7.3 Measuring and Improving Cache Performance

In this section, we begin by looking at how to measure and analyze cache performance; we then explore two different techniques for improving cache performance. One focuses on reducing the miss rate by reducing the probability that two different memory blocks will contend for the same cache location. The second technique reduces the miss penalty by adding an additional level to the hierarchy. This technique, called *multilevel caching*, first appeared in high-end computers selling for over $100,000 in 1990; since then it has become common on desktop computers selling for less than $1000!

CPU time can be divided into the clock cycles that the CPU spends executing the program and the clock cycles that the CPU spends waiting for the memory system. Normally, we assume that the costs of cache accesses that are hits are part of the normal CPU execution cycles. Thus,

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

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

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

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

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

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

Writes are more complicated. For a write-through scheme, we have two sources of stalls: write misses, which usually require that we fetch the block before continuing the write (see the Elaboration on page 484 for more details on dealing with writes), and write buffer stalls, which occur when the write buffer is full when a write occurs. Thus, the cycles stalled for writes equals the sum of these two:

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

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

Write-back schemes also have potential additional stalls arising from the need to write a cache block back to memory when the block is replaced. We will discuss this more in Section 7.5.

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

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

We can also factor this as

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

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

Calculating Cache Performance

EXAMPLE

Assume an instruction cache miss rate for a program is 2% and a data cache miss rate is 4%. If a processor has a CPI of 2 without any memory stalls and the miss penalty is 100 cycles for all misses, determine how much faster a processor would run with a perfect cache that never missed. ~~Use the instruction frequencies for SPECint2000 from Chapter 3, Figure 3.26, on page 228.~~

The frequency of load and store instructions is 36%

ANSWER

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

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

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

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

The total number of memory-stall cycles is 2.00 I + 1.44 I = 3.44 I. This is more than 3 cycles of memory stall per instruction. Accordingly, the CPI with memory stalls is 2 + 3.44 = 5.44. Since there is no change in instruction count or clock rate, the ratio of the CPU execution times is

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

$$= \frac{\text{CPI}_{\text{stall}}}{\text{CPI}_{\text{perfect}}} = \frac{5.44}{2}$$

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

What happens if the processor is made faster, but the memory system is not? The amount of time spent on memory stalls will take up an increasing fraction of the execution time; ~~Amdahl's law, which we examined in Chapter 4, reminds us of this fact.~~ A few simple examples show how serious this problem can be. Suppose we speed up the computer in the previous example by reducing its CPI from 2 to 1 without changing the clock rate, which might be done with an improved pipeline. The system with cache misses would then have a CPI of 1 + 3.44 = 4.44, and the system with the perfect cache would be

Performance Difference Increases

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

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

When CPI is 2

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

to

In Case when CPI is reduced to 1

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

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

Cache Performance with Increased Clock Rate

EXAMPLE

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

In Previous case It was I X 2% X 100

In Previous case It was I X 36% X 4% X 100

ANSWER

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

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

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

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

$$\frac{\text{Performance with fast clock}}{\text{Performance with slow clock}} = \frac{\text{Execution time with slow clock}}{\text{Execution time with fast clock}}$$

$$= \frac{\text{IC} \times \text{CPI}_{\text{slow clock}} \times \text{Clock cycle}}{\text{IC} \times \text{CPI}_{\text{fast clock}} \times \frac{\text{Clock cycle}}{2}}$$

$$= \frac{5.44}{8.88 \times \frac{1}{2}} = 1.23$$

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

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

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

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

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

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

Reducing Cache Misses by More Flexible Placement of Blocks

So far, when we place a block in the cache, we have used a simple placement scheme: A block can go in exactly one place in the cache. As mentioned earlier, it

is called *direct mapped* because there is a direct mapping from any block address in memory to a single location in the upper level of the hierarchy. There is actually a whole range of schemes for placing blocks. At one extreme is direct mapped, where a block can be placed in exactly one location.

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

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 (at least two) where each block can be placed; a set-associative cache with n locations for a block is called an n-way set-associative cache. An n-way set-associative cache consists of a number of sets, each of which consists of n blocks. Each block in the memory maps to a unique *set* in the cache given by the index field, and a block can be placed in *any* element of that set. Thus, a set-associative placement combines direct-mapped placement and fully associative placement: a block is directly mapped into a set, and then all the blocks in the set are searched for a match.

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

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

(Block number) modulo (Number of cache blocks)

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

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

Since the block may be placed in any element of the set, *all the tags of all the elements of the set* must be searched. In a fully associative cache, the block can go anywhere and *all tags of all the blocks in the cache* must be searched. For example, Figure 7.13 shows where block 12 may be placed in a cache with eight blocks total, according to the block placement policy for direct-mapped, two-way set-associative, and fully associative caches.

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

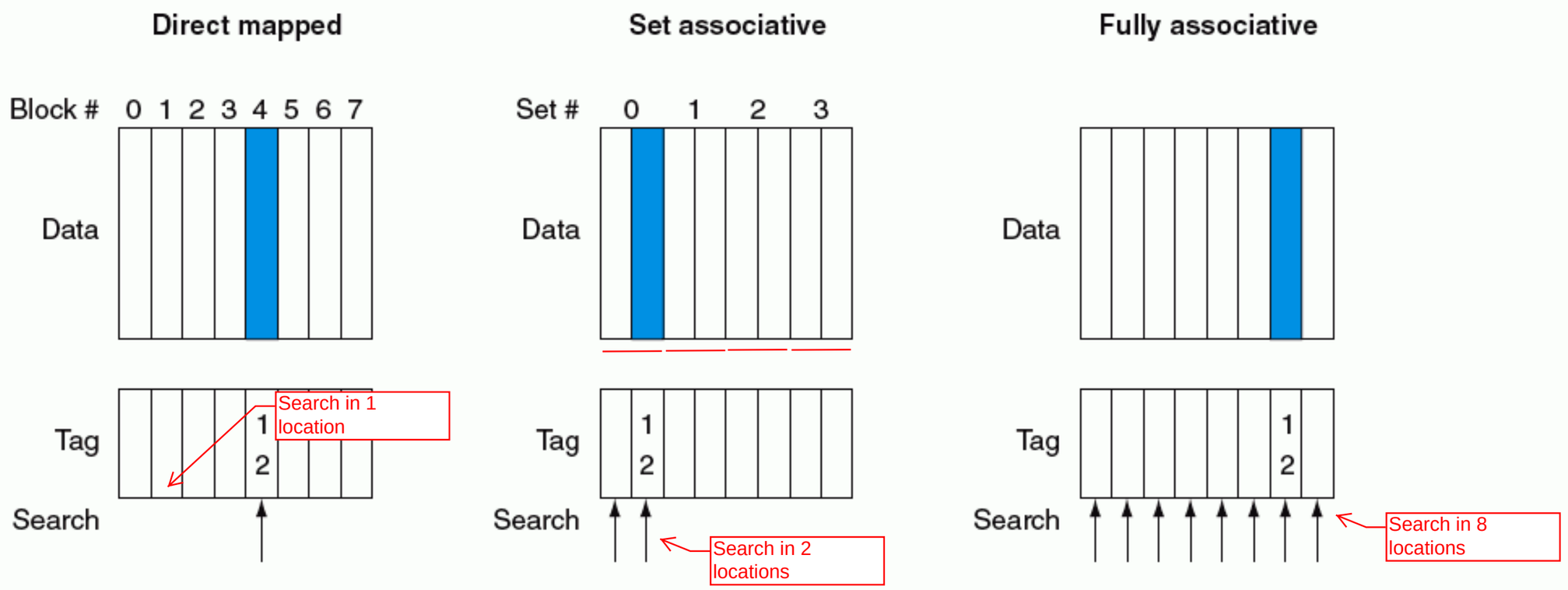

FIGURE 7.13 The location of a memory block whose address is 12 in a cache with 8 blocks varies for direct-mapped, set-associative, and fully associative placement. In direct-mapped placement, there is only one cache block where memory block 12 can be found, and that block is given by (12 modulo 8) = 4. In a two-way set-associative cache, there would be four sets, and memory block 12 must be in set (12 mod 4) = 0; the memory block could be in either element of the set. In a fully associative placement, the memory block for block address 12 can appear in any of the eight cache blocks.

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

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

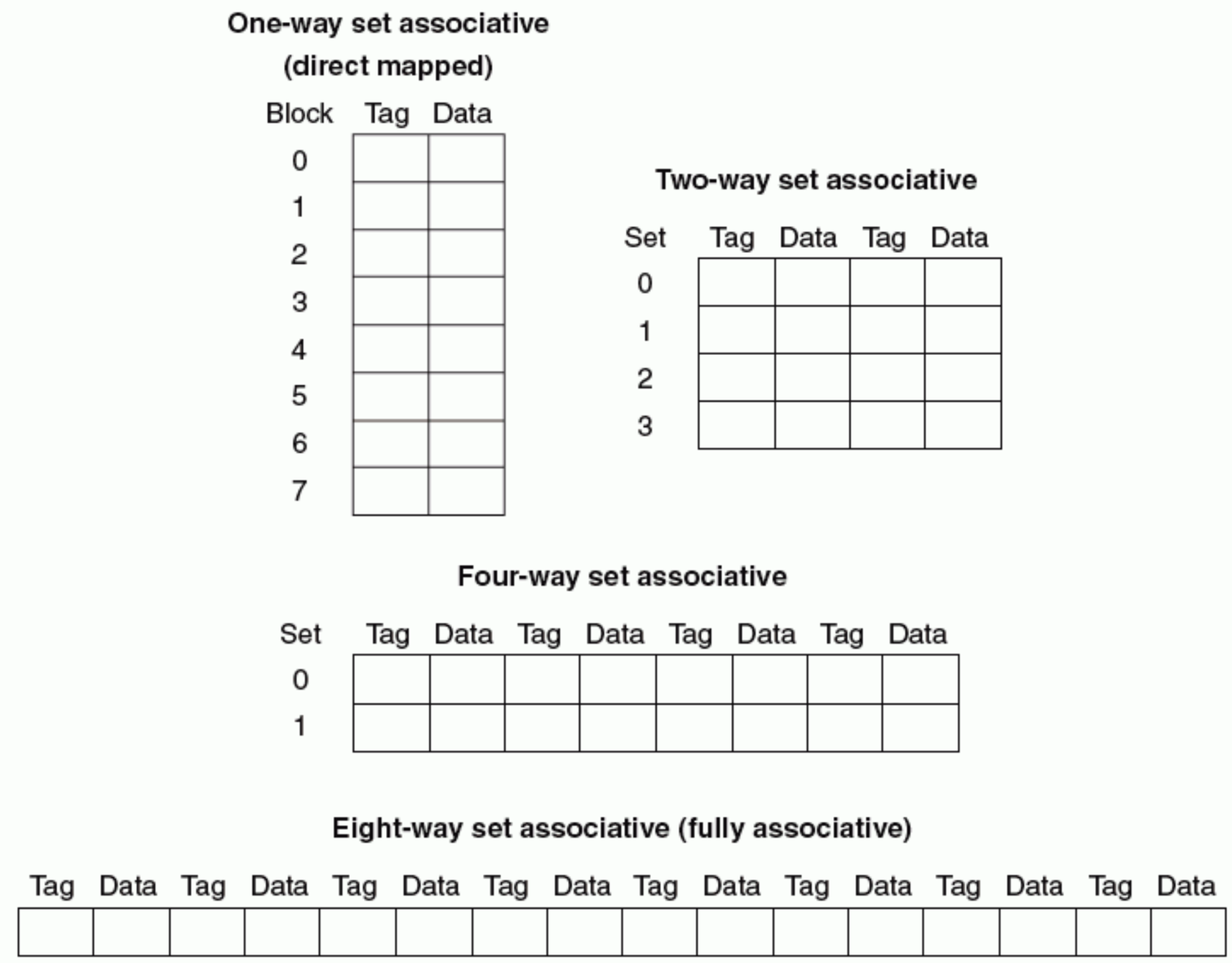

FIGURE 7.14 An eight-block cache configured as direct mapped, two-way set associative, four-way set associative, and fully associative. The total size of the cache in blocks is equal to the number of sets times the associativity. Thus, for a fixed cache size, increasing the associativity decreases the number of sets, while increasing the number of elements per set. With eight blocks, an eight-way set-associative cache is the same as a fully associative cache.

EXAMPLE

Misses and Associativity in Caches

Assume there are three small caches, each consisting of four one-word blocks. One cache is fully associative, a second is two-way set associative, and the third is direct mapped. Find the number of misses for each cache organization given the following sequence of block addresses: 0, 8, 0, 6, 8.

The Block Addresses of Memory

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 associate reference, and a plain text to show an old entry in the cache:

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

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

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

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

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

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

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

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

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

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

How much of a reduction in the miss rate is achieved by associativity? Figure 7.15 shows the improvement for the SPEC2000 benchmarks for a 64 KB data cache with a 16-word block, and associativity ranging from direct mapped to eight-way. Going from one-way to two-way associativity decreases the miss rate by about 15%, but there is little further improvement in going to higher associativity.

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

FIGURE 7.15 The data cache miss rates for an organization like the Intrinsity FastMATH processor for SPEC2000 benchmarks with associativity varying from one-way to eight-way. These results for 10 SPEC2000 programs are from Hennessy and Patterson [2003].

Locating a Block in the Cache

Now, let's consider the task of finding a block in a cache that is set associative. Just as in a direct-mapped cache, each block in a set-associative cache includes an address tag that gives the block address. The tag of every cache block within the appropriate set is checked to see if it matches the block address from the processor. Figure 7.16 shows how the address is decomposed. The index value is used to select the set containing the address of interest, and the tags of all the blocks in the set must be searched. Because speed is of the essence, all the tags in the selected set are searched in parallel. As in a fully associative cache, a sequential search would make the hit time of a set-associative cache too slow.

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

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

FIGURE 7.16 The three portions of an address in a set-associative or direct-mapped cache. The index is used to select the set, then the tag is used to choose the block by comparison with the blocks in the selected set. The block offset is the address of the desired data within the block.

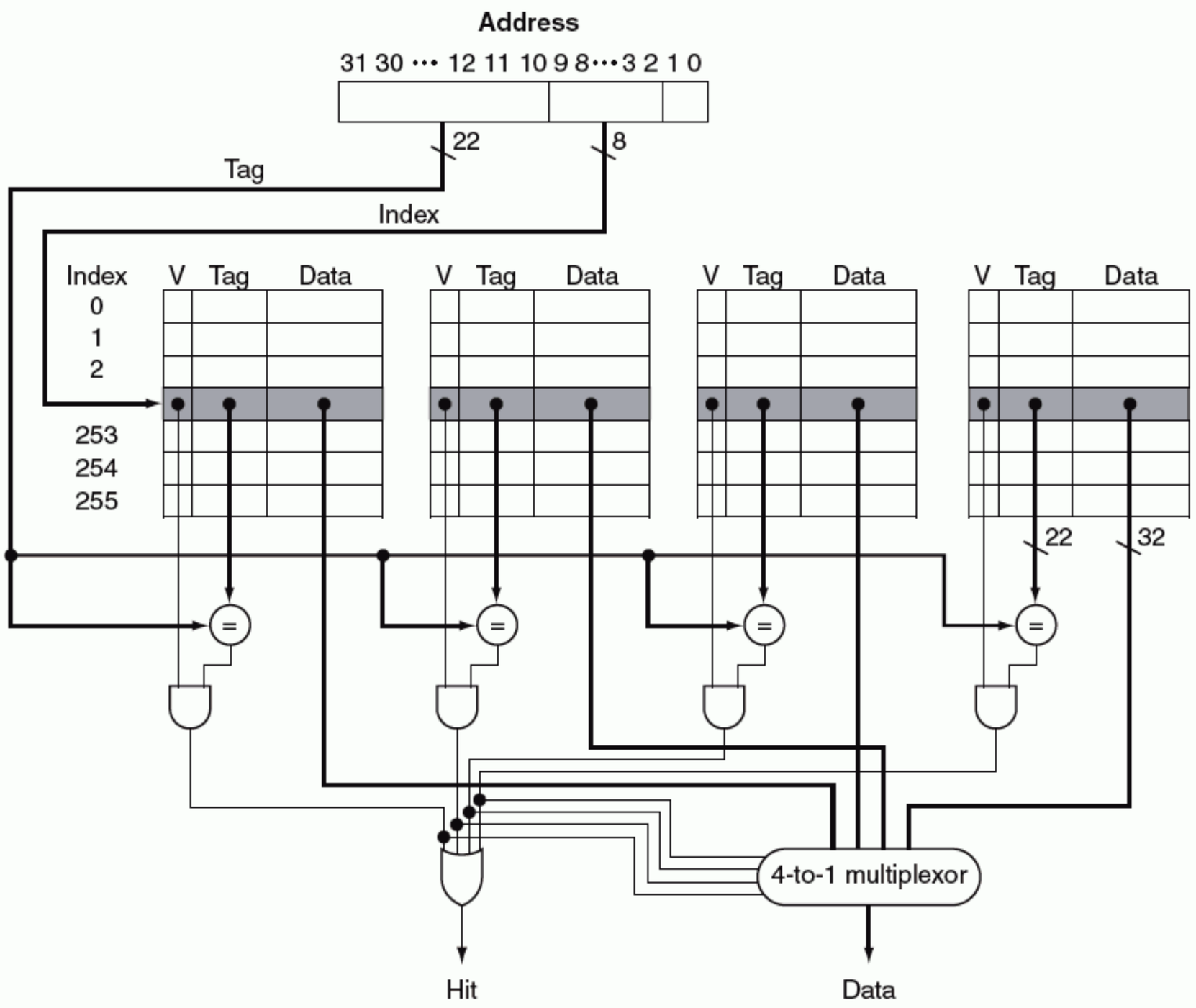

FIGURE 7.17 The implementation of a four-way set-associative cache requires four comparators and a 4-to-1 multiplexor. The comparators determine which element of the selected set (if any) matches the tag. The output of the comparators is used to select the data from one of the four blocks of the indexed set, using a multiplexor with a decoded select signal. In some implementations, the Output enable signals on the data portions of the cache RAMs can be used to select the entry in the set that drives the output. The Output enable signal comes from the comparators, causing the element that matches to drive the data outputs. This organization eliminates the need for the multiplexor.

among the four potential members of the selected set. The cache access consists of indexing the appropriate set and then searching the tags of the set. The costs of an associative cache are the extra comparators and any delay imposed by having to do the compare and select from among the elements of the set.

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

EXAMPLE

Size of Tags versus Set Associativity

Increasing associativity requires more comparators, and more tag bits per cache block. Assuming a cache of 4K blocks, a four-word block size, and a 32-bit address, find the total number of sets and the total number of tag bits for caches that are direct mapped, two-way and four-way set associative, and fully associative.

ANSWER

Since there are 16 ($=2^4$) bytes per block, a 32-bit address yields $32 - 4 = 28$ bits to be used for index and tag. The direct-mapped cache has the same number of sets as blocks, and hence 12 bits of index, since $\log_2(4K) = 12$; hence, the total number of tag bits is $(28 - 12) \times 4K = 16 \times 4K = 64$ Kbits.

Each degree of associativity decreases the number of sets by a factor of two and thus decreases the number of bits used to index the cache by one and increases the number of bits in the tag by one. Thus, for a two-way set-associative cache, there are 2K sets, and the total number of tag bits is $(28 - 11) \times 2 \times 2K = 34 \times 2K = 68$ Kbits. For a four-way set-associative cache, the total number of sets is 1K, and the total number of tag bits is $(28 - 10) \times 4 \times 1K = 72 \times 1K = 72$ Kbits.

Each location/set has 2 blocks and hence 2 tags

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

Choosing Which Block to Replace

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

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

The most commonly used scheme is **least recently used** (LRU), which we used in the previous example. In an LRU scheme. The block replaced is the one that has been unused for the longest time. LRU replacement is implemented by keeping track of when each element in a set was used relative to the other elements in the set. For a two-way set-associative cache, tracking when the two elements were used can be implemented by keeping a single bit in each set and setting the bit to indicate an element whenever that element is referenced. As associativity increases, implementing LRU gets harder; in Section 7.5, we will see an alternative scheme for replacement.

Reducing the Miss Penalty Using Multilevel Caches

All modern computers make use of caches. In most cases, these caches are implemented on the same die as the microprocessor that forms the processor. To further close the gap between the fast clock rates of modern processors and the relatively long time required to access DRAMs, many microprocessors support an additional level of caching. This second-level cache, which can be on the same chip or off-chip in a separate set of SRAMs, is accessed whenever a miss occurs in the primary cache. If the second-level cache contains the desired data, the miss penalty for the first-level cache will be the access time of the second-level cache, which will be much less than the access time of main memory. If neither the primary nor secondary cache contains the data, a main memory access is required, and a larger miss penalty is incurred.

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

Performance of Multilevel Caches

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

ANSWER

The miss penalty to main memory is

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

The effective CPI with one level of caching is given by

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

For the processor with one level of caching,

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

With two levels of cache, 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

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

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

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

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

Thus, the processor with the secondary cache is faster by

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

Alternatively, we could have computed the stall cycles by summing the stall cycles of those references that hit in the secondary cache $((2\% - 0.5\%) \times 25 = 0.4)$ and those references that go to main memory, which must include the cost to access the secondary cache as well as the main memory access time $(0.5\% \times (25 + 500) = 2.6)$. The sum, $1.0 + 0.4 + 2.6$, is again 4.0.

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

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

The effect of these changes on the two caches can be seen by comparing each cache to the optimal design for a single level of cache. In comparison to a single-level cache, the primary cache of a **multilevel cache** is often smaller. Furthermore, the primary cache often uses a smaller block size, to go with the smaller cache size and reduced miss penalty. In comparison, the secondary cache will often be larger than in a single-level cache, since the access time of the secondary cache is less critical. With a larger total size, the secondary cache often will use a larger block size than appropriate with a single-level cache

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

Understanding Program Performance

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

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