

## 7.1 Introduction

From the earliest days of computing, programmers have wanted unlimited amounts of fast memory. The topics we will look at in this chapter aid programmers by creating the illusion of unlimited fast memory. Before we look at how the illusion is actually created, 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 good 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 going back 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** The 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.

- **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 find out about the EDSAC, you also noticed that there was another book shelved next to it about early mechanical computers, so you also brought back that book too and, later on, found something useful in that book. Books on the same topic are shelved together in the library to increase spatial locality. We'll see how spatial locality is used in memory hierarchies a little later in this chapter.

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 high amounts of temporal locality. Since instructions are normally accessed sequentially, programs show high spatial locality. Accesses to data also exhibit a natural spatial locality. For example, accesses to elements of an array or a record will naturally have high degrees of spatial locality.

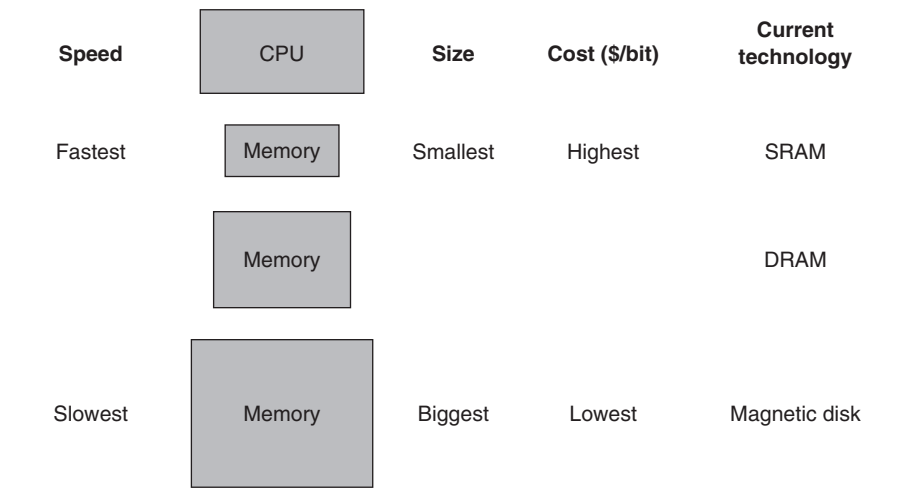
We take advantage of the principle of locality by implementing the memory of a computer as a **memory hierarchy**. A memory hierarchy consists of multiple levels of memory with different speeds and sizes. The faster memories are more expensive per bit than the slower memories and thus smaller.

Today, there are three primary technologies used in building 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 B.8 of [Appendix B](#). The third technology, used to implement the largest and slowest level in the hierarchy, is magnetic disk. The access time and price per bit vary widely among these technologies, as the table below shows, using typical values for 2004:

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

Memory technology	Typical access time	\$ per GB in 2004
SRAM	0.5–5 ns	\$4000–\$10,000
DRAM	50–70 ns	\$100–\$200
Magnetic disk	5,000,000–20,000,000 ns	\$0.50–\$2

Because of these differences in cost and access time, it is advantageous to build memory as a hierarchy of levels. Figure 7.1 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 7.1 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.

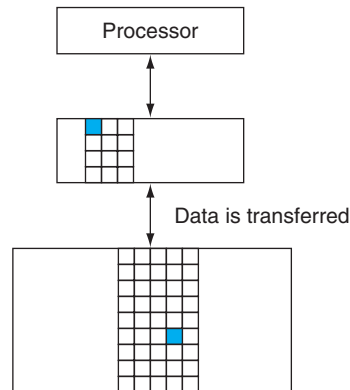
The memory system is organized as a hierarchy: a level closer to the processor is generally a subset of any level further away, and all the data is stored at the lowest level. By analogy, the books on your desk form a subset of the library you are working in, which is in turn a subset of all the libraries 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 is 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 (since it uses more expensive technology) than the lower level. Figure 7.2 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.

If the data requested by the processor appears 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 is 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 perfor-

**block** The minimum unit of information that can be either present or not present in the two-level hierarchy.

**hit rate** The fraction of memory accesses found in a cache.



**FIGURE 7.2** 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*. Usually we transfer an entire block when we copy something between levels.

mance of the memory hierarchy. The **miss rate** ( $1 - \text{hit rate}$ ) is the fraction of memory accesses not found in the upper level.

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.)

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

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

**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.

memory hierarchies to get good performance. We show how important this understanding is with two examples.

Since memory systems are so critical to performance, computer designers have devoted a lot of attention to these systems and developed sophisticated mechanisms for improving the performance of the memory system. In this chapter, we will see the major conceptual ideas, although many simplifications and abstractions have been used to keep the material manageable in length and complexity. We could easily have written hundreds of pages on memory systems, as dozens of recent doctoral theses have demonstrated.

### Check Yourself

Which of the following statements are generally true?

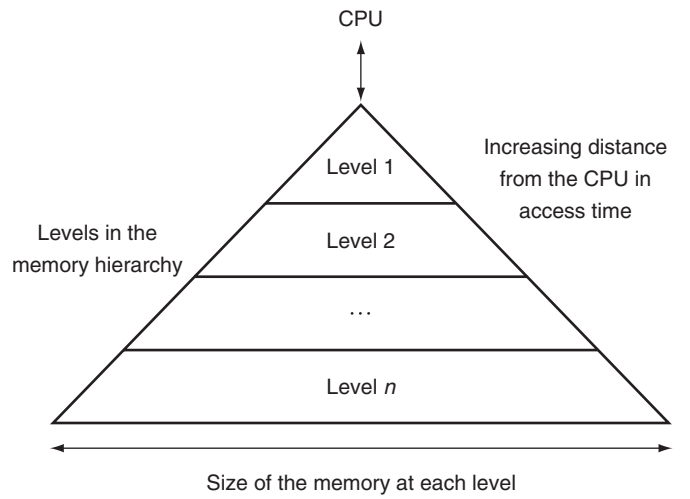
1. Caches take advantage of temporal locality.
2. On a read, the value returned depends on which blocks are in the cache.
3. Most of the cost of the memory hierarchy is at the highest level.

### 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 7.3 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.

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



**FIGURE 7.3** 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.

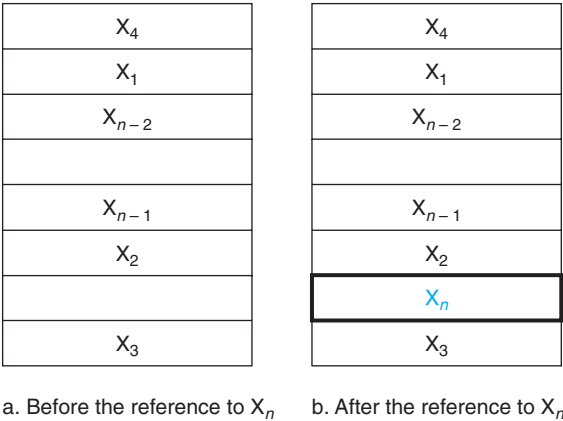
## 7.2 The Basics of Caches

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. 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 today, 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 7.3 on page 492.)

*Cache: a safe place for hiding or storing things.*

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



**FIGURE 7.4** 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.

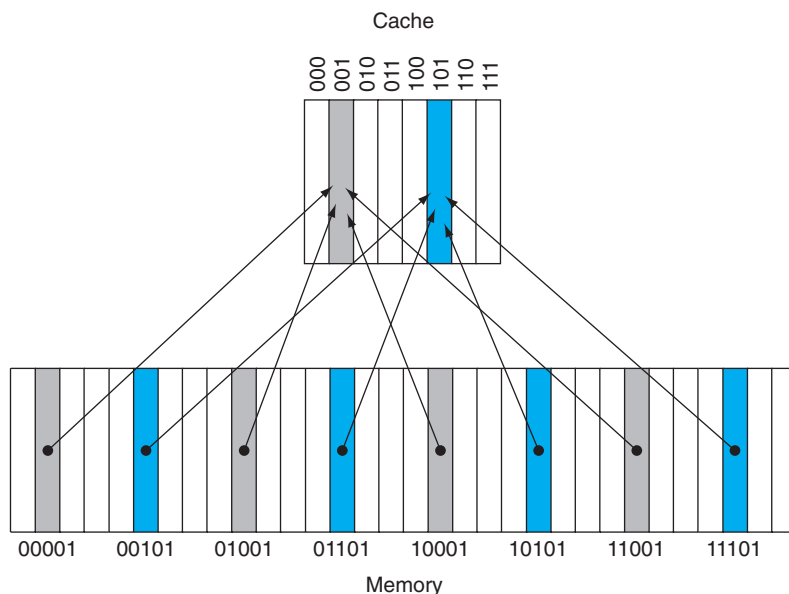
Figure 7.4 shows such a simple cache, before and after requesting a data item that is not initially in the cache. Before the request, the cache contains a collection of recent references  $X_1, X_2, \dots, X_{n-1}$ , and the processor requests a word  $X_n$  that is not in the cache. This request results in a miss, and the word  $X_n$  is brought from memory into the cache.

In looking at the scenario in Figure 7.4, 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 to these two questions 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 the mapping

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

This mapping is attractive because 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; hence the cache may be accessed directly with the low-order bits. For example, Figure 7.5 shows how the memory addresses between

**direct-mapped cache** A cache structure in which each memory location is mapped to exactly one location in the cache.



**FIGURE 7.5 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 cache word  $X \bmod 8$ . That is, the low-order  $\log_2(8) = 3$  bits are used as the cache index. Thus, addresses  $00001_{\text{two}}$ ,  $01001_{\text{two}}$ ,  $10001_{\text{two}}$ , and  $11001_{\text{two}}$  all map to entry  $001_{\text{two}}$  of the cache, while addresses  $00101_{\text{two}}$ ,  $01101_{\text{two}}$ ,  $10101_{\text{two}}$ , and  $11101_{\text{two}}$  all map to entry  $101_{\text{two}}$  of the cache.

$1_{\text{ten}}$  ( $00001_{\text{two}}$ ) and  $29_{\text{ten}}$  ( $11101_{\text{two}}$ ) map to locations  $1_{\text{ten}}$  ( $001_{\text{two}}$ ) and  $5_{\text{ten}}$  ( $101_{\text{two}}$ ) in a direct-mapped cache of eight words.

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 only 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 7.5 we need only have the upper 2 of the 5 address bits in the tag, since the lower 3-bit index field of the address selects the block. We exclude the index bits because they are redundant, since by definition the index field of every address must have the same value.

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,

**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.



**valid bit** A field in the tables of a memory hierarchy that indicates that the associated block in the hierarchy contains valid 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 7.4. 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.

For the rest of this section, we will focus on explaining how reads work in a cache and how the cache control works for 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.

### Accessing a Cache

Figure 7.6 shows the contents of an eight-word direct-mapped cache as it responds to a series of requests from the processor. Since there are eight blocks in the cache, the low-order 3 bits of an address give the block number. Here is the action for each reference:

Decimal address of reference	Binary address of reference	Hit or miss in cache	Assigned cache block (where found or placed)
22	10110 <sub>two</sub>	miss (7.6b)	(10110 <sub>two</sub> mod 8) = 110 <sub>two</sub>
26	11010 <sub>two</sub>	miss (7.6c)	(11010 <sub>two</sub> mod 8) = 010 <sub>two</sub>
22	10110 <sub>two</sub>	hit	(10110 <sub>two</sub> mod 8) = 110 <sub>two</sub>
26	11010 <sub>two</sub>	hit	(11010 <sub>two</sub> mod 8) = 010 <sub>two</sub>
16	10000 <sub>two</sub>	miss (7.6d)	(10000 <sub>two</sub> mod 8) = 000 <sub>two</sub>
3	00011 <sub>two</sub>	miss (7.6e)	(00011 <sub>two</sub> mod 8) = 011 <sub>two</sub>
16	10000 <sub>two</sub>	hit	(10000 <sub>two</sub> mod 8) = 000 <sub>two</sub>
18	10010 <sub>two</sub>	miss (7.6f)	(10010 <sub>two</sub> mod 8) = 010 <sub>two</sub>

When the word at address 18 (10010<sub>two</sub>) is brought into cache block 2 (010<sub>two</sub>), the word at address 26 (11010<sub>two</sub>), which was in cache block 2 (010<sub>two</sub>), must be replaced by the newly requested data. This behavior allows a cache to take advantage of temporal locality: recently accessed 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 only one choice of what to replace.

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 <sub>two</sub>	Memory(10110 <sub>two</sub> )
111	N		

b. After handling a miss of address (10110<sub>two</sub>)

Index	V	Tag	Data
000	N		
001	N		
010	Y	11 <sub>two</sub>	Memory (11010 <sub>two</sub> )
011	N		
100	N		
101	N		
110	Y	10 <sub>two</sub>	Memory (10110 <sub>two</sub> )
111	N		

c. After handling a miss of address (11010<sub>two</sub>)

Index	V	Tag	Data
000	Y	10 <sub>two</sub>	Memory (10000 <sub>two</sub> )
001	N		
010	Y	11 <sub>two</sub>	Memory (11010 <sub>two</sub> )
011	N		
100	N		
101	N		
110	Y	10 <sub>two</sub>	Memory (10110 <sub>two</sub> )
111	N		

d. After handling a miss of address (10000<sub>two</sub>)

Index	V	Tag	Data
000	Y	10 <sub>two</sub>	Memory (10000 <sub>two</sub> )
001	N		
010	Y	11 <sub>two</sub>	Memory (11010 <sub>two</sub> )
011	Y	00 <sub>two</sub>	Memory (00011 <sub>two</sub> )
100	N		
101	N		
110	Y	10 <sub>two</sub>	Memory (10110 <sub>two</sub> )
111	N		

e. After handling a miss of address (00011<sub>two</sub>)

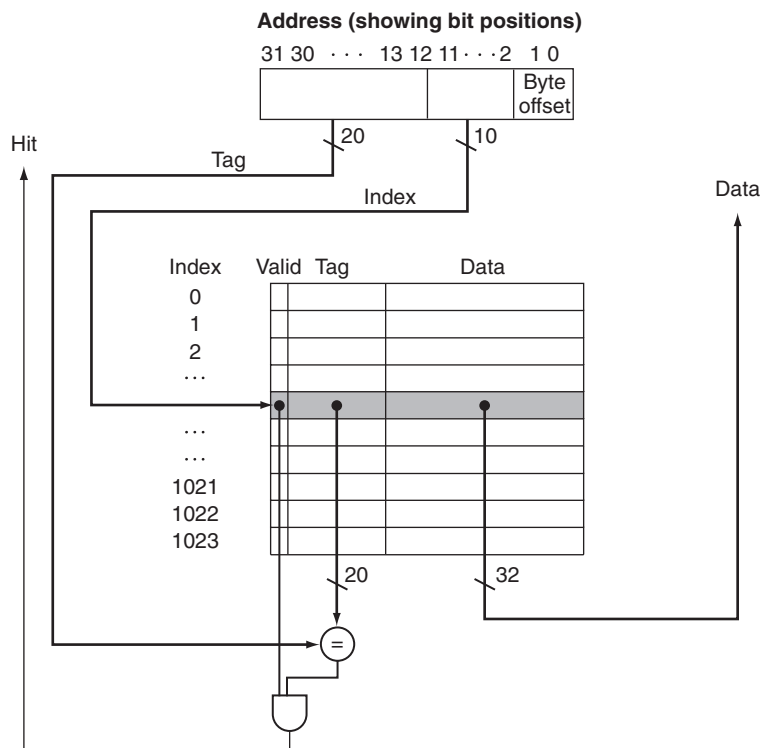
Index	V	Tag	Data
000	Y	10 <sub>two</sub>	Memory (10000 <sub>two</sub> )
001	N		
010	Y	10 <sub>two</sub>	Memory (10010 <sub>two</sub> )
011	Y	00 <sub>two</sub>	Memory (00011 <sub>two</sub> )
100	N		
101	N		
110	Y	10 <sub>two</sub>	Memory (10110 <sub>two</sub> )
111	N		

f. After handling a miss of address (10010<sub>two</sub>)

**FIGURE 7.6 The cache contents are shown after each reference request that misses, with the index and tag fields shown in binary.** The cache is initially empty, with all valid bits (V entry in cache) turned off (N). The processor requests the following addresses: 10110<sub>two</sub> (miss), 11010<sub>two</sub> (miss), 10110<sub>two</sub> (hit), 11010<sub>two</sub> (hit), 10000<sub>two</sub> (miss), 00011<sub>two</sub> (miss), 10000<sub>two</sub> (hit), and 10010<sub>two</sub> (miss). The figures show the cache contents after each miss in the sequence has been handled. When address 10010<sub>two</sub> (18) is referenced, the entry for address 11010<sub>two</sub> (26) must be replaced, and a reference to 11010<sub>two</sub> 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  $j \times 8 + i$ , or equivalently the concatenation of the tag field  $j$  and the index  $i$ . For example, in cache f above, index 010 has tag 10 and corresponds to address 10010.

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 7.7 shows how a referenced address is divided into

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



**FIGURE 7.7** For this cache, the lower portion of the address is used to select a cache entry consisting of a data word and a tag. 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 1 word, 10 bits are used to index the cache, leaving  $32 - 10 - 2 = 20$  bits to be compared against the tag. If the tag and upper 20 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 access 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. In the MIPS architecture, since words are aligned to multiples of 4 bytes, the least significant 2 bits of every address specify a byte within a word and hence are ignored when selecting the word in the block.

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, but normally it is several. Assuming the 32-bit byte address, a direct-mapped cache of size  $2^n$  blocks with  $2^m$ -word ( $2^{m+2}$ -byte) blocks will require a tag field whose size is  $32 - (n + m + 2)$  bits because  $n$  bits are used for the index,  $m$  bits are used for the word within the block, and 2 bits are used for the byte part of the address. 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 the address size is 32 bits, the number of bits in such a cache is  $2^n \times (2m \times 32 + (32 - n - m - 2) + 1) = 2^n \times (2m \times 32 + 31 - n - m)$ . However, the naming convention is to exclude the size of the tag and valid field and to count only the size of the data.

### Bits in a Cache

How many total bits are required for a direct-mapped cache with 16 KB of data and 4-word blocks, assuming a 32-bit address?

#### EXAMPLE

We know that 16 KB is 4K words, which is  $2^{12}$  words, and, with a block size of 4 words ( $2^2$ ),  $2^{10}$  blocks. Each block has  $4 \times 32$  or 128 bits of data plus a tag, which is  $32 - 10 - 2 - 2$  bits, plus a valid bit. Thus, the total cache size is

#### ANSWER

$$2^{10} \times (128 + (32 - 10 - 2 - 2) + 1) = 2^{10} \times 147 = 147 \text{ Kbits}$$

or 18.4 KB for a 16 KB cache. For this cache, the total number of bits in the cache is about 1.15 times as many as needed just for the storage of the data.

**EXAMPLE****ANSWER****Mapping an Address to a Multiword Cache Block**

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

We saw the formula on page 474. The block is given by

$$(\text{Block address}) \bmod (\text{Number of cache blocks})$$

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

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

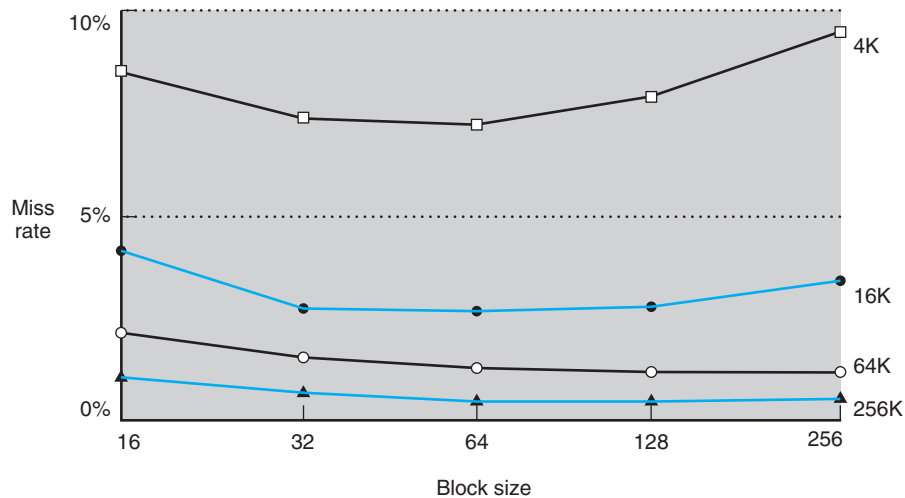
Thus, with 16 bytes per block, byte address 1200 is block address

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

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

Larger blocks exploit spatial locality to lower miss rates. As Figure 7.8 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 in the miss rate become smaller.

A more serious problem associated with just increasing the block size is that the cost of a miss increases. The miss penalty is determined by the time required to



**FIGURE 7.8 Miss rate versus block size.** Note that 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, SPEC2000 traces would take too long if block size were included, so this data is based on SPEC92.

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 increase as the block size grows. 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 large blocks, 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:** The major disadvantage of increasing the block size is that the cache miss penalty increases. Although it is hard to do anything about the latency component of the miss penalty, we may be able to hide some of the transfer time so that the miss penalty is effectively smaller. The simplest 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 limit early restart.

### Handling Cache Misses

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

Before we look at the cache of a real system, let's see how the control unit deals with **cache misses**. 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. Consequently, we can use the same basic control that we developed in Chapter 5 and enhanced to accommodate pipelining in Chapter 6. The memories in the datapath in Chapters 5 and 6 are simply replaced by caches.

Modifying the control of a processor to handle a hit is trivial; misses, however, require some extra work. The cache miss handling is done 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 stall, similar to the pipeline stalls discussed in Chapter 6, as opposed to an 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. In contrast, pipeline stalls, discussed in Chapter 6, are more complex because we must continue executing some instructions while we stall others.

Let's look a little more closely at how instruction misses are handled for either the multicycle or pipelined datapath; 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 instruct the lower level in the memory hierarchy to perform a read. Since the program counter is incremented in the first clock cycle of execution in both the pipelined and multicycle processors, 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 cycles), and then write the words into the cache.

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

1. Send the original PC value (current PC – 4) 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 to always write the data into both the memory and the cache. This scheme is called **write-through**.

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 very 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 the SPEC2000 integer benchmarks, for example, 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  $1.0 + 100 \times 10\% = 11$ , 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 it is 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 gener-

**write-through** A scheme in which writes always update both the cache and the memory, ensuring that data is always consistent between the two.

**write buffer** A queue that holds data while the data is waiting to be written to memory.



**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.

ating writes, no amount of buffering can help because writes are being generated faster than the memory system can accept them.

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.

In the rest of this section, we describe caches from real processors, and we examine how they handle both reads and writes. In Section 7.5, 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 strategy followed in most write-through cache designs, called *fetch-on-miss*, *fetch-on-write*, or sometimes *allocate-on-miss*, allocates a cache block to the address that missed and fetches the rest of the block into the cache before writing the data and continuing execution. Alternatively, we could either allocate the block in the cache but not fetch the data (called *no-fetch-on-write*), or even not allocate the block (called *no-allocate-on-write*). Another name for these strategies that do not place the written data into the cache is *write-around*, since the data is written around the cache to get to memory. The motivation for these schemes is the observation that sometimes programs write entire blocks of data before reading them. In such cases, the fetch associated with the initial write miss may be unnecessary. There are a number of subtle issues involved in implementing these schemes in multiword blocks, including complicating the handling of write hits by requiring mechanisms similar to those used for write-back caches.

Actually implementing stores efficiently in a cache that uses a write-back strategy is more complex than in a write-through cache. In a write-back cache, we must write the block back to memory if the data in the cache is dirty 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 memory. 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, 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 an extra buffer, called a *store 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 is 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. There are some extra complications with multiword blocks, however, since we cannot simply overwrite the tag when we write the data. Instead, 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. Because it is always safe to overwrite the data, write hits still take one cycle.

Many write-back caches also include write buffers that are used to reduce the miss penalty when a miss replaces a dirty block. In such a case, the dirty 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.

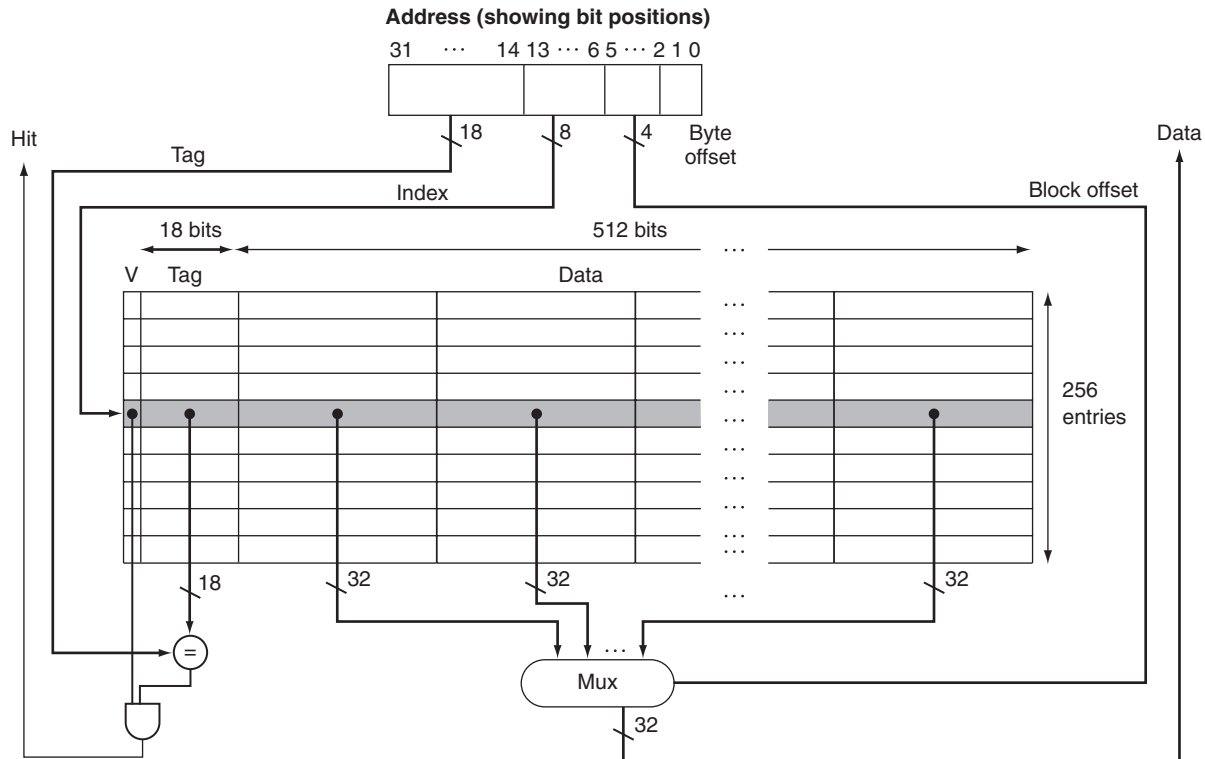
### An Example Cache: The Intrinsity FastMATH Processor

The Intrinsity FastMATH is a fast 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 design of the Intel Pentium P4, but we start with this simple, yet real, example for pedagogical reasons. Figure 7.9 shows the organization of the Intrinsity FastMATH data cache.

This processor has a 12-stage pipeline, similar to that discussed in Chapter 6. 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 KB, or 4K words, with 16-word blocks.

Read requests for the cache are straightforward. Because there are separate data and instruction caches, separate control signals will be needed 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



**FIGURE 7.9 The 16 KB caches in the Intrinsity FastMATH each contain 256 blocks with 16 words per block.** 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.

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.

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

**FIGURE 7.10 Approximate instruction and data miss rates for the Intrinsity FastMATH processor for SPEC2000 benchmarks.** The combined miss rate is the effective miss rate seen for the combination of the 16 KB instruction cache and 16 KB data cache. It is obtained by weighting the instruction and data individual miss rates by the frequency of instruction and data references.

What cache miss rates are attained with a cache structure like that used by the Intrinsity FastMATH? Figure 7.10 shows the miss rates for the instruction and data caches for the SPEC2000 integer benchmarks. 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.

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, many processors use a split instruction and data cache to increase cache *bandwidth*.

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 total of the two caches:

- Total cache size: 32 KB
- 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 is another reminder that we cannot use miss rate as the sole measure of cache performance, as Section 7.3 shows.

## Designing the Memory System to Support Caches

Cache misses are satisfied from main memory, which is constructed from DRAMs. In Section 7.1, we saw that DRAMs are designed with the primary emphasis on density rather than access time. Although it is difficult to reduce the latency to fetch the first word from memory, we can reduce the miss penalty if we increase the bandwidth from the memory to the cache. This reduction allows

**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.

larger block sizes to be used while still maintaining a low miss penalty, similar to that for a smaller block.

The processor is typically connected to memory over a bus. The clock rate of the bus is usually much slower than the processor, by as much as a factor of 10. The speed of this bus affects the miss penalty.

To understand the impact of different organizations of memory, let's define a set of hypothetical memory access times. Assume

- 1 memory bus clock cycle to send the address
- 15 memory bus clock cycles for each DRAM access initiated
- 1 memory bus clock cycle to send a word of data

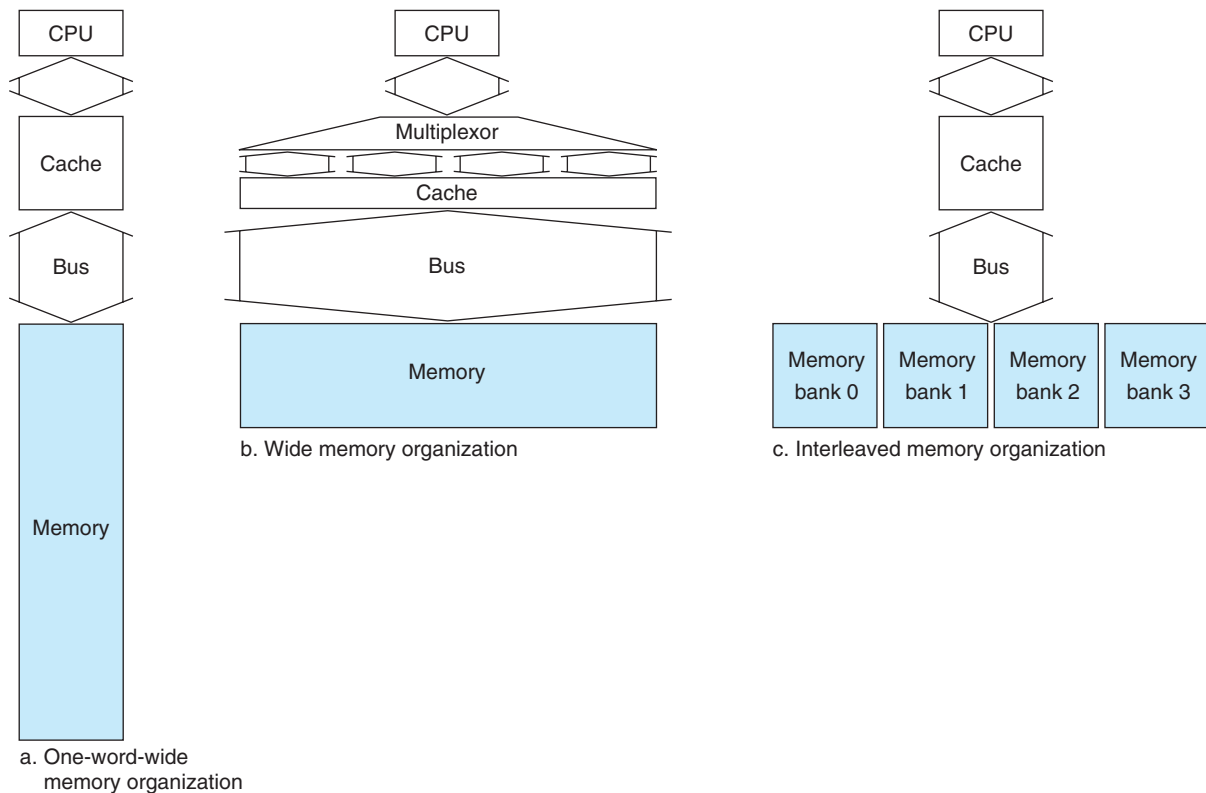
If we have a cache block of four words and a one-word-wide bank of DRAMs, the miss penalty would be  $1 + 4 \times 15 + 4 \times 1 = 65$  memory bus clock cycles. Thus, the number of bytes transferred per bus clock cycle for a single miss would be

$$\frac{4 \times 4}{65} = 0.25$$

Figure 7.11 shows three options for designing the memory system. The first option follows what we have been assuming: memory is one word wide, and all accesses are made sequentially. The second option increases the bandwidth to memory by widening the memory and the buses between the processor and memory; this allows parallel access to all the words of the block. The third option increases the bandwidth by widening the memory but not the interconnection bus. Thus, we still pay a cost to transmit each word, but we can avoid paying the cost of the access latency more than once. Let's look at how much these other two options improve the 65-cycle miss penalty that we would see for the first option [Figure 7.11(a)].

Increasing the width of the memory and the bus will increase the memory bandwidth proportionally, decreasing both the access time and transfer time portions of the miss penalty. With a main memory width of two words, the miss penalty drops from 65 memory bus clock cycles to  $1 + (2 \times 15) + 2 \times 1 = 33$  memory bus clock cycles. With a four-word-wide memory, the miss penalty is just 17 memory bus clock cycles. The bandwidth for a single miss is then 0.48 (almost twice as high) bytes per bus clock cycle for a memory that is two words wide, and 0.94 bytes per bus clock cycle when the memory is four words wide (almost four times higher). The major costs of this enhancement are the wider bus and the potential increase in cache access time due to the multiplexor and control logic between the processor and cache.

Instead of making the entire path between the memory and cache wider, the memory chips can be organized in banks to read or write multiple words in one



**FIGURE 7.11 The primary method of achieving higher memory bandwidth is to increase the physical or logical width of the memory system.** In this figure, memory bandwidth is improved two ways. The simplest design, (a), uses a memory where all components are one word wide; (b) shows a wider memory, bus, and cache; while (c) shows a narrow bus and cache with an interleaved memory. In (b), the logic between the cache and processor consists of a multiplexor used on reads and control logic to update the appropriate words of the cache on writes.

access time rather than reading or writing a single word each time. Each bank could be one word wide so that the width of the bus and the cache need not change, but sending an address to several banks permits them all to read simultaneously. This scheme, which is called *interleaving*, retains the advantage of incurring the full memory latency only once. For example, with four banks, the time to get a four-word block would consist of 1 cycle to transmit the address and read request to the banks, 15 cycles for all four banks to access memory, and 4 cycles to send the four words back to the cache. This yields a miss penalty of  $1 + (1 \times 15) + 4 \times 1 = 20$  memory bus clock cycles. This is an effective bandwidth per miss of 0.80 bytes per clock, or about three times the bandwidth for the one-word-wide

memory and bus. Banks are also valuable on writes. Each bank can write independently, quadrupling the write bandwidth and leading to fewer stalls in a write-through cache. As we will see, an alternative strategy for writes makes interleaving even more attractive.


**Elaboration:** Memory chips are organized to produce a number of output bits, usually 4 to 32, with 16 being the most popular in 2006. We describe the organization of a RAM as  $d \times w$ , where  $d$  is the number of addressable locations (the depth) and  $w$  is the output (or width of each location). One path to improving the rate at which we transfer data from the memory to the caches is to take advantage of the structure of DRAMs. DRAMs are logically organized as rectangular arrays, and access time is divided into row access and column access. DRAMs buffer a row of bits inside the DRAM for column access. They also come with optional timing signals that allow repeated accesses to the buffer without a row access time. This capability, originally called *page mode*, has gone through a series of enhancements. In page mode, the buffer acts like an SRAM; by changing column address, random bits can be accessed in the buffer until the next row access. This capability changes the access time significantly, since the access time to bits in the row is much lower. Figure 7.12 shows how the density, cost, and access time of DRAMs have changed over the years.

The newest development is DDR SDRAMs (double data rate synchronous DRAMs). SDRAMs provide for a burst access to data from a series of sequential locations in the DRAM. An SDRAM is supplied with a starting address and a burst length. The data in the burst is transferred under control of a clock signal, which in 2006 can run at up to

Year introduced	Chip size	\$ per MB	Total access time to a new row/column	Column access time to existing row
1980	64 Kbit	\$1500	250 ns	150 ns
1983	256 Kbit	\$500	185 ns	100 ns
1985	1 Mbit	\$200	135 ns	40 ns
1989	4 Mbit	\$50	110 ns	40 ns
1992	16 Mbit	\$15	90 ns	30 ns
1996	64 Mbit	\$10	60 ns	12 ns
1998	128 Mbit	\$4	60 ns	10 ns
2000	256 Mbit	\$1	55 ns	7 ns
2004	512 Mbit	\$0.25	50 ns	5 ns
2006	512 Mbit	\$0.20	40 ns	2.5 ns

**FIGURE 7.12 DRAM size increased by multiples of four approximately once every three years until 1996, and thereafter considerably slower.** The improvements in access time have been slower but continuous, and cost almost tracks density improvements, although cost is often affected by other issues, such as availability and demand. The cost per megabyte is not adjusted for inflation.

400 MHz. The two key advantages of SDRAMs are the use of a clock that eliminates the need to synchronize and the elimination of the need to supply successive addresses in the burst. The DDR part of the name means data transfers on both the leading 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. To deliver such high bandwidth, the internal DRAM is organized as interleaved memory banks.

The advantage of these optimizations is that they use the circuitry already largely on the DRAMs, adding little cost to the system while achieving a significant improvement in bandwidth. The internal architecture of DRAMs and how these optimizations are implemented are described in Section B.8 of  [Appendix B](#).

## 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 larger 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 this, the bandwidth of main memory is increased to transfer cache blocks more efficiently. The two common methods for doing this are making the memory wider and interleaving. In both cases, we reduce the time to fetch the block by minimizing the number of times we must start a new memory access to fetch a block, and, with a wider bus, we can also decrease the time needed to send the block from the memory to the cache.

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 are 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.

**Check  
Yourself**



## 7.3

**Measuring and Improving Cache Performance**

In this section, we begin by looking at how 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 different 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 over \$100,000 in 1990; since then it has become common on desktop computers selling for less than \$1000!

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 484 for more details on dealing with writes), and write buffer stalls, which occur when the write buffer is full when a write occurs. Thus, the cycles stalled for writes equals 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 timing of writes, and not just the frequency, it is not possible 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 7.5.

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

Assume an instruction cache miss rate for a program is 2% and a data cache miss rate 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. Use the instruction frequencies for SPECint2000 from Chapter 3, Figure 3.26, on page 228.

### EXAMPLE

**ANSWER**

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

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

Assume the frequency of all loads and stores in SPECint2000 is 36%. Therefore, we can find the number of memory miss cycles for data references:

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

The total number of memory-stall cycles is  $2.00 I + 1.44 I = 3.44 I$ . This is more than 3 cycles of memory stall per instruction. Accordingly, the CPI with 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

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

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 4, 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

$$\frac{4.44}{1} = 4.44 \text{ times faster}$$

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 clock rate without changing the memory system also increases the performance lost due to cache misses, as the next example shows.

#### Cache Performance with Increased Clock Rate

Suppose we increase the performance of the computer in the previous example by doubling its clock rate. Since the main memory speed is unlikely to change, assume that the absolute time to handle a cache miss does not change. How much faster will the computer be with the faster clock, assuming the same miss rate as the previous example?

#### EXAMPLE

Measured in the faster clock cycles, the new miss penalty will be twice as many clock cycles, or 200 clock cycles. Hence:

#### ANSWER

$$\text{Total miss cycles per instruction} = (2\% \times 200) + 36\% \times (4\% \times 200) = 6.88$$

Thus, the faster computer with cache misses will have a CPI of  $2 + 6.88 = 8.88$ , compared to a CPI with cache misses of 5.44 for the slower computer.

Using the formula for CPU time from the previous example, we can compute the relative performance as

$$\begin{aligned} \frac{\text{Performance with fast clock}}{\text{Performance with slow clock}} &= \frac{\text{Execution time with slow clock}}{\text{Execution time with fast clock}} \\ &= \frac{\text{IC} \times \text{CPI}_{\text{slow clock}} \times \text{Clock cycle}}{\text{IC} \times \text{CPI}_{\text{fast clock}} \times \frac{\text{Clock cycle}}{2}} \\ &= \frac{5.44}{8.88 \times \frac{1}{2}} = 1.23 \end{aligned}$$

Thus, the computer with the faster clock is about 1.2 times faster rather than 2 times faster, which it would have been if we ignored cache misses.

As these examples illustrate, relative cache penalties increase as a processor becomes faster. Furthermore, if a processor improves both clock rate and CPI, it suffers a double hit:

1. The lower the CPI, the more pronounced the impact of stall cycles.
2. The main memory system is unlikely to improve as fast as processor cycle time, primarily because the performance of the underlying DRAM is not getting much faster. When calculating CPI, the cache miss penalty is measured in processor clock cycles needed for a miss. Therefore, if the main memories of two processors have the same absolute access times, a higher processor clock rate leads to a larger miss penalty.

Thus, the importance of cache performance for processors with low CPI and high clock rates is greater, and consequently the danger of neglecting cache behavior in assessing the performance of such processors is greater. As we will see in Section 7.6, the use of fast, pipelined processors in desktop PCs and workstations has led to the use of sophisticated cache systems even in computers selling for less than \$1000.

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 increase hit time shortly, one example is increasing the cache size. A larger cache could clearly have a longer 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. With pipelines deeper than five stages, 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.

The next subsection discusses alternative cache organizations that decrease miss rate but may sometimes increase hit time; additional examples appear in “Fallacies and Pitfalls” (Section 7.7).

### Reducing Cache Misses by More Flexible Placement of Blocks

So far, when we place 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. There is actually a whole range of schemes for placing blocks. At one extreme is direct mapped, where a block can be placed in exactly one location.

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.

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 (at least two) 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.

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

$$(\text{Block number}) \bmod (\text{Number of cache blocks})$$

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

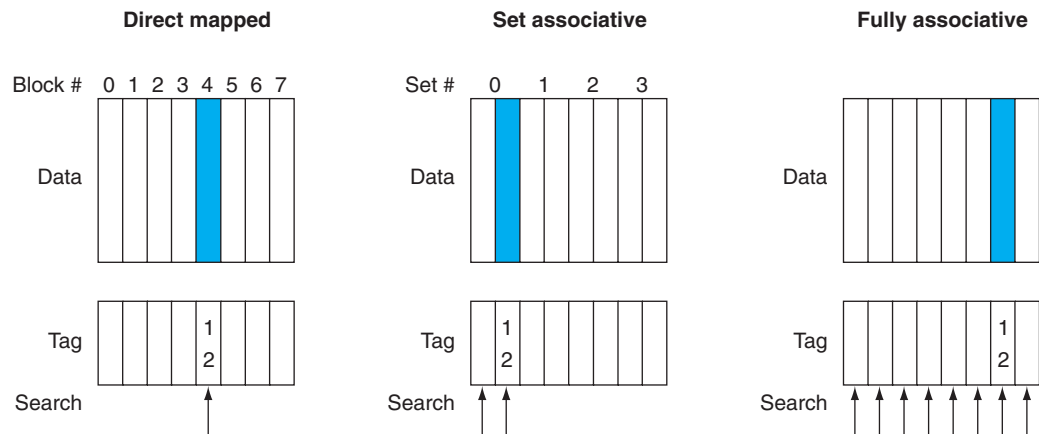
$$(\text{Block number}) \bmod (\text{Number of sets in the cache})$$

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. For example, Figure 7.13 shows where block 12 may be placed in a cache with eight blocks total, according to the block placement policy for direct-mapped, two-way set-associative, and fully associative caches.

We can think of every block placement strategy as a variation on set associativity. Figure 7.14 shows the possible associativity structures for an eight-block cache. A direct-mapped cache is simply a one-way set-associative cache: each

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

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



**FIGURE 7.13** 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  $(12 \text{ modulo } 8) = 4$ . In a two-way set-associative cache, there would be four sets, and memory block 12 must be in set  $(12 \text{ mod } 4) = 0$ ; 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.

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.

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 an increase in the hit time.

### EXAMPLE

#### Misses and Associativity in Caches

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, 8.

One-way set associative  
(direct mapped)

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

Four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

Eight-way set associative (fully associative)

Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data

**FIGURE 7.14** 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 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

ANSWER



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 associate 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 replacement rules in more detail shortly.) Using this replacement rule, the contents of the set-associative cache after each reference looks 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 same number of misses. This change in miss rate shows us 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 7.15 shows the improvement for the SPEC2000 benchmarks for a 64 KB 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 7.15** The data cache miss rates for an organization like the Intrinsic FastMATH processor for SPEC2000 benchmarks with associativity varying from one-way to eight-way. These results for 10 SPEC2000 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 7.16 shows how the address is decomposed. 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.

If the total cache size is kept the same, increasing the associativity increases 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 increases 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 entire cache without any indexing.

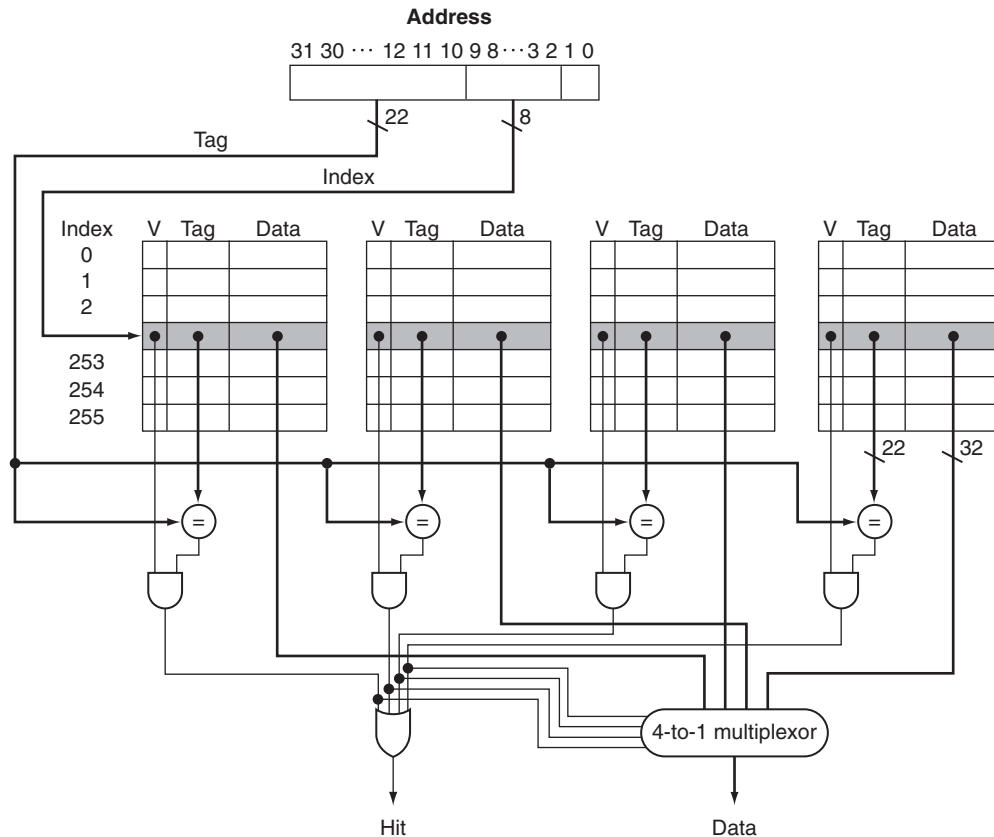
In a direct-mapped cache, such as in Figure 7.7 on page 478, only a single comparator is needed, because the entry can be in only one block, and we access the cache simply by indexing. Figure 7.17 shows that in a four-way set-associative cache, four comparators are needed, together with a 4-to-1 multiplexor 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.

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.

Tag	Index	Block offset
-----	-------	--------------

**FIGURE 7.16 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.

Copyright © 2007. Elsevier Science. All rights reserved.



**FIGURE 7.17 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.

### Size of Tags versus Set Associativity

Increasing associativity requires more comparators, and more tag bits per cache block. Assuming a cache of 4K blocks, a four-word block size, and a 32-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.

### EXAMPLE

**ANSWER**

Since there are 16 ( $= 2^4$ ) bytes per block, a 32-bit address yields  $32 - 4 = 28$  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  $\log_2(4K) = 12$ ; hence, the total number of tag bits is  $(28 - 12) \times 4K = 16 \times 4K = 64$  Kbits.

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 2K sets, and the total number of tag bits is  $(28 - 11) \times 2 \times 2K = 34 \times 2K = 68$  Kbits. For a four-way set-associative cache, the total number of sets is 1K, and the total number of tag bits is  $(28 - 10) \times 4 \times 1K = 72 \times 1K = 72$  Kbits.

For a fully associative cache, there is only one set with 4K blocks, and the tag is 28 bits, leading to a total of  $28 \times 4K \times 1 = 112K$  tag bits.

**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. 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 7.5, we will see an alternative scheme for replacement.

**Reducing the Miss Penalty Using Multilevel Caches**

All modern computers make use of caches. In most cases, these caches are implemented on the same die as the microprocessor that forms the processor. To further close the gap between the fast clock rates of modern processors and the relatively long time required to access DRAMs, many microprocessors support an additional level of caching. This second-level cache, which can be on the same chip or off-chip in a separate set of SRAMs, is accessed whenever a miss occurs in

**least recently used (LRU)** A replacement scheme in which the block replaced is the one that has been unused for the longest time.

the primary cache. If the second-level cache contains the desired data, the miss penalty for the first-level cache will be 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 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

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 5 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%?

#### EXAMPLE

The miss penalty to main memory is

$$\frac{100 \text{ ns}}{0.2 \frac{\text{ns}}{\text{clock cycle}}} = 500 \text{ clock cycles}$$

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,

$$\text{Total CPI} = 1.0 + \text{Memory-stall cycles per instruction} = 1.0 + 2\% \times 500 = 11.0$$

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

#### ANSWER

$$\frac{5 \text{ ns}}{0.2 \frac{\text{ns}}{\text{clock cycle}}} = 25 \text{ clock cycles}$$

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:

$$\begin{aligned} \text{Total CPI} &= 1 + \text{Primary stalls per instruction} \\ &\quad + \text{Secondary stalls per instruction} \\ &= 1 + 2\% \times 25 + 0.5\% \times 500 = 1 + 0.5 + 2.5 = 4.0 \end{aligned}$$

Thus, the processor with the secondary cache is faster by

$$\frac{11.0}{4.0} = 2.8$$

Alternatively, we could have computed the stall cycles by summing the stall cycles of those references that hit in the secondary cache  $((2\% - 0.5\%) \times 25 = 0.4)$  and 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  $(0.5\% \times (25 + 500) = 2.6)$ . The sum,  $1.0 + 0.4 + 2.6$ , is again 4.0.

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, while allowing the secondary cache to focus on miss rate to reduce the penalty of long memory access times.

The interaction of the two caches permits such a focus. The miss penalty of the primary cache is significantly reduced by the presence of the secondary cache, allowing the primary to be smaller and have a higher miss rate. For the secondary cache, access time becomes less important with the presence of the primary cache, since the access time of the secondary cache affects the miss penalty of the primary cache, rather than directly affecting the primary cache hit time or the processor cycle time.

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 often uses a smaller block size, to go with the smaller cache size and reduced miss penalty. In comparison, the secondary cache will often be 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 often will use a larger block size than appropriate with a single-level cache.

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

In Chapter 2, we saw that Quicksort had an algorithmic advantage over Bubble Sort that could not be overcome by language or compiler optimization. Figure 7.18(a) shows instructions executed by item searched for Radix Sort versus Quicksort. Indeed, for large arrays, Radix Sort has an algorithmic advantage over Quicksort in terms of number of operations. Figure 7.18(b) shows time per key instead of instructions executed. We see that the lines start on the same trajectory as Figure 7.18(a), but then the Radix Sort line diverges as the data to sort increases. What is going on? Figure 7.18(c) answers by looking at the cache misses per item sorted: Quicksort consistently has many fewer misses per item to be sorted.

Alas, standard algorithmic analysis ignores the impact of the memory hierarchy. As faster clock rates and Moore's law allow architects to squeeze all of the performance out of a stream of instructions, using the memory hierarchy well is critical 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.

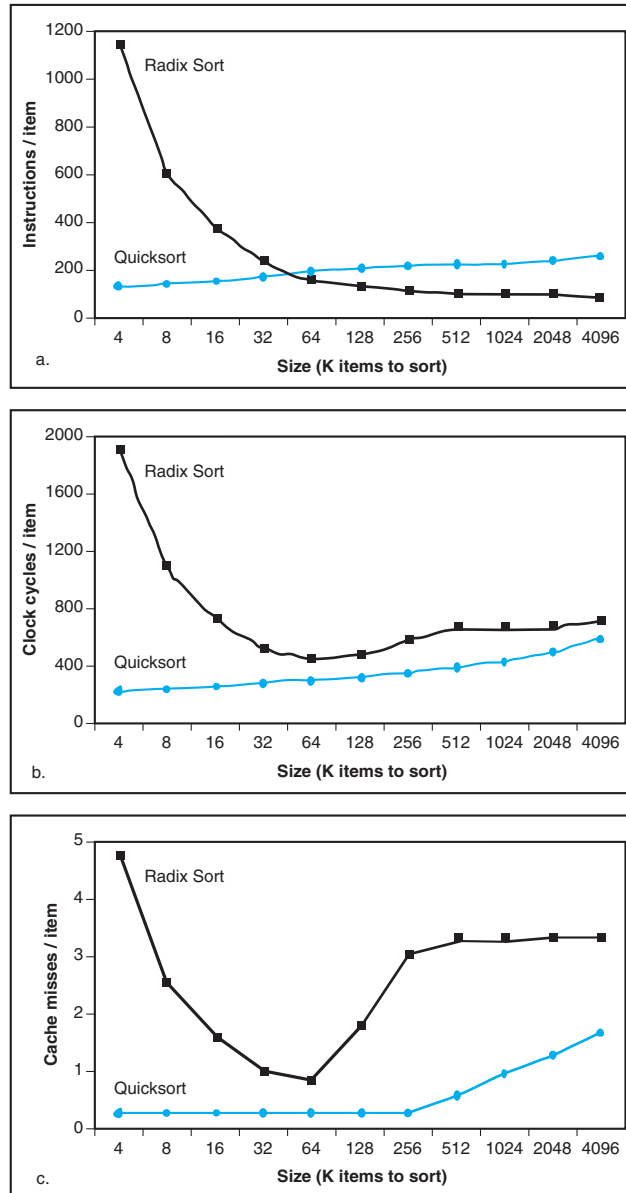
## Understanding Program Performance

**Elaboration:** Multilevel caches create several complications. First, there are now several different types of misses and corresponding miss rates. In the example on page 498, 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. 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 page 498, 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.





**FIGURE 7.18 Comparing Quicksort and Radix Sort by (a) instructions executed per item sorted, (b) time per item sorted, and (c) cache misses per item sorted.** This data is from a paper by LaMarca and Ladner [1996]. Although the numbers would change for newer computers, the idea still holds. 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 7.7). The basic idea of cache optimizations is to use all the data in a block repeatedly before it is replaced on a miss.

Additional complications arise because the caches may have different block sizes to match the larger or smaller total size. Likewise, the associativity of the cache may change. On-chip caches are often built with associativity of four or higher, while off-chip caches rarely have associativity of greater than two. On-chip L1 caches tend to have lower associativity than on-chip L2 caches since fast hit time is more important for L1 caches. These changes in block size and associativity introduce complications in the modeling of the caches, which typically means that all levels need to be simulated together to understand the behavior.

**Elaboration:** With out-of-order processors, 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:

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

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 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.

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.

**Check Yourself**

## Summary

In this section, we focused on three topics: cache performance, using associativity to reduce miss rates, and the use of multilevel cache hierarchies to reduce miss penalties.

Since the total number of cycles spent on a program is the sum of the processor cycles and the memory-stall cycles, the memory system can have a significant effect on program execution time. In fact, as processors get faster (by lowering CPI or by increasing the clock rate or both), the relative effect of the memory-stall cycles increases, making good memory systems critical to achieving high performance. The number of memory-stall cycles depends on both the miss rate and the miss penalty. The challenge, as we will see in Section 7.5, 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. This search is usually implemented by having a comparator per cache block and searching the tags in parallel. The cost of the comparators makes 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.

Finally, 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 trade-offs between the size of the secondary cache and its access time depend on a number of aspects of the implementation.

## 7.4 Virtual Memory

In the previous section, 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, usually 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 multiple programs, and to remove the programming burdens of a small, limited amount of main memory. Four decades after its invention, it's the former reason that reigns today.

Consider a collection of programs running at once on a computer. The total memory required by all the programs may be much larger than the amount of main memory available on the computer, but only a fraction of this memory is actively being used at any point in time. Main memory need contain only the active portions of the many programs, 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 efficiently share the processor as well as the main memory. Of course, to allow multiple programs to share the same memory, we must be able to protect the programs from each other, ensuring that a program can only read and write the portions of main memory that have been assigned to it.

We cannot know which programs will share the memory with other programs when we compile them. In fact, the programs sharing the memory change dynamically while the programs 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 programs.

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 never 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.

*... 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

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

**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.

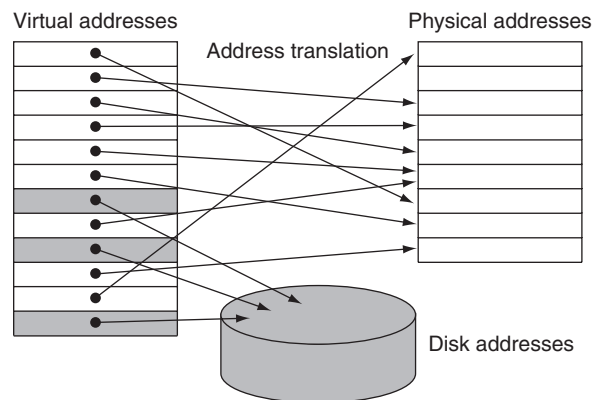
**page fault** An event that occurs when an accessed page is not present in main memory.

**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.

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 7.19 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 DRAMs



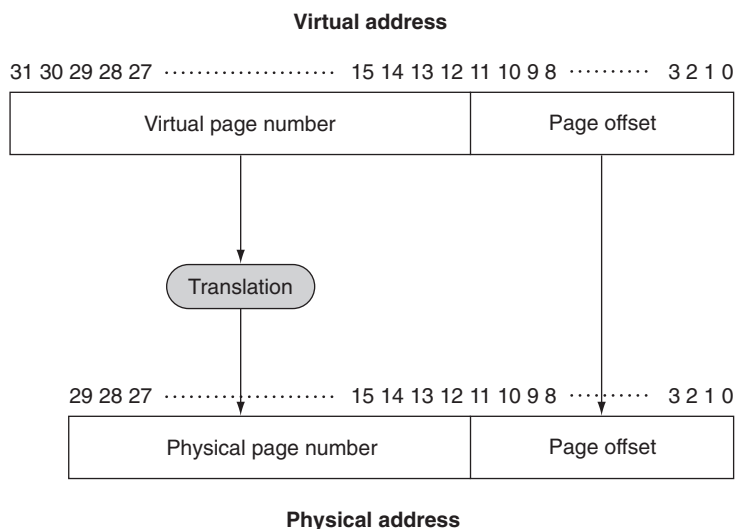
**FIGURE 7.19** In virtual memory, blocks of memory (called *pages*) are mapped from one set of addresses (called *virtual addresses*) to another set (called *physical addresses*).

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 really 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, residing instead 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.

and magnetic disks (see Chapter 1, page 23). 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.

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 need only find a sufficient number of pages in main memory. Formerly, relocation problems required special hardware and special support in the operating system; today, virtual memory also provides this function.

In virtual memory, the address is broken into a *virtual page number* and a *page offset*. Figure 7.20 shows the translation of the virtual page number to a *physical page number*. The physical page number constitutes the upper portion of the physical address, while the page offset, which is not changed, constitutes the lower



**FIGURE 7.20 Mapping from a virtual to a physical address.** The page size is  $2^{12} = 4$  KB. The number of physical pages allowed in memory is  $2^{18}$ , since the physical page number has 18 bits in it. Thus, main memory can have at most 1 GB, while the virtual address space is 4 GB.

portion. The number of bits in the page offset field determines the page size. The number of pages addressable with the virtual address need not match 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.

Many design choices in virtual memory systems are motivated by the high cost of a miss, which in virtual memory is traditionally called a *page fault*. A page fault will take millions of clock cycles to process. (The table on page 469 shows that main memory is about 100,000 times faster 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 KB to 16 KB are typical today. New desktop and server systems are being developed to support 32 KB and 64 KB pages, but new embedded systems are going in the other direction, to 1 KB 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 small 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:** 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 can benefit, but a collection of programs running at the same time can benefit from not having to be swapped to memory or by running on parallel processors. Given that Moore's law applies to DRAM, 32-bit processors are already problematic for servers and soon for desktops.

**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.

**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 register 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 logically share the address space. 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.

### 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, which 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 7.3, 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 memory; this structure is called a **page table** and resides in memory. A page table is indexed with 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 program. If we want to allow another program to use the processor, we must save this state. Later, after restoring this state, the program 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 different processes do not collide. As we will see shortly, the use of separate page tables also provides protection of one process from another.

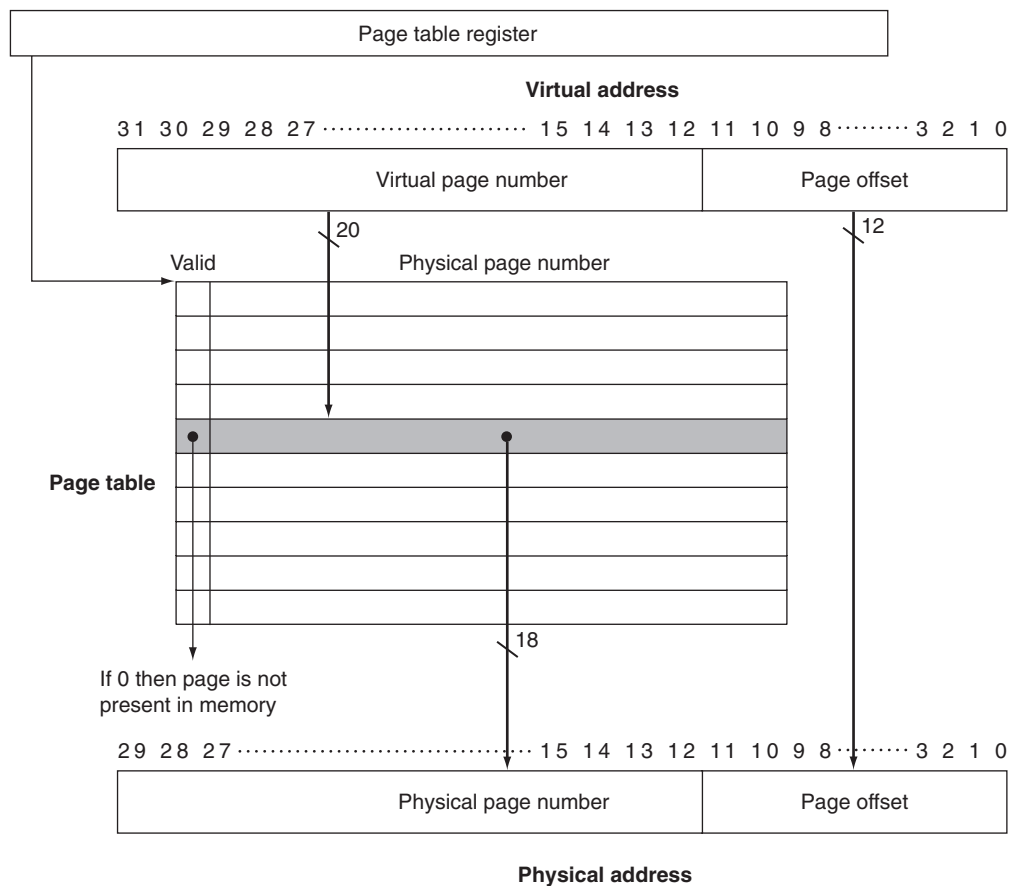
Figure 7.21 uses the page table register, the virtual address, and the indicated page table to show how the hardware can form a physical address. A valid bit is used in each page table entry, just as we did in a cache. If the bit is off, the page is not present in main memory and a page fault occurs. If the bit is on, the page is in memory and the entry contains the physical page number.

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 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 discuss later in this section. Once the operating system gets control, it must find the page in the next level of the hierarchy (usually magnetic disk) and decide where to place the requested page in main memory.

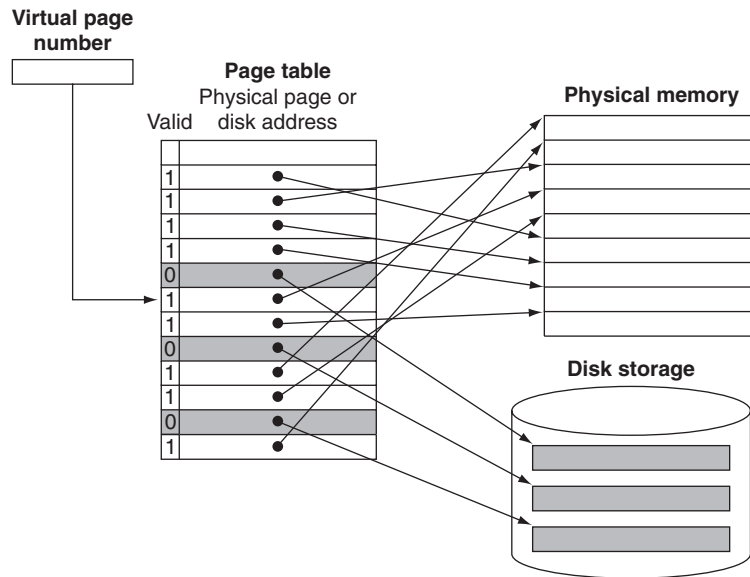
The virtual address alone does not immediately tell us where the page is on disk. Returning to our library analogy, we cannot find the location of a library book on the shelves just by knowing its title. Instead, we go to the catalog and look up the book, obtaining an address for the location on the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



**FIGURE 7.21 The page table is indexed with the virtual page number to obtain the corresponding portion of the physical address.** The starting address of the page table is given by the page table pointer. In this figure, the page size is  $2^{12}$  bytes, or 4 KB. The virtual address space is  $2^{32}$  bytes, or 4 GB, and the physical address space is  $2^{30}$  bytes, which allows main memory of up to 1 GB. The number of entries in the page table is  $2^{20}$ , or 1 million entries. 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 19 bits wide, it would typically be rounded up to 32 bits for ease of indexing. The extra bits would be used to store additional information that needs to be kept on a per-page basis, such as protection.

Because we do not know ahead of time when a page in memory will be chosen to be replaced, the operating system usually creates the space on disk for all the pages of a process when it creates the process. This disk 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 7.22

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



**FIGURE 7.22 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 identical in size.

shows the organization when a single table holds either the physical page number or the disk address.

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 in the near future. Using the past to predict the future, operating systems follow the least recently used (LRU) replacement scheme, which we mentioned in Section 7.3. The operating system searches for the least recently used page, making the assumption 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 on the disk. 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.

For example, suppose the page references (in order) were 10, 12, 9, 7, 11, 10, and then we referenced page 8, which was not present in memory. The LRU page is 12; in LRU replacement, we would replace page 12 in main memory with page 8. If the next reference also generated a page fault, we would replace page 9, since it would then be the LRU among the pages present in memory.

Implementing a completely accurate LRU scheme is too expensive, since it requires updating a data structure on *every* memory reference. Instead, 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, some computers provide a **use bit** or **reference 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.

## Hardware/ Software Interface

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

**Elaboration:** With a 32-bit virtual address, 4 KB pages, and 4 bytes per page table entry, we can compute the total page table size:

$$\text{Number of page table entries} = \frac{2^{32}}{2^{12}} = 2^{20}$$

$$\text{Size of page table} = 2^{20} \text{ page table entries} \times 2^2 \frac{\text{bytes}}{\text{page table entry}} = 4 \text{ MB}$$

That is, we would need to use 4 MB of memory for each program in execution at any time. On a computer with tens to hundreds of active programs and a fixed-size page table, most or all of the memory would be tied up in page tables!

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 only 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 segment is specified by the high-order address bit, 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. This type of segmentation is used by many architectures, including MIPS. Unlike the type of segmentation discussed in the second elaboration on page 514, 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 data structure 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. Multiple levels of page tables can also be used to reduce the total amount of page table storage. The first level maps large fixed-size blocks of virtual address space, perhaps 64 to 256 pages in total. These large blocks are sometimes called segments, and this first-level mapping table is sometimes called a segment table, though the segments are invisible to the user. Each entry in the segment table indicates whether any pages in that segment are allocated and, if so, points to a page table for that segment. Address translation happens by first looking in the segment table, using the highest-order bits of the address. If the segment address is valid, the next set of high-order bits is used to index the page table indicated by the segment table entry. 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 two-level mapping is the more complex process for address translation.
5. 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 system in a portion of main memory that is physically addressed and is always present and thus never on disk.

### 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) 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 disk when it is replaced in the memory. This copying back to the lower level in the hierarchy is the source of the other name for this technique of handling writes, namely, *copy-back*.

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 more efficient than transferring individual 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.

## Hardware/ Software Interface

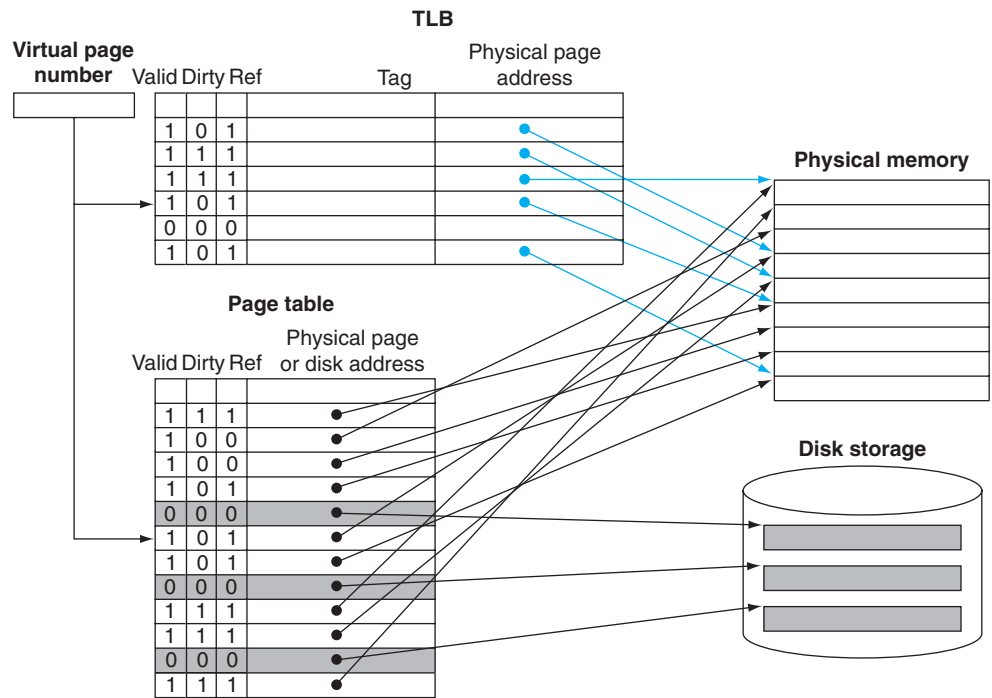
### Making Address Translation Fast: The TLB

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 in the near future because the references to the words on that page have both temporal and spatial locality.

**translation-lookaside buffer (TLB)** A cache that keeps track of recently used address mappings to avoid an access to the page table.

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.

Figure 7.23 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 will no longer access the page table on every reference, instead accessing the TLB, the TLB will need to include other bits, such as the dirty and the reference bit.



**FIGURE 7.23 The TLB acts as a cache on 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, it 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 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 7.5.

### The Intrinsicity FastMATH TLB

To see these ideas in a real processor, let's take a closer look at the TLB of the Intrinsicity FastMATH. The memory system uses 4 KB pages and a 32-bit address space; thus, the virtual page number is 20 bits long, as in the top of Figure 7.24. The physical address is the same size as the virtual address. The TLB contains 16 entries, is fully associative, and 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.

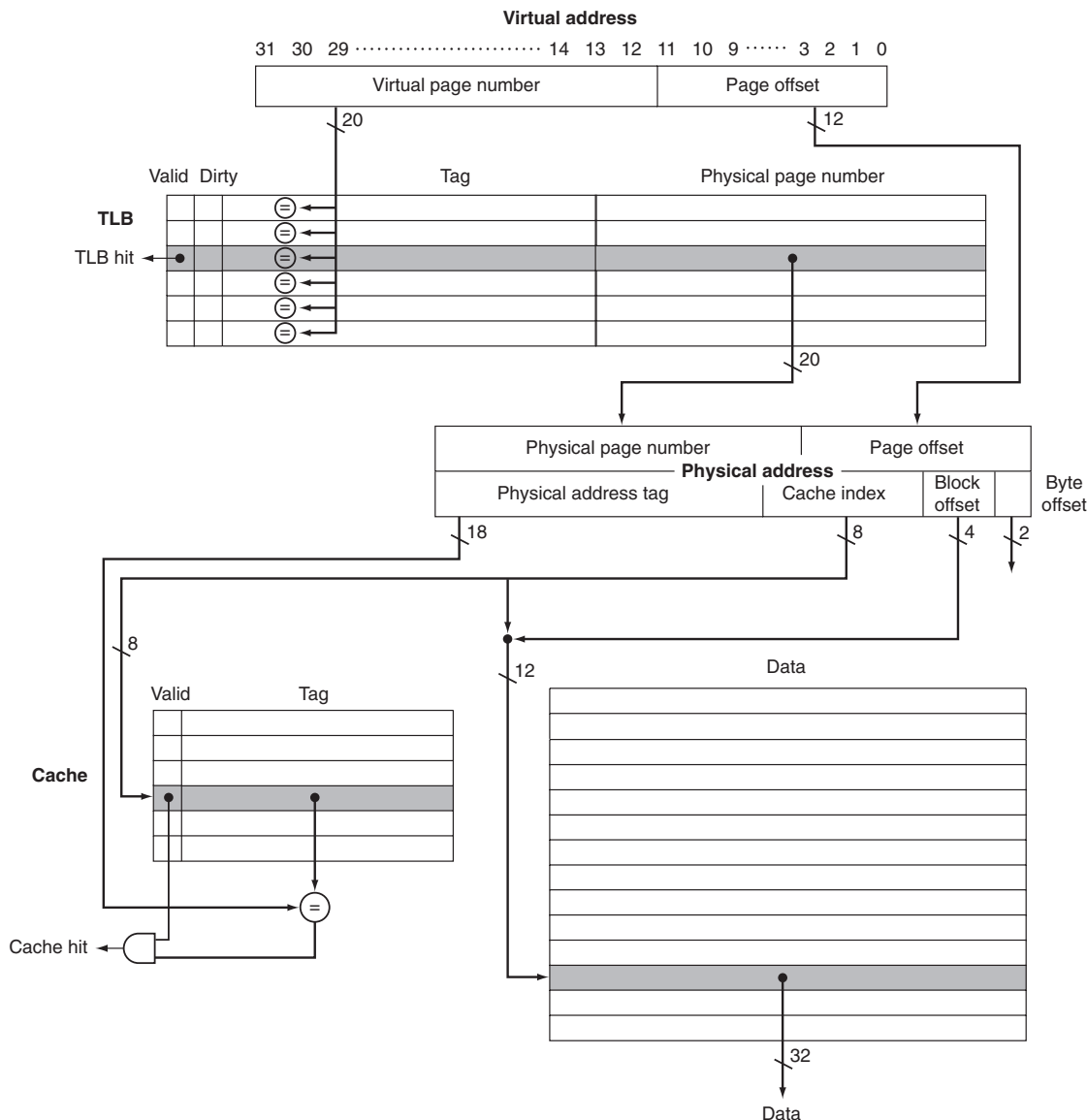
Figure 7.24 shows the TLB and one of the caches, while Figure 7.25 shows the steps in processing a read or write request. When a TLB miss occurs, the MIPS hardware saves the page number of the reference in a special register and generates an exception. The exception invokes the operating system, which handles the miss in software. To find the physical address for the missing page, the TLB miss routine 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. (We will see the MIPS TLB code on page 534.) 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; the recommended entry is chosen randomly.

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 discuss shortly.

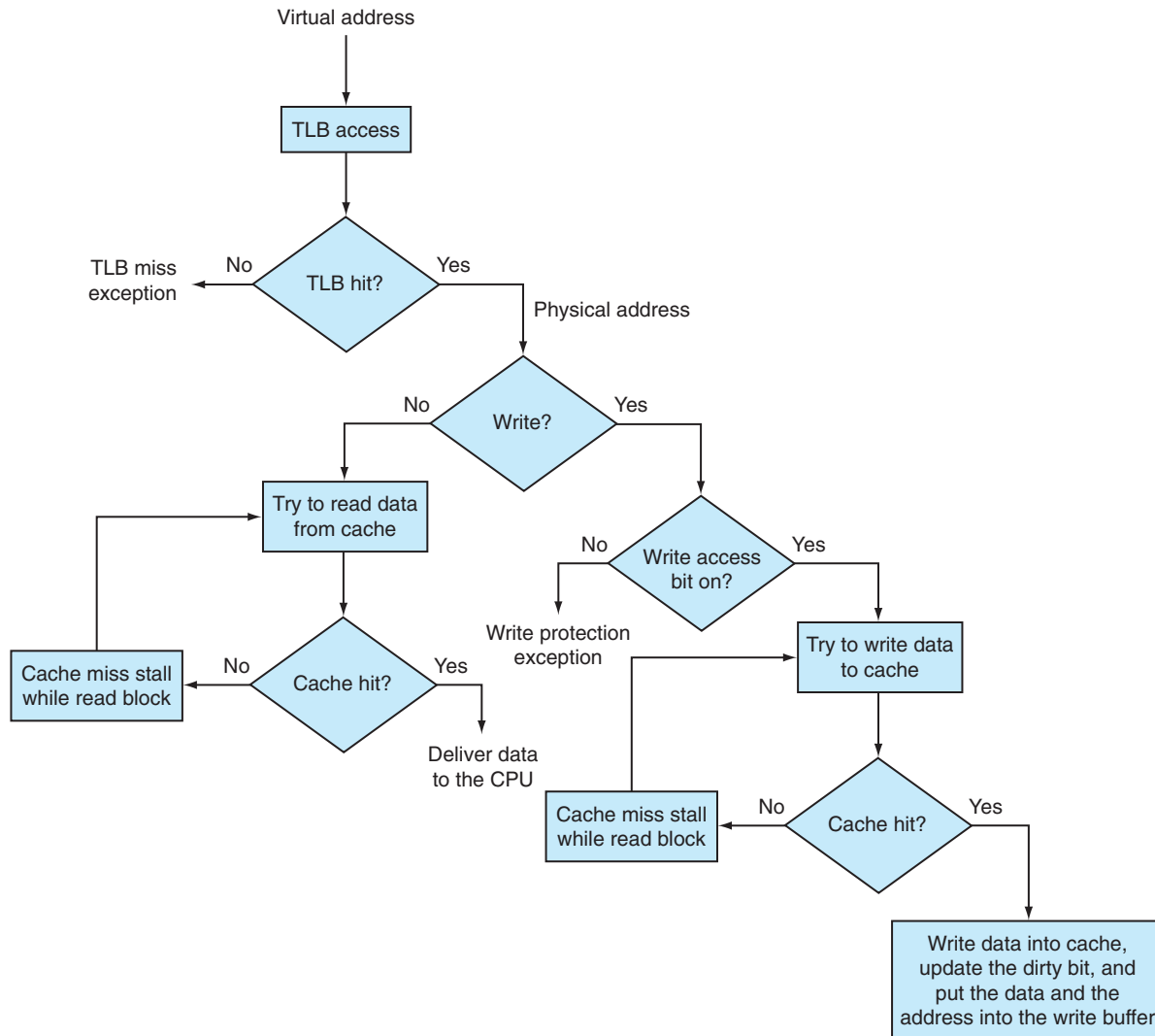
### Integrating Virtual Memory, TLBs, and Caches

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 plays an important role in maintaining this hierarchy by flushing the contents of any page from the cache, when it decides to migrate that page to disk. At the same time, the OS modifies the page tables and TLB, so that an attempt to access any data on the 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 is found, retrieved, and sent back to the processor. In the worst case, a reference can miss in all three components of



**FIGURE 7.24 The TLB and cache implement the process of going from a virtual address to a data item in the Intrinsity Fast-MATH.** This figure shows the organization of the TLB and the data cache assuming a 4 KB page size. This diagram focuses on a read; Figure 7.25 describes how to handle writes. Note that unlike Figure 7.9 on page 486, 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. 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. (The Intrinsity actually has a 16 KB page size; the elaboration on page 528 explains how it works.)



**FIGURE 7.25 Processing a read or a write-through in the Intrinsic 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 is brought from memory. If the operation is a write, a portion of the cache entry is overwritten for a hit and the data is 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.

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

In a memory hierarchy like that of Figure 7.24 that 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.

Figure 7.26 shows the possible circumstances and whether they can arise in practice or not.

### EXAMPLE

### ANSWER

**Elaboration:** Figure 7.26 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

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

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 7.26 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, virtual memory hit, cache miss) but never detected.

**aliasing** A situation in which the same object is accessed by two addresses; can occur in virtual memory when there are two virtual addresses for the same physical page.

**physically addressed cache** A cache that is addressed by a physical address.

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.

When the cache is accessed with a virtual address and pages are shared between programs (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 different 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.

Figure 7.24 assumed a 4 KB page size, but it's really 16 KB. The Intrinsity FastMATH uses such a memory system organization. The cache and TLB are still accessed in parallel, so the upper 2 bits of the cache index must be virtual. Hence, up to four cache entries could be aliased to the same physical memory address. As the L2 cache on the chip includes all entries in the L1 caches, on an L1 miss it checks the other three possible cache locations in the L2 cache for aliases. If it finds one, it flushes it from the caches to prevent aliases from occurring.

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 untranslated), 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. The L1 data cache of the Pentium 4 is an example, as would the Intrinsity L1 data cache if the page size was 4 KB. To pull off this trick, there must be careful coordination between the minimum page size, the cache size, and associativity.

**Elaboration:** The FastMATH TLB is a bit more complicated than in Figure 7.24. MIPS includes two physical page mappings per virtual page number, thereby mapping an even-odd pair of virtual page numbers into two physical page numbers. Hence, the tag is 1 bit narrower since each entry corresponds to two pages. The least significant bit of the virtual page number selects between the two physical pages. There are separate book-keeping bits for each physical page. This optimization doubles the amount of memory mapped per TLB entry. As the elaboration on page 530 explains, the tag field actually includes an 8-bit address space ID field to reduce the cost of context switches. To support the variable page sizes mentioned on page 537, there is also a 32-bit mask field that determines the dividing line between the virtual page address and the page offset.

## Implementing Protection with Virtual Memory

One of the most important functions for virtual memory 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. For example, if the program that maintains student grades were running on a computer at the same time as the programs of the students in the first programming course, we wouldn't want the errant program of a beginner to write over someone's grades. 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.

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.

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 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 (*syscall* in the MIPS 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 exception PC (EPC), and the processor is placed in supervisor mode. To return to user mode from the exception, use the *return from exception* (ERET) instruction, which resets to user mode and jumps to the address in EPC.

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.

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 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!

## Hardware/ Software Interface

**kernel mode** Also called **supervisor 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.

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. Yet, 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*.

**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.

**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. 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 different solutions, such as process identifiers, are used to ensure that a process gets its own data.

## 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, handling TLB misses and page faults is more complex. A TLB miss occurs when no entry in the TLB matches a virtual address. 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.

How do we know which of these two circumstances has occurred? When we process the TLB miss, we will look for a page table entry to bring into the TLB. If the matching page table entry has a valid bit that is turned off, then the corresponding page is not in memory and we have a page fault, rather than just a TLB miss. If the valid bit is on, we can simply retrieve the desired entry.

A TLB miss can be handled in software or hardware because it will require only a short sequence of operations to copy a valid page table entry from memory into the TLB. MIPS traditionally handles a TLB miss in software. It brings in the page table entry from memory and then reexecutes the instruction that caused the TLB miss. Upon reexecuting it will get a TLB hit. If the page table entry indicates the page is not in memory, this time it will get a page fault exception.

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. Just as in Chapters 5 and 6, the exception program counter (EPC) 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 `lw $1, 0($1)`: 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 `$1` 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.



Register	CP0 register number	Description
EPC	14	Where to restart after exception
Cause	13	Cause of exception
BadVAddr	8	Address that caused exception
Index	0	Location in TLB to be read or written
Random	1	Pseudorandom location in TLB
EntryLo	2	Physical page address and flags
EntryHi	10	Virtual page address
Context	4	Page table address and page number

**FIGURE 7.27 MIPS control registers.** These are considered to be in coprocessor 0, and hence are read using `mfc0` and written using `mtc0`.

## Hardware/ Software Interface

**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.

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 example, if another exception occurred when we were processing the first exception in the operating system, the control unit would overwrite the exception program counter, 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 exception program counter and Cause register. EPC and Cause are two of the special control registers that help with exceptions, TLB misses, and page faults; Figure 7.27 shows the rest. The operating system can then reenale 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.

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 on disk.
2. Choose a physical page to replace; if the chosen page is dirty, it must be written out to disk before we can bring a new virtual page into this physical page.
3. Start a read to bring the referenced page from disk into the chosen physical page.

Of course, this last step will take millions of processor clock cycles (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 disk 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 reexecutes 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 MIPS. 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.

For processors with much more complex instructions that may 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, some 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 MIPS 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.

Let's look in more detail at MIPS. When a TLB miss occurs, the MIPS hardware saves the page number of the reference in a special register called `BadVAddr` and generates an exception.

The exception invokes the operating system, which handles the miss in software. Control is transferred to address `8000 0000hex`, the location of the TLB miss **handler**. To find the physical address for the missing page, the TLB miss routine indexes

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

**handler** Name of a software routine invoked to "handle" an exception or interrupt.

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. To make this indexing fast, MIPS hardware places everything you need in the special Context register: the upper 12 bits have the address of the base of the page table, and the next 18 bits have the virtual address of the missing page. Each page table entry is one word, so the last 2 bits are 0. Thus, the first two instructions copy the Context register into the kernel temporary register \$k1 and then load the page table entry from that address into \$k1. Recall that \$k0 and \$k1 are reserved for the operating system to use without saving; a major reason for this convention is to make the TLB miss handler fast. Below is the MIPS code for a typical TLB miss handler:

```
TLBmiss:
    mfc0 $k1,Context    # copy address of PTE into temp $k1
    lw   $k1, 0($k1)    # put PTE into temp $k1
    mtc0 $k1,EntryLo    # put PTE into special register EntryLo
    tlbwr                    # put EntryLo into TLB entry at Random
    eret                # return from TLB miss exception
```

As shown above, MIPS has a special set of system instructions to update the TLB. The instruction `tlbwr` copies from control register `EntryLo` into the TLB entry selected by the control register `Random`. `Random` implements random replacement, so it is basically a free-running counter. A TLB miss takes about a dozen clock cycles.

Note that the TLB miss handler does not check to see if the page table entry is valid. Because the exception for TLB entry missing is much more frequent than a page fault, the operating system loads the TLB from the page table without examining the entry and restarts the instruction. If the entry is invalid, another and different exception occurs, and the operating system recognizes the page fault. This method makes the frequent case of a TLB miss fast, at a slight performance penalty for the infrequent case of a page fault.

Once the process that generated the page fault has been interrupted, it transfers control to 8000 0180<sub>hex</sub>, a different address than the TLB miss handler. This is the general address for exception; TLB miss has a special entry point to lower the penalty for a TLB miss. The operating system uses the exception Cause register to diagnose the cause of the exception. Because the exception is a page fault, the operating system knows that extensive processing will be required. Thus, unlike a TLB miss, it saves the entire state of the active process. This state includes all the general-purpose and floating-point registers, the page table address register, the EPC, and the exception Cause register. Since exception handlers do not usually use the floating-point registers, the general entry point does not save them, leaving that to the few handlers that need them.

Figure 7.28 sketches the MIPS code of an exception handler. Note that we save and restore the state in MIPS code, taking care when we enable and disable exceptions, but we invoke C code to handle the particular exception.

Save state			
Save GPR	addi	\$k1,\$sp, -XCPSIZE	# save space on stack for state
	sw	\$sp, XCT_SP(\$k1)	# save \$sp on stack
	sw	\$v0, XCT_V0(\$k1)	# save \$v0 on stack
	...		# save \$v1, \$ai, \$si, \$ti,... on stack
	sw	\$ra, XCT_RA(\$k1)	# save \$ra on stack
Save hi, lo	mfhi	\$v0	# copy Hi
	mflo	\$v1	# copy Lo
	sw	\$v0, XCT_HI(\$k1)	# save Hi value on stack
	sw	\$v1, XCT_LI(\$k1)	# save Lo value on stack
Save exception registers	mfc0	\$a0, \$cr	# copy cause register
	sw	\$a0, XCT_CR(\$k1)	# save \$cr value on stack
	...		# save \$v1,....
	mfc0	\$a3, \$sr	# copy status register
	sw	\$a3, XCT_SR(\$k1)	# save \$sr on stack
Set sp	move	\$sp, \$k1	# sp = sp - XCPSIZE
Enable nested exceptions			
	andi	\$v0, \$a3, MASK1	# \$v0 = \$sr & MASK1, enable exceptions
	mtc0	\$v0, \$sr	# \$sr = value that enables exceptions
Call C exception handler			
Set \$gp	move	\$gp, GPINIT	# set \$gp to point to heap area
Call C code	move	\$a0, \$sp	# arg1 = pointer to exception stack
	jal	xcpt_deliver	# call C code to handle exception
Restoring state			
Restore most GPR, hi, lo	move	\$at, \$sp	# temporary value of \$sp
	lw	\$ra, XCT_RA(\$at)	# restore \$ra from stack
	...		# restore \$t0,...., \$a1
	lw	\$a0, XCT_A0(\$k1)	# restore \$a0 from stack
Restore status register	lw	\$v0, XCT_SR(\$at)	# load old \$sr from stack
	li	\$v1, MASK2	# mask to disable exceptions
	and	\$v0, \$v0, \$v1	# \$v0 = \$sr & MASK2, disable exceptions
	mtc0	\$v0, \$sr	# set status register
Exception return			
Restore \$sp and rest of GPR used as temporary registers	lw	\$sp, XCT_SP(\$at)	# restore \$sp from stack
	lw	\$v0, XCT_V0(\$at)	# restore \$v0 from stack
	lw	\$v1, XCT_V1(\$at)	# restore \$v1 from stack
	lw	\$k1, XCT_EPC(\$at)	# copy old \$epc from stack
	lw	\$at, XCT_AT(\$at)	# restore \$at from stack
Restore ERC and return	mtc0	\$k1, \$epc	# restore \$epc
	eret	\$ra	# return to interrupted instruction

FIGURE 7.28 MIPS code to save and restore state on an exception.

The virtual address that caused the fault depends on whether the fault was an instruction or data fault. The address of the instruction that generated the fault is in the EPC. If it was an instruction page fault, the EPC contains the virtual address of the faulting page; otherwise, the faulting virtual address can be computed by examining the instruction (whose address is in the EPC) to find the base register and offset field.

**unmapped** A portion of the address space that cannot have page faults.

**Elaboration:** This simplified version assumes that the stack pointer (sp) is valid. To avoid the problem of a page fault during this low-level exception code, MIPS sets aside a portion of its address space that cannot have page faults, called **unmapped**. The operating system places exception entry point code and the exception stack in unmapped memory. MIPS hardware translates virtual addresses  $8000\ 0000_{\text{hex}}$  to  $\text{BFFF FFFF}_{\text{hex}}$  to physical addresses simply by ignoring the upper bits of the virtual address, thereby placing these addresses in the low part of physical memory. Thus, the operating system places exception entry points and exception stacks in unmapped memory.

**Elaboration:** The code in Figure 7.28 shows the MIPS-32 exception return sequence. MIPS-I uses `rfe` and `jr` instead of `eret`.

## Summary

Virtual memory is the name for the level of memory hierarchy that manages caching between the main memory and disk. Virtual memory allows a single program to expand its address space beyond the limits of main memory. More importantly, in recent computer systems virtual memory supports sharing of the main memory among multiple, simultaneously active processes, which together require far more total physical main memory than exists. To support sharing, virtual memory also provides mechanisms for memory protection.

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. Blocks, called 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 disk 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 unchanged pages back to disk.

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. To ensure 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 changing the page tables. Controlled sharing of pages among 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 have too much overhead and 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.

Although virtual memory was invented to enable a small memory to act as a large one, the performance difference between disk and 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 memory and disk, 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 reexamine 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 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 megabyte directly:  $64 \times 4 \text{ KB} = 0.25 \text{ MB}$ . For example, TLB misses are often a challenge for Radix Sort. To try to alleviate this problem, most computer architectures now support variable page sizes. For example, in addition to the standard 4 KB page, MIPS hardware supports 16 KB, 64 KB, 256 KB, 1 MB, 4 MB, 16 MB, 64 MB, and 256 MB 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 reexamine the algorithm and data structures to reduce the working set of pages.

## Understanding Program Performance

### Check Yourself

Match the memory hierarchy element on the left with the closest phrase on the right:

- |                |                                   |
|----------------|-----------------------------------|
| 1. L1 cache    | a. A cache for a cache            |
| 2. L2 cache    | b. A cache for disks              |
| 3. Main memory | c. A cache for a main memory      |
| 4. TLB         | d. A cache for page table entries |

## 7.5

### A Common Framework for Memory Hierarchies

By now, you've recognized that the different types of memory hierarchies share 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 7.29 shows how some of the quantitative characteristics of memory hierarchies can differ. In the rest of this section, we will discuss the common operational aspects of memory hierarchies and how these determine their behavior. We will examine these policies as 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.

#### Question 1: Where Can a Block Be Placed?

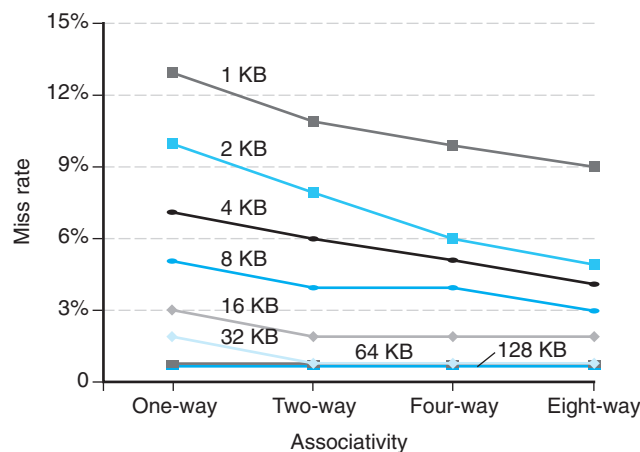
We have seen that block placement in the upper level of the hierarchy can use a range of schemes, from direct mapped to set associative to fully associative. As mentioned above, this entire range of schemes can be thought of as variations on a set-associative scheme where the number of sets and the number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

Scheme name	Number of sets	Blocks per set
Direct mapped	Number of blocks in cache	1
Set associative	<u>Number of blocks in cache</u> 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 7.30 shows the data for a workload consisting of the SPEC2000 benchmarks with caches of 4

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	15,000–50,000	16,000–250,000	40–1024
Total size in kilobytes	16–64	2000–3000	500,000–1,000,000,000	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 7.29** 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 2006. 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.



**FIGURE 7.30** 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 7.15 explains how this data was collected.

KB to 512 KB, varying 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.

### 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 required 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 four 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.
4. The large page size means the page table size overhead is relatively small. (The use of a separate lookup table, like a page table for virtual memory, is not practical for a cache because the table would be much larger than a page table and could not be accessed quickly.)

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.

We have already mentioned the two primary strategies for replacement in set-associative or fully associative caches:

- *Random*: Candidate blocks are randomly selected, possibly using some hardware assistance. For example, MIPS supports random replacement for TLB misses.
- *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 costly. Even for four-way set associativity, LRU is often approximated—for example, by keeping track of which of a 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 is often provided to make it easier for