5.4 Measuring and improving cache performance

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

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

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

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

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

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

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

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

Writes are more complicated. For a write-through scheme, we have two sources of stalls: write misses, which usually require that we fetch the block before continuing the write (see the *Elaboration* in COD Section 5.3 (The basics of caches) for more details on dealing with writes), and write buffer stalls, which occur when the write buffer is full when a write happens. Thus, the cycles stalled for writes equal the sum of these two:

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

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

Write-back schemes also have potential additional stalls arising from the need to write a cache block back to memory when the block is replaced. We will discuss this more in COD Section 5.8 (A common framework for memory hierarchy).

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.

Example 5.4.1: Calculating cache performance.

Assume the miss rate of an instruction cache is 2% and the miss rate of the data cache is 4%. If a processor has a CPI of 2 without any memory stalls, and the miss penalty is 100 cycles for all misses, determine how much faster a processor would run with a perfect cache that never missed. Assume the frequency of all loads and stores is 36%.

Answer

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

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

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

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

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

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

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

PARTICIPATION ACTIVITY 5.4.1: Calculating cache performance.

Consider the example above.

1) The instruction cache miss rate is _____.
   - ○ 2%
   - ○ 4%
   - ○ 36%

2) The number of memory-stall cycles for data misses in terms of the instruction count (I) is _____.
   - ○ I × 4% × 100
   - ○ I × 36% × 4%
   - ○ I × 36% × 4% × 100

3) The total CPI is _____.
   - ○ 1.44
   - ○ 3.44
   - ○ 5.44

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 COD Chapter 1 (Computer Abstractions and Technology), reminds us of this fact. A few simple examples show how serious this problem can be. Suppose we speed-up the computer in the previous example by reducing its CPI from 2 to 1 without changing the clock rate, which might be done with an improved pipeline. The system with cache misses would then have a CPI of 1 + 3.44 = 4.44, and the system with the perfect cache would be

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

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

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

to

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

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

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

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

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

Example 5.4.2: Calculating average memory access time.

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

Answer

The average memory access time per instruction is

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

or 2 ns.

PARTICIPATION ACTIVITY 5.4.2: Affecting cache performance.

1) If the clock rate is increased without changing the memory system, the fraction of execution time due to cache misses _____ relative to total execution time.
   - ○ increases
   - ○ decreases

2) AMAT considers the average time to

access data for ____.

- misses
- both hits and misses

The next subsection discusses alternative cache organizations that decrease miss rate but may sometimes increase hit time; additional examples appear in COD Section 5.16 (Fallacies and pitfalls).

Reducing cache misses by more flexible placement of blocks

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

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

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

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

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

Figure 5.4.1: The location of a memory block whose address is 12 in a cache with eight blocks varies for direct-mapped, set-associative, and fully associative placement (COD Figure 5.14).

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.

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

(Block number) modulo (Number of *blocks* in the cache)

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.

PARTICIPATION ACTIVITY 5.4.3: Mapping a memory block to different types of caches.

Assume a cache with 8 one-word blocks. Determine the cache position given the cache configuration and memory block.

1) Cache configuration: Direct-mapped
   Memory block: 15
   - Block #7
   - Block #15
   - Any of the eight cache blocks

2) Cache configuration: Two-way set-associative
   Memory block: 15
   - Set #7
   - Set #3
   - Any of the eight cache blocks

3) Cache configuration: Fully associative

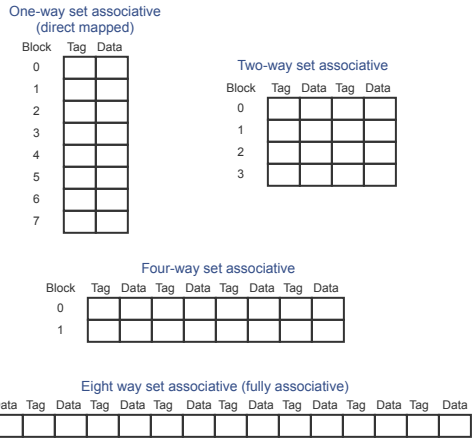
Memory block: 15

- ○ Block #7
- ○ Set #3
- ○ Any of the eight cache blocks

We can also think of all block placement strategies as a variation on set associativity. The figure below shows the possible associativity structures for an eight-block cache. A direct-mapped cache is just a one-way set-associative cache: each cache entry holds one block and each set has one element. A fully associative cache with *m* entries is simply an *m*-way set-associative cache; it has one set with *m* blocks, and an entry can reside in any block within that set.

PARTICIPATION ACTIVITY 5.4.4: An eight-block cache configured as direct mapped, two-way set associative, four-way set associative, and fully associative (COD Figure 5.15).

Start ☐ 2x speed



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

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

Example 5.4.3: 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, and 8.

Answer

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

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

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

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

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

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

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

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

detail shortly.) Using this replacement rule, the contents of the set-associative cache after each reference look like this:

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

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

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

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

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

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

Figure 5.4.2: The data cache miss rates for an organization like the Intrinsity FastMATH processor for SPEC CPU2000 benchmarks with associativity varying from one-way to eight-way (COD Figure 5.16).

These results for 10 SPEC CPU2000 programs are from Hennessy and Patterson (2003).

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

PARTICIPATION ACTIVITY 5.4.5: Misses in a direct-mapped cache.

Assume a direct-mapped cache with 8-one word blocks. Given the following sequence of block addresses, indicate if each request results in a cache hit or miss: 1, 9, 6, 5, 1, 6.

Cache block mapping:

- Block address 1 maps to cache block 1
- Block address 5 maps to cache block 5
- Block address 6 maps to cache block 6
- Block address 9 maps to cache block 1

1) Memory block 1
   - ○ Hit
   - ○ Miss

2) Memory block 9
   - ○ Hit
   - ○ Miss

3) Memory block 6
   - ○ Hit
   - ○ Miss

4) Memory block 5
   - ○ Hit
   - ○ Miss

5) Memory block 1
   - ○ Hit
   - ○ Miss

6) Memory block 6
   - ○ Hit
   - ○ Miss

PARTICIPATION

ACTIVITY 5.4.6: Misses in a two-way set associative cache.

Assume a two-way set-associative cache with 8-one word blocks. Given the following sequence of block addresses, indicate if each request results in a cache hit or miss: 1, 9, 6, 5, 1, 6.

Cache set mapping:

- Block address 1 maps to cache set 1
- Block address 5 maps to cache set 1
- Block address 6 maps to cache set 2
- Block address 9 maps to cache set 1

1) Memory block 1
   - ○ Hit
   - ○ Miss
2) Memory block 9
   - ○ Hit
   - ○ Miss
3) Memory block 6
   - ○ Hit
   - ○ Miss
4) Memory block 5
   - ○ Hit
   - ○ Miss
5) Memory block 1
   - ○ Hit
   - ○ Miss
6) Memory block 6
   - ○ Hit
   - ○ Miss

PARTICIPATION ACTIVITY 5.4.7: Misses in a fully associative cache.

Assume a fully associative cache with 8-one word blocks. Given the following sequence of block addresses, indicate if each request results in a cache hit or miss: 1, 9, 6, 5, 1, 6.

1) Memory block 1
   - ○ Hit
   - ○ Miss
2) Memory block 9
   - ○ Hit
   - ○ Miss
3) Memory block 6
   - ○ Hit
   - ○ Miss
4) Memory block 5
   - ○ Hit
   - ○ Miss
5) Memory block 1
   - ○ Hit
   - ○ Miss
6) Memory block 6
   - ○ Hit
   - ○ Miss

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. The figure below decomposes the address. The index value is used to select the set containing the address of interest, and the tags of all the blocks in the set must be searched. Because speed is of the essence, all the tags in the selected set are searched in parallel. As in a fully associative cache, a sequential search would make the hit time of a set-associative cache too slow.

Figure 5.4.3: The three portions of an address in a set-associative or direct-mapped cache (COD Figure 5.17).

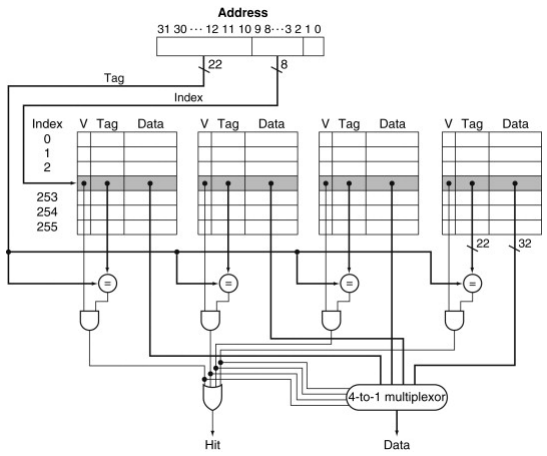
The index is used to select the set, then the tag is used to choose the block by comparison with the blocks in the selected set. The block offset is the address of the desired data within the block.

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

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

Figure 5.4.4: The implementation of a four-way set-associative cache requires four comparators and a 4-to-1 multiplexor (COD Figure 5.18).

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



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

Elaboration

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

Choosing which block to replace

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

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

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

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

Example 5.4.4: Size of tags versus set associativity.

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

Answer

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

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

For a fully associative cache, there is only one set with 4096 blocks, and the tag is 60 bits, leading to 60 × 4096 × 1 = 246 K tag bits.

PARTICIPATION ACTIVITY 5.4.8: Finding and replacing a block in a cache.

1) The _____ of every cache block within the appropriate set of a set-associative cache is checked for a match against the memory block address.
   - ○ index
   - ○ tag
   - ○ block offset

2) A four-way set-associative cache with 32-one word blocks requires _____ comparators to compare the tags of each element within the set.
   - ○ 4
   - ○ 8
   - ○ 32

3) A direct mapped cache with 32-one word blocks requires _____ comparator(s) to compare the tags of of an element with the memory block address.
   - ○ 1
   - ○ 32

4) Which block in the cache is replaced by memory block 29?

   Cache configuration: 4-way set-associative cache with 8-one word blocks
   Replacement scheme: LRU
   Sequence of previously accessed block addresses: 5, 13, 21, 13, 5
   (Note: All memory block addresses map to cache set 1)
   - ○ Mem[5]
   - ○ Mem[13]
   - ○ Mem[21]
   - ○ None. An element in set 1 is unused, so Mem[29] is placed in the fourth element of set 1.

Reducing the miss penalty using multilevel caches

All modern computers make use of caches. To close the gap further between the fast clock rates of modern processors and the increasingly long time required to access DRAMs, most microprocessors support an additional level of caching. This second-level cache is normally on the same chip and is accessed whenever a miss occurs in the primary cache. If the second-level cache contains the desired data, the miss penalty for the first-level cache will be essentially the access time of the second-level cache, which will be much less than the access time of main memory. If neither the primary nor the secondary cache contains the data, a main memory access is required, and a larger miss penalty is incurred.

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

Example 5.4.5: Performance of multilevel caches.

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

Answer

The miss penalty to main memory is

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

The effective CPI with one level of caching is given by

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

For the processor with one level of caching.

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

With two levels of caching, a miss in the primary (or first-level) cache can be satisfied either by the secondary cache or by main memory. The miss penalty for an access to the second-level cache is

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

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

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

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

Thus, the processor with the secondary cache is faster by

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

Alternatively, we could have computed the stall cycles by summing the stall cycles of those references that hit in the secondary cache ((2% - 0.5%) × 20 = 0.3). Those references that go to main memory, which must include the cost to access the secondary cache as well as the main memory

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

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

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

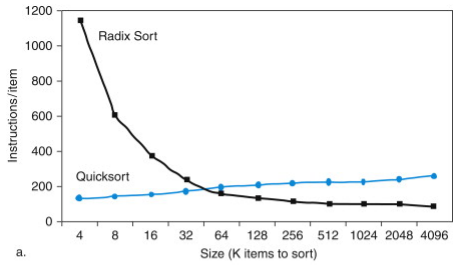
Understanding program performance

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

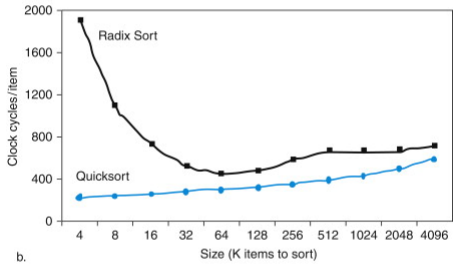
Alas, standard algorithmic analysis often ignores the impact of the memory hierarchy. As faster clock rates and **Moore's Law** allow architects to squeeze all of the performance out of a stream of instructions, using the memory hierarchy well is vital to high performance. As we said in the introduction,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

Figure 5.4.5: Comparing Quicksort and Radix Sort (a) instructions executed per item sorted, (b) time per item sorted, and (c) cache misses per item sorted (COD Figure 5.19).

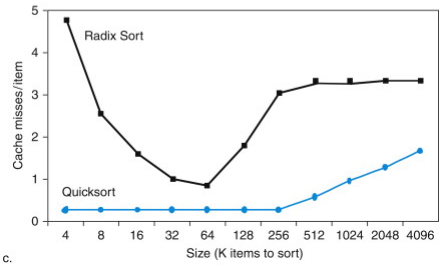
These data are from a paper by LaMarca and Ladner [1996]. Due to such results, new versions of Radix Sort have been invented that take memory hierarchy into account, to regain its algorithmic advantages (see COD Section 5.16 (Fallacies and pitfalls)). The basic idea of cache optimizations is to use all the data in a block repeatedly before it is replaced on a miss.



a.



b.



c.

PARTICIPATION ACTIVITY 5.4.9: Cache miss penalty.

1) The second-level cache in a multi-level cache is typically used to reduce the multi-level cache's _____.
   - O hit time
   - O miss penalty

2) Refer the the above figure (COD Figure 5.19 (Comparing Quicksort and Radix Sort …)). As the number of items to sort increases, Radix Sort requires _____ clock cycles compared to Quicksort.
   - O fewer
   - O the same
   - O more

Software optimization via blocking

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

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

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

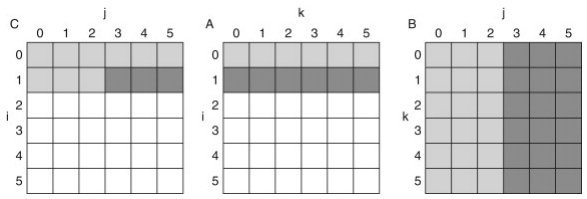
For example, the inner loops of DGEMM (lines 4 through 9 of COD Figure 3.22 (Unoptimized C version of a double precision matrix multiply …)) are

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

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

   C[i + j * n] = cij;                     /* C[i][j] = cij */
}
```

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

Figure 5.4.6: A snapshot of three arrays C, A, and B when N = 6 and i = 1 (COD Figure 5.20).

The age of accesses to the array elements is indicated by shade: white means not yet touched, light means older accesses, and dark means newer accesses. Compared to COD Figure 5.22 (The age of accesses to the arrays C, A, and B when BLOCKSIZE = 3), elements of A and B are read repeatedly to calculate new elements of C. The variables i, j, and k are shown along the rows or columns used to access the arrays.



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

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

To ensure that the elements being accessed can fit in the cache, the original code is changed to compute on a submatrix. Hence, we essentially invoke the version of DGEMM from COD Figure 4.78 (Optimized C version of DGEMM using C intrinsics …) repeatedly on matrices of size BLOCKSIZE by BLOCKSIZE. BLOCKSIZE is called the *blocking factor*.

The figure below shows the blocked version of DGEMM. The function do_block is DGEMM from COD Figure 3.22 (Unoptimized C version of a double precision matrix multiply …) with three new parameters si, sj, and sk to specify the starting position of each submatrix of of A, B, and C. The two inner loops of the do_block now compute in steps of size BLOCKSIZE rather than the full length of B and C. The gcc optimizer removes any function call overhead by "inlining" the function; that is, it inserts the code directly to avoid the conventional parameter passing and return address bookkeeping instructions.

Figure 5.4.7: Cache blocked version of DGEMM (COD Figure 5.21).

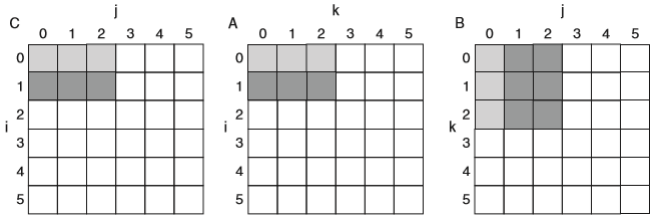
Assume C is initialized to zero. The do_block function is basically DGEMM from COD Chapter 3 (Arithmetic for Computers) with new parameters to specify the starting positions of the submatrices of BLOCKSIZE. The gcc optimizer can remove the function overhead instructions by inlining the do_block function.

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

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

Figure 5.4.8: The age of accesses to the arrays C, A, and B when BLOCKSIZE = 3 (COD Figure 5.22).

Note that, in contrast to COD Figure 5.20 (A snapshot of the three arrays C, A, and B when N = 6 and i = 1), fewer elements are accessed.

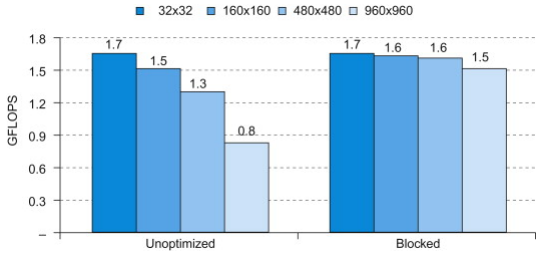


Although we have aimed at reducing cache misses, blocking can also be used to help register allocation. By taking a small blocking size, such that the block can be held in registers, we can minimize the number of loads and stores in the program, which again improves performance.

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

Figure 5.4.9: Performance of unoptimized DGEMM versus cache block DGEMM (COD Figure 5.23).

Performance of unoptimized DGEMM (COD Figure 3.22 (Unoptimized C version of a double precision matrix multiply ...)) versus cache blocked DGEMM (COD Figure 5.21 (Cache Blocked Version of DGEMM)) as the matrix dimension varies from 32 × 32 (where all three matrices fit in the cache) to 960 × 960.



PARTICIPATION ACTIVITY 5.4.10: Cache blocked DGEMM example.

1) The cache blocked version of DGEMM improves performance by operating on submatrices, rather than operating on entire rows or columns of an array.
   - ○ True
   - ○ False

2) Calls to `do_block()` degrade performance due to parameter passing and return address bookkeeping instructions associated with function calls.
   - ○ True
   - ○ False

3) The performance of the cached blocked version of DGEMM is halved for the largest matrix.
   - ○ True
   - ○ False

Elaboration

Multilevel caches create many complications. First, there are now several different types of misses and corresponding miss rates. In the previous example (Performance of multilevel caches), we saw the primary cache miss rate and the global miss rate—the fraction of references that missed in all cache levels. There is also a miss rate for the secondary cache, which is the ratio of all misses in the secondary cache divided by the number of accesses to it. This miss rate is called the local miss rate of the secondary cache. Because the primary cache filters accesses, especially those with good spatial and temporal locality, the local miss rate of the secondary cache is much higher than the global miss rate. For this example, we can compute the local miss rate of the secondary cache as 0.5%/2% = 25%! Luckily, the global miss rate dictates how often we must access the main memory.

Global miss rate: The fraction of references that miss in all levels of a mutlilevel cache.

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

Elaboration

With out-of-order processors (see COD Chapter 4 (The Processor)), performance is more complex, si execute instructions during the miss penalty. Instead of instruction miss rates and data miss rates, w misses per instruction, and this formula:

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

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

Elaboration

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

PARTICIPATION ACTIVITY 5.4.11: Check yourself: Multi-level caches.

1) First-level caches are more concerned about _____.
   - ○ hit time
   - ○ miss rate

2) Second-level caches are more concerned about _____.
   - ○ hit time
   - ○ miss rate

Summary

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

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

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

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

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

Provide feedback on this section