field of the address selects the block. Architects omit the index bits because they are redundant, since by definition, the index field of any address of a cache block must be that block number.

tag

A field in a table used for a memory hierarchy that contains the address information required to identify whether the associated block in the hierarchy corresponds to a requested word.

We also need a way to recognize that a cache block does not have valid information. For instance, when a processor starts up, the cache does not have good data, and the tag fields will be meaningless. Even after executing many instructions, some of the cache entries may still be empty, as in Figure 5.7. Thus, we need to know that the tag should be ignored for such entries. The most common method is to add a **valid bit** to indicate whether an entry contains a valid address. If the bit is not set, there cannot be a match for this block.

valid bit

A field in the tables of a memory hierarchy that indicates that the associated block in the hierarchy contains valid data.

For the rest of this section, we will focus on explaining how a cache deals with reads. In general, handling reads is a little simpler than handling writes, since reads do not have to change the contents of the cache. After seeing the basics of how reads work and how cache misses can be handled, we'll examine the cache designs for real computers and detail how these caches handle writes.

The BIG Picture

Caching is perhaps the most important example of the big idea of **prediction**. It relies on the principle of locality to try to find the desired data in the higher levels of the memory hierarchy, and provides mechanisms to ensure that when the prediction is wrong it finds and uses the proper data from the lower levels of the memory hierarchy. The hit rates of the cache prediction on modern computers are often above 95% (see Figure 5.46).

Accessing a Cache

Below is a sequence of nine memory references to an empty eight-block cache, including the action for each reference. Figure 5.9 shows how the contents of the cache change on each miss. Since there are eight blocks in the cache, the low-order 3 bits of an address give the block number:

Decimal address of reference	Binary address of reference	Hit or miss in cache	Assigned cache block (where found or placed)
22	10110_{two}	miss (5.9b)	$(10110_{two} \bmod 8) = 110_{two}$
26	11010_{two}	miss (5.9c)	$(11010_{two} \bmod 8) = 010_{two}$
22	10110_{two}	hit	$(10110_{two} \bmod 8) = 110_{two}$
26	11010_{two}	hit	$(11010_{two} \bmod 8) = 010_{two}$
16	10000_{two}	miss (5.9d)	$(10000_{two} \bmod 8) = 000_{two}$
3	00011_{two}	miss (5.9e)	$(00011_{two} \bmod 8) = 011_{two}$
16	10000_{two}	hit	$(10000_{two} \bmod 8) = 000_{two}$
18	10010_{two}	miss (5.9f)	$(10010_{two} \bmod 8) = 010_{two}$
16	10000_{two}	hit	$(10000_{two} \bmod 8) = 000_{two}$

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

a. The initial state of the cache after power-on

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

b. After handling a miss of address (10110_{two})

Index	V	Tag	Data
000	N		
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

c. After handling a miss of address (11010_{two})

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

d. After handling a miss of address (10000_{two})

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	Y	00_{two}	Memory (00011_{two})
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

e. After handling a miss of address (00011_{two})

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	10_{two}	Memory (10010_{two})
011	Y	00_{two}	Memory (00011_{two})
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

f. After handling a miss of address (10010_{two})

FIGURE 5.9 The cache contents are shown after each reference request that misses, with the index and tag fields shown in binary for the sequence of addresses on page 379.

The cache is initially empty, with all valid bits (V entry in cache) turned off (N). The processor requests the following addresses: 10110_{two} (miss), 11010_{two} (miss), 10110_{two} (hit), 11010_{two} (hit), 10000_{two} (miss), 00011_{two} (miss), 10000_{two} (hit), 10010_{two} (miss), and 10000_{two} (hit). The figures show the cache contents after each miss in the sequence has been handled. When address 10010_{two} (18) is referenced, the entry for address 11010_{two} (26) must be replaced, and a reference to 11010_{two} will cause a subsequent miss. The tag field will contain only the upper portion of the address. The full address of a word contained in cache block *i* with tag field *j* for this cache is *j*×8+*i*, or equivalently the concatenation of the tag field *j* and the index *i*. For example, in cache *f* above, index 010_{two}

has tag 10_{two} and corresponds to address 10010_{two}.

Since the cache is empty, several of the first references are misses; the caption of Figure 5.9 describes the actions for each memory reference. On the eighth reference we have conflicting demands for a block. The word at address 18 (10010_{two}) should be brought into cache block 2 (010_{two}). Hence, it must replace the word at address 26 (11010_{two}), which is already in cache block 2 (010_{two}). This behavior allows a cache to take advantage of temporal locality: recently referenced words replace less recently referenced words.

This situation is directly analogous to needing a book from the shelves and having no more space on your desk—some book already on your desk must be returned to the shelves. In a direct-mapped cache, there is only one place to put the newly requested item and hence just one choice of what to replace.

We know where to look in the cache for each possible address: the low-order bits of an address can be used to find the unique cache entry to which the address could map. Figure 5.10 shows how a referenced address is divided into

- A *tag field*, which is used to compare with the value of the tag field of the cache
- A *cache index*, which is used to select the block

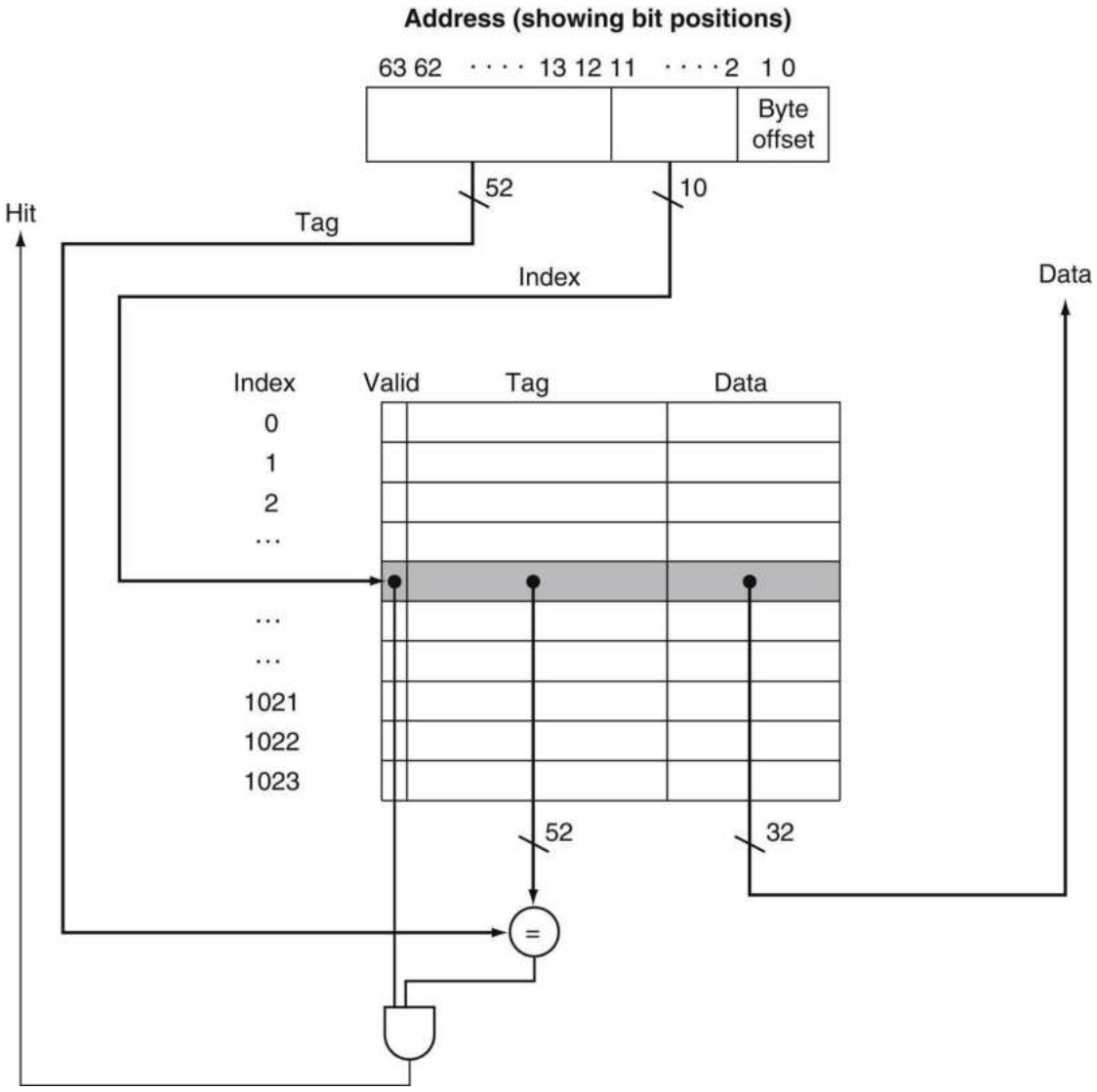


FIGURE 5.10 For this cache, the lower portion of the address is used to select a cache entry consisting of a data word and a tag.
This cache holds 1024 words or 4 KiB. Unless noted otherwise, we assume 64-bit addresses in this chapter. The tag from the cache is compared against the upper portion of the address to determine whether the entry in the cache corresponds to the requested address. Because the cache has 2^{10} (or 1024) words and a block size of one word, 10 bits are used to index the cache, leaving 64 −10 −2 =52 bits to be compared against the tag. If the tag and upper 52 bits of the address are equal and the valid bit is on, then the request hits in the cache, and the word is supplied to the processor. Otherwise, a miss occurs.

The index of a cache block, together with the tag contents of that

block, uniquely specifies the memory address of the word contained in the cache block. Because the index field is used as an address to reference the cache, and because an n-bit field has 2^n values, the total number of entries in a direct-mapped cache must be a power of 2. Since words are aligned to multiples of four bytes, the least significant two bits of every address specify a byte within a word. Hence, if the words are aligned in memory, the least significant two bits can be ignored when selecting a word in the block. For this chapter, we'll assume that data are aligned in memory, and discuss how to handle unaligned cache accesses in an Elaboration.

The total number of bits needed for a cache is a function of the cache size and the address size, because the cache includes both the storage for the data and the tags. The size of the block above was one word (4 bytes), but normally it is several. For the following situation:

- 64-bit addresses
- A direct-mapped cache
- The cache size is 2^n blocks, so n bits are used for the index
- The block size is 2^m words (2^{m+2} bytes), so m bits are used for the word within the block, and two bits are used for the byte part of the address

The size of the tag field is

$$64 - (n + m + 2).$$

The total number of bits in a direct-mapped cache is

$$2^n \times (\text{block size} + \text{tag size} + \text{valid field size}).$$

Since the block size is 2^m words (2^{m+5} bits), and we need 1 bit for the valid field, the number of bits in such a cache is

$$2^n \times (2^m \times 32 + (64 - n - m - 2) + 1) = 2^n \times (2^m \times 32 + 63 - n - m).$$

Although this is the actual size in bits, the naming convention is to exclude the size of the tag and valid field and to count only the

size of the data. Thus, the cache in Figure 5.10 is called a 4 KiB cache.

Bits in a Cache

Example

How many total bits are required for a direct-mapped cache with 16 KiB of data and four-word blocks, assuming a 64-bit address?

Answer

We know that 16 KiB is 4096 (2^{12}) words. With a block size of four words (2^2), there are 1024 (2^{10}) blocks. Each block has 4 ×32 or 128 bits of data plus a tag, which is 64 −10 −2 −2 bits, plus a valid bit. Thus, the complete cache size is

$$2^{10} \times (4 \times 32 + (64 - 10 - 2 - 2) + 1) = 2^{10} \times 179 = 179 \text{ Kibibits}$$

or 22.4 KiB for a 16 KiB cache. For this cache, the total number of bits in the cache is about 1.4 times as many as needed just for the storage of the data.

Mapping an Address to a Multiword Cache Block

Example

Consider a cache with 64 blocks and a block size of 16 bytes. To what block number does byte address 1200 map?

Answer

We saw the formula on page 376. The block is given by

$$\text{(Block address) modulo (Number of blocks in the cache)}$$

where the address of the block is

$$\frac{\text{Byte address}}{\text{Bytes per block}}$$

Notice that this block address is the block containing all addresses between

$$\left\lfloor \frac{\text{Byte address}}{\text{Bytes per block}} \right\rfloor \times \text{Bytes per block}$$

and

$$\left\lfloor \frac{\text{Byte address}}{\text{Bytes per block}} \right\rfloor \times \text{Bytes per block} + (\text{Bytes per block} - 1)$$

Thus, with 16 bytes per block, byte address 1200 is block address

$$\left\lfloor \frac{1200}{16} \right\rfloor = 75$$

which maps to cache block number (75 modulo 64)=11. In fact, this block maps all addresses between 1200 and 1215.

Larger blocks exploit spatial locality to lower miss rates. As Figure 5.11 shows, increasing the block size usually decreases the miss rate. The miss rate may go up eventually if the block size becomes a significant fraction of the cache size, because the number of blocks that can be held in the cache will become small, and there will be a great deal of competition for those blocks. As a result, a block will be bumped out of the cache before many of its words are accessed. Stated alternatively, spatial locality among the words in a block decreases with a very large block; consequently, the benefits to the miss rate become smaller.

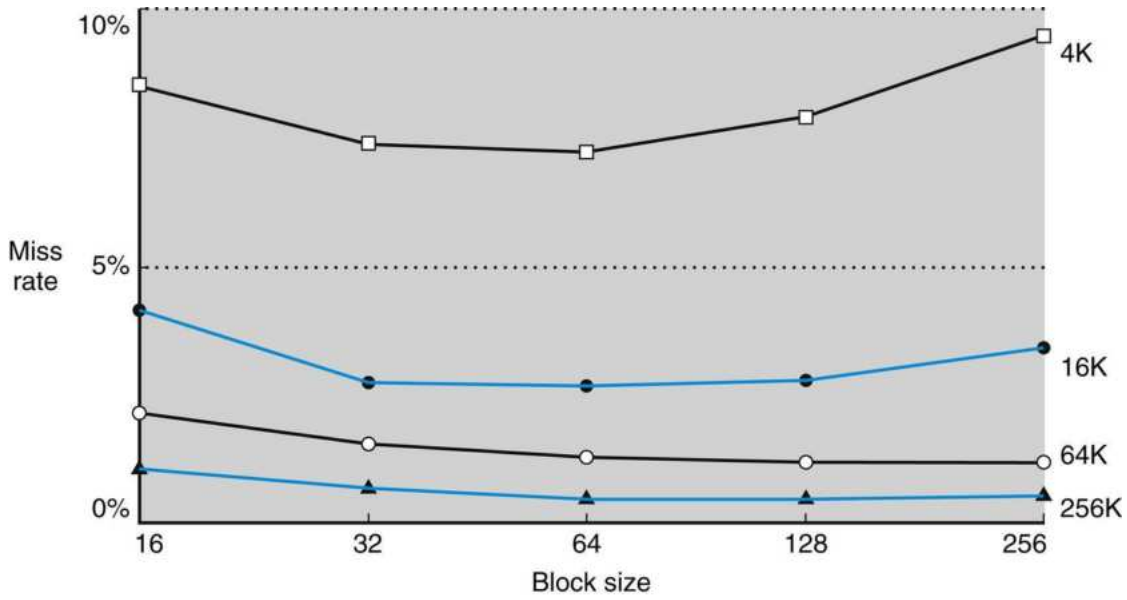


FIGURE 5.11 Miss rate versus block size. Note that the miss rate actually goes up if the block size is too large relative to the cache size. Each line represents a cache of different size. (This figure is independent of associativity, discussed soon.) Unfortunately, SPEC CPU2000 traces would take too long if block size were included, so these data are based on SPEC92.

A more serious problem associated with just increasing the block size is that the cost of a miss rises. The miss penalty is determined by the time required to fetch the block from the next lower level of the hierarchy and load it into the cache. The time to fetch the block has two parts: the latency to the first word and the transfer time for the rest of the block. Clearly, unless we change the memory system, the transfer time—and hence the miss penalty—will likely increase as the block size expands. Furthermore, the improvement in the miss rate starts to decrease as the blocks become larger. The result is that the increase in the miss penalty overwhelms the decrease in the miss rate for blocks that are too large, and cache performance thus decreases. Of course, if we design the memory to transfer larger blocks more efficiently, we can increase the block size and obtain further improvements in cache performance. We discuss this topic in the next section.

Elaboration

Although it is hard to do anything about the longer latency

component of the miss penalty for large blocks, we may be able to hide some of the transfer time so that the miss penalty is effectively smaller. The easiest method for doing this, called *early restart*, is simply to resume execution as soon as the requested word of the block is returned, rather than wait for the entire block. Many processors use this technique for instruction access, where it works best. Instruction accesses are largely sequential, so if the memory system can deliver a word every clock cycle, the processor may be able to restart operation when the requested word is returned, with the memory system delivering new instruction words just in time. This technique is usually less effective for data caches because it is likely that the words will be requested from the block in a less predictable way, and the probability that the processor will need another word from a different cache block before the transfer completes is high. If the processor cannot access the data cache because a transfer is ongoing, then it must stall.

An even more sophisticated scheme is to organize the memory so that the requested word is transferred from the memory to the cache first. The remainder of the block is then transferred, starting with the address after the requested word and wrapping around to the beginning of the block. This technique, called *requested word first* or *critical word first*, can be slightly faster than early restart, but it is limited by the same properties that restrain early restart.

Handling Cache Misses

Before we look at the cache of a real system, let's see how the control unit deals with **cache misses**. (We describe a cache controller in detail in Section 5.9.) The control unit must detect a miss and process the miss by fetching the requested data from memory (or, as we shall see, a lower-level cache). If the cache reports a hit, the computer continues using the data as if nothing happened.

cache miss

A request for data from the cache that cannot be filled because the data are not present in the cache.

Modifying the control of a processor to handle a hit is trivial;

misses, however, require some extra work. The cache miss handling is done in collaboration with the processor control unit and with a separate controller that initiates the memory access and refills the cache. The processing of a cache miss creates a pipeline stall (Chapter 4) in contrast to an exception or interrupt, which would require saving the state of all registers. For a cache miss, we can stall the entire processor, essentially freezing the contents of the temporary and programmer-visible registers, while we wait for memory. More sophisticated out-of-order processors can allow execution of instructions while waiting for a cache miss, but we'll assume in-order processors that stall on cache misses in this section.

Let's look a little more closely at how instruction misses are handled; the same approach can be easily extended to handle data misses. If an instruction access results in a miss, then the content of the Instruction register is invalid. To get the proper instruction into the cache, we must be able to tell the lower level in the memory hierarchy to perform a read. Since the program counter is incremented in the first clock cycle of execution, the address of the instruction that generates an instruction cache miss is equal to the value of the program counter minus 4. Once we have the address, we need to instruct the main memory to perform a read. We wait for the memory to respond (since the access will take multiple clock cycles), and then write the words containing the desired instruction into the cache.

We can now define the steps to be taken on an instruction cache miss:

1. Send the original PC value to the memory.
2. Instruct main memory to perform a read and wait for the memory to complete its access.
3. Write the cache entry, putting the data from memory in the data portion of the entry, writing the upper bits of the address (from the ALU) into the tag field, and turning the valid bit on.
4. Restart the instruction execution at the first step, which will refetch the instruction, this time finding it in the cache.

The control of the cache on a data access is essentially identical: on a miss, we simply stall the processor until the memory responds with the data.

Handling Writes

Writes work somewhat differently. Suppose on a store instruction, we wrote the data into only the data cache (without changing main memory); then, after the write into the cache, memory would have a different value from that in the cache. In such a case, the cache and memory are said to be *inconsistent*. The simplest way to keep the main memory and the cache consistent is always to write the data into both the memory and the cache. This scheme is called **write-through**.

write-through

A scheme in which writes always update both the cache and the next lower level of the memory hierarchy, ensuring that data are always consistent between the two.

The other key aspect of writes is what occurs on a write miss. We first fetch the words of the block from memory. After the block is fetched and placed into the cache, we can overwrite the word that caused the miss into the cache block. We also write the word to main memory using the full address.

Although this design handles writes very simply, it would not provide good performance. With a write-through scheme, every write causes the data to be written to main memory. These writes will take a long time, likely at least 100 processor clock cycles, and could slow down the processor considerably. For example, suppose 10% of the instructions are stores. If the CPI without cache misses was 1.0, spending 100 extra cycles on every write would lead to a CPI of 1.0 +100 ×10%=11, reducing performance by more than a factor of 10.

One solution to this problem is to use a **write buffer**. A write buffer stores the data while they are waiting to be written to memory. After writing the data into the cache and into the write buffer, the processor can continue execution. When a write to main memory completes, the entry in the write buffer is freed. If the write buffer is full when the processor reaches a write, the processor must stall until there is an empty position in the write buffer. Of course, if the rate at which the memory can complete writes is less than the rate at which the processor is generating writes, no amount of buffering can help, because writes are being generated faster than

the memory system can accept them.

write buffer

A queue that holds data while the data are waiting to be written to memory.

The rate at which writes are generated may also be *less* than the rate at which the memory can accept them, and yet stalls may still occur. This can happen when the writes occur in bursts. To reduce the occurrence of such stalls, processors usually increase the depth of the write buffer beyond a single entry.

The alternative to a write-through scheme is a scheme called **write-back**. In a write-back scheme, when a write occurs, the new value is written only to the block in the cache. The modified block is written to the lower level of the hierarchy when it is replaced. Write-back schemes can improve performance, especially when processors can generate writes as fast or faster than the writes can be handled by main memory; a write-back scheme is, however, more complex to implement than write-through.

write-back

A scheme that handles writes by updating values only to the block in the cache, then writing the modified block to the lower level of the hierarchy when the block is replaced.

In the rest of this section, we describe caches from real processors, and we examine how they handle both reads and writes. In Section 5.8, we will describe the handling of writes in more detail.

Elaboration

Writes introduce several complications into caches that are not present for reads. Here we discuss two of them: the policy on write misses and efficient implementation of writes in write-back caches.

Consider a miss in a write-through cache. The most common strategy is to allocate a block in the cache, called *write allocate*. The block is fetched from memory and then the appropriate portion of

the block is overwritten. An alternative strategy is to update the portion of the block in memory but not put it in the cache, called *no write allocate*. The motivation is that sometimes programs write entire blocks of data, such as when the operating system zeros a page of memory. In such cases, the fetch associated with the initial write miss may be unnecessary. Some computers allow the write allocation policy to be changed on a per-page basis.

Actually implementing stores efficiently in a cache that uses a write-back strategy is more complex than in a write-through cache. A write-through cache can write the data into the cache and read the tag; if the tag mismatches, then a miss occurs. Because the cache is write-through, the overwriting of the block in the cache is not catastrophic, since memory has the correct value. In a write-back cache, we must first write the block back to memory if the data in the cache are modified and we have a cache miss. If we simply overwrote the block on a store instruction before we knew whether the store had hit in the cache (as we could for a write-through cache), we would destroy the contents of the block, which is not backed up in the next lower level of the memory hierarchy.

In a write-back cache, because we cannot overwrite the block, stores either require two cycles (a cycle to check for a hit followed by a cycle to actually perform the write) or require a write buffer to hold that data—effectively allowing the store to take only one cycle by pipelining it. When a store buffer is used, the processor does the cache lookup and places the data in the store buffer during the normal cache access cycle. Assuming a cache hit, the new data are written from the store buffer into the cache on the next unused cache access cycle.

By comparison, in a write-through cache, writes can always be done in one cycle. We read the tag and write the data portion of the selected block. If the tag matches the address of the block being written, the processor can continue normally, since the correct block has been updated. If the tag does not match, the processor generates a write miss to fetch the rest of the block corresponding to that address.

Many write-back caches also include write buffers that are used to reduce the miss penalty when a miss replaces a modified block. In such a case, the modified block is moved to a write-back buffer associated with the cache while the requested block is read from

memory. The write-back buffer is later written back to memory. Assuming another miss does not occur immediately, this technique halves the miss penalty when a dirty block must be replaced.

An Example Cache: The Intrinsity FastMATH Processor

The Intrinsity FastMATH is an embedded microprocessor that uses the MIPS architecture and a simple cache implementation. Near the end of the chapter, we will examine the more complex cache designs of ARM and Intel microprocessors, but we start with this simple, yet real, example for pedagogical reasons. Figure 5.12 shows the organization of the Intrinsity FastMATH data cache. Note that the address size for this computer is just 32 bits, not 64 as in the rest of the book.

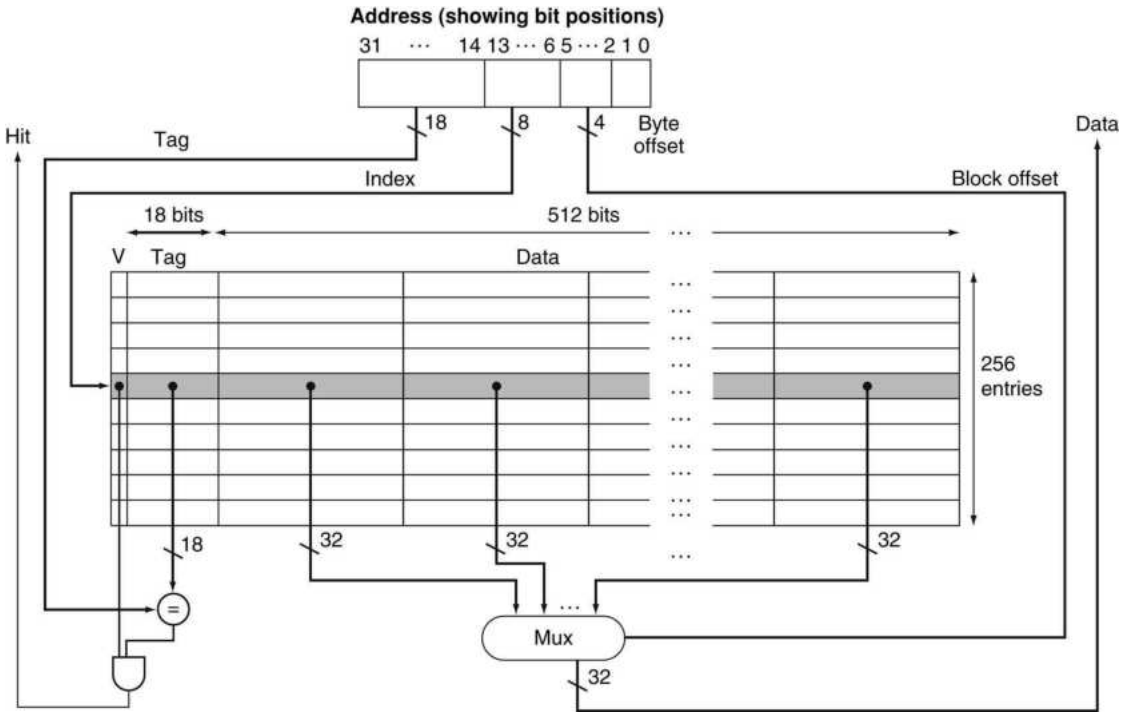


FIGURE 5.12 The 16 KiB caches in the Intrinsity FastMATH each contain 256 blocks with 16 words per block.

Note that the address size for this computer is just 32 bits. The tag field is 18 bits wide and the index field is 8 bits wide, while a 4-bit field (bits 5–2) is used to index the block and select the word from the block using a 16-to-1 multiplexor. In practice, to eliminate the multiplexor, caches use a separate large RAM for the data and a smaller RAM for the tags, with the block offset supplying the extra address bits for the large data RAM. In this case, the large RAM is 32 bits wide and must have 16 times as many words as blocks in the cache.

This processor has a 12-stage pipeline. When operating at peak speed, the processor can request both an instruction word and a data word on every clock. To satisfy the demands of the pipeline without stalling, separate instruction and data caches are used. Each cache is 16 KiB, or 4096 words, with 16-word blocks.

Read requests for the cache are straightforward. Because there are separate data and instruction caches, we need separate control signals to read and write each cache. (Remember that we need to update the instruction cache when a miss occurs.) Thus, the steps for a read request to either cache are as follows:

1. Send the address to the appropriate cache. The address comes either from the PC (for an instruction) or from the ALU (for data).
2. If the cache signals hit, the requested word is available on the data lines. Since there are 16 words in the desired block, we need to select the right one. A block index field is used to control the multiplexor (shown at the bottom of the figure), which selects the requested word from the 16 words in the indexed block.
3. If the cache signals miss, we send the address to the main memory. When the memory returns with the data, we write it into the cache and then read it to fulfill the request.

For writes, the Intrinsity FastMATH offers both write-through and write-back, leaving it up to the operating system to decide which strategy to use for an application. It has a one-entry write buffer.

What cache miss rates are attained with a cache structure like that used by the Intrinsity FastMATH? Figure 5.13 shows the miss rates for the instruction and data caches. The combined miss rate is the effective miss rate per reference for each program after accounting for the differing frequency of instruction and data accesses.

Instruction miss rate	Data miss rate	Effective combined miss rate
0.4%	11.4%	3.2%

FIGURE 5.13 Approximate instruction and data miss rates for the Intrinsity FastMATH processor for SPEC CPU2000 benchmarks.
The combined miss rate is the effective miss rate seen for the combination of the 16 KiB instruction cache and 16 KiB data cache. It is obtained by weighting the instruction and data individual miss rates by the frequency of instruction and data references.

Although miss rate is an important characteristic of cache designs, the ultimate measure will be the effect of the memory system on program execution time; we'll see how miss rate and execution time are related shortly.

Elaboration

A combined cache with a total size equal to the sum of the two

split caches will usually have a better hit rate. This higher rate occurs because the combined cache does not rigidly divide the number of entries that may be used by instructions from those that may be used by data. Nonetheless, almost all processors today use split instruction and data caches to increase cache *bandwidth* to match what modern pipelines expect. (There may also be fewer conflict misses; see Section 5.8.)

Here are miss rates for caches the size of those found in the Intrinsity FastMATH processor, and for a combined cache whose size is equal to the sum of the two caches:

- Total cache size: 32 KiB
- Split cache effective miss rate: 3.24%
- Combined cache miss rate: 3.18%

The miss rate of the split cache is only slightly worse.

The advantage of doubling the cache bandwidth, by supporting both an instruction and data access simultaneously, easily overcomes the disadvantage of a slightly increased miss rate. This observation cautions us that we cannot use miss rate as the sole measure of cache performance, as Section 5.4 shows.

split cache

A scheme in which a level of the memory hierarchy is composed of two independent caches that operate in parallel with each other, with one handling instructions and one handling data.

Summary

We began the previous section by examining the simplest of caches: a direct-mapped cache with a one-word block. In such a cache, both hits and misses are simple, since a word can go in exactly one location and there is a separate tag for every word. To keep the cache and memory consistent, a write-through scheme can be used, so that every write into the cache also causes memory to be updated. The alternative to write-through is a write-back scheme that copies a block back to memory when it is replaced; we'll discuss this scheme further in upcoming sections.

To take advantage of spatial locality, a cache must have a block size larger than one word. The use of a bigger block decreases the miss rate and improves the efficiency of the cache by reducing the

amount of tag storage relative to the amount of data storage in the cache. Although a larger block size decreases the miss rate, it can also increase the miss penalty. If the miss penalty increased linearly with the block size, larger blocks could easily lead to lower performance.

To avoid performance loss, the bandwidth of main memory is increased to transfer cache blocks more efficiently. Common methods for increasing bandwidth external to the DRAM are making the memory wider and interleaving. DRAM designers have steadily improved the interface between the processor and memory to increase the bandwidth of burst mode transfers to reduce the cost of larger cache block sizes.

Check Yourself

The speed of the memory system affects the designer's decision on the size of the cache block. Which of the following cache designer guidelines is generally valid?

1. The shorter the memory latency, the smaller the cache block
2. The shorter the memory latency, the larger the cache block
3. The higher the memory bandwidth, the smaller the cache block
4. The higher the memory bandwidth, the larger the cache block

5.4 Measuring and Improving Cache Performance

In this section, we begin by examining ways to measure and analyze cache performance. We then explore two different techniques for improving cache performance. One focuses on reducing the miss rate by reducing the probability that two distinct memory blocks will contend for the same cache location. The second technique reduces the miss penalty by adding an additional level to the hierarchy. This technique, called *multilevel caching*, first appeared in high-end computers selling for more than $100,000 in 1990; since then it has become common on personal mobile devices selling for a few hundred dollars!

CPU time can be divided into the clock cycles that the CPU spends executing the program and the clock cycles that the CPU

spends waiting for the memory system. Normally, we assume that the costs of cache accesses that are hits are part of the normal CPU execution cycles. Thus,

$$\text{CPU time} = (\text{CPU execution clock cycles} + \text{Memory-stall clock cycles}) \times \text{Clock cycle time}$$

The memory-stall clock cycles come primarily from cache misses, and we make that assumption here. We also restrict the discussion to a simplified model of the memory system. In real processors, the stalls generated by reads and writes can be quite complex, and accurate performance prediction usually requires very detailed simulations of the processor and memory system.

Memory-stall clock cycles can be defined as the sum of the stall cycles coming from reads plus those coming from writes:

$$\text{Memory-stall clock cycles} = (\text{Read-stall cycles} + \text{Write-stall cycles})$$

The read-stall cycles can be defined in terms of the number of read accesses per program, the miss penalty in clock cycles for a read, and the read miss rate:

$$\text{Read-stall cycles} = \frac{\text{Reads}}{\text{Program}} \times \text{Read miss rate} \times \text{Read miss penalty}$$

Writes are more complicated. For a write-through scheme, we have two sources of stalls: write misses, which usually require that we fetch the block before continuing the write (see the *Elaboration* on page 386 for more details on dealing with writes), and write buffer stalls, which occur when the write buffer is full when a write happens. Thus, the cycles stalled for writes equal the sum of these two:

$$\text{Write-stall cycles} = \left(\frac{\text{Writes}}{\text{Program}} \times \text{Write miss rate} \times \text{Write miss penalty}\right) + \text{Write buffer stalls}$$

Because the write buffer stalls depend on the proximity of writes, and not just the frequency, it is impossible to give a simple equation to compute such stalls. Fortunately, in systems with a reasonable write buffer depth (e.g., four or more words) and a memory capable of accepting writes at a rate that significantly exceeds the average write frequency in programs (e.g., by a factor of 2), the write buffer stalls will be small, and we can safely ignore them. If a system did not meet these criteria, it would not be well designed; instead, the designer should have used either a deeper write buffer or a write-back organization.

Write-back schemes also have potential additional stalls arising from the need to write a cache block back to memory when the block is replaced. We will discuss this more in Section 5.8.

In most write-through cache organizations, the read and write miss penalties are the same (the time to fetch the block from memory). If we assume that the write buffer stalls are negligible, we can combine the reads and writes by using a single miss rate and the miss penalty:

$$\text{Memory-stall clock cycles} = \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}$$

We can also factor this as

$$\text{Memory-stall clock cycles} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

Let's consider a simple example to help us understand the impact of cache performance on processor performance.

Calculating Cache Performance

Example

Assume the miss rate of an instruction cache is 2% and the miss rate of the data cache is 4%. If a processor has a CPI of 2 without any memory stalls, and the miss penalty is 100 cycles for all misses, determine how much faster a processor would run with a perfect

cache that never missed. Assume the frequency of all loads and stores is 36%.

Answer

The number of memory miss cycles for instructions in terms of the Instruction count (I) is

$$\text{Instruction miss cycles} = \text{I} \times 2\% \times 100 = 2.00 \times \text{I}$$

As the frequency of all loads and stores is 36%, we can find the number of memory miss cycles for data references:

$$\text{Data miss cycles} = \text{I} \times 36\% \times 4\% \times 100 = 1.44 \times \text{I}$$

The total number of memory-stall cycles is 2.00 I +1.44 I =3.44 I. This is more than three cycles of memory stall per instruction. Accordingly, the total CPI including memory stalls is 2 +3.44 =5.44. Since there is no change in instruction count or clock rate, the ratio of the CPU execution times is

$$\frac{\text{CPU time with stalls}}{\text{CPU time with perfect cache}} = \frac{\text{I} \times \text{CPI}_{\text{stall}} \times \text{Clock cycle}}{\text{I} \times \text{CPI}_{\text{perfect}} \times \text{Clock cycle}} = \frac{\text{CPI}_{\text{stall}}}{\text{CPI}_{\text{perfect}}} = \frac{5.44}{2}$$

The performance with the perfect cache is better by $\frac{5.44}{2} = 2.72$.

What happens if the processor is made faster, but the memory system is not? The amount of time spent on memory stalls will take up an increasing fraction of the execution time; Amdahl's Law, which we examined in Chapter 1, reminds us of this fact. A few simple examples show how serious this problem can be. Suppose we speed-up the computer in the previous example by reducing its CPI from 2 to 1 without changing the clock rate, which might be done with an improved pipeline. The system with cache misses

would then have a CPI of 1 +3.44 =4.44, and the system with the perfect cache would be

$$\frac{4.44}{1} = 4.44 \text{ times as fast.}$$

The amount of execution time spent on memory stalls would have risen from

$$\frac{3.44}{5.44} = 63\%$$

to

$$\frac{3.44}{4.44} = 77\%$$

Similarly, increasing the clock rate without changing the memory system also increases the performance lost due to cache misses.

The previous examples and equations assume that the hit time is not a factor in determining cache performance. Clearly, if the hit time increases, the total time to access a word from the memory system will increase, possibly causing an increase in the processor cycle time. Although we will see additional examples of what can raise hit time shortly, one example is increasing the cache size. A larger cache could clearly have a bigger access time, just as, if your desk in the library was very large (say, 3 square meters), it would take longer to locate a book on the desk. An increase in hit time likely adds another stage to the pipeline, since it may take multiple cycles for a cache hit. Although it is more complex to calculate the performance impact of a deeper pipeline, at some point the increase in hit time for a larger cache could dominate the improvement in hit rate, leading to a decrease in processor performance.

To capture the fact that the time to access data for both hits and misses affects performance, designers sometime use *average memory access time* (AMAT) as a way to examine alternative cache designs. Average memory access time is the average time to access memory considering both hits and misses and the frequency of different

accesses; it is equal to the following:

$$\text{AMAT} = \text{Time for a hit} + \text{Miss rate} \times \text{Miss penalty}$$

Calculating Average Memory Access Time

Example

Find the AMAT for a processor with a 1 ns clock cycle time, a miss penalty of 20 clock cycles, a miss rate of 0.05 misses per instruction, and a cache access time (including hit detection) of 1 clock cycle. Assume that the read and write miss penalties are the same and ignore other write stalls.

Answer

The average memory access time per instruction is

$$\begin{aligned}\text{AMAT} &= \text{Time for a hit} + \text{Miss rate} \times \text{Miss penalty} \\ &= 1 + 0.05 \times 20 \\ &= 2 \text{ clock cycles}\end{aligned}$$

or 2 ns.

The next subsection discusses alternative cache organizations that decrease miss rate but may sometimes increase hit time; additional examples appear in Section 5.16.

Reducing Cache Misses by More Flexible Placement of Blocks

So far, when we put a block in the cache, we have used a simple placement scheme: A block can go in exactly one place in the cache. As mentioned earlier, it is called *direct mapped* because there is a direct mapping from any block address in memory to a single location in the upper level of the hierarchy. However, there is actually a whole range of schemes for placing blocks. Direct mapped, where a block can be placed in exactly one location, is at one extreme.

At the other extreme is a scheme where a block can be placed in *any* location in the cache. Such a scheme is called **fully associative**, because a block in memory may be associated with any entry in the cache. To find a given block in a fully associative cache, all the entries in the cache must be searched because a block can be placed in any one. To make the search practical, it is done in parallel with a comparator associated with each cache entry. These comparators significantly increase the hardware cost, effectively making fully associative placement practical only for caches with small numbers of blocks.

fully associative cache

A cache structure in which a block can be placed in any location in the cache.

The middle range of designs between direct mapped and fully associative is called **set associative**. In a set-associative cache, there are a fixed number of locations where each block can be placed. A set-associative cache with n locations for a block is called an n-way set-associative cache. An n-way set-associative cache consists of a number of sets, each of which consists of n blocks. Each block in the memory maps to a unique *set* in the cache given by the index field, and a block can be placed in *any* element of that set. Thus, a set-associative placement combines direct-mapped placement and fully associative placement: a block is directly mapped into a set, and then all the blocks in the set are searched for a match. For example, Figure 5.14 shows where block 12 may be put in a cache with eight blocks total, according to the three block placement policies.

set-associative cache

A cache that has a fixed number of locations (at least two) where each block can be placed.

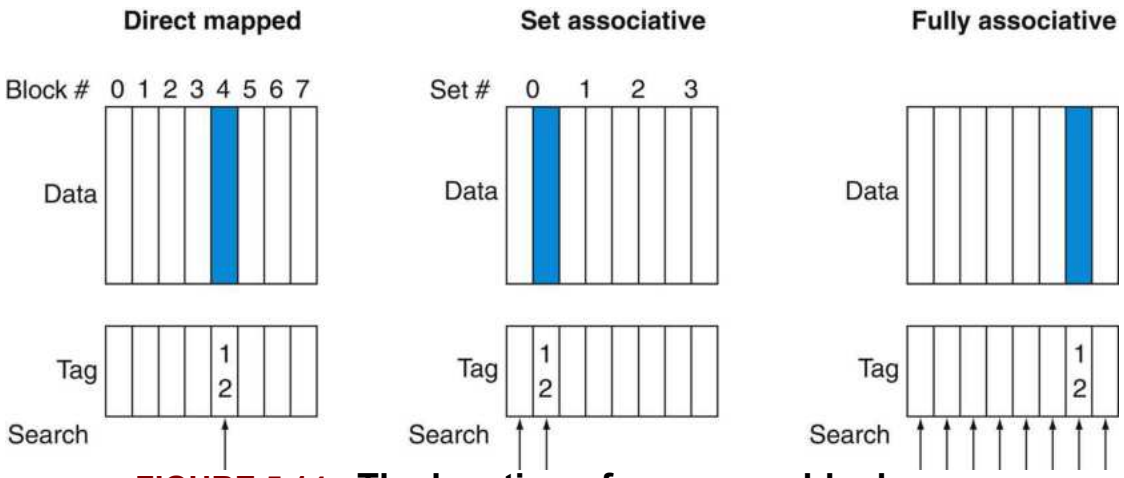


FIGURE 5.14 The location of a memory block whose address is 12 in a cache with eight blocks varies for direct-mapped, set-associative, and fully associative placement.
In direct-mapped placement, there is only one cache block where memory block 12 can be found, and that block is given by (12 modulo 8)=4. In a two-way set-associative cache, there would be four sets, and memory block 12 must be in set (12 mod 4)=0; the memory block could be in either element of the set. In a fully associative placement, the memory block for block address 12 can appear in any of the eight cache blocks.

Remember that in a direct-mapped cache, the position of a memory block is given by

(Block number) modulo (Number of *blocks* in the cache)

In a set-associative cache, the set containing a memory block is given by

(Block number) modulo (Number of *sets* in the cache)

Since the block may be placed in any element of the set, *all the tags of all the elements of the set* must be searched. In a fully associative cache, the block can go anywhere, and *all tags of all the blocks in the cache* must be searched.

We can also think of all block placement strategies as a variation on set associativity. Figure 5.15 shows the possible associativity

structures for an eight-block cache. A direct-mapped cache is just a one-way set-associative cache: each cache entry holds one block and each set has one element. A fully associative cache with *m* entries is simply an *m*-way set-associative cache; it has one set with *m* blocks, and an entry can reside in any block within that set.

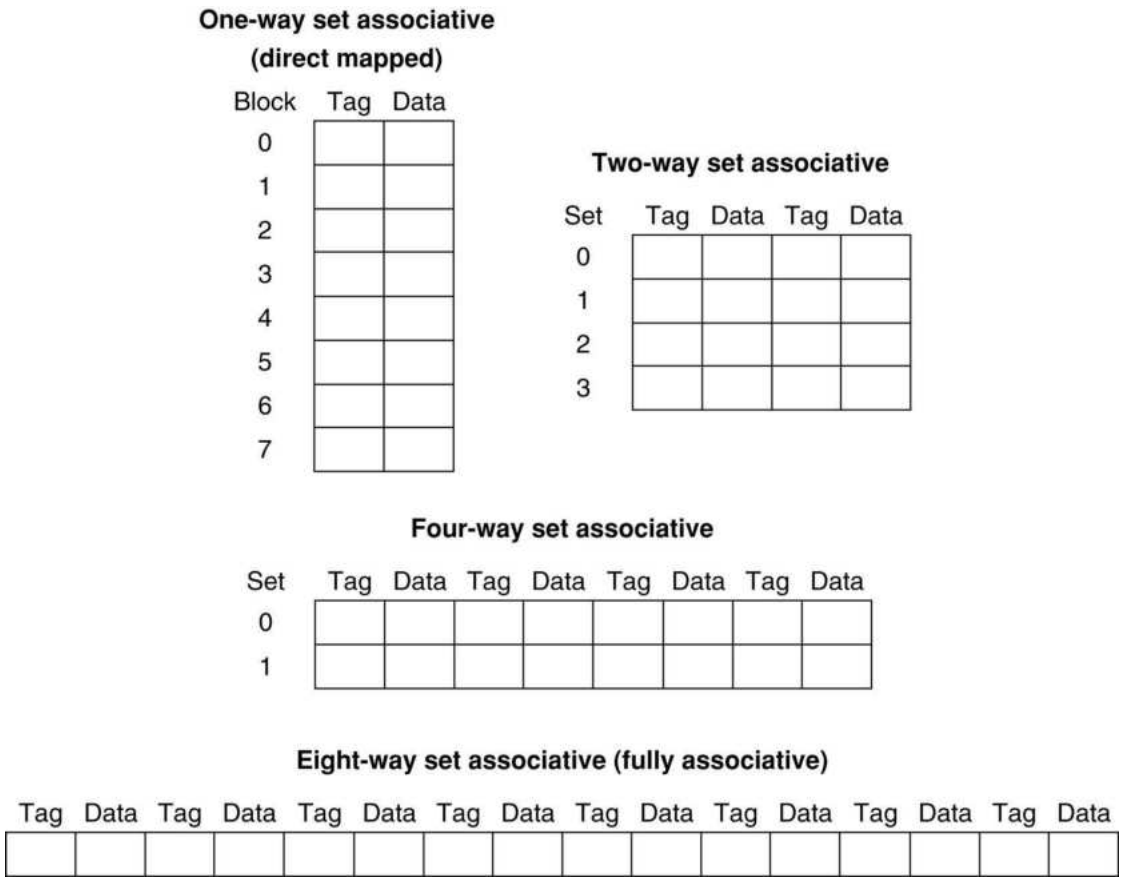


FIGURE 5.15 An eight-block cache configured as direct-mapped, two-way set associative, four-way set associative, and fully associative.
The total size of the cache in blocks is equal to the number of sets times the associativity. Thus, for a fixed cache size, increasing the associativity decreases the number of sets while increasing the number of elements per set. With eight blocks, an eight-way set-associative cache is the same as a fully associative cache.

The advantage of increasing the degree of associativity is that it usually decreases the miss rate, as the next example shows. The main disadvantage, which we discuss in more detail shortly, is a potential increase in the hit time.

Misses and Associativity in Caches

Example

Assume there are three small caches, each consisting of four one-word blocks. One cache is fully associative, a second is two-way set

associative, and the third is direct-mapped. Find the number of misses for each cache organization given the following sequence of block addresses: 0, 8, 0, 6, and 8.

Answer

The direct-mapped case is easiest. First, let's determine to which cache block each block address maps:

Block address	Cache block
0	(0 modulo 4)=0
6	(6 modulo 4)=2
8	(8 modulo 4)=0

Now we can fill in the cache contents after each reference, using a blank entry to mean that the block is invalid, colored text to show a new entry added to the cache for the associated reference, and plain text to show an old entry in the cache:

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		0	1	2	3
0	miss	Memory[0]			
8	miss	Memory[8]			
0	miss	Memory[0]			
6	miss	Memory[0]		Memory[6]	
8	miss	Memory[8]		Memory[6]	

The direct-mapped cache generates five misses for the five accesses.

The set-associative cache has two sets (with indices 0 and 1) with two elements per set. Let's first determine to which set each block address maps:

Block address	Cache set
0	(0 modulo 2)=0
6	(6 modulo 2)=0
8	(8 modulo 2)=0

Because we have a choice of which entry in a set to replace on a miss, we need a replacement rule. Set-associative caches usually replace the least recently used block within a set; that is, the block that was used furthest in the past is replaced. (We will discuss other replacement rules in more detail shortly.) Using this

replacement rule, the contents of the set-associative cache after each reference look like this:

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Set 0	Set 0	Set 1	Set 1
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[6]		
8	miss	Memory[8]	Memory[6]		

Notice that when block 6 is referenced, it replaces block 8, since block 8 has been less recently referenced than block 0. The two-way set-associative cache has four misses, one less than the direct-mapped cache.

The fully associative cache has four cache blocks (in a single set); any memory block can be stored in any cache block. The fully associative cache has the best performance, with only three misses:

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Block 0	Block 1	Block 2	Block 3
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[8]	Memory[6]	
8	hit	Memory[0]	Memory[8]	Memory[6]	

For this series of references, three misses is the best we can do, because three unique block addresses are accessed. Notice that if we had eight blocks in the cache, there would be no replacements in the two-way set-associative cache (check this for yourself), and it would have the same number of misses as the fully associative cache. Similarly, if we had 16 blocks, all three caches would have the identical number of misses. Even this trivial example shows that cache size and associativity are not independent in determining cache performance.

How much of a reduction in the miss rate is achieved by

associativity? Figure 5.16 shows the improvement for a 64 KiB data cache with a 16-word block, and associativity ranging from direct-mapped to eight-way. Going from one-way to two-way associativity decreases the miss rate by about 15%, but there is little further improvement in going to higher associativity.

Associativity	Data miss rate
1	10.3%
2	8.6%
4	8.3%
8	8.1%

FIGURE 5.16 The data cache miss rates for an organization like the Intrinsity FastMATH processor for SPEC CPU2000 benchmarks with associativity varying from one-way to eight-way. These results for 10 SPEC CPU2000 programs are from Hennessy and Patterson (2003).

Locating a Block in the Cache

Now, let's consider the task of finding a block in a cache that is set associative. Just as in a direct-mapped cache, each block in a set-associative cache includes an address tag that gives the block address. The tag of every cache block within the appropriate set is checked to see if it matches the block address from the processor. Figure 5.17 decomposes the address. The index value is used to select the set containing the address of interest, and the tags of all the blocks in the set must be searched. Because speed is of the essence, all the tags in the selected set are searched in parallel. As in a fully associative cache, a sequential search would make the hit time of a set-associative cache too slow.

FIGURE 5.17 The three portions of an address in a set-associative or direct-mapped cache. The index is used to select the set, then the tag is used to choose the block by comparison with the blocks in

the selected set. The block offset is the address of the desired data within the block.

If the total cache size is kept the same, increasing the associativity raises the number of blocks per set, which is the number of simultaneous compares needed to perform the search in parallel: each increase by a factor of 2 in associativity doubles the number of blocks per set and halves the number of sets. Accordingly, each factor-of-2 increase in associativity decreases the size of the index by 1 bit and expands the size of the tag by 1 bit. In a fully associative cache, there is effectively only one set, and all the blocks must be checked in parallel. Thus, there is no index, and the entire address, excluding the block offset, is compared against the tag of every block. In other words, we search the full cache without any indexing.

In a direct-mapped cache, only a single comparator is needed, because the entry can be in only one block, and we access the cache simply by indexing. Figure 5.18 shows that in a four-way set-associative cache, four comparators are needed, together with a 4-to-1 multiplexor to choose among the four potential members of the selected set. The cache access consists of indexing the appropriate set and then searching the tags of the set. The costs of an associative cache are the extra comparators and any delay imposed by having to do the compare and select from among the elements of the set.

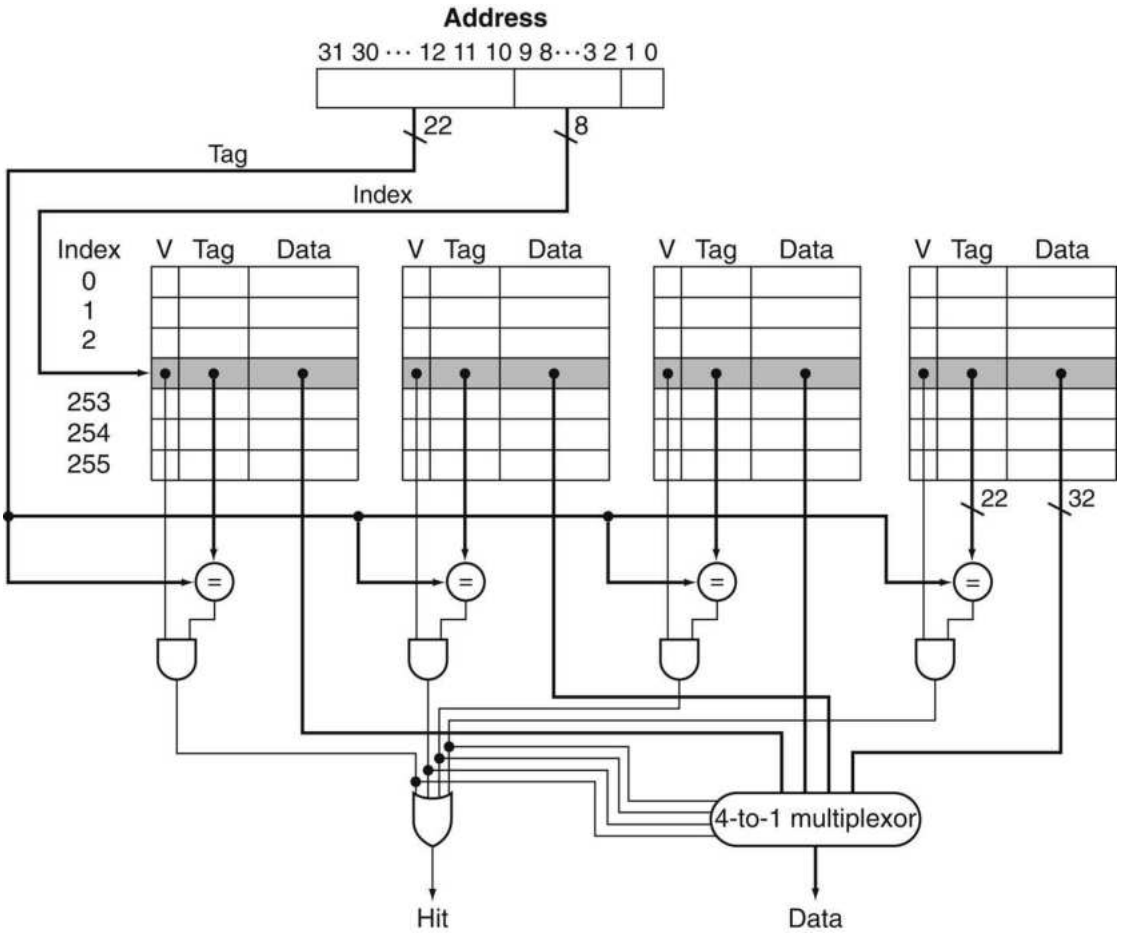


FIGURE 5.18 The implementation of a four-way set-associative cache requires four comparators and a 4-to-1 multiplexor.

The comparators determine which element of the selected set (if any) matches the tag. The output of the comparators is used to select the data from one of the four blocks of the indexed set, using a multiplexor with a decoded select signal. In some implementations, the Output enable signals on the data portions of the cache RAMs can be used to select the entry in the set that drives the output. The Output enable signal comes from the comparators, causing the element that matches to drive the data outputs. This organization eliminates the need for the multiplexor.

The choice among direct-mapped, set-associative, or fully associative mapping in any memory hierarchy will depend on the cost of a miss versus the cost of implementing associativity, both in time and in extra hardware.

Elaboration

A *Content Addressable Memory* (CAM) is a circuit that combines comparison and storage in a single device. Instead of supplying an address and reading a word like a RAM, you send the data and the CAM looks to see if it has a copy and returns the index of the matching row. CAMs mean that cache designers can afford to implement much higher set associativity than if they needed to build the hardware out of SRAMs and comparators. In 2013, the greater size and power of CAM generally leads to two-way and four-way set associativity being built from standard SRAMs and comparators, with eight-way and above built using CAMs.

Choosing Which Block to Replace

When a miss occurs in a direct-mapped cache, the requested block can go in exactly one position, and the block occupying that position must be replaced. In an associative cache, we have a choice of where to place the requested block, and hence a choice of which block to replace. In a fully associative cache, all blocks are candidates for replacement. In a set-associative cache, we must choose among the blocks in the selected set.

The most commonly used scheme is **least recently used (LRU)**, which we used in the previous example. In an LRU scheme, the block replaced is the one that has been unused for the longest time. The set-associative example on page 397 uses LRU, which is why we replaced Memory(0) instead of Memory(6).

least recently used (LRU)

A replacement scheme in which the block replaced is the one that has been unused for the longest time.

LRU replacement is implemented by keeping track of when each element in a set was used relative to the other elements in the set. For a two-way set-associative cache, tracking when the two elements were used can be implemented by keeping a single bit in each set and setting the bit to indicate an element whenever that element is referenced. As associativity increases, implementing LRU gets harder; in Section 5.8, we will see an alternative scheme

for replacement.

Size of Tags versus Set Associativity

Example

Increasing associativity requires more comparators and more tag bits per cache block. Assuming a cache of 4096 blocks, a four-word block size, and a 64-bit address, find the total number of sets and the total number of tag bits for caches that are direct-mapped, two-way and four-way set associative, and fully associative.

Answer

Since there are 16 (=2^4) bytes per block, a 64-bit address yields 64 −4 =60 bits to be used for index and tag. The direct-mapped cache has the same number of sets as blocks, and hence 12 bits of index, since $\log_2(4096)=12$; hence, the total number is (60 −12)×4096 =48 ×4096 =197 K tag bits.

Each degree of associativity decreases the number of sets by a factor of 2 and thus decreases the number of bits used to index the cache by 1 and increases the number of bits in the tag by 1. Thus, for a two-way set-associative cache, there are 2048 sets, and the total number of tag bits is (60 −11)×2 ×2048 =98 ×2048 =401 Kbits. For a four-way set-associative cache, the total number of sets is 1024, and the total number is (60 −10)×4 ×1024 =100 ×1024 =205 K tag bits.

For a fully associative cache, there is only one set with 4096 blocks, and the tag is 60 bits, leading to 60 ×4096 ×1 =246 K tag bits.

Reducing the Miss Penalty Using Multilevel Caches

All modern computers make use of caches. To close the gap further between the fast clock rates of modern processors and the increasingly long time required to access DRAMs, most microprocessors support an additional level of caching. This second-level cache is normally on the same chip and is accessed whenever a miss occurs in the primary cache. If the second-level cache contains the desired data, the miss penalty for the first-level

cache will be essentially the access time of the second-level cache, which will be much less than the access time of main memory. If neither the primary nor the secondary cache contains the data, a main memory access is required, and a larger miss penalty is incurred.

How significant is the performance improvement from the use of a secondary cache? The next example shows us.

Performance of Multilevel Caches

Example

Suppose we have a processor with a base CPI of 1.0, assuming all references hit in the primary cache, and a clock rate of 4 GHz. Assume a main memory access time of 100 ns, including all the miss handling. Suppose the miss rate per instruction at the primary cache is 2%. How much faster will the processor be if we add a secondary cache that has a 5- ns access time for either a hit or a miss and is large enough to reduce the miss rate to main memory to 0.5%?

Answer

The miss penalty to main memory is

$$\frac{100\text{ ns}}{0.25\frac{\text{ns}}{\text{clock cycle}}} = 400 \text{ clock cycles}$$

The effective CPI with one level of caching is given by

$$\text{Total CPI} = \text{Base CPI} + \text{Memory-stall cycles per instruction}$$

For the processor with one level of caching,

$$\text{Total CPI} = 1.0 + \text{Memory-stall cycles per instruction} = 1.0 + 2\% \times 400 = 9$$

With two levels of caching, a miss in the primary (or first-level) cache can be satisfied either by the secondary cache or by main

memory. The miss penalty for an access to the second-level cache is

$$\frac{5\text{ns}}{0.25\frac{\text{ns}}{\text{clock cycle}}} = 20 \text{ clock cycles}$$

If the miss is satisfied in the secondary cache, then this is the entire miss penalty. If the miss needs to go to main memory, then the total miss penalty is the sum of the secondary cache access time and the main memory access time.

Thus, for a two-level cache, total CPI is the sum of the stall cycles from both levels of cache and the base CPI:

$$\begin{aligned}\text{Total CPI} &= 1 + \text{Primary stalls per instruction} + \text{Secondary stalls per instruction}\\ &= 1 + 2\% \times 20 + 0.5\% \times 400 = 1 + 0.4 + 2.0 = 3.4\end{aligned}$$

Thus, the processor with the secondary cache is faster by

$$\frac{9.0}{3.4} = 2.6$$

Alternatively, we could have computed the stall cycles by summing the stall cycles of those references that hit in the secondary cache ((2%−0.5%)×20 =0.3). Those references that go to main memory, which must include the cost to access the secondary cache as well as the main memory access time, are (0.5%×(20 +400)=2.1). The sum, 1.0 +0.3 +2.1, is again 3.4.

The design considerations for a primary and secondary cache are significantly different, because the presence of the other cache changes the best choice versus a single-level cache. In particular, a two-level cache structure allows the primary cache to focus on minimizing hit time to yield a shorter clock cycle or fewer pipeline stages, while allowing the secondary cache to focus on miss rate to reduce the penalty of long memory access times.

The effect of these changes on the two caches can be seen by comparing each cache to the optimal design for a single level of

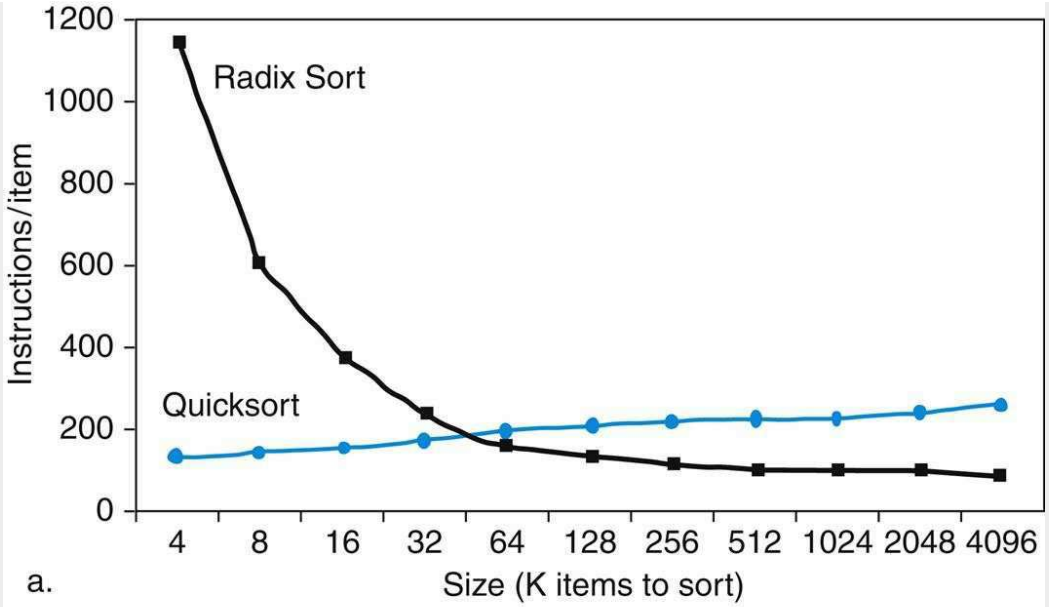
cache. In comparison to a single-level cache, the primary cache of a **multilevel cache** is often smaller. Furthermore, the primary cache may use a smaller block size, to go with the smaller cache size and also to reduce the miss penalty. In comparison, the secondary cache will be much larger than in a single-level cache, since the access time of the secondary cache is less critical. With a larger total size, the secondary cache may use a larger block size than appropriate with a single-level cache. It often uses higher associativity than the primary cache given the focus of reducing miss rates.

multilevel cache

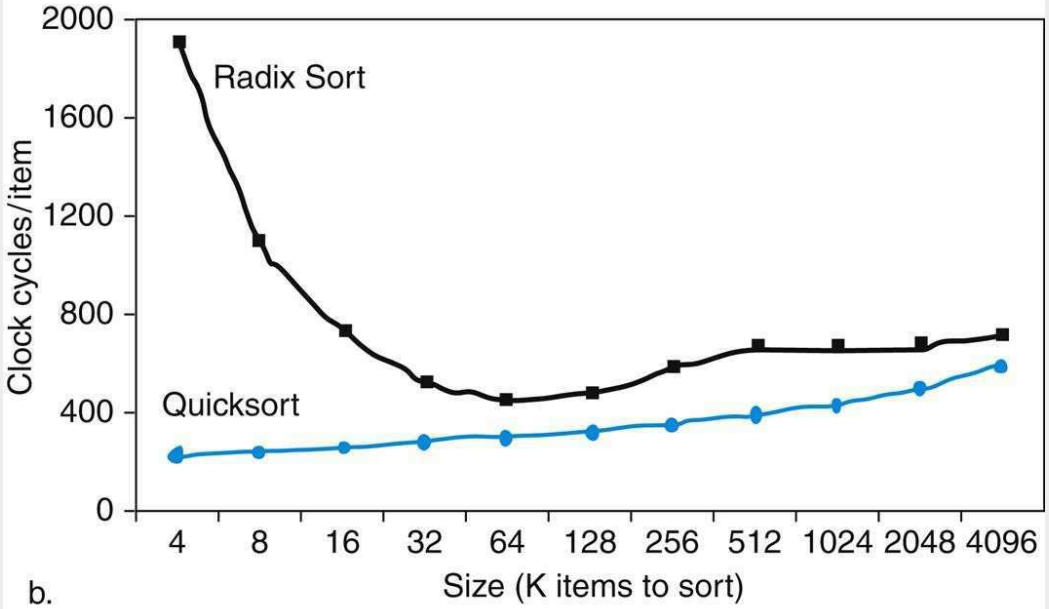
A memory hierarchy with multiple levels of caches, rather than just a cache and main memory.

Understanding Program Performance

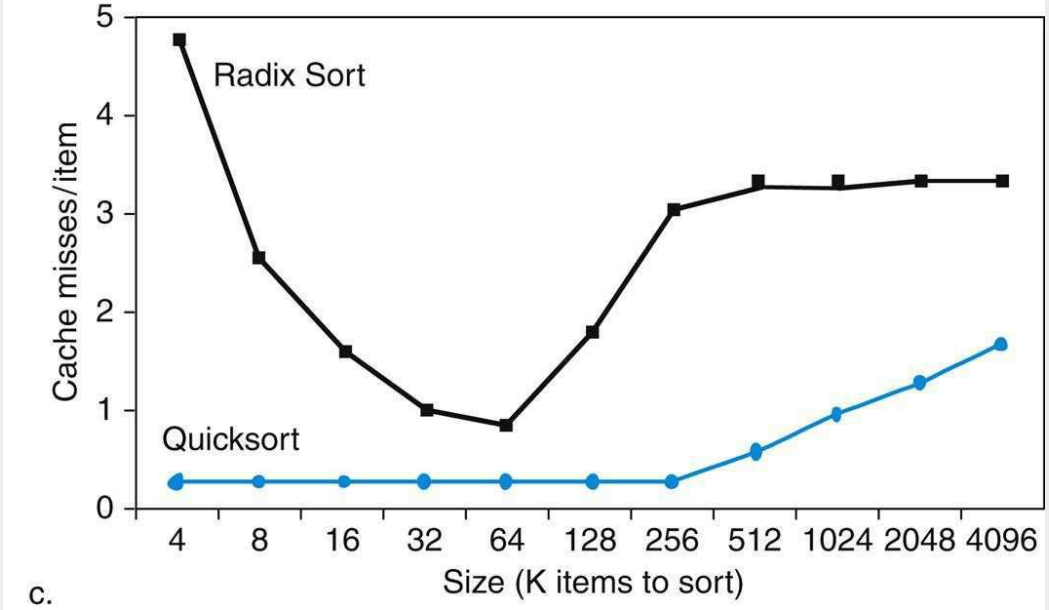
Sorting has been exhaustively analyzed to find better algorithms: Bubble Sort, Quicksort, Radix Sort, and so on. Figure 5.19(a) shows instructions executed by item searched for Radix Sort versus Quicksort. As expected, for large arrays, Radix Sort has an algorithmic advantage over Quicksort in terms of number of operations. Figure 5.19(b) shows time per key instead of instructions executed. We see that the lines start on the same trajectory as in Figure 5.19(a), but then the Radix Sort line diverges as the data to sort increase. What is going on? Figure 5.19(c) answers by looking at the cache misses per item sorted: Quicksort consistently has many fewer misses per item to be sorted.


1200
1000
800
600
400
200
0
Instructions/item
Radix Sort
Quicksort
4
8
16
32
64
128
256
512
1024
2048
4096
Size (K items to sort)

a.


2000
1600
1200
800
400
0
Clock cycles/item
Radix Sort
Quicksort
4
8
16
32
64
128
256
512
1024
2048
4096
Size (K items to sort)

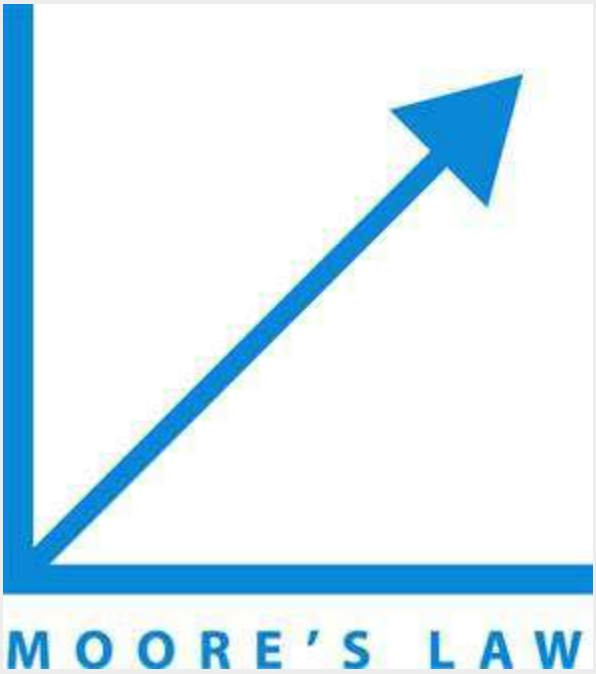
b.


5
4
3
2
1
0
Cache misses/item
Radix Sort
Quicksort
4
8
16
32
64
128
256
512
1024
2048
4096
Size (K items to sort)

c.

FIGURE 5.19 Comparing Quicksort and Radix Sort by (a) instructions executed per item sorted, (b) time per item sorted, and (c) cache misses per item sorted.
These data are from a paper by LaMarca and Ladner [1996]. Due to such results, new versions of Radix Sort have been invented that take memory hierarchy into account, to regain its algorithmic advantages (see Section 5.15). The basic idea of cache optimizations is to use all the data in a block repeatedly before they are replaced on a miss.

Alas, standard algorithmic analysis often ignores the impact of the memory hierarchy. As faster clock rates and **Moore's Law** allow architects to squeeze all the performance out of a stream of instructions, using the memory hierarchy well is vital to high performance. As we said in the introduction, understanding the behavior of the memory hierarchy is critical to understanding the performance of programs on today's computers.



Software Optimization via Blocking

Given the importance of the memory hierarchy to program performance, not surprisingly many software optimizations were invented that can dramatically improve performance by reusing data within the cache and hence lower miss rates due to improved temporal locality.

When dealing with arrays, we can get good performance from the memory system if we store the array in memory so that accesses to the array are sequential in memory. Suppose that we are dealing with multiple arrays, however, with some arrays accessed by rows and some by columns. Storing the arrays row-by-row (called *row major order*) or column-by-column (*column major order*) does not solve the problem because both rows and columns are used in every loop iteration.

Instead of operating on entire rows or columns of an array, *blocked* algorithms operate on submatrices or *blocks*. The goal is to maximize accesses to the data loaded into the cache before the data are replaced; that is, improve temporal locality to reduce cache misses.

For example, the inner loops of DGEMM (lines 4 through 9 of Figure 3.22 in Chapter 3) are

```
for (int j = 0; j < n; ++j)
 {
  double cij = C[i+j*n]; /* cij = C[i][j] */
  for( int k = 0; k < n; k++ )
   cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
  C[i+j*n] = cij; /* C[i][j] = cij */
  }
}
```

It reads all N-by-N elements of B, reads the same N elements in what corresponds to one row of A repeatedly, and writes what corresponds to one row of N elements of C. (The comments make the rows and columns of the matrices easier to identify.) Figure 5.20 gives a snapshot of the accesses to the three arrays. A dark shade indicates a recent access, a light shade indicates an older access, and white means not yet accessed.

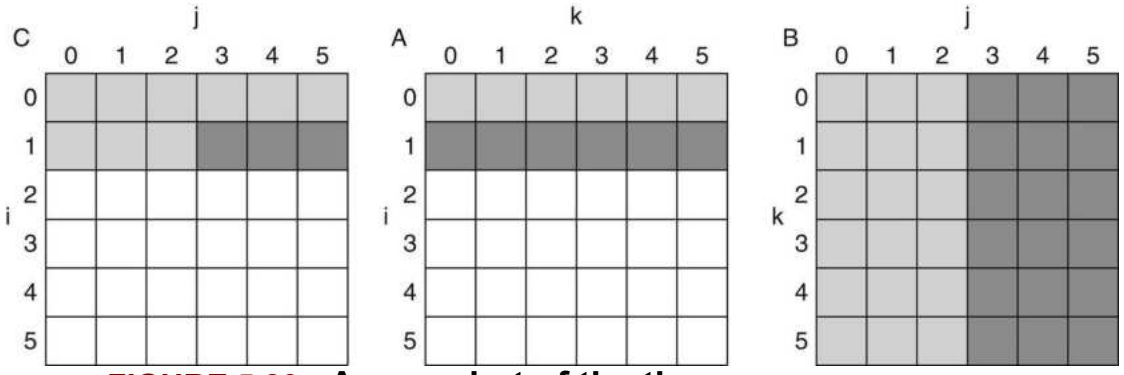


FIGURE 5.20 A snapshot of the three arrays `C`, `A`, and `B` when `N` = 6 and `i` = 1.
The age of accesses to the array elements is indicated by shade: white means not yet touched, light means older accesses, and dark means newer accesses. Compared to Figure 5.22, elements of `A` and `B` are read repeatedly to calculate new elements of `C`. The variables `i`, `j`, and `k` are shown along the rows or columns used to access the arrays.

The number of capacity misses clearly depends on `N` and the size of the cache. If it can hold all three `N`-by-`N` matrices, then all is well, provided there are no cache conflicts. We purposely picked the matrix size to be 32 by 32 in DGEMM for Chapters 3 and 4 so that this would be the case. Each matrix is 32 ×32 =1024 elements and each element is 8 bytes, so the three matrices occupy 24 KiB, which comfortably fit in the 32 KiB data cache of the Intel Core i7 (Sandy Bridge).

If the cache can hold one `N`-by-`N` matrix and one row of `N`, then at least the `i`th row of `A` and the array `B` may stay in the cache. Less than that and misses may occur for both `B` and `C`. In the worst case, there would be $2 \texttt{N}^3+\texttt{N}^2$ memory words accessed for $\texttt{N}^3$ operations.

To ensure that the elements being accessed can fit in the cache, the original code is changed to compute on a submatrix. Hence, we essentially invoke the version of DGEMM from Figure 4.78 in Chapter 4 repeatedly on matrices of size `BLOCKSIZE` by `BLOCKSIZE`. `BLOCKSIZE` is called the *blocking factor*.

Figure 5.21 shows the blocked version of DGEMM. The function `do_block` is DGEMM from Figure 3.22 with three new parameters `si`, `sj`, and `sk` to specify the starting position of each submatrix of `A`, `B`, and `C`. The two inner loops of the `do_block` now compute in steps of size `BLOCKSIZE` rather than the full length of `B` and `C`. The gcc

optimizer removes any function call overhead by "inlining" the function; that is, it inserts the code directly to avoid the conventional parameter passing and return address bookkeeping instructions.

```
1  #define BLOCKSIZE 32
2  void do_block (int n, int si, int sj, int sk, double *A, double
3  *B, double *C)
4  {
5     for (int i = si; i < si+BLOCKSIZE; ++i)
6        for (int j = sj; j < sj+BLOCKSIZE; ++j)
7           {
8              double cij = C[i+j*n];/* cij = C[i][j] */
9              for( int k = sk; k < sk+BLOCKSIZE; k++ )
10                cij += A[i+k*n] * B[k+j*n];/* cij+=A[i][k]*B[k][j] */
11             C[i+j*n] = cij;/* C[i][j] = cij */
12          }
13 }
14 void dgemm (int n, double* A, double* B, double* C)
15 {
16    for ( int sj = 0; sj < n; sj += BLOCKSIZE )
17       for ( int si = 0; si < n; si += BLOCKSIZE )
18          for ( int sk = 0; sk < n; sk += BLOCKSIZE )
19             do_block(n, si, sj, sk, A, B, C);
20 }
```

FIGURE 5.21 Cache blocked version of DGEMM in Figure 3.22.

Assume `C` is initialized to zero. The `do_block` function is basically DGEMM from Chapter 3 with new parameters to specify the starting positions of the submatrices of `BLOCKSIZE`. The gcc optimizer can remove the function overhead instructions by inlining the `do_block` function.

Figure 5.22 illustrates the accesses to the three arrays using blocking. Looking only at capacity misses, the total number of memory words accessed is $2\,N^3/\text{BLOCKSIZE} + N^2$. This total is an improvement by about a factor of `BLOCKSIZE`. Hence, blocking exploits a combination of spatial and temporal locality, since `A` benefits from spatial locality and `B` benefits from temporal locality.

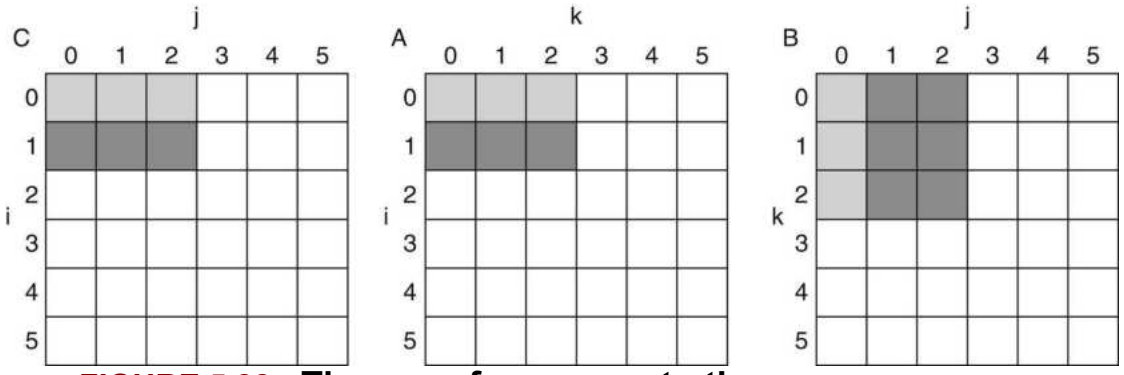


FIGURE 5.22 **The age of accesses to the arrays `C`, `A`, and `B` when *BLOCKSIZE* = 3.**
Note that, in contrast to Figure 5.20, fewer elements are accessed.

Although we have aimed at reducing cache misses, blocking can also be used to help register allocation. By taking a small blocking size, such that the block can be held in registers, we can minimize the number of loads and stores in the program, which again improves performance.

Figure 5.23 shows the impact of cache blocking on the performance of the unoptimized DGEMM as we increase the matrix size beyond where all three matrices fit in the cache. The unoptimized performance is halved for the largest matrix. The cache-blocked version is less than 10% slower even at matrices that are 960 ×960, or 900 times larger than the 32 ×32 matrices in Chapters 3 and 4.

Elaboration

Multilevel caches create many complications. First, there are now several different types of misses and corresponding miss rates. In the example on pages 402–403, we saw the primary cache miss rate and the **global miss rate**—the fraction of references that missed in all cache levels. There is also a miss rate for the secondary cache, which is the ratio of all misses in the secondary cache divided by the number of accesses to it. This miss rate is called the **local miss rate** of the secondary cache. Because the primary cache filters accesses, especially those with good spatial and temporal locality, the local miss rate of the secondary cache is much higher than the global miss rate. For the example on pages 402–403, we can compute the local miss rate of the secondary cache as

0.5%/2%=25%! Luckily, the global miss rate dictates how often we must access the main memory.

global miss rate

The fraction of references that miss in all levels of a multilevel cache.

local miss rate

The fraction of references to one level of a cache that miss; used in multilevel hierarchies.

Elaboration

With out-of-order processors (see Chapter 4), performance is more complex, since they execute instructions during the miss penalty. Instead of instruction miss rates and data miss rates, we use misses per instruction, and this formula:

$$\frac{\text{Memory} - \text{stall cycles}}{\text{Instruction}} = \frac{\text{Misses}}{\text{Instruction}} \times (\text{Total miss latency} - \text{Overlapped miss latency})$$

There is no general way to calculate overlapped miss latency, so evaluations of memory hierarchies for out-of-order processors inevitably require simulation of the processor and the memory hierarchy. Only by seeing the execution of the processor during each miss can we see if the processor stalls waiting for data or simply finds other work to do. A guideline is that the processor often hides the miss penalty for an L1 cache miss that hits in the L2 cache, but it rarely hides a miss to the L2 cache.

Elaboration

The performance challenge for algorithms is that the memory hierarchy varies between different implementations of the same architecture in cache size, associativity, block size, and number of caches. To cope with such variability, some recent numerical libraries parameterize their algorithms and then search the parameter space at runtime to find the best combination for a

particular computer. This approach is called *autotuning*.

Check Yourself

Which of the following is generally true about a design with multiple levels of caches?

1. First-level caches are more concerned about hit time, and second-level caches are more concerned about miss rate.
2. First-level caches are more concerned about miss rate, and second-level caches are more concerned about hit time.

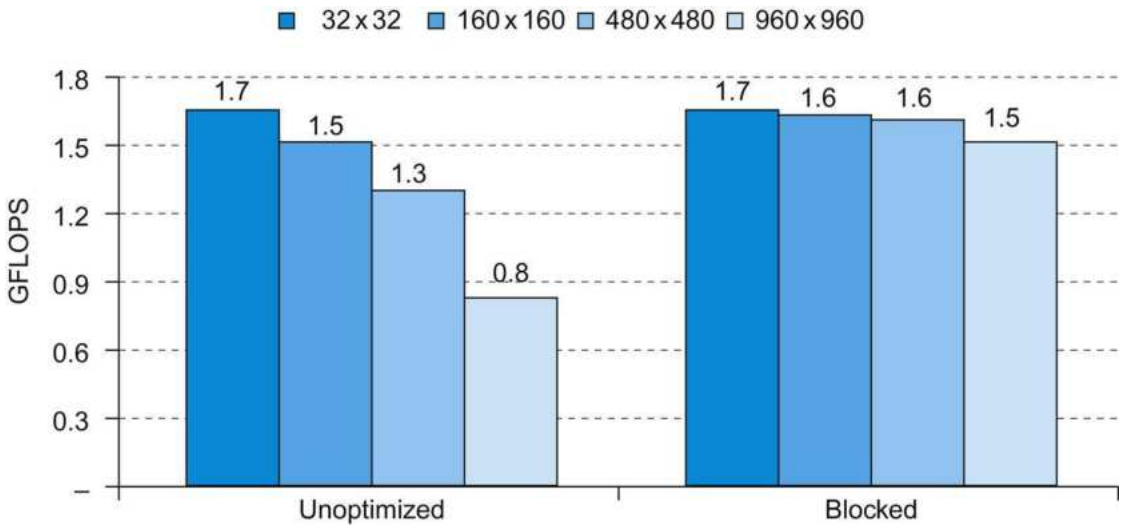


FIGURE 5.23 Performance of unoptimized DGEMM (Figure 3.22) versus cache blocked DGEMM (Figure 5.21) as the matrix dimension varies from 32 ×32 (where all three matrices fit in the cache) to 960 ×960.

Summary

In this section, we focused on four topics: cache performance, using associativity to reduce miss rates, the use of multilevel cache hierarchies to reduce miss penalties, and software optimizations to improve effectiveness of caches.

The memory system has a significant effect on program execution time. The number of memory-stall cycles depends on both the miss rate and the miss penalty. The challenge, as we will see in Section 5.8, is to reduce one of these factors without significantly affecting

other critical factors in the memory hierarchy.

To reduce the miss rate, we examined the use of associative placement schemes. Such schemes can reduce the miss rate of a cache by allowing more flexible placement of blocks within the cache. Fully associative schemes allow blocks to be placed anywhere, but also require that every block in the cache be searched to satisfy a request. The higher costs make large fully associative caches impractical. Set-associative caches are a practical alternative, since we need only search among the elements of a unique set that is chosen by indexing. Set-associative caches have higher miss rates but are faster to access. The amount of associativity that yields the best performance depends on both the technology and the details of the implementation.

We looked at multilevel caches as a technique to reduce the miss penalty by allowing a larger secondary cache to handle misses to the primary cache. Second-level caches have become commonplace as designers find that limited silicon and the goals of high clock rates prevent primary caches from becoming large. The secondary cache, which is often 10 or more times larger than the primary cache, handles many accesses that miss in the primary cache. In such cases, the miss penalty is that of the access time to the secondary cache (typically <10 processor cycles) versus the access time to memory (typically >100 processor cycles). As with associativity, the design tradeoffs between the size of the secondary cache and its access time depend on a number of aspects of the implementation.

Finally, given the importance of the memory hierarchy in performance, we looked at how to change algorithms to improve cache behavior, with blocking being an important technique when dealing with large arrays.

5.5 Dependable Memory Hierarchy

Implicit in all the prior discussion is that the memory hierarchy doesn't forget. Fast but undependable is not very attractive. As we learned in Chapter 1, the one great idea for **dependability** is redundancy. In this section we'll first go over the terms to define terms and measures associated with failure, and then show how redundancy can make nearly unforgettable memories.

DEPENDABILITY

Defining Failure

We start with an assumption that you have a specification of proper service. Users can then see a system alternating between two states of delivered service with respect to the service specification:
1. *Service accomplishment*, where the service is delivered as specified
2. *Service interruption*, where the delivered service is different from the specified service

Transitions from state 1 to state 2 are caused by *failures*, and transitions from state 2 to state 1 are called *restorations*. Failures can be permanent or intermittent. The latter is the more difficult case; it is harder to diagnose the problem when a system oscillates between the two states. Permanent failures are far easier to diagnose.

This definition leads to two related terms: reliability and availability.

Reliability is a measure of the continuous service accomplishment —or, equivalently, of the time to failure—from a reference point. Hence, *mean time to failure* (MTTF) is a reliability measure. A related term is *annual failure rate* (AFR), which is just the percentage of devices that would be expected to fail in a year for a given MTTF. When MTTF gets large it can be misleading, while AFR leads to better intuition.

MTTF vs. AFR of Disks

Example

Some disks today are quoted to have a 1,000,000-hour MTTF. As 1,000,000 hours is 1,000,000/(365 ×24)=114 years, it would seem like they practically never fail. Warehouse-scale computers that run Internet services such as Search might have 50,000 servers. Assume

each server has two disks. Use AFR to calculate how many disks we would expect to fail per year.

Answer

One year is 365 ×24 =8760 hours. A 1,000,000-hour MTTF means an AFR of 8760/1,000,000 =0.876%. With 100,000 disks, we would expect 876 disks to fail per year, or on average more than two disk failures per day!

Service interruption is measured as *mean time to repair* (MTTR). *Mean time between failures* (MTBF) is simply the sum of MTTF +MTTR. Although MTBF is widely used, MTTF is often the more appropriate term. *Availability* is then a measure of service accomplishment with respect to the alternation between the two states of accomplishment and interruption. Availability is statistically quantified as

$$\text{Availability} = \frac{\text{MTTF}}{(\text{MTTF} + \text{MTTR})}$$

Note that reliability and availability are actually quantifiable measures, rather than just synonyms for dependability. Shrinking MTTR can help availability as much as increasing MTTF. For example, tools for fault detection, diagnosis, and repair can help reduce the time to repair faults and thereby improve availability.

We want availability to be very high. One shorthand is to quote the number of "nines of availability" per year. For instance, a very good Internet service today offers 4 or 5 nines of availability. Given 365 days per year, which is 365 ×24 ×60 =526,000 minutes, then the shorthand is decoded as follows:

One nine: 90% =>36.5 days of repair/year
Two nines: 99% =>3.65 days of repair/year
Three nines: 99.9% =>526 minutes of repair/year
Four nines: 99.99% =>52.6 minutes of repair/year
Five nines: 99.999% =>5.26 minutes of repair/year

and so on.

To increase MTTF, you can improve the quality of the components or design systems to continue operation in the

presence of components that have failed. Hence, failure needs to be defined with respect to a context, as failure of a component may not lead to a failure of the system. To make this distinction clear, the term *fault* is used to mean failure of a component. Here are three ways to improve MTTF:
1. *Fault avoidance:* Preventing fault occurrence by construction.
2. *Fault tolerance:* Using redundancy to allow the service to comply with the service specification despite faults occurring.
3. *Fault forecasting:* **Predicting** the presence and creation of faults, allowing the component to be replaced *before* it fails.

The Hamming Single Error Correcting, Double Error Detecting Code (SEC/DED)

Richard Hamming invented a popular redundancy scheme for memory, for which he received the Turing Award in 1968. To invent redundant codes, it is helpful to talk about how "close" correct bit patterns can be. What we call the *Hamming distance* is just

the minimum number of bits that are different between any two correct bit patterns. For example, the distance between 011011 and 001111 is two. What happens if the minimum distance between members of a code is two, and we get a one-bit error? It will turn a valid pattern in a code to an invalid one. Thus, if we can detect whether members of a code are accurate or not, we can detect single bit errors, and can say we have a single bit **error detection code**.

error detection code
A code that enables the detection of an error in data, but not the precise location and, hence, correction of the error.

Hamming used a *parity code* for error detection. In a parity code, the number of 1 s in a word is counted; the word has odd parity if the number of 1 s is odd and even otherwise. When a word is written into memory, the parity bit is also written (1 for odd, 0 for even). That is, the parity of the N+1 bit word should always be even. Then, when the word is read out, the parity bit is read and checked. If the parity of the memory word and the stored parity bit do not match, an error has occurred.

Example

Calculate the parity of a byte with the value 31_{ten} and show the pattern stored to memory. Assume the parity bit is on the right. Suppose the most significant bit was inverted in memory, and then you read it back. Did you detect the error? What happens if the two most significant bits are inverted?

Answer

31_{ten} is 00011111_{two}, which has five 1 s. To make parity even, we need to write a 1 in the parity bit, or 000111111_{two}. If the most significant bit is inverted when we read it back, we would see 100111111_{two} which has seven 1 s. Since we expect even parity and calculated odd parity, we would signal an error. If the *two* most significant bits are inverted, we would see 110111111_{two} which has eight 1 s or even parity, and we would *not* signal an error.

If there are 2 bits of error, then a 1-bit parity scheme will not detect any errors, since the parity will match the data with two errors. (Actually, a 1-bit parity scheme can detect any odd number of errors; however, the probability of having three errors is much lower than the probability of having two, so, in practice, a 1-bit parity code is limited to detecting a single bit of error.)

Of course, a parity code cannot correct errors, which Hamming wanted to do as well as detect them. If we used a code that had a minimum distance of 3, then any single bit error would be closer to the correct pattern than to any other valid pattern. He came up with an easy to understand mapping of data into a distance 3 code that we call *Hamming Error Correction Code* (ECC) in his honor. We use extra parity bits to allow the position identification of a single error. Here are the steps to calculate Hamming ECC

1. Start numbering bits from 1 on the left, contrary to the traditional numbering of the rightmost bit being 0.
2. Mark all bit positions that are powers of 2 as parity bits (positions 1, 2, 4, 8, 16, …).
3. All other bit positions are used for data bits (positions 3, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15, …).
4. The position of parity bit determines sequence of data bits that it checks (Figure 5.24 shows this coverage graphically) is:

- Bit 1 (0001_{two}) checks bits (1,3,5,7,9,11,...), which are bits where rightmost bit of address is 1 (0001_{two}, 0011_{two}, 0101_{two}, 0111_{two}, 1001_{two}, 1011_{two},…).
- Bit 2 (0010_{two}) checks bits (2,3,6,7,10,11,14,15,…), which are the bits where the second bit to the right in the address is 1.
- Bit 4 (0100_{two}) checks bits (4–7, 12–15, 20–23,…), which are the bits where the third bit to the right in the address is 1.
- Bit 8 (1000_{two}) checks bits (8–15, 24–31, 40–47,...), which are the bits where the fourth bit to the right in the address is 1.

Note that each data bit is covered by two or more parity bits.

5. Set parity bits to create even parity for each group.

Bit position		1	2	3	4	5	6	7	8	9	10	11	12
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8
Parity bit coverage	p1	X		X		X		X		X		X	
	p2		X	X			X	X			X	X	
	p4				X	X	X	X					X
	p8								X	X	X	X	X

FIGURE 5.24 Parity bits, data bits, and field coverage in a Hamming ECC code for eight data bits.

In what seems like a magic trick, you can determine whether bits are incorrect by looking at the parity bits. Using the 12 bit code in Figure 5.24, if the value of the four parity calculations (p8,p4,p2,p1) was 0000, then there was no error. However, if the pattern was, say, 1010, which is 10_{ten}, then Hamming ECC tells us that bit 10 (d6) is an error. Since the number is binary, we can correct the error just by inverting the value of bit 10.

Example

Assume one byte data value is 10011010_{two}. First show the Hamming ECC code for that byte, and then invert bit 10 and show that the ECC code finds and corrects the single bit error.

Leaving spaces for the parity bits, the 12 bit pattern is__ 1_ 0 0 1_ 1 0 1 0.

Answer

Position 1 checks bits 1,3,5,7,9, and 11, which we highlight:__ **1**_ **0** 0 **1**_ **1** 0 **1** 0. To make the group even parity, we should set bit 1 to 0.

Position 2 checks bits 2,3,6,7,10,11, which is 0_ **1**_ 0 **0** **1**_ 1 **0** **1** 0 or odd parity, so we set position 2 to a 1.

Position 4 checks bits 4,5,6,7,12, which is 0 1 1_ **0** **0** **1**_ 1 0 1, so we set it to a 1.

Position 8 checks bits 8,9,10,11,12, which is 0 1 1 1 0 0 1_ **1** **0** **1** **0**, so we set it to a 0.

The final code word is 011100101010. Inverting bit 10 changes it to 011100101**1**10.

Parity bit 1 is 0 (**0**1**1**1**0**0**1**0**1**1**1**0 is four 1 s, so even parity; this group is OK).

Parity bit 2 is 1 (0**1**1**1**0**0**1**0**1**1**1**0 is five 1 s, so odd parity; there is an error somewhere).

Parity bit 4 is 1 (011**1001**01110 is two 1 s, so even parity; this group is OK).

Parity bit 8 is 1 (01110010**1110** is three 1 s, so odd parity; there is an error somewhere).

Parity bits 2 and 8 are incorrect. As 2 +8 =10, bit 10 must be wrong. Hence, we can correct the error by inverting bit 10: 01110010**0**10. Voila!

Hamming did not stop at single bit error correction code. At the cost of one more bit, we can make the minimum Hamming distance in a code be 4. This means we can correct single bit errors *and detect double bit errors*. The idea is to add a parity bit that is calculated over the whole word. Let's use a 4-bit data word as an example, which would only need 7 bits for single bit error detection. Hamming parity bits H (p1 p2 p3) are computed (even parity as usual) plus the even parity over the entire word, p4:

1 2 3 4 5 6 7 **8**

p_1 p_2 d_1 p_3 d_2 d_3 d_4 $\mathbf{p_4}$

Then the algorithm to correct one error and detect two is just to calculate parity over the ECC groups (H) as before plus one more over the whole group (p_4). There are four cases:

1. H is even and p_4 is even, so no error occurred.
2. H is odd and p_4 is odd, so a correctable single error occurred. (p_4 should calculate odd parity if one error occurred.)
3. H is even and p_4 is odd, a single error occurred in p_4 bit, not in the rest of the word, so correct the p_4 bit.
4. H is odd and p_4 is even, a double error occurred. (p_4 should calculate even parity if two errors occurred.)

Single Error Correcting/Double Error Detecting (SEC/DED) is common in memory for servers today. Conveniently, 8-byte data blocks can get SEC/DED with just one more byte, which is why many DIMMs are 72 bits wide.

Elaboration

To calculate how many bits are needed for SEC, let p be total number of parity bits and d number of data bits in $p+d$ bit word. If p error correction bits are to point to error bit ($p+d$ cases) plus one case to indicate that no error exists, we need:

$$2^p \geq p + d + 1 \text{ bits, and thus } p \geq \log(p + d + 1).$$

For example, for 8 bits data means $d = 8$ and $2^p \geq p + 8 + 1$, so $p = 4$. Similarly, $p = 5$ for 16 bits of data, 6 for 32 bits, 7 for 64 bits, and so on.

Elaboration

In very large systems, the possibility of multiple errors as well as complete failure of a single wide memory chip becomes significant. IBM introduced *chipkill* to solve this problem, and many big systems use this technology. (Intel calls their version SDDC.)

Similar in nature to the RAID approach used for disks (see Section 5.11), Chipkill distributes the data and ECC information, so that the complete failure of a single memory chip can be handled by supporting the reconstruction of the missing data from the remaining memory chips. Assuming a 10,000-processor cluster with 4 GiB per processor, IBM calculated the following rates of *unrecoverable* memory errors in 3 years of operation:

- Parity only—about 90,000, or one unrecoverable (or undetected) failure every 17 minutes.
- SEC/DED only—about 3500, or about one undetected or unrecoverable failure every 7.5 hours.
- Chipkill—6, or about one undetected or unrecoverable failure every 2 months.

Hence, Chipkill is a requirement for warehouse-scale computers.

Elaboration

While single or double bit errors are typical for memory systems, networks can have bursts of bit errors. One solution is called *Cyclic Redundancy Check*. For a block of k bits, a transmitter generates an

n-k bit frame check sequence. It transmits *n* bits exactly divisible by some number. The receiver divides the frame by that number. If there is no remainder, it assumes there is no error. If there is, the receiver rejects the message, and asks the transmitter to send again. As you might guess from Chapter 3, it is easy to calculate division for some binary numbers with a shift register, which made CRC codes popular even when hardware was more precious. Going even further, Reed-Solomon codes use Galois fields to *correct* multibit transmission errors, but now data are considered coefficients of a polynomial and the code space is values of a polynomial. The Reed-Solomon calculation is considerably more complicated than binary division!

5.6 Virtual Machines

Virtual machines (VM) were first developed in the mid-1960s, and they have remained an important part of mainframe computing over the years. Although largely ignored in the single-user PC era in the 1980s and 1990s, they have recently gained popularity due to

- The increasing importance of isolation and security in modern systems
- The failures in security and reliability of standard operating systems
- The sharing of a single computer among many unrelated users, in particular for Cloud computing
- The dramatic increases in raw speed of processors over the decades, which makes the overhead of VMs more acceptable

The broadest definition of VMs includes basically all emulation methods that provide a standard software interface, such as the Java VM. In this section, we are interested in VMs that provide a complete system-level environment at the binary *instruction set architecture* (ISA) level. Although some VMs run different ISAs in the VM from the native hardware, we assume they always match the hardware. Such VMs are called (Operating) *System Virtual Machines*. IBM VM/370, VirtualBox, VMware ESX Server, and Xen are examples.

System virtual machines present the illusion that the users have an entire computer to themselves, including a copy of the operating system. A single computer runs multiple VMs and can support a

number of different *operating systems* (OSes). On a conventional platform, a single OS "owns" all the hardware resources, but with a VM, multiple OSes all share the hardware resources.

The software that supports VMs is called a *virtual machine monitor* (VMM) or *hypervisor*; the VMM is the heart of virtual machine technology. The underlying hardware platform is called the *host*, and its resources are shared among the *guest* VMs. The VMM determines how to map virtual resources to physical resources: a physical resource may be time-shared, partitioned, or even emulated in software. The VMM is much smaller than a traditional OS; the isolation portion of a VMM is perhaps only 10,000 lines of code.

Although our interest here is in VMs for improving protection, VMs provide two other benefits that are commercially significant:

1. *Managing software*. VMs provide an abstraction that can run the complete software stack, even including old operating systems like DOS. A typical deployment might be some VMs running legacy OSes, many running the current stable OS release, and a few testing the next OS release.
2. *Managing hardware*. One reason for multiple servers is to have each application running with the compatible version of the operating system on separate computers, as this separation can improve dependability. VMs allow these separate software stacks to run independently yet share hardware, thereby consolidating the number of servers. Another example is that some VMMs support migration of a running VM to a different computer, either to balance load or to evacuate from failing hardware.

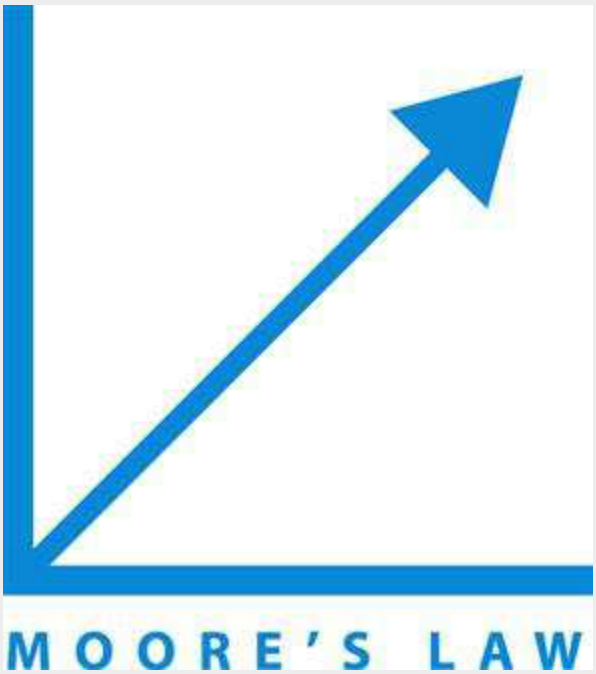
Hardware/Software Interface

Amazon Web Services (AWS) uses the virtual machines in its Cloud computing offering EC2 for five reasons:

1. It allows AWS to protect users from each other while sharing the same server.
2. It simplifies software distribution within a warehouse-scale computer. A customer installs a virtual machine image configured with the appropriate software, and AWS distributes it to all the instances a customer wants to use.
3. Customers (and AWS) can reliably "kill" a VM to control

resource usage when customers complete their work.

4. Virtual machines hide the identity of the hardware on which the customer is running, which means AWS can keep using old servers *and* introduce new, more efficient servers. The customer expects performance for instances to match their ratings in "EC2 Compute Units," which AWS defines: to "provide the equivalent CPU capacity of a 1.0–1.2 GHz 2007 AMD Opteron or 2007 Intel Xeon processor." Thanks to **Moore's Law**, newer servers clearly offer more EC2 Compute Units than older ones, but AWS can keep renting old servers as long as they are economical.



5. Virtual machine monitors can control the rate that a VM uses the processor, the network, and disk space, which allows AWS to offer many price points of instances of different types running on the same underlying servers. For example, in 2012 AWS offered 14 instance types, from small standard instances at $0.08 per hour to high I/O quadruple extra-large instances at $3.10 per hour.

In general, the cost of processor virtualization depends on the workload. User-level processor-bound programs have zero virtualization overhead, because the OS is rarely invoked, so everything runs at native speeds. I/O-intensive workloads are generally also OS-intensive, executing many system calls and

privileged instructions that can result in high virtualization overhead. On the other hand, if the I/O-intensive workload is also *I/O-bound*, the cost of processor virtualization can be completely hidden, since the processor is often idle waiting for I/O.

The overhead is determined by both the number of instructions that must be emulated by the VMM and by how much time each takes to emulate. Hence, when the guest VMs run the same ISA as the host, as we assume here, the goal of the architecture and the VMM is to run almost all instructions directly on the native hardware.

Requirements of a Virtual Machine Monitor

What must a VM monitor do? It presents a software interface to guest software, it must isolate the state of guests from each other, and it must protect itself from guest software (including guest OSes). The qualitative requirements are:

- Guest software should behave on a VM exactly as if it were running on the native hardware, except for performance-related behavior or limitations of fixed resources shared by multiple VMs.
- Guest software should not be able to change the allocation of real system resources directly.

To "virtualize" the processor, the VMM must control just about everything—access to privileged state, I/O, exceptions, and interrupts—even though the guest VM and OS presently running are temporarily using them.

For example, in the case of a timer interrupt, the VMM would suspend the currently running guest VM, save its state, handle the interrupt, determine which guest VM to run next, and then load its state. Guest VMs that rely on a timer interrupt are provided with a virtual timer and an emulated timer interrupt by the VMM.

To be in charge, the VMM must be at a higher privilege level than the guest VM, which generally runs in user mode; this also ensures that the execution of any privileged instruction will be handled by the VMM. The basic system requirements to support VMMs are:

- At least two processor modes, system and user.
- A privileged subset of instructions that is available only in system mode, resulting in a trap if executed in user mode; all system

resources must be controllable just via these instructions.

(Lack of) Instruction Set Architecture Support for Virtual Machines

If VMs are planned for during the design of the ISA, it's relatively easy to reduce both the number of instructions that must be executed by a VMM and improve their emulation speed. An architecture that allows the VM to execute directly on the hardware earns the title *virtualizable*, and the IBM 370 and the RISC-V architectures proudly bear that label.

Alas, since VMs have been considered for PC and server applications only fairly recently, most instruction sets were created without virtualization in mind. These culprits include x86 and most RISC architectures, including ARMv7 and MIPS.

Because the VMM must ensure that the guest system only interacts with virtual resources, a conventional guest OS runs as a user mode program on top of the VMM. Then, if a guest OS attempts to access or modify information related to hardware resources via a privileged instruction—for example, reading or writing a status bit that enables interrupts—it will trap to the VMM. The VMM can then affect the appropriate changes to corresponding real resources.

Hence, if any instruction that tries to read or write such sensitive information traps when executed in user mode, the VMM can intercept it and support a virtual version of the sensitive information, as the guest OS expects.

In the absence of such support, other measures must be taken. A VMM must take special precautions to locate all problematic instructions and ensure that they behave correctly when executed by a guest OS, thereby increasing the complexity of the VMM and reducing the performance of running the VM.

Protection and Instruction Set Architecture

Protection is a joint effort of architecture and operating systems, but architects had to modify some awkward details of existing instruction set architectures when virtual memory became popular.

For example, the x86 instruction POPF loads the flag registers

from the top of the stack in memory. One of the flags is the *Interrupt Enable* (IE) flag. If you run the POPF instruction in user mode, rather than trap it, it simply changes all the flags except IE. In system mode, it does change the IE. Since a guest OS runs in user mode inside a VM, this is a problem, as it expects to see a changed IE.

Historically, IBM mainframe hardware and VMM took three steps to improve the performance of virtual machines:

1. Reduce the cost of processor virtualization.
2. Reduce interrupt overhead cost due to the virtualization.
3. Reduce interrupt cost by steering interrupts to the proper VM without invoking VMM.

AMD and Intel tried to address the first point in 2006 by reducing the cost of processor virtualization. It will be interesting to see how many generations of architecture and VMM modifications it will take to address all three points, and how long before virtual machines of the 21st century for x86 will be as efficient as the IBM mainframes and VMMs of the 1970s.

Elaboration

RISC-V traps all privileged instructions when running in user mode, so it supports *classical virtualization*, wherein the guest OS runs in user mode and the VMM runs in supervisor mode.

5.7 Virtual Memory

> *… a system has been devised to make the core drum combination appear to the programmer as a single level store, the requisite transfers taking place automatically.*
>
> *Kilburn et al., One-level storage system, 1962*

In earlier sections, we saw how caches provided fast access to recently-used portions of a program's code and data. Similarly, the main memory can act as a "cache" for the secondary storage, traditionally implemented with magnetic disks. This technique is called **virtual memory**. Historically, there were two major motivations for virtual memory: to allow efficient and safe sharing

of memory among several programs, such as for the memory needed by multiple virtual machines for Cloud computing, and to remove the programming burdens of a small, limited amount of main memory. Five decades after its invention, it's the former reason that reigns today.

virtual memory

A technique that uses main memory as a "cache" for secondary storage.

Of course, to allow multiple virtual machines to share the same memory, we must be able to protect the virtual machines from each other, ensuring that a program can just read and write the portions of main memory that have been assigned to it. Main memory need contain only the active portions of the many virtual machines, just as a cache contains only the active portion of one program. Thus, the principle of locality enables virtual memory as well as caches, and virtual memory allows us to share the processor efficiently as well as the main memory.

We cannot know which virtual machines will share the memory with other virtual machines when we compile them. In fact, the virtual machines sharing the memory change dynamically while they are running. Because of this dynamic interaction, we would like to compile each program into its own *address space*—a separate range of memory locations accessible only to this program. Virtual memory implements the translation of a program's address space to **physical addresses**. This translation process enforces **protection** of a program's address space from other virtual machines.

physical address

An address in main memory.

protection

A set of mechanisms for ensuring that multiple processes sharing the processor, memory, or I/O devices cannot interfere, intentionally or unintentionally, with one another by reading or writing each other's data. These mechanisms also isolate the

operating system from a user process.

The second motivation for virtual memory is to allow a single-user program to exceed the size of primary memory. Formerly, if a program became too large for memory, it was up to the programmer to make it fit. Programmers divided programs into pieces and then identified the pieces that were mutually exclusive. These *overlays* were loaded or unloaded under user program control during execution, with the programmer ensuring that the program at no time tried to access an overlay that was not loaded and that the overlays loaded never exceeded the total size of the memory. Overlays were traditionally organized as modules, each containing both code and data. Calls between procedures in different modules would lead to overlaying of one module with another.

As you can well imagine, this responsibility was a substantial burden on programmers. Virtual memory, which was invented to relieve programmers of this difficulty, automatically manages the two levels of the memory hierarchy represented by main memory (sometimes called *physical memory* to distinguish it from virtual memory) and secondary storage.

Although the concepts at work in virtual memory and in caches are the same, their differing historical roots have led to the use of different terminology. A virtual memory block is called a *page*, and a virtual memory miss is called a **page fault**. With virtual memory, the processor produces a **virtual address**, which is translated by a combination of hardware and software to a *physical address*, which in turn can be used to access main memory. Figure 5.25 shows the virtually addressed memory with pages mapped to main memory. This process is called *address mapping* or **address translation**. Today, the two memory hierarchy levels controlled by virtual memory are usually DRAMs and flash memory in personal mobile devices and DRAMs and magnetic disks in servers (see Section 5.2). If we return to our library analogy, we can think of a virtual address as the title of a book and a physical address as the location of that book in the library, such as might be given by the Library of Congress call number.

page fault

An event that occurs when an accessed page is not present in main memory.

virtual address

An address that corresponds to a location in virtual space and is translated by address mapping to a physical address when memory is accessed.

address translation

Also called **address mapping**. The process by which a virtual address is mapped to an address used to access memory.

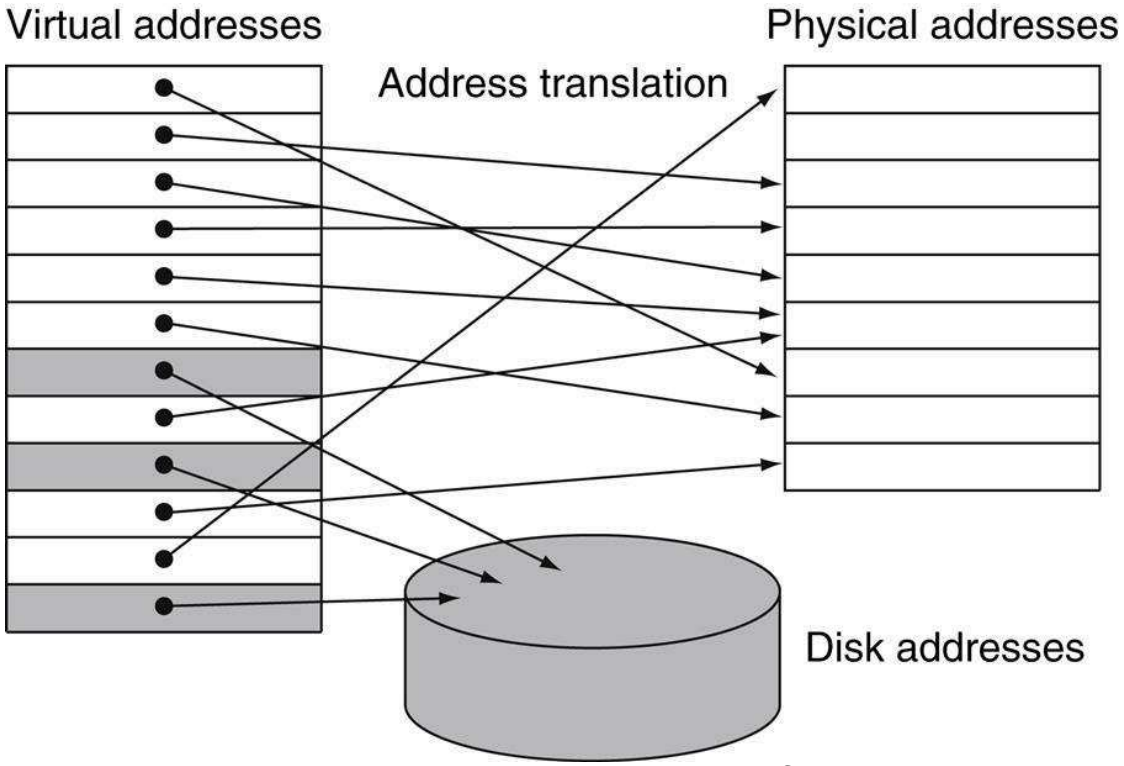


FIGURE 5.25 In virtual memory, blocks of memory (called *pages*) are mapped from one set of addresses (called *virtual addresses*) to another set (called *physical addresses*).
The processor generates virtual addresses while the memory is accessed using physical addresses. Both the virtual memory and the physical memory are broken into pages, so that a virtual page is mapped to a physical page. Of course, it is also possible for a

virtual page to be absent from main memory and not be mapped to a physical address; in that case, the page resides on disk. Physical pages can be shared by having two virtual addresses point to the same physical address. This capability is used to allow two different programs to share data or code.

Virtual memory also simplifies loading the program for execution by providing *relocation*. Relocation maps the virtual addresses used by a program to different physical addresses before the addresses are used to access memory. This relocation allows us to load the program anywhere in main memory. Furthermore, all virtual memory systems in use today relocate the program as a set of fixed-size blocks (pages), thereby eliminating the need to find a contiguous block of memory to allocate to a program; instead, the operating system needs only to find enough pages in main memory.

In virtual memory, the address is broken into a *virtual page number* and a *page offset*. Figure 5.26 shows the translation of the virtual page number to a *physical page number*. While RISC-V has a 64-bit address, the upper 16 bits are not used, so the address to be mapped is 48 bits. This figure assumes the physical memory is 1 TiB, or 2^{40} bytes, which needs a 40-bit address. The physical page number constitutes the upper portion of the physical address, while the page offset, which is not changed, constitutes the lower portion. The number of bits in the page offset field determines the page size. The number of pages addressable with the virtual address can be different than the number of pages addressable with the physical address. Having a larger number of virtual pages than physical pages is the basis for the illusion of an essentially unbounded amount of virtual memory.

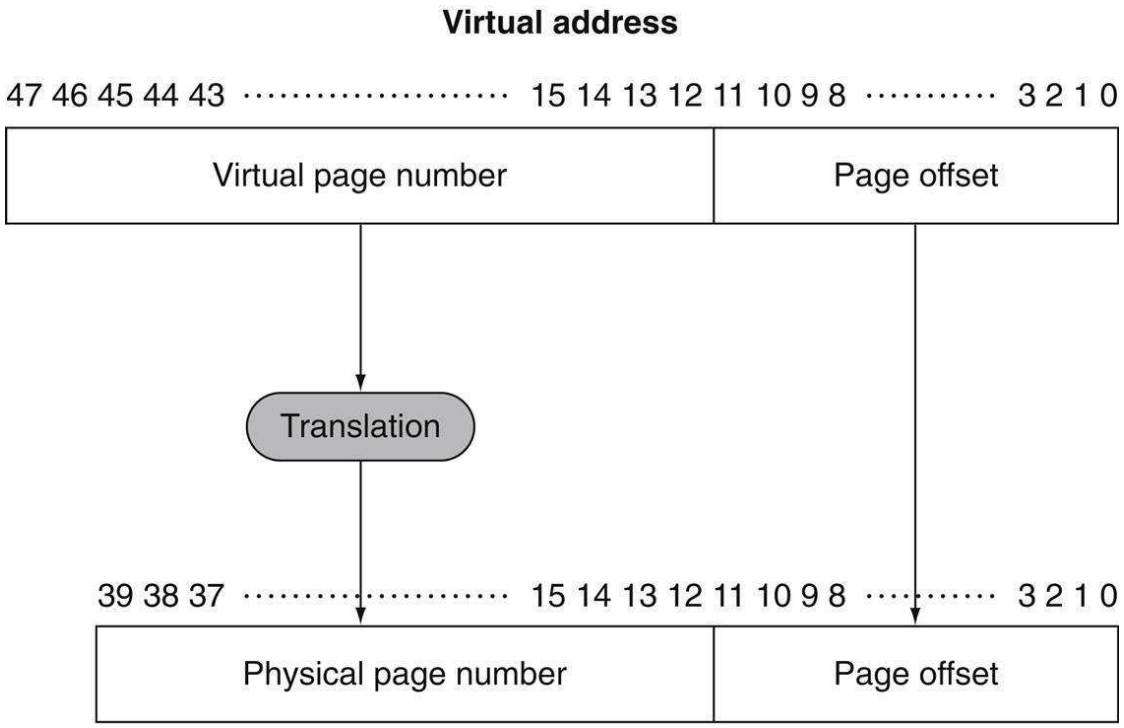


FIGURE 5.26 Mapping from a virtual to a physical address.
The page size is 2^{12} = 4 KiB. The number of physical pages allowed in memory is 2^{28}, since the physical page number has 28 bits in it. Thus, main memory can have at most 1 TiB, while the virtual address space is 256 TiB. RISC-V allows physical memory to be up to 1 PiB; we chose 1 TiB because it is ample for many computers in 2016.

Many design choices in virtual memory systems are motivated by the high cost of a page fault. A page fault to disk will take millions of clock cycles to process. (The table on page 372 shows that main memory latency is about 100,000 times quicker than disk.) This enormous miss penalty, dominated by the time to get the first word for typical page sizes, leads to several key decisions in designing virtual memory systems:

- Pages should be large enough to try to amortize the high access time. Sizes from 4 KiB to 64 KiB are typical today. New desktop and server systems are being developed to support 32 KiB and 64 KiB pages, but new embedded systems are going in the other direction, to 1 KiB pages.
- Organizations that reduce the page fault rate are attractive. The

primary technique used here is to allow fully associative placement of pages in memory.

- Page faults can be handled in software because the overhead will be small compared to the disk access time. In addition, software can afford to use clever algorithms for choosing how to place pages because even little reductions in the miss rate will pay for the cost of such algorithms.
- Write-through will not work for virtual memory, since writes take too long. Instead, virtual memory systems use write-back.

The next few subsections address these factors in virtual memory design.

Elaboration

We present the motivation for virtual memory as many virtual machines sharing the same memory, but virtual memory was originally invented so that many programs could share a computer as part of a timesharing system. Since many readers today have no experience with time-sharing systems, we use virtual machines to motivate this section.

Elaboration

RISC-V supports a variety of virtual memory configurations. In addition to the 48-bit virtual address scheme, which is a good fit for large servers in 2016, the architecture can support 39-bit and 57-bit virtual address spaces. All of these configurations use a page size of 4 Kibibytes.

Elaboration

For servers and even PCs, 32-bit address processors are problematic. Although we normally think of virtual addresses as much larger than physical addresses, the opposite can occur when the processor address size is small relative to the state of the memory technology. No single program or virtual machine can benefit, but a collection of programs or virtual machines running at the same time can benefit from not having to be swapped out of main memory or by running on parallel processors.

Elaboration

The discussion of virtual memory in this book focuses on paging, which uses fixed-size blocks. There is also a variable-size block scheme called **segmentation**. In segmentation, an address consists of two parts: a segment number and a segment offset. The segment number is mapped to a physical address, and the offset is *added* to find the actual physical address. Because the segment can vary in size, a bounds check is also needed to make sure that the offset is within the segment. The major use of segmentation is to support more powerful methods of protection and sharing in an address space. Most operating system textbooks contain extensive discussions of segmentation compared to paging and of the use of segmentation to share the address space logically. The major disadvantage of segmentation is that it splits the address space into logically separate pieces that must be manipulated as a two-part address: the segment number and the offset. Paging, in contrast, makes the boundary between page number and offset invisible to programmers and compilers.

Segments have also been used as a method to extend the address space without changing the word size of the computer. Such attempts have been unsuccessful because of the awkwardness and performance penalties inherent in a two-part address, of which programmers and compilers must be aware.

Many architectures divide the address space into large fixed-size blocks that simplify protection between the operating system and user programs and increase the efficiency of implementing paging. Although these divisions are often called "segments," this mechanism is much simpler than variable block size segmentation and is not visible to user programs; we discuss it in more detail shortly.

segmentation

A variable-size address mapping scheme in which an address consists of two parts: a segment number, which is mapped to a physical address, and a segment offset.

Placing a Page and Finding It Again

Because of the incredibly high penalty for a page fault, designers reduce page fault frequency by optimizing page placement. If we allow a virtual page to be mapped to any physical page, the operating system can then choose to replace any page it wants when a page fault occurs. For example, the operating system can use a sophisticated algorithm and complex data structures that track page usage to try to choose a page that will not be needed for a long time. The ability to use a clever and flexible replacement scheme reduces the page fault rate and simplifies the use of fully associative placement of pages.

As mentioned in Section 5.4, the difficulty in using fully associative placement is in locating an entry, since it can be anywhere in the upper level of the hierarchy. A full search is impractical. In virtual memory systems, we locate pages by using a table that indexes the main memory; this structure is called a **page table**, and it resides in main memory. A page table is indexed by the page number from the virtual address to discover the corresponding physical page number. Each program has its own page table, which maps the virtual address space of that program to main memory. In our library analogy, the page table corresponds to a mapping between book titles and library locations. Just as the card catalog may contain entries for books in another library on campus rather than the local branch library, we will see that the page table may contain entries for pages not present in memory. To indicate the location of the page table in memory, the hardware includes a register that points to the start of the page table; we call this the *page table register*. Assume for now that the page table is in a fixed and contiguous area of memory.

page table

The table containing the virtual to physical address translations in a virtual memory system. The table, which is stored in memory, is typically indexed by the virtual page number; each entry in the table contains the physical page number for that virtual page if the page is currently in memory.

Hardware/Software Interface

The page table, together with the program counter and the

registers, specifies the *state* of a virtual machine. If we want to allow another virtual machine to use the processor, we must save this state. Later, after restoring this state, the virtual machine can continue execution. We often refer to this state as a *process*. The process is considered *active* when it is in possession of the processor; otherwise, it is considered *inactive*. The operating system can make a process active by loading the process's state, including the program counter, which will initiate execution at the value of the saved program counter.

The process's address space, and hence all the data it can access in memory, is defined by its page table, which resides in memory. Rather than save the entire page table, the operating system simply loads the page table register to point to the page table of the process it wants to make active. Each process has its own page table, since different processes use the same virtual addresses. The operating system is responsible for allocating the physical memory and updating the page tables, so that the virtual address spaces of distinct processes do not collide. As we will see shortly, the use of separate page tables also provides protection of one process from another.

Figure 5.27 uses the page table register, the virtual address, and the indicated page table to show how the hardware can form a physical address. A valid bit is used in each page table entry, just as we did in a cache. If the bit is off, the page is not present in main memory and a page fault occurs. If the bit is on, the page is in memory and the entry contains the physical page number.

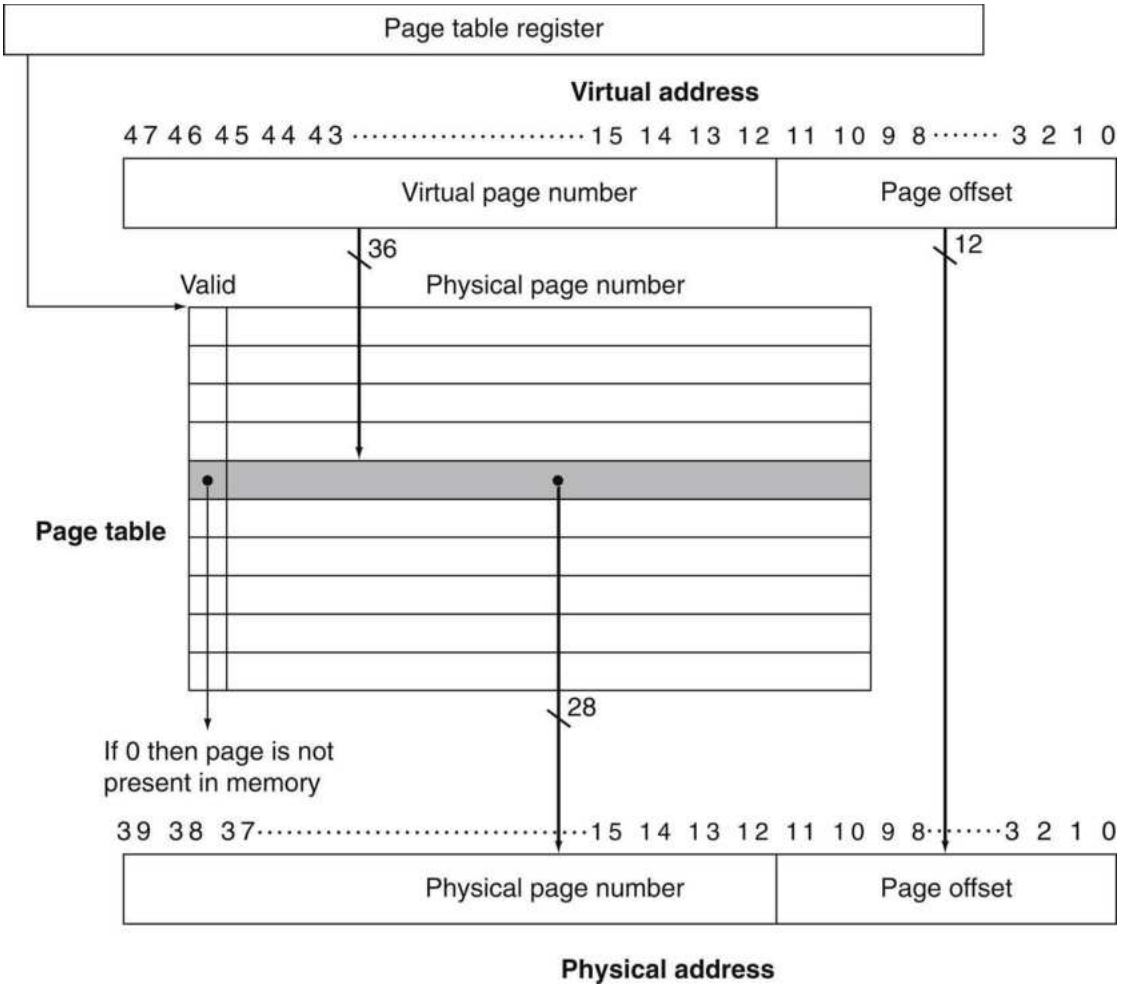


FIGURE 5.27 The page table is indexed with the virtual page number to obtain the corresponding portion of the physical address.

We assume a 48-bit address. The page table pointer gives the starting address of the page table. In this figure, the page size is 2^{12} bytes, or 4 KiB. The virtual address space is 2^{48} bytes, or 256 TiB, and the physical address space is 2^{40} bytes, which allows main memory of up to 1 TiB. If RISC-V used a single page table as shown in this figure, the number of entries in the page table would be 2^{36}, or about 64 billion entries. (We'll see what RISC-V does to reduce the number of entries shortly.) The valid bit for each entry indicates whether the mapping is legal. If it is off, then the page is not present in memory. Although the page table entry shown here need only be 29 bits wide, it would typically be rounded up to a power of 2 bits for ease of indexing. The page table entries in RISC-V are 64 bits. The extra bits would be used to store additional information that needs to be kept on a per-page basis,

such as protection.

Because the page table contains a mapping for every possible virtual page, no tags are required. In cache terminology, the index that is used to access the page table consists of the full block address, which in this case is the virtual page number.

Page Faults

If the valid bit for a virtual page is off, a page fault occurs. The operating system must be given control. This transfer is done with the exception mechanism, which we saw in Chapter 4 and will discuss again later in this section. Once the operating system gets control, it must find the page in the next level of the hierarchy (usually flash memory or magnetic disk) and decide where to place the requested page in the main memory.

The virtual address alone does not immediately tell us where the page is in secondary memory. Returning to our library analogy, we cannot find the location of a library book on the shelves just by knowing its title. Instead, we go to the catalog and look up the book, obtaining an address for the location on the shelves, such as the Library of Congress call number. Likewise, in a virtual memory system, we must keep track of the location in secondary memory of each page in virtual address space.

Because we do not know ahead of time when a page in memory will be replaced, the operating system usually creates the space on flash memory or disk for all the pages of a process when it creates the process. This space is called the **swap space**. At that time, it also creates a data structure to record where each virtual page is stored on disk. This data structure may be part of the page table or may be an auxiliary data structure indexed in the same way as the page table. Figure 5.28 shows the organization when a single table holds either the physical page number or the secondary memory address.

swap space

The space on the disk reserved for the full virtual memory space of a process.

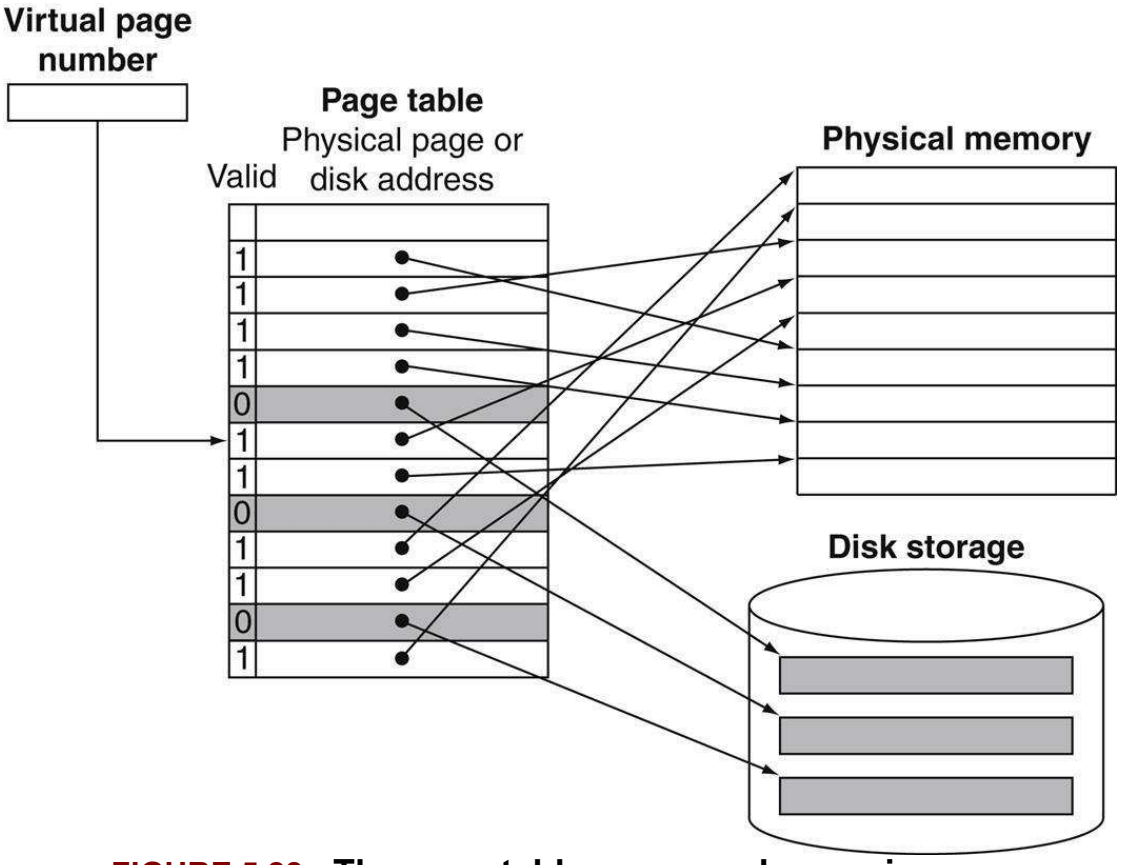


FIGURE 5.28 The page table maps each page in virtual memory to either a page in main memory or a page stored on disk, which is the next level in the hierarchy.

The virtual page number is used to index the page table. If the valid bit is on, the page table supplies the physical page number (i.e., the starting address of the page in memory) corresponding to the virtual page. If the valid bit is off, the page currently resides only on disk, at a specified disk address. In many systems, the table of physical page addresses and disk page addresses, while logically one table, is stored in two separate data structures. Dual tables are justified in part because we must keep the disk addresses of all the pages, even if they are currently in main memory. Remember that the pages in main memory and the pages on disk are the same size.

The operating system also creates a data structure that tracks which processes and which virtual addresses use each physical page. When a page fault occurs, if all the pages in main memory are in use, the operating system must choose a page to replace. Because we want to minimize the number of page faults, most operating

systems try to choose a page that they hypothesize will not be needed soon. Using the past to predict the future, operating systems follow the *least recently used* (LRU) replacement scheme, which we mentioned in Section 5.4. The operating system searches for the least recently used page, assuming that a page that has not been used in a long time is less likely to be needed than a more recently accessed page. The replaced pages are written to swap space in secondary memory. In case you are wondering, the operating system is just another process, and these tables controlling memory are in memory; the details of this seeming contradiction will be explained shortly.

Hardware/Software Interface

Implementing a completely accurate LRU scheme is too expensive, since it requires updating a data structure on *every* memory reference. Thus, most operating systems approximate LRU by keeping track of which pages have and which pages have not been recently used. To help the operating system estimate the LRU pages, RISC-V computers provide a **reference bit**, sometimes called a **use bit** or **access bit**, which is set whenever a page is accessed. The operating system periodically clears the reference bits and later records them so it can determine which pages were touched during a particular time period. With this usage information, the operating system can select a page that is among the least recently referenced (detected by having its reference bit off). If this bit is not provided by the hardware, the operating system must find another way to estimate which pages have been accessed.

reference bit

Also called **use bit** or **access bit**. A field that is set whenever a page is accessed and that is used to implement LRU or other replacement schemes.

Virtual Memory for Large Virtual Addresses

The caption in Figure 5.27 points out that with a single level page table for a 48-bit address with 4 KiB pages, we need 64 billion table

entries. As each page table entry is 8 bytes for RISC-V, it would require 0.5 TiB just to map the virtual addresses to physical addresses! Moreover, there could be hundreds of processes running, each with its own page table. That much memory for translation would be unaffordable even for the largest systems.

A range of techniques is used to reduce the amount of storage required for the page table. The five techniques below aim at reducing the total maximum storage required as well as minimizing the main memory dedicated to page tables:

1. The simplest technique is to keep a limit register that restricts the size of the page table for a given process. If the virtual page number becomes larger than the contents of the limit register, entries must be added to the page table. This technique allows the page table to grow as a process consumes more space. Thus, the page table will only be large if the process is using many pages of virtual address space. This technique requires that the address space expand in just one direction.
2. Allowing growth in only one direction is not sufficient, since most languages require two areas whose size is expandable: one area holds the stack, and the other area holds the heap. Because of this duality, it is convenient to divide the page table and let it grow from the highest address down, as well as from the lowest address up. This means that there will be two separate page tables and two separate limits. The use of two page tables breaks the address space into two segments. The high-order bit of an address usually determines which segment and thus which page table to use for that address. Since the high-order address bit specifies the segment, each segment can be as large as one-half of the address space. A limit register for each segment specifies the current size of the segment, which grows in units of pages. Unlike the type of segmentation discussed in the second elaboration on page 423, this form of segmentation is invisible to the application program, although not to the operating system. The major disadvantage of this scheme is that it does not work well when the address space is used in a sparse fashion rather than as a contiguous set of virtual addresses.
3. Another approach to reducing the page table size is to apply a hashing function to the virtual address so that the page table need be only the size of the number of *physical* pages in main memory.

Such a structure is called an *inverted page table*. Of course, the lookup process is slightly more complex with an inverted page table, because we can no longer just index the page table.
4. To reduce the actual main memory tied up in page tables, most modern systems also allow the page tables to be paged. Although this sounds tricky, it works by using the same basic ideas of virtual memory and simply allowing the page tables to reside in the virtual address space. In addition, there are some small but critical problems, such as a never-ending series of page faults, which must be avoided. How these problems are overcome is both very detailed and typically highly processor-specific. In brief, these problems are avoided by placing all the page tables in the address space of the operating system and placing at least some of the page tables for the operating system in a portion of main memory that is physically addressed and is always present and thus never in secondary memory.
5. Multiple levels of page tables can also be used to reduce the total amount of page table storage, and this is the solution that RISC-V uses to reduce the memory footprint of address translation. Figure 5.29 above shows the four levels of address translation to go from a 48-bit virtual address to a 40-bit physical address of a 4 KiB page. Address translation happens by first looking in the level 0 table, using the highest-order bits of the address. If the address in this table is valid, the next set of high-order bits is used to index the page table indicated by the segment table entry, and so on. Thus, the level 0 table maps the virtual address to a 512 GB (2^{39} bytes) region. The level 1 table in turn maps the virtual address to a 1 GB (2^{30}) region. The next level maps this down to a 2 MB (2^{21}) region. The final table maps the virtual address to the 4 KiB (2^{12}) memory page. This scheme allows the address space to be used in a sparse fashion (multiple noncontiguous segments can be active) without having to allocate the entire page table. Such schemes are particularly useful with very large address spaces and in software systems that require noncontiguous allocation. The primary disadvantage of this multi-level mapping is the more complex process for address translation.

What about Writes?

The difference between the access time to the cache and main memory is tens to hundreds of cycles, and write-through schemes can be used, although we need a write buffer to hide the latency of the write from the processor. In a virtual memory system, writes to the next level of the hierarchy (disk) can take millions of processor clock cycles; therefore, building a write buffer to allow the system to write-through to disk would be completely impractical. Instead, virtual memory systems must use write-back, performing the individual writes into the page in memory, and copying the page back to secondary memory when it is replaced in the main memory.

Hardware/Software Interface

A write-back scheme has another major advantage in a virtual memory system. Because the disk transfer time is small compared with its access time, copying back an entire page is much more efficient than writing individual words back to the disk. A write-back operation, although faster than transferring separate words, is still costly. Thus, we would like to know whether a page *needs* to be copied back when we choose to replace it. To track whether a page has been written since it was read into the memory, a *dirty bit* is added to the page table. The dirty bit is set when any word in a page is written. If the operating system chooses to replace the page, the dirty bit indicates whether the page needs to be written out before its location in memory can be given to another page. Hence, a modified page is often called a *dirty* page.

Making Address Translation Fast: the TLB

Since the page tables are stored in main memory, every memory access by a program can take at least twice as long: one memory access to obtain the physical address and a second access to get the data. The key to improving access performance is to rely on locality of reference to the page table. When a translation for a virtual page number is used, it will probably be needed again soon, because the references to the words on that page have both temporal and spatial locality.

Accordingly, modern processors include a special cache that keeps track of recently used translations. This special address translation cache is traditionally referred to as a **translation-**

lookaside buffer (TLB), although it would be more accurate to call it a translation cache. The TLB corresponds to that little piece of paper we typically use to record the location of a set of books we look up in the card catalog; rather than continually searching the entire catalog, we record the location of several books and use the scrap of paper as a cache of Library of Congress call numbers.

translation-lookaside buffer (TLB)

A cache that keeps track of recently used address mappings to try to avoid an access to the page table.

Figure 5.30 shows that each tag entry in the TLB holds a portion of the virtual page number, and each data entry of the TLB holds a physical page number. Because we access the TLB instead of the page table on every reference, the TLB will need to include other status bits, such as the dirty and the reference bits. Although Figure 5.30 shows a single page table, TLBs work fine with multi-level page tables as well. The TLB simply loads the physical address and protection tags from the last level page table.

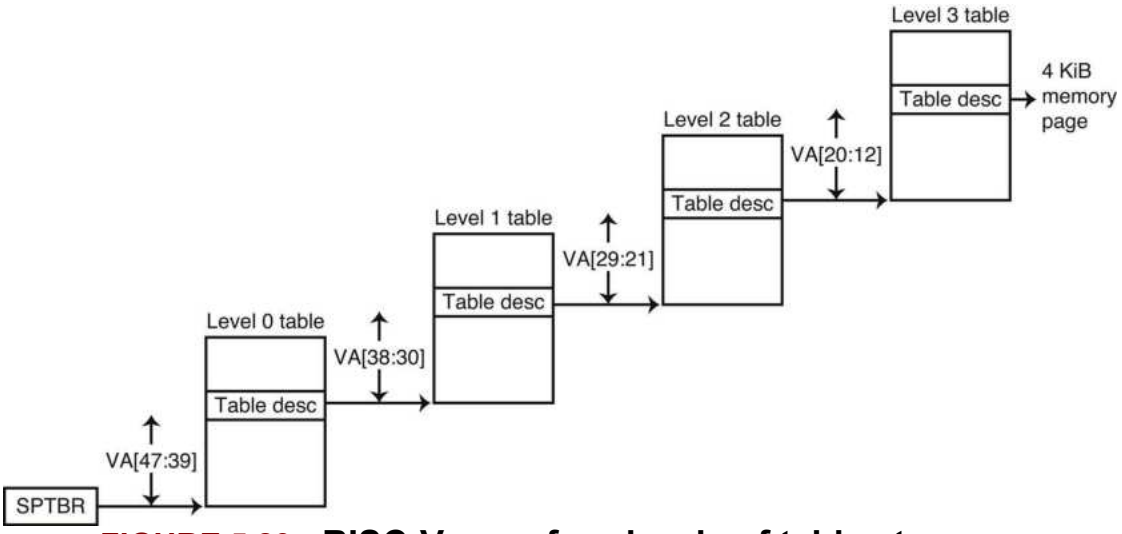


FIGURE 5.29 RISC-V uses four levels of tables to translate a 48-bit virtual address into a 40-bit physical address.
Rather than needing 64 billion page table entries for the single page table in Figure 5.27, this hierarchical approach needs just a tiny fraction. Each step of the translation uses 9 bits of the virtual address to find the

next level table, until the upper 36 bits of the virtual address are mapped to the physical address of the desired 4 KiB page. Each RISC-V page table entry is 8 bytes, so the 512 entries of a table fill a single 4 KiB page. The *Supervisor Page Table Base Register* (SPTBR) gives the starting address of the first page table.

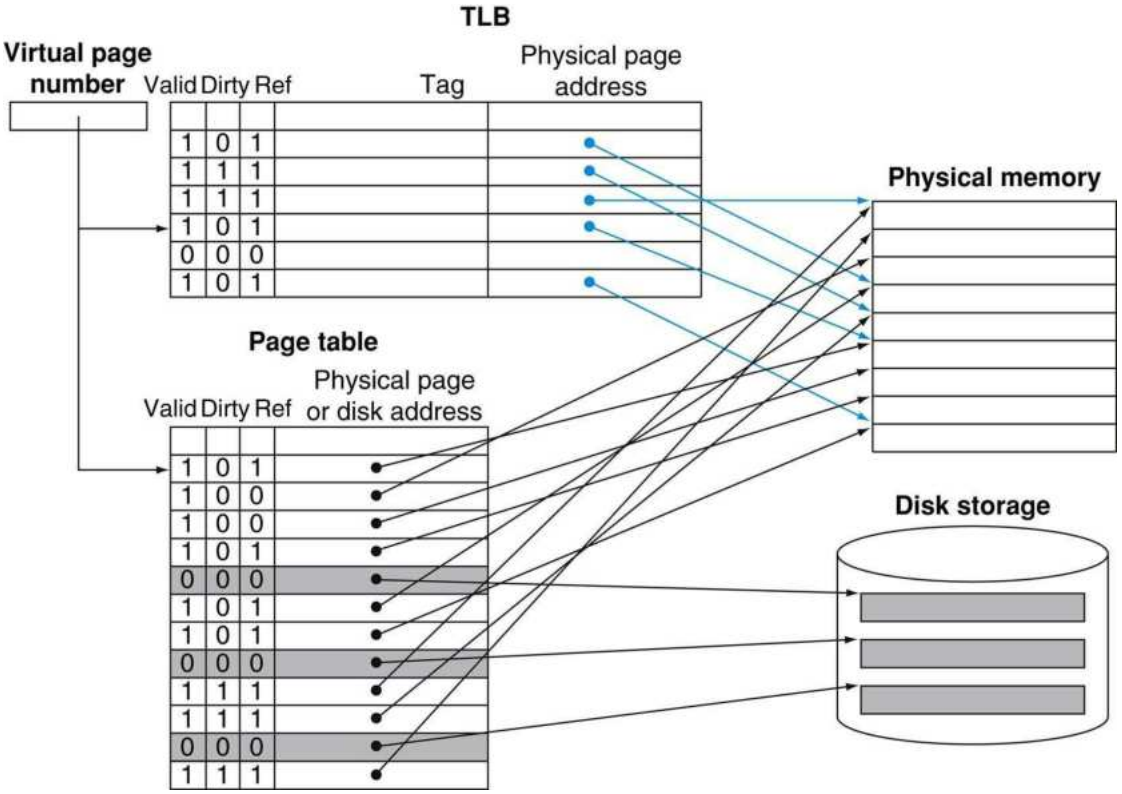


FIGURE 5.30 The TLB acts as a cache of the page table for the entries that map to physical pages only.
The TLB contains a subset of the virtual-to-physical page mappings that are in the page table. The TLB mappings are shown in color. Because the TLB is a cache, it must have a tag field. If there is no matching entry in the TLB for a page, the page table must be examined. The page table either supplies a physical page number for the page (which can then be used to build a TLB entry) or indicates that the page resides on disk, in which case a page fault occurs. Since the page table has an entry for every virtual page, no tag field is needed; in other words, unlike a TLB, a page table is *not* a cache.

On every reference, we look up the virtual page number in the TLB. If we get a hit, the physical page number is used to form the address, and the corresponding reference bit is turned on. If the processor is performing a write, the dirty bit is also turned on. If a miss in the TLB occurs, we must determine whether it is a page fault or merely a TLB miss. If the page exists in memory, then the TLB miss indicates only that the translation is missing. In such

cases, the processor can handle the TLB miss by loading the translation from the (last-level) page table into the TLB and then trying the reference again. If the page is not present in memory, then the TLB miss indicates a true page fault. In this case, the processor invokes the operating system using an exception. Because the TLB has many fewer entries than the number of pages in main memory, TLB misses will be much more frequent than true page faults.

TLB misses can be handled either in hardware or in software. In practice, with care there can be little performance difference between the two approaches, because the basic operations are the same in either case.

After a TLB miss occurs and the missing translation has been retrieved from the page table, we will need to select a TLB entry to replace. Because the reference and dirty bits are contained in the TLB entry, we need to copy these bits back to the page table entry when we replace an entry. These bits are the only portion of the TLB entry that can be changed. Using write-back—that is, copying these entries back at miss time rather than when they are written—is very efficient, since we expect the TLB miss rate to be small. Some systems use other techniques to approximate the reference and dirty bits, eliminating the need to write into the TLB except to load a new table entry on a miss.

Some typical values for a TLB might be

- TLB size: 16–512 entries
- Block size: 1–2 page table entries (typically 4–8 bytes each)
- Hit time: 0.5–1 clock cycle
- Miss penalty: 10–100 clock cycles
- Miss rate: 0.01%–1%

Designers have used a wide variety of associativities in TLBs. Some systems use small, fully associative TLBs because a fully associative mapping has a lower miss rate; furthermore, since the TLB is small, the cost of a fully associative mapping is not too high. Other systems use large TLBs, often with small associativity. With a fully associative mapping, choosing the entry to replace becomes tricky since implementing a hardware LRU scheme is too expensive. Furthermore, since TLB misses are much more frequent than page faults and thus must be handled more cheaply, we cannot afford an expensive software algorithm, as we can for page

faults. As a result, many systems provide some support for randomly choosing an entry to replace. We'll examine replacement schemes in a little more detail in Section 5.8.

The Intrinsity FastMATH TLB

To see these ideas in a real processor, let's take a closer look at the TLB of the Intrinsity FastMATH. The memory system uses 4 KiB pages and just a 32-bit address space; thus, the virtual page number is 20 bits long. The physical address is the same size as the virtual address. The TLB contains 16 entries, it is fully associative, and it is shared between the instruction and data references. Each entry is 64 bits wide and contains a 20-bit tag (which is the virtual page number for that TLB entry), the corresponding physical page number (also 20 bits), a valid bit, a dirty bit, and other bookkeeping bits. Like most MIPS systems, it uses software to handle TLB misses.

Figure 5.31 shows the TLB and one of the caches, while Figure 5.32 shows the steps in processing a read or write request. When a TLB miss occurs, the hardware saves the page number of the reference in a special register and generates an exception. The exception invokes the operating system, which handles the miss in software. To find the physical address for the missing page, a TLB miss indexes the page table using the page number of the virtual address and the page table register, which indicates the starting address of the active process page table. Using a special set of system instructions that can update the TLB, the operating system places the physical address from the page table into the TLB. A TLB miss takes about 13 clock cycles, assuming the code and the page table entry are in the instruction cache and data cache, respectively. A true page fault occurs if the page table entry does not have a valid physical address. The hardware maintains an index that indicates the recommended entry to replace; it is chosen randomly.

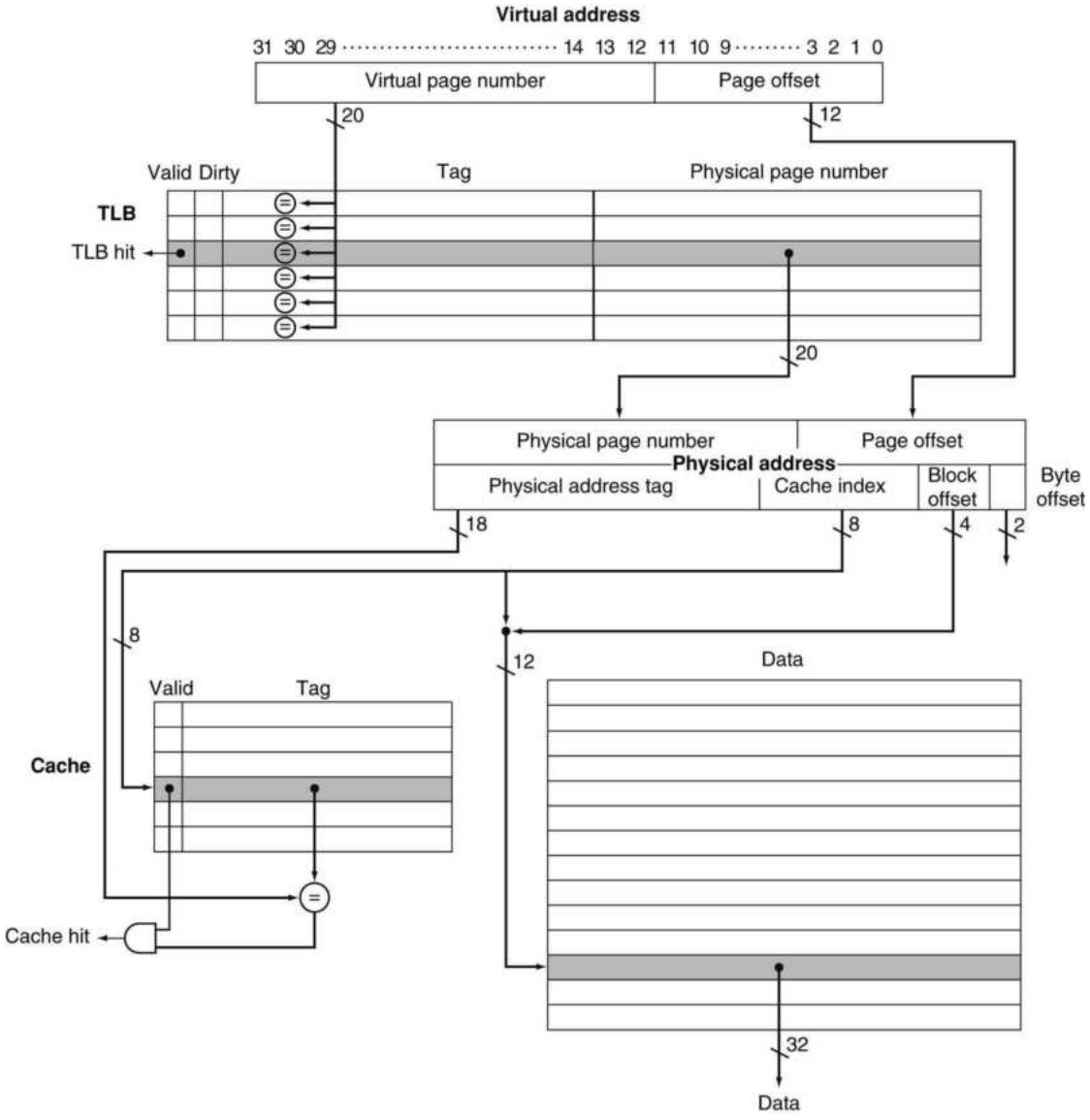


FIGURE 5.31 The TLB and cache implement the process of going from a virtual address to a data item in the Intrinsity FastMATH.
This figure shows the organization of the TLB and the data cache, assuming a 4 KiB page size. Note that the address size for this computer is just 32 bits. This diagram focuses on a read; Figure 5.32 describes how to handle writes. Note that unlike Figure 5.12, the tag and data RAMs are split. By addressing the long but narrow data RAM with the cache index concatenated with the block offset, we select the desired word in the block without a 16:1 multiplexor. While the cache is direct mapped, the TLB is fully associative. Implementing a fully associative TLB requires that every TLB tag be compared against the virtual page number, since the entry of interest can be anywhere in

the TLB. (See content addressable memories in the *Elaboration* on page 400.) If the valid bit of the matching entry is on, the access is a TLB hit, and bits from the physical page number together with bits from the page offset form the index that is used to access the cache.

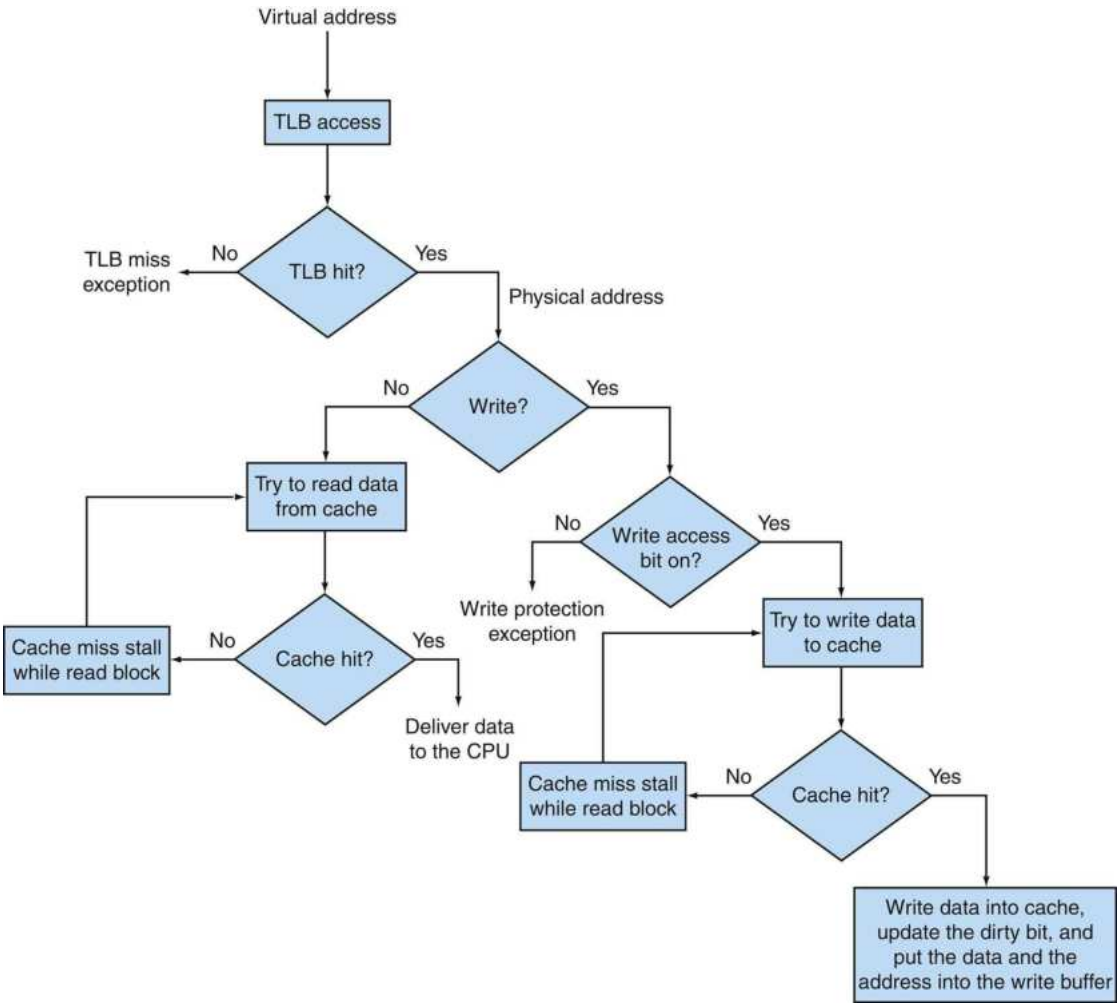


FIGURE 5.32 Processing a read or a write-through in the Intrinsity FastMATH TLB and cache. If the TLB generates a hit, the cache can be accessed with the resulting physical address. For a read, the cache generates a hit or miss and supplies the data or causes a stall while the data are brought from memory. If the operation is a write, a portion of the cache entry is overwritten for a hit and the data are sent to the write buffer if we assume write-through. A write miss is just like a read miss except that the block is modified after it is read from memory. Write-back requires writes to set a dirty bit for the cache block, and a write buffer is loaded with the whole block only on a read miss or write miss if the block to be replaced is dirty. Notice that a TLB hit and a cache hit are independent events, but a cache hit can only occur after a TLB hit occurs, which means that the data must be present in memory. The relationship between TLB misses and cache misses is examined further in the following example and the exercises at the end of this chapter. Note that

the address size for this computer is just 32 bits.

There is an extra complication for write requests: namely, the write access bit in the TLB must be checked. This bit prevents the program from writing into pages for which it has only read access. If the program attempts a write and the write access bit is off, an exception is generated. The write access bit forms part of the protection mechanism, which we will discuss shortly.

Integrating Virtual Memory, TLBs, and Caches

Our virtual memory and cache systems work together as a hierarchy, so that data cannot be in the cache unless it is present in main memory. The operating system helps maintain this hierarchy by flushing the contents of any page from the cache when it decides to migrate that page to secondary memory. At the same time, the OS modifies the page tables and TLB, so that an attempt to access any data on the migrated page will generate a page fault.

Under the best of circumstances, a virtual address is translated by the TLB and sent to the cache where the appropriate data are found, retrieved, and sent back to the processor. In the worst case, a reference can miss in all three components of the memory hierarchy: the TLB, the page table, and the cache. The following example illustrates these interactions in more detail.

Overall Operation of a Memory Hierarchy

Example

In a memory hierarchy like that of Figure 5.31, which includes a TLB and a cache organized as shown, a memory reference can encounter three different types of misses: a TLB miss, a page fault, and a cache miss. Consider all the combinations of these three events with one or more occurring (seven possibilities). For each possibility, state whether this event can actually occur and under what circumstances.

Answer

Figure 5.33 shows all combinations and whether each is possible in

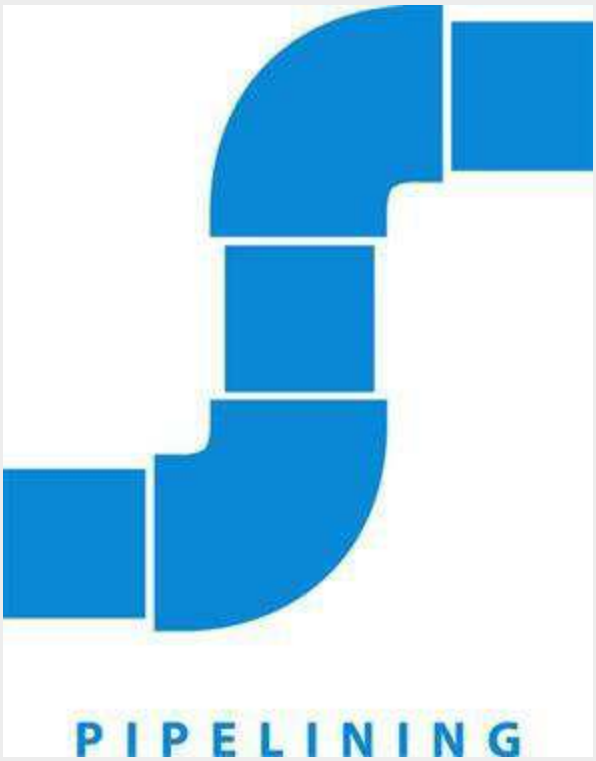
practice.

TLB	Page table	Cache	Possible? If so, under what circumstance?
Hit	Hit	Miss	Possible, although the page table is never really checked if TLB hits.
Miss	Hit	Hit	TLB misses, but entry found in page table; after retry, data is found in cache.
Miss	Hit	Miss	TLB misses, but entry found in page table; after retry, data misses in cache.
Miss	Miss	Miss	TLB misses and is followed by a page fault; after retry, data must miss in cache.
Hit	Miss	Miss	Impossible: cannot have a translation in TLB if page is not present in memory.
Hit	Miss	Hit	Impossible: cannot have a translation in TLB if page is not present in memory.
Miss	Miss	Hit	Impossible: data cannot be allowed in cache if the page is not in memory.

FIGURE 5.33 The possible combinations of events in the TLB, virtual memory system, and cache. Three of these combinations are impossible, and one is possible (TLB hit, page table hit, cache miss) but never detected.

Elaboration

Figure 5.33 assumes that all memory addresses are translated to physical addresses before the cache is accessed. In this organization, the cache is *physically indexed* and *physically tagged* (both the cache index and tag are physical, rather than virtual, addresses). In such a system, the amount of time to access memory, assuming a cache hit, must accommodate both a TLB access and a cache access; of course, these accesses can be **pipelined**.



Alternatively, the processor can index the cache with an address that is completely or partially virtual. This is called a **virtually addressed cache**, and it uses tags that are virtual addresses; hence, such a cache is *virtually indexed* and *virtually tagged*. In such caches, the address translation hardware (TLB) is unused during the normal cache access, since the cache is accessed with a virtual address that has not been translated to a physical address. This takes the TLB out of the critical path, reducing cache latency. When a cache miss occurs, however, the processor needs to translate the address to a physical address so that it can fetch the cache block from main memory.

virtually addressed cache

A cache that is accessed with a virtual address rather than a physical address.

When the cache is accessed with a virtual address and pages are shared between processes (which may access them with different virtual addresses), there is the possibility of **aliasing**. Aliasing occurs when the same object has two names—in this case, two virtual addresses for the same page. This ambiguity creates a problem, because a word on such a page may be cached in two

different locations, each corresponding to distinct virtual addresses. This ambiguity would allow one program to write the data without the other program being aware that the data had changed. Completely virtually addressed caches either introduce design limitations on the cache and TLB to reduce aliases or require the operating system, and possibly the user, to take steps to ensure that aliases do not occur.

aliasing

A situation in which two addresses access the same object; it can occur in virtual memory when there are two virtual addresses for the same physical page.

A common compromise between these two design points is caches that are virtually indexed—sometimes using just the page-offset portion of the address, which is really a physical address since it is not translated—but use physical tags. These designs, which are *virtually indexed but physically tagged*, attempt to achieve the performance advantages of virtually indexed caches with the architecturally simpler advantages of a **physically addressed cache**. For example, there is no alias problem in this case. Figure 5.31 assumed a 4 KiB page size, but it's really 16 KiB, so the Intrinsity FastMATH can use this trick. To pull it off, there must be careful coordination between the minimum page size, the cache size, and associativity. RISC-V requires caches to *behave as though* physically tagged and indexed, but it does not mandate this implementation. For example, virtually indexed, physically tagged data caches could use additional logic to ensure that software cannot tell the difference.

physically addressed cache

A cache that is addressed by a physical address.

Implementing Protection with Virtual Memory

Perhaps the most important function of virtual memory today is to allow sharing of a single main memory by multiple processes, while providing memory protection among these processes and the operating system. The protection mechanism must ensure that although multiple processes are sharing the same main memory, one renegade process cannot write into the address space of another user process or into the operating system either intentionally or unintentionally. The write access bit in the TLB can protect a page from being written. Without this level of protection,

computer viruses would be even more widespread.

Hardware/Software Interface

To enable the operating system to implement protection in the virtual memory system, the hardware must provide at least the three basic capabilities summarized below. Note that the first two are the same requirements as needed for virtual machines (Section 5.6).

1. Support at least two modes that indicate whether the running process is a user process or an operating system process, variously called a **supervisor** process, a **kernel** process, or an *executive* process.
2. Provide a portion of the processor state that a user process can read but not write. This state includes the user/supervisor mode bit, which dictates whether the processor is in user or supervisor mode, the page table pointer, and the TLB. To write these elements, the operating system uses special instructions that are only available in supervisor mode.
3. Provide mechanisms whereby the processor can go from user mode to supervisor mode and vice versa. The first direction is typically accomplished by a **system call** exception, implemented as a special instruction (`ecall` in the RISC-V instruction set) that transfers control to a dedicated location in supervisor code space. As with any other exception, the program counter from the point of the system call is saved in the *supervisor exception program counter* (SEPC), and the processor is placed in supervisor mode. To return to user mode from the exception, use the *supervisor exception return* (`sret`) instruction, which resets to user mode and jumps to the address in SEPC.

By using these mechanisms and storing the page tables in the operating system's address space, the operating system can change the page tables while preventing a user process from changing them, ensuring that a user process can access only the storage provided to it by the operating system.

supervisor mode

Also called **kernel mode**. A mode indicating that a running process is an operating system process.

system call

A special instruction that transfers control from user mode to a dedicated location in supervisor code space, invoking the exception mechanism in the process.

We also want to prevent a process from reading the data of another process. For example, we wouldn't want a student program to read the teacher's grades while they were in the processor's memory. Once we begin sharing main memory, we must provide the ability for a process to protect its data from both reading and writing by another process; otherwise, sharing the main memory will be a mixed blessing!

Remember that each process has its own virtual address space. Thus, if the operating system keeps the page tables organized so that the independent virtual pages map to disjoint physical pages, one process will not be able to access another's data. Of course, this also requires that a user process be unable to change the page table mapping. The operating system can assure safety if it prevents the user process from modifying its own page tables. However, the operating system must be able to modify the page tables. Placing the page tables in the protected address space of the operating system satisfies both requirements.

When processes want to share information in a limited way, the operating system must assist them, since accessing the information of another process requires changing the page table of the accessing process. The write access bit can be used to restrict the sharing to just read sharing, and, like the rest of the page table, this bit can be changed only by the operating system. To allow another process, say, P1, to read a page owned by process P2, P2 would ask the operating system to create a page table entry for a virtual page in P1's address space that points to the same physical page that P2 wants to share. The operating system could use the write protection bit to prevent P1 from writing the data, if that was P2's wish. Any bits that determine the access rights for a page must be included in both the page table and the TLB, because the page table is accessed only on a TLB *miss*.

Elaboration

When the operating system decides to change from running process P1 to running process P2 (called a **context switch** or *process switch*), it must ensure that P2 cannot get access to the page tables of P1 because that would compromise protection. If there is no TLB, it suffices to change the page table register to point to P2's page table (rather than to P1's); with a TLB, we must clear the TLB entries that belong to P1—both to protect the data of P1 and to force the TLB to load the entries for P2. If the process switch rate were high, this could be quite inefficient. For example, P2 might load only a few TLB entries before the operating system switched back to P1. Unfortunately, P1 would then find that all its TLB entries were gone and would have to pay TLB misses to reload them. This problem arises because the virtual addresses used by P1 and P2 are the same, and we must clear out the TLB to avoid confusing these addresses.

A common alternative is to extend the virtual address space by adding a *process identifier* or *task identifier*. The Intrinsity FastMATH has an 8-bit *address space ID* (ASID) field for this purpose. This small field identifies the currently running process; it is kept in a register loaded by the operating system when it switches processes. RISC-V also offers ASID to reduce TLB flushes on context switches. The process identifier is concatenated to the tag portion of the TLB, so that a TLB hit occurs only if both the page number *and* the process identifier match. This combination eliminates the need to clear the TLB, except on rare occasions.

Similar problems can occur for a cache, since on a process switch, the cache will contain data from the running process. These problems arise in different ways for physically addressed and virtually addressed caches, and a variety of solutions, such as process identifiers, are used to ensure that a process gets its own data.

context switch

A changing of the internal state of the processor to allow a different process to use the processor that includes saving the state needed to return to the currently executing process.

Handling TLB Misses and Page Faults

Although the translation of virtual to physical addresses with a TLB is straightforward when we get a TLB hit, as we saw earlier, handling TLB misses and page faults is more complex. A TLB miss occurs when no entry in the TLB matches a virtual address. Recall that a TLB miss can indicate one of two possibilities:
1. The page is present in memory, and we need only create the missing TLB entry.
2. The page is not present in memory, and we need to transfer control to the operating system to deal with a page fault.

Handling a TLB miss or a page fault requires using the exception mechanism to interrupt the active process, transferring control to the operating system, and later resuming execution of the interrupted process. A page fault will be recognized sometime during the clock cycle used to access memory. To restart the instruction after the page fault is handled, the program counter of the instruction that caused the page fault must be saved. The *supervisor exception program counter* (SEPC) register is used to hold this value.

In addition, a TLB miss or page fault exception must be asserted by the end of the same clock cycle that the memory access occurs, so that the next clock cycle will begin exception processing rather than continue normal instruction execution. If the page fault was not recognized in this clock cycle, a load instruction could overwrite a register, and this could be disastrous when we try to restart the instruction. For example, consider the instruction `lb x10, 0(x10)`: the computer must be able to prevent the write pipeline stage from occurring; otherwise, it could not properly restart the instruction, since the contents of `x10` would have been destroyed. A similar complication arises on stores. We must prevent the write into memory from actually completing when there is a page fault; this is usually done by deasserting the write control line to the memory.

Hardware/Software Interface

Between the time we begin executing the exception handler in the operating system and the time that the operating system has saved all the state of the process, the operating system is particularly vulnerable. For instance, if another exception occurred when we

were processing the first exception in the operating system, the control unit would overwrite the exception link register, making it impossible to return to the instruction that caused the page fault! We can avoid this disaster by providing the ability to **disable** and **enable exceptions**. When an exception first occurs, the processor sets a bit that disables all other exceptions; this could happen at the same time the processor sets the supervisor mode bit. The operating system will then save just enough state to allow it to recover if another exception occurs—namely, the *supervisor exception rogram counter* (SEPC) and the *supervisor exception cause* (SCAUSE) registers, which as we saw in Chapter 4 records the reason for the exception. SEPC and SCAUSE in RISC-V are two of the special control registers that help with exceptions, TLB misses, and page faults. The operating system can then re-enable exceptions. These steps make sure that exceptions will not cause the processor to lose any state and thereby be unable to restart execution of the interrupting instruction.

exception enable

Also called interrupt enable. A signal or action that controls whether the process responds to an exception or not; necessary for preventing the occurrence of exceptions during intervals before the processor has safely saved the state needed to restart.

Once the operating system knows the virtual address that caused the page fault, it must complete three steps:
1. Look up the page table entry using the virtual address and find the location of the referenced page in secondary memory.
2. Choose a physical page to replace; if the chosen page is dirty, it must be written out to secondary memory before we can bring a new virtual page into this physical page.
3. Start a read to bring the referenced page from secondary memory into the chosen physical page.

Of course, this last step will take millions of processor clock cycles for disks (so will the second if the replaced page is dirty); accordingly, the operating system will usually select another process to execute in the processor until the disk access completes. Because the operating system has saved the state of the process, it

can freely give control of the processor to another process.

When the read of the page from secondary memory is complete, the operating system can restore the state of the process that originally caused the page fault and execute the instruction that returns from the exception. This instruction will reset the processor from kernel to user mode, as well as restore the program counter. The user process then re-executes the instruction that faulted, accesses the requested page successfully, and continues execution.

Page fault exceptions for data accesses are difficult to implement properly in a processor because of a combination of three characteristics:

1. They occur in the middle of instructions, unlike instruction page faults.
2. The instruction cannot be completed before handling the exception.
3. After handling the exception, the instruction must be restarted as if nothing had occurred.

Making instructions **restartable**, so that the exception can be handled and the instruction later continued, is relatively easy in an architecture like the RISC-V. Because each instruction writes only one data item and this write occurs at the end of the instruction cycle, we can simply prevent the instruction from completing (by not writing) and restart the instruction at the beginning.

restartable instruction

An instruction that can resume execution after an exception is resolved without the exception's affecting the result of the instruction.

Elaboration

For processors with more complex instructions that can touch many memory locations and write many data items, making instructions restartable is much harder. Processing one instruction may generate a number of page faults in the middle of the instruction. For example, x86 processors have block move instructions that touch thousands of data words. In such processors, instructions often cannot be restarted from the beginning, as we do for RISC-V instructions. Instead, the

instruction must be interrupted and later continued midstream in its execution. Resuming an instruction in the middle of its execution usually requires saving some special state, processing the exception, and restoring that special state. Making this work properly requires careful and detailed coordination between the exception-handling code in the operating system and the hardware.

Elaboration

Rather than pay an extra level of indirection on every memory access, the Virtual Memory Monitor (Section 5.6) maintains a *shadow page table* that maps directly from the guest virtual address space to the physical address space of the hardware. By detecting all modifications to the guest's page table, the VMM can ensure the shadow page table entries being used by the hardware for translations correspond to those of the guest OS environment, with the exception of the correct physical pages substituted for the real pages in the guest tables. Hence, the VMM must trap any attempt by the guest OS to change its page table or to access the page table pointer. This is commonly done by write protecting the guest page tables and trapping any access to the page table pointer by a guest OS. As noted above, the latter happens naturally if accessing the page table pointer is a privileged operation.

Elaboration

The final portion of the architecture to virtualize is I/O. This is by far the most difficult part of system virtualization because of the increasing number of I/O devices attached to the computer *and* the expanding diversity of I/O device types. Another difficulty is the sharing of a real device among multiple VMs, and yet another comes from supporting the myriad of device drivers that are required, especially if different guest OSes are supported on the same VM system. The VM illusion can be maintained by giving each VM generic versions of each type of I/O device driver, and then leaving it to the VMM to handle real I/O.

Elaboration

In addition to virtualizing the instruction set for a virtual machine, another challenge is virtualization of virtual memory, as each guest OS in every virtual machine manages its own set of page tables. To make this work, the VMM separates the notions of *real* and *physical memory* (which are often treated synonymously), and makes real memory a separate, intermediate level between virtual memory and physical memory. (Some use the terms *virtual memory*, *physical memory*, and *machine memory* to name the same three levels.) The guest OS maps virtual memory to real memory via its page tables, and the VMM page tables map the guest's real memory to physical memory. The virtual memory architecture is typically specified via page tables, as in IBM VM/370, the x86, and RISC-V.

Summary

Virtual memory is the name for the level of memory hierarchy that manages caching between the main memory and secondary memory. Virtual memory allows a single program to expand its address space beyond the limits of main memory. More importantly, virtual memory supports sharing of the main memory among multiple, simultaneously active processes, in a protected manner.

Managing the memory hierarchy between main memory and disk is challenging because of the high cost of page faults. Several techniques are used to reduce the miss rate:

1. Pages are made large to take advantage of spatial locality and to reduce the miss rate.
2. The mapping between virtual addresses and physical addresses, which is implemented with a page table, is made fully associative so that a virtual page can be placed anywhere in main memory.
3. The operating system uses techniques, such as LRU and a reference bit, to choose which pages to replace.

Writes to secondary memory are expensive, so virtual memory uses a write-back scheme and also tracks whether a page is unchanged (using a dirty bit) to avoid writing clean pages.

The virtual memory mechanism provides address translation from a virtual address used by the program to the physical address space used for accessing memory. This address translation allows protected sharing of the main memory and provides several

additional benefits, such as simplifying memory allocation. Ensuring that processes are protected from each other requires that only the operating system can change the address translations, which is implemented by preventing user programs from altering the page tables. Controlled sharing of pages between processes can be implemented with the help of the operating system and access bits in the page table that indicate whether the user program has read or write access to a page.

If a processor had to access a page table resident in memory to translate every access, virtual memory would be too expensive, as caches would be pointless! Instead, a TLB acts as a cache for translations from the page table. Addresses are then translated from virtual to physical using the translations in the TLB.

Caches, virtual memory, and TLBs all rely on a common set of principles and policies. The next section discusses this common framework.

Understanding Program Performance

Although virtual memory was invented to enable a small memory to act as a large one, the performance difference between secondary memory and main memory means that if a program routinely accesses more virtual memory than it has physical memory, it will run very slowly. Such a program would be continuously swapping pages between main memory and secondary memory, called *thrashing*. Thrashing is a disaster if it occurs, but it is rare. If your program thrashes, the easiest solution is to run it on a computer with more memory or buy more memory for your computer. A more complex choice is to re-examine your algorithm and data structures to see if you can change the locality and thereby reduce the number of pages that your program uses simultaneously. This set of popular pages is informally called the *working set*.

A more common performance problem is TLB misses. Since a TLB might handle only 32–64 page entries at a time, a program could easily see a high TLB miss rate, as the processor may access less than a quarter mebibyte directly: 64 × 4 KiB =0.25 MiB. For example, TLB misses are often a challenge for Radix Sort. To try to alleviate this problem, most computer architectures now offer support for larger page sizes. For instance, in addition to the

minimum 4 KiB page, RISC-V hardware supports 2 MiB and 1 GiB pages. Hence, if a program uses large page sizes, it can access more memory directly without TLB misses.

The practical challenge is getting the operating system to allow programs to select these larger page sizes. Once again, the more complex solution to reducing TLB misses is to re-examine the algorithm and data structures to reduce the working set of pages; given the importance of memory accesses to performance and the frequency of TLB misses, some programs with large working sets have been redesigned with that goal.

Elaboration

RISC-V supports the larger page sizes via the multi-level page table of Figure 5.29. In addition to pointing at the next level page table in levels 1 and 2, it allows a *superpage translation* to map the virtual address to a 1 GiB physical address (if the block translation is in level 1) or a 2 MiB physical address (if the block translation is in level 2).

5.8 A Common Framework for Memory Hierarchy

By now, you've recognized that the different types of memory hierarchies have a great deal in common. Although many of the aspects of memory hierarchies differ quantitatively, many of the policies and features that determine how a hierarchy functions are similar qualitatively. Figure 5.34 shows how some of the quantitative characteristics of memory hierarchies can differ. In the rest of this section, we will discuss the common operational alternatives for memory hierarchies, and how these determine their behavior. We will examine these policies in a series of four questions that apply between any two levels of a memory hierarchy, although for simplicity, we will primarily use terminology for caches.

Feature	Typical values for L1 caches	Typical values for L2 caches	Typical values for paged memory	Typical values for a TLB
Total size in blocks	250–2000	2500–25,000	16,000–250,000	40–1024
Total size in kilobytes	16–64	125–2000	1,000,000–1,000,000,000	0.25–16
Block size in bytes	16–64	64–128	4000–64,000	4–32
Miss penalty in clocks	10–25	100–1000	10,000,000–100,000,000	10–1000
Miss rates (global for L2)	2%–5%	0.1%–2%	0.00001%–0.0001%	0.01%–2%

FIGURE 5.34 The key quantitative design parameters that characterize the major elements of memory hierarchy in a computer.
These are typical values for these levels as of 2012. Although the range of values is wide, this is partially because many of the values that have shifted over time are related; for example, as caches become larger to overcome larger miss penalties, block sizes also grow. While not shown, server microprocessors today also have L3 caches, which can be 2 to 8 MiB and contain many more blocks than L2 caches. L3 caches lower the L2 miss penalty to 30 to 40 clock cycles.

Question 1: Where Can a Block Be Placed?

We have seen that block placement in the upper level of the hierarchy can use a range of schemes, from direct mapped to set associative to fully associative. As mentioned above, this entire range of schemes can be thought of as variations on a set-associative scheme where the number of sets and the number of blocks per set varies:

Scheme name	Number of sets	Blocks per set
Direct mapped	Number of blocks in cache	1
Set associative	$\frac{\text{Number of blocks in the cache}}{\text{Associativity}}$	Associativity (typically 2–16)
Fully associative	1	Number of blocks in the cache

The advantage of increasing the degree of associativity is that it usually decreases the miss rate. The improvement in miss rate comes from reducing misses that compete for the same location. We will examine these in more detail shortly. First, let's look at how much improvement is gained. Figure 5.35 shows the miss rates for several cache sizes as associativity varies from direct mapped to

eight-way set associative. The largest gains are obtained in going from direct mapped to two-way set associative, which yields between a 20% and 30% reduction in the miss rate. As cache sizes grow, the relative improvement from associativity increases only slightly; since the overall miss rate of a larger cache is lower, the opportunity for improving the miss rate decreases and the absolute improvement in the miss rate from associativity shrinks significantly. The potential disadvantages of associativity, as we mentioned earlier, are increased cost and slower access time.

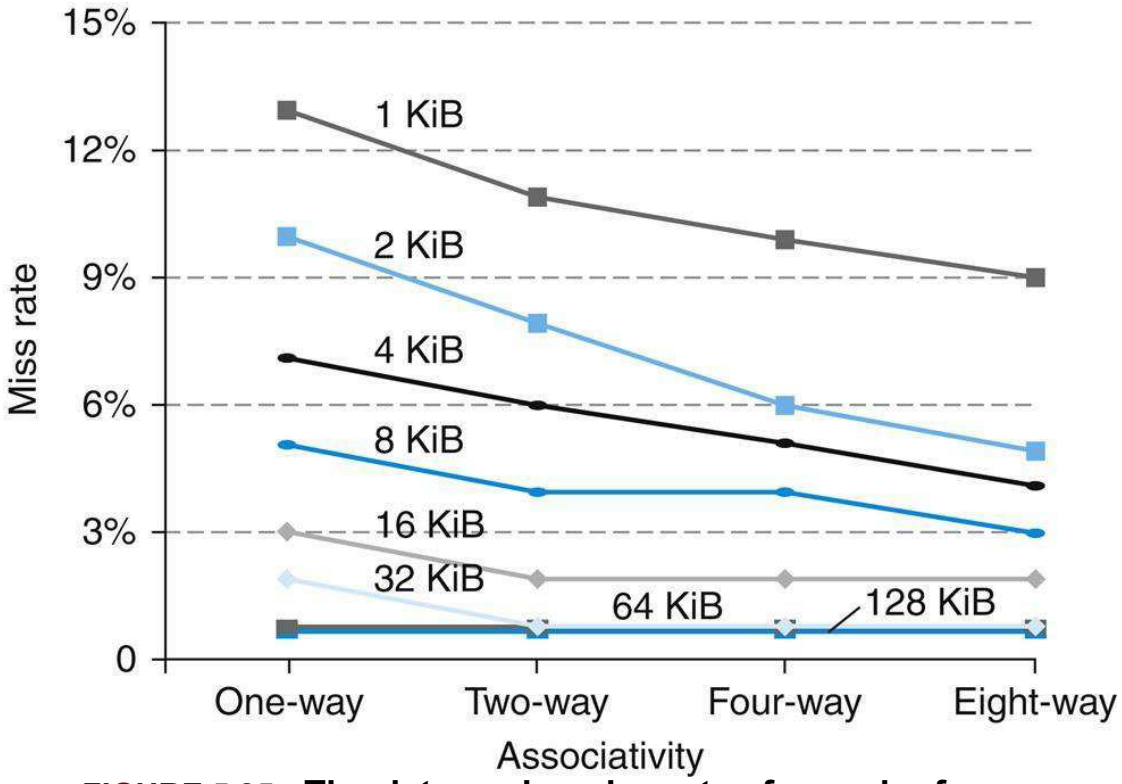


FIGURE 5.35 The data cache miss rates for each of eight cache sizes improve as the associativity increases.

While the benefit of going from one-way (direct mapped) to two-way set associative is significant, the benefits of further associativity are smaller (e.g., 1–10% improvement going from two-way to four-way versus 20–30% improvement going from one-way to two-way). There is even less improvement in going from four-way to eight-way set associative, which, in turn, comes very close to the miss rates of a fully associative cache. Smaller caches obtain a significantly larger absolute benefit from associativity

because the base miss rate of a small cache is larger. Figure 5.16 explains how these data were collected.

Question 2: How Is a Block Found?

The choice of how we locate a block depends on the block placement scheme, since that dictates the number of possible locations. We can summarize the schemes as follows:

Associativity	Location method	Comparisons required
Direct mapped	Index	1
Set associative	Index the set, search among elements	Degree of associativity
Full	Search all cache entries	Size of the cache
	Separate lookup table	0

The choice among direct-mapped, set-associative, or fully associative mapping in any memory hierarchy will depend on the cost of a miss versus the cost of implementing associativity, both in time and in extra hardware. Including the L2 cache on the chip enables much higher associativity, because the hit times are not as critical and the designer does not have to rely on standard SRAM chips as the building blocks. Fully associative caches are prohibitive except for small sizes, where the cost of the comparators is not overwhelming and where the absolute miss rate improvements are greatest.

In virtual memory systems, a separate mapping table—the page table—is kept to index the memory. In addition to the storage needed for the table, using an index table requires an extra memory access. The choice of full associativity for page placement and the extra table is motivated by these facts:

1. Full associativity is beneficial, since misses are very expensive.
2. Full associativity allows software to use sophisticated replacement schemes that are designed to reduce the miss rate.
3. The full map can be easily indexed with no extra hardware and no searching required.

Therefore, virtual memory systems almost always use fully associative placement.

Set-associative placement is often used for caches and TLBs, where the access combines indexing and the search of a small set. A few systems have used direct-mapped caches because of their advantage in access time and simplicity. The advantage in access time occurs because finding the requested block does not depend on a comparison. Such design choices depend on many details of the implementation, such as whether the cache is on-chip, the technology used for implementing the cache, and the critical role of cache access time in determining the processor cycle time.

Question 3: Which Block Should Be Replaced on a Cache Miss?

When a miss occurs in an associative cache, we must decide which block to replace. In a fully associative cache, all blocks are candidates for replacement. If the cache is set associative, we must choose among the blocks in the set. Of course, replacement is easy in a direct-mapped cache because there is only one candidate.

There are the two primary strategies for replacement in set-associative or fully associative caches:

- *Random:* Candidate blocks are randomly selected, possibly using some hardware assistance.
- *Least recently used* (LRU): The block replaced is the one that has been unused for the longest time.

In practice, LRU is too costly to implement for hierarchies with more than a small degree of associativity (two to four, typically), since tracking the usage information is expensive. Even for four-way set associativity, LRU is often approximated—for example, by keeping track of which pair of blocks is LRU (which requires 1 bit), and then tracking which block in each pair is LRU (which requires 1 bit per pair).

For larger associativity, either LRU is approximated or random replacement is used. In caches, the replacement algorithm is in hardware, which means that the scheme should be easy to implement. Random replacement is simple to build in hardware, and for a two-way set-associative cache, random replacement has a miss rate about 1.1 times higher than LRU replacement. As the caches become larger, the miss rate for both replacement strategies falls, and the absolute difference becomes small. In fact, random

replacement can sometimes be better than the simple LRU approximations that are easily implemented in hardware.

In virtual memory, some form of LRU is always approximated, since even a tiny reduction in the miss rate can be important when the cost of a miss is enormous. Reference bits or equivalent functionality are often provided to make it easier for the operating system to track a set of less recently used pages. Because misses are so expensive and relatively infrequent, approximating this information primarily in software is acceptable.

Question 4: What Happens on a Write?

A key characteristic of any memory hierarchy is how it deals with writes. We have already seen the two basic options:

- *Write-through:* The information is written to both the block in the cache and the block in the lower level of the memory hierarchy (main memory for a cache). The caches in Section 5.3 used this scheme.
- *Write-back:* The information is written just to the block in the cache. The modified block is written to the lower level of the hierarchy only when it is replaced. Virtual memory systems always use write-back, for the reasons discussed in Section 5.7.

Both write-back and write-through have their advantages. The key advantages of write-back are the following:

- Individual words can be written by the processor at the rate that the cache, rather than the memory, can accept them.
- Multiple writes within a block require only one write to the lower level in the hierarchy.
- When blocks are written back, the system can make effective use of a high-bandwidth transfer, since the entire block is written.

Write-through has these advantages:

- Misses are simpler and cheaper because they never require a block to be written back to the lower level.
- Write-through is easier to implement than write-back, although to be realistic, a write-through cache will still need to use a write buffer.

The BIG Picture

Caches, TLBs, and virtual memory may initially look very

different, but they rely on the same two principles of locality, and they can be understood by their answers to four questions:

Question 1:	Where can a block be placed?
Answer:	One place (direct mapped), a few places (set associative), or any place (fully associative).
Question 2:	How is a block found?
Answer:	There are four methods: indexing (as in a direct-mapped cache), limited search (as in a set-associative cache), full search (as in a fully associative cache), and a separate lookup table (as in a page table).
Question 3:	What block is replaced on a miss?
Answer:	Typically, either the least recently used or a random block.
Question 4:	How are writes handled?
Answer:	Each level in the hierarchy can use either write-through or write-back.

In virtual memory systems, only a write-back policy is practical because of the long latency of a write to the lower level of the hierarchy. The rate at which writes are generated by a processor generally exceeds the rate at which the memory system can process them, even allowing for physically and logically wider memories and burst modes for DRAM. Consequently, today lowest-level caches typically use write-back.

The Three Cs: An Intuitive Model for Understanding the Behavior of Memory Hierarchies

In this subsection, we look at a model that provides insight into the sources of misses in a memory hierarchy and how the misses will be affected by changes in the hierarchy. We will explain the ideas in terms of caches, although the ideas carry over directly to any other level in the hierarchy. In this model, all misses are classified into one of three categories (the **three Cs**):

three Cs model

A cache model in which all cache misses are classified into one of three categories: compulsory misses, capacity misses, and conflict misses.

- **Compulsory misses**: These are cache misses caused by the first access to a block that has never been in the cache. These are also called **cold-start misses**.
- **Capacity misses**: These are cache misses caused when the cache cannot contain all the blocks needed during execution of a program. Capacity misses occur when blocks are replaced and then later retrieved.
- **Conflict misses**: These are cache misses that occur in set-associative or direct-mapped caches when multiple blocks compete for the same set. Conflict misses are those misses in a direct-mapped or set-associative cache that are eliminated in a fully associative cache of the same size. These cache misses are also called **collision misses**.

compulsory miss

Also called **cold-start miss**. A cache miss caused by the first access to a block that has never been in the cache.

capacity miss

A cache miss that occurs because the cache, even with full associativity, cannot contain all the blocks needed to satisfy the request.

conflict miss

Also called **collision miss**. A cache miss that occurs in a set-associative or direct-mapped cache when multiple blocks compete for the same set and that are eliminated in a fully associative cache of the same size.

Figure 5.36 shows how the miss rate divides into the three sources. These sources of misses can be directly attacked by changing some aspect of the cache design. Since conflict misses arise straight from contention for the same cache block, increasing associativity reduces conflict misses. Associativity, however, may slow access time, leading to lower overall performance.

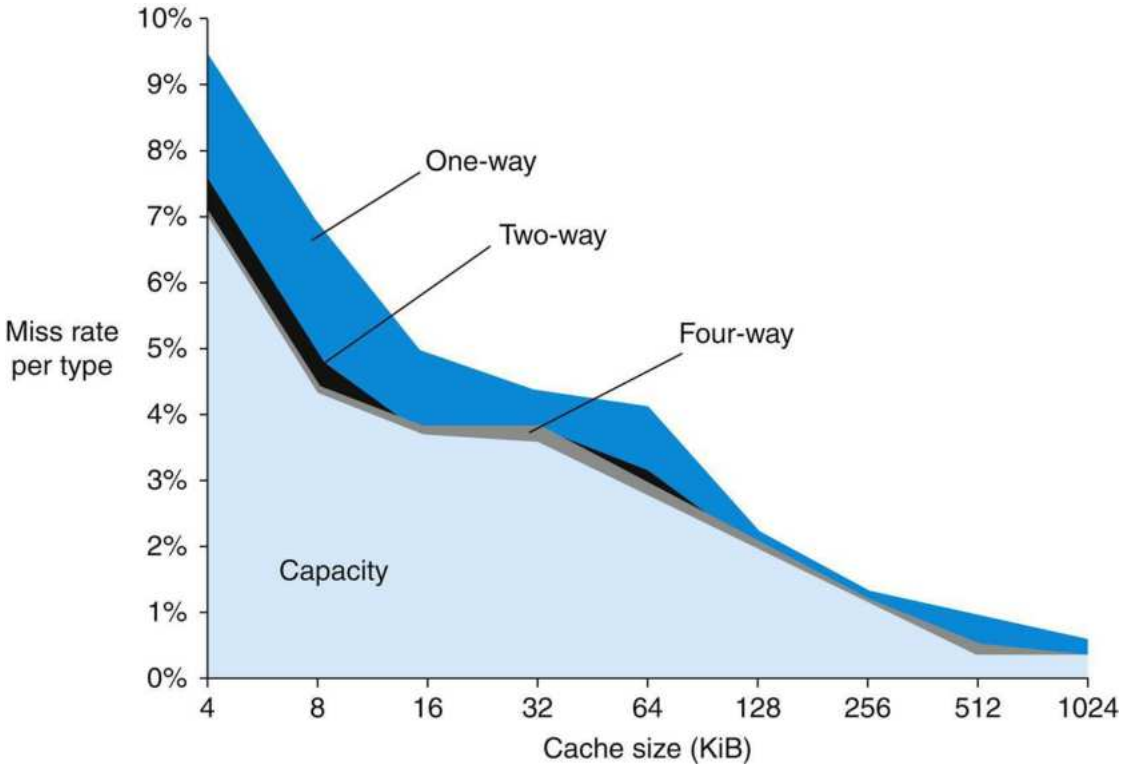


FIGURE 5.36 The miss rate can be broken into three sources of misses.
This graph shows the total miss rate and its components for a range of cache sizes. These data are for the SPEC CPU2000 integer and floating-point benchmarks and are from the same source as the data in Figure 5.35. The compulsory miss component is 0.006% and cannot be seen in this graph. The next component is the capacity miss rate, which depends on cache size. The conflict portion, which depends both on associativity and on cache size, is shown for a range of associativities from one-way to eight-way. In each case, the labeled section corresponds to the increase in the miss rate that occurs when the associativity is changed from the next higher degree to the labeled degree of associativity. For example, the section labeled *two-way* indicates the additional misses arising when the cache has associativity of two rather than four. Thus, the difference in the miss rate incurred by a direct-mapped cache versus a fully associative cache of the same size is given by the sum of the sections marked *four-way, two-way*, and *one-way*. The difference between eight-way and four-way is so small that it is difficult to see on this graph.

Capacity misses can easily be reduced by enlarging the cache;

indeed, second-level caches have been growing steadily bigger for many years. Of course, when we make the cache larger, we must also be careful about increasing the access time, which could lead to lower overall performance. Thus, first-level caches have been growing slowly, if at all.

Because compulsory misses are generated by the first reference to a block, the primary way for the cache system to reduce the number of compulsory misses is to increase the block size. This will reduce the number of references required to touch each block of the program once, because the program will consist of fewer cache blocks. As mentioned above, increasing the block size too much can have a negative effect on performance because of the increase in the miss penalty.

The BIG Picture

The challenge in designing memory hierarchies is that every change that potentially improves the miss rate can also negatively affect overall performance, as Figure 5.37 summarizes. This combination of positive and negative effects is what makes the design of a memory hierarchy interesting.

Design change	Effect on miss rate	Possible negative performance effect
Increases cache size	Decreases capacity misses	May increase access time
Increases associativity	Decreases miss rate due to conflict misses	May increase access time
Increases block size	Decreases miss rate for a wide range of block sizes due to spatial locality	Increases miss penalty. Very large block could increase miss rate

FIGURE 5.37 Memory hierarchy design challenges.

The decomposition of misses into the three Cs is a useful qualitative model. In real cache designs, many of the design choices interact, and changing one cache characteristic will often affect several components of the miss rate. Despite such shortcomings, this model is a useful way to gain insight into the performance of cache designs.

Check Yourself

Which of the following statements (if any) is generally true?
1. There is no way to reduce compulsory misses.
2. Fully associative caches have no conflict misses.
3. In reducing misses, associativity is more important than capacity.

5.9 Using a Finite-State Machine to Control a Simple Cache

We can now build control for a cache, just as we implemented control for the single-cycle and pipelined datapaths in Chapter 4. This section starts with a definition of a simple cache and then a description of *finite-state machines* (FSMs). It finishes with the FSM of a controller for this simple cache.  Section 5.12 goes into more depth, showing the cache and controller in a new hardware description language.

A Simple Cache

We're going to design a controller for a straightforward cache. Here are the key characteristics of the cache:
- Direct-mapped cache
- Write-back using write allocate
- Block size is four words (16 bytes or 128 bits)
- Cache size is 16 KiB, so it holds 1024 blocks
- 32-bit addresses
- The cache includes a valid bit and dirty bit per block

From Section 5.3, we can now calculate the fields of an address for the cache:
- Cache index is 10 bits
- Block offset is 4 bits
- Tag size is 32−(10+4) or 18 bits

The signals between the processor to the cache are
- 1-bit Read or Write signal
- 1-bit Valid signal, saying whether there is a cache operation or not
- 32-bit address
- 32-bit data from processor to cache

- 32-bit data from cache to processor
- 1-bit Ready signal, saying the cache operation is complete

The interface between the memory and the cache has the same fields as between the processor and the cache, except that the data fields are now 128 bits wide. The extra memory width is generally found in microprocessors today, which deal with either 32-bit or 64-bit words in the processor while the DRAM controller is often 128 bits. Making the cache block match the width of the DRAM simplified the design. Here are the signals:

- 1-bit Read or Write signal
- 1-bit Valid signal, saying whether there is a memory operation or not
- 32-bit address
- 128-bit data from cache to memory
- 128-bit data from memory to cache
- 1-bit Ready signal, saying the memory operation is complete

Note that the interface to memory is not a fixed number of cycles. We assume a memory controller that will notify the cache via the Ready signal when the memory read or write is finished.

Before describing the cache controller, we need to review finite-state machines, which allow us to control an operation that can take multiple clock cycles.

Finite-State Machines

To design the control unit for the single-cycle datapath, we used truth tables that specified the setting of the control signals based on the instruction class. For a cache, the control is more complex because the operation can be a series of steps. The control for a cache must specify both the signals to be set in any step and the next step in the sequence.

The most common multistep control method is based on **finite-state machines**, which are usually represented graphically. A finite-state machine consists of a set of states and directions on how to change states. The directions are defined by a **next-state function**, which maps the current state and the inputs to a new state. When we use a finite-state machine for control, each state also specifies a set of outputs that are asserted when the machine is in that state. The implementation of a finite-state machine usually

assumes that all outputs that are not explicitly asserted are deasserted. Similarly, the correct operation of the datapath depends on the fact that a signal that is not explicitly asserted is deasserted, rather than acting as a don't care.

finite-state machine

A sequential logic function consisting of a set of inputs and outputs, a next-state function that maps the current state and the inputs to a new state, and an output function that maps the current state and possibly the inputs to a set of asserted outputs.

next-state function

A combinational function that, given the inputs and the current state, determines the next state of a finite-state machine.

Multiplexor controls are slightly different, since they select one of the inputs, whether they are 0 or 1. Thus, in the finite-state machine, we always specify the setting of all the multiplexor controls that we care about. When we implement the finite-state machine with logic, setting a control to 0 may be the default and therefore may not require any gates. A simple example of a finite-state machine appears in Appendix A, and if you are unfamiliar with the concept of a finite-state machine, you may want to examine Appendix A before proceeding.

A finite-state machine can be implemented with a temporary register that holds the current state and a block of combinational logic that determines both the data-path signals to be asserted and the next state. Figure 5.38 shows how such an implementation might look.  Appendix C describes in detail how the finite-state machine is implemented using this structure. In Section A.3, the combinational control logic for a finite-state machine is implemented both with either a ROM (*read-only memory*) or a PLA (*programmable logic array*). (Also see Appendix A for a description of these logic elements.)

Elaboration

Note that this simple design is called a *blocking* cache, in that the processor must wait until the cache has finished the request.  Section 5.12 describes the alternative, which is called a *nonblocking* cache.

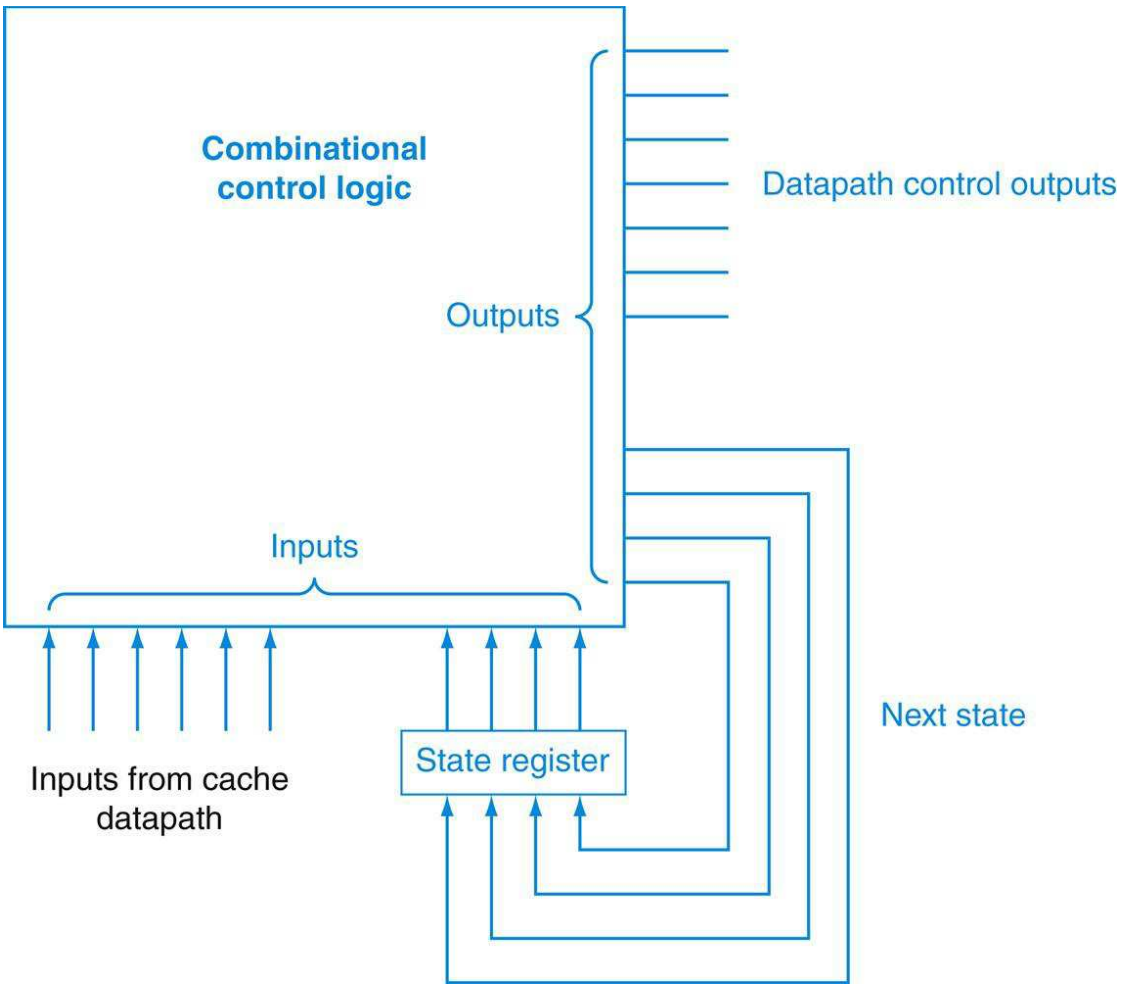


FIGURE 5.38 Finite-state machine controllers are typically implemented using a block of combinational logic and a register to hold the current state.
The outputs of the combinational logic are the next-state number and the control signals to be asserted for the current state. The inputs to the combinational logic are the current state and any inputs used to determine the next state. Notice that in the finite-state machine used in this chapter, the outputs depend only on the current state, not on the inputs. We use color to indicate that these are control lines and logic versus data lines and logic. The *Elaboration* below explains this in more detail.

Elaboration

The style of finite-state machine in this book is called a Moore machine, after Edward Moore. Its identifying characteristic is that

the output depends only on the current state. For a Moore machine, the box labeled combinational control logic can be split into two pieces. One piece has the control output and only the state input, while the other has just the next-state output.

An alternative style of machine is a Mealy machine, named after George Mealy. The Mealy machine allows both the input and the current state to be used to determine the output. Moore machines have potential implementation advantages in speed and size of the control unit. The speed advantages arise because the control outputs, which are needed early in the clock cycle, do not depend on the inputs, but only on the current state. In Appendix A, when the implementation of this finite-state machine is taken down to logic gates, the size advantage can be clearly seen. The potential disadvantage of a Moore machine is that it may require additional states. For example, in situations where there is a one-state difference between two sequences of states, the Mealy machine may unify the states by making the outputs depend on the inputs.

FSM for a Simple Cache Controller

Figure 5.39 shows the four states of our simple cache controller:

- *Idle:* This state waits for a valid read or write request from the processor, which moves the FSM to the Compare Tag state.
- *Compare Tag:* As the name suggests, this state tests to see if the requested read or write is a hit or a miss. The index portion of the address selects the tag to be compared. If the data in the cache block referred to by the index portion of the address are valid, and the tag portion of the address matches the tag, then it is a hit. Either the data are read from the selected word if it is a load or written to the selected word if it is a store. The Cache Ready signal is then set. If it is a write, the dirty bit is set to 1. Note that a write hit also sets the valid bit and the tag field; while it seems unnecessary, it is included because the tag is a single memory, so to change the dirty bit we likewise need to change the valid and tag fields. If it is a hit and the block is valid, the FSM returns to the idle state. A miss first updates the cache tag and then goes either to the Write-Back state, if the block at this location has dirty bit value of 1, or to the Allocate state if it is 0.
- *Write-Back:* This state writes the 128-bit block to memory using the

address composed from the tag and cache index. We remain in this state waiting for the Ready signal from memory. When the memory write is complete, the FSM goes to the Allocate state.

- *Allocate:* The new block is fetched from memory. We remain in this state waiting for the Ready signal from memory. When the memory read is complete, the FSM goes to the Compare Tag state. Although we could have gone to a new state to complete the operation instead of reusing the Compare Tag state, there is a good deal of overlap, including the update of the appropriate word in the block if the access was a write.

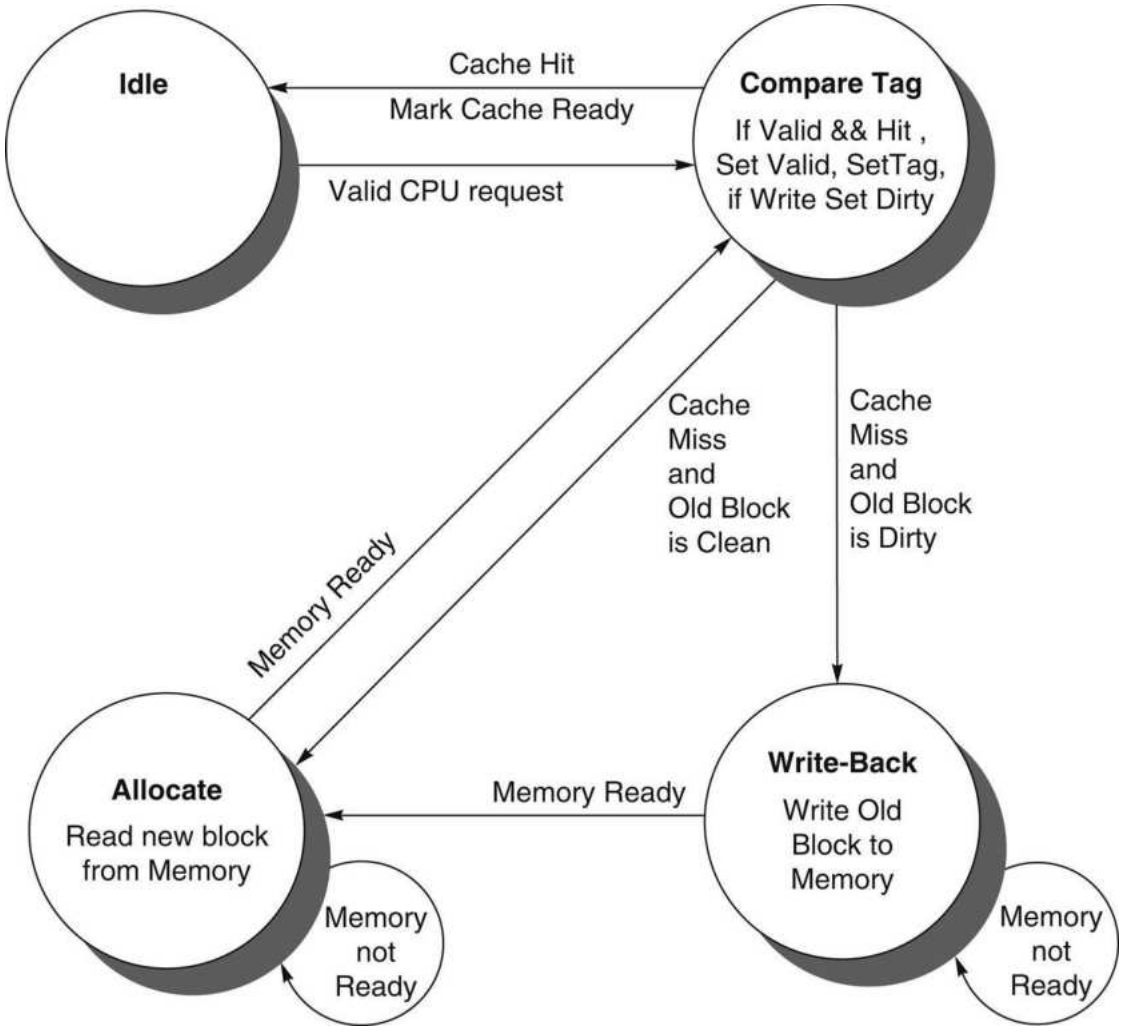


FIGURE 5.39 Four states of the simple controller.

This simple model could easily be extended with more states to try to improve performance. For example, the Compare Tag state does both the compare and the read or write of the cache data in a single clock cycle. Often the compare and cache access are done in separate states to try to improve the clock cycle time. Another optimization would be to add a write buffer so that we could save the dirty block and then read the new block first so that the processor doesn't have to wait for two memory accesses on a dirty miss. The cache would next write the dirty block from the write buffer while the processor is operating on the requested data.

 Section 5.12 goes into more detail about the FSM, showing the full controller in a hardware description language and a block diagram of this simple cache.

5.10 Parallelism and Memory Hierarchy: Cache Coherence

Given that a multicore multiprocessor means multiple processors on a single chip, these processors very likely share a common physical address space. Caching shared data introduces a new problem, because the view of memory held by two different processors is through their individual caches, which, without any additional precautions, could end up seeing two distinct values. Figure 5.40 illustrates the problem and shows how two different processors can have two different values for the same location. This difficulty is generally referred to as the *cache coherence problem*.

Time step	Event	Cache contents for CPU A	Cache contents for CPU B	Memory contents for location X
0				0
1	CPU A reads X	0		0
2	CPU B reads X	0	0	0
3	CPU A stores 1 into X	1	0	1

FIGURE 5.40 The cache coherence problem for a single memory location (X), read and written by two processors (A and B).
We initially assume that neither cache contains the variable and that X has the value 0. We also assume a write-through cache; a write-back cache adds some additional but similar complications. After the value of X has been written by A, A's cache and the memory both contain the new value, but B's cache does not, and if B reads the value of X, it will receive 0!

Informally, we could say that a memory system is coherent if any read of a data item returns the most recently written value of that data item. This definition, although intuitively appealing, is vague and simplistic; the reality is much more complex. This simple definition contains two different aspects of memory system behavior, both of which are critical to writing correct shared memory programs. The first aspect, called *coherence*, defines *what values* can be returned by a read. The second aspect, called

consistency, determines *when* a written value will be returned by a read.

Let's look at coherence first. A memory system is coherent if
1. A read by a processor P to a location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P. Thus, in Figure 5.40, if CPU A were to read X after time step 3, it should see the value 1.
2. A read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated in time and no other writes to X occur between the two accesses. Thus, in Figure 5.40, we need a mechanism so that the value 0 in the cache of CPU B is replaced by the value 1 after CPU A stores 1 into memory at address X in time step 3.
3. Writes to the same location are *serialized*; that is, two writes to the same location by any two processors are seen in the same order by all processors. For example, if CPU B stores 2 into memory at address X after time step 3, processors can never read the value at location X as 2 and then later read it as 1.

The first property simply preserves program order—we certainly expect this property to be true in uniprocessors, for instance. The second property defines the notion of what it means to have a coherent view of memory: if a processor could continuously read an old data value, we would clearly say that memory was incoherent.

The need for *write serialization* is more subtle, but equally important. Suppose we did not serialize writes, and processor P1 writes location X followed by P2 writing location X. Serializing the writes ensures that every processor will see the write done by P2 at some point. If we did not serialize the writes, it might be the case that some processor could see the write of P2 first and then see the write of P1, maintaining the value written by P1 indefinitely. The simplest way to avoid such difficulties is to ensure that all writes to the same location are seen in the identical order, which we call *write serialization*.

Basic Schemes for Enforcing Coherence

In a cache coherent multiprocessor, the caches provide both

migration and *replication* of shared data items:

- *Migration:* A data item can be moved to a local cache and used there in a transparent fashion. Migration reduces both the latency to access a shared data item that is allocated remotely and the bandwidth demand on the shared memory.
- *Replication:* When shared data are being simultaneously read, the caches make a copy of the data item in the local cache. Replication reduces both latency of access and contention for a read shared data item.

Supporting migration and replication is critical to performance in accessing shared data, so many multiprocessors introduce a hardware protocol to maintain coherent caches. The protocols to maintain coherence for multiple processors are called *cache coherence protocols*. Key to implementing a cache coherence protocol is tracking the state of any sharing of a data block.

The most popular cache coherence protocol is *snooping*. Every cache that has a copy of the data from a block of physical memory also has a copy of the sharing status of the block, but no centralized state is kept. The caches are all accessible via some broadcast medium (a bus or network), and all cache controllers monitor or *snoop* on the medium to determine whether or not they have a copy of a block that is requested on a bus or switch access.

In the following section we explain snooping-based cache coherence as implemented with a shared bus, but any communication medium that broadcasts cache misses to all processors can be used to implement a snooping-based coherence scheme. This broadcasting to all caches makes snooping protocols simple to implement but also limits their scalability.

Snooping Protocols

One method of enforcing coherence is to ensure that a processor has exclusive access to a data item before it writes that item. This style of protocol is called a *write invalidate protocol* because it invalidates copies in other caches on a write. Exclusive access ensures that no other readable or writable copies of an item exist when the write occurs: all other cached copies of the item are invalidated.

Figure 5.41 shows an example of an invalidation protocol for a snooping bus with write-back caches in action. To see how this

protocol ensures coherence, consider a write followed by a read by another processor: since the write requires exclusive access, any copy held by the reading processor must be invalidated (hence the protocol name). Thus, when the read occurs, it misses in the cache, and the cache is forced to fetch a new copy of the data. For a write, we require that the writing processor have exclusive access, preventing any other processor from being able to write simultaneously. If two processors do attempt to write the same data at the same time, one of them wins the race, causing the other processor's copy to be invalidated. For the other processor to complete its write, it must obtain a new copy of the data, which must now contain the updated value. Therefore, this protocol also enforces write serialization.

Hardware/Software Interface

One insight is that block size plays an important role in cache coherency. For example, take the case of snooping on a cache with a block size of eight words, with a single word alternatively written and read by two processors. Most protocols exchange full blocks between processors, thereby increasing coherency bandwidth demands.

Large blocks can also cause what is called **false sharing**: when two unrelated shared variables are located in the same cache block, the whole block is exchanged between processors even though the processors are accessing different variables. Programmers and compilers should lay out data carefully to avoid false sharing.

false sharing

When two unrelated shared variables are located in the same cache block and the full block is exchanged between processors even though the processors are accessing different variables.

Elaboration

Although the three properties on page 455 are sufficient to ensure coherence, the question of when a written value is seen is also important. To see why, observe that we cannot require that a read of X in Figure 5.40 instantaneously sees the value written for X by

some other processor. If, for example, a write of X on one processor precedes a read of X on another processor very shortly beforehand, it may be impossible to ensure that the read returns the value of the data written, since the written data may not even have left the processor at that point. The issue of exactly *when* a written value must be seen by a reader is defined by a *memory consistency model*.

We make the following two assumptions. First, a write does not complete (and allow the next write to occur) until all processors have seen the effect of that write. Second, the processor does not change the order of any write with respect to any other memory access. These two conditions mean that if a processor writes location X followed by location Y, any processor that sees the new value of Y must also see the new value of X. These restrictions allow the processor to reorder reads, but force the processor to finish a write in program order.

Elaboration

Since input can change memory behind the caches, and since output could need the latest value in a write back cache, there is also a cache coherency problem for I/O with the caches of a single processor as well as just between caches of multiple processors. The cache coherence problem for multiprocessors and I/O (see Chapter 6), although similar in origin, has different characteristics that affect the appropriate solution. Unlike I/O, where multiple data copies are a rare event—one to be avoided whenever possible—a program running on multiple processors will normally have copies of the same data in several caches.

Elaboration

In addition to the snooping cache coherence protocol where the status of shared blocks is distributed, a *directory-based* cache coherence protocol keeps the sharing status of a block of physical memory in just one location, called the *directory*. Directory-based coherence has slightly higher implementation overhead than snooping, but it can reduce traffic between caches and thus scale to larger processor counts.

Processor activity	Bus activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of memory location X
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes a 1 to X	Invalidation for X	1		0
CPU B reads X	Cache miss for X	1	1	1

FIGURE 5.41 An example of an invalidation protocol working on a snooping bus for a single cache block (X) with write-back caches.

We assume that neither cache initially holds X and that the value of X in memory is 0. The CPU and memory contents show the value after the processor and bus activity have both completed. A blank indicates no activity or no copy cached. When the second miss by B occurs, CPU A responds with the value canceling the response from memory. In addition, both the contents of B's cache and the memory contents of X are updated. This update of memory, which occurs when a block becomes shared, simplifies the protocol, but it is possible to track the ownership and force the write-back only if the block is replaced. This requires the introduction of an additional state called "owner," which indicates that a block may be shared, but the owning processor is responsible for updating any other processors and memory when it changes the block or replaces it.

Parallelism and Memory Hierarchy: Redundant Arrays of Inexpensive Disks

This online section describes how using many disks in conjunction can offer much higher throughput, which was the original inspiration of *Redundant Arrays of Inexpensive Disks* (RAID). The real popularity of RAID, however, was due more to the considerably greater dependability offered by including a modest number of redundant disks. The section explains the differences in performance, cost, and **dependability** between the RAID levels.

5.11 Parallelism and Memory Hierarchy: Redundant Arrays of Inexpensive Disks

Amdahl's law in Chapter 1 reminds us that neglecting I/O in this parallel revolution is foolhardy. A simple example demonstrates this.

Impact of I/O on System Performance

Example

Suppose we have a benchmark that executes in 100 seconds of elapsed time, of which 90 seconds is CPU time, and the rest is I/O time. Suppose the number of processors doubles every 2 years, but the processors remain at the same speed, and I/O time doesn't improve. How much faster will our program run at the end of 6 years?

Answer

We know that

$$\begin{aligned}\text{Elapsed time} &= \text{CPU time} + \text{I/O time}\\ 100 &= 90 + \text{I/O time}\\ \text{I/O time} &= 10\,\text{seconds}\end{aligned}$$

The new CPU times and the resulting elapsed times are computed in the following table.

After *n* years	CPU time	I/O time	Elapsed time	% I/O time
0 years	90 seconds	10 seconds	100 seconds	10%
2 years	$\frac{90}{2} = 45$ seconds	10 seconds	55 seconds	18%
4 years	$\frac{45}{2} = 23$ seconds	10 seconds	33 seconds	31%
6 years	$\frac{23}{2} = 11$ seconds	10 seconds	21 seconds	47%

The improvement in CPU performance after 6 years is

$$\frac{90}{11} = 8$$

However, the improvement in elapsed time is only

$$\frac{100}{21} = 4.7$$

and the I/O time has increased from 10% to 47% of the elapsed time.

Hence, the parallel revolution needs to come to I/O as well as to computation, or the effort spent in parallelizing could be squandered whenever programs do I/O, which they all must do.

Accelerating I/O performance was the original motivation of disk arrays. In the late 1980s, the high-performance storage of choice was

large, expensive disks. The argument was that by replacing a few big disks with many small disks, performance would improve because there would be more read heads. This shift is a good match for multiple processors as well, since many read/write heads mean the storage system could support many more independent accesses as well as large transfers spread across many disks. That is, you could get both high I/Os per second and high data transfer rates. In addition to higher performance, there could be advantages in cost, power, and floor space, since smaller disks are generally more efficient per gigabyte than larger disks.

The flaw in the argument was that disk arrays could make reliability much worse. These smaller, inexpensive drives had lower MTTF ratings than the large drives, but more importantly, by replacing a single drive with, say, 50 small drives, the failure rate would go up by at least a factor of 50.

The solution was to add redundancy so that the system could cope with disk failures without losing information. By having many little disks, the cost of extra redundancy to improve dependability is small, relative to the solutions for a few large disks. Thus, **dependability** was more affordable if you constructed a redundant array of inexpensive disks. This observation led to its name: **redundant arrays of inexpensive disks**, abbreviated **RAID**.

redundant arrays of inexpensive disks (RAID)

An organization of disks that uses an array of small and inexpensive disks so as to increase both performance and reliability.

In retrospect, although its invention was motivated by performance, dependability was the key reason for the widespread popularity of RAID. The parallel revolution has resurfaced the original performance side of the argument for RAID. The rest of this section surveys the options for dependability and their impacts on cost and performance.

How much redundancy do you need? Do you need extra information to find the faults? Does it matter how you organize the data and the additional check information on these disks? The paper that coined the term gave an evolutionary answer to these questions, starting with the simplest but most expensive solution. Figure e5.11.1 shows the evolution and example cost in the number of extra check disks. To keep track of the evolution, the authors numbered the stages of RAID, and they are still used today.

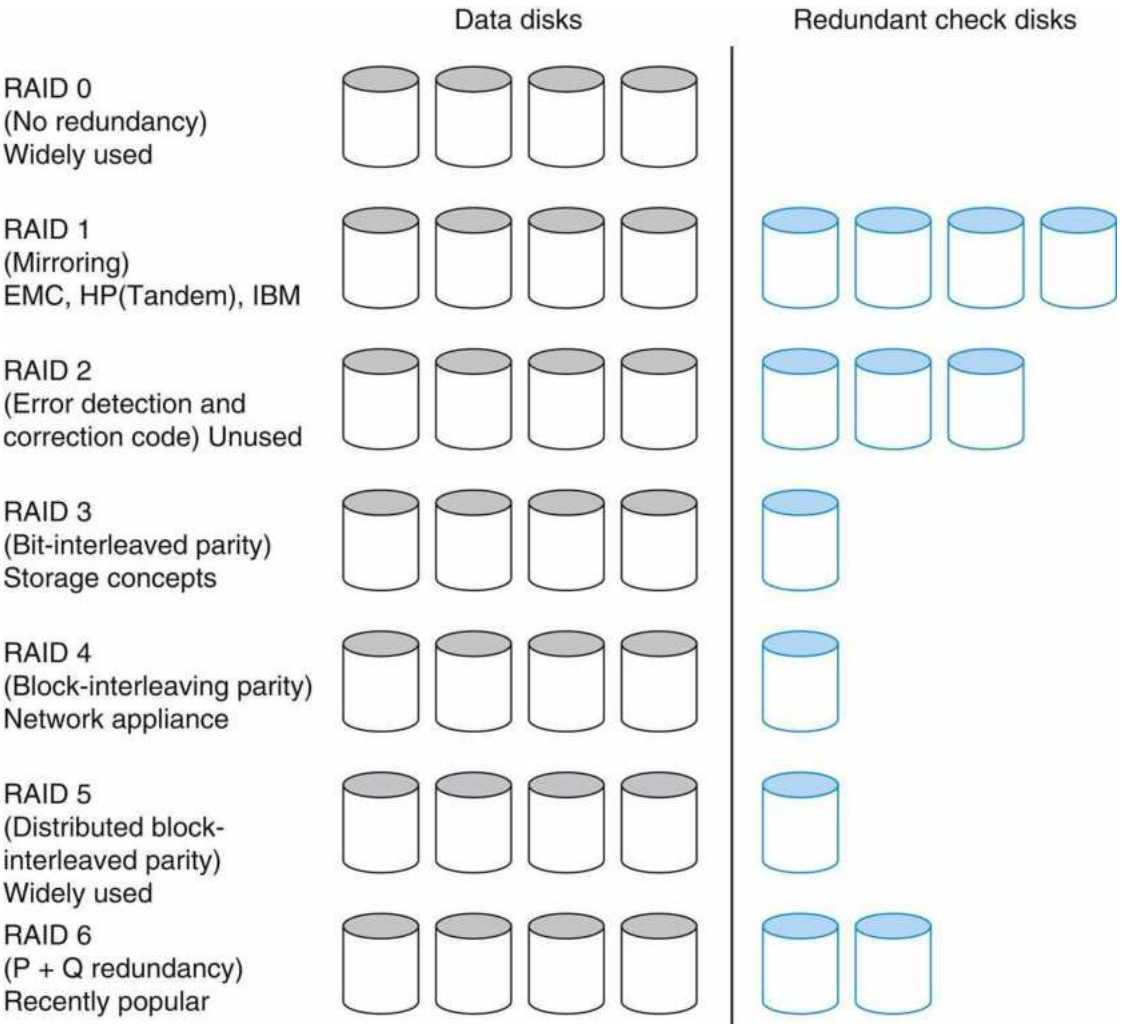


FIGURE E5.11.1 RAID for an example of four data disks showing extra check disks per RAID level and companies that use each level.
Figures e5.11.2 and e5.11.3 explain the difference between RAID 3, RAID 4, and RAID 5.

No Redundancy (RAID 0)

Simply spreading data over multiple disks, called **striping**, automatically forces accesses to several disks. Striping across a set of disks makes the collection appear to software as a single large disk, which simplifies storage management. It also improves performance for large accesses, since many disks can operate at once. Video-editing systems, for example, frequently stripe their data and may not worry about dependability as much as, say, databases.

striping

Allocation of logically sequential blocks to separate disks to allow higher performance than a single disk can deliver.

RAID 0 is something of a misnomer, as there is no redundancy. However, RAID levels are often left to the operator to set when creating a storage system, and RAID 0 is often listed as one of the options. Hence, the term RAID 0 has become widely used.

Mirroring (RAID 1)

This traditional scheme for tolerating disk failure, called **mirroring** or *shadowing*, uses twice as many disks as does RAID 0. Whenever data are written to one disk, that data are also written to a redundant disk, so that there are always two copies of the information. If a disk fails, the system just goes to the "mirror" and reads its contents to get the desired information. Mirroring is the most expensive RAID solution, since it requires the most disks.

mirroring

Writing identical data to multiple disks to increase data availability.

Error Detecting and Correcting Code (RAID 2)

RAID 2 borrows an error detection and correction scheme most often used for memories (see Section 5.5). Since RAID 2 has fallen into disuse, we'll not describe it here.

Bit-Interleaved Parity (RAID 3)

The cost of higher availability can be reduced to $1/n$, where n is the number of disks in a **protection group**. Rather than have a complete copy of the original data for each disk, we need only add enough redundant information to restore the lost information on a failure. Reads or writes go to all disks in the group, with one extra disk to hold the check information in case there is a failure. RAID 3

is popular in applications with large data sets, such as multimedia and some scientific codes.

protection group
The group of data disks or blocks that share a common check disk or block.

Parity is one such scheme. Readers unfamiliar with parity can think of the redundant disk as having the sum of all the data in the other disks. When a disk fails, then you subtract all the data in the good disks from the parity disk; the remaining information must be the missing information. Parity is simply the sum modulo two.

Unlike RAID 1, many disks must be read to determine the missing data. The assumption behind this technique is that taking longer to recover from failure but spending less on redundant storage is a good tradeoff.

Block-Interleaved Parity (RAID 4)

RAID 4 uses the same ratio of data disks and check disks as RAID 3, but they access data differently. The parity is stored as blocks and associated with a set of data blocks.

In RAID 3, every access went to all disks. However, some applications prefer smaller accesses, allowing independent accesses to occur in parallel. That is the purpose of the RAID levels 4 to 7. Since error detection information in each sector is checked on reads to see if the data are correct, such "small reads" to each disk can occur independently as long as the minimum access is one sector. In the RAID context, a small access goes to just one disk in a protection group while a large access goes to all the disks in a protection group.

Writes are another matter. It would seem that each small write would demand that all other disks be accessed to read the rest of the information needed to recalculate the new parity, as in the left in Figure e5.11.2. A "small write" would require reading the old data and old parity, adding the new information, and then writing the new parity to the parity disk and the new data to the data disk.

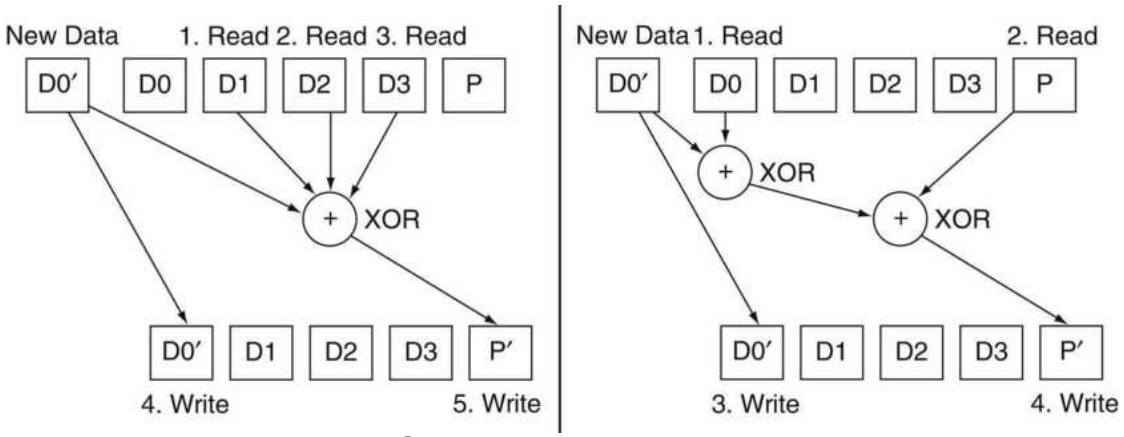


FIGURE E5.11.2 Small write update on RAID 4. This optimization for small writes reduces the number of disk accesses as well as the number of disks occupied. This figure assumes we have four blocks of data and one block of parity. The naive RAID 4 parity calculation in the left of the figure reads blocks D1, D2, and D3 before adding block D0? to calculate the new parity P? (In case you were wondering, the new data D0? comes directly from the CPU, so disks are not involved in reading it.) The RAID 4 shortcut on the right reads the old value D0 and compares it to the new value D0? to see which bits will change. You next read the old parity P and then change the corresponding bits to form P? The logical function exclusive OR does exactly what we want. This example replaces three disk reads (D1, D2, D3) and two disk writes (D0?, P?) involving all the disks for two disk reads (D0, P) and two disk writes (D0?, P?), which involve just two disks. Enlarging the size of the parity group increases the savings of the shortcut. RAID 5 uses the same shortcut.

The key insight to reduce this overhead is that parity is simply a sum of information; by watching which bits change when we write the new information, we need only change the corresponding bits on the parity disk. The right of Figure e5.11.2 shows the shortcut. We must read the old data from the disk being written, compare old data to the new data to see which bits change, read the old parity, change the corresponding bits, and then write the new data and new parity. Thus, the small write involves four disk accesses to two disks instead of accessing all disks. This organization is RAID 4.

Distributed Block-Interleaved Parity (RAID 5)

RAID 4 efficiently supports a mixture of large reads, large writes, and small reads, plus it allows small writes. One drawback to the system is that the parity disk must be updated on every write, so the parity disk is the bottleneck for back-to-back writes.

To fix the parity-write bottleneck, the parity information can be spread throughout all the disks so that there is no single bottleneck for writes. The distributed parity organization is RAID 5.

Figure e5.11.3 shows how data are distributed in RAID 4 versus RAID 5. As the organization on the right shows, in RAID 5 the parity associated with each row of data blocks is no longer restricted to a single disk. This organization allows multiple writes to occur simultaneously as long as the parity blocks are not located on the same disk. For example, a write to block 8 on the right must also access its parity block P2, thereby occupying the first and third disks. A second write to block 5 on the right, implying an update to its parity block P1, accesses the second and fourth disks and thus could occur concurrently with the write to block 8. Those same writes to the organization on the left result in changes to blocks P1 and P2, both on the fifth disk, which is a bottleneck.

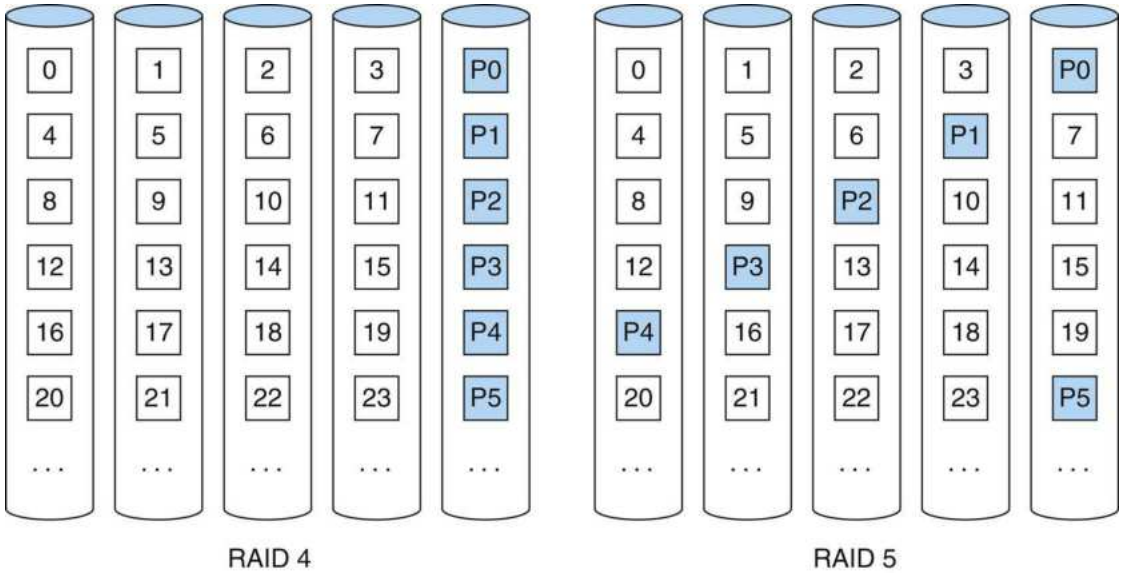


FIGURE E5.11.3 Block-interleaved parity (RAID 4) versus distributed block-interleaved parity (RAID 5).

By distributing parity blocks to all disks, some small writes can be performed in parallel.

P+Q Redundancy (RAID 6)

Parity-based schemes protect against a single self-identifying failure. When a single failure correction is not sufficient, parity can be generalized to have a second calculation over the data and another check disk of information. This second check block allows recovery from a second failure. Thus, the storage overhead is twice that of RAID 5. The small write shortcut of Figure e5.11.2 works as well, except now there are six disk accesses instead of four to update both P and Q information.

RAID Summary

RAID 1 and RAID 5 are widely used in servers; one estimate is that 80% of disks in servers are found in a RAID organization.

One weakness of the RAID systems is repair. First, to avoid making the data unavailable during repair, the array must be designed to allow the failed disks to be replaced without having to turn off the system. RAIDs have enough redundancy to allow continuous operation, but **hot-swapping** disks place demands on the physical and electrical design of the array and the disk interfaces. Second, another failure could occur during repair, so the repair time affects the chances of losing data: the longer the repair time, the greater the chances of another failure that will lose data. Rather than having to wait for the operator to bring in a good disk, some systems include **standby spares** so that the data can be reconstructed instantly upon discovery of the failure. The operator can then replace the failed disks in a more leisurely fashion. Note that a human operator ultimately determines which disks to remove. Operators are only human, so they occasionally remove the good disk instead of the broken disk, leading to an unrecoverable disk failure.

hot-swapping

Replacing a hardware component while the system is running.

standby spares

Reserve hardware resources that can immediately take the place of a failed component.

In addition to designing the RAID system for repair, there are questions about how disk technology changes over time. Although disk manufacturers quote very high MTTF for their products, those numbers are under nominal conditions. If a particular disk array has been subject to temperature cycles due to, say, the failure of the air-conditioning system, or to shaking due to a poor rack design, construction, or installation, the failure rates can be three to six times higher (see the fallacy on page 470). The calculation of RAID reliability assumes independence between disk failures, but disk failures could be correlated, because such damage due to the environment would likely happen to all the disks in the array. Another concern is that since disk bandwidth is growing more slowly than disk capacity, the time to repair a disk in a RAID system is increasing, which in turn enhances the chances of a second failure. For example, a 3-TB disk could take almost 9 hours to read sequentially, assuming no interference. Given that the damaged RAID is likely to continue to serve data, reconstruction could be stretched considerably. Besides increasing that time, another concern is that reading much more data during reconstruction means increasing the chance of an uncorrectable read media failure, which would result in data loss. Other arguments for concern about simultaneous multiple failures are the increasing number of disks in arrays and the use of higher-capacity disks.

Hence, these trends have led to a growing interest in protecting against more than one failure, and so RAID 6 is increasingly being offered as an option and being used in the field.

Check Yourself

Which of the following are true about RAID levels 1, 3, 4, 5, and 6?

1. RAID systems rely on redundancy to achieve high availability.
2. RAID 1 (mirroring) has the highest check disk overhead.
3. For small writes, RAID 3 (bit-interleaved parity) has the worst throughput.
4. For large writes, RAID 3, 4, and 5 have the same throughput.

Elaboration

One issue is how mirroring interacts with striping. Suppose you had, say, four disks' worth of data to store and eight physical disks to use. Would you create four pairs of disks—each organized as RAID 1—and then stripe data across the four RAID 1 pairs? Alternatively, would you create two sets of four disks—each organized as RAID 0—and then mirror writes to both RAID 0 sets? The RAID terminology has evolved to call the former RAID 1+0 or RAID 10 ("striped mirrors") and the latter RAID 0+1 or RAID 01 ("mirrored stripes").

Advanced Material: Implementing Cache Controllers

This online section shows how to implement control for a cache, just as we implemented control for the single-cycle and pipelined datapaths in Chapter 4. This section starts with a description of finite-state machines and the implementation of a cache controller for a simple data cache, including a description of the cache controller in a hardware description language. It then goes into details of an example cache coherence protocol and the difficulties in implementing such a protocol.

5.12 Advanced Material: Implementing Cache Controllers

The section starts with the SystemVerilog of the cache controller from Section 5.9 in eight figures. It then goes into details of an example cache coherency protocol and the difficulties in

implementing such a protocol.

SystemVerilog of a Simple Cache Controller

The hardware description language we are using in this section is SystemVerilog. The biggest change from prior versions of Verilog is that it borrows structures from C to make the code easier to read. Figures e5.12.1 through e5.12.8 show the SystemVerilog description of the cache controller.

```
package cache_def;
  // data structures for cache tag & data

  parameter int TAGMSB = 31;     //tag msb
  parameter int TAGLSB = 14;     //tag lsb

  //data structure for cache tag
  typedef struct packed {
    bit    valid;                //valid bit
    bit    dirty;                //dirty bit
    bit [TAGMSB:TAGLSB]tag;      //tag bits
  }cache_tag_type;

  //data structure for cache memory request
  typedef struct {
    bit [9:0]index;              //10-bit index
    bit    we;                   //write enable
  }cache_req_type;

  //128-bit cache line data
  typedef bit [127:0]cache_data_type;
```

FIGURE E5.12.1 Type declarations in SystemVerilog for the cache tags and data.
The tag field is 18 bits wide and the index field is 10 bits wide, while a 2-bit field (bits 3–2) is used to index the block and select the word from the block. The rest of the type declaration is found in the following figure.

```
  // data structures for CPU<->Cache controller interface

  // CPU request (CPU->cache controller)
  typedef struct {
    bit [31:0]addr;             //32-bit request addr
    bit [31:0]data;             //32-bit request data (used when write)
    bit rw;                     //request type : 0 = read, 1 = write
    bit valid;                  //request is valid
  }cpu_req_type;

  // Cache result (cache controller->cpu)
  typedef struct {
    bit [31:0]data;             //32-bit data
    bit ready;                  //result is ready
  }cpu_result_type;

  //-----------------------------------------------------------------------
  // data structures for cache controller<->memory interface

  // memory request (cache controller->memory)
  typedef struct {
    bit [31:0]addr;             //request byte addr
    bit [127:0]data;            //128-bit request data (used when write)
    bit rw;                     //request type : 0 = read, 1 = write
    bit valid;                  //request is valid
  }mem_req_type;

  // memory controller response (memory -> cache controller)
  typedef struct {
    cache_data_type data;       //128-bit read back data
    bit    ready;               //data is ready
  }mem_data_type;

endpackage
```

FIGURE E5.12.2 Type declarations in SystemVerilog for the CPU-cache and cache-memory interfaces. These are nearly identical except that the data are 32 bits wide between the CPU and cache and are 128 bits wide between the cache and memory.

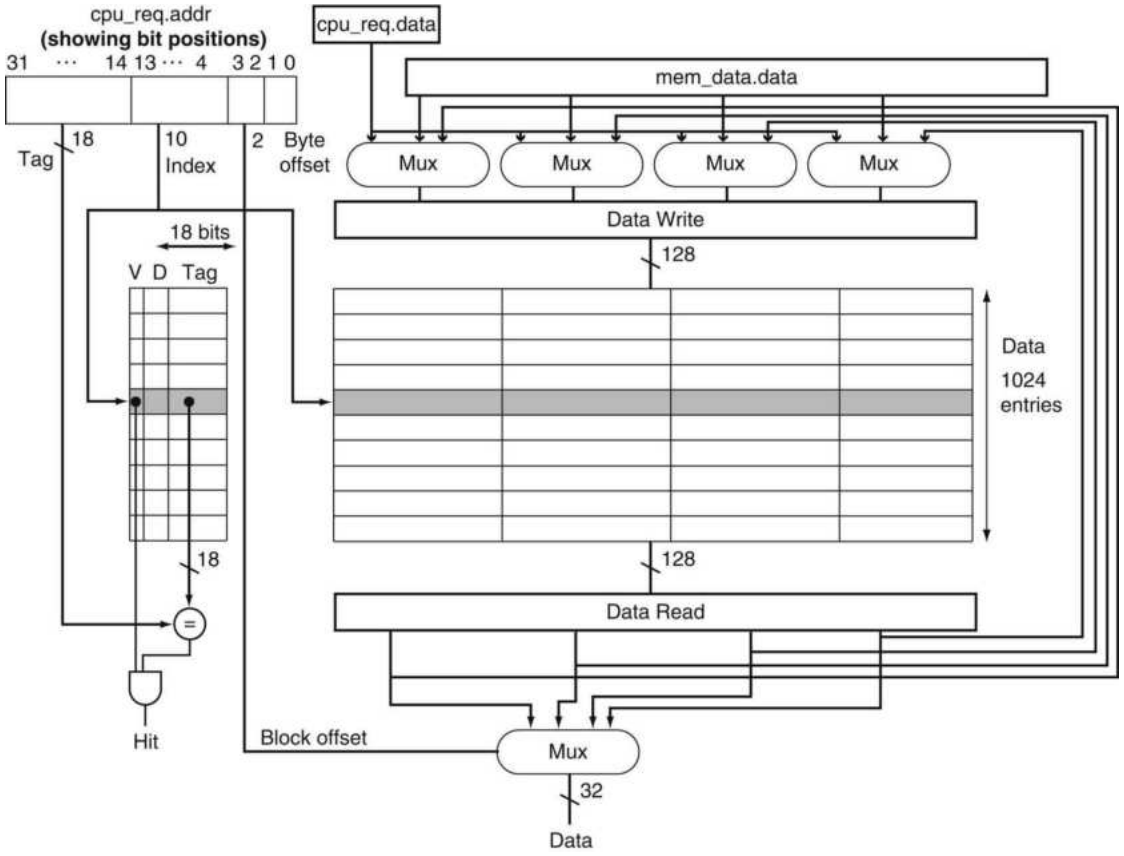


FIGURE E5.12.3 Block diagram of the simple cache using the Verilog names.

Not shown are the write enables for the cache tag memory and for the cache data memory, or the control signals for multiplexors that supply data for the Data Write variable. Rather than have separate write enables on every word of the cache data block, the Verilog reads the old value of the block into Data Write and then updates the word in that variable on a write. It then writes the whole 128-bit block.

```
/*cache: data memory, single port, 1024 blocks*/
module dm_cache_data(input  bit clk,
    input  cache_req_type  data_req,//data request/command, e.g. RW, valid
    input  cache_data_type data_write, //write port (128-bit line)
    output cache_data_type data_read); //read port
  timeunit 1ns; timeprecision 1ps;

  cache_data_typedata_mem[0:1023];

  initial begin
    for (int i=0; i<1024; i++)
        data_mem[i] = '0;
  end

  assign data_read = data_mem[data_req.index];

  always_ff @(posedge(clk)) begin
    if (data_req.we)
      data_mem[data_req.index] <= data_write;
  end
endmodule

/*cache: tag memory, single port, 1024 blocks*/
module dm_cache_tag(input  bit clk, //write clock
    input  cache_req_type tag_req, //tag request/command, e.g. RW, valid
    input  cache_tag_type tag_write,//write port
    output cache_tag_type tag_read);//read port
  timeunit 1ns; timeprecision 1ps;

  cache_tag_typetag_mem[0:1023];

  initial begin
     for (int i=0; i<1024; i++)
     tag_mem[i] = '0;
  end

  assign tag_read = tag_mem[tag_req.index];

  always_ff @(posedge(clk)) begin
    if (tag_req.we)
      tag_mem[tag_req.index] <= tag_write;
  end

endmodule
```

FIGURE E5.12.4 Cache data and tag modules in SystemVerilog.
These are nearly identical except that the data are 32 bits wide between the CPU and cache and are 128 bits wide between the cache and memory. Both only write on positive clock edges if the write enable is set.

```
/*cache finite state machine*/

module dm_cache_fsm(input  bit clk, input  bit rst,
        input  cpu_req_type   cpu_req,     //CPU request input (CPU->cache)
        input  mem_data_type  mem_data,    //memory response (memory->cache)
        output mem_req_type   mem_req,     //memory request (cache->memory)
        output cpu_result_type cpu_res     //cache result (cache->CPU)
    );

  timeunit 1ns;
  timeprecision 1ps;

  /*write clock*/
  typedef enum {idle, compare_tag, allocate, write_back} cache_state_type;

  /*FSM state register*/
  cache_state_typevstate, rstate;

  /*interface signals to tag memory*/
  cache_tag_typetag_read;                  //tag read result
  cache_tag_typetag_write;                 //tag write data
  cache_req_typetag_req;                   //tag request

  /*interface signals to cache data memory*/
  cache_data_typedata_read;                //cache line read data
  cache_data_typedata_write;               //cache line write data
  cache_req_typedata_req;                  //data req


  /*temporary variable for cache controller result*/
  cpu_result_typev_cpu_res;

  /*temporary variable for memory controller request*/
  mem_req_typev_mem_req;

  assign mem_req = v_mem_req;              //connect to output ports
  assign cpu_res = v_cpu_res;
```

FIGURE E5.12.5 FSM in SystemVerilog, part I.
These modules instantiate the memories according to the type definitions in the previous figure.

```
always_comb begin

    /*-------------------------default values for all signals------------*/
    /*no state change by default*/
    vstate = rstate;
    v_cpu_res = '{0, 0}; tag_write = '{0, 0, 0};

    /*read tag by default*/
    tag_req.we = '0;
    /*direct map index for tag*/
    tag_req.index = cpu_req.addr[13:4];

    /*read current cache line by default*/
    data_req.we = '0;
    /*direct map index for cache data*/
    data_req.index = cpu_req.addr[13:4];

    /*modify correct word (32-bit) based on address*/
    data_write = data_read;
    case(cpu_req.addr[3:2])
    2'b00:data_write[31:0] = cpu_req.data;
    2'b01:data_write[63:32] = cpu_req.data;
    2'b10:data_write[95:64] = cpu_req.data;
    2'b11:data_write[127:96] = cpu_req.data;
    endcase

    /*read out correct word(32-bit) from cache (to CPU)*/
    case(cpu_req.addr[3:2])
    2'b00:v_cpu_res.data = data_read[31:0];
    2'b01:v_cpu_res.data = data_read[63:32];
    2'b10:v_cpu_res.data = data_read[95:64];
    2'b11:v_cpu_res.data = data_read[127:96];
    endcase

    /*memory request address (sampled from CPU request)*/
    v_mem_req.addr = cpu_req.addr;
    /*memory request data (used in write)*/
    v_mem_req.data = data_read;
    v_mem_req.rw = '0;
```

FIGURE E5.12.6 FSM in SystemVerilog, part II. This section describes the default value of all signals. The following figures will set these values for one clock cycle, and this Verilog will reset it to these values for the following clock cycle.

```
//-------------------------------------Cache FSM--------------------------
case(rstate)
/*idle state*/
idle : begin
    /*If there is a CPU request, then compare cache tag*/
    if (cpu_req.valid)
       vstate = compare_tag;
       end
/*compare_tag state*/
compare_tag : begin
          /*cache hit (tag match and cache entry is valid)*/
    if (cpu_req.addr[TAGMSB:TAGLSB] == tag_read.tag && tag_read.valid) begin
          v_cpu_res.ready = '1;

       /*write hit*/
       if (cpu_req.rw) begin
          /*read/modify cache line*/
          tag_req.we = '1; data_req.we = '1;

          /*no change in tag*/
          tag_write.tag = tag_read.tag;
          tag_write.valid = '1;
          /*cache line is dirty*/
          tag_write.dirty = '1;
       end

          /*xaction is finished*/
          vstate = idle;
    end
    /*cache miss*/
    else begin
      /*generate new tag*/
      tag_req.we = '1;
      tag_write.valid = '1;
      /*new tag*/
      tag_write.tag = cpu_req.addr[TAGMSB:TAGLSB];
      /*cache line is dirty if write*/
      tag_write.dirty = cpu_req.rw;

      /*generate memory request on miss*/
      v_mem_req.valid = '1;
      /*compulsory miss or miss with clean block*/
      if (tag_read.valid == 1'b0 || tag_read.dirty == 1'b0)
         /*wait till a new block is allocated*/
         vstate = allocate;
```

FIGURE E5.12.7 FSM in SystemVerilog, part III. Actual FSM states via case statement in this figure and the next. This figure has the Idle state and most of the Compare Tag state.

```
      else begin
      /*miss with dirty line*/
         /*write back address*/
         v_mem_req.addr = {tag_read.tag, cpu_req.addr[TAGLSB-1:0]};
         v_mem_req.rw = '1;
         /*wait till write is completed*/
         vstate = write_back;
      end
     end
 end
 /*wait for allocating a new cache line*/
 allocate: begin
     /*memory controller has responded*/
     if (mem_data.ready) begin
        /*re-compare tag for write miss (need modify correct word)*/
        vstate = compare_tag;
        data_write = mem_data.data;
        /*update cache line data*/
        data_req.we = '1;
     end
      end
 /*wait for writing back dirty cache line*/
 write_back : begin
     /*write back is completed*/
     if (mem_data.ready) begin
        /*issue new memory request (allocating a new line)*/
        v_mem_req.valid = '1;
        v_mem_req.rw = '0;

        vstate = allocate;
     end
   end
 endcase
end

always_ff @(posedge(clk)) begin
 if (rst)
   rstate <= idle;        //reset to idle state
 else
   rstate <= vstate;
end
/*connect cache tag/data memory*/
dm_cache_tag  ctag(.*);
dm_cache_data cdata(.*);
endmodule
```

FIGURE E5.12.8 FSM in SystemVerilog, part IV. Actual FSM states via the case statement in the prior figure and this one. This figure has the last part of the Compare Tag state, plus Allocate and Write-Back states.

Figures e5.12.1 and e5.12.2 declare the structures that are used in the definition of the cache in the following figures. For example, the cache tag structure (`cache_tag_type`) contains a valid bit (`valid`), a dirty bit (`dirty`), and an 18-bit tag field ([`TAGMSB:TAGLSB] tag`). Figure e5.12.3 shows the block diagram of the cache using the names from the Verilog description.

Figure e5.12.4 defines modules for the cache data (`dm_cache_data`) and cache tag (`dm_cache_tag`). These memories can be read at any time, but writes only occur on the positive clock edge (`posedge(clk)`) and only if write enable is a 1 (`data_req.we` or `tag_req.we`).

Figure e5.12.5 defines the inputs, outputs, and states of the FSM. The inputs are the requests from the CPU (`cpu_req`) and responses from memory (`mem_data`), and the outputs are responses to the CPU (`cpu_res`) and requests to memory (`mem_req`). The figure also declares the internal variables needed by the FSM. For example, the current state and next state registers of the FSM are `rstate` and `vstate`, respectively.

Figure e5.12.6 lists the default values of the control signals, including the word to be read or written from a block, setting the cache write enables to 0, and so on. These values are set every clock cycle, so the write enable for a portion of the cache—for example, `tag_req.we`—would be set to 1 for one clock cycle in the figures below and then would be reset to 0 according to the Verilog in this figure.

The last two figures show the FSM as a large case statement (`case(rstate)`), with the four states split across the two figures. Figure e5.12.7 starts with the Idle state (`idle`), which simply goes to the Compare Tag state (`compare_tag`) if the CPU makes a valid request. It then describes most of the Compare Tag state. The Compare Tag state checks to see if the tags match and the entry is valid. If so, then it first sets the Cache Ready signal (`v_cpu_res.ready`). If the request is a write, it sets the tag field, the valid bit, and the dirty bit. The next state is Idle. If it is a miss, then the state prepares to change the tag entry and valid and dirty bits. If the block to be replaced is clean or invalid, the next state is Allocate.

Figure e5.12.8 continues the Compare Tag state. If the block to be replaced is dirty, then the next state is Write-Back. The figure shows

the Allocate state (`allocate`) next, which simply reads the new block. It keeps looping until the memory is ready; when it is, it goes to the Compare Tag state. This is followed in the figure by the Write-Back state (`write_back`). As the figure shows, the Write-Back state merely writes the dirty block to memory, once again looping until memory is ready. When memory is ready, indicating the write is complete, we go to the Allocate state.

The code at the end sets the current state from the next state or resets the FSM to the Idle state on the next clock edge, depending on a reset signal (`rst`).

The online material includes a Test Case module that will be useful to check the code in these figures. This SystemVerilog could be used to create a cache and cache controller in an FPGA.

Basic Coherent Cache Implementation Techniques

The key to implementing an invalidate protocol is the use of the bus, or another broadcast medium, to perform invalidates. To invalidate, the processor simply acquires bus access and broadcasts the address to be invalidated on the bus. All processors continuously snoop on the bus, watching the addresses. The processors check whether the address on the bus is in their cache. If so, the corresponding data in the cache are invalidated.

When a write to a block that is shared occurs, the writing processor must acquire bus access to broadcast its invalidation. If two processors try to write shared blocks at the same time, their attempts to broadcast an invalidate operation will be serialized when they arbitrate for the bus. The first processor to obtain bus access will cause any other copies of the block it is writing to be invalidated. If the processors were attempting to write the same block, the serialization enforced by the bus also serializes their writes. One implication of this scheme is that a write to a shared data item cannot actually complete until it obtains bus access. All coherence schemes require some method of serializing accesses to the same cache block, by serializing access either to the communication medium or another shared structure.

In addition to invalidating outstanding copies of a cache block that is being written into, we also need to locate a data item when a

cache miss occurs. In a write-through cache, it is easy to find the recent value of a data item, since all written data are unfailingly sent to the memory, from which the most-recent value of a data item can always be fetched. In a design with adequate memory bandwidth to support the write traffic from the processors, using write-through simplifies the implementation of cache coherence.

For a write-back cache, finding the most-recent data value is more difficult, since the most recent value of a data item can be in a cache rather than in memory. Happily, write-back caches can use the same snooping scheme both for cache misses and for writes: each processor snoops all addresses placed on the bus. If a processor finds that it has a dirty copy of the requested cache block, it provides that cache block in response to the read request and causes the memory access to be aborted. The increased complexity comes from having to retrieve the cache block from a processor's cache, which can often take longer than retrieving it from the shared memory if the processors are in separate chips. Since write-back caches generate lower requirements for memory bandwidth, they can support larger numbers of faster processors and have been the approach chosen in most multiprocessors, despite the additional complexity of maintaining coherence. Therefore, we will examine the implementation of coherence with write-back caches.

The normal cache tags can be used to implement the process of snooping, and the valid bit for each block makes invalidation easy to implement. Read misses, whether generated by an invalidation or by some other event, are also straightforward, since they simply rely on the snooping capability. For writes, we'd like to know whether any other copies of the block are cached, because if there are no other cached copies, the write need not be placed on the bus in a write-back cache. Not sending the write reduces both the time taken by the write and the required bandwidth.

To track whether or not a cache block is shared, we can add an extra state bit associated with each cache block, just as we have a valid bit and a dirty bit. By adding a bit indicating whether the block is shared, we can decide whether a write must generate an invalidate. When a write to a block in the shared state occurs, the cache generates an invalidation on the bus and marks the block as *exclusive*. No further invalidations will be sent by that processor for that block. The processor with the sole copy of a cache block is

normally called the *owner* of the cache block.

When an invalidation is sent, the state of the owner's cache block is changed from shared to unshared (or exclusive). If another processor later requests this cache block, the state must be made shared again. Since our snooping cache also sees any misses, it knows when the exclusive cache block has been requested by another processor, and the state should be made shared.

Every bus transaction must check the cache-address tags, which could potentially interfere with processor cache accesses. One way to reduce this interference is to duplicate the tags. The interference can also be reduced in a multilevel cache by directing the snoop requests to the L2 cache, which the processor uses only when it has a miss in the L1 cache. For this scheme to work, every entry in the L1 cache must be present in the L2 cache, a property called the *inclusion property*. If the snoop gets a hit in the L2 cache, then it must arbitrate for the L1 cache to update the state and possibly retrieve the data, which usually requires a stall of the processor. Sometimes it may even be useful to duplicate the tags of the secondary cache to further decrease contention between the processor and the snooping activity.

An Example Cache Coherency Protocol

A snooping coherence protocol is usually implemented by incorporating a finite-state controller in each node. This controller responds to requests from the processor and from the bus (or other broadcast medium), changing the state of the selected cache block, as well as using the bus to access data or to invalidate it. Logically, you can think of a separate controller being associated with each block; that is, snooping operations or cache requests for different blocks can proceed independently. In actual implementations, a single controller allows multiple operations to distinct blocks to proceed in interleaved fashion (that is, one operation may be initiated before another is completed, even though only one cache access or one bus access is allowed at a time). Also, remember that although we refer to a bus in the following description, any interconnection network that supports a broadcast to all the coherence controllers and their associated caches can be used to implement snooping.

The simple protocol we consider has three states: invalid, shared, and modified. The shared state indicates that the block is potentially shared, while the modified state indicates that the block has been updated in the cache; note that the modified state *implies* that the block is exclusive. Figure e5.12.9 shows the requests generated by the processor-cache module in a node (in the first nine rows of the table) as well as those coming from the bus (in the last five rows of the table). This protocol is for a write-back cache, but it can be easily changed to work for a write-through cache by reinterpreting the modified state as an exclusive state and updating the cache on writes in the normal fashion for a write-through cache. The most common extension of this basic protocol is the addition of an exclusive state, which describes a block that is unmodified but held in only one cache; the caption of Figure e5.12.9 describes this state and its addition in more detail.

Request	Source	State of addressed cache block	Type of cache action	Function and explanation
Read hit	processor	shared or modified	normal hit	Read data in cache.
Read miss	processor	invalid	normal miss	Place read miss on bus.
Read miss	processor	shared	replacement	Address conflict miss: place read miss on bus.
Read miss	processor	modified	replacement	Address conflict miss: write-back block, then place read miss on bus.
Write hit	processor	modified	normal hit	Write data in cache.
Write hit	processor	shared	coherence	Place invalidate on bus. These operations are often called upgrade or *ownership* misses, since they do not fetch the data but only change the state.
Write miss	processor	invalid	normal miss	Place write miss on bus.
Write miss	processor	shared	replacement	Address conflict miss: place write miss on bus.
Write miss	processor	modified	replacement	Address conflict miss: write-back block, then place write miss on bus.
Read miss	bus	shared	no action	Allow memory to service read miss.
Read miss	bus	modified	coherence	Attempt to share data: place cache block on bus and change state to shared.
Invalidate	bus	shared	coherence	Attempt to write shared block; invalidate the block.
Write miss	bus	shared	coherence	Attempt to write block that is shared; invalidate the cache block.
Write miss	bus	modified	coherence	Attempt to write block that is exclusive elsewhere: write-back the cache block and make its state invalid.

FIGURE E5.12.9 The cache coherence mechanism receives requests from both the processor and the bus and responds to these based on the type of request, whether it hits or misses in the cache, and the state of the cache block specified in the request.

The fourth column describes the type of cache action as normal hit or miss (the same as a uniprocessor

cache would see), replacement (a uniprocessor cache replacement miss), or coherence (required to maintain cache coherence); a normal or replacement action may cause a coherence action depending on the state of the block in other caches. For read misses, write misses, or invalidates snooped from the bus, an action is required only if the read or write addresses match a block in the cache and the block is valid. Some protocols also introduce a state to designate when a block is exclusively in one cache but has not yet been written. This state can arise if a write access is broken into two pieces: getting the block exclusively in one cache and then subsequently updating it; in such a protocol this "exclusive unmodified state" is transient, ending as soon as the write is completed. Other protocols use and maintain an exclusive state for an unmodified block. In a snooping protocol, this state can be entered when a processor reads a block that is not resident in any other cache. Because all subsequent accesses are snooped, it is possible to maintain the accuracy of this state. In particular, if another processor issues a read miss, the state is changed from exclusive to shared. The advantage of adding this state is that a subsequent write to a block in the exclusive state by the same processor need not acquire bus access or generate an invalidate, since the block is known to be exclusively in this cache; the processor merely changes the state to modified. This state is easily added by using the bit that encodes the coherent state as an exclusive state and using the dirty bit to indicate that a block is modified. The popular MESI protocol, which is named for the four states it includes (modified, exclusive, shared, and invalid), uses this structure. The MOESI protocol introduces another extension: the "owned" state.

When an invalidate or a write miss is placed on the bus, any processors with copies of the cache block invalidate it. For a write-through cache, the data for a write miss can always be retrieved from the memory. For a write miss in a writeback cache, if the block is exclusive in just one cache, that cache also writes back the block; otherwise, the data can be read from memory.

Figure e5.12.10 shows a finite-state transition diagram for a single cache block using a write invalidation protocol and a write-back

cache. For simplicity, the three states of the protocol are duplicated to represent transitions based on processor requests (on the left, which corresponds to the top half of the table in Figure e5.12.9), contrary to transitions based on bus requests (on the right, which corresponds to the last five rows of the table in Figure e5.12.9). Boldface type is used to distinguish the bus actions, in contrast to the conditions on which a state transition depends. The state in each node represents the state of the selected cache block specified by the processor or bus request.

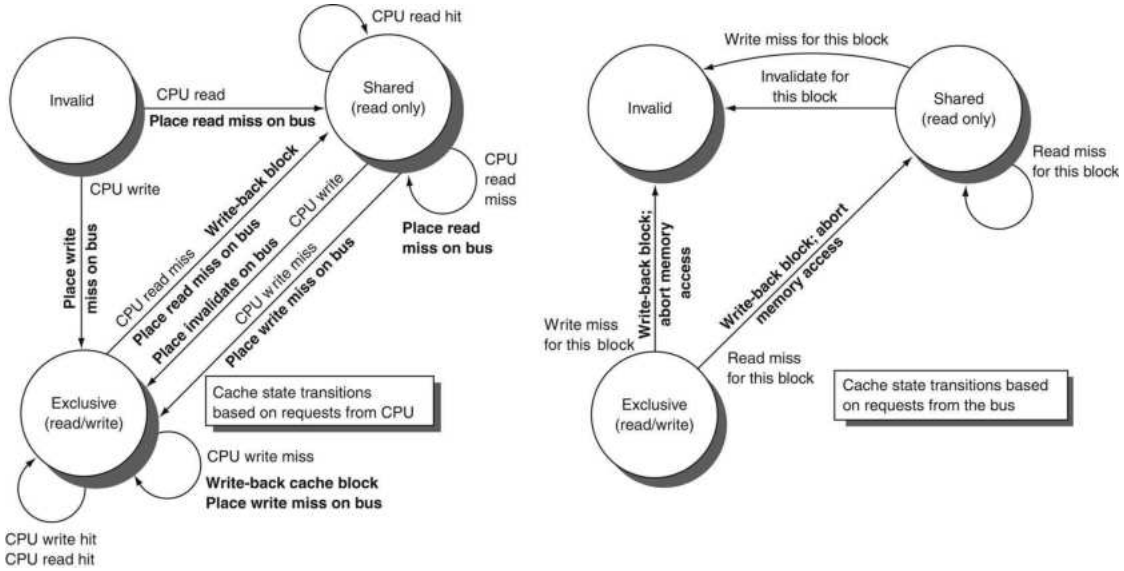


FIGURE E5.12.10 A write-invalidate, cache-coherence protocol for a write-back cache, showing the states and state transitions for each block in the cache.

The cache states are shown in circles, with any access permitted by the processor without a state transition shown in parentheses under the name of the state. The stimulus causing a state change is shown on the transition arcs in regular type, and any bus actions generated as part of the state transition are shown on the transition arc in bold. The stimulus actions apply to a block in the cache, not to a specific address in the cache. Hence, a read miss to a block in the shared state is a miss for that cache block but for a different address. The left side of the diagram shows state transitions based on actions of the processor associated with this cache; the right side shows transitions based on operations on the bus. A read

miss in the exclusive or shared state and a write miss in the exclusive state occur when the address requested by the processor does not match the address in the cache block. Such a miss is a standard cache replacement miss. An attempt to write a block in the shared state generates an invalidate. Whenever a bus transaction occurs, all caches that contain the cache block specified in the bus transaction take the action dictated by the right half of the diagram. The protocol assumes that memory provides data on a read miss for a block that is clean in all caches. In actual implementations, these two sets of state diagrams are combined. In practice, there are many subtle variations on invalidate protocols, including the introduction of the exclusive unmodified state, as to whether a processor or memory provides data on a miss.

All of the states in this cache protocol would be needed in a uniprocessor cache, where they would correspond to the invalid, valid (and clean), and dirty states. Most of the state changes indicated by arcs in the left half of Figure e5.12.10 would be needed in a write-back uniprocessor cache, with the exception being the invalidate on a write hit to a shared block. The state changes represented by the arcs in the right half of Figure e5.12.10 are needed only for coherence and would not appear at all in an uniprocessor cache controller.

As mentioned earlier, there is only one finite-state machine per cache, with stimuli coming either from the attached processor or from the bus. Figure e5.12.11 shows how the state transitions in the right half of Figure e5.12.10 are combined with those in the left half of the figure to form a single state diagram for each cache block.

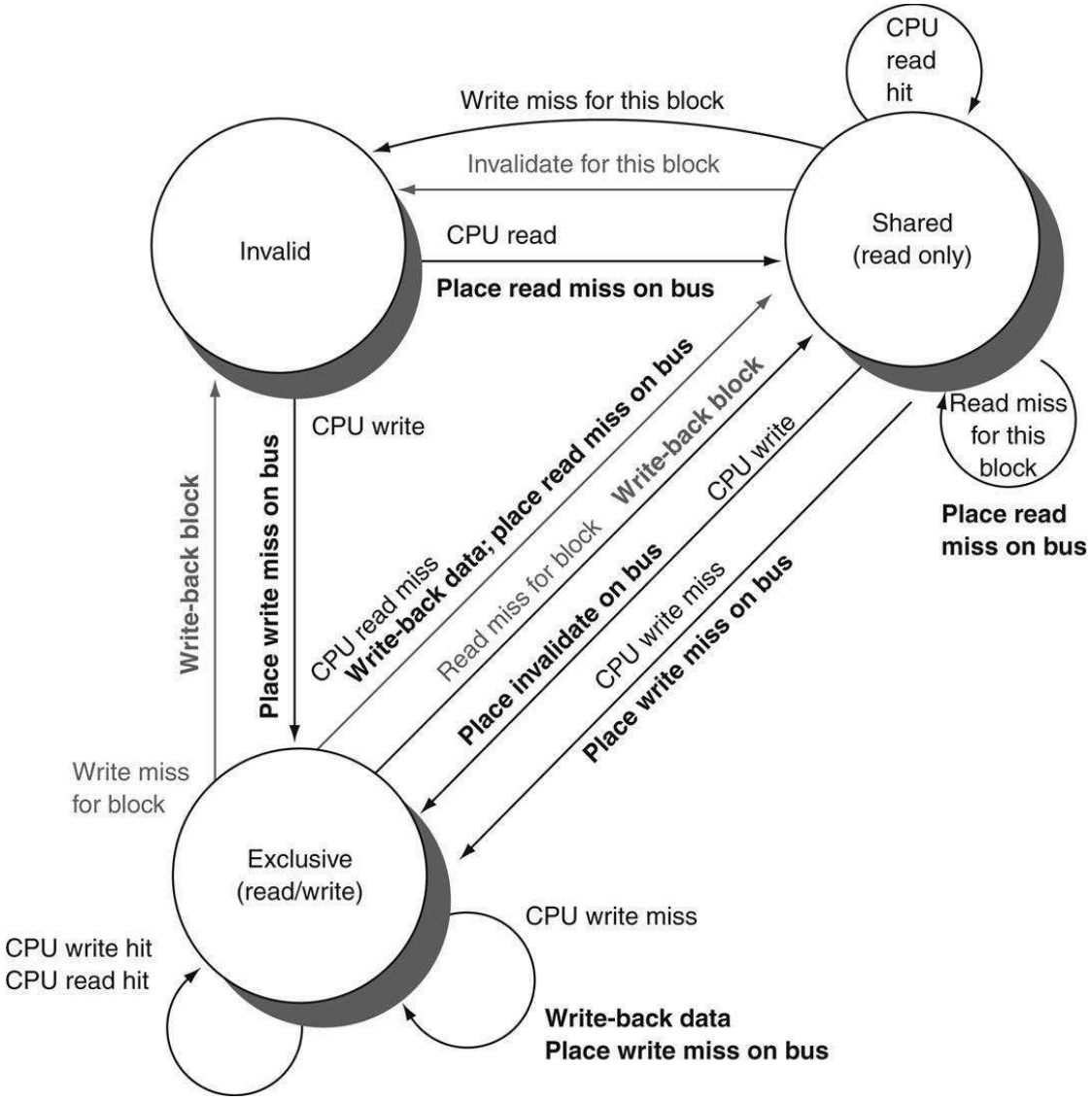


FIGURE E5.12.11 Cache coherence state diagram with the state transitions induced by the local processor shown in black and by the bus activities shown in gray.
As in Figure e5.12.10, the activities on a transition are shown in bold.

To understand why this protocol works, observe that any valid cache block is either in the shared state in one or more caches or in the exclusive state in exactly one cache. Any transition to the exclusive state (which is required for a processor to write to the block) requires an invalidate or write miss to be placed on the bus, causing all caches to make the block invalid. In addition, if some other cache had the block in the exclusive state, that cache generates a write back, which supplies the block containing the desired

address. Finally, if a read miss occurs on the bus to a block in the exclusive state, the cache with the exclusive copy changes its state to shared.

The actions in gray in Figure e5.12.11, which handle read and write misses on the bus, are essentially the snooping component of the protocol. One other property that is preserved in this protocol, and in most other protocols, is that any memory block in the shared state is always up to date in the memory, which simplifies the implementation.

Although our simple cache protocol is correct, it omits a number of complications that make the implementation much trickier. The most important of these is that the protocol assumes that operations are *atomic*—that is, an operation can be done in such a way that no intervening operation can occur. For example, the protocol described assumes that write misses can be detected, acquire the bus, and receive a response as a single atomic action. In reality, this is not true. Similarly, if we used a switch, as all recent multiprocessors do, then even read misses would also not be atomic.

Nonatomic actions introduce the possibility that the protocol can *deadlock*, meaning that it reaches a state where it cannot continue. On the next page, we will discuss how these protocols are implemented without a bus.

Constructing small-scale (two to four processors) multiprocessors has become very easy. For example, the Intel Nehalem and AMD Opteron processors are designed for use in cache-coherent multiprocessors and have an external interface that supports snooping and allows two to four processors to be directly connected. They also have larger on-chip caches to reduce bus utilization. In the case of the Opteron processors, the support for interconnecting multiple processors is integrated onto the processor chip, as are the memory interfaces. In the case of the Intel design, a two-processor system can be built with only a few additional external chips to interface with the memory system and I/O. Although these designs cannot be easily scaled to larger processor counts, they offer an extremely cost-effective solution for two to four processors.

The devil is in the details.

Implementing Snoopy Cache Coherence

As we said earlier, the major complication in actually implementing the snooping coherence protocol we have described is that write and upgrade misses are not atomic in any recent multiprocessor. The steps of detecting a write or upgrade miss; communicating with the other processors and memory; getting the most-recent value for a write miss and ensuring that any invalidates are processed; and updating the cache cannot be done as if they took a single cycle.

In a simple single-bus system, these steps can be made effectively atomic by arbitrating for the bus first (before changing the cache state) and not releasing the bus until all actions are complete. How can the processor know when all the invalidates are complete? In most bus-based multiprocessors, a single line is used to signal when all necessary invalidates have been received and are being processed. Following that signal, the processor that generated the miss can release the bus, knowing that any required actions will be completed before any activity related to the next miss. By holding the bus exclusively during these steps, the processor effectively makes the individual steps atomic.

In a system without a bus, we must find some other method of making the steps in a miss atomic. In particular, we must ensure that two processors that attempt to write the same block at the same time, a situation which is called a *race*, are strictly ordered: one write is processed before the next is begun. It does not matter which of two writes in a race wins the race, just that there be only a single winner whose coherence actions are completed first. In a snoopy system, ensuring that a race has only one winner is accomplished by using broadcast for all misses, as well as some basic properties of the interconnection network. These properties, together with the ability to restart the miss handling of the loser in a race, are the keys to implementing snoopy cache coherence without a bus.

5.13 Real Stuff: The ARM Cortex-A53 and Intel Core i7 Memory Hierarchies

In this section, we will look at the memory hierarchy of the same two microprocessors described in Chapter 4: the ARM Cortex-A53 and Intel Core i7. This section is in part based on Section 2.6 of *Computer Architecture: A Quantitative Approach*, 5th edition.

Figure 5.42 summarizes the address sizes and TLBs of the two processors. Note that the Cortex-A53 has two 10-entry fully associative micro-TLBs backed by a shared 512-entry four-way set associative main TLB with a 48-bit virtual address space and a 40-bit physical address space. The Core i7 has three TLBs with a 48-bit virtual address and a 44-bit physical address. Although the 64-bit registers of these processors could hold a larger virtual address, there was no software need for such a large space, and 48-bit virtual addresses shrinks both the page table memory footprint and the TLB hardware.

Characteristic	ARM Cortex-A53	Intel Core i7
Virtual address	48 bits	48 bits
Physical address	40 bits	44 bits
Page size	Variable: 4, 16, 64 KiB, 1, 2 MiB, 1 GiB	Variable: 4 KiB, 2/4 MiB
TLB organization	1 TLB for instructions and 1 TLB for data per core Both micro TLBs are fully associative, with 10 entries, round robin replacement 64-entry, four-way set-associative TLBs TLB misses handled in hardware	1 TLB for instructions and 1 TLB for data per core Both L1 TLBs are four-way set associative, LRU replacement L1 I-TLB has 128 entries for small pages, seven per thread for large pages L1 D-TLB has 64 entries for small pages, 32 for large pages The L2 TLB is four-way set associative, LRU replacement The L2 TLB has 512 entries TLB misses handled in hardware

FIGURE 5.42 Address translation and TLB hardware for the ARM Cortex-A53 and Intel Core i7 920.
Both processors provide support for large pages, which are used for things like the operating system or mapping a frame buffer. The large-page scheme avoids using a large number of entries to map a single object that is always present.

Figure 5.43 shows their caches. The Cortex-A53 has between one and four processors or cores while the Core i7 is fixed at four. Cortex-A53 has a 16 to 64 KiB, two-way L1 instruction cache (per core) and the Core i7 has a 32 KiB, four-way set associative, L1 instruction cache (per core). Both use 64 byte blocks. The Cortex-A53 increases the associativity to four-way for the data cache, other variables remain the same. Similarly, the Core i7 keeps everything the same except the associativity, which it increases to eight-way. The Core i7 provides a 256 KiB, eight-way set associative unified L2 cache (per core) with 64 byte blocks. In contrast, the Cortex-A53 provides a L2 cache that is shared between one and four cores. This cache is 16-way set associative with 64 byte blocks and between 128 KiB and 2 MiB in size. As the Core i7 is used for servers, it also offers an L3 cache shared by all the cores on the chip. Its size varies

depending on the number of cores. With four cores, as in this case, the size is 8 MiB.

Characteristic	ARM Cortex-A53	Intel Core i7
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	Configurable 16 to 64 KiB each for instructions/data	32 KiB each for instructions/data per core
L1 cache associativity	Two-way (I), four-way (D) set associative	Four-way (I), eight-way (D) set associative
L1 replacement	Random	Approximated LRU
L1 block size	64 bytes	64 bytes
L1 write policy	Write-back, variable allocation policies (default is Write-allocate)	Write-back, No-write-allocate
L1 hit time (load-use)	Two clock cycles	Four clock cycles, pipelined
L2 cache organization	Unified (instruction and data)	Unified (instruction and data) per core
L2 cache size	128 KiB to 2 MiB	256 KiB (0.25 MiB)
L2 cache associativity	16-way set associative	8-way set associative
L2 replacement	Approximated LRU	Approximated LRU
L2 block size	64 bytes	64 bytes
L2 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L2 hit time	12 clock cycles	10 clock cycles
L3 cache organization	–	Unified (instruction and data)
L3 cache size	–	8 MiB, shared
L3 cache associativity	–	16-way set associative
L3 replacement	–	Approximated LRU
L3 block size	–	64 bytes
L3 write policy	–	Write-back, Write-allocate
L3 hit time	–	35 clock cycles

FIGURE 5.43 Caches in the ARM Cortex-A53 and Intel Core i7 920.

A significant challenge facing cache designers is to support processors like the Cortex-A53 and the Core i7 that can execute more than one memory instruction per clock cycle. A popular technique is to break the cache into banks and allow multiple, independent, **parallel** accesses, provided the accesses are to different banks. The technique is similar to interleaved DRAM

banks (see Section 5.2).

The Cortex-A53 and the Core i7 have additional optimizations that allow them to reduce the miss penalty. The first of these is the return of the requested word first on a miss. They also continue to execute instructions that access the data cache during a cache miss. Designers who are attempting to hide the cache miss latency commonly use this technique, called a **nonblocking cache.** They implement two flavors of nonblocking. *Hit under miss* allows additional cache hits during a miss, while *miss under miss* allows multiple outstanding cache misses. The aim of the first of these two is hiding some miss latency with other work, while the aim of the second is overlapping the latency of two different misses.

nonblocking cache

A cache that allows the processor to make references to the cache while the cache is handling an earlier miss.

Overlapping a large fraction of miss times for multiple outstanding misses requires a high-bandwidth memory system capable of handling multiple misses in parallel. In a personal mobile device, the memory system below it can often pipeline, merge, reorder, or prioritize requests appropriately. Large servers

and multiprocessors typically have memory systems capable of handling several outstanding misses in parallel.

The Cortex-A53 and the Core i7 have prefetch mechanisms for data accesses. They look at a pattern of data misses and uses this information to try to predict the next address to start fetching the data before the miss occurs. Such techniques generally work best when accessing arrays in loops.

The sophisticated memory hierarchies of these chips and the large fraction of the dies dedicated to caches and TLBs show the significant design effort expended to try to close the gap between processor cycle times and memory latency.

Performance of the Cortex-A53 and Core i7 Memory Hierarchies

The memory hierarchy of the Cortex-A53 was measured using a 32 KiB two-way set associative L1 instruction cache, a 32 KiB four-way set associative L1 data cache, and a 1 MiB 16-way set associative L2 cache running the integer SPEC2006 benchmarks.

The Cortex-A53 instruction cache miss rates for these benchmarks are very small. Figure 5.44 shows the data cache results for the Cortex-A53, which have significant L1 and L2 miss rates. The L1 data cache miss rates go from 0.5% to 37.3%, with a mean of 6.4% and a median of 2.4%. The (global) L2 cache miss rates vary from 0.1% to 9.0%, with a mean of 1.3% and a median of 0.3%. The L1 miss penalty for a 1 GHz Cortex-A53 is 12 clock cycles, while the L2 miss penalty is 124 clock cycles. Using these miss penalties, Figure 5.45 shows the average miss penalty per data access. When these low miss rates are multiplied by their high miss penalties, you can see that they can represent a significant fraction of the CPI for 5 of the 12 SPEC2006 programs.

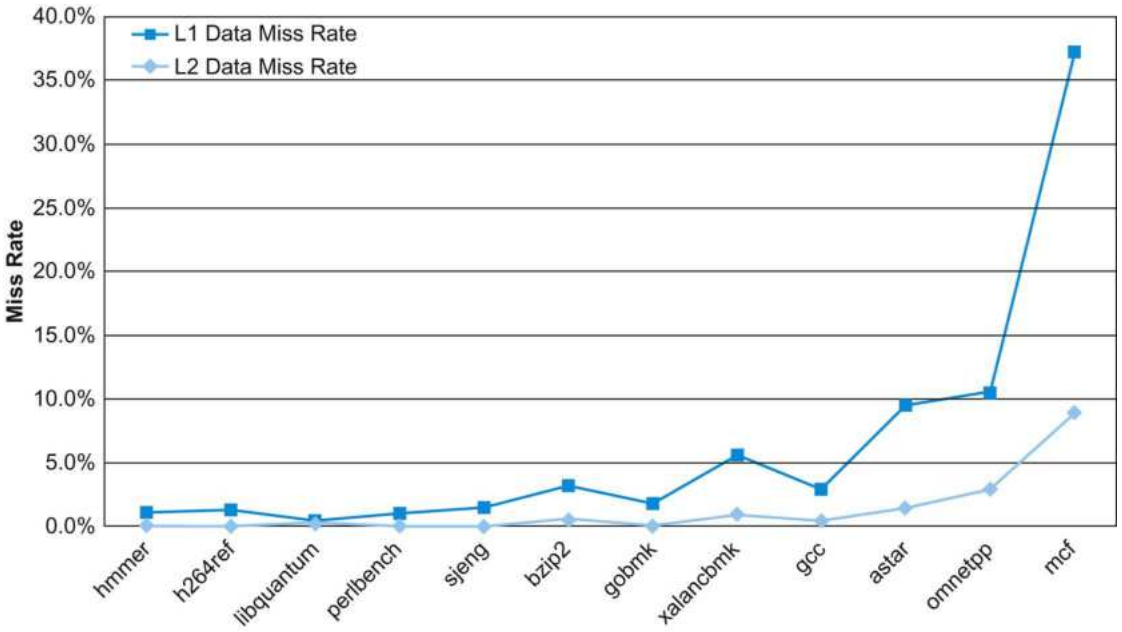


FIGURE 5.44 Data cache miss rates for ARM Cortex-A53 when running SPEC2006int. Applications with larger memory footprints tend to have higher miss rates in both L1 and L2. Note that the L2 rate is the global miss rate; that is, counting all references, including those that hit in L1. (See the *Elaboration* in Section 5.4.) mcf is known as a cache buster. Note that this figure is for the same systems and benchmarks as Figure 4.74 in Chapter 4.

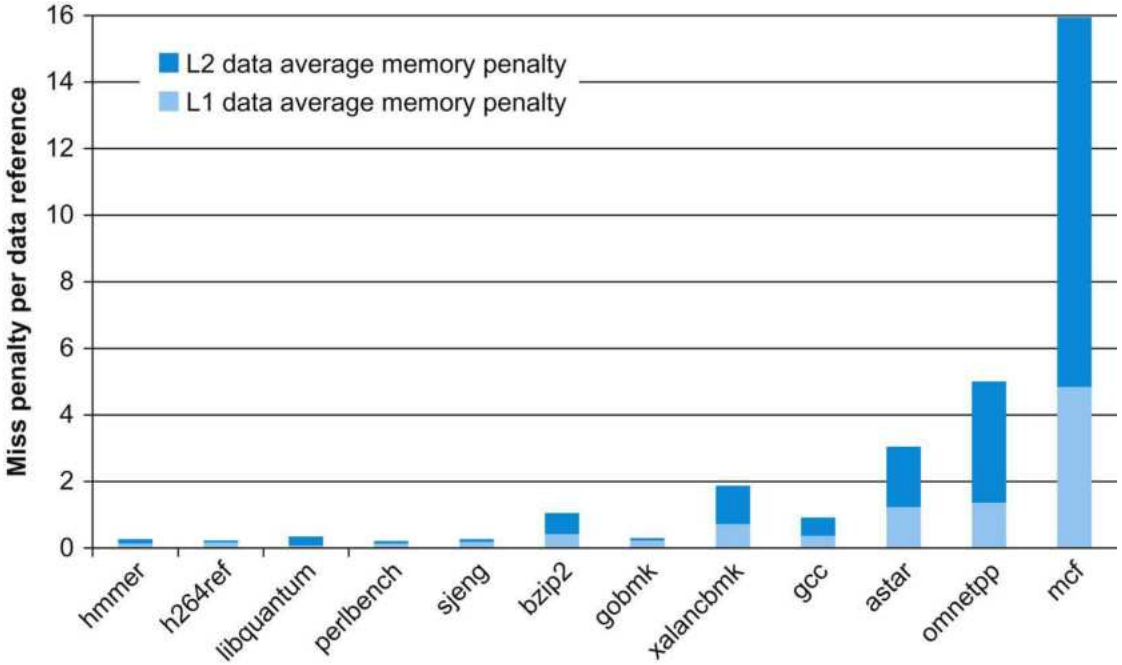


FIGURE 5.45 The average memory access penalty

in clock cycles per data memory reference coming from L1 and L2 is shown for the ARM processor when running SPEC2006int.
Although the miss rates for L1 are significantly higher, the L2 miss penalty, which is more than five times higher, means that the L2 misses can contribute significantly.

Figure 5.46 shows the miss rates for the caches of the Core i7 using the SPEC2006 benchmarks. The L1 instruction cache miss rate varies from 0.1% to 1.8%, averaging just over 0.4%. This rate is in keeping with other studies of instruction cache behavior for the SPECCPU2006 benchmarks, which show low instruction cache miss rates. With L1 data cache miss rates running 5% to 10%, and sometimes higher, the importance of the L2 and L3 caches should be obvious. Since the cost for a miss to memory is over 100 cycles, and the average data miss rate in L2 is 4%, L3 is obviously critical. Assuming about half the instructions is loads or stores, without L3 the L2 cache misses could add two cycles per instruction to the CPI! In comparison, the average L3 data miss rate of 1% is still significant but four times lower than the L2 miss rate and six times less than the L1 miss rate.

Elaboration

Because speculation may sometimes be wrong (see Chapter 4), there are references to the L1 data cache that do not correspond to loads or stores that eventually complete execution. The data in Figure 5.44 are measured against all data requests, including some that are cancelled. The miss rate when measured against only completed data accesses is 1.6 times higher (an average of 9.5% versus 5.9% for L1 Dcache misses).

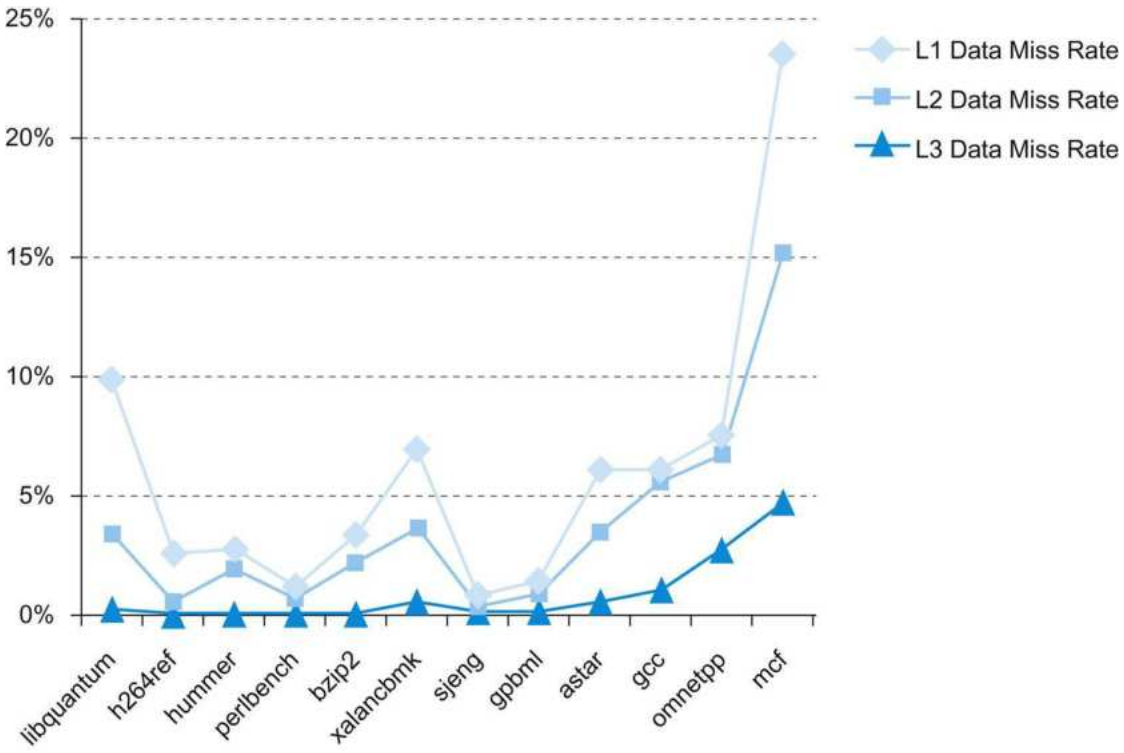


FIGURE 5.46 The L1, L2, and L3 data cache miss rates for the Intel Core i7 920 running the full integer SPECCPU2006 benchmarks.

5.14 Real Stuff: The Rest of the RISC-V System and Special Instructions

Figure 5.48 lists the 13 remaining RISC-V instructions in the special purpose and systems category.

The fence instructions provide synchronization barriers for instructions (**`fence.i`**), data (**`fence`**), and address translations (**`sfence.vma`**). The first, `fence.i`, informs the processor that software has modified instruction memory, so that it can guarantee that instruction fetch will reflect the updated instructions. The second, `fence`, affects data memory access ordering for multiprocessing and I/O. The third, `sfence.vma`, informs the processor that software has modified the page tables, so that it can guarantee that address translations will reflect the updates.

The six *control and status register* (CSR) access instructions move data between general-purpose registers and CSRs. The `csrrwi` instruction (CSR read/write immediate) copies a CSR to an integer

register, then overwrites the CSR with an immediate. `csrrsi` (CSR read/set immediate) copies a CSR to an integer register, and overwrites the CSR with the bitwise OR of the CSR and an immediate. `csrrci` (CSR read/clear) is like `csrrsi`, but clears bits instead of setting them. The `csrrw`, `csrrs`, and `csrrc` instructions use a register operand instead of an immediate, but otherwise do the same thing.

Two instructions' only purpose is to generate exceptions: `ecall` generates an environment call exception to invoke the OS, and `ebreak` generates a breakpoint exception to invoke the debugger. The supervisor exception-return instruction (`sret`), naturally enough, allows the program to return from an exception handler.

Finally, the wait-for-interrupt instruction, `wfi`, informs the processor that it may enter an idle state until an interrupt occurs.

5.15 Going Faster: Cache Blocking and Matrix Multiply

Our next step in the continuing saga of improving performance of DGEMM by tailoring it to the underlying hardware is to add cache blocking to the subword parallelism and instruction level parallelism optimizations of Chapters 3 and 4. Figure 5.47 shows the blocked version of DGEMM from Figure 4.78. The changes are the same as was made earlier in going from unoptimized DGEMM in Figure 3.22 to blocked DGEMM in Figure 5.21 above. This time we take the unrolled version of DGEMM from Chapter 4 and invoke it many times on the submatrices of `A`, `B`, and `C`. Indeed, lines 28–34 and lines 7–8 in Figure 5.47 mirror lines 14–20 and lines 5–6 in Figure 5.21, except for incrementing the for loop in line 7 by the amount unrolled.

```
 1 #include <x86intrin.h>
 2 #define UNROLL (4)
 3 #define BLOCKSIZE 32
 4 void do_block (int n, int si, int sj, int sk,
 5                 double *A, double *B, double *C)
 6 {
 7   for ( int i = si; i < si+BLOCKSIZE; i+=UNROLL*4 )
 8     for ( int j = sj; j < sj+BLOCKSIZE; j++ ) {
 9       __m256d c[4];
10       for ( int x = 0; x < UNROLL; x++ )
11         c[x] = _mm256_load_pd(C+i+x*4+j*n);
12      /* c[x] = C[i][j] */
13       for( int k = sk; k < sk+BLOCKSIZE; k++ )
14       {
15         __m256d b = _mm256_broadcast_sd(B+k+j*n);
16      /* b = B[k][j] */
17         for (int x = 0; x < UNROLL; x++)
18           c[x] = _mm256_add_pd(c[x], /* c[x]+=A[i][k]*b */
19                  _mm256_mul_pd(_mm256_load_pd(A+n*k+x*4+i), b));
20       }
21
22
23       for ( int x = 0; x < UNROLL; x++ )
24         _mm256_store_pd(C+i+x*4+j*n, c[x]);
           /* C[i][j] = c[x] */
25     }
26 }
27
28 void dgemm (int n, double* A, double* B, double* C)
29 {
30   for ( int sj = 0; sj < n; sj += BLOCKSIZE )
31     for ( int si = 0; si < n; si += BLOCKSIZE )
32       for ( int sk = 0; sk < n; sk += BLOCKSIZE )
33         do_block(n, si, sj, sk, A, B, C);
34 }
```

FIGURE 5.47 Optimized C version of DGEMM from Figure 4.78 using cache blocking. These changes are the same ones found in Figure 5.21. The assembly language produced by the compiler for the `do_block` function is nearly identical to Figure 4.79. Once again, there is no overhead to call the `do_block` because the compiler inlines the function call.

Unlike the earlier chapters, we do not show the resulting x86 code because the inner loop code is nearly identical to Figure 4.79, as the blocking does not affect the computation, just the order that it accesses data in memory. What does change is the bookkeeping integer instructions to implement the loops. It expands from 14

instructions before the inner loop and eight after the loop for Figure 4.78 to 40 and 28 instructions respectively for the bookkeeping code generated for Figure 5.47. Nevertheless, the extra instructions executed pale in comparison to the performance improvement of reducing cache misses. Figure 5.49 compares unoptimized to optimized for subword parallelism, instruction level parallelism, and caches. Blocking improves performance over unrolled AVX code by factors of 2 to 2.5 for the larger matrices. When we compare unoptimized code to the code with all three optimizations, the performance improvement is factors of 8 to 15, with the largest increase for the largest matrix.

Elaboration

As mentioned in the *Elaboration* in Section 3.9, these results are with Turbo mode turned off. As in Chapters 3 and 4, when we turn it on, we improve all the results by the temporary increase in the clock rate of 3.3/2.6 =1.27. Turbo mode works particularly well in this case because it is using only a single core of an eight-core chip. However, if we want to run fast we should use all cores, which we'll see in Chapter 6.

Type	Mnemonic	Name
Mem. Ordering	FENCE.I	Instruction Fence
	FENCE	Fence
	SFENCE.VMA	Address Translation Fence
CSR Access	CSRRWI	CSR Read/Write Immediate
	CSRRSI	CSR Read/Set Immediate
	CSRRCI	CSR Read/Clear Immediate
	CSRRW	CSR Read/Write
	CSRRS	CSR Read/Set
	CSRRC	CSR Read/Clear
System	ECALL	Environment Call
	EBREAK	Environment Breakpoint
	SRET	Supervisor Exception Return
	WFI	Wait for Interrupt

FIGURE 5.48 The list of assembly language instructions for the systems and special operations in the full RISC-V instruction set.

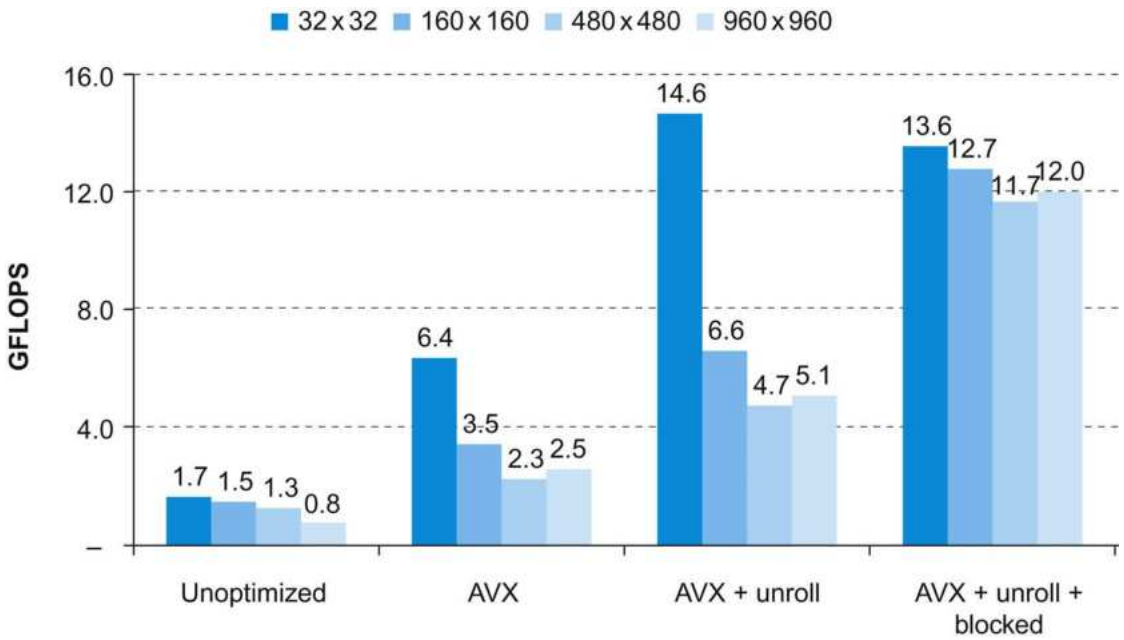


FIGURE 5.49 Performance of four versions of DGEMM from matrix dimensions 32 ×32 to 960 ×960.
The fully optimized code for the largest matrix is almost 15 times as fast the unoptimized version in Figure 3.22 in Chapter 3.

5.16 Fallacies and Pitfalls

As one of the most naturally quantitative aspects of computer architecture, the memory hierarchy would seem to be less vulnerable to fallacies and pitfalls. Not only have there been many fallacies propagated and pitfalls encountered, but some have led to major negative outcomes. We start with a pitfall that often traps students in exercises and exams.

Pitfall: Ignoring memory system behavior when writing programs or when generating code in a compiler.

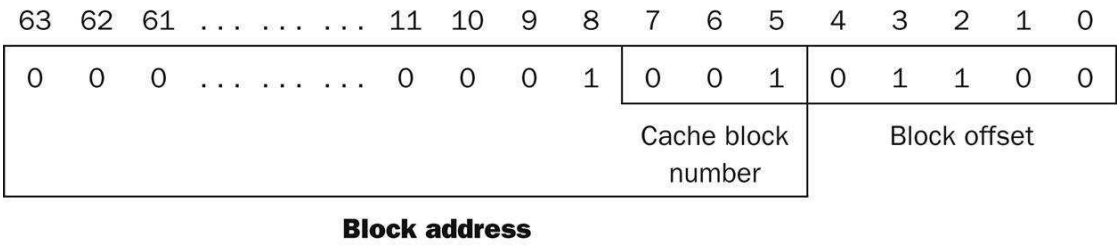
This could be rewritten as a fallacy: "Programmers can ignore

memory hierarchies in writing code." The evaluation of sort in Figure 5.19 and of cache blocking in Section 5.14 demonstrate that programmers can easily double performance if they factor the behavior of the memory system into the design of their algorithms.

Pitfall: Forgetting to account for byte addressing or the cache block size in simulating a cache.

When simulating a cache (by hand or by computer), we need to make sure we account for the effect of byte addressing and multiword blocks in determining into which cache block a given address maps. For example, if we have a 32-byte direct-mapped cache with a block size of 4 bytes, the byte address 36 maps into block 1 of the cache, since byte address 36 is block address 9 and (9 modulo 8)=1. On the other hand, if address 36 is a word address, then it maps into block (36 mod 8)=4. Make sure the problem clearly states the base of the address.

In like fashion, we must account for the block size. Suppose we have a cache with 256 bytes and a block size of 32 bytes. Into which block does the byte address 300 fall? If we break the address 300 into fields, we can see the answer:



Byte address 300 is block address

$$\left\lfloor \frac{300}{32} \right\rfloor = 9$$

The number of blocks in the cache is

$$\left\lfloor \frac{256}{32} \right\rfloor = 8$$

Block number 9 falls into cache block number (9 modulo 8)=1.

This mistake catches many people, including the authors (in earlier drafts) and instructors who forget whether they intended the addresses to be in doublewords, words, bytes, or block numbers. Remember this pitfall when you tackle the exercises.

Pitfall: Having less set associativity for a shared cache than the number of cores or threads sharing that cache.

Without extra care, a **parallel** program running on 2^n processors or threads can easily allocate data structures to addresses that would map to the same set of a shared L2 cache. If the cache is at least 2^n-way associative, then these accidental conflicts are hidden by the hardware from the program. If not, programmers could face apparently mysterious performance bugs—actually due to L2 conflict misses—when migrating from, say, a 16-core design to 32-core design if both use 16-way associative L2 caches.

Pitfall: Using average memory access time to evaluate the memory hierarchy of an out-of-order processor.

If a processor stalls during a cache miss, then you can separately calculate the memory-stall time and the processor execution time, and hence evaluate the memory hierarchy independently using average memory access time (see page 391).

If the processor continues to execute instructions, and may even sustain more cache misses during a cache miss, then the only accurate assessment of the memory hierarchy is to simulate the out-of-order processor along with the memory hierarchy.

Pitfall: Extending an address space by adding segments on top of an unsegmented address space.

During the 1970s, many programs grew so large that not all the code and data could be addressed with just a 16-bit address. Computers were then revised to offer 32-bit addresses, either through an unsegmented 32-bit address space (also called a *flat address space*) or by adding 16 bits of segment to the existing 16-bit address. From a marketing point of view, adding segments that were programmer-visible and that forced the programmer and compiler to decompose programs into segments could solve the addressing problem. Unfortunately, there is trouble any time a programming language wants an address that is larger than one segment, such as indices for large arrays, unrestricted pointers, or reference parameters. Moreover, adding segments can turn every address into two words—one for the segment number and one for the segment offset—causing problems in the use of addresses in registers.

Fallacy: Disk failure rates in the field match their specifications.

Two recent studies evaluated large collections of disks to check the relationship between results in the field compared to specifications. One study was of almost 100,000 disks that had quoted MTTF of 1,000,000 to 1,500,000 hours, or AFR of 0.6% to 0.8%. They found AFRs of 2% to 4% to be common, often three to five times higher than the specified rates [Schroeder and Gibson, 2007]. A second study of more than 100,000 disks at Google, which had a quoted AFR of about 1.5%, saw failure rates of 1.7% for drives in their first year rise to 8.6% for drives in their third year, or about

five to six times the declared rate [Pinheiro, Weber, and Barroso, 2007].

> *Fallacy: Operating systems are the best place to schedule disk accesses.*

As mentioned in Section 5.2, higher-level disk interfaces offer logical block addresses to the host operating system. Given this high-level abstraction, the best an OS can do to try to help performance is to sort the logical block addresses into increasing order. However, since the disk knows the actual mapping of the logical addresses onto the physical geometry of sectors, tracks, and surfaces, it can reduce the rotational and seek latencies by rescheduling.

For example, suppose the workload is four reads [Anderson, 2003]:

Operation	Starting LBA	Length
Read	724	8
Read	100	16
Read	9987	1
Read	26	128

The host might reorder the four reads into logical block order:

Operation	Starting LBA	Length
Read	26	128
Read	100	16
Read	724	8
Read	9987	1

Depending on the relative location of the data on the disk, reordering could make it worse, as Figure 5.50 shows. The disk-scheduled reads complete in three-quarters of a disk revolution, but the OS-scheduled reads take three revolutions.

> *Pitfall: Implementing a virtual machine monitor on an instruction set architecture that wasn't designed to be virtualizable.*

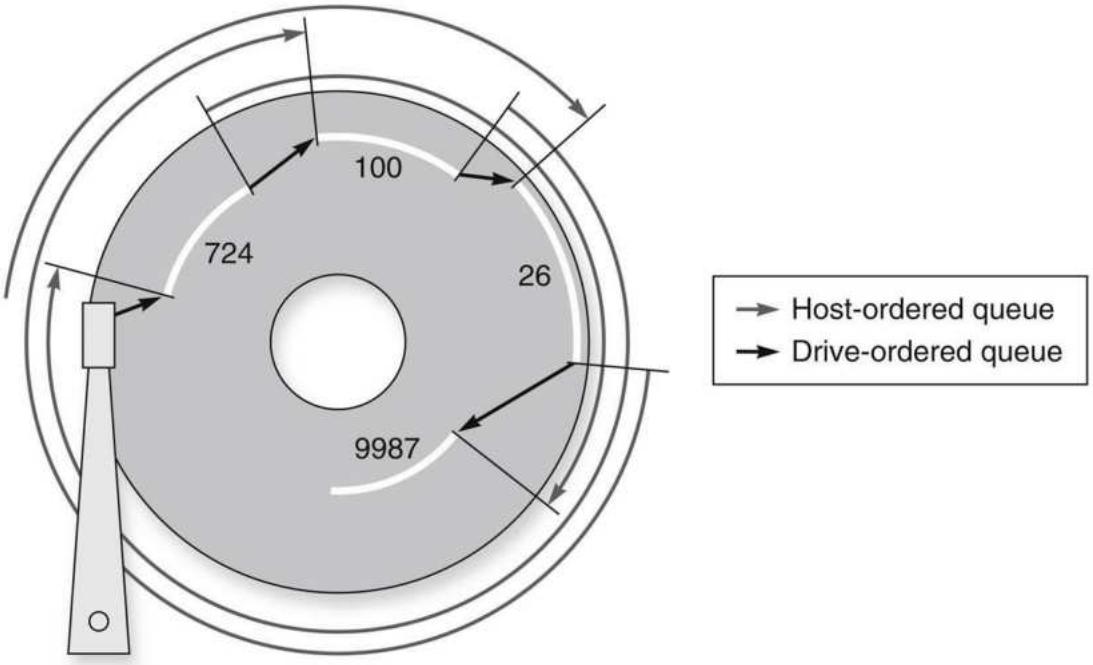


FIGURE 5.50 Example showing OS versus disk schedule accesses, labeled host-ordered versus drive-ordered.
The former takes three revolutions to complete the four reads, while the latter completes them in just three-fourths of a revolution. From Anderson [2003].

Many architects in the 1970s and 1980s weren't careful to make sure that all instructions reading or writing information related to hardware resource information were privileged. This *laissez-faire* attitude causes problems for VMMs for all of these architectures, including the x86, which we use here as an example.

Figure 5.51 describes the 18 instructions that cause problems for virtualization [Robin and Irvine, 2000]. The two broad classes are instructions that

- Read control registers in user mode that reveals that the guest operating system is running in a virtual machine (such as POPF, mentioned earlier)
- Check protection as required by the segmented architecture but assume that the operating system is running at the highest privilege level

Problem category	Problem x86 instructions
Access sensitive registers without trapping when running in user mode	Store global descriptor table register (SGDT) Store local descriptor table register (SLDT) Store interrupt descriptor table register (SIDT) Store machine status word (SMSW) Push flags (PUSHF, PUSHFD) Pop flags (POPF, POPFD)
When accessing virtual memory mechanisms in user mode, instructions fail the x86 protection checks	Load access rights from segment descriptor (LAR) Load segment limit from segment descriptor (LSL) Verify if segment descriptor is readable (VERR) Verify if segment descriptor is writable (VERW) Pop to segment register (POP CS, POP SS, . . .) Push segment register (PUSH CS, PUSH SS, . . .) Far call to different privilege level (CALL) Far return to different privilege level (RET) Far jump to different privilege level (JMP) Software interrupt (INT) Store segment selector register (STR) Move to/from segment registers (MOVE)

FIGURE 5.51 Summary of 18 x86 instructions that cause problems for virtualization [Robin and Irvine, 2000]. The first five instructions in the top group allow a program in user mode to read a control register, such as descriptor table registers, without causing a trap. The pop flags instruction modifies a control register with sensitive information but fails silently when in user mode. The protection checking of the segmented architecture of the x86 is the downfall of the bottom group, as each of these instructions checks the privilege level implicitly as part of instruction execution when reading a control register. The checking assumes that the OS must be at the highest privilege level, which is not the case for guest VMs. Only the Move to segment register tries to modify control state, and protection checking foils it as well.

To simplify implementations of VMMs on the x86, both AMD and Intel have proposed extensions to the architecture via a new mode. Intel's VT-x provides a new execution mode for running VMs, an architected definition of the VM state, instructions to swap VMs rapidly, and a large set of parameters to select the circumstances where a VMM must be invoked. Altogether, VT-x adds 11 new instructions for the x86. AMD's Pacifica makes similar proposals.

An alternative to modifying the hardware is to make small

changes to the operating system to avoid using the troublesome pieces of the architecture. This technique is called *paravirtualization*, and the open source Xen VMM is a good example. The Xen VMM provides a guest OS with a virtual machine abstraction that uses only the easy-to-virtualize parts of the physical x86 hardware on which the VMM runs.

5.17 Concluding Remarks

The difficulty of building a memory system to keep pace with faster processors is underscored by the fact that the raw material for main memory, DRAMs, is essentially the same in the fastest computers as it is in the slowest and cheapest.

It is the principle of locality that gives us a chance to overcome the long latency of memory access—and the soundness of this strategy is demonstrated at all levels of the **memory hierarchy**. Although these levels of the hierarchy look quite different in quantitative terms, they follow similar strategies in their operation and exploit the same properties of locality.

Multilevel caches make it possible to use more cache optimizations more easily for two reasons. First, the design parameters of a lower-level cache are different from a first-level

cache. For example, because a lower-level cache will be much larger, it is possible to use bigger block sizes. Second, a lower-level cache is not constantly being used by the processor, as a first-level cache is. This allows us to consider having the lower-level cache do something when it is idle that may be useful in preventing future misses.

Another trend is to seek software help. Efficiently managing the memory hierarchy using a variety of program transformations and hardware facilities is a major focus of compiler enhancements. Two different ideas are being explored. One idea is to reorganize the program to enhance its spatial and temporal locality. This approach focuses on loop-oriented programs that use sizable arrays as the major data structure; large linear algebra problems are a typical example, such as DGEMM. By restructuring the loops that access the arrays, substantially improved locality—and, therefore, cache performance—can be obtained.

Another approach is **prefetching**. In prefetching, a block of data is brought into the cache before it is actually referenced. Many

microprocessors use hardware prefetching to try to *predict* accesses that may be difficult for software to notice.

prefetching
A technique in which data blocks needed in the future are brought into the cache early by using special instructions that specify the address of the block.

A third approach is special cache-aware instructions that optimize memory transfer. For example, the microprocessors in Section 6.10 in Chapter 6 use an optimization that does not fetch the contents of a block from memory on a write miss because the program is going to write the full block. This optimization significantly reduces memory traffic for one kernel.

As we will see in Chapter 6, memory systems are a central design issue for parallel processors. The growing significance of the memory hierarchy in determining system performance means that this important area will continue to be a focus for both designers and researchers for some years to come.

Historical Perspective and Further Reading

This section, which appears online, gives an overview of memory technologies, from mercury delay lines to DRAM, the invention of the memory hierarchy, protection mechanisms, and virtual machines, and concludes with a brief history of operating systems, including CTSS, MULTICS, UNIX, BSD UNIX, MS-DOS, Windows, and Linux.

5.18 Historical Perspective and Further Reading

> *…the one single development that put computers on their feet was the invention of a reliable form of memory, namely, the core memory…. Its cost was reasonable, it was reliable and, because it was reliable, it could in due course be made large.*
>
> *Maurice Wilkes, Memoirs of a Computer Pioneer, 1985*

This history section gives an overview of memory technologies, from mercury delay lines to DRAM, the invention of the memory hierarchy, and protection mechanisms, and concludes with a brief history of operating systems, including CTSS, MULTICS, UNIX, BSD UNIX, MS-DOS, Windows, and Linux.

The developments of most of the concepts in this chapter have been driven by revolutionary advances in the technology we use for memory. Before we discuss how memory hierarchies were evolved, let's take a brief tour of the development of memory technology.

The ENIAC had only a small number of registers (about 20) for its storage and implemented these with the same basic vacuum tube technology that it used for building logic circuitry. However, the vacuum tube technology was far too expensive to be used to build a larger memory capacity. Eckert came up with the idea of developing a new technology based on mercury delay lines. In this technology, electrical signals were converted into vibrations that were sent down a tube of mercury, reaching the other end, where they were read out and recirculated. One mercury delay line could store about 0.5 Kbits. Although these bits were accessed serially, the mercury delay line was about a hundred times more cost-effective than vacuum tube memory. The first known working mercury delay lines were developed at Cambridge for the EDSAC. Figure e5.17.1 shows the mercury delay lines of the EDSAC, which had 32 tanks and 512 36-bit words.

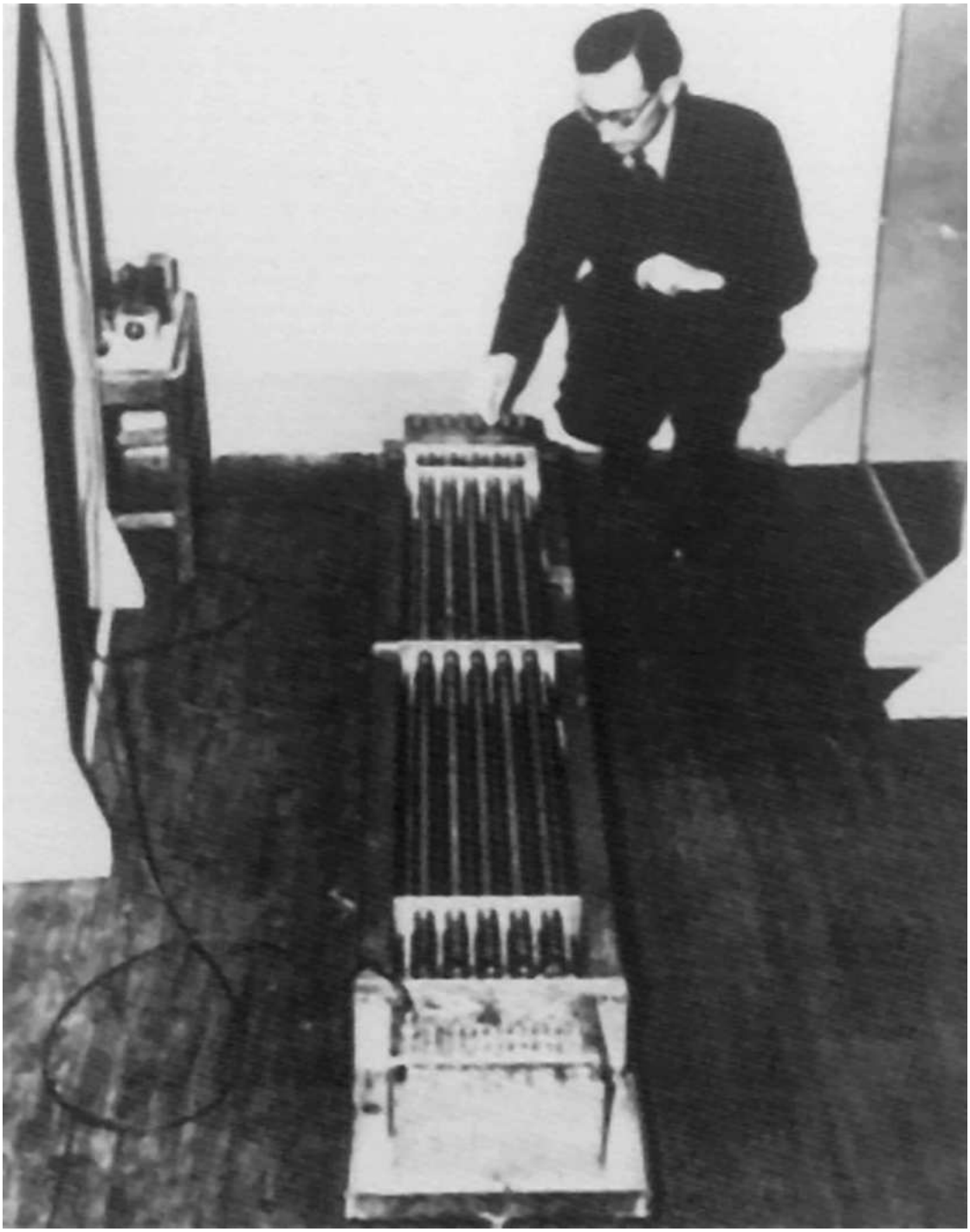


FIGURE E5.17.1 The mercury delay lines in the EDSAC.
This technology made it possible to build the first stored-program computer. The young engineer in this photograph is none other than Maurice Wilkes, the lead architect of the EDSAC.

Despite the tremendous advance offered by the mercury delay lines, they were terribly unreliable and still rather expensive. The

breakthrough came with the invention of core memory by J. Forrester at MIT as part of the Whirlwind project in the early 1950s (see Figure e5.17.2). Core memory uses a ferrite core, which can be magnetized, and once magnetized, it acts as a store (just as a magnetic recording tape stores information). A set of wires running through the center of the core, which had a dimension of 0.1–1.0 millimeters, makes it possible to read the value stored on any ferrite core. The Whirlwind eventually included a core memory with 2048 16-bit words, or 32 Kbits. Core memory was a tremendous advance: it was cheaper, faster, considerably more reliable, and had higher density. Core memory was so much better than the alternatives that it became the dominant memory technology only a few years after its invention and remained so for nearly 20 years.

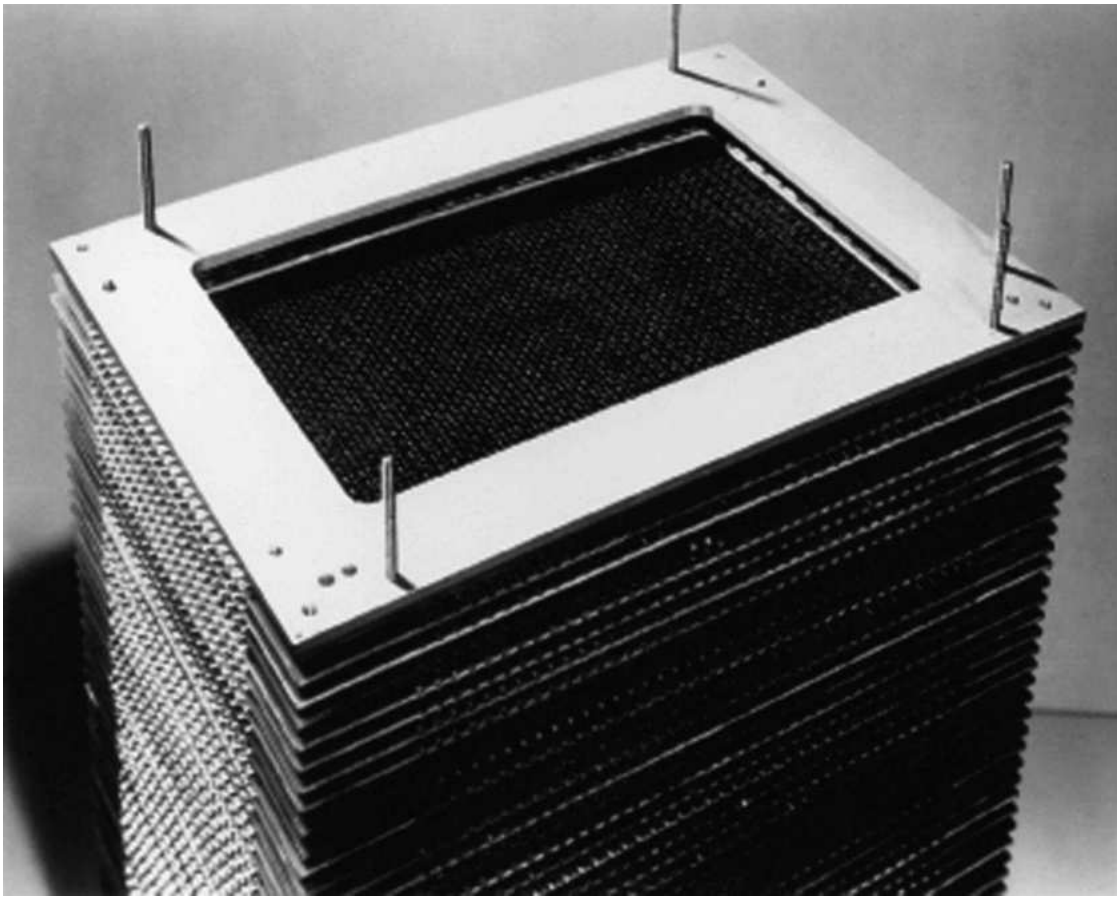


FIGURE E5.17.2 A core memory plane from the Whirlwind containing 256 cores arranged in a 16×16 array.
Core memory was invented for the Whirlwind, which was used for air defense problems, and is now on display at the Smithsonian. (Incidentally, Ken Olsen, the founder of Digital and its president for 20 years, built the computer that tested these core memories; it was his first computer.)

The technology that replaced core memory was the same one that we now use both for logic and for memory: the integrated circuit. While registers were built out of transistorized memory in the 1960s, and IBM computers used transistorized memory for microcode store and caches in 1970, building main memory out of transistors remained prohibitively expensive until the development of the integrated circuit. With the integrated circuit, it became possible to build a DRAM (dynamic random access memory—see Appendix A for a description). The first DRAMs were built at Intel in 1970, and the computers using DRAM memories (as a high-speed

option to core) came shortly thereafter; they used 1 Kbit DRAMs. In fact, computer folklore says that Intel developed the microprocessor partly to help sell more DRAM. Figure e5.17.3 shows an early DRAM board. By the late 1970s, core memory had become a historical curiosity. Just as core memory technology had allowed a tremendous expansion in memory size, DRAM technology allowed a comparable expansion. In the 1990s, many personal computers had as much memory as the largest computers using core memory ever had.

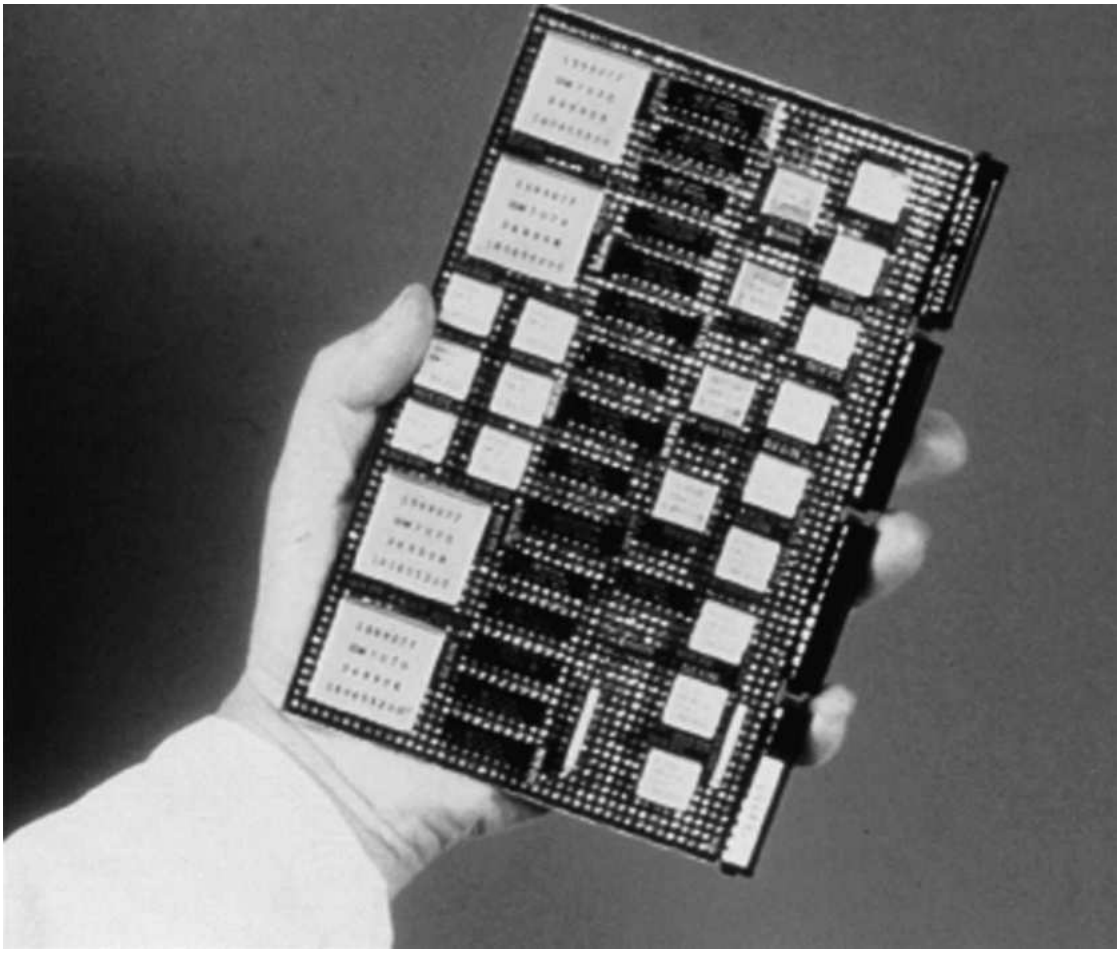


FIGURE E5.17.3 An early DRAM board. This board uses 18 Kbit chips.

Nowadays, DRAMs are typically packaged with multiple chips on a little board called a DIMM (dual inline memory module). The SIMM (single inline memory module) shown in Figure e5.17.4 contains a total of 1 MB and sold for about $5 in 1997. As of 2004, DIMMs were available with up to 1024 MB and sold for about $100. While DRAMs will remain the dominant memory technology for some time to come, innovations in the packaging of DRAMs to provide both higher bandwidth and greater density are ongoing.

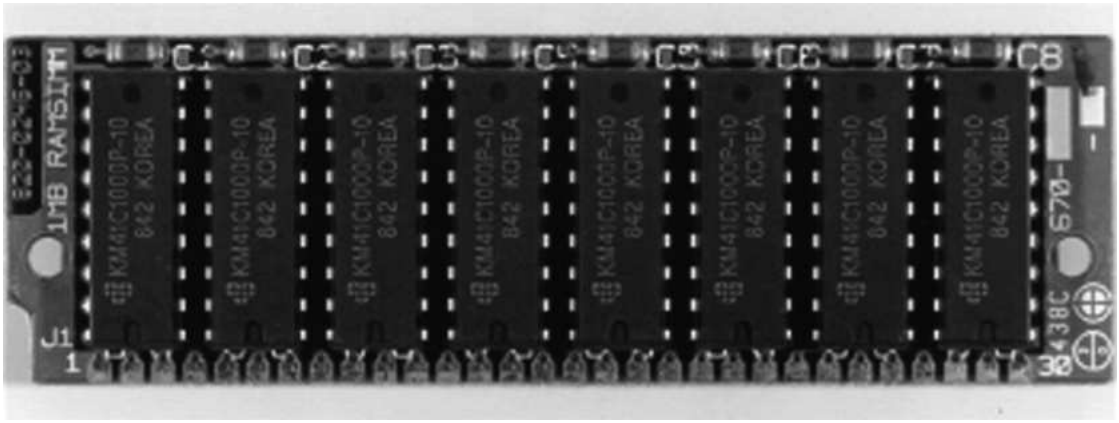


FIGURE E5.17.4 A 1 MB SIMM, built in 1986, using 1 Mbit chips.
This SIMM sold for about $5/MB in 1997. As of 2006, most main memory is packed in DIMMs similar to this, though using much higher-density memory chips (1 Gbit).

The Development of Memory Hierarchies

Although the pioneers of computing foresaw the need for a **memory hierarchy** and coined the term, the automatic management of two levels was first proposed by Kilburn and his colleagues and demonstrated at the University of Manchester with the Atlas computer, which implemented virtual memory. This was the year *before* the IBM 360 was announced. IBM planned to include virtual memory with the next generation (System/370), but the OS/360 operating system wasn't up to the challenge in 1970. Virtual memory was announced for the 370 family in 1972, and it was for this computer that the term *translation-lookaside buffer* was coined. All but some embedded computers use virtual memory today.

The problems of inadequate address space have plagued designers repeatedly. The architects of the PDP-11 identified a small address space as the only architectural mistake from which it is difficult to recover. When the PDP-11 was designed, core memory densities were increasing at a very slow rate, and the competition from 100 other minicomputer companies meant that DEC might not have a cost-competitive product if every address had to go through the 16-bit datapath twice—hence, the decision to add just 4 more address bits than the predecessor of the PDP-11, to 16 from 12. The architects of the IBM 360 were aware of the importance of address size and planned for the architecture to extend to 32 bits of address. Only 24 bits were used in the IBM 360, however, because the low-end 360 models would have been even slower with the larger addresses. Unfortunately, the expansion effort was greatly complicated by programmers who stored extra information in the upper 8 "unused" address bits. The wider address lasted until 2000, when IBM expanded the architecture to 64 bits in the z-series.

Running out of address space has often been the cause of death for an architecture, while other architectures have managed to make the transition to a larger address space. For example, the PDP-11, a 16-bit computer, was replaced by the 32-bit VAX. The 80386 extended the 80286 architecture from a segmented 24-bit address space to a flat 32-bit address space in 1985. In the 1990s, several RISC instruction sets made the transition from 32-bit addressing to 64-bit addressing by providing a compatible

extension of their instruction sets. MIPS was the first to do so. A decade later, Intel and HP announced the IA-64 in large part to provide a 64-bit address successor to the 32-bit Intel IA-32 and HP Precision architectures. The evolutionary AMD64 won that battle versus the revolutionary IA-64, and all but a few thousand of the 64-bit address computers from Intel are based on the x86.

Many of the early ideas in memory hierarchies originated in England. Just a few years after the Atlas paper, Wilkes [1965] published the first paper describing the concept of a cache, calling it a "slave":

> *The use is discussed of a fast core memory of, say, 32,000 words as slave to a slower core memory of, say, one million words in such a way that in practical cases the effective access time is nearer that of the fast memory than that of the slow memory.*

This two-page paper describes a direct-mapped cache. Although this was the first publication on caches, the first implementation was probably a direct-mapped instruction cache built at the University of Cambridge by Scarrott and described at the 1965 IFIP Congress. It was based on tunnel diode memory, the fastest form of memory available at the time.

Subsequent to that publication, IBM started a project that led to the first commercial computer with a cache, the IBM 360/85. Gibson at IBM recognized that memory-accessing behavior would have a significant impact on performance. He described how to measure program behavior and cache behavior and showed that the miss rate varies between programs. Using a sample of 20 programs (each with 3 million references—an incredible number for that time), Gibson analyzed the effectiveness of caches using average memory access time as the metric. Conti, Gibson, and Pitowsky described the resulting performance of the 360/85 in the first paper to use the term *cache* in 1968. Since this early work, it has become clear that caches are one of the most important ideas not only in computer architecture but in software systems as well. The idea of caching has found applications in operating systems, networking systems, databases, and compilers, to name a few. There are thousands of papers on the topic of caching, and it continues to be a popular area of research.

One of the first papers on nonblocking caches was by Kroft in 1981, who may have coined the term. He later explained that he was the first to design a computer with a cache at Control Data Corporation, and when using old concepts for new mechanisms, he hit upon the idea of allowing his two-ported cache to continue to service other accesses on a miss.

Multilevel caches were the inevitable resolution to the lack of improvement in main memory latency and the higher clock rates of microprocessors. Only those in the field for a while are surprised by the size of some second- or third-level caches, as they are larger than main memories of past machines. The other surprise is that the number of levels is continually increasing, even on a single-chip microprocessor.

Disk Storage

In 1956, IBM developed the first disk storage system with both moving heads and multiple disk surfaces in San Jose, helping to seed the birth of the magnetic storage industry in the southern end of Silicon Valley. Reynold B. Johnson led the development of the IBM 305 RAMAC (Random Access Method of Accounting and Control). It could store 5 million characters (5 MB) of data on 50 disks, each 24 inches in diameter. The RAMAC is shown in Figures e5.17.5 and e5.17.6. Although the disk pioneers would be amazed at the size, cost, and capacity of modern disks, the basic mechanical design is the same as the RAMAC.

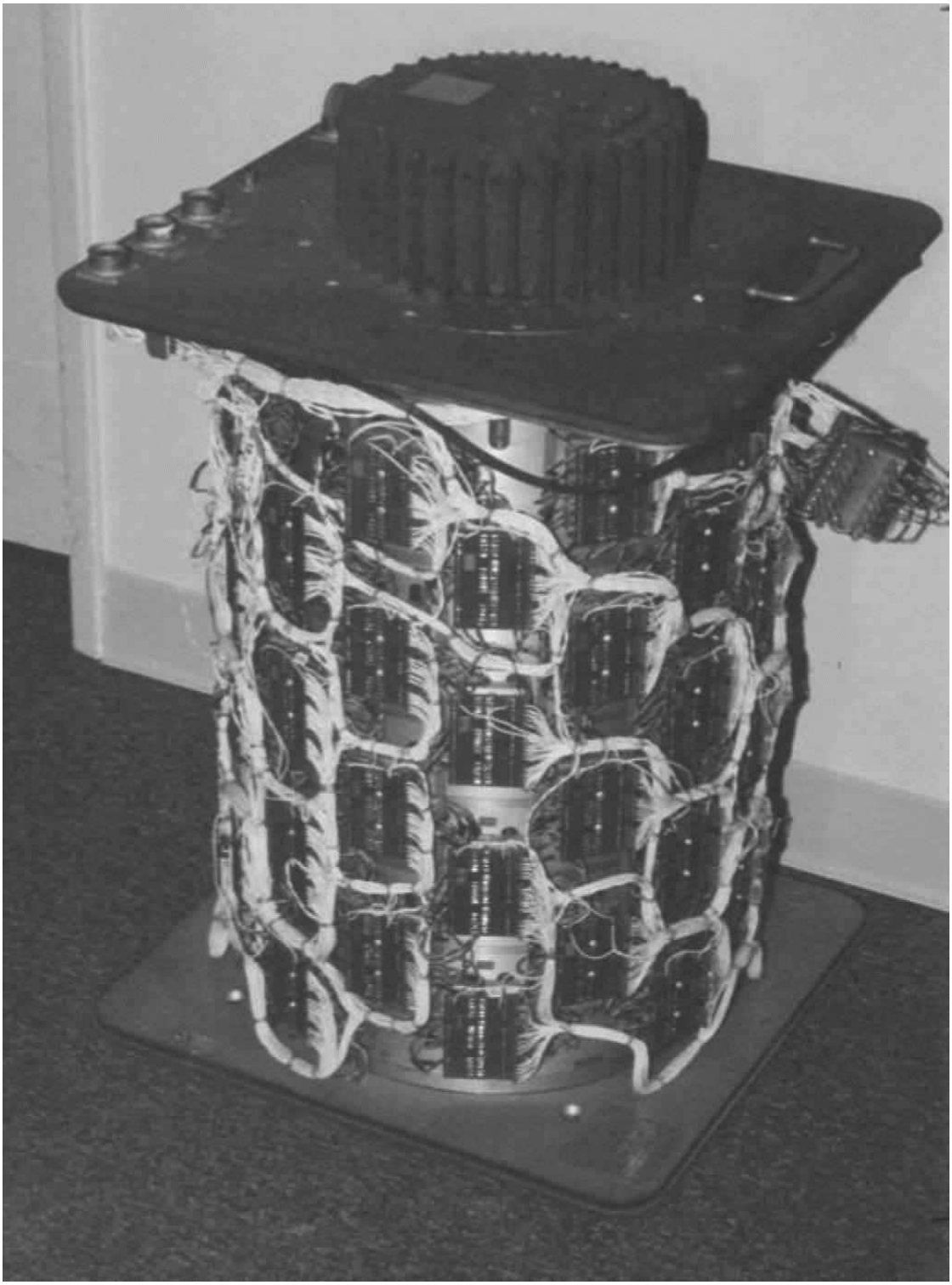


FIGURE E5.17.5 A magnetic drum made by Digital Development Corporation in the 1960s and used on a CDC machine.
The electronics supporting the read/write heads can be seen on the outside of the drum.

FIGURE E5.17.6 The RAMAC disk drive from IBM, made in 1956, was the first disk drive with a moving head and the first with multiple platters.
The IBM storage technology Web site has a discussion of IBM's major contributions to storage technology.

Moving-head disks quickly became the dominant high-speed magnetic storage, though their high cost meant that magnetic tape continued to be used extensively until the 1970s. The next key milestone for hard disks was the removable hard disk drive developed by IBM in 1962; this made it possible to share the expensive drive electronics and helped disks overtake tapes as the preferred storage medium. Figure e5.17.7 shows a removable disk drive and the multiplatter disk used in the drive. IBM also invented the floppy disk drive in 1970, originally to hold microcode for the IBM 370 series. Floppy disks became popular with the PC about 10 years later.

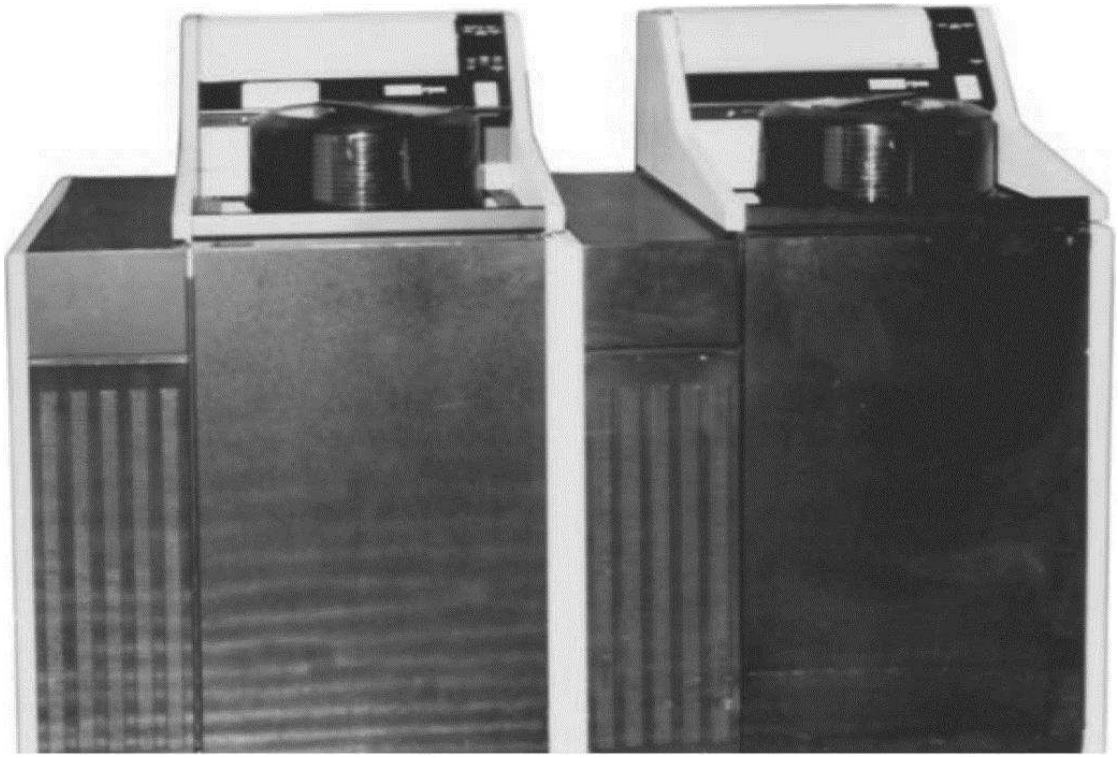


FIGURE E5.17.7 This is a DEC disk drive and the removable pack.
These disks became popular starting in the mid-1960s and dominated disk technology until Winchester drives in the late 1970s. This drive was made in the mid-1970s; each disk pack in this drive could hold 80 MB.

The sealed Winchester disk, which was developed by IBM in 1973, completely dominates disk technology today. Winchester disks benefited from two related properties. First, reductions in the cost of the disk electronics made it unnecessary to share the electronics and thus made nonremovable disks economical. Since the disk was fixed and could be in a sealed enclosure, both the environmental and control problems were greatly reduced, allowing significant gains in density. The first disk that IBM shipped had two spindles, each with a 30 MB disk; the moniker "30-30" for the disk led to the name Winchester. Winchester disks grew rapidly in popularity in the 1980s, completely replacing removable disks by the middle of that decade.

The historic role of IBM in the disk industry came to an end in 2002, when IBM sold its disk storage division to Hitachi. IBM continues to make storage subsystems, but it purchases its disk drives from others.

A Very Brief History of Flash Memory

Flash memory was invented by researchers at Toshiba in the 1980s. They invented both the NOR-based Flash memory in 1984 and the denser NAND-based Flash memory in 1989. The first use was in digital cameras, starting with the CompactFlash form factor for NOR Flash memory and the SmartMedia form factor for NAND Flash memory. Today, all digital cameras, cell phones, music players, and tablets rely on Flash memory, and an increasing fraction of laptops use flash memory instead of disk.

A Brief History of Databases

Although there had been data stores of punch cards and later magnetic tapes, the emergence of the magnetic disk led to modern databases.

In 1961, Charles Bachman at General Electric created a pioneering database management system called Integrated Data Store (IDS) to take advantage of the new magnetic disks. In 1971, Bachman and others published standards on how to manage databases using Cobol programs, named the Codasyl approach after the standards committee on which they served. Many companies offered Codasyl-compatible databases, but not IBM. IBM had introduced IMS in 1968, which was derived from IBM's work on the NASA Apollo project. Both Codasyl databases and IMS are classified as navigational databases because programs had to navigate through the data.

Ted Codd, a researcher at IBM, thought the navigational approach was wrong-headed. He recalled that people didn't write programs when dealing with the old punch card databases. Instead, they set up data flows through series of punch card machines that would perform simple functions like copy or sort. Once the card machines were set up, you just pushed all the cards through to get your results. In his view, users should only declare the type of data they were looking for and leave it up to computers to process it. In 1970, he published a new way to organize and access data called the relational model. It was based on set theory; data were independent of the implementation and users described what they were looking for in a declarative, nonprocedural language.

This paper led to considerable controversy within IBM, because it

already had a database product. Codd even arranged a public debate between him and Bachman, which led to internal criticism at IBM that Codd was undermining IMS. The good news was that the debate led researchers at IBM and U.C. Berkeley to try to demonstrate the viability of relational databases by building System R and Ingres.

System R in 1974–79 demonstrated its feasibility and, perhaps more importantly, created the Structured Query Language (SQL) that is still widely used today. However, these results were not sufficient to convince IBM, and some of the researchers left IBM to build relational databases for other companies.

Mike Stonebraker and Gene Wong were interested in geographic data systems, and in 1973 they decided to pursue relational databases. Rather than build on IBM mainframes, the Ingres project was built on DEC minicomputers and Unix. Ingres was important because it led to a company that tried to commercialize the ideas, because 1000 copies of its source code were openly distributed, and because it trained a generation of database developers and researchers. The code and people led to many other companies, including Sybase. Larry Ellison started Oracle by first reading the papers from the System R and Ingres groups and then by hiring people who worked on those projects. Microsoft later purchased a copy of Sybase sources that became the foundation of its SQL Server product.

Relational databases matured in the 1980s, with IBM developing its own relational databases, including DB2. The 1990s saw both the development of object-oriented databases, to address the impedance mismatch between databases and programming, and the evolution of parallel databases for analytic processing and data mining.

ACM showered awards on this community. The ACM Turing Award went to Charles Bachman in 1973 for his contributions via IDS and the Codasyl group. Codd won it in 1980 for the relational model. In 1988, the developers of System R (Donald Chamberlin, Jim Gray, Raymond Lorie, Gianfranco Putzolu, Patricia Selinger, and Irving Traiger) shared the ACM Systems Software Award with the developers of Ingres (Gerald Held, Michael Stonebraker, and Eugene Wong). Jim Gray won the Turing Award in 1998 for his contributions to transaction processing and databases. Stonebraker

won it in 2014 for contributions to the concepts and practices underlying modern database systems. Finally, the first two ACM SIGMOD Innovations Awards went to Stonebraker and Gray, and the 2002 and 2003 editions went to Selinger and Chamberlin.

RAID

The small-form-factor hard disks for PCs in the mid-1980s led a group at Berkeley to propose redundant arrays of inexpensive disks (RAID). This group had worked on the reduced instruction set computer effort and so expected much faster processors to become available. Their two questions were: What could be done with the small disks that accompanied their PCs? What could be done in the area of I/O to keep up with much faster processors? They argued to replace one large mainframe drive with 50 small drives, as you could get much greater performance with that many independent arms. The many small drives even offered savings in power consumption and floor space.

The downside of many disks was much lower MTTF. Hence, on their own they reasoned out the advantages of redundant disks and rotating parity to address how to get greater performance with many small drives yet have reliability as high as that of a single mainframe disk.

The problem they experienced when explaining their ideas was that some researchers had heard of disk arrays with some form of redundancy, and they didn't understand the Berkeley proposal. Hence, the first RAID paper [Patterson, Gibson, and Katz 1987] is not only a case for arrays of small-form-factor disk drives, but also something of a tutorial and classification of existing work on disk arrays. Mirroring (RAID 1) had long been used in fault-tolerant computers such as those sold by Tandem. Thinking Machines had arrays with 32 data disks and seven check disks using ECC for correction (RAID 2) in 1987, and Honeywell Bull had a RAID 2 product even earlier. Also, disk arrays with a single parity disk had been used in scientific computers in the same time frame (RAID 3). Their paper then described a single parity disk with support for sector accesses (RAID 4) and rotated parity (RAID 5). Chen et al. [1994] survey the original RAID ideas, commercial products, and other developments.

Unknown to the Berkeley group, engineers at IBM working on the AS/400 computer also came up with rotated parity to give greater reliability for a collection of large disks. IBM filed a patent on RAID 5 shortly before the Berkeley group submitted their paper. Patents for RAID 1, RAID 2, and RAID 3 from several companies predate the IBM RAID 5 patent, which has led to plenty of courtroom action.

EMC had been a supplier of DRAM boards for IBM computers, but around 1988 new policies from IBM made it nearly impossible for EMC to continue to sell IBM memory boards. The Berkeley paper crossed the desks of EMC executives, and so they decided to go after the market dominated by IBM disk storage products. As the paper advocated, their model was to use many small drives to compete with mainframe drives, and EMC announced a RAID product in 1990. It relied on mirroring (RAID 1) for reliability; RAID 5 products came much later for EMC. Over the next year, Micropolis offered a RAID 3 product; Compaq offered a RAID 4 product; and Data General, IBM, and NCR offered RAID 5 products.

The RAID ideas soon spread to the rest of the workstation and server industry. An article explaining RAID in *Byte* magazine led to RAID products being offered on desktop PCs, which was something of a surprise to the Berkeley group. They had focused on performance with good availability, but higher availability was attractive to the PC market.

Another surprise was the cost of the disk arrays. With redundant power supplies and fans, the ability to "hot-swap" a disk drive, the RAID hardware controller itself, the redundant disks, and so on, the first disk arrays cost many times the cost of the disks. Perhaps as a result, the "inexpensive" in RAID morphed into "independent." Many marketing departments and technical writers today know of RAID only as "redundant arrays of independent disks."

In 2004, more than 80% of the nondesktop drive sales were found in RAIDs. In recognition of their role, in 1999 Garth Gibson, Randy Katz, and David Patterson received the IEEE Reynold B. Johnson Information Storage Award "for the development of Redundant Arrays of Inexpensive Disks (RAID)."

Protection Mechanisms

Architectural support for protection has varied greatly over the past 20 years. In early computers, before virtual memory, protection was very simple at best. In the 1960s, more sophisticated mechanisms that supported different protection levels (called *rings*) were invented. In the late 1970s and early 1980s, very elaborate mechanisms for protection were devised and later built; these mechanisms supported a variety of powerful protection schemes that allowed controlled instances of sharing, in such a way that a process could share data while controlling exactly what was done to the data. The most powerful method, called *capabilities*, created a data object that described the access rights to some portion of memory. These capabilities could then be passed to other processes, thus granting access to the object described by the capability. Supporting this sophisticated protection mechanism was both complex and costly, because creation, copying, and manipulation of capabilities required a combination of operating system and hardware support. Recent computers all support a simpler protection scheme based on virtual memory, similar to that discussed in Section 5.7. Given current concerns about computer security due to the costs of worms and viruses, perhaps we will see a renaissance in protection research, potentially renewing interest in 20-year-old publications.

As mentioned in the text, system virtual machines were pioneered at IBM as part of its investigation into virtual memory. IBM's first computer with virtual memory was the IBM 360/67, introduced in 1967. IBM researchers wrote the program CP-67, which created the illusion of several independent 360 computers. They then wrote an interactive, single-user operating system called CMS that ran on these virtual machines. CP-67 led to the product VM/370, and today IBM sells z/VM for its mainframe computers.

A Brief History of Modern Operating Systems

MIT developed the first timesharing system, CTSS (Compatible Time-Sharing System), in 1961. John McCarthy is generally given credit for the idea of timesharing, but Fernando Corbato was the systems person who realized the concept in the form of the CTSS. CTSS allowed three people to share a machine, and its response time of minutes or seconds was a dramatic improvement over the

batch processing system it replaced. Moreover, it demonstrated the value of interactive computing.

Flush with the success of their first system, this group launched into their second system, MULTICS (Multiplexed Information and Computing Service). They included many innovations, such as strong protection, controlled sharing, and dynamic libraries. However, it suffered from the "second system effect." Fred Brooks, Jr. described the second system effect in his classic book about lessons learned from developing an operating system for the IBM mainframe, *The Mythical Man Month:*

> *When one is designing the successor to a relatively small, elegant, and successful system, there is a tendency to become grandiose in one's success and design an elephantine feature-laden monstrosity.*

MULTICS took sharing to a logical extreme to discover the issues, including that it was too extreme. MIT, General Electric, and later Bell Labs all tried to build an economical and useful system. Despite a great deal of time and money, they failed.

UC Berkeley was building its own timesharing system, Cal TSS. ("Cal" is a nickname for University of California.) The people leading that project included Peter Deutsch, Butler Lampson, Chuck Thacker, and Ken Thompson. They added paging virtual memory hardware to an SDS 920 and wrote an operating system for it. SDS sold this computer as the SDS-930, and it was the first commercially available timesharing system to have operational hardware and software. Thompson graduated and joined Bell Labs. The others founded Berkeley Computer Corporation (BCC), with the goal of selling time-sharing hardware and software. We'll pick up BCC later in the story, but for now let's follow Thompson.

At Bell Labs in 1971, Thompson led the development of a simple timesharing system that had some of the good ideas of MULTICS but left out many of the complex features. To demonstrate the contrast, it was first called UNICS. As they were joined by others at Bell Labs who had been burned from the MULTICS experience, it was renamed UNIX, with the *x* coming from Phoenix, the legendary bird that rose from the ashes.

Their result was the most elegant operating system ever built. Forced to live in the 16-bit address space of the DEC

minicomputers, it had an amazing amount of functionality per line of code. Major contributions were pipes, a uniform file system, a uniform process model, and the shell user interface that allowed users to connect programs together using pipes and files.

Dennis Ritchie joined the UNIX team in 1973 from MIT, where he had experience in MULTICS, which was written in a high-level language. Like prior operating systems, UNIX had been written in assembly language. Ritchie designed a language for system implementation called C, and it was used to make UNIX portable.

Between 1971 and 1976, Bell released six editions of the UNIX timesharing system. Thompson took a sabbatical at his alma mater and brought UNIX with him. Berkeley and many other universities began to use UNIX on the popular PDP-11 minicomputer.

When DEC announced the VAX, a 32-bit virtual address successor to the PDP-11, the question arose as to what operating system should be run. UNIX became the first operating system to be migrated to a different computer when it was ported to the VAX.

Students at Berkeley had one of the first VAXes, and they were soon adding features to UNIX for the VAX, such as paging and a very efficient implementation of the TCP/IP protocol. The Berkeley implementation of TCP/IP was notable not just because it was fast. It was essentially the *only* implementation of TCP/IP for years, since early implementations in most other operating systems consisted of copying the Berkeley code verbatim, with minimal changes to integrate into the local system.

The Advanced Research Project Agency (ARPA), which funded computer science research, asked a Stanford professor, Forrest Basket, to recommend which system the academic community should use: the DEC operating system VMS, led by David Cutler, or the Berkeley version of UNIX, led by a graduate student named Bill Joy. He recommended the latter, and Berkeley UNIX soon became the academic standard bearer.

The Berkeley Software Distribution (BSD) of UNIX, first released in 1978, was essentially one of the first open source movements. The sources were shipped with the tapes, and systems developers around the world learned their craft by studying the UNIX code.

BSD was also the first split of UNIX, because AT&T Bell Labs continued to develop UNIX on its own. This eventually led to a forest of UNIXes, as each company compiled the UNIX source code

for their architecture. Bill Joy graduated from Berkeley and helped found Sun Microsystems, so naturally Sun OS was based on BSD UNIX. Among the many UNIX flavors were Santa Cruz Operation UNIX, HP-UX, and IBM's AIX. AT&T and Sun attempted to unify UNIX by striking a deal whereby AT&T and Sun would combine forces and jointly develop AT&T UNIX. This led to an adverse reaction from HP, IBM, and others, because they did not want a competitor supplying their code, so they created the Open Source Foundation as a competing organization.

In addition to the UNIX variants from companies, public domain versions also proliferated. The BSD team at Berkeley rewrote substantial portions of UNIX so that they could distribute it without needing a license from AT&T. This eventually led to a lawsuit, which Berkeley won. BSD UNIX soon split into FreeBSD, NetBSD, and OpenBSD, provided by competing camps of developers. Apple's current operating system, OS X, is based on Free BSD.

Let's go back to Berkeley Computer Corporation. Alas, this effort was not commercially viable. About the same time as BCC was getting in trouble, Xerox hired Robert Taylor to build the computer science division of the new Xerox Palo Alto Research Center (PARC) in 1970. He had just returned from a tour of duty at ARPA, where he had funded the Berkeley research. He recruited Deutsch, Lampson, and Thacker from BCC to form the core of PARC's team: 11 of the initial 20 employees were from BCC, and they decided to build small computers for individuals rather than large computers for groups. This first personal computer, called the Alto, was built from the same technology as minicomputers, but it had a keyboard, mouse, graphical display, and windows. It popularized windows and led to many inventions, including client-server computing, the Ethernet, and print servers. It directly inspired the Macintosh, which was the successor to the popular Apple II.

IBM had long been interested in selling to the home, so the success of the Apple II led IBM to start a competing project. In contrast to its tradition, for this project IBM designed everything from components outside of the company. They selected the new 16-bit microprocessor from Intel, the 8086. (To lower costs, they started with the version with the 8-bit bus, called the 8088.) They visited Microsoft to see if this small company would be willing to sell their popular Basic interpreter and asked for recommendations

for an operating system. Gates volunteered that Microsoft could deliver both an interpreter and an operating system, as long as they were paid a royalty fee of between $10 and $50 for each copy rather than a flat fee. IBM agreed, provided Microsoft could meet their deadlines. Microsoft didn't have an operating system, nor the time and resources to build one, but Gates knew that a Seattle company had developed an operating system for the Intel 8086. Microsoft purchased QDOS (Quick and Dirty Operating System) for $15,000, made a small change and relabeled it MS-DOS. MS-DOS was a simple operating system without any modern features—no protection, no processes, and no virtual memory—in part because they believed it wasn't necessary for a personal computer.

Announced in 1980, the IBM PC became a tremendous success for IBM and the companies it relied upon. Microsoft sold 500,000 copies of MS-DOS by 1983, and the $10 million income allowed Microsoft to start new software projects.

After seeing a version of the Macintosh under development, Microsoft hired some people from PARC to lead its reply. The Macintosh was announced in 1984, and Windows was available on PCs the following year. It was originally an application that ran on top of DOS, but was later integrated with DOS and renamed Windows 2.0. Microsoft hired Cutler from DEC to lead the development of Windows NT, a new operating system. NT was a modern operating system with protection, processors, and so on and has much in common with DEC's VMS. Today's PC operating systems are more sophisticated than any of the timesharing systems of 20 years ago, yet they still suffer from the need to maintain compatibility with the crippled first PC operating systems such as MS-DOS.

The popularity of the PC led to a desire for a UNIX that ran on it. Many tried to develop one, but the most successful was written from scratch in 1991 by Linus Torvalds. In addition to making the source code available, like BSD, he allowed everyone to make changes and submit them for inclusion in his next release. Linux popularized open source development as we know it today, with such software getting hundreds of volunteers to test releases and add new features.

Many people in this story won awards for their roles in the development of modern operating systems. McCarthy received an

ACM Turing Award in 1971 in part for his contributions to timesharing. In 1983, Thompson and Ritchie received one for UNIX. The announcement said that "the genius of the UNIX system is its framework, which enables programmers to stand on the work of others." In 1990, Corbato received the Turing Award for his contributions to CTSS and MULTICS. Two years later, Lampson won it in part for his work on personal computing and operating systems.

Further Reading

Brooks, F.P. [1975]. *The mythical man-month*. Reading: Addison-Wesley

The classic book that explains the challenge of software engineering using IBM OS development as the example.

Cantin, J.F. and M.D. Hill [2001]. "Cache performance for selected SPEC CPU2000 benchmarks", *SIGARCH Computer Architecture News* 29:4 þ (September), 13–18.

A reference paper of cache miss rates for many cache sizes for the SPEC2000 benchmarks.

Chen, P.M., E.K. Lee, G.A. Gibson, R.H. Katz, and D.A. Patterson [1994]. "RAID: High-performance, reliable secondary storage", *ACM Computing Surveys* 26:2 (June) 145–88.

A tutorial covering disk arrays and the advantages of such an organization.

Conti, C., D.H. Gibson, and S.H. Pitowsky [1968]. "Structural aspects of the System/360 Model 85, part I: General organization", *IBM Systems J.* 7:1, 2–14.

A classic paper that describes the first commercial computer to use a cache and its resulting performance.

Hennessy, J. and D. Patterson [2003]. Chapter 5 in *Computer Architecture: A Quantitative Approach*, third edition, Morgan Kaufmann Publishers, San Francisco.

For more in-depth coverage of a variety of topics including protection, cache performance of out-of-order processors, virtually addressed caches, multilevel caches, compiler optimizations, additional latency tolerance mechanisms, and cache coherency.

Kilburn, T., D.B.G. Edwards, M.J. Lanigan, and F.H. Sumner [1962]. "One-level storage system", *IRE Transactions on Electronic*

Computers EC-11(April), 223–335. Also appears in D.P. Siewiorek, C.G. Bell, and A. Newell [1982], Computer Structures: Principles and Examples, McGraw-Hill, New York, 135–48.

This classic paper is the first proposal for virtual memory.

LaMarca, A. and R.E. Ladner [1996]. "The influence of caches on the performance of heaps", *ACM J. of Experimental Algorithmics*, Vol. 1.

This paper shows the difference between complexity analysis of an algorithm, instruction count performance, and memory hierarchy for four sorting algorithms.

McCalpin, J.D. [1995]. *"STREAM: Sustainable Memory Bandwidth in High Performance Computers"*, https://www.cs.virginia.edu/stream/.

A widely used microbenchmark that measures the performance of the memory system behind the caches.

Patterson, D., G. Gibson, and R. Katz [1988]. *"A case for redundant arrays of inexpensive disks (RAID)"*, SIGMOD Conference, 109–116.

A classic paper that advocates arrays of smaller disks and introduces RAID levels.

Przybylski, S.A. [1990]. *Cache and Memory Hierarchy Design: A Performance-Directed Approach*, Morgan Kaufmann Publishers, San Francisco.

A thorough exploration of multilevel memory hierarchies and their performance.

Ritchie, D. [1984]. "The evolution of the UNIX time-sharing system", *AT&T Bell Laboratories Technical Journal* 1984, 1577–1593.

The history of UNIX from one of its inventors.

Ritchie, D.M. and K. Thompson [1978]. "The UNIX time-sharing system", *Bell System Technical Journal (August)*, 1991–2019.

A paper describing the most elegant operating system ever invented.

Silberschatz, A., P. Galvin, and G. Grange [2003]. *Operating System Concepts*, sixth edition, Addison-Wesley, Reading, MA.

An operating systems textbook with a thorough discussion of virtual memory, processes and process management, and protection issues.

Smith, A.J. [1982]. "Cache memories," *Computing Surveys* 14:3 (September), 473–530.

The classic survey paper on caches. This paper defined the terminology for the field and has served as a reference for many computer designers.

Smith, D.K. and R.C. Alexander [1988]. *Fumbling the Future: How*

Xerox Invented, Then Ignored, the First Personal Computer, Morrow, New York.

A popular book that explains the role of Xerox PARC in laying the foundation for today's computing, but which Xerox did not substantially benefit from.

Tanenbaum, A. [2001]. *Modern Operating Systems*, second edition, Upper Saddle River, Prentice Hall, NJ.

An operating system textbook with a good discussion of virtual memory.

Wilkes, M. [1965]. "Slave memories and dynamic storage allocation", *IEEE Trans. Electronic Computers* EC-14:2 (April), 270–71.

The first classic paper on caches.

5.19 Exercises

Assume memory is byte addressable and words are 64 bits, unless specified otherwise.

5.1 In this exercise we look at memory locality properties of matrix computation. The following code is written in C, where elements within the same row are stored contiguously. Assume each word is a 64-bit integer.

```
for (I=0; I<8; I++)
 for (J=0; J<8000; J++)
  A[I][J]=B[I][0]+A[J][I];
```

5.1.1 [5] <§5.1> How many 64-bit integers can be stored in a 16-byte cache block?

5.1.2 [5] <§5.1> Which variable references exhibit temporal locality?

5.1.3 [5] <§5.1> Which variable references exhibit spatial locality? Locality is affected by both the reference order and data layout. The same computation can also be written below in Matlab, which differs from C in that it stores matrix elements within the same column contiguously in memory.

```
for I=1:8
 for J=1:8000
  A(I,J)=B(I,0)+A(J,I);
 end
end
```

5.1.4 [5] <§5.1> Which variable references exhibit temporal

locality?

5.1.5 [5] <§5.1> Which variable references exhibit spatial locality?

5.1.6 [15] <§5.1> How many 16-byte cache blocks are needed to store all 64-bit matrix elements being referenced using Matlab's matrix storage? How many using C's matrix storage? (Assume each row contains more than one element.)

5.2 Caches are important to providing a high-performance memory hierarchy to processors. Below is a list of 64-bit memory address references, given as word addresses.

`0x03, 0xb4, 0x2b, 0x02, 0xbf, 0x58, 0xbe, 0x0e, 0xb5, 0x2c, 0xba, 0xfd`

5.2.1 [10] <§5.3> For each of these references, identify the binary word address, the tag, and the index given a direct-mapped cache with 16 one-word blocks. Also list whether each reference is a hit or a miss, assuming the cache is initially empty.

5.2.2 [10] <§5.3> For each of these references, identify the binary word address, the tag, the index, and the offset given a direct-mapped cache with two-word blocks and a total size of eight blocks. Also list if each reference is a hit or a miss, assuming the cache is initially empty.

5.2.3 [20] <§§5.3, 5.4> You are asked to optimize a cache design for the given references. There are three direct-mapped cache designs possible, all with a total of eight words of data:

- C1 has 1-word blocks,
- C2 has 2-word blocks, and
- C3 has 4-word blocks.

5.3 By convention, a cache is named according to the amount of data it contains (i.e., a 4 KiB cache can hold 4 KiB of data); however, caches also require SRAM to store metadata such as tags and valid bits. For this exercise, you will examine how a cache's configuration affects the total amount of SRAM needed to implement it as well as the performance of the cache. For all parts, assume that the caches are byte addressable, and that addresses and words are 64 bits.

5.3.1 [10] <§5.3> Calculate the total number of bits required to implement a 32 KiB cache with two-word blocks.

5.3.2 [10] <§5.3> Calculate the total number of bits required to implement a 64 KiB cache with 16-word blocks. How much

bigger is this cache than the 32 KiB cache described in Exercise 5.3.1? (Notice that, by changing the block size, we doubled the amount of data without doubling the total size of the cache.)

5.3.3 [5] <§5.3> Explain why this 64 KiB cache, despite its larger data size, might provide slower performance than the first cache.

5.3.4 [10] <§§5.3, 5.4> Generate a series of read requests that have a lower miss rate on a 32 KiB two-way set associative cache than on the cache described in Exercise 5.3.1.

5.4 [15] <§5.3> Section 5.3 shows the typical method to index a direct-mapped cache, specifically (Block address) modulo (Number of blocks in the cache). Assuming a 64-bit address and 1024 blocks in the cache, consider a different indexing function, specifically (Block address[63:54] XOR Block address[53:44]). Is it possible to use this to index a direct-mapped cache? If so, explain why and discuss any changes that might need to be made to the cache. If it is not possible, explain why.

5.5 For a direct-mapped cache design with a 64-bit address, the following bits of the address are used to access the cache.

Tag	Index	Offset
63–10	9–5	4–0

5.5.1 [5] <§5.3> What is the cache block size (in words)?

5.5.2 [5] <§5.3> How many blocks does the cache have?

5.5.3 [5] <§5.3> What is the ratio between total bits required for such a cache implementation over the data storage bits?

Beginning from power on, the following byte-addressed cache references are recorded.

Address											
Hex	00	04	10	84	E8	A0	400	1E	8C	C1C	B4	884
Dec	0	4	16	132	232	160	1024	30	140	3100	180	2180

5.5.4 [20] <§5.3> For each reference, list (1) its tag, index, and offset, (2) whether it is a hit or a miss, and (3) which bytes were replaced (if any).

5.5.5 [5] <§5.3> What is the hit ratio?

5.5.6 [5] <§5.3> List the final state of the cache, with each valid

entry represented as a record of <index, tag, data>. For example,

```
<0, 3, Mem[0xC00]-Mem[0xC1F]>
```

5.6 Recall that we have two write policies and two write allocation policies, and their combinations can be implemented either in L1 or L2 cache. Assume the following choices for L1 and L2 caches:

L1	L2
Write through, non-write allocate	Write back, write allocate

5.6.1 [5] <§§5.3, 5.8> Buffers are employed between different levels of memory hierarchy to reduce access latency. For this given configuration, list the possible buffers needed between L1 and L2 caches, as well as L2 cache and memory.

5.6.2 [20] <§§5.3, 5.8> Describe the procedure of handling an L1 write-miss, considering the components involved and the possibility of replacing a dirty block.

5.6.3 [20] <§§5.3, 5.8> For a multilevel exclusive cache configuration (a block can only reside in one of the L1 and L2 caches), describe the procedures of handling an L1 write-miss and an L1 read-miss, considering the components involved and the possibility of replacing a dirty block.

5.7 Consider the following program and cache behaviors.

Data Reads per 1000 Instructions	Data Writes per 1000 Instructions	Instruction Cache Miss Rate	Data Cache Miss Rate	Block Size (bytes)
250	100	0.30%	2%	64

5.7.1 [10] <§§5.3, 5.8> Suppose a CPU with a write-through, write-allocate cache achieves a CPI of 2. What are the read and write bandwidths (measured by bytes per cycle) between RAM and the cache? (Assume each miss generates a request for one block.)

5.7.2 [10] <§§5.3, 5.8> For a write-back, write-allocate cache, assuming 30% of replaced data cache blocks are dirty, what are the read and write bandwidths needed for a CPI of 2?

5.8 Media applications that play audio or video files are part of a class of workloads called “streaming” workloads (i.e., they bring in large amounts of data but do not reuse much of it). Consider a video streaming workload that accesses a 512 KiB working set

sequentially with the following word address stream:
`0, 1, 2, 3, 4, 5, 6, 7, 8, 9 …`

5.8.1 [10] <§§5.4, 5.8> Assume a 64 KiB direct-mapped cache with a 32-byte block. What is the miss rate for the address stream above? How is this miss rate sensitive to the size of the cache or the working set? How would you categorize the misses this workload is experiencing, based on the 3C model?

5.8.2 [5] <§§5.1, 5.8> Re-compute the miss rate when the cache block size is 16 bytes, 64 bytes, and 128 bytes. What kind of locality is this workload exploiting?

5.8.3 [10] <§5.13> "Prefetching" is a technique that leverages predictable address patterns to speculatively bring in additional cache blocks when a particular cache block is accessed. One example of prefetching is a stream buffer that prefetches sequentially adjacent cache blocks into a separate buffer when a particular cache block is brought in. If the data are found in the prefetch buffer, it is considered as a hit, moved into the cache, and the next cache block is prefetched. Assume a two-entry stream buffer; and, assume that the cache latency is such that a cache block can be loaded before the computation on the previous cache block is completed. What is the miss rate for the address stream above?

5.9 Cache block size (B) can affect both miss rate and miss latency. Assuming a machine with a base CPI of 1, and an average of 1.35 references (both instruction and data) per instruction, find the block size that minimizes the total miss latency given the following miss rates for various block sizes.

8: 4%	16: 3%	32: 2%	64: 1.5%	128: 1%

5.9.1 [10] <§5.3> What is the optimal block size for a miss latency of 20 ×B cycles?

5.9.2 [10] <§5.3> What is the optimal block size for a miss latency of 24 +B cycles?

5.9.3 [10] <§5.3> For constant miss latency, what is the optimal block size?

5.10 In this exercise, we will look at the different ways capacity affects overall performance. In general, cache access time is

proportional to capacity. Assume that main memory accesses take 70 ns and that 36% of all instructions access data memory. The following table shows data for L1 caches attached to each of two processors, P1 and P2.

	L1 Size	L1 Miss Rate	L1 Hit Time
P1	2 KiB	8.0%	0.66 ns
P2	4 KiB	6.0%	0.90 ns

5.10.1 [5] <§5.4> Assuming that the L1 hit time determines the cycle times for P1 and P2, what are their respective clock rates?

5.10.2 [10] <§5.4> What is the Average Memory Access Time for P1 and P2 (in cycles)?

5.10.3 [5] <§5.4> Assuming a base CPI of 1.0 without any memory stalls, what is the total CPI for P1 and P2? Which processor is faster? (When we say a "base CPI of 1.0", we mean that instructions complete in one cycle, unless either the instruction access or the data access causes a cache miss.)

For the next three problems, we will consider the addition of an L2 cache to P1 (to presumably make up for its limited L1 cache capacity). Use the L1 cache capacities and hit times from the previous table when solving these problems. The L2 miss rate indicated is its local miss rate.

L2 Size	L2 Miss Rate	L2 Hit Time
1 MiB	95%	5.62 ns

5.10.4 [10] <§5.4> What is the AMAT for P1 with the addition of an L2 cache? Is the AMAT better or worse with the L2 cache?

5.10.5 [5] <§5.4> Assuming a base CPI of 1.0 without any memory stalls, what is the total CPI for P1 with the addition of an L2 cache?

5.10.6 [10] <§5.4> What would the L2 miss rate need to be in order for P1 with an L2 cache to be faster than P1 without an L2 cache?

5.10.7 [15] <§5.4> What would the L2 miss rate need to be in order for P1 with an L2 cache to be faster than P2 without an L2 cache?

5.11 This exercise examines the effect of different cache designs, specifically comparing associative caches to the direct-mapped caches from Section 5.4. For these exercises, refer to the sequence of word address shown below.

```
0x03, 0xb4, 0x2b, 0x02, 0xbe, 0x58, 0xbf, 0x0e, 0x1f, 0xb5,
0xbf, 0xba, 0x2e, 0xce
```

5.11.1 [10] <§5.4> Sketch the organization of a three-way set associative cache with two-word blocks and a total size of 48 words. Your sketch should have a style similar to Figure 5.18, but clearly show the width of the tag and data fields.

5.11.2 [10] <§5.4> Trace the behavior of the cache from Exercise 5.11.1. Assume a true LRU replacement policy. For each reference, identify

- the binary word address,
- the tag,
- the index,
- the offset
- whether the reference is a hit or a miss, and
- which tags are in each way of the cache after the reference has been handled.

5.11.3 [5] <§5.4> Sketch the organization of a fully associative cache with one-word blocks and a total size of eight words. Your sketch should have a style similar to Figure 5.18, but clearly show the width of the tag and data fields.

5.11.4 [10] <§5.4> Trace the behavior of the cache from Exercise 5.11.3. Assume a true LRU replacement policy. For each reference, identify

- the binary word address,
- the tag,
- the index,
- the offset,
- whether the reference is a hit or a miss, and
- the contents of the cache after each reference has been handled.

5.11.5 [5] <§5.4> Sketch the organization of a fully associative cache with two-word blocks and a total size of eight words.

Your sketch should have a style similar to Figure 5.18, but clearly show the width of the tag and data fields.

5.11.6 [10] <§5.4> Trace the behavior of the cache from Exercise 5.11.5. Assume an LRU replacement policy. For each reference, identify

- the binary word address,
- the tag,
- the index,
- the offset,
- whether the reference is a hit or a miss, and
- the contents of the cache after each reference has been handled.

5.11.7 [10] <§5.4> Repeat Exercise 5.11.6 using MRU (*most recently used*) replacement.

5.11.8 [15] <§5.4> Repeat Exercise 5.11.6 using the optimal replacement policy (i.e., the one that gives the lowest miss rate).

5.12 Multilevel caching is an important technique to overcome the limited amount of space that a first-level cache can provide while still maintaining its speed. Consider a processor with the following parameters:

Base CPI, No Memory Stalls	Processor Speed	Main Memory Access Time	First-Level Cache Miss Rate per Instruction**	Second-Level Cache, Direct-Mapped Speed	Miss Rate with Second-Level Cache, Direct-Mapped	Second-Level Cache, Eight-Way Set Associative Speed	Miss Rate with Second-Level Cache, Eight-Way Set Associative
1.5	2 GHz	100 ns	7%	12 cycles	3.5%	28 cycles	1.5%

**First Level Cache miss rate is per instruction. Assume the total number of L1 cache misses (instruction and data combined) is equal to 7% of the number of instructions.

5.12.1 [10] <§5.4> Calculate the CPI for the processor in the table using: 1) only a first-level cache, 2) a second-level direct-mapped cache, and 3) a second-level eight-way set associative cache. How do these numbers change if main memory access time doubles? (Give each change as both an absolute CPI and a percent change.) Notice the extent to which an L2 cache can hide the effects of a slow memory.

5.12.2 [10] <§5.4> It is possible to have an even greater cache hierarchy than two levels? Given the processor above with a second-level, direct-mapped cache, a designer wants to add a third-level cache that takes 50 cycles to access and will have a 13% miss rate. Would this provide better performance? In general, what are the advantages and disadvantages of adding

a third-level cache?

5.12.3 [20] <§5.4> In older processors, such as the Intel Pentium or Alpha 21264, the second level of cache was external (located on a different chip) from the main processor and the first-level cache. While this allowed for large second-level caches, the latency to access the cache was much higher, and the bandwidth was typically lower because the second-level cache ran at a lower frequency. Assume a 512 KiB off-chip second-level cache has a miss rate of 4%. If each additional 512 KiB of cache lowered miss rates by 0.7%, and the cache had a total access time of 50 cycles, how big would the cache have to be to match the performance of the second-level direct-mapped cache listed above?

5.13 *Mean time between failures* (MTBF), *mean time to replacement* (MTTR), and *mean time to failure* (MTTF) are useful metrics for evaluating the reliability and availability of a storage resource. Explore these concepts by answering the questions about a device with the following metrics:

MTTF	MTTR
3 Years	1 Day

5.13.1 [5] <§5.5> Calculate the MTBF for such a device.

5.13.2 [5] <§5.5> Calculate the availability for such a device.

5.13.3 [5] <§5.5> What happens to availability as the MTTR approaches 0? Is this a realistic situation?

5.13.4 [5] <§5.5> What happens to availability as the MTTR gets very high, i.e., a device is difficult to repair? Does this imply the device has low availability?

5.14 This exercise examines the *single error correcting, double error detecting* (SEC/DED) Hamming code.

5.14.1 [5] <§5.5> What is the minimum number of parity bits required to protect a 128-bit word using the SEC/DED code?

5.14.2 [5] <§5.5> Section 5.5 states that modern server memory modules (DIMMs) employ SEC/DED ECC to protect each 64 bits with 8 parity bits. Compute the cost/performance ratio of this code to the code from Exercise 5.14.1. In this case, cost is the relative number of parity bits needed while performance is the relative number of errors that can be corrected. Which is better?

5.14.3 [5] <§5.5> Consider a SEC code that protects 8 bit words with 4 parity bits. If we read the value 0x375, is there an error? If so, correct the error.

5.15 For a high-performance system such as a B-tree index for a database, the page size is determined mainly by the data size and disk performance. Assume that, on average, a B-tree index page is 70% full with fix-sized entries. The utility of a page is its B-tree depth, calculated as $\log_2$(entries). The following table shows that for 16-byte entries, and a 10-year-old disk with a 10 ms latency and 10 MB/s transfer rate, the optimal page size is 16 K.

Page Size (KiB)	Page Utility or B-Tree Depth (Number of Disk Accesses Saved)	Index Page Access Cost (ms)	Utility/Cost
2	6.49 (or $\log_2(2048/16 \times 0.7)$)	10.2	0.64
4	7.49	10.4	0.72
8	8.49	10.8	0.79
16	9.49	11.6	0.82
32	10.49	13.2	0.79
64	11.49	16.4	0.70
128	12.49	22.8	0.55
256	13.49	35.6	0.38

5.15.1 [10] <§5.7> What is the best page size if entries now become 128 bytes?

5.15.2 [10] <§5.7> Based on Exercise 5.15.1, what is the best page size if pages are half full?

5.15.3 [20] <§5.7> Based on Exercise 5.15.2, what is the best page size if using a modern disk with a 3 ms latency and 100 MB/s transfer rate? Explain why future servers are likely to have larger pages.

Keeping “frequently used” (or “hot”) pages in DRAM can save disk accesses, but how do we determine the exact meaning of “frequently used” for a given system? Data engineers use the cost ratio between DRAM and disk access to quantify the reuse time threshold for hot pages. The cost of a disk access is $Disk/accesses_per_sec, while the cost to keep a page in DRAM is $DRAM_MiB/page_size. The typical DRAM and disk costs and typical database page sizes at several time points are listed below:

Year	DRAM Cost ($/MiB)	Page Size (KiB)	Disk Cost ($/disk)	Disk Access Rate (access/sec)
1987	5000	1	15,000	15
1997	15	8	2000	64
2007	0.05	64	80	83

5.15.4 [20] <§5.7> What other factors can be changed to keep using the same page size (thus avoiding software rewrite)? Discuss their likeliness with current technology and cost trends.

5.16 As described in Section 5.7, virtual memory uses a page table to track the mapping of virtual addresses to physical addresses. This exercise shows how this table must be updated as addresses are accessed. The following data constitute a stream of virtual byte addresses as seen on a system. Assume 4 KiB pages, a four-entry fully associative TLB, and true LRU replacement. If pages must be brought in from disk, increment the next largest page number.

Decimal	4669	2227	13916	34587	48870	12608	49225
hex	0x123d	0x08b3	0x365c	0x871b	0xbee6	0x3140	0xc049

TLB

Valid	Tag	Physical Page Number	Time Since Last Access
1	0xb	12	4
1	0x7	4	1
1	0x3	6	3
0	0x4	9	7

Page table

Index	Valid	Physical Page or in Disk

0	1	5
1	0	Disk
2	0	Disk
3	1	6
4	1	9
5	1	11
6	0	Disk
7	1	4
8	0	Disk
9	0	Disk
a	1	3
b	1	12

5.16.1 [10] <§5.7> For each access shown above, list
- whether the access is a hit or miss in the TLB,
- whether the access is a hit or miss in the page table,
- whether the access is a page fault,
- the updated state of the TLB.

5.16.2 [15] <§5.7> Repeat Exercise 5.16.1, but this time use 16 KiB pages instead of 4 KiB pages. What would be some of the advantages of having a larger page size? What are some of the disadvantages?

5.16.3 [15] <§5.7> Repeat Exercise 5.16.1, but this time use 4 KiB pages and a two-way set associative TLB.

5.16.4 [15] <§5.7> Repeat Exercise 5.16.1, but this time use 4 KiB pages and a direct mapped TLB.

5.16.5 [10] <§§5.4, 5.7> Discuss why a CPU must have a TLB for high performance. How would virtual memory accesses be handled if there were no TLB?

5.17 There are several parameters that affect the overall size of the page table. Listed below are key page table parameters.

Virtual Address Size	Page Size	Page Table Entry Size
32 bits	8 KiB	4 bytes

5.17.1 [5] <§5.7> Given the parameters shown above, calculate the maximum possible page table size for a system running five processes.

5.17.2 [10] <§5.7> Given the parameters shown above, calculate the total page table size for a system running five applications that each utilize half of the virtual memory available, given a two-level page table approach with up to 256 entries at the 1st level. Assume each entry of the main page table is 6 bytes.

Calculate the minimum and maximum amount of memory required for this page table.

5.17.3 [10] <§5.7> A cache designer wants to increase the size of a 4 KiB virtually indexed, physically tagged cache. Given the page size shown above, is it possible to make a 16 KiB direct-mapped cache, assuming two 64-bit words per block? How would the designer increase the data size of the cache?

5.18 In this exercise, we will examine space/time optimizations for page tables. The following list provides parameters of a virtual memory system.

Virtual Address (bits)	Physical DRAM Installed	Page Size	PTE Size (byte)
43	16 GiB	4 KiB	4

5.18.1 [10] <§5.7> For a single-level page table, how many *page table entries* (PTEs) are needed? How much physical memory is needed for storing the page table?

5.18.2 [10] <§5.7> Using a multi-level page table can reduce the physical memory consumption of page tables by only keeping active PTEs in physical memory. How many levels of page tables will be needed if the segment tables (the upper-level page tables) are allowed to be of unlimited size? How many memory references are needed for address translation if missing in TLB?

5.18.3 [10] <§5.7> Suppose the segments are limited to the 4 KiB page size (so that they can be paged). Is 4 bytes large enough for all page table entries (including those in the segment tables?

5.18.4 [10] <§5.7> How many levels of page tables are needed if the segments are limited to the 4 KiB page size?

5.18.5 [15] <§5.7> An inverted page table can be used to further optimize space and time. How many PTEs are needed to store the page table? Assuming a hash table implementation, what are the common case and worst case numbers of memory references needed for servicing a TLB miss?

5.19 The following table shows the contents of a four-entry TLB.

Entry-ID	Valid	VA Page	Modified	Protection	PA Page
1	1	140	1	RW	30
2	0	40	0	RX	34
3	1	200	1	RO	32
4	1	280	0	RW	31

5.19.1 [5] <§5.7> Under what scenarios would entry 3's valid bit be set to zero?

5.19.2 [5] <§5.7> What happens when an instruction writes to VA page 30? When would a software managed TLB be faster than a hardware managed TLB?

5.19.3 [5] <§5.7> What happens when an instruction writes to VA page 200?

5.20 In this exercise, we will examine how replacement policies affect miss rate. Assume a two-way set associative cache with four one-word blocks. Consider the following word address sequence: 0, 1, 2, 3, 4, 2, 3, 4, 5, 6, 7, 0, 1, 2, 3, 4, 5, 6, 7, 0. Consider the following address sequence: 0, 2, 4, 8, 10, 12, 14, 16, 0

5.20.1 [5] <§§5.4, 5.8> Assuming an LRU replacement policy, which accesses are hits?

5.20.2 [5] <§§5.4, 5.8> Assuming an MRU (*most recently used*) replacement policy, which accesses are hits?

5.20.3 [5] <§§5.4, 5.8> Simulate a random replacement policy by flipping a coin. For example, "heads" means to evict the first block in a set and "tails" means to evict the second block in a set. How many hits does this address sequence exhibit?

5.20.4 [10] <§§5.4, 5.8> Describe an optimal replacement policy for this sequence. Which accesses are hits using this policy?

5.20.5 [10] <§§5.4, 5.8> Describe why it is difficult to implement a cache replacement policy that is optimal for all address sequences.

5.20.6 [10] <§§5.4, 5.8> Assume you could make a decision upon each memory reference whether or not you want the requested

address to be cached. What effect could this have on miss rate?

5.21 One of the biggest impediments to widespread use of virtual machines is the performance overhead incurred by running a virtual machine. Listed below are various performance parameters and application behavior.

Base CPI	Privileged O/S accesses per 10,000 instructions	Overhead to trap to the guest O/S	Overhead to trap to VMM	I/O access per 10,000 instructions	I/O access time (includes time to trap to guest O/S)
1.5	120	15 cycles	175 cycles	30	1100 cycles

5.21.1 [10] <§5.6> Calculate the CPI for the system listed above assuming that there are no accesses to I/O. What is the CPI if the VMM overhead doubles? If it is cut in half? If a virtual machine software company wishes to limit the performance degradation to 10%, what is the longest possible penalty to trap to the VMM?

5.21.2 [15] <§5.6> I/O accesses often have a large effect on overall system performance. Calculate the CPI of a machine using the performance characteristics above, assuming a non-virtualized system. Calculate the CPI again, this time using a virtualized system. How do these CPIs change if the system has half the I/O accesses?

5.22 [15] <§§5.6, 5.7> Compare and contrast the ideas of virtual memory and virtual machines. How do the goals of each compare? What are the pros and cons of each? List a few cases where virtual memory is desired, and a few cases where virtual machines are desired.

5.23 [10] <§5.6> Section 5.6 discusses virtualization under the assumption that the virtualized system is running the same ISA as the underlying hardware. However, one possible use of virtualization is to emulate non-native ISAs. An example of this is QEMU, which emulates a variety of ISAs such as MIPS, SPARC, and PowerPC. What are some of the difficulties involved in this kind of virtualization? Is it possible for an emulated system to run faster than on its native ISA?

5.24 In this exercise, we will explore the control unit for a cache controller for a processor with a write buffer. Use the finite state machine found in Figure 5.39 as a starting point for designing your own finite state machines. Assume that the cache controller is for the simple direct-mapped cache described on page 453 (Figure 5.39 in Section 5.9), but you will add a write buffer with a

capacity of one block.

Recall that the purpose of a write buffer is to serve as temporary storage so that the processor doesn't have to wait for two memory accesses on a dirty miss. Rather than writing back the dirty block before reading the new block, it buffers the dirty block and immediately begins reading the new block. The dirty block can then be written to main memory while the processor is working.

5.24.1 [10] <§§5.8, 5.9> What should happen if the processor issues a request that *hits* in the cache while a block is being written back to main memory from the write buffer?

5.24.2 [10] <§§5.8, 5.9> What should happen if the processor issues a request that *misses* in the cache while a block is being written back to main memory from the write buffer?

5.24.3 [30] <§§5.8, 5.9> Design a finite state machine to enable the use of a write buffer.

5.25 Cache coherence concerns the views of multiple processors on a given cache block. The following data show two processors and their read/write operations on two different words of a cache block X (initially X[0]=X[1]=0).

P1	P2
`X[0] ++; X[1] = 3;`	`X[0] = 5; X[1] +=2;`

5.25.1 [15] <§5.10> List the possible values of the given cache block for a correct cache coherence protocol implementation. List at least one more possible value of the block if the protocol doesn't ensure cache coherency.

5.25.2 [15] <§5.10> For a snooping protocol, list a valid operation sequence on each processor/cache to finish the above read/write operations.

5.25.3 [10] <§5.10> What are the best-case and worst-case numbers of cache misses needed to execute the listed read/write instructions?

Memory consistency concerns the views of multiple data items. The following data show two processors and their read/write operations on different cache blocks (A and B initially 0).

P1	P2
`A = 1; B = 2; A+=2; B++;`	`C = B; D = A;`

5.25.4 [15] <§5.10> List the possible values of C and D for all

implementations that ensure both consistency assumptions on page 455.

5.25.5 [15] <§5.10> List at least one more possible pair of values for C and D if such assumptions are not maintained.

5.25.6 [15] <§§5.3, 5.10> For various combinations of write policies and write allocation policies, which combinations make the protocol implementation simpler?

5.26 Chip multiprocessors (CMPs) have multiple cores and their caches on a single chip. CMP on-chip L2 cache design has interesting trade-offs. The following table shows the miss rates and hit latencies for two benchmarks with private vs. shared L2 cache designs. Assume the L1 cache has a 3% miss rate and a 1-cycle access time.

	Private	Shared
Benchmark A miss rate	10%	4%
Benchmark B miss rate	2%	1%

Assume the following hit latencies:

Private Cache	Shared Cache	Memory
5	20	180

5.26.1 [15] <§5.13> Which cache design is better for each of these benchmarks? Use data to support your conclusion.

5.26.2 [15] <§5.13> Off-chip bandwidth becomes the bottleneck as the number of CMP cores increases. How does this bottleneck affect private and shared cache systems differently? Choose the best design if the latency of the first off-chip link doubles.

5.26.3 [10] <§5.13> Discuss the pros and cons of shared vs. private L2 caches for both single-threaded, multi-threaded, and multiprogrammed workloads, and reconsider them if having on-chip L3 caches.

5.26.4 [10] <§5.13> Would a non-blocking L2 cache produce more improvement on a CMP with a shared L2 cache or a private L2 cache? Why?

5.26.5 [10] <§5.13> Assume new generations of processors double the number of cores every 18 months. To maintain the same level of per-core performance, how much more off-chip memory bandwidth is needed for a processor released in three

years?

5.26.6 [15] <§5.13> Consider the entire memory hierarchy. What kinds of optimizations can improve the number of concurrent misses?

5.27 In this exercise we show the definition of a web server log and examine code optimizations to improve log processing speed. The data structure for the log is defined as follows:

```
struct entry {
 int srcIP; // remote IP address
 char URL[128]; // request URL (e.g., "GET index.html")
 long long refTime; // reference time
 int status; // connection status
 char browser[64]; // client browser name
} log [NUM_ENTRIES];
```

Assume the following processing function for the log:

```
topK_sourceIP (int hour);
```

This function determines the most frequently observed source IPs during the given hour.

5.27.1 [5] <§5.15> Which fields in a log entry will be accessed for the given log processing function? Assuming 64-byte cache blocks and no prefetching, how many cache misses per entry does the given function incur on average?

5.27.2 [5] <§5.15> How can you reorganize the data structure to improve cache utilization and access locality?

5.27.3 [10] <§5.15> Give an example of another log processing function that would prefer a different data structure layout. If both functions are important, how would you rewrite the program to improve the overall performance? Supplement the discussion with code snippet and data.

5.28 For the problems below, use data from "Cache Performance for SPEC CPU2000 Benchmarks" (http://www.cs.wisc.edu/multifacet/misc/spec2000cache-data/) for the pairs of benchmarks shown in the following table.

a.	Mesa/gcc
b.	mcf/swim

5.28.1 [10] <§5.15> For 64 KiB data caches with varying set associativities, what are the miss rates broken down by miss types (cold, capacity, and conflict misses) for each benchmark?

5.28.2 [10] <§5.15> Select the set associativity to be used by a 64 KiB L1 data cache shared by both benchmarks. If the L1 cache has to be directly mapped, select the set associativity for the 1 MiB L2 cache.

5.28.3 [20] <§5.15> Give an example in the miss rate table where higher set associativity actually increases miss rate. Construct a cache configuration and reference stream to demonstrate this.

5.29 To support multiple virtual machines, two levels of memory virtualization are needed. Each virtual machine still controls the mapping of *virtual address* (VA) to *physical address* (PA), while the hypervisor maps the *physical address* (PA) of each virtual machine to the actual *machine address* (MA). To accelerate such mappings, a software approach called "shadow paging" duplicates each virtual machine's page tables in the hypervisor, and intercepts VA to PA mapping changes to keep both copies consistent. To remove the complexity of shadow page tables, a hardware approach called *nested page table* (NPT) explicitly supports two classes of page tables (VA⇒PA and PA⇒MA) and can walk such tables purely in hardware.

Consider the following sequence of operations: (1) Create process; (2) TLB miss; (3) page fault; (4) context switch;

5.29.1 [10] <§§5.6, 5.7> What would happen for the given operation sequence for shadow page table and nested page table, respectively?

5.29.2 [10] <§§5.6, 5.7> Assuming an x86-based four-level page table in both guest and nested page table, how many memory references are needed to service a TLB miss for native vs. nested page table?

5.29.3 [15] <§§5.6, 5.7> Among TLB miss rate, TLB miss latency, page fault rate, and page fault handler latency, which metrics are more important for shadow page table? Which are important for nested page table?

Assume the following parameters for a shadow paging system.

TLB Misses per 1000 Instructions	NPT TLB Miss Latency	Page Faults per 1000 Instructions	Shadowing Page Fault Overhead
0.2	200 cycles	0.001	30,000 cycles

5.29.4 [10] <§5.6> For a benchmark with native execution CPI of 1, what are the CPI numbers if using shadow page tables vs. NPT (assuming only page table virtualization overhead)?

5.29.5 [10] <§5.6> What techniques can be used to reduce page table shadowing induced overhead?

5.29.6 [10] <§5.6> What techniques can be used to reduce NPT induced overhead?

Answers to Check Yourself

§5.1, page 369: 1 and 4. (3 is false because the cost of the memory hierarchy varies per computer, but in 2016 the highest cost is usually the DRAM.)

§5.3, page 390: 1 and 4: A lower miss penalty can enable smaller blocks, since you don't have that much latency to amortize, yet higher memory bandwidth usually leads to larger blocks, since the miss penalty is only slightly larger.

§5.4, page 409: 1.

§5.8, page 449: 2. (Both large block sizes and prefetching may reduce compulsory misses, so 1 is false.)