Type	What cached	Where cached	Latency (cycles)	Managed by
CPU registers	4-byte or 8-byte words	On-chip CPU registers	0	Compiler
TLB	Address translations	On-chip TLB	0	Hardware MMU
L1 cache	64-byte blocks	On-chip L1 cache	4	Hardware
L2 cache	64-byte blocks	On-chip L2 cache	10	Hardware
L3 cache	64-byte blocks	On-chip L3 cache	50	Hardware
Virtual memory	4-KB pages	Main memory	200	Hardware + OS
Buffer cache	Parts of files	Main memory	200	OS
Disk cache	Disk sectors	Disk controller	100,000	Controller firmware
Network cache	Parts of files	Local disk	10,000,000	NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

Figure 6.23 The ubiquity of caching in modern computer systems. Acronyms: TLB: translation lookaside buffer; MMU: memory management unit; OS: operating system; NFS: network file system.

6.3.2 Summary of Memory Hierarchy Concepts

To summarize, memory hierarchies based on caching work because slower storage is cheaper than faster storage and because programs tend to exhibit locality:

Exploiting temporal locality. Because of temporal locality, the same data objects are likely to be reused multiple times. Once a data object has been copied into the cache on the first miss, we can expect a number of subsequent hits on that object. Since the cache is faster than the storage at the next lower level, these subsequent hits can be served much faster than the original miss.

Exploiting spatial locality. Blocks usually contain multiple data objects. Because of spatial locality, we can expect that the cost of copying a block after a miss will be amortized by subsequent references to other objects within that block.

Caches are used everywhere in modern systems. As you can see from Figure 6.23, caches are used in CPU chips, operating systems, distributed file systems, and on the World Wide Web. They are built from and managed by various combinations of hardware and software. Note that there are a number of terms and acronyms in Figure 6.23 that we haven't covered yet. We include them here to demonstrate how common caches are.

6.4 Cache Memories

The memory hierarchies of early computer systems consisted of only three levels: CPU registers, main memory, and disk storage. However, because of the increasing gap between CPU and main memory, system designers were compelled to insert

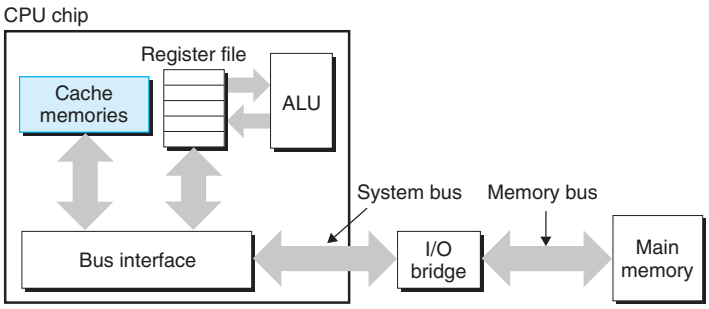


Figure 6.24
Typical bus structure for cache memories.

a small SRAM *cache memory*, called an *L1 cache* (level 1 cache) between the CPU register file and main memory, as shown in Figure 6.24. The L1 cache can be accessed nearly as fast as the registers, typically in about 4 clock cycles.

As the performance gap between the CPU and main memory continued to increase, system designers responded by inserting an additional larger cache, called an *L2 cache*, between the L1 cache and main memory, that can be accessed in about 10 clock cycles. Many modern systems include an even larger cache, called an *L3 cache*, which sits between the L2 cache and main memory in the memory hierarchy and can be accessed in about 50 cycles. While there is considerable variety in the arrangements, the general principles are the same. For our discussion in the next section, we will assume a simple memory hierarchy with a single L1 cache between the CPU and main memory.

6.4.1 Generic Cache Memory Organization

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

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

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

The parameters S and B induce a partitioning of the m address bits into the three fields shown in Figure 6.25(b). The s *set index bits* in A form an index into

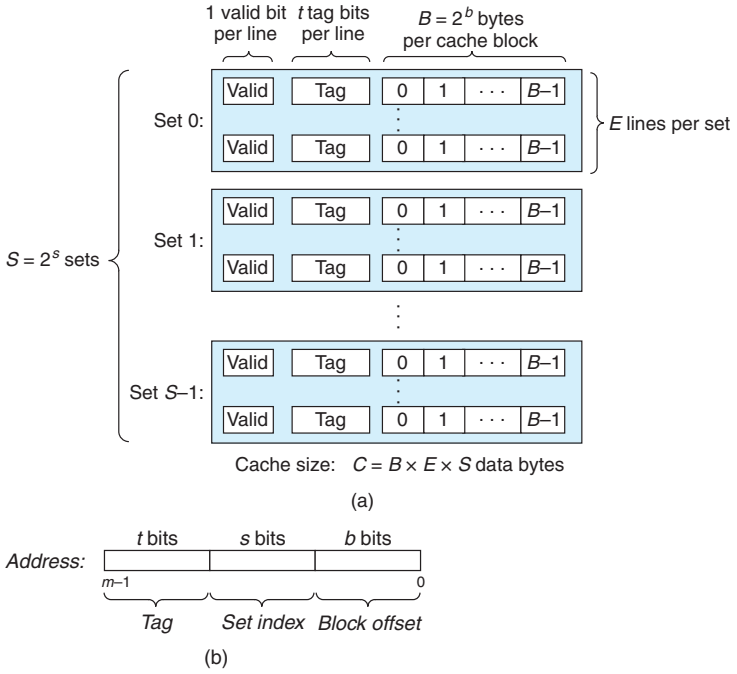


Figure 6.25
General organization of cache (S, E, B, m). (a) A cache is an array of sets. Each set contains one or more lines. Each line contains a valid bit, some tag bits, and a block of data. (b) The cache organization induces a partition of the m address bits into t tag bits, s set index bits, and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

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

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

Practice Problem 6.9 (solution page 699)

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

Cache	m	C	B	E	S	t	s	b
1.	32	1,024	4	1				
2.	32	1,024	8	4				
3.	32	1,024	32	32				

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

Figure 6.26 **Summary of cache parameters.**

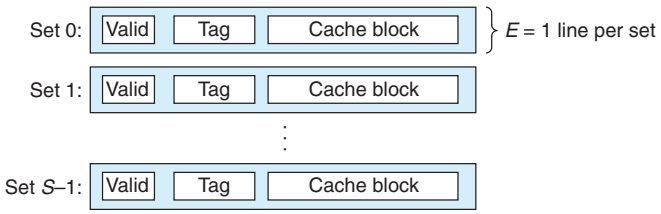


Figure 6.27 **Direct-mapped cache ($E = 1$).** There is exactly one line per set.

6.4.2 Direct-Mapped Caches

Caches are grouped into different classes based on E, the number of cache lines per set. A cache with exactly one line per set ($E = 1$) is known as a *direct-mapped* cache (see Figure 6.27). Direct-mapped caches are the simplest both to implement and to understand, so we will use them to illustrate some general concepts about how caches work.

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

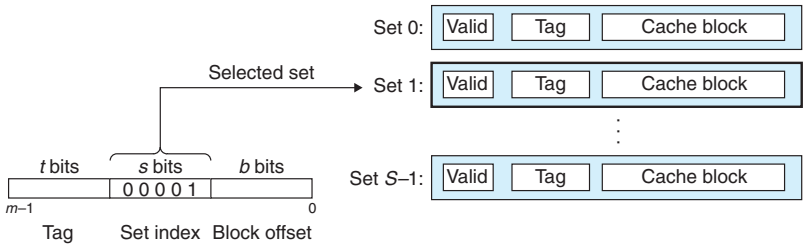


Figure 6.28
Set selection in a direct-mapped cache.

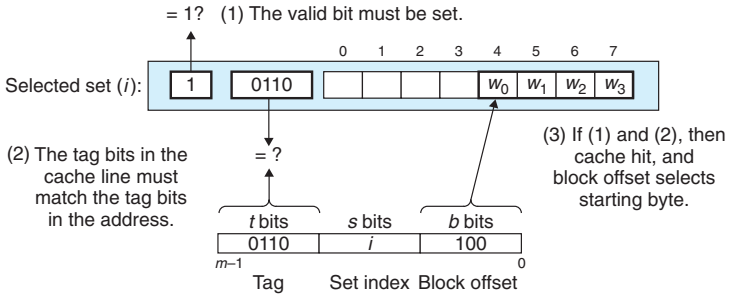


Figure 6.29
Line matching and word selection in a direct-mapped cache. Within the cache block, w_0 denotes the low-order byte of the word w, w_1 the next byte, and so on.

Set Selection in Direct-Mapped Caches

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

Line Matching in Direct-Mapped Caches

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

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

Word Selection in Direct-Mapped Caches

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

Line Replacement on Misses in Direct-Mapped Caches

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

Putting It Together: A Direct-Mapped Cache in Action

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

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

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

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

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

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

Figure 6.30 4-bit address space for example direct-mapped cache.

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

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

Set	Valid	Tag	block[0]	block[1]
0	0			
1	0			
2	0			
3	0			

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

1. **Read word at address 0.** Since the valid bit for set 0 is 0, this is a cache miss. The cache fetches block 0 from memory (or a lower-level cache) and stores the

block in set 0. Then the cache returns m[0] (the contents of memory location 0) from block[0] of the newly fetched cache line.

Set	Valid	Tag	block[0]	block[1]
0	1	0	m[0]	m[1]
1	0			
2	0			
3	0			

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

Set	Valid	Tag	block[0]	block[1]
0	1	0	m[0]	m[1]
1	0			
2	1	1	m[12]	m[13]
3	0			

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

Set	Valid	Tag	block[0]	block[1]
0	1	1	m[8]	m[9]
1	0			
2	1	1	m[12]	m[13]
3	0			

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

Set	Valid	Tag	block[0]	block[1]
0	1	0	m[0]	m[1]
1	0			
2	1	1	m[12]	m[13]
3	0			

Conflict Misses in Direct-Mapped Caches

Conflict misses are common in real programs and can cause baffling performance problems. Conflict misses in direct-mapped caches typically occur when programs access arrays whose sizes are a power of 2. For example, consider a function that computes the dot product of two vectors:

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

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

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

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

At run time, the first iteration of the loop references `x[0]`, a miss that causes the block containing `x[0]`–`x[3]` to be loaded into set 0. The next reference is to `y[0]`, another miss that causes the block containing `y[0]`–`y[3]` to be copied into set 0, overwriting the values of `x` that were copied in by the previous reference. During the next iteration, the reference to `x[1]` misses, which causes the `x[0]`–`x[3]` block to be loaded back into set 0, overwriting the `y[0]`–`y[3]` block. So now we have a conflict miss, and in fact each subsequent reference to `x` and `y` will result in a conflict miss as we thrash back and forth between blocks of `x` and `y`. The term *thrashing* describes any situation where a cache is repeatedly loading and evicting the same sets of cache blocks.

Aside Why index with the middle bits?

You may be wondering why caches use the middle bits for the set index instead of the high-order bits. There is a good reason why the middle bits are better. Figure 6.31 shows why. If the high-order bits are used as an index, then some contiguous memory blocks will map to the same cache set. For example, in the figure, the first four blocks map to the first cache set, the second four blocks map to the second set, and so on. If a program has good spatial locality and scans the elements of an array sequentially, then the cache can only hold a block-size chunk of the array at any point in time. This is an inefficient use of the cache. Contrast this with middle-bit indexing, where adjacent blocks always map to different cache sets. In this case, the cache can hold an entire C-size chunk of the array, where C is the cache size.

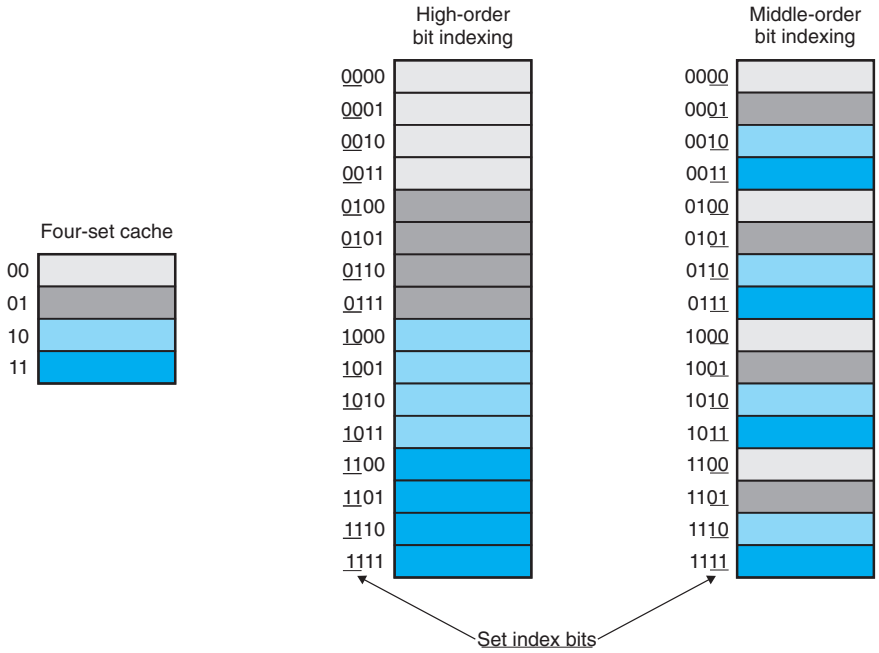


Figure 6.31 **Why caches index with the middle bits.**

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

Luckily, thrashing is easy for programmers to fix once they recognize what is going on. One easy solution is to put B bytes of padding at the end of each array.

For example, instead of defining `x` to be `float x[8]`, we define it to be `float x[12]`. Assuming `y` starts immediately after `x` in memory, we have the following mapping of array elements to sets:

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

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

Practice Problem 6.10 **(solution page 699)**

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

Practice Problem 6.11 **(solution page 699)**

Imagine a hypothetical cache that uses the high-order s bits of an address as the set index. For such a cache, contiguous chunks of memory blocks are mapped to the same cache set.

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

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

```
int array[4096];

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

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

6.4.3 Set Associative Caches

The problem with conflict misses in direct-mapped caches stems from the constraint that each set has exactly one line (or in our terminology, $E = 1$). A *set associative cache* relaxes this constraint so that each set holds more than one cache line. A cache with $1 < E < C/B$ is often called an E-way set associative cache. We

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

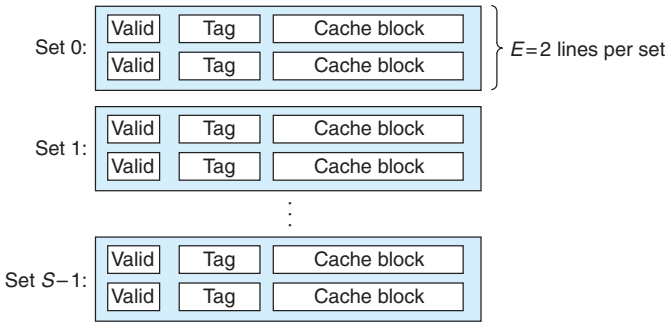
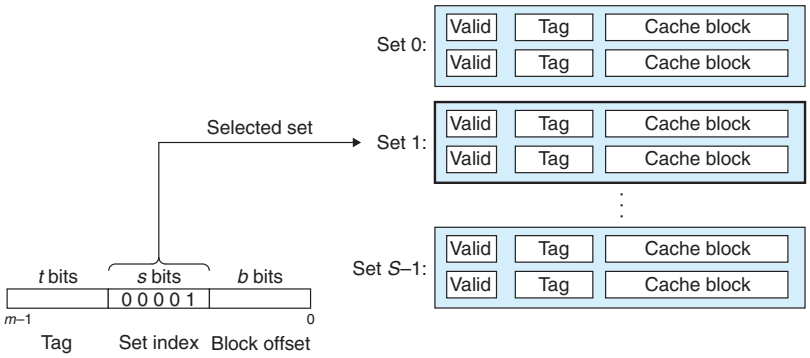


Figure 6.33
Set selection in a set associative cache.



will discuss the special case, where $E = C/B$, in the next section. Figure 6.32 shows the organization of a two-way set associative cache.

Set Selection in Set Associative Caches

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

Line Matching and Word Selection in Set Associative Caches

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

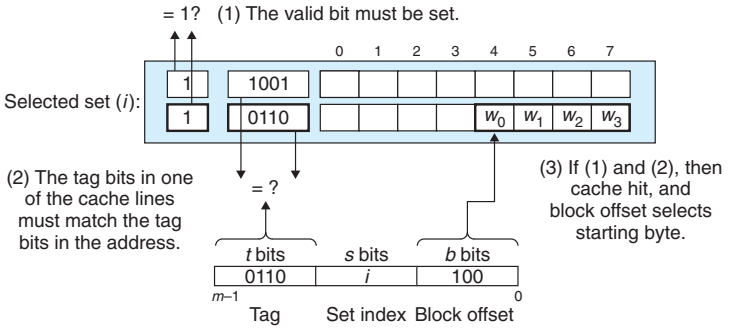


Figure 6.34
Line matching and word selection in a set associative cache.

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

Line Replacement on Misses in Set Associative Caches

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

It is very difficult for programmers to exploit knowledge of the cache replacement policy in their codes, so we will not go into much detail about it here. The simplest replacement policy is to choose the line to replace at random. Other more sophisticated policies draw on the principle of locality to try to minimize the probability that the replaced line will be referenced in the near future. For example, a *least frequently used (LFU)* policy will replace the line that has been referenced the fewest times over some past time window. A *least recently used (LRU)* policy will replace the line that was last accessed the furthest in the past. All of these policies require additional time and hardware. But as we move further down the memory hierarchy, away from the CPU, the cost of a miss becomes more expensive and it becomes more worthwhile to minimize misses with good replacement policies.

6.4.4 Fully Associative Caches

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

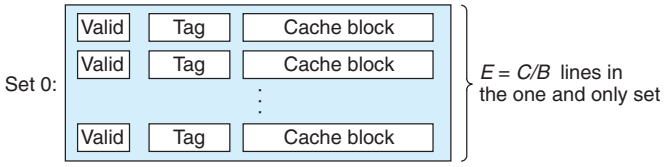


Figure 6.35
Fully associative cache ($E = C/B$). In a fully associative cache, a single set contains all of the lines.

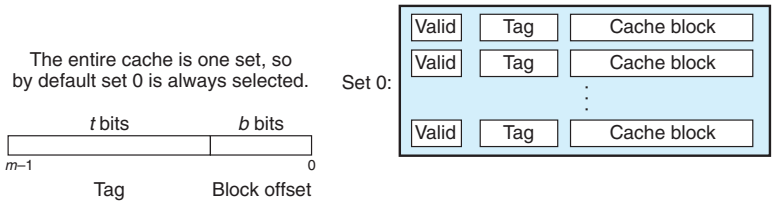


Figure 6.36
Set selection in a fully associative cache. Notice that there are no set index bits.

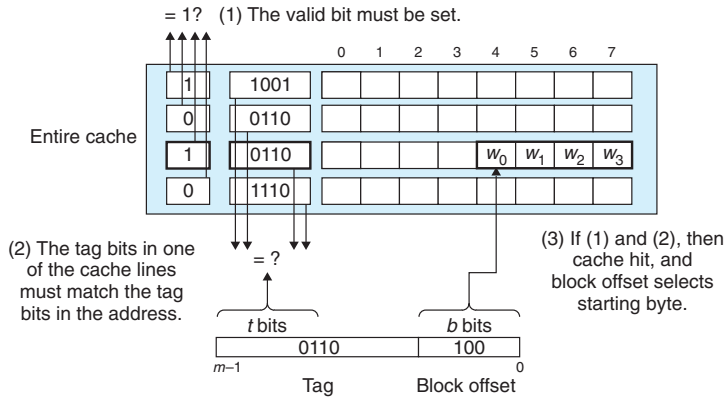


Figure 6.37
Line matching and word selection in a fully associative cache.

Set Selection in Fully Associative Caches

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

Line Matching and Word Selection in Fully Associative Caches

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

Because the cache circuitry must search for many matching tags in parallel, it is difficult and expensive to build an associative cache that is both large and fast. As a result, fully associative caches are only appropriate for small caches, such

as the translation lookaside buffers (TLBs) in virtual memory systems that cache page table entries (Section 9.6.2).

Practice Problem 6.12 (solution page 699)

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

- The memory is byte addressable.
- Memory accesses are to 1-byte words (not to 4-byte words).
- Addresses are 13 bits wide.
- The cache is two-way set associative ($E = 2$), with a 4-byte block size ($B = 4$) and eight sets ($S = 8$).

The contents of the cache are as follows, with all numbers given in hexadecimal notation.

2-way set associative cache

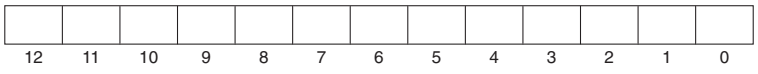
	Line 0						Line 1					
Set index	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3
0	09	1	86	30	3F	10	00	0	—	—	—	—
1	45	1	60	4F	E0	23	38	1	00	BC	0B	37
2	EB	0	—	—	—	—	0B	0	—	—	—	—
3	06	0	—	—	—	—	32	1	12	08	7B	AD
4	C7	1	06	78	07	C5	05	1	40	67	C2	3B
5	71	1	0B	DE	18	4B	6E	0	—	—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

The following figure shows the format of an address (1 bit per box). Indicate (by labeling the diagram) the fields that would be used to determine the following:

CO. The cache block offset

CI. The cache set index

CT. The cache tag



Practice Problem 6.13 (solution page 700)

Suppose a program running on the machine in Problem 6.12 references the 1-byte word at address `0x0D53`. Indicate the cache entry accessed and the cache byte

value returned in hexadecimal notation. Indicate whether a cache miss occurs. If there is a cache miss, enter "—" for "Cache byte returned."

A. Address format (1 bit per box):

B. Memory reference:

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

Practice Problem 6.14 (solution page 700)

Repeat Problem 6.13 for memory address `0x0CB4`.

A. Address format (1 bit per box):

B. Memory reference:

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

Practice Problem 6.15 (solution page 700)

Repeat Problem 6.13 for memory address `0x0A31`.

A. Address format (1 bit per box):

B. Memory reference:

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

Practice Problem 6.16 (solution page 701)

For the cache in Problem 6.12, list all of the hexadecimal memory addresses that will hit in set 3.

6.4.5 Issues with Writes

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

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

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

Optimizing caches for writes is a subtle and difficult issue, and we are only scratching the surface here. The details vary from system to system and are often proprietary and poorly documented. To the programmer trying to write reason-

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

6.4.6 Anatomy of a Real Cache Hierarchy

So far, we have assumed that caches hold only program data. But, in fact, caches can hold instructions as well as data. A cache that holds instructions only is called an *i-cache*. A cache that holds program data only is called a *d-cache*. A cache that holds both instructions and data is known as a *unified cache*. Modern processors include separate i-caches and d-caches. There are a number of reasons for this. With two separate caches, the processor can read an instruction word and a data word at the same time. I-caches are typically read-only, and thus simpler. The two caches are often optimized to different access patterns and can have different block sizes, associativities, and capacities. Also, having separate caches ensures that data accesses do not create conflict misses with instruction accesses, and vice versa, at the cost of a potential increase in capacity misses.

Figure 6.38 shows the cache hierarchy for the Intel Core i7 processor. Each CPU chip has four cores. Each core has its own private L1 i-cache, L1 d-cache, and L2 unified cache. All of the cores share an on-chip L3 unified cache. An interesting feature of this hierarchy is that all of the SRAM cache memories are contained in the CPU chip.

Figure 6.39 summarizes the basic characteristics of the Core i7 caches.

6.4.7 Performance Impact of Cache Parameters

Cache performance is evaluated with a number of metrics:

Miss rate. The fraction of memory references during the execution of a program, or a part of a program, that miss. It is computed as # *misses*/# *references*.

Hit rate. The fraction of memory references that hit. It is computed as $1 - $ *miss rate*.

Hit time. The time to deliver a word in the cache to the CPU, including the time for set selection, line identification, and word selection. Hit time is on the order of several clock cycles for L1 caches.

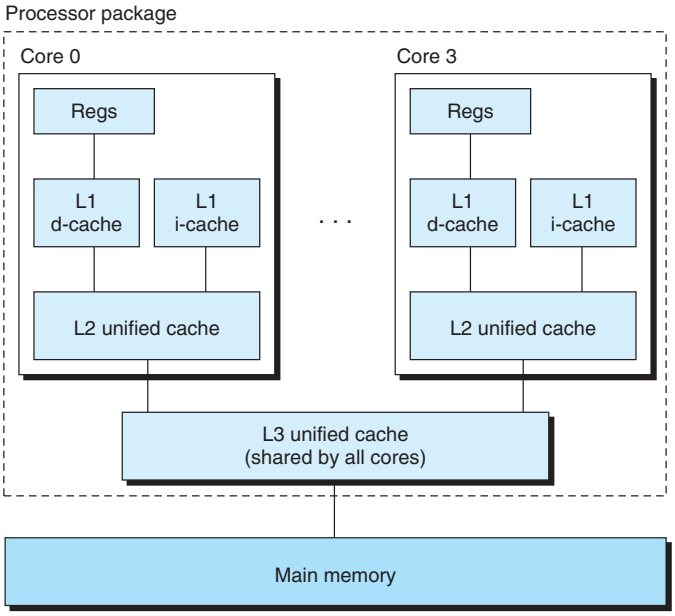


Figure 6.38 **Intel Core i7 cache hierarchy.**

Cache type	Access time (cycles)	Cache size (C)	Assoc. (E)	Block size (B)	Sets (S)
L1 i-cache	4	32 KB	8	64 B	64
L1 d-cache	4	32 KB	8	64 B	64
L2 unified cache	10	256 KB	8	64 B	512
L3 unified cache	40–75	8 MB	16	64 B	8,192

Figure 6.39 **Characteristics of the Intel Core i7 cache hierarchy.**

Miss penalty. Any additional time required because of a miss. The penalty for L1 misses served from L2 is on the order of 10 cycles; from L3, 50 cycles; and from main memory, 200 cycles.

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

Impact of Cache Size

On the one hand, a larger cache will tend to increase the hit rate. On the other hand, it is always harder to make large memories run faster. As a result, larger caches tend to increase the hit time. This explains why an L1 cache is smaller than an L2 cache, and an L2 cache is smaller than an L3 cache.

Impact of Block Size

Large blocks are a mixed blessing. On the one hand, larger blocks can help increase the hit rate by exploiting any spatial locality that might exist in a program. However, for a given cache size, larger blocks imply a smaller number of cache lines, which can hurt the hit rate in programs with more temporal locality than spatial locality. Larger blocks also have a negative impact on the miss penalty, since larger blocks cause larger transfer times. Modern systems such as the Core i7 compromise with cache blocks that contain 64 bytes.

Impact of Associativity

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

The choice of associativity ultimately boils down to a trade-off between the hit time and the miss penalty. Traditionally, high-performance systems that pushed the clock rates would opt for smaller associativity for L1 caches (where the miss penalty is only a few cycles) and a higher degree of associativity for the lower levels, where the miss penalty is higher. For example, in Intel Core i7 systems, the L1 and L2 caches are 8-way associative, and the L3 cache is 16-way.

Impact of Write Strategy

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

6.5 Writing Cache-Friendly Code

In Section 6.2, we introduced the idea of locality and talked in qualitative terms about what constitutes good locality. Now that we understand how cache memories work, we can be more precise. Programs with better locality will tend to have lower miss rates, and programs with lower miss rates will tend to run faster than programs with higher miss rates. Thus, good programmers should always try to