

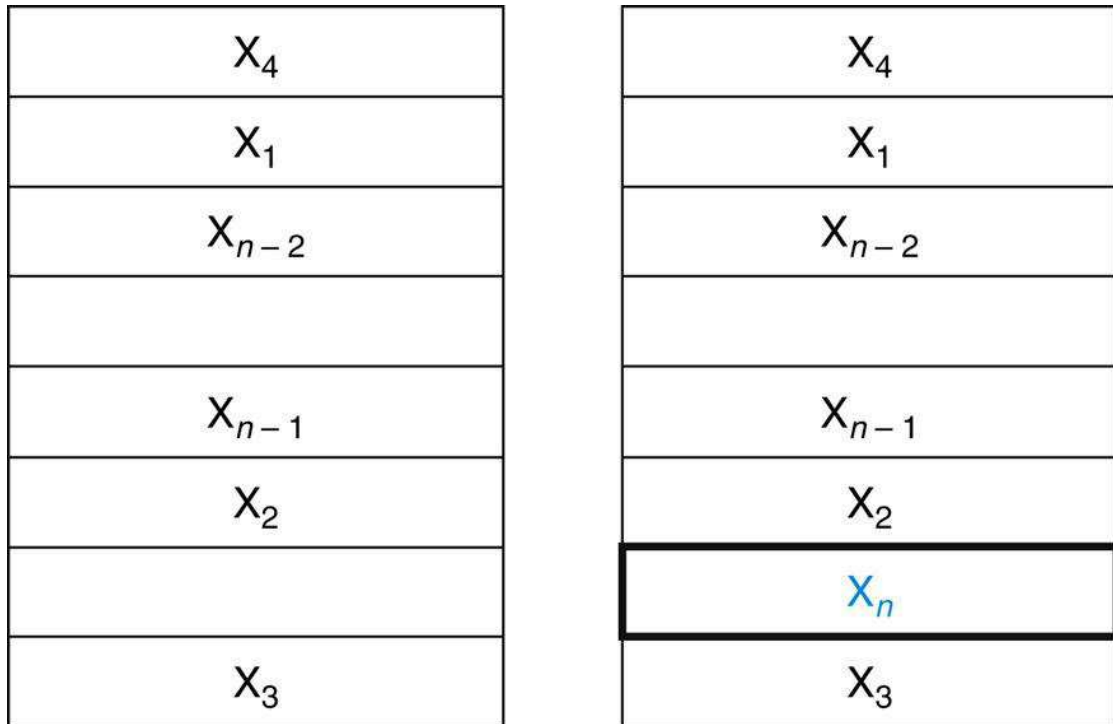
5.3 The Basics of Caches

Cache: a safe place for hiding or storing things.

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

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

In this section, we begin by looking at a very simple cache in which the processor requests are each one word, and the blocks also consist of a single word. (Readers already familiar with cache basics may want to skip to [Section 5.4](#).) [Figure 5.7](#) shows such a simple cache, before and after requesting a data item that is not initially in the cache. Before the request, the cache contains a collection of recent references X_1, X_2, \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.



a. Before the reference to X_n b. After the reference to X_n

FIGURE 5.7 The cache just before and just after a reference to a word X_n that is not initially in the cache.

This reference causes a miss that forces the cache to fetch X_n from memory and insert it into the cache.

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

direct-mapped cache

A cache structure in which each memory location is mapped to

exactly one location in the cache.

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

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

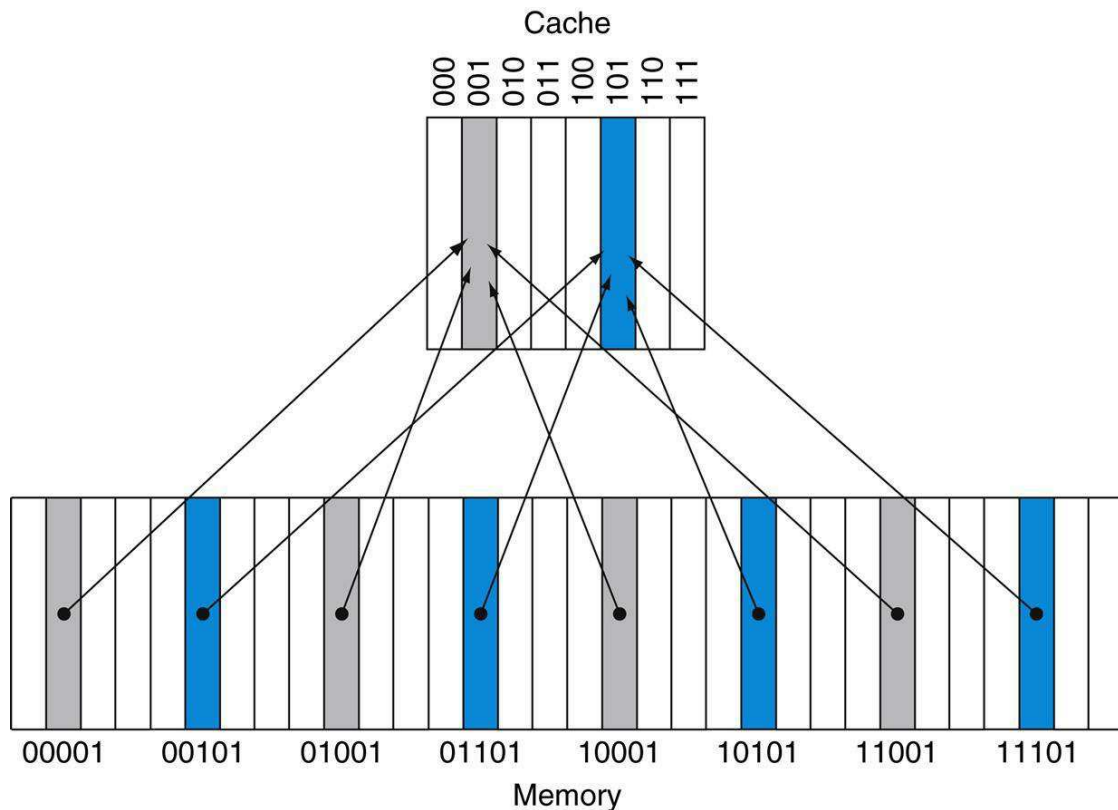


FIGURE 5.8 A direct-mapped cache with eight entries showing the addresses of memory words between 0 and 31 that map to the same cache locations.

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

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

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

tag

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

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

valid bit

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

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

The BIG Picture

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



Accessing a Cache

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

Decimal address of reference	Binary address of reference	Hit or miss in cache	Assigned cache block (where found or placed)
22	10110_{two}	miss (5.9b)	$(10\mathbf{110}_{\text{two}} \bmod 8) = \mathbf{110}_{\text{two}}$
26	11010_{two}	miss (5.9c)	$(11\mathbf{010}_{\text{two}} \bmod 8) = \mathbf{010}_{\text{two}}$
22	10110_{two}	hit	$(10\mathbf{110}_{\text{two}} \bmod 8) = \mathbf{110}_{\text{two}}$
26	11010_{two}	hit	$(11\mathbf{010}_{\text{two}} \bmod 8) = \mathbf{010}_{\text{two}}$
16	10000_{two}	miss (5.9d)	$(10\mathbf{000}_{\text{two}} \bmod 8) = \mathbf{000}_{\text{two}}$
3	00011_{two}	miss (5.9e)	$(00\mathbf{011}_{\text{two}} \bmod 8) = \mathbf{011}_{\text{two}}$
16	10000_{two}	hit	$(10\mathbf{000}_{\text{two}} \bmod 8) = \mathbf{000}_{\text{two}}$
18	10010_{two}	miss (5.9f)	$(10\mathbf{010}_{\text{two}} \bmod 8) = \mathbf{010}_{\text{two}}$
16	10000_{two}	hit	$(10\mathbf{000}_{\text{two}} \bmod 8) = \mathbf{000}_{\text{two}}$

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

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

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

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

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

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

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

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

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

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

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

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

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

The cache is initially empty, with all valid bits (V entry in cache) turned off (N). The processor requests the following addresses: 10110_{two} (miss), 11010_{two} (miss), 10110_{two} (hit), 11010_{two} (hit), 10000_{two} (miss), 00011_{two} (miss), 10000_{two} (hit), 10010_{two} (miss), and 10000_{two} (hit). The figures show the cache contents after each miss in the sequence has been handled. When address 10010_{two} (18) is referenced, the entry for address 11010_{two} (26) must be replaced, and a reference to 11010_{two} will cause a subsequent miss.

The tag field will contain only the upper portion of the address. The full address of a word contained in cache block i with tag field j for this cache is $j \times 8 + i$, or equivalently the concatenation of the tag field j and the index i . For example, in cache *f* above, index 010_{two}

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

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

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

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

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

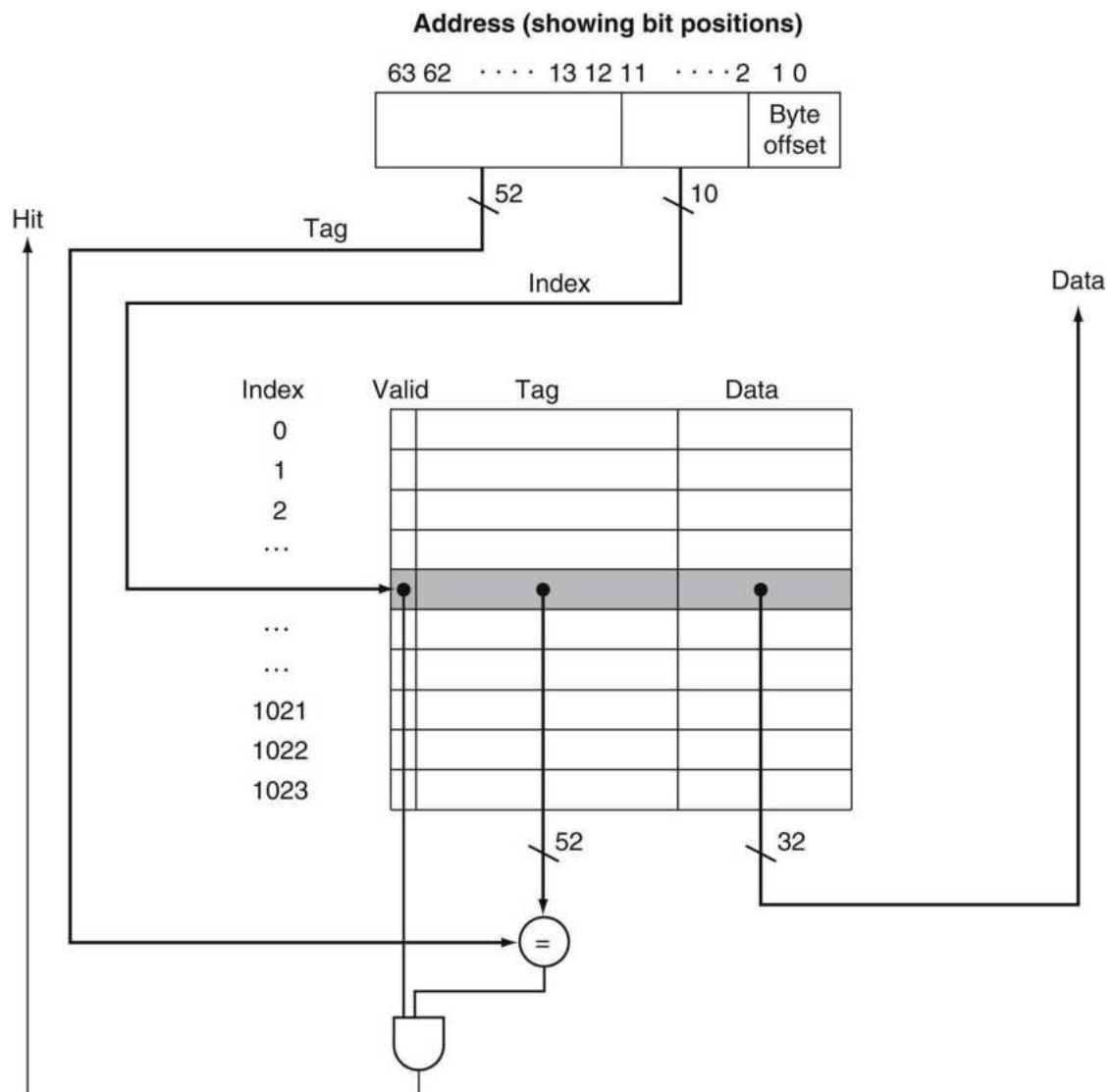


FIGURE 5.10 For this cache, the lower portion of the address is used to select a cache entry consisting of a data word and a tag.

This cache holds 1024 words or 4 KiB. Unless noted otherwise, we assume 64-bit addresses in this chapter. The tag from the cache is compared against the upper portion of the address to determine whether the entry in the cache corresponds to the requested address.

Because the cache has 2^{10} (or 1024) words and a block size of one word, 10 bits are used to index the cache, leaving $64 - 10 - 2 = 52$ bits to be compared against the tag. If the tag and upper 52 bits of the address are equal and the valid bit is on, then the request hits in the cache, and the word is supplied to the processor. Otherwise, a miss occurs.

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

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

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

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

The size of the tag field is

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

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

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

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

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

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

size of the data. Thus, the cache in [Figure 5.10](#) is called a 4 KiB cache.

Bits in a Cache

Example

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

Answer

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

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

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

Mapping an Address to a Multiword Cache Block

Example

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

Answer

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

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

where the address of the block is

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

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

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

and

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

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

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

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

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

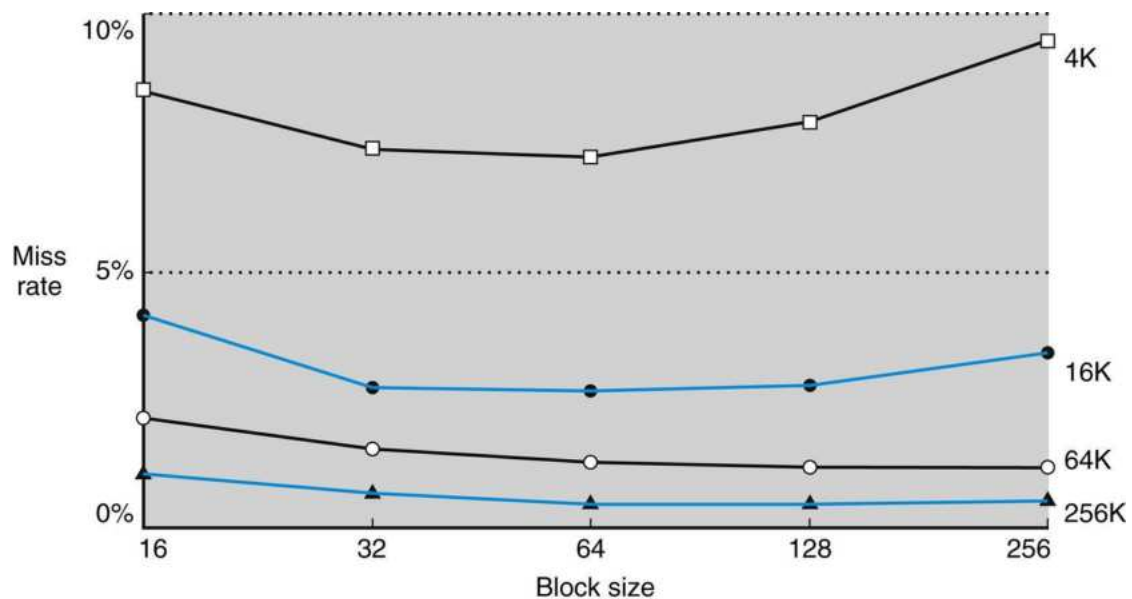


FIGURE 5.11 Miss rate versus block size.

Note that the miss rate actually goes up if the block size is too large relative to the cache size. Each line represents a cache of different size. (This figure is independent of associativity, discussed soon.)

Unfortunately, SPEC CPU2000 traces would take too long if block size were included, so these data are based on SPEC92.

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

Elaboration

Although it is hard to do anything about the longer latency

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

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

Handling Cache Misses

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

cache miss

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

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

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

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

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

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

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

Handling Writes

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

write-through

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

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

Although this design handles writes very simply, it would not provide good performance. With a write-through scheme, every write causes the data to be written to main memory. These writes will take a long time, likely at least 100 processor clock cycles, and could slow down the processor considerably. For example, suppose 10% of the instructions are stores. If the CPI without cache misses was 1.0, spending 100 extra cycles on every write would lead to a CPI of $1.0 + 100 \times 10\% = 11$, 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 they are waiting to be written to memory. After writing the data into the cache and into the write buffer, the processor can continue execution. When a write to main memory completes, the entry in the write buffer is freed. If the write buffer is full when the processor reaches a write, the processor must stall until there is an empty position in the write buffer. Of course, if the rate at which the memory can complete writes is less than the rate at which the processor is generating writes, no amount of buffering can help, because writes are being generated faster than

the memory system can accept them.

write buffer

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

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

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

write-back

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

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

Elaboration

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

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

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

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

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

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

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

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

An Example Cache: The Intrinsicity FastMATH Processor

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

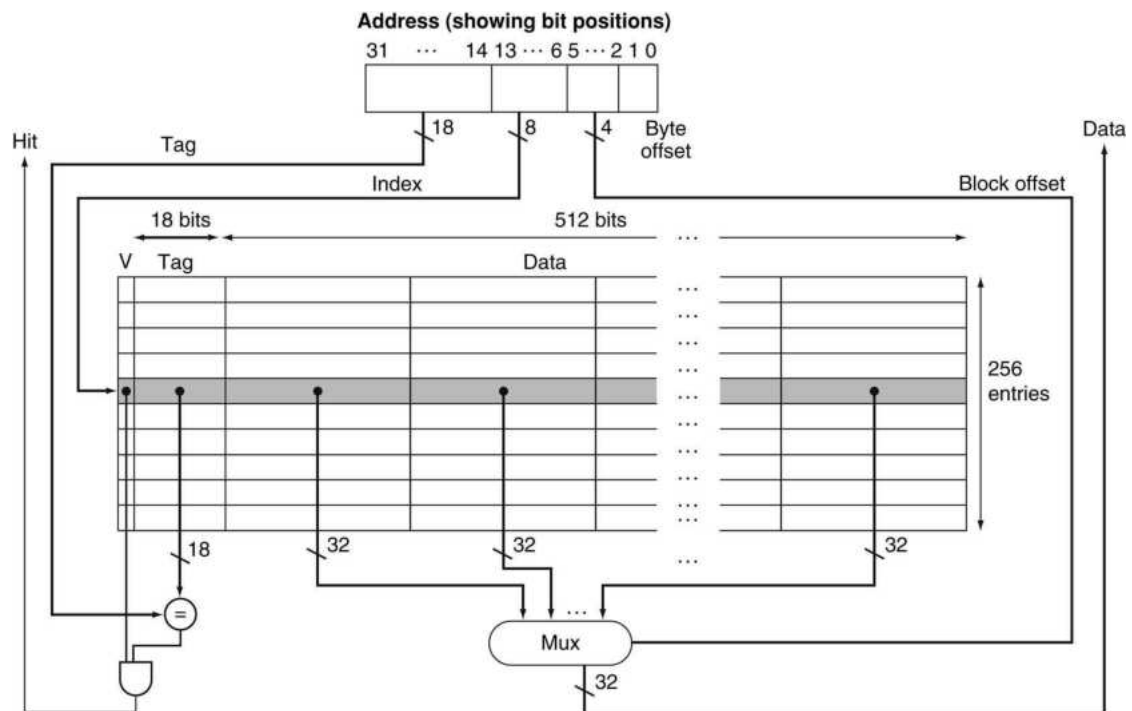


FIGURE 5.12 The 16 KiB caches in the Intrinsity FastMATH each contain 256 blocks with 16 words per block.

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

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

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

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

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

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

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

FIGURE 5.13 Approximate instruction and data miss rates for the Intrinsicity FastMATH processor for SPEC CPU2000 benchmarks.

The combined miss rate is the effective miss rate seen for the combination of the 16 KiB instruction cache and 16 KiB data cache. It is obtained by weighting the instruction and data individual miss rates by the frequency of instruction and data references.

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

Elaboration

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

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

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

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

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

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

split cache

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

Summary

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

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

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

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

Check Yourself

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

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

5.4 Measuring and Improving Cache Performance

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

We can also factor this as

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

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

Calculating Cache Performance

Example

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

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

Answer

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

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

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

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

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

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

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 1](#), reminds us of this fact. A few simple examples show how serious this problem can be. Suppose we speed-up the computer in the previous example by reducing its CPI from 2 to 1 without changing the clock rate, which might be done with an improved pipeline. The system with cache misses

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

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

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

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

to

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

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

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

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

accesses; it is equal to the following:

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

Calculating Average Memory Access Time

Example

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

Answer

The average memory access time per instruction is

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

or 2 ns.

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

Reducing Cache Misses by More Flexible Placement of Blocks

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

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

fully associative cache

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

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

set-associative cache

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

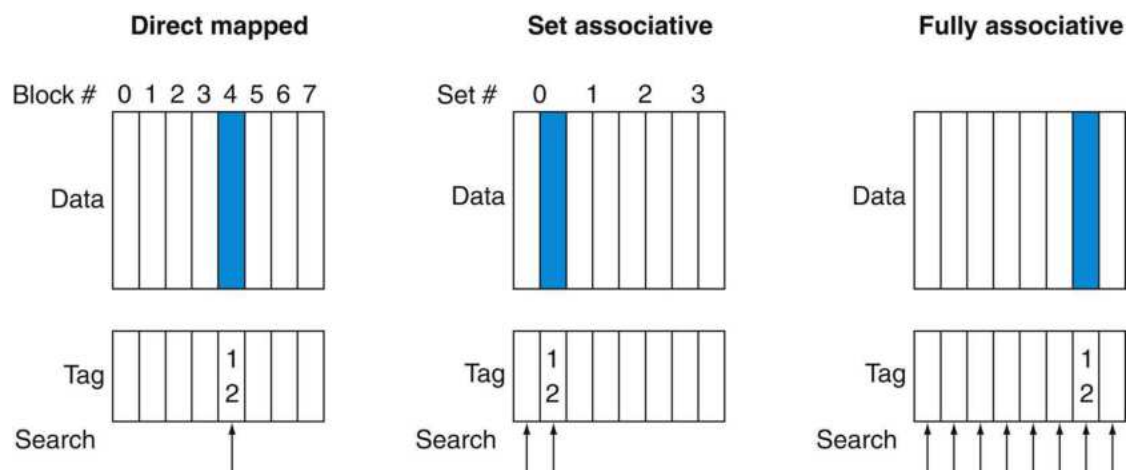


FIGURE 5.14 The location of a memory block whose address is 12 in a cache with eight blocks varies for direct-mapped, set-associative, and fully associative placement.

In direct-mapped placement, there is only one cache block where memory block 12 can be found, and that block is given by $(12 \text{ modulo } 8) = 4$. 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.

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

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

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

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

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.

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

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

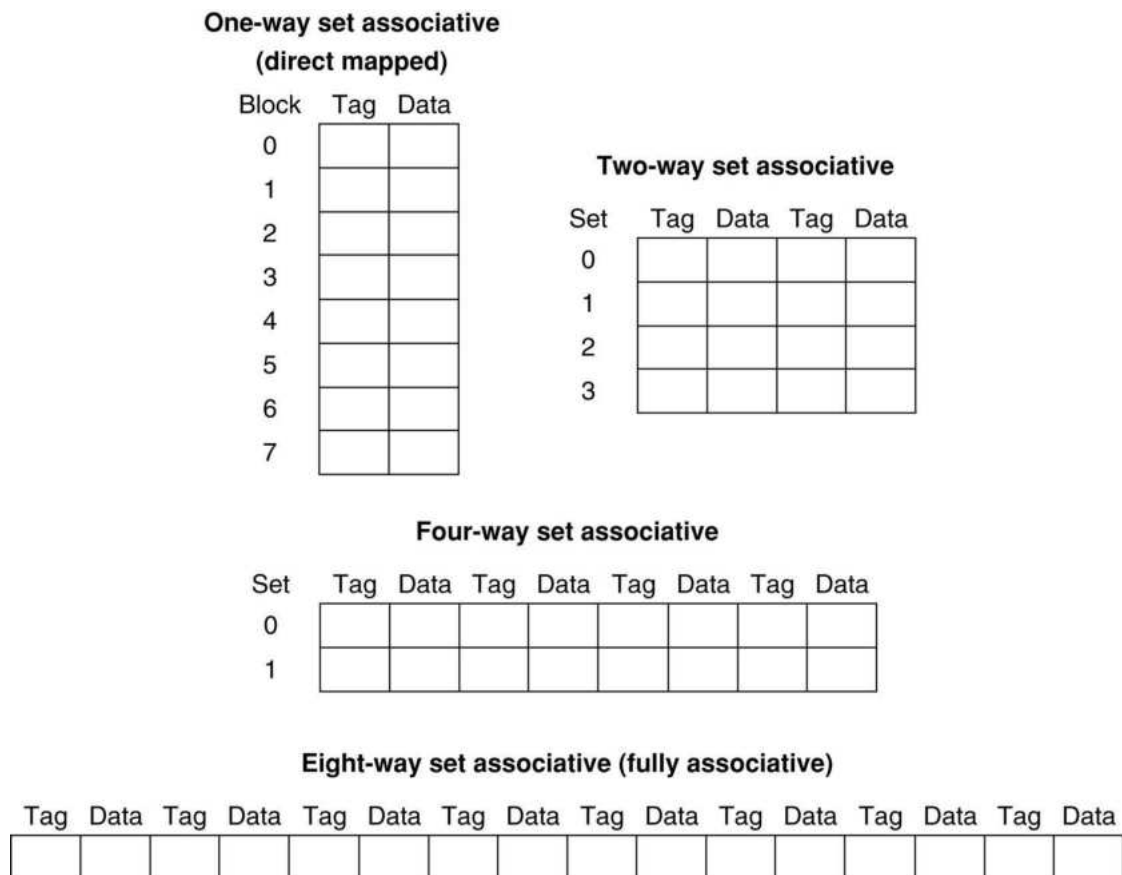


FIGURE 5.15 An eight-block cache configured as direct-mapped, two-way set associative, four-way set associative, and fully associative.

The total size of the cache in blocks is equal to the number of sets times the associativity. Thus, for a fixed cache size, increasing the associativity decreases the number of sets while increasing the number of elements per set. With eight blocks, an eight-way set-associative cache is the same as a fully associative cache.

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

Misses and Associativity in Caches

Example

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

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

Answer

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Locating a Block in the Cache

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

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

FIGURE 5.17 The three portions of an address in a set-associative or direct-mapped cache.

The index is used to select the set, then the tag is used to choose the block by comparison with the blocks in

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

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

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

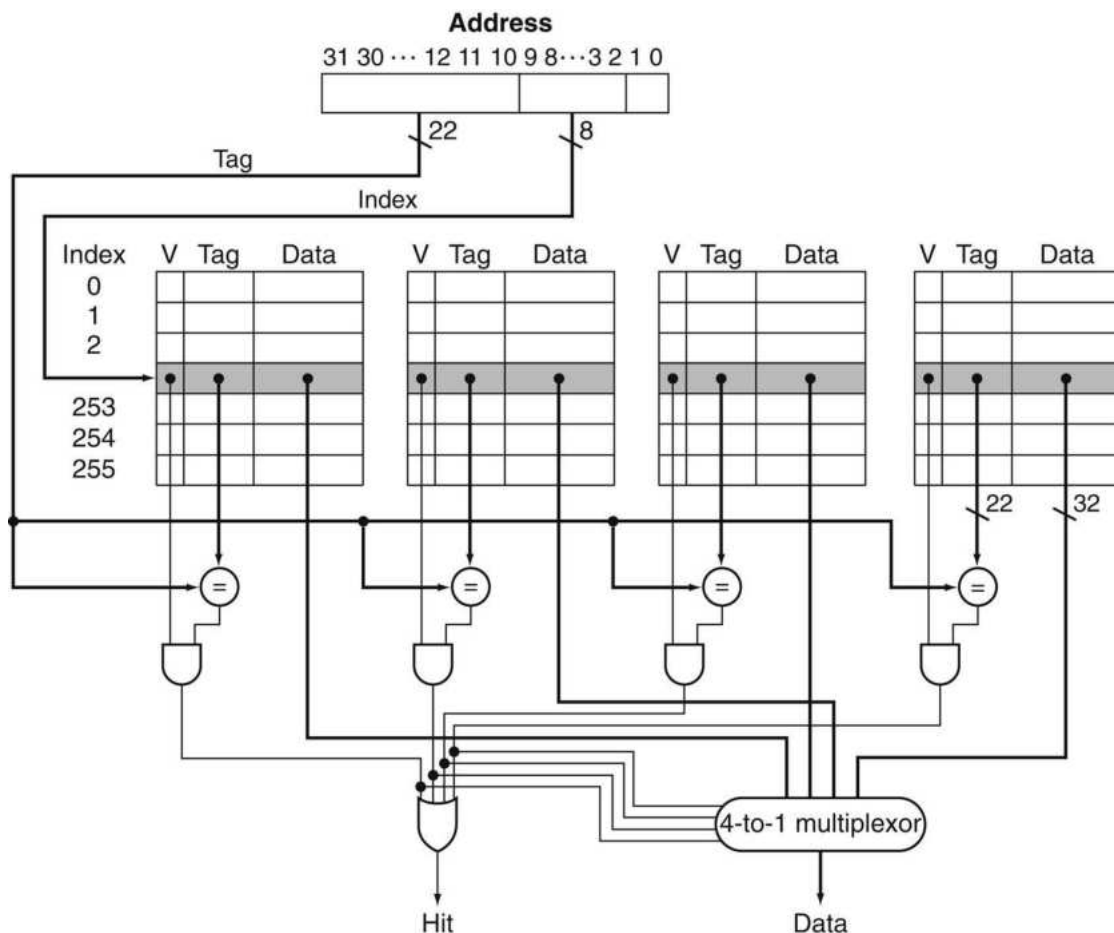


FIGURE 5.18 The implementation of a four-way set-associative cache requires four comparators and a 4-to-1 multiplexor.

The comparators determine which element of the selected set (if any) matches the tag. The output of the comparators is used to select the data from one of the four blocks of the indexed set, using a multiplexor with a decoded select signal. In some implementations, the

Output enable signals on the data portions of the cache RAMs can be used to select the entry in the set that drives the output. The Output enable signal comes from the comparators, causing the element that matches to drive the data outputs. This organization eliminates the need for the multiplexor.

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

Elaboration

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

Choosing Which Block to Replace

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

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

least recently used (LRU)

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

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

for replacement.

Size of Tags versus Set Associativity

Example

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

Answer

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

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

For a fully associative cache, there is only one set with 4096 blocks, and the tag is 60 bits, leading to $60 \times 4096 \times 1 = 246 \text{ K}$ tag bits.

Reducing the Miss Penalty Using Multilevel Caches

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

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

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

Performance of Multilevel Caches

Example

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

Answer

The miss penalty to main memory is

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

The effective CPI with one level of caching is given by

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

For the processor with one level of caching,

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

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

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

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

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

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

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

Thus, the processor with the secondary cache is faster by

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

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

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

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

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

multilevel cache

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

Understanding Program Performance

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

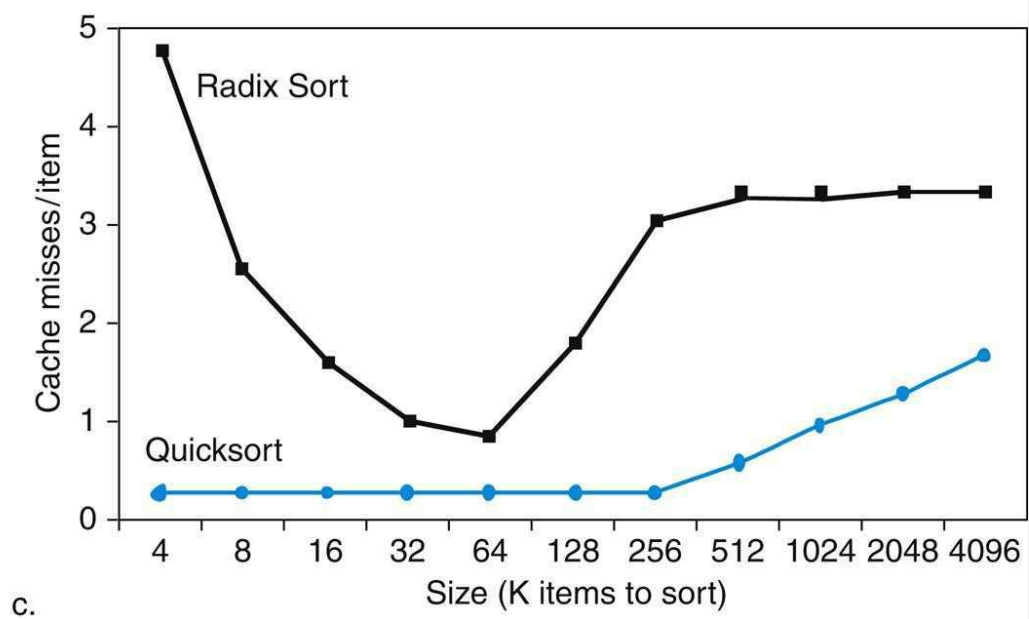
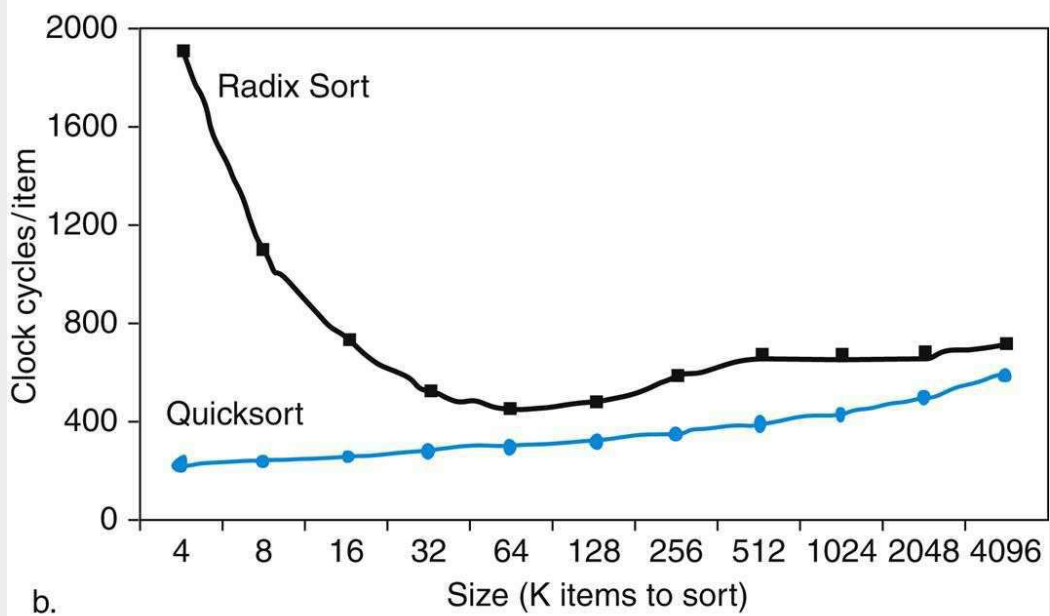
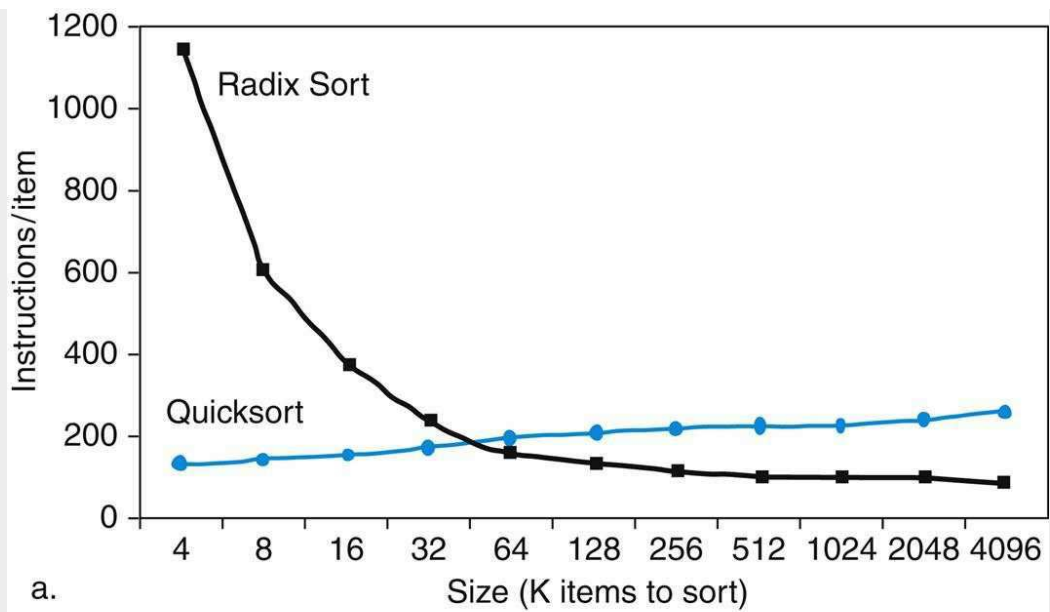
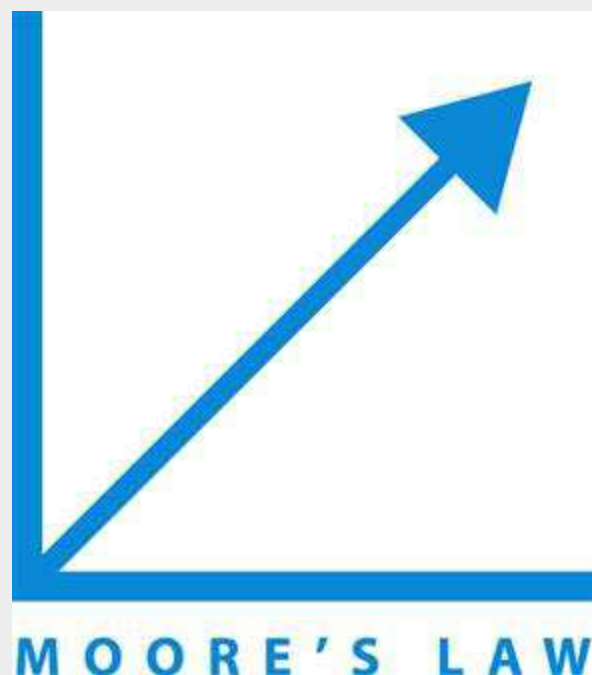


FIGURE 5.19 Comparing Quicksort and Radix Sort by (a) instructions executed per item sorted, (b) time per item sorted, and (c) cache misses per item sorted.

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

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



Software Optimization via Blocking

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

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

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

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

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

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

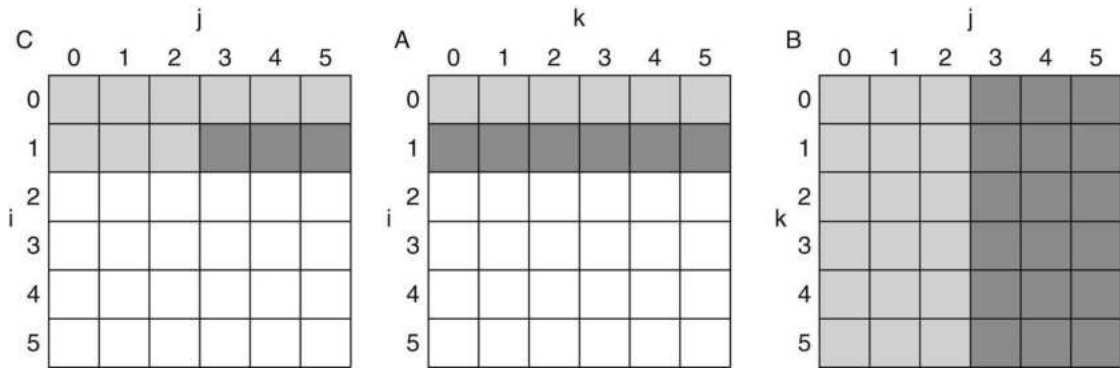


FIGURE 5.20 A snapshot of the three arrays **C**, **A**, and **B** when $N = 6$ and $i = 1$.

The age of accesses to the array elements is indicated by shade: white means not yet touched, light means older accesses, and dark means newer accesses. Compared to [Figure 5.22](#), elements of **A** and **B** are read repeatedly to calculate new elements of **C**. The variables i , j , and k are shown along the rows or columns used to access the arrays.

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

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

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

[Figure 5.21](#) shows the blocked version of DGEMM. The function `do_block` is DGEMM from [Figure 3.22](#) with three new parameters s_i , s_j , and s_k to specify the starting position of each submatrix of **A**, **B**, and **C**. The two inner loops of the `do_block` now compute in steps of size `BLOCKSIZE` rather than the full length of **B** and **C**. The gcc

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

```

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

```

FIGURE 5.21 Cache blocked version of DGEMM in Figure 3.22.

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

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

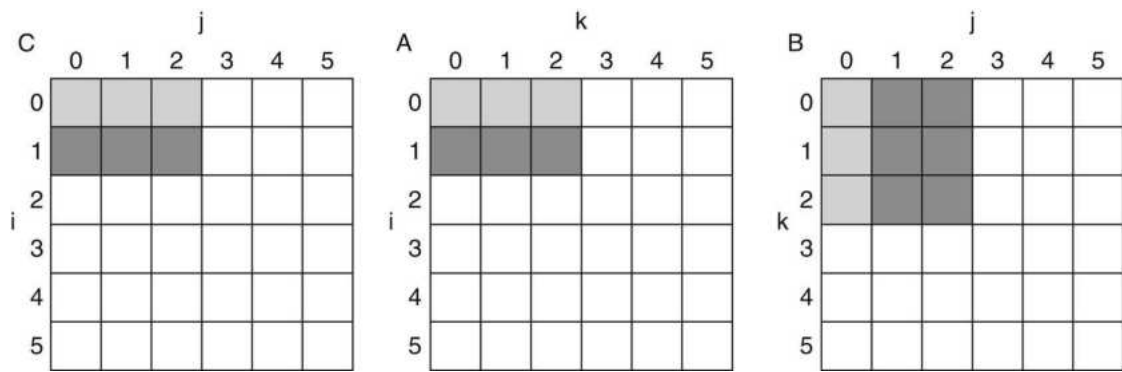


FIGURE 5.22 The age of accesses to the arrays **C**, **A**, and **B** when **BLOCKSIZE = 3**.

Note that, in contrast to [Figure 5.20](#), fewer elements are accessed.

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

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

Elaboration

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

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

global miss rate

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

local miss rate

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

Elaboration

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

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

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

Elaboration

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

particular computer. This approach is called *autotuning*.

Check Yourself

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

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

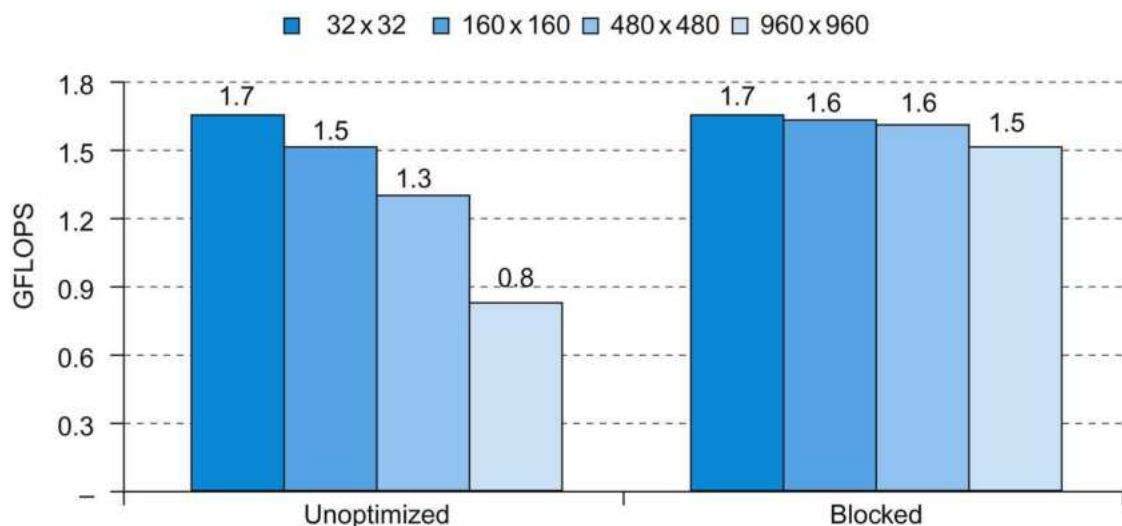


FIGURE 5.23 Performance of unoptimized DGEMM (Figure 3.22) versus cache blocked DGEMM (Figure 5.21) as the matrix dimension varies from 32 × 32 (where all three matrices fit in the cache) to 960 × 960.

Summary

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

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

other critical factors in the memory hierarchy.

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

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

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