5.3 The basics of caches

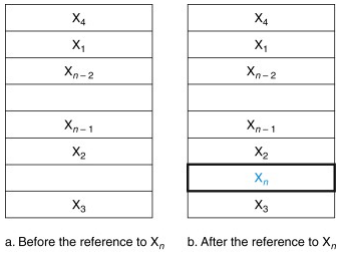
> "Cache: a safe place for hiding or storing things.
> *Webster's New World Dictionary of the American Language, Third College Edition, 1988*

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 COD Chapter 4 (The Processor) 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 COD Section 5.4 (Measuring and improving cache performance)). The figure below shows such a simple cache, before and after requesting a data item that is not initially in the cache. Before the request, the cache contains a collection of recent references X_1 , X_2 , ..., X_{n-1} , and the processor requests a word X_n that is not in the cache. This request results in a miss, and the word X_n is brought from memory into the cache.

Figure 5.3.1: The cache just before and just after a reference to a word Xn that is not initially in the cache (COD Figure 5.7).

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



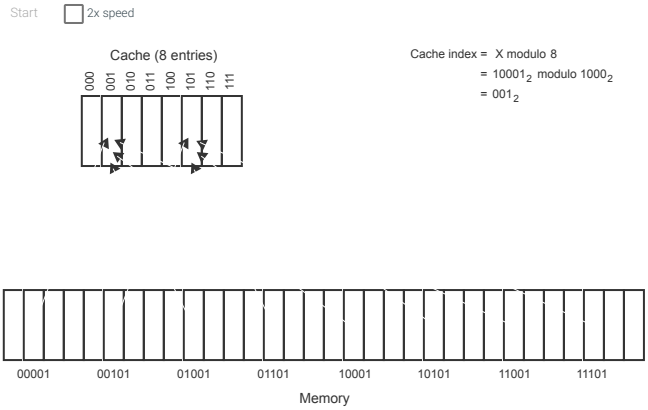
In looking at the scenario in the figure above, there are two questions to answer: How do we know if a data item is in the cache? Moreover, if it is, how do we find it? The answers are related. If each word can go in exactly one place in the cache, then it is straightforward to find the word if it is 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:

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

Direct-mapped cache: A cache structure in which each memory location is mapped to exactly one location 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, the animation below 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.

PARTICIPATION ACTIVITY 5.3.1: A direct-mapped cache with eight entries showing the addresses of memory words between 0 and 31 that map to the same cache locations (COD Figure 5.8).



PARTICIPATION ACTIVITY 5.3.2: Cache index.

Determine the cache index given the direct-mapped cache size and block address.
Type the cache index as a binary value. Ex: 110

1) Direct-mapped cache size: 8 one-word blocks
Block address: 00011_2

Check Show answer

2) Direct-mapped cache size: 8 one-word blocks
Block address: 10101_2

Check Show answer

3) Direct-mapped cache size: 8 one-word blocks
Block address: 10000101_2

Check Show answer

4) Direct-mapped cache size: 16 one-word blocks
Block address: 00101100_2

Check Show answer

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 the animation above 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 COD Figure 5.7 (The cache just before and just after a reference to a word ...). 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.

PARTICIPATION ACTIVITY 5.3.3: Accessing a cache using the tag and valid bits.

Start 2x speed

Memory reference:

10111_{two} hit
10000_{two} miss (fetched from memory)
11101_{two} miss

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	Y	11_{two}	Memory (11001_{two})
010	N		
011	N		
100	Y	00_{two}	Memory (00100_{two})
101	N		
110	N		
111	Y	10_{two}	Memory(10111_{two})

Invalid 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 COD Figure 5.46 (The L1, L2, and L3 data cache miss rates ...)).

Accessing a cache

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

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

Since the cache is empty, several of the first references are misses; the caption of the figure below 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.

Figure 5.3.2: 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 in the above table (COD Figure 5.9).

The cache is initially empty, with all valid bits (V entry in cache) turned off (N). The processor requests the following addresses: 10110_{two} (miss), 11010_{two} (miss), 10110_{two} (hit), 11010_{two} (hit), 10000_{two} (miss), 00011_{two} (miss), 10000_{two} (hit), 10010_{two} (miss), and 10000_{two} (hit). The figures show the cache contents after each miss in the sequence has been handled. When address 10010_{two} (18) is referenced, the entry for address 11010_{two} (26) must be replaced, and a reference to 11010_{two} will cause a subsequent miss. The tag field will contain only the upper portion of the address. The full address of a word contained in cache block *i* with tag field *j* for this cache is *j* × 8 + *i*, or equivalently the concatenation of the tag field *j* and the index *i*. For example, in cache *f* below, index 010_{two} has tag 10_{two} and corresponds to address 10010_{two}.

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

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

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

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

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

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

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

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

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

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

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

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

PARTICIPATION ACTIVITY 5.3.4: Cache hits and misses.

1) A request to address 00101_{two} results in a cache _____.

Index	V	Tag	Data
000	N		
001	Y	00_{two}	Memory (00001_{two})
010	N		
011	Y	11_{two}	Memory (11011_{two})
100	Y	11_{two}	Memory (11100_{two})
101	N		
110	Y	01_{two}	Memory (01110_{two})
111	Y	10_{two}	Memory (10111_{two})

- ○ hit
- ○ miss

2) After a request to address 00110_{two}, the tag in cache block 110_{two} is _____two.

Index	V	Tag	Data
000	N		
001	Y	00_{two}	Memory (00001_{two})
010	N		
011	N		
100	Y	11_{two}	Memory (11100_{two})
101	N		
110	N		
111	Y	10_{two}	Memory (10111_{two})

- ○ 10
- ○ 00

3) A request to address 00001_{two} results in a cache _____.

Index	V	Tag	Data
000	Y	01_{two}	Memory (01000_{two})
001	Y	11_{two}	Memory (11001_{two})
010	Y	01_{two}	Memory (01010_{two})
011	Y	00_{two}	Memory (00011_{two})
100	N		
101	N		
110	N		
111	N		

- ○ hit
- ○ miss

4) After a request to address 00101_{two}, the data in cache block 101_{two} is Memory(_____two).

Index	V	Tag	Data
000	Y	01_{two}	Memory (01000_{two})
001	N		
010	N		
011	Y	00_{two}	Memory (00011_{two})
100	N		
101	Y	11_{two}	Memory (11101_{two})
110	Y	00_{two}	Memory (00110_{two})
111	N		

- ○ 11101
- ○ 00101

5) A request to address 10111_{two} results in a cache _____.

Index	V	Tag	Data
000	N		
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	N		
100	Y	10_{two}	Memory (10100_{two})
101	N		
110	Y	00_{two}	Memory (00110_{two})
111	Y	10_{two}	Memory (10111_{two})

- ○ hit
- ○ miss

6) After a request to address 10000_{two}, the data in cache block 000_{two} _____.

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	Y	00_{two}	Memory (00001_{two})
010	Y	11_{two}	Memory (11010_{two})
011	Y	11_{two}	Memory (11011_{two})
100	Y	10_{two}	Memory (10100_{two})
101	Y	00_{two}	Memory (00101_{two})
110	Y	00_{two}	Memory (00110_{two})
111	Y	10_{two}	Memory (10111_{two})

- ○ is empty
- ○ does not change

7) Cache block 111_{two} with tag 00_{two} corresponds to memory address ——two.

- ○ 11100
- ○ 00111

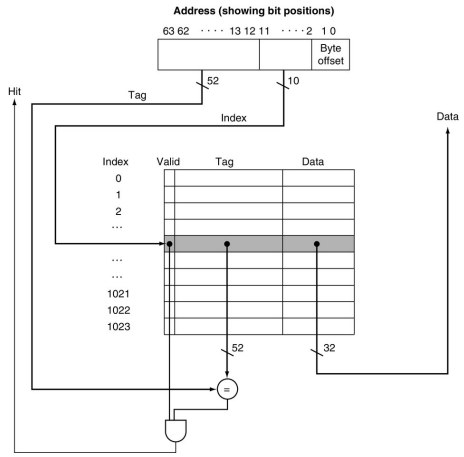
This situation is directly analogous to needing a book from the shelves and having no more space on your desk—some book already on your desk must be returned to the shelves. In a direct-mapped cache, there is only one place to put the newly requested item and hence just one choice of what to replace.

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. The figure below 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 5.3.3: For this cache, the lower portion of the address is used to select a cache entry consisting of a data word and a tag (COD Figure 5.10).

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 the figure above is called a 4 KiB cache.

Example 5.3.1: Bits in a cache.

How many total bits are required for a direct-mapped cache with 16 KiB of data and 4-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 total 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.

PARTICIPATION ACTIVITY 5.3.5: Calculating cache size.

Assume a direct-mapped cache with the following parameters:
Address size: 64 bits
Cache data size: 2 KiB
Cache block: 2 words

1) The cache contains ____ words.

Check Show answer

2) The cache contains ____ blocks.

Check Show answer

3) The size of the tag field is ____ bits.

Check Show answer

4) The total block size is ____ bits.

Check Show answer

5) The total cache size is ____ bits.

Check Show answer

Example 5.3.2: Mapping an address to a multiword cache block.

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

Answer

The block is given by

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

where the address of the block is

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

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

$$\left[\frac{\textbf{Byte address}}{\textbf{Bytes per block}}\right] \times \textbf{Bytes per block}$$

and

$$\left[\frac{\textbf{Byte address}}{\textbf{Bytes per block}}\right] \times \textbf{Bytes per block} + (\textbf{Bytes per block} - 1)$$

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

$$\left[\frac{1200}{16}\right] = 75$$

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

PARTICIPATION

ACTIVITY 5.3.6: Calculating cache size.

Assume a direct-mapped cache with 64 blocks and a block size of 8 bytes.

1) Byte address 600 maps to block address _____.

Check Show answer

2) Byte address 600 maps to block number _____.

Check Show answer

3) Byte address 560 maps to block number _____.

Check Show answer

Larger blocks exploit spatial locality to lower miss rates. As the figure below 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 5.3.4: Miss rate versus block size (COD Figure 5.11).

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 COD Section 5.9 (Using a finite-state machine to control a simple cache). 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 (COD Chapter 4 (The Processor)) 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.

PARTICIPATION ACTIVITY 5.3.7: Cache misses.

1) The miss rate may increase if the block size becomes a significant fraction of the cache size.
   - ○ True
   - ○ False

2) Which of the following items does NOT contribute to the cost of a miss penalty?
   - ○ Latency to access the first word
   - ○ Transfer time of the block
   - ○ Latency to determine the cache block

3) The processing of a cache miss creates a ____.
   - ○ pipeline stall
   - ○ interrupt

4) If an instruction access results in a miss, then the address of the instruction at _____ is fetched from the memory.
   - ○ PC
   - ○ PC - 4
   - ○ PC + 4

Handling writes

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

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

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

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

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

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

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

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

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

In the rest of this section, we describe caches from real processors, and we examine how they handle both reads and writes. In COD Section 5.8 (A common framework for memory hierarchy), 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.

PARTICIPATION ACTIVITY 5.3.8: Cache writes.

Write buffer | Write-through scheme | Write-back scheme

A value is read from the cache and modified. The modified value is written to the cache and the corresponding memory location.

A value is read from the cache and modified. The modified value is written to the cache and to a queue that stores the value while waiting to be written to the corresponding memory location.

A value is read from the cache and modified. The modified value is written to the cache. The modified value is only written from the cache to memory when the cache block is replaced.

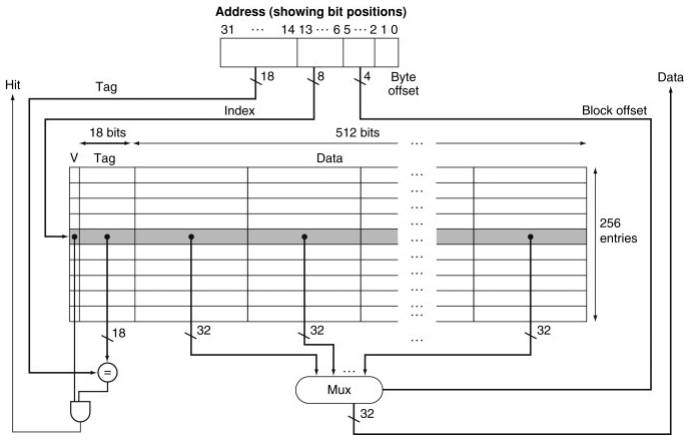
Reset

An example cache: The Intrinsity FastMATH processor

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

Figure 5.3.5: The 16 KiB caches in the Intrinsity FastMATH each contain 256 blocks with 16 words per block (COD Figure 5.12).

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



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

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

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

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

What cache miss rates are attained with a cache structure like that used by the Intrinsity FastMATH? The figure below 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.

Figure 5.3.6: Approximate instruction and data miss rates for the Intrinsity FastMATH processor for SPEC CPU2000 benchmarks (COD Figure 5.13).

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.

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

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 COD Section 5.8 (A common framework for memory hierarchy)

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 COD Section 5.4 (Measuring and improving cache performance) 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.

PARTICIPATION ACTIVITY 5.3.9: Check yourself: Cache design.

The speed of the memory system affects the designer's decision on the size of the cache block. Complete the following cache designer guidelines.

1) The shorter the memory latency, the ____ the cache block.
   - ○ smaller
   - ○ larger

2) The higher the memory bandwidth, the ____ the cache block.
   - ○ smaller
   - ○ larger

Provide feedback on this section