5 Memory-Hierarchy Design

Ideally one would desire an indefinitely large memory capacity such that any particular . . . word would be immediately available. . . . We are . . . forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible.

A. W. Burks, H. H. Goldstine, and J. von Neumann
Preliminary Discussion of the Logical Design of an Electronic Computing Instrument (1946)

5.1 Introduction

Computer pioneers correctly predicted that programmers would want unlimited amounts of fast memory. An economical solution to that desire is a *memory hierarchy*, which takes advantage of locality and cost/performance of memory technologies. The *principle of locality*, presented in the first chapter, says that most programs do not access all code or data uniformly (see section 1.6, page 38). This principle, plus the guideline that smaller hardware is faster, led to the hierarchy based on memories of different speeds and sizes. Since fast memory is expensive, a memory hierarchy is organized into several levels—each smaller, faster, and more expensive per byte than the next level. The goal is to provide a memory system with cost almost as low as the cheapest level of memory and speed almost as fast as the fastest level. The levels of the hierarchy usually subset one another; all data in one level is also found in the level below, and all data in that lower level is found in the one below it, and so on until we reach the bottom of the hierarchy. Note that each level maps addresses from a larger memory to a smaller but faster memory higher in the hierarchy. As part of address mapping,

the memory hierarchy is given the responsibility of address checking; hence protection schemes for scrutinizing addresses are also part of the memory hierarchy.

The importance of the memory hierarchy has increased with advances in performance of processors. For example, in 1980 microprocessors were often designed without caches, while in 1995 they often come with two levels of caches. As noted in Chapter 1, microprocessor performance improved 55% per year since 1987, and 35% per year until 1986. Figure 5.1 plots CPU performance projections against the historical performance improvement in main memory access time. Clearly there is a processor-memory performance gap that computer architects must try to close.

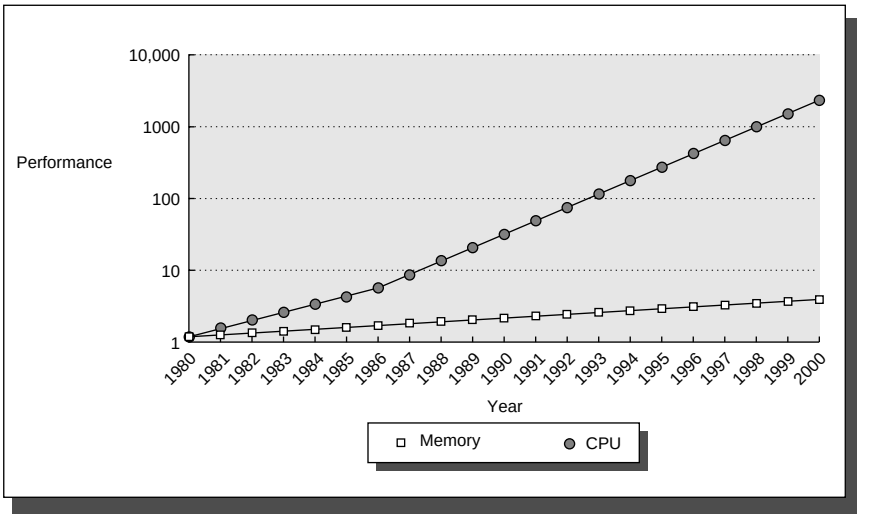

FIGURE 5.1 Starting with 1980 performance as a baseline, the performance of memory and CPUs are plotted over time. The memory baseline is 64-KB DRAM in 1980, with three years to the next generation and a 7% per year performance improvement in latency (see Figure 5.30 on page 429). The CPU line assumes a 1.35 improvement per year until 1986, and a 1.55 improvement thereafter. Note that the vertical axis must be on a logarithmic scale to record the size of the CPU-DRAM performance gap.

In addition to giving us the trends that highlight the importance of the memory hierarchy, Chapter 1 gives us a formula to evaluate the effectiveness of the memory hierarchy:

$$\text{Memory stall cycles} = \text{Instruction count} \times \text{Memory references per instruction} \times \text{Miss rate} \times \text{Miss penalty}$$

where *Miss rate* is the fraction of accesses that are not in the cache and *Miss penalty* is the additional clock cycles to service the miss. Recall that a *block* is the minimum unit of information that can be present in the cache (*hit* in the cache) or not (*miss* in the cache).

This chapter uses a related formula to evaluate many examples of using the principle of locality to improve performance while keeping the memory system affordable. This common principle allows us to pose four questions about *any* level of the hierarchy:

Q1: Where can a block be placed in the upper level? (*Block placement*)

Q2: How is a block found if it is in the upper level? (*Block identification*)

Q3: Which block should be replaced on a miss? (*Block replacement*)

Q4: What happens on a write? (*Write strategy*)

The answers to these questions help us understand the different trade-offs of memories at different levels of a hierarchy; hence we ask these four questions on every example.

To put these abstract ideas into practice, throughout the chapter we show examples from the four levels of the memory hierarchy in a computer using the Alpha AXP 21064 microprocessor. Toward the end of the chapter we evaluate the impact of these levels on performance using the SPEC92 benchmark programs.

5.2 The ABCs of Caches

Cache: a safe place for hiding or storing things.

Webster's New World Dictionary of the American Language, Second College Edition (1976)

Cache is the name generally given to the first level of the memory hierarchy encountered once the address leaves the CPU. Since the principle of locality applies at many levels, and taking advantage of locality to improve performance is so popular, the term *cache* is now applied whenever buffering is employed to reuse commonly occurring items; examples include *file caches*, *name caches*, and so on. We start our description of caches by answering the four common questions for the first level of the memory hierarchy; you'll see similar questions and answers later.

Q1: Where can a block be placed in a cache?

Figure 5.2 shows that the restrictions on where a block is placed create three categories of cache organization:

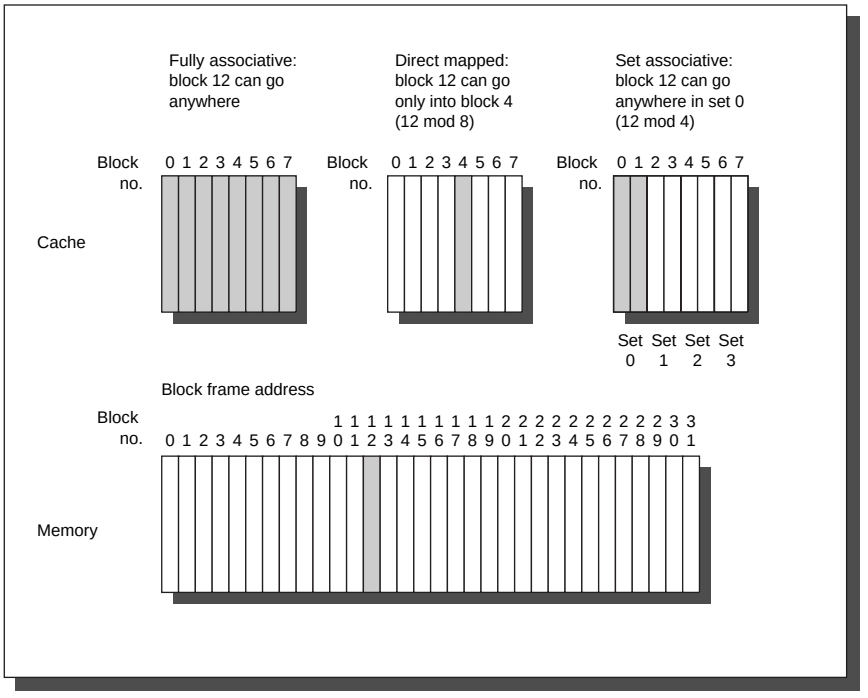

FIGURE 5.2 This example cache has eight block frames and memory has 32 blocks. Real caches contain hundreds of block frames and real memories contain millions of blocks. The set-associative organization has four sets with two blocks per set, called *two-way set associative*. Assume that there is nothing in the cache and that the block address in question identifies lower-level block 12. The three options for caches are shown left to right. In fully associative, block 12 from the lower level can go into any of the eight block frames of the cache. With direct mapped, block 12 can only be placed into block frame 4 (12 modulo 8). Set associative, which has some of both features, allows the block to be placed anywhere in set 0 (12 modulo 4). With two blocks per set, this means block 12 can be placed either in block 0 or block 1 of the cache.

- If each block has only one place it can appear in the cache, the cache is said to be *direct mapped*. The mapping is usually

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

- If a block can be placed anywhere in the cache, the cache is said to be *fully associative*.
- If a block can be placed in a restricted set of places in the cache, the cache is said to be *set associative*. A *set* is a group of blocks in the cache. A block is first mapped onto a set, and then the block can be placed anywhere within that set. The set is usually chosen by *bit selection*; that is,

  (Block address) MOD *(Number of sets in cache)*

  If there are *n* blocks in a set, the cache placement is called *n-way set associative*.

The range of caches from direct mapped to fully associative is really a continuum of levels of set associativity: Direct mapped is simply one-way set associative and a fully associative cache with *m* blocks could be called *m*-way set associative; equivalently, direct mapped can be thought of as having *m* sets and fully associative as having one set. The vast majority of processor caches today are direct mapped, two-way set associative, or four-way set associative, for reasons we shall see shortly.

Q2: How is a block found if it is in the cache?

Caches have an address tag on each block frame that gives the block address. The tag of every cache block that might contain the desired information is checked to see if it matches the block address from the CPU. As a rule, all possible tags are searched in parallel because speed is critical.

There must be a way to know that a cache block does not have valid information. The most common procedure is to add a *valid bit* to the tag to say whether or not this entry contains a valid address. If the bit is not set, there cannot be a match on this address.

Before proceeding to the next question, let's explore the relationship of a CPU address to the cache. Figure 5.3 shows how an address is divided. The first division is between the *block address* and the *block offset*. The block frame address can be further divided into the *tag* field and the *index* field. The block offset field selects the desired data from the block, the index field selects the set, and the tag field is compared against it for a hit. While the comparison could be made on more of the address than the tag, there is no need because of the following:

- Checking the index would be redundant, since it was used to select the set to be checked; an address stored in set 0, for example, must have 0 in the index field or it couldn't be stored in set 0.
- The offset is unnecessary in the comparison since the entire block is present or not, and hence all block offsets must match.

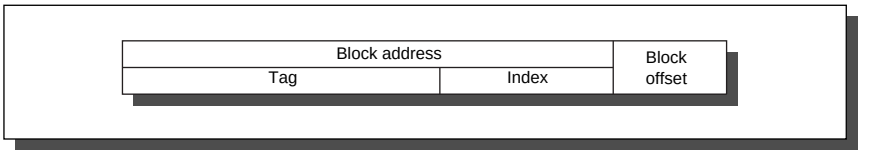

FIGURE 5.3 The three portions of an address in a set-associative or direct-mapped cache. The tag is used to check all the blocks in the set and the index is used to select the set. The block offset is the address of the desired data within the block.

If the total cache size is kept the same, increasing associativity increases the number of blocks per set, thereby decreasing the size of the index and increasing the size of the tag. That is, the tag-index boundary in Figure 5.3 moves to the right with increasing associativity, with the end case of fully associative caches having no index field.

Q3: Which block should be replaced on a cache miss?

When a miss occurs, the cache controller must select a block to be replaced with the desired data. A benefit of direct-mapped placement is that hardware decisions are simplified—in fact, so simple that there is no choice: Only one block frame is checked for a hit, and only that block can be replaced. With fully associative or set-associative placement, there are many blocks to choose from on a miss. There are two primary strategies employed for selecting which block to replace:

- *Random*—To spread allocation uniformly, candidate blocks are randomly selected. Some systems generate pseudorandom block numbers to get reproducible behavior, which is particularly useful when debugging hardware.
- *Least-recently used* (LRU)—To reduce the chance of throwing out information that will be needed soon, accesses to blocks are recorded. The block replaced is the one that has been unused for the longest time. LRU makes use of a corollary of locality: If recently used blocks are likely to be used again, then the best candidate for disposal is the least-recently used block.

A virtue of random replacement is that it is simple to build in hardware. As the number of blocks to keep track of increases, LRU becomes increasingly expensive and is frequently only approximated. Figure 5.4 shows the difference in miss rates between LRU and random replacement.

Q4: What happens on a write?

Reads dominate processor cache accesses. All instruction accesses are reads, and most instructions don't write to memory. Figure 2.26 on page 105 in Chapter 2 suggests a mix of 9% stores and 26% loads for DLX programs, making writes 9%/(100% + 26% + 9%) or about 7% of the overall memory traffic and

	Associativity					
	Two-way		Four-way		Eight-way	
Size	LRU	Random	LRU	Random	LRU	Random
16 KB	5.18%	5.69%	4.67%	5.29%	4.39%	4.96%
64 KB	1.88%	2.01%	1.54%	1.66%	1.39%	1.53%
256 KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

FIGURE 5.4 Miss rates comparing least-recently used versus random replacement for several sizes and associativities. These data were collected for a block size of 16 bytes using one of the VAX traces containing user and operating system code. There is little difference between LRU and random for larger-size caches in this trace. Although not included in the table, a first-in, first-out order replacement policy is worse than random or LRU.

9%/(26% + 9%) or about 25% of the data cache traffic. Making the common case fast means optimizing caches for reads, especially since processors traditionally wait for reads to complete but need not wait for writes. Amdahl's Law (section 1.6, page 29) reminds us, however, that high-performance designs cannot neglect the speed of writes.

Fortunately, the common case is also the easy case to make fast. The block can be read from cache at the same time that the tag is read and compared, so the block read begins as soon as the block address is available. If the read is a hit, the requested part of the block is passed on to the CPU immediately. If it is a miss, there is no benefit—but also no harm; just ignore the value read.

Such is not the case for writes. Modifying a block cannot begin until the tag is checked to see if the address is a hit. Because tag checking cannot occur in parallel, writes normally take longer than reads. Another complexity is that the processor also specifies the size of the write, usually between 1 and 8 bytes; only that portion of a block can be changed. In contrast, reads can access more bytes than necessary without fear.

The write policies often distinguish cache designs. There are two basic options when writing to the cache:

- *Write through* (or *store through*)—The information is written to both the block in the cache *and* to the block in the lower-level memory.
- *Write back* (also called *copy back* or *store in*)—The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.

To reduce the frequency of writing back blocks on replacement, a feature called the *dirty bit* is commonly used. This status bit indicates whether the block is *dirty* (modified while in the cache) or *clean* (not modified). If it is clean, the

block is not written on a miss, since the lower level has identical information to the cache.

Both write back and write through have their advantages. With write back, writes occur at the speed of the cache memory, and multiple writes within a block require only one write to the lower-level memory. Since some writes don't go to memory, write back uses less memory bandwidth, making write back attractive in multiprocessors. With write through, read misses never result in writes to the lower level, and write through is easier to implement than write back. Write through also has the advantage that the next lower level has the most current copy of the data. This is important for I/O and for multiprocessors, which we examine in Chapters 6 and 8. As we shall see, I/O and multiprocessors are fickle: they want write back for processor caches to reduce the memory traffic and write through to keep the cache consistent with lower levels of the memory hierarchy.

When the CPU must wait for writes to complete during write through, the CPU is said to *write stall*. A common optimization to reduce write stalls is a *write buffer*, which allows the processor to continue as soon as the data is written to the buffer, thereby overlapping processor execution with memory updating. As we shall see shortly, write stalls can occur even with write buffers.

Since the data are not needed on a write, there are two common options on a write miss:

- *Write allocate* (also called *fetch on write*)—The block is loaded on a write miss, followed by the write-hit actions above. This is similar to a read miss.
- *No-write allocate* (also called *write around*)—The block is modified in the lower level and not loaded into the cache.

Although either write-miss policy could be used with write through or write back, write-back caches generally use write allocate (hoping that subsequent writes to that block will be captured by the cache) and write-through caches often use no-write allocate (since subsequent writes to that block will still have to go to memory).

An Example: The Alpha AXP 21064 Data Cache and Instruction Cache

To give substance to these ideas, Figure 5.5 shows the organization of the data cache in the Alpha AXP 21064 microprocessor that is found in the DEC 3000 Model 800 workstation. The cache contains 8192 bytes of data in 32-byte blocks with direct-mapped placement, write through with a four-block write buffer, and no-write allocate on a write miss.

Let's trace a cache hit through the steps of a hit as labeled in Figure 5.5. (The four steps are shown as circled numbers.) As we shall see later (Figure 5.41), the 21064 microprocessor presents a 34-bit physical address to the cache for tag comparison. The address coming into the cache is divided into two fields: the 29-bit block address and 5-bit block offset. The block address is further divided into an address tag and cache index. Step 1 shows this division.

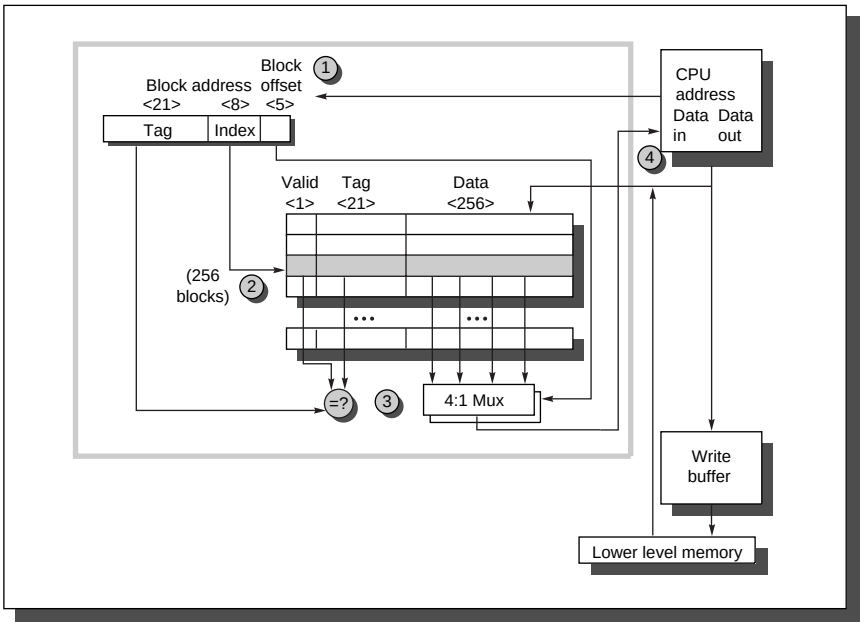

FIGURE 5.5 The organization of the data cache in the Alpha AXP 21064 microprocessor. The 8-KB cache is direct mapped with 32-byte blocks. It has 256 blocks selected by the 8-bit index. The four steps of a read hit, shown as circled numbers in order of occurrence, label this organization. Although we show a 4:1 multiplexer to select the desired 8 bytes, in reality the data RAM is organized 8 bytes wide and the multiplexer is unnecessary: 2 bits of the block offset join the index to supply the RAM address to select the proper 8 bytes (see Figure 5.8). Although not exercised in this example, the line from memory to the cache is used on a miss to load the cache.

The cache index selects the tag to be tested to see if the desired block is in the cache. The size of the index depends on cache size, block size, and set associativity. The 21064 cache is direct mapped, so set associativity is set to one, and we calculate the index as follows:

$$2^{\text{index}} = \frac{\text{Cache size}}{\text{Block size} \times \text{Set associativity}} = \frac{8192}{32 \times 1} = 256 = 2^8$$

Hence the index is 8 bits wide, and the tag is 29 – 8 or 21 bits wide.

Index selection is step 2 in Figure 5.5. Remember that direct mapping allows the data to be read and sent to the CPU in parallel with the tag being read and checked.

After reading the tag from the cache, it is compared to the tag portion of the block address from the CPU. This is step 3 in the figure. To be sure the tag con-

tains valid information, the valid bit must be set or else the results of the comparison are ignored.

Assuming the tag does match, the final step is to signal the CPU to load the data from the cache. The 21064 allows two clock cycles for these four steps, so the instructions in the following two clock cycles would stall if they tried to use the result of the load.

Handling writes is more complicated than handling reads in the 21064, as it is in any cache. If the word to be written is in the cache, the first three steps are the same. After the tag comparison indicates a hit, the data are written. (Section 5.5 shows how the 21064 avoids the extra time on write hits that this description implies.)

Since this is a write-through cache, the write process isn't yet over. The data are also sent to a write buffer that can contain up to four blocks that each can hold four 64-bit words. If the write buffer is empty, the data and the full address are written in the buffer, and the write is finished from the CPU's perspective; the CPU continues working while the write buffer prepares to write the word to memory. If the buffer contains other modified blocks, the addresses are checked to see if the address of this new data matches the address of the valid write buffer entry; if so, the new data are combined with that entry, called *write merging*. Without this optimization, four stores to sequential addresses would fill the buffer, even though these four words easily fit within a single block of the write buffer when merged. Figure 5.6 shows a write buffer with and without write merging. If the buffer is full and there is no address match, the cache (and CPU) must wait until the buffer has an empty entry.

So far we have assumed the common case of a cache hit. What happens on a miss? On a read miss, the cache sends a stall signal to the CPU telling it to wait, and 32 bytes are read from the next level of the hierarchy. The path to the next lower level is 16 bytes wide in the DEC 3000 model 800 workstation, one of several models that use the 21064. That takes 5 clock cycles per transfer, or 10 clock cycles for all 32 bytes. Since the data cache is direct mapped, there is no choice on which block to replace. Replacing a block means updating the data, the address tag, and the valid bit. On a write miss, the CPU writes "around" the cache to lower-level memory and does not affect the cache; that is, the 21064 follows the no-write-allocate rule.

We have seen how it works, but the *data* cache cannot supply all the memory needs of the processor: the processor also needs instructions. Although a single cache could try to supply both, it can be a bottleneck. For example, when a load or store instruction is executed, the pipelined processor will simultaneously request both a data word *and* an instruction word. Hence a single cache would present a structural hazard for loads and stores, leading to stalls. One simple way to conquer this problem is to divide it: one cache is dedicated to instructions and another to data. Separate caches are found in most recent processors, including the Alpha AXP 21064. It has an 8-KB instruction cache that is nearly identical to its 8-KB data cache in Figure 5.5.

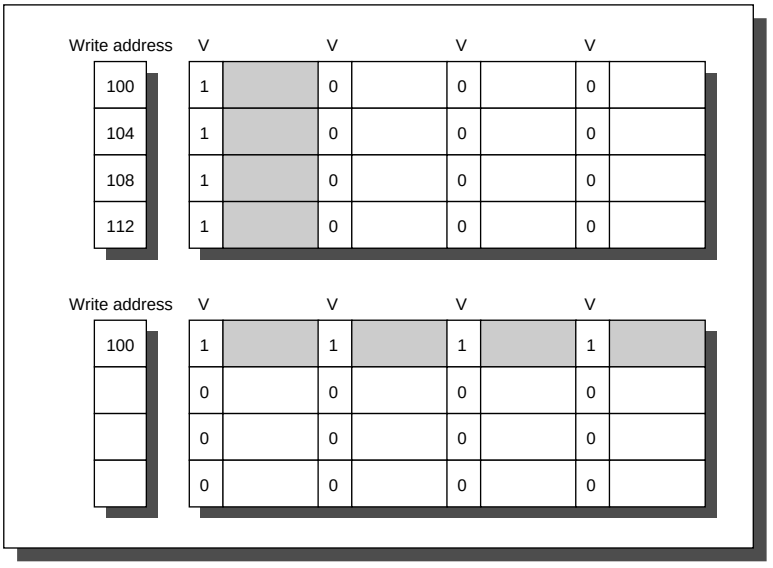

FIGURE 5.6 To illustrate write merging, the write buffer on top does not use it while the write buffer on the bottom does. Each buffer has four entries, and each entry holds four 64-bit words. The address for each entry is on the left, with valid bits (V) indicating whether or not the next sequential four bytes are occupied in this entry. The four writes are merged into a single buffer entry with write merging; without it, all four entries are used. Without write merging, the blocks to the right in the upper drawing would only be used for instructions that wrote multiple words at the same time. (The Alpha is a 64-bit architecture so its buffer is really 8 bytes per word.)

The CPU knows whether it is issuing an instruction address or a data address, so there can be separate ports for both, thereby doubling the bandwidth between the memory hierarchy and the CPU. Separate caches also offer the opportunity of optimizing each cache separately: different capacities, block sizes, and associativities may lead to better performance. (In contrast to the instruction caches and data caches of the 21064, the terms *unified* or *mixed* are applied to caches that can contain either instructions or data.)

Figure 5.7 shows that instruction caches have lower miss rates than data caches. Separating instructions and data removes misses due to conflicts between instruction blocks and data blocks, but the split also fixes the cache space devoted to each type. Which is more important to miss rates? A fair comparison of separate instruction and data caches to unified caches requires the total cache size to be the same. For example, a separate 1-KB instruction cache and 1-KB data cache should be compared to a 2-KB unified cache. Calculating the average miss rate with separate instruction and data caches necessitates knowing the percentage of memory references to each cache. Figure 2.26 on page 105 suggests the

Size	Instruction cache	Data cache	Unified cache
1 KB	3.06%	24.61%	13.34%
2 KB	2.26%	20.57%	9.78%
4 KB	1.78%	15.94%	7.24%
8 KB	1.10%	10.19%	4.57%
16 KB	0.64%	6.47%	2.87%
32 KB	0.39%	4.82%	1.99%
64 KB	0.15%	3.77%	1.35%
128 KB	0.02%	2.88%	0.95%

FIGURE 5.7 Miss rates for instruction, data, and unified caches of different sizes. The data are for a direct-mapped cache with 32-byte blocks for an average of SPEC92 benchmarks on the DECstation 5000 [Gee et al. 1993]. The percentage of instruction references is about 75%.

split is 100%/(100% + 26% + 9%) or about 75% instruction references to (26% + 9%)/(100% + 26% + 9%) or about 25% data references. Splitting affects performance beyond what is indicated by the change in miss rates, as we shall see in a little bit.

Cache Performance

Because instruction count is independent of the hardware, it is tempting to evaluate CPU performance using that number. As we saw in Chapter 1, however, such indirect performance measures have waylaid many a computer designer. The corresponding temptation for evaluating memory-hierarchy performance is to concentrate on miss rate, because it, too, is independent of the speed of the hardware. As we shall see, miss rate can be just as misleading as instruction count. A better measure of memory-hierarchy performance is the average time to access memory:

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

where *Hit time* is the time to hit in the cache; we have seen the other two terms before. The components of average access time can be measured either in absolute time—say, 2 nanoseconds on a hit—or in the number of clock cycles that the CPU waits for the memory—such as a miss penalty of 50 clock cycles. Remember that average memory access time is still an indirect measure of performance; although it is a better measure than miss rate, it is not a substitute for execution time.

This formula can help us decide between split caches and a unified cache.

EXAMPLE Which has the lower miss rate: a 16-KB instruction cache with a 16-KB data cache or a 32-KB unified cache? Use the miss rates in Figure 5.7 to help calculate the correct answer. Assume a hit takes 1 clock cycle and the miss penalty is 50 clock cycles, and a load or store hit takes 1 extra clock cycle on a unified cache since there is only one cache port to satisfy

two simultaneous requests. Using the pipelining terminology of the previous chapter, the unified cache leads to a structural hazard. What is the average memory access time in each case? Assume write-through caches with a write buffer and ignore stalls due to the write buffer.

ANSWER As stated above, about 75% of the memory accesses are instruction references. Thus, the overall miss rate for the split caches is

$$(75\% \times 0.64\%) + (25\% \times 6.47\%) = 2.10\%$$

According to Figure 5.7, a 32-KB unified cache has a slightly lower miss rate of 1.99%.

The average memory access time formula can be divided into instruction and data accesses:

$$\begin{aligned}&\text{Average memory access time}\\&= \%\ \text{instructions} \times (\text{Hit time} + \text{Instruction miss rate} \times \text{Miss penalty}) +\\&\%\ \text{data} \times (\text{Hit time} + \text{Data miss rate} \times \text{Miss penalty})\end{aligned}$$

So the time for each organization is

$$\begin{aligned}&\text{Average memory access time}_{\text{split}}\\&= 75\% \times (1 + 0.64\% \times 50) + 25\% \times (1 + 6.47\% \times 50)\\&= (75\% \times 1.32) + (25\% \times 4.235) = 0.990 + 1.059 = 2.05\\&\text{Average memory access time}_{\text{unified}}\\&= 75\% \times (1 + 1.99\% \times 50) + 25\% \times (1 + 1 + 1.99\% \times 50)\\&= (75\% \times 1.995) + (25\% \times 2.995) = 1.496 + 0.749 = 2.24\end{aligned}$$

Hence the split caches in this example—which offer two memory ports per clock cycle, thereby avoiding the structural hazard—have a better average memory access time than the single-ported unified cache even though their effective miss rate is *higher*. ■

In Chapter 1 we saw another formula for the memory hierarchy:

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

To simplify evaluation of cache alternatives, sometimes designers assume that all memory stalls are due to cache misses since the memory hierarchy typically dominates other reasons for stalls, such as contention due to I/O devices using memory. We use this simplifying assumption here, but it is important to account for *all* memory stalls when calculating final performance!

The CPU time formula above raises the question whether the clock cycles for a cache hit should be considered part of CPU execution clock cycles or part of memory stall clock cycles. Although either convention is defensible, the most widely accepted is to include hit clock cycles in CPU execution clock cycles.

Memory stall clock cycles can then be defined in terms of the number of memory accesses per program, miss penalty (in clock cycles), and miss rate for reads and writes:

$$\begin{aligned}\text{Memory stall clock cycles} &= \text{Reads} \times \text{Read miss rate} \times \text{Read miss penalty} \\ &\quad + \text{Writes} \times \text{Write miss rate} \times \text{Write miss penalty}\end{aligned}$$

We often simplify the complete formula by combining the reads and writes and finding the average miss rates and miss penalty for reads *and* writes:

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

This formula is an approximation since the miss rates and miss penalties are often different for reads and writes.

Factoring instruction count (IC) from execution time and memory stall cycles, we now get a CPU time formula that includes memory accesses per instruction, miss rate, and miss penalty:

$$\text{CPU time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss rate} \times \text{Miss penalty} \right) \times \text{Clock cycle time}$$

Some designers prefer measuring miss rate as *misses per instruction* rather than misses per memory reference:

$$\frac{\text{Misses}}{\text{Instruction}} = \frac{\text{Memory accesses} \times \text{Miss rate}}{\text{Instruction}}$$

The advantage of this measure is that it is independent of the hardware implementation. For example, the 21064 instruction prefetch unit can make repeated references to a single word (see section 5.10), which can artificially reduce the miss rate if measured as misses per memory reference rather than per instruction executed. The drawback is that misses per instruction is architecture dependent; for example, the average number of memory accesses per instruction may be very different for an 80x86 versus DLX. Thus misses per instruction is most popular with architects working with a single computer family. They then use this version of the CPU time formula:

$$\text{CPU time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \frac{\text{Memory stall clock cycles}}{\text{Instruction}} \right) \times \text{Clock cycle time}$$

We can now explore the impact of caches on performance.

EXAMPLE Let's use a machine similar to the Alpha AXP as a first example. Assume the cache miss penalty is 50 clock cycles, and all instructions normally take 2.0 clock cycles (ignoring memory stalls). Assume the miss rate is 2%, and

there is an average of 1.33 memory references per instruction. What is the impact on performance when behavior of the cache is included?

ANSWER

$$\text{CPU time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \frac{\text{Memory stall clock cycles}}{\text{Instruction}} \right) \times \text{Clock cycle time}$$

The performance, including cache misses, is

$$\begin{aligned} \text{CPU time}_{\text{with cache}} &= \text{IC} \times (2.0 + (1.33 \times 2\% \times 50)) \times \text{Clock cycle time} \\ &= \text{IC} \times 3.33 \times \text{Clock cycle time} \end{aligned}$$

The clock cycle time and instruction count are the same, with or without a cache, so CPU time increases with CPI from 2.0 for a "perfect cache" to 3.33 with a cache that can miss. Hence, including the memory hierarchy in the CPI calculations stretches the CPU time by a factor of 1.67. Without any memory hierarchy at all the CPI would increase to $2.0 + 50 \times 1.33$ or 68.5—a factor of over 30 times longer! ■

As this example illustrates, cache behavior can have enormous impact on performance. Furthermore, cache misses have a double-barreled impact on a CPU with a low CPI and a fast clock:

1. The lower the $\text{CPI}_{\text{execution}}$, the higher the *relative* impact of a fixed number of cache miss clock cycles.
2. When calculating CPI, the cache miss penalty is measured in CPU clock cycles for a miss. Therefore, even if memory hierarchies for two computers are identical, the CPU with the higher clock rate has a larger number of clock cycles per miss and hence the memory portion of CPI is higher.

The importance of the cache for CPUs with low CPI and high clock rates is thus greater, and, consequently, greater is the danger of neglecting cache behavior in assessing performance of such machines. Amdahl's Law strikes again!

Although minimizing average memory access time is a reasonable goal and we will use it in much of this chapter, keep in mind that the final goal is to reduce CPU execution time. The next example shows how these two can differ.

EXAMPLE

What is the impact of two different cache organizations on the performance of a CPU? Assume that the CPI with a perfect cache is 2.0 and the clock cycle time is 2 ns, that there are 1.3 memory references per instruction, and that the size of both caches is 64 KB and both have a block size of 32 bytes. One cache is direct mapped and the other is two-way set associative. Figure 5.8 shows that for set-associative caches we must add a multiplexer to select between the blocks in the set depending on the tag

match. Since the speed of the CPU is tied directly to the speed of a cache hit, assume the CPU clock cycle time must be stretched 1.10 times to accommodate the selection multiplexer of the set-associative cache. To the first approximation, the cache miss penalty is 70 ns for either cache organization. (In practice it must be rounded up or down to an integer number of clock cycles.) First, calculate the average memory access time, and then CPU performance. Assume the hit time is one clock cycle. Assume that the miss rate of a direct-mapped 64-KB cache is 1.4%, and the miss rate for a two-way set-associative cache of the same size is 1.0%.

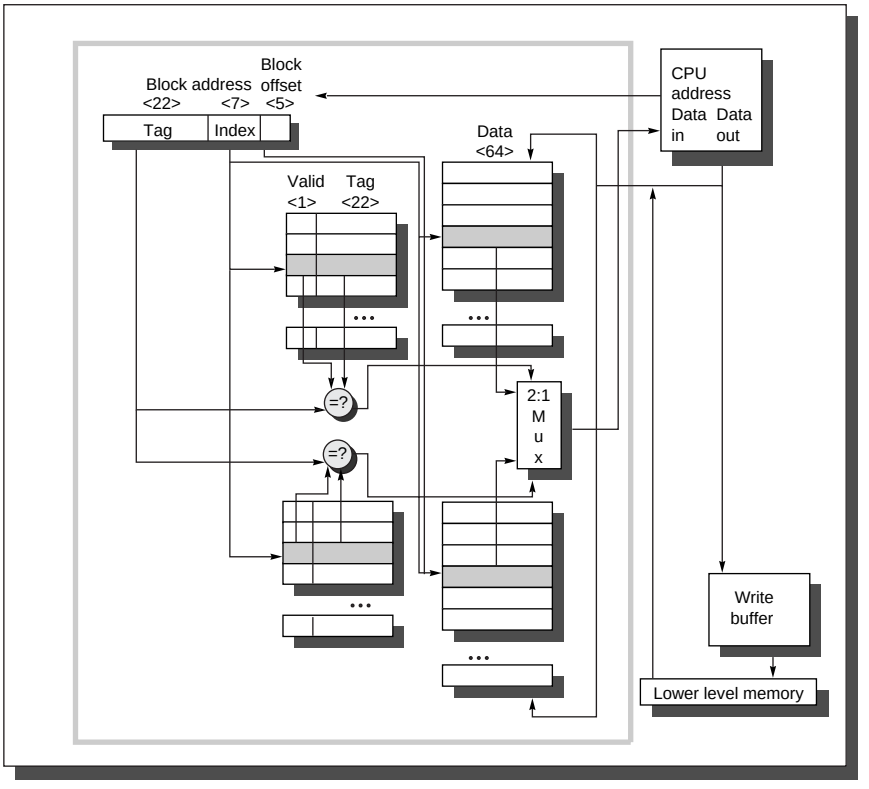

FIGURE 5.8 A two-way set-associative version of the 8-KB cache of Figure 5.5, showing the extra multiplexer in the path. Unlike the prior figure, the data portion of the cache is drawn more realistically, with the two leftmost bits of the block offset combined with the index to address the desired 64-bit word in memory, which is then sent to the CPU.

ANSWER Average memory access time is

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

Thus, the time for each organization is

$$\text{Average memory access time}_{\text{1-way}} = 2.0 + (.014 \times 70) = 2.98 \text{ ns}$$
$$\text{Average memory access time}_{\text{2-way}} = 2.0 \times 1.10 + (.010 \times 70) = 2.90 \text{ ns}$$

The average memory access time is better for the two-way set-associative cache.

CPU performance is

$$\begin{aligned}\text{CPU time} &= \text{IC} \times \left(\text{CPI}_{\text{Execution}} + \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}\right) \times \text{Clock cycle time} \\ &= \text{IC} \times \left[(\text{CPI}_{\text{Execution}} \times \text{Clock cycle time}) \right. \\ &\quad \left. + \left(\frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss rate} \times \text{Miss penalty} \times \text{Clock cycle time}\right)\right]\end{aligned}$$

Substituting 70 ns for (Miss penalty × Clock cycle time), the performance of each cache organization is

$$\text{CPU time}_{\text{1-way}} = \text{IC} \times (2 \times 2.0 + (1.3 \times 0.014 \times 70)) = 5.27 \times \text{IC}$$
$$\text{CPU time}_{\text{2-way}} = \text{IC} \times (2 \times 2.0 \times 1.10 + (1.3 \times 0.010 \times 70)) = 5.31 \times \text{IC}$$

and relative performance is

$$\frac{\text{CPU time}_{\text{2-way}}}{\text{CPU time}_{\text{1-way}}} = \frac{5.31 \times \text{Instruction count}}{5.27 \times \text{Instruction count}} = \frac{5.31}{5.27} = 1.01$$

In contrast to the results of average memory access time comparison, the direct-mapped cache leads to slightly better average performance because the clock cycle is stretched for *all* instructions for the two-way case, even if there are fewer misses. Since CPU time is our bottom-line evaluation, and since direct mapped is simpler to build, the preferred cache is direct mapped in this example. ■

Improving Cache Performance

The increasing gap between CPU and main memory speeds shown in Figure 5.1 has attracted the attention of many architects. A bibliographic search for the years 1989 –95 revealed more than 1600 research papers on the subject of caches. Your authors' job was to survey all 1600 papers, decide what is and is not worthwhile, translate the results into a common terminology, reduce the results to their essence, write in an intriguing fashion, and provide just the right amount of detail! Fortunately, the average memory access time formula gave us a framework to present cache optimizations as well as the techniques for improving caches:

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

Hence we organize 15 cache optimizations into three categories:

- Reducing the miss rate (Section 5.3)
- Reducing the miss penalty (Section 5.4)
- Reducing the time to hit in the cache (Section 5.5)

Figure 5.29 on page 427 concludes with a summary of the implementation complexity and the performance benefits of the 15 techniques presented.

5.3 Reducing Cache Misses

Most cache research has concentrated on reducing the miss rate, so that is where we start our exploration. To gain better insights into the causes of misses, we start with a model that sorts all misses into three simple categories:

- *Compulsory*—The very first access to a block *cannot be* in the cache, so the block must be brought into the cache. These are also called *cold start misses* or *first reference misses*.
- *Capacity*—If the cache cannot contain all the blocks needed during execution of a program, capacity misses will occur because of blocks being discarded and later retrieved.
- *Conflict*—If the block placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory and capacity misses) will occur because a block can be discarded and later retrieved if too many blocks map to its set. These are also called *collision misses* or *interference misses*.

Figure 5.9 shows the relative frequency of cache misses, broken down by the "three C's." Figure 5.10 presents the same data graphically. The top graph shows absolute miss rates; the bottom graph plots percentage of all the misses by type of miss as a function of cache size. To show the benefit of associativity, conflict misses are divided into misses caused by each decrease in associativity. Here are the four divisions:

- *Eight-way*—conflict misses due to going from fully associative (no conflicts) to eight-way associative
- *Four-way*—conflict misses due to going from eight-way associative to four-way associative
- *Two-way*—conflict misses due to going from four-way associative to two-way associative
- *One-way*—conflict misses due to going from two-way associative to one-way associative (direct mapped)

Cache size	Degree associative	Total miss rate	Miss rate components (relative percent) (Sum = 100% of total miss rate)					
			Compulsory		Capacity		Conflict	
1 KB	1-way	0.133	0.002	1%	0.080	60%	0.052	39%
1 KB	2-way	0.105	0.002	2%	0.080	76%	0.023	22%
1 KB	4-way	0.095	0.002	2%	0.080	84%	0.013	14%
1 KB	8-way	0.087	0.002	2%	0.080	92%	0.005	6%
2 KB	1-way	0.098	0.002	2%	0.044	45%	0.052	53%
2 KB	2-way	0.076	0.002	2%	0.044	58%	0.030	39%
2 KB	4-way	0.064	0.002	3%	0.044	69%	0.018	28%
2 KB	8-way	0.054	0.002	4%	0.044	82%	0.008	14%
4 KB	1-way	0.072	0.002	3%	0.031	43%	0.039	54%
4 KB	2-way	0.057	0.002	3%	0.031	55%	0.024	42%
4 KB	4-way	0.049	0.002	4%	0.031	64%	0.016	32%
4 KB	8-way	0.039	0.002	5%	0.031	80%	0.006	15%
8 KB	1-way	0.046	0.002	4%	0.023	51%	0.021	45%
8 KB	2-way	0.038	0.002	5%	0.023	61%	0.013	34%
8 KB	4-way	0.035	0.002	5%	0.023	66%	0.010	28%
8 KB	8-way	0.029	0.002	6%	0.023	79%	0.004	15%
16 KB	1-way	0.029	0.002	7%	0.015	52%	0.012	42%
16 KB	2-way	0.022	0.002	9%	0.015	68%	0.005	23%
16 KB	4-way	0.020	0.002	10%	0.015	74%	0.003	17%
16 KB	8-way	0.018	0.002	10%	0.015	80%	0.002	9%
32 KB	1-way	0.020	0.002	10%	0.010	52%	0.008	38%
32 KB	2-way	0.014	0.002	14%	0.010	74%	0.002	12%
32 KB	4-way	0.013	0.002	15%	0.010	79%	0.001	6%
32 KB	8-way	0.013	0.002	15%	0.010	81%	0.001	4%
64 KB	1-way	0.014	0.002	14%	0.007	50%	0.005	36%
64 KB	2-way	0.010	0.002	20%	0.007	70%	0.001	10%
64 KB	4-way	0.009	0.002	21%	0.007	75%	0.000	3%
64 KB	8-way	0.009	0.002	22%	0.007	78%	0.000	0%
128 KB	1-way	0.010	0.002	20%	0.004	40%	0.004	40%
128 KB	2-way	0.007	0.002	29%	0.004	58%	0.001	14%
128 KB	4-way	0.006	0.002	31%	0.004	61%	0.001	8%
128 KB	8-way	0.006	0.002	31%	0.004	62%	0.000	7%

FIGURE 5.9 Total miss rate for each size cache and percentage of each according to the "three C's." Compulsory misses are independent of cache size, while capacity misses decrease as capacity increases, and conflict misses decrease as associativity increases. Gee et al. [1993] calculated the average D-cache miss rate for the SPEC92 benchmark suite with 32-byte blocks and LRU replacement on a DECstation 5000. Figure 5.10 shows the same information graphically. The compulsory rate was calculated as the miss rate of a fully associative 1-MB cache. Note that the 2:1 cache rule of thumb (inside front cover) is supported by the statistics in this table: a direct-mapped cache of size *N* has about the same miss rate as a 2-way set-associative cache of size *N*/2.

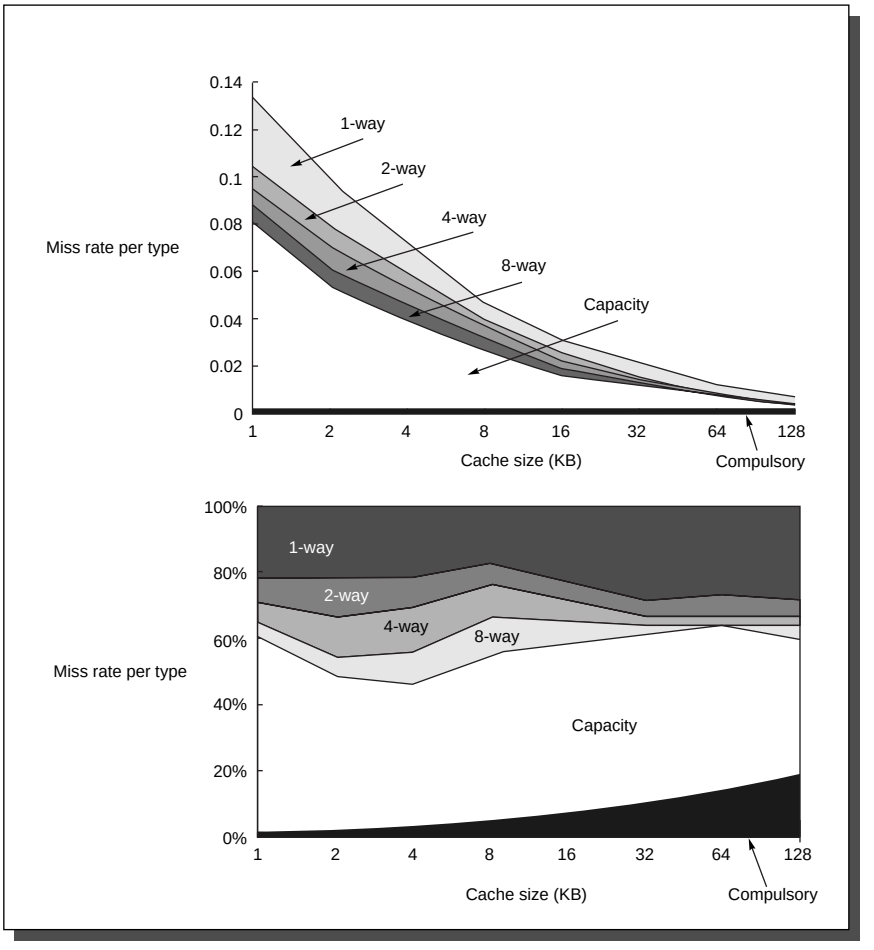

FIGURE 5.10 Total miss rate (top) and distribution of miss rate (bottom) for each size cache according to three C's for the data in Figure 5.9. The top diagram is the actual D-cache miss rates, while the bottom diagram is scaled to the direct-mapped miss ratios.

As we can see from the figures, the compulsory miss rate of the SPEC92 programs is very small, as it is for many long-running programs.

Having identified the three C's, what can a computer designer do about them? Conceptually, conflicts are the easiest: Fully associative placement avoids all conflict misses. Full associativity is expensive in hardware, however, and may slow the processor clock rate (see the example above), leading to lower overall performance.

There is little to be done about capacity except to enlarge the cache. If the upper-level memory is much smaller than what is needed for a program, and a

significant percentage of the time is spent moving data between two levels in the hierarchy, the memory hierarchy is said to *thrash*. Because so many replacements are required, thrashing means the machine runs close to the speed of the lower-level memory, or maybe even slower because of the miss overhead.

Another approach to improving the three C's is to make blocks larger to reduce the number of compulsory misses, but, as we shall see, large blocks can increase other kinds of misses.

The three C's give insight into the cause of misses, but this simple model has its limits; it gives you insight into average behavior but may not explain an individual miss. For example, changing cache size changes conflict misses as well as capacity misses, since a larger cache spreads out references to more blocks. Thus, a miss might move from a capacity miss to a conflict miss as cache size changes. Note that the three C's also ignore replacement policy, since it is difficult to model and since, in general, it is less significant. In specific circumstances the replacement policy can actually lead to anomalous behavior, such as poorer miss rates for larger associativity, which is contradictory to the three C's model.

Alas, many of the techniques that reduce miss rates also increase hit time or miss penalty. The desirability of reducing miss rates using the seven techniques presented in the rest of this section must be balanced against the goal of making the whole system fast. This first example shows the importance of a balanced perspective.

First Miss Rate Reduction Technique: Larger Block Size

This simplest way to reduce miss rate is to increase the block size. Figure 5.11 shows the trade-off of block size versus miss rate for a set of programs and cache sizes. Larger block sizes will reduce compulsory misses. This reduction occurs because the principle of locality has two components: temporal locality and spatial locality. Larger blocks take advantage of spatial locality.

At the same time, larger blocks increase the miss penalty. Since they reduce the number of blocks in the cache, larger blocks may increase conflict misses and even capacity misses if the cache is small. Clearly there is little reason to increase the block size to such a size that it *increases* the miss rate, but there is also no benefit to reducing miss rate if it increases the average memory access time; the increase in miss penalty may outweigh the decrease in miss rate.

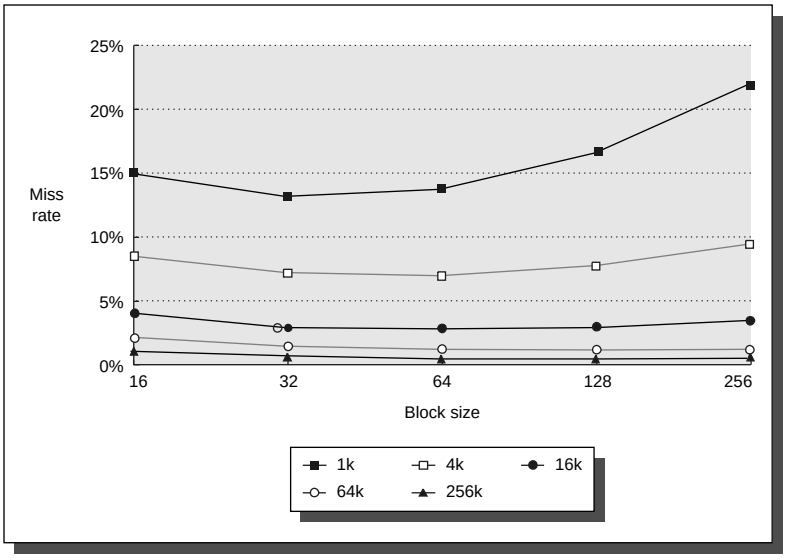

FIGURE 5.11 Miss rate versus block size for five different-sized caches. Each line represents a cache of different size. Figure 5.12 shows the data used to plot these lines. This graph is based on the same measurements found in Figure 5.10.

	Cache size				
Block size	**1K**	**4K**	**16K**	**64K**	**256K**
16	15.05%	8.57%	3.94%	2.04%	1.09%
32	13.34%	7.24%	2.87%	1.35%	0.70%
64	13.76%	7.00%	2.64%	1.06%	0.51%
128	16.64%	7.78%	2.77%	1.02%	0.49%
256	22.01%	9.51%	3.29%	1.15%	0.49%

FIGURE 5.12 Actual miss rate versus block size for five different-sized caches in Figure 5.11. Note that for a 1-KB cache, 64-byte, 128-byte, and 256-byte blocks have a higher miss rate than 32-byte blocks. In this example, the cache would have to be 256 KB in order for a 256-byte block to decrease misses.

EXAMPLE Figure 5.12 shows the actual miss rates plotted in Figure 5.11. Assume the memory system takes 40 clock cycles of overhead and then delivers 16 bytes every 2 clock cycles. Thus, it can supply 16 bytes in 42 clock cycles, 32 bytes in 44 clock cycles, and so on. Which block size has the minimum average memory access time for each cache size in Figure 5.12?

ANSWER Average memory access time is

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

If we assume the hit time is one clock cycle independent of block size, then the access time for a 16-byte block in a 1-KB cache is

$$\text{Average memory access time} = 1 + (15.05\% \times 42) = 7.321 \text{ clock cycles}$$

and for a 256-byte block in a 256-KB cache the average memory access time is

$$\text{Average memory access time} = 1 + (0.49\% \times 72) = 1.353 \text{ clock cycles}$$

Figure 5.13 shows the average memory access time for all block and cache sizes between those two extremes. The boldfaced entries show the fastest block size for a given cache size: 32 bytes for 1-KB, 4-KB, and 16-KB caches and 64 bytes for the larger caches. These sizes are, in fact, popular block sizes for processor caches today.

		Cache size				
Block size	**Miss penalty**	**1K**	**4K**	**16K**	**64K**	**256K**
16	42	7.321	4.599	2.655	1.857	1.458
32	44	**6.870**	**4.186**	**2.263**	1.594	1.308
64	48	7.605	4.360	2.267	**1.509**	**1.245**
128	56	10.318	5.357	2.551	1.571	1.274
256	72	16.847	7.847	3.369	1.828	1.353

FIGURE 5.13 Average memory access time versus block size for five different-sized caches in Figure 5.11. The smallest average time per cache size is boldfaced.

■

As in all of these techniques, the cache designer is trying to minimize both the miss rate and the miss penalty. The selection of block size depends on both the latency and bandwidth of the lower-level memory: high latency and high bandwidth encourage large block size since the cache gets many more bytes per miss for a small increase in miss penalty. Conversely, low latency and low bandwidth encourage smaller block sizes since there is little time saved from a larger block—twice the miss penalty of a small block may be close to the penalty of a block twice the size—and the larger number of small blocks may reduce conflict misses.

After seeing the positive and negative impact of larger block size on compulsory and capacity misses, we next look at the potential of higher associativity to reduce conflict misses.

Second Miss Rate Reduction Technique: Higher Associativity

Figures 5.9 and 5.10 above show how miss rates improve with higher associativity. There are two general rules of thumb that can be gleaned from these figures. The first is that eight-way set associative is for practical purposes as effective in reducing misses for these sized caches as fully associative. The second observation, called the *2:1 cache rule of thumb* and found on the front inside cover, is that a direct-mapped cache of size N has about the same miss rate as a 2-way set-associative cache of size $N/2$.

Like many of these examples, improving one aspect of the average memory access time comes at the expense of another. Increasing block size reduced miss rate while increasing miss penalty, and greater associativity can come at the cost of increased hit time. Hill [1988] found about a 10% difference in hit times for TTL or ECL board-level caches and a 2% difference for custom CMOS caches for direct-mapped caches versus two-way set-associative caches. Hence the pressure of a fast processor clock cycle encourages simple cache designs, but the increasing miss penalty rewards associativity, as the following example suggests.

EXAMPLE Assume that going to higher associativity would increase the clock cycle as suggested below:

$$\text{Clock cycle time}_{2\text{-way}} = 1.10 \times \text{Clock cycle time}_{1\text{-way}}$$
$$\text{Clock cycle time}_{4\text{-way}} = 1.12 \times \text{Clock cycle time}_{1\text{-way}}$$
$$\text{Clock cycle time}_{8\text{-way}} = 1.14 \times \text{Clock cycle time}_{1\text{-way}}$$

Assume that the hit time is 1 clock cycle, that the miss penalty for the direct-mapped case is 50 clock cycles, and that the miss penalty need not be rounded to an integral number of clock cycles. Using Figure 5.9 for miss rates, for which cache sizes are each of these three statements true?

$$\text{Average memory access time}_{8\text{-way}} < \text{Average memory access time}_{4\text{-way}}$$
$$\text{Average memory access time}_{4\text{-way}} < \text{Average memory access time}_{2\text{-way}}$$
$$\text{Average memory access time}_{2\text{-way}} < \text{Average memory access time}_{1\text{-way}}$$

ANSWER Average memory access time for each associativity is

$$\text{Average memory access time}_{8\text{-way}} = \text{Hit time}_{8\text{-way}} + \text{Miss rate}_{8\text{-way}} \times \text{Miss penalty}_{1\text{-way}} = 1.14 + \text{Miss rate}_{8\text{-way}} \times 50$$
$$\text{Average memory access time}_{4\text{-way}} = 1.12 + \text{Miss rate}_{4\text{-way}} \times 50$$
$$\text{Average memory access time}_{2\text{-way}} = 1.10 + \text{Miss rate}_{2\text{-way}} \times 50$$
$$\text{Average memory access time}_{1\text{-way}} = 1.00 + \text{Miss rate}_{1\text{-way}} \times 50$$

The miss penalty is the same time in each case, so we leave it as 50 clock cycles. For example, the average memory access time for a 1-KB direct-mapped cache is

$$\text{Average memory access time}_{\text{1-way}} = 1.00 + (0.133 \times 50) = 7.65$$

and the time for a 128-KB, eight-way set-associative cache is

$$\text{Average memory access time}_{\text{8-way}} = 1.14 + (0.006 \times 50) = 1.44$$

Using these formulas and the miss rates from Figure 5.9, Figure 5.14 shows the average memory access time for each cache and associativity. The figure shows that the formulas in this example hold for caches less than or equal to 16 KB. Starting with 32 KB, the average memory access time of four-way is less than two-way, and two-way is less than one-way, but eight-way cache is not less than four-way.

Note that we did not account for the slower clock rate on the rest of the program in this example, thereby understating the advantage of direct-mapped cache.

	Associativity			
Cache size (KB)	**One-way**	**Two-way**	**Four-way**	**Eight-way**
1	7.65	6.60	6.22	5.44
2	5.90	4.90	4.62	4.09
4	4.60	3.95	3.57	3.19
8	3.30	3.00	2.87	2.59
16	2.45	2.20	2.12	2.04
32	2.00	1.80	1.77	**1.79**
64	1.70	1.60	1.57	**1.59**
128	1.50	1.45	1.42	**1.44**

FIGURE 5.14 Average memory access time using miss rates in Figure 5.9 for parameters in the example. Boldface type means that this time is higher than the number to the left; that is, higher associativity *increases* average memory access time.

■

Third Miss Rate Reduction Technique: Victim Caches

Larger block size and higher associativity are two classic techniques to reduce miss rates that have been considered by architects since the earliest caches. Starting with this subsection, we see more recent inventions to reduce miss rate without affecting the clock cycle time or the miss penalty.

One solution that reduces conflict misses without impairing clock rate is to add a small, fully associative cache between a cache and its refill path. Figure 5.15 shows the organization. This *victim cache* contains only blocks that

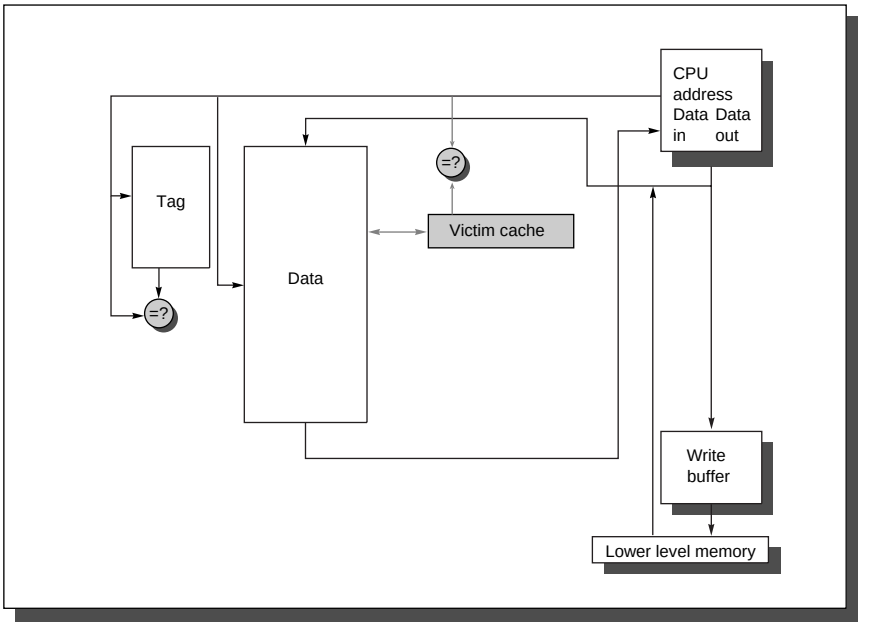

FIGURE 5.15 Placement of victim cache in the memory hierarchy.

are discarded from a cache because of a miss—"victims"—and are checked on a miss to see if they have the desired data before going to the next lower-level memory. If it is found there, the victim block and cache block are swapped. Jouppi [1990] found that victim caches of one to five entries are effective at reducing conflict misses, especially for small, direct-mapped data caches. Depending on the program, a four-entry victim cache removed 20% to 95% of the conflict misses in a 4-KB direct-mapped data cache.

Fourth Miss Rate Reduction Technique: Pseudo-Associative Caches

Another approach to getting the miss rate of set-associative caches and the hit speed of direct mapped is called *pseudo-associative* or *column associative*. A cache access proceeds just as in the direct-mapped cache for a hit. On a miss, however, before going to the next lower level of the memory hierarchy, another

cache entry is checked to see if it matches there. A simple way is to invert the most significant bit of the index field to find the other block in the "pseudo set."

Pseudo-associative caches then have one fast and one slow hit time—corresponding to a regular hit and a pseudo hit—in addition to the miss penalty. Figure 5.16 shows the relative times. The danger is if many of the fast hit times of the direct-mapped cache became slow hit times in the pseudo-associative cache, then the performance would be degraded by this optimization. Hence it is important to be able to indicate for each set which block should be the fast hit and which should be the slow one; one way is simply to swap the contents of the blocks.

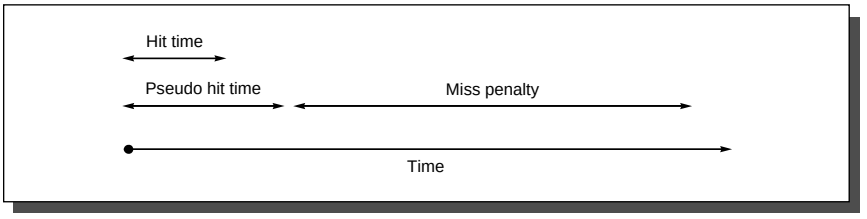

FIGURE 5.16 Relationship between a regular hit time, pseudo hit time, and miss penalty.

Let's do an example to see how well pseudo-associativity works.

EXAMPLE Assume that it takes two extra cycles to find the entry in the alternative location if it is not found in the direct-mapped location: one cycle to check and one cycle to swap. Using the parameters from the previous example, which of direct-mapped, two-way set-associative, and pseudo-associative organizations is fastest for 2-KB and 128-KB sizes?

ANSWER The average memory access time for pseudo-associative caches starts with the standard formula:

$$\text{Average memory access time}_{\text{pseudo}} = \text{Hit time}_{\text{pseudo}} + \text{Miss rate}_{\text{pseudo}} \times \text{Miss penalty}_{\text{pseudo}}$$

Let's start with the last part of the equation. The pseudo miss penalty is one cycle more than a normal miss penalty, to account for the time to check the alternative location.To determine the miss rate we need to see when misses occur. As long as we invert the most significant bit of the index to find the other block, the two blocks in the "pseudo set" are selected using the same index that would be used in a two-way set-associative cache and hence have the same miss rates. Thus the last part of the equation is

$$\text{Miss rate}_{\text{pseudo}} \times \text{Miss penalty}_{\text{pseudo}} = \text{Miss rate}_{\text{2-way}} \times \text{Miss penalty}_{\text{1-way}}$$

Returning to the beginning of the equation, the hit time for a pseudo-associative cache is the time to hit in a direct-mapped cache plus the fraction of accesses that are found in the pseudo-associative search times the extra time it takes to find the hit:

$$\text{Hit time}_{\text{pseudo}} = \text{Hit time}_{\text{1-way}} + \text{Alternate hit rate}_{\text{pseudo}} \times 2$$

The hit rate for the pseudo-associative search is the difference between the hits that would occur in a two-way set-associative cache and the number of hits in a direct-mapped cache:

$$\begin{aligned}\text{Alternate hit rate}_{\text{pseudo}} &= \text{Hit rate}_{\text{2-way}} - \text{Hit rate}_{\text{1-way}} \\ &= (1 - \text{Miss rate}_{\text{2-way}}) - (1 - \text{Miss rate}_{\text{1-way}}) \\ &= \text{Miss rate}_{\text{1-way}} - \text{Miss rate}_{\text{2-way}}\end{aligned}$$

But it is slightly more complex. The miss rate is of a direct-mapped cache half the size—since half of the cache is reserved for alternate locations—while the whole cache has the contents of a two-way set-associative cache. Putting the pieces back together:

$$\text{Average memory access time}_{\text{pseudo}} = \text{Hit time}_{\text{1-way}} + (\text{Miss rate}_{\text{1-way}} - \text{Miss rate}_{\text{2-way}}) \times 2 + \text{Miss rate}_{\text{2-way}} \times \text{Miss penalty}_{\text{1-way}}$$

Figure 5.9 supplies the values we need to plug into our formulas:

$$\text{Average memory access time}_{\text{pseudo 2 KB}} = 1 + (0.113 - 0.076) \times 2 + (0.076 \times (50 + 1)) = 1 + 0.074 + 3.876 = 4.950$$

$$\text{Average memory access time}_{\text{pseudo 128 KB}} = 1 + (0.014 - 0.007) \times 2 + (0.007 \times (50 + 1)) = 1 + 0.014 + 0.357 = 1.371$$

From Figure 5.14 in the last example we know these results for 2-KB caches:

$$\text{Average memory access time}_{\text{1-way}} = 5.90 \text{ clock cycles}$$
$$\text{Average memory access time}_{\text{2-way}} = 4.90 \text{ clock cycles}$$

For 128-KB caches the times are

$$\text{Average memory access time}_{\text{1-way}} = 1.50 \text{ clock cycles}$$
$$\text{Average memory access time}_{\text{2-way}} = 1.45 \text{ clock cycles}$$

The pseudo-associative cache is fastest for the 128-KB cache while the two-way set associative is fastest for the 2-KB cache. ■

Although an attractive idea on paper, variable hit times can complicate a pipelined CPU design. Hence the authors expect the most likely use of pseudo-associativity is with caches further from the processor (see the description of second-level caches in the next section).

Fifth Miss Rate Reduction Technique: Hardware Prefetching of Instructions and Data

Victim caches and pseudo-associativity both promise to improve miss rates without affecting the processor clock rate. A third way is to prefetch items before they are requested by the processor. Both instructions and data can be prefetched,

either directly into the caches or into an external buffer that can be more quickly accessed than main memory.

Instruction prefetch is frequently done in hardware outside of the cache. For example, the Alpha AXP 21064 microprocessor fetches two blocks on a miss: the requested block and the next consecutive block. The requested block is placed in the instruction cache when it returns, and the prefetched block is placed into the instruction stream buffer. If the requested block is present in the instruction stream buffer, the original cache request is canceled, the block is read from the stream buffer, and the next prefetch request is issued. There is never more than one 32-byte block in the 21064 instruction stream buffer. Jouppi [1990] found that a single instruction stream buffer would catch 15% to 25% of the misses from a 4-KB direct-mapped instruction cache with 16-byte blocks. With 4 blocks in the instruction stream buffer the hit rate improves to about 50%, and with 16 blocks to 72%.

A similar approach can be applied to data accesses. Jouppi found that a single data stream buffer caught about 25% of the misses from the 4-KB direct-mapped cache. Instead of having a single stream, there could be multiple stream buffers beyond the data cache, each prefetching at different addresses. Jouppi found that four data stream buffers increased the data hit rate to 43%. Palacharla and Kessler [1994] looked at a set of scientific programs and considered stream buffers that could handle either instructions or data. They found that eight stream buffers could capture 50% to 70% of all misses from a processor with two 64-KB four-way set-associative caches, one for instructions and the other for data.

EXAMPLE What is the effective miss rate of the Alpha AXP 21064 using instruction prefetching? How much bigger an instruction cache would be needed in the Alpha AXP 21064 to match the average access time if prefetching were removed?

ANSWER We assume it takes 1 extra clock cycle if the instruction misses the cache but is found in the prefetch buffer. Here is our revised formula:

$$\text{Average memory access time}_{\text{prefetch}} = \text{Hit time} + \text{Miss rate} \times \text{Prefetch hit rate} \times 1 + \text{Miss rate} \times (1 - \text{Prefetch hit rate}) \times \text{Miss penalty}$$

Let's assume the prefetch hit rate is 25%. Figure 5.7 on page 384 gives the miss rate for an 8-KB instruction cache as 1.10%. Using the parameters from the Example on page 386, the hit time is 2 clock cycles, and the miss penalty is 50 clock cycles:

$$\text{Average memory access time}_{\text{prefetch}} = 2 + (1.10\% \times 25\% \times 1) + (1.10\% \times (1 - 25\%) \times 50) = 2 + 0.00275 + 0.413 = 2.415$$

To find the effective miss rate with the equivalent performance, we start with the original formula and solve for the miss rate:

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

$$\text{Miss rate} = \frac{\text{Average memory access time} - \text{Hit time}}{\text{Miss penalty}}$$

$$\text{Miss rate} = \frac{2.415 - 2}{50} = \frac{0.415}{50} = 0.83\%$$

Our calculation suggests that the effective miss rate of prefetching with an 8-KB cache is 0.83%. Figure 5.7 on page 384 gives the miss rate of a 16-KB instruction cache as 0.64%, so 8 KB with prefetching is midway between the 1.10% and 0.64% miss rates of the 8-KB and 16-KB caches. ■

Prefetching relies on utilizing memory bandwidth that otherwise would be unused, and can actually lower performance if it interferes with demand misses. Help from compilers can reduce useless prefetching.

Sixth Miss Rate Reduction Technique: Compiler-Controlled Prefetching

An alternative to hardware prefetching is for the compiler to insert prefetch instructions to request the data before they are needed. There are several flavors of prefetch:

- *Register prefetch* will load the value into a register.
- *Cache prefetch* loads data only into the cache and not the register.

Either of these can be *faulting* or *nonfaulting*; that is, the address does or does not cause an exception for virtual address faults and protection violations. Using this terminology, a normal load instruction could be considered a "faulting register prefetch instruction." Nonfaulting prefetches simply turn into no-ops if they would normally result in an exception. The most effective prefetch is "semantically invisible" to a program: it doesn't change the contents of registers or memory and it cannot cause virtual memory faults. This section assumes nonfaulting cache prefetch, also called *nonbinding* prefetch.

Prefetching makes sense only if the processor can proceed while the prefetched data are being fetched; that is, the caches continue to supply instructions and data while waiting for the prefetched data to return. Such a nimble cache is called a *nonblocking* cache or *lockup-free* cache; we'll discuss it in more detail later.

Like hardware-controlled prefetching, the goal is to overlap execution with the prefetching of data. Loops are the key targets, as they lend themselves to prefetch optimizations. If the miss penalty is small, the compiler just unrolls the loop once or twice and it schedules the prefetches with the execution. If the miss

penalty is large, it uses software pipelining (page 290 in Chapter 4) or unrolls many times to prefetch data for a future iteration.

Issuing prefetch instructions incurs an instruction overhead, however, so care must be taken to ensure that such overheads do not exceed the benefits. By concentrating on references that are likely to be cache misses, programs can avoid unnecessary prefetches while improving average memory access time significantly.

EXAMPLE For the code below, determine which accesses are likely to cause data cache misses. Next, insert prefetch instructions to reduce misses. Finally, calculate the number of prefetch instructions executed and the misses avoided due to prefetching. Let's assume we have an 8-KB direct-mapped data cache with 16-byte blocks, it is a write-back cache that does write allocate, and that the elements of a and b are 8 bytes long as they are double-precision floating-point arrays with 3 rows and 100 columns for a and 101 rows and 3 columns for b. Let's also assume they are not in the cache at the start of the program.

```
for (i = 0; i < 3; i = i+1)
        for (j = 0; j < 100; j = j+1)
                a[i][j] = b[j][0] * b[j+1][0];
```

ANSWER The compiler will first determine which accesses are likely to cause cache misses; otherwise, we will waste time on issuing prefetch instructions for data that would be hits. Elements of a are written in the order that they are stored in memory, so a will benefit from spatial locality: the even values of `j` will miss and the odd values will hit. Since a has 3 rows and 100 columns, its accesses will lead to $\frac{3 \times 100}{2}$ or 150 misses. The array b does not benefit from spatial locality since the accesses are not in the order it is stored. The array b does benefit twice from temporal locality: the same elements are accessed for each iteration of `i`, and each iteration of `j` uses the same value of b as the last iteration. Ignoring potential conflict misses, the misses due to b will be for `b[j+1][0]` accesses when `i` = 0, and also the first access to `b[j][0]` when `j` = 0. Since `j` goes from 0 to 99 when `i` = 0, accesses to b lead to 100 + 1 or 101 misses. Thus this loop will miss the data cache approximately 150 + 101 or 251 times.

To simplify our optimization, we will not worry about prefetching the first accesses of the loop nor suppressing the prefetches at the end of the loop; if these were *faulting* prefetches, we could not take this luxury. Given our analysis of misses, we split the loop so the first loop will prefetch b as well as a, and the second loop will just prefetch a, since b will have already been prefetched. Let's assume that the miss penalty is so large we need to prefetch at least seven iterations in advance.

```
for (j = 0; j < 100; j = j+1) {
        prefetch(b[j+7][0]);
        /* b(j,0) for 7 iterations later */
        prefetch(a[0][j+7]);
        /* a(0,j) for 7 iterations later */
        a[0][j] = b[j][0] * b[j+1][0];};
for (i = 1; i < 3; i = i+1)
        for (j = 0; j < 100; j = j+1) {
                prefetch(a[i][j+7]);
                /* a(i,j) for +7 iterations */
                a[i–1][j] = b[j][0] *b[j+1][0];}
```

This revised code prefetches `a[i][7]` through `a[i][99]` and `b[7][0]` through `b[99][0]`, reducing the number of nonprefetched misses to

$$\frac{3 \times 7}{2} + 8 = 11 + 8 = 19$$

The cost of avoiding 232 cache misses is executing 400 prefetch instructions, very likely a good trade-off. ■

EXAMPLE Calculate the time saved in the example above. Ignore instruction cache misses and assume there are no conflict or capacity misses in the data cache. Assume that prefetches can overlap with each other and with cache misses, thereby transferring at the maximum memory bandwidth. Here are the key loop times ignoring cache misses: the original loop takes 7 clock cycles per iteration, the first prefetch loop takes 9 clock cycles per iteration, and the second prefetch loop takes 8 clock cycles per iteration (including the overhead of the outer for loop). A miss takes 50 clock cycles.

ANSWER The original doubly nested loop executes the multiply 3×100 or 300 times. Since the loop takes 7 clock cycles per iteration, the total is 300×7 or 2100 clock cycles plus cache misses. Cache misses add 251×50 or 12,550 clock cycles, giving a total of 14,650 clock cycles. The first prefetch loop iterates 100 times; at 9 clock cycles per iteration the total is 900 clock cycles plus cache misses. They add 11×50 or 550 clock cycles for cache misses, giving a total of 1450. The second loop executes 2×100 or 200 times, and at 8 clock cycles per iteration it takes 1600 clock cycles plus 8×50 or 400 clock cycles for cache misses. This gives a total of 2000 clock cycles. From the prior example we know that this code executes 400 prefetch instructions during the 1450 + 2000 or 3450 clock cycles to execute these two loops. If we assume that the prefetches are completely overlapped with the rest of the execution, then the prefetch code is 14,650/3450 or 4.2 times faster. ■

Seventh Miss Rate Reduction Technique: Compiler Optimizations

Thus far our techniques to reduce misses have required changes to or additions to the hardware: larger blocks, higher associativity, pseudo-associativity, hardware prefetching, or prefetch instructions. This final technique reduces miss rates without any hardware changes!

This magical reduction comes from optimized software—the hardware designer's favorite solution. The increasing performance gap between processors and main memory has inspired compiler writers to scrutinize the memory hierarchy to see if compile time optimizations can improve performance. Once again research is split between improvements in instruction misses and improvements in data misses.

Code can easily be rearranged without affecting correctness; for example, reordering the procedures of a program might reduce instruction miss rates by reducing conflict misses. McFarling [1989] looked at using profiling information to determine likely conflicts between groups of instructions, and reordered the instructions to reduce misses by 50% for a 2-KB direct-mapped instruction cache with 4-byte blocks, and by 75% in an 8-KB cache. McFarling got the best performance when it was possible to prevent some instructions from ever entering the cache, but even without that feature, optimized programs on a direct-mapped cache had lower miss rates than unoptimized programs on an eight-way set-associative cache of the same size.

Data have even fewer restrictions on location than code. The goal of such transformations is to try to improve the spatial and temporal locality of the data. For example, array calculations can be changed to operate on all the data in a cache block rather than blindly striding through arrays in the order the programmer happened to place the loop.

To give a feeling of this type of optimization, we will show four examples, transforming the C code by hand to reduce cache misses. Figure 5.17 shows the performance improvement in using these optimizations on a subset of the SPEC92 floating-point benchmarks.

Merging Arrays

This first technique reduces misses by improving spatial locality. Some programs reference multiple arrays in the same dimension with the same indices at the same time. The danger is that these accesses will interfere with each other, leading to conflict misses. This danger is removed by combining these independent matrices into a single compound array so that a single cache block can contain the desired elements.

```
/* Before */
int val[SIZE];
int key[SIZE];
```

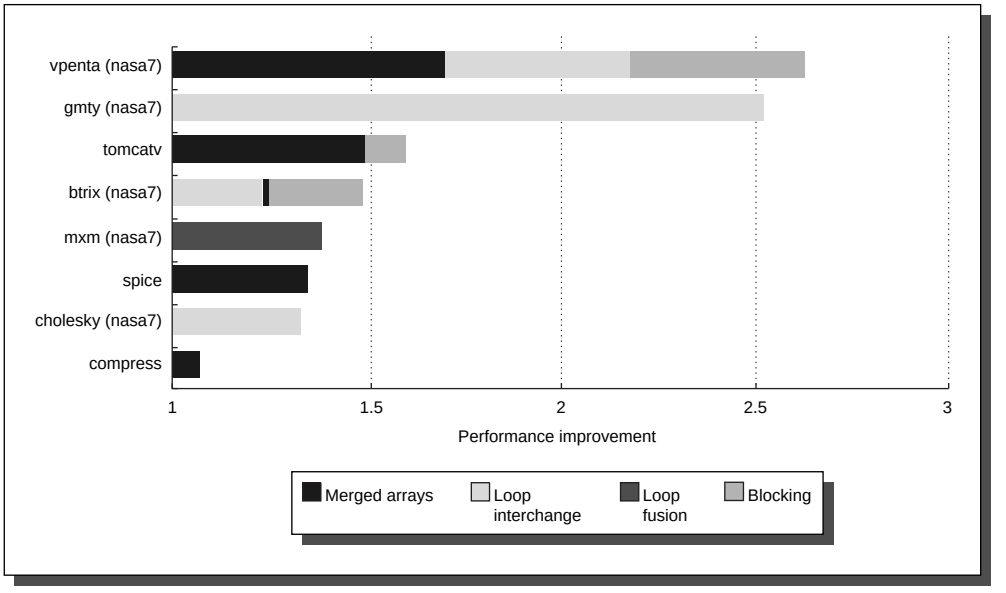

FIGURE 5.17 Lebeck and Wood [1994] performed the four optimizations in this section by hand on three SPEC92 programs and five separate portions of the nasa7 benchmark.

```
/* After */
struct merge {
       int val;
       int key;
};
struct merge merged_array[SIZE];
```

An interesting characteristic of this example is that the proper coding practice of using an array of records would achieve the same benefits as this optimization.

Loop Interchange

Some programs have nested loops that access data in memory in nonsequential order. Simply exchanging the nesting of the loops can make the code access the data in the order it is stored. Like the prior example, this technique reduces misses by improving spatial locality; reordering maximizes use of data in a cache block before it is discarded.

```
/* Before */
for (j = 0; j < 100; j = j+1)
       for (i = 0; i < 5000; i = i+1)
              x[i][j] = 2 * x[i][j];

/* After */
for (i = 0; i < 5000; i = i+1)
       for (j = 0; j < 100; j = j+1)
              x[i][j] = 2 * x[i][j];
```

The original code would skip through memory in strides of 100 words, while the revised version accesses all the words in the cache block before going to the next one. This optimization improves cache performance without affecting the number of instructions executed, unlike the prior example.

Loop Fusion

Some programs have separate sections of code that access the same arrays with the same loops, performing different computations on the common data. By "fusing" the code into a single loop, the data that are fetched into the cache can be used repeatedly before being swapped out. Hence, in contrast to our first two techniques, the target of this optimization is reducing misses via improved temporal locality.

```
/* Before */
for (i = 0; i < N; i = i+1)
       for (j = 0; j < N; j = j+1)
              a[i][j] = 1/b[i][j] * c[i][j];

for (i = 0; i < N; i = i+1)
       for (j = 0; j < N; j = j+1)
              d[i][j] = a[i][j] + c[i][j];

/* After */
for (i = 0; i < N; i = i+1)
       for (j = 0; j < N; j = j+1)
       {
              a[i][j] = 1/b[i][j] * c[i][j];
              d[i][j] = a[i][j] + c[i][j];
       }
```

The original code would take all the misses to access arrays `a` and `c` twice, once in the first loop and then again in the second. In the fused loop, the second statement freeloads on the cache accesses of the first statement.

Blocking

This optimization, perhaps the most famous of the cache optimizations, again tries to reduce misses via improved temporal locality. We are again dealing with multiple arrays, with some arrays accessed by rows and some by columns. Storing the arrays row by row (*row major order*) or column by column (*column major order*) does not solve the problem because both rows and columns are used in every iteration of the loop. Such orthogonal accesses mean the earlier transformations, such as loop interchange, are not helpful.

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. The code example below, which performs matrix multiplication, helps motivate the optimization:

```
/* Before */
for (i = 0; i < N; i = i+1)
       for (j = 0; j < N; j = j+1)
              {r = 0;
               for (k = 0; k < N; k = k+1) {
                     r = r + y[i][k]*z[k][j];};
               x[i][j] = r;
              };
```

The two inner loops read all `N` by `N` elements of `z`, access the same `N` elements in a row of `y` repeatedly, and write one row of `N` elements of `x`. Figure 5.18 gives a

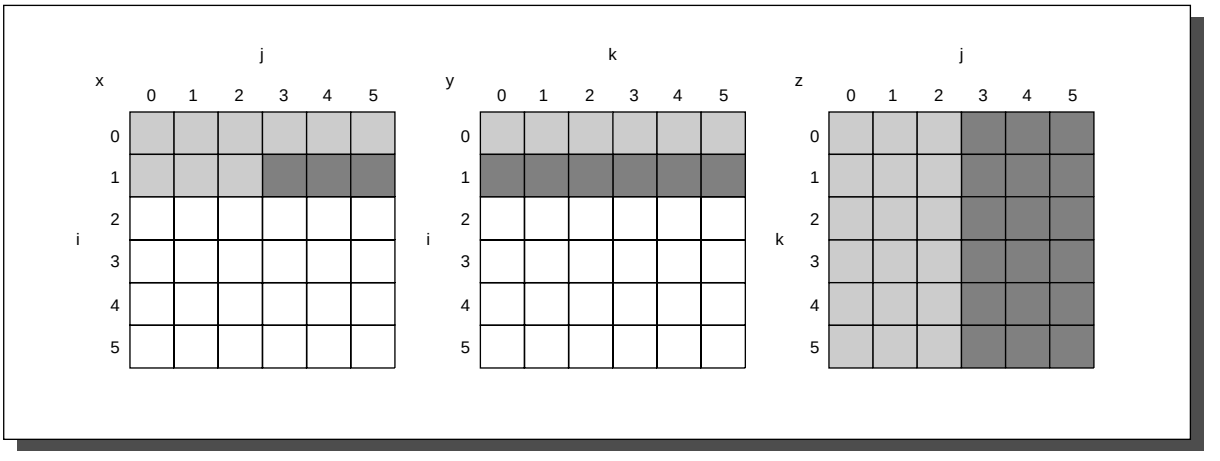

FIGURE 5.18 A snapshot of the three arrays `x`, `y`, and `z` when `i = 1`. The age of accesses to the array elements is indicated by shade: white means not yet touched, light means older accesses and dark means newer accesses. The variables `i`, `j`, and `k` are shown along the rows or columns used to access the arrays.

snapshot of the accesses to the three arrays, with a dark shade indicating a recent access, a light shade indicating an older access, and white meaning not yet accessed.

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. If the cache can hold one N by N matrix and one row of N, then at least the ith row of y and the array z may stay in the cache. Less than that and misses may occur for both x and z. In the worst case, there would be $2N^3 + N^2$ words read from memory 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 of size B by B by having the two inner loops compute in steps of size B rather than going from beginning to end of x and z. B is called the *blocking factor*. (Assume x is initialized to zero.)

```
/* After */
for (jj = 0; jj < N; jj = jj+B)
for (kk = 0; kk < N; kk = kk+B)
for (i = 0; i < N; i = i+1)
        for (j = jj; j < min(jj+B,N); j = j+1)
            {r = 0;
             for (k = kk; k < min(kk+B,N); k = k+1) {
                    r = r + y[i][k]*z[k][j];};
             x[i][j] = x[i][j] + r;
            };
```

Figure 5.19 illustrates the accesses to the three arrays using blocking. Looking only at capacity misses, the total number of memory words accessed is $2N^3/B + N^2$, which is an improvement by about a factor of B. Thus blocking exploits a combination of spatial and temporal locality, since y benefits from spatial locality and z benefits from temporal locality.

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.

Traditionally blocking has been aimed at reducing capacity misses, under the simplifying assumption that conflict misses are either not significant or can be removed by more associative caches. Since blocking reduces the number of words that are active in a cache at a given time, choosing a blocking size smaller than capacity can also reduce conflict misses. Figure 5.20 gives a qualitative view of this trade-off.

These last two subsections have concentrated on the potential benefit of cache-aware compilers and programs. Given that increasing gap in processor speed and memory access times, this benefit will only increase in importance over time.

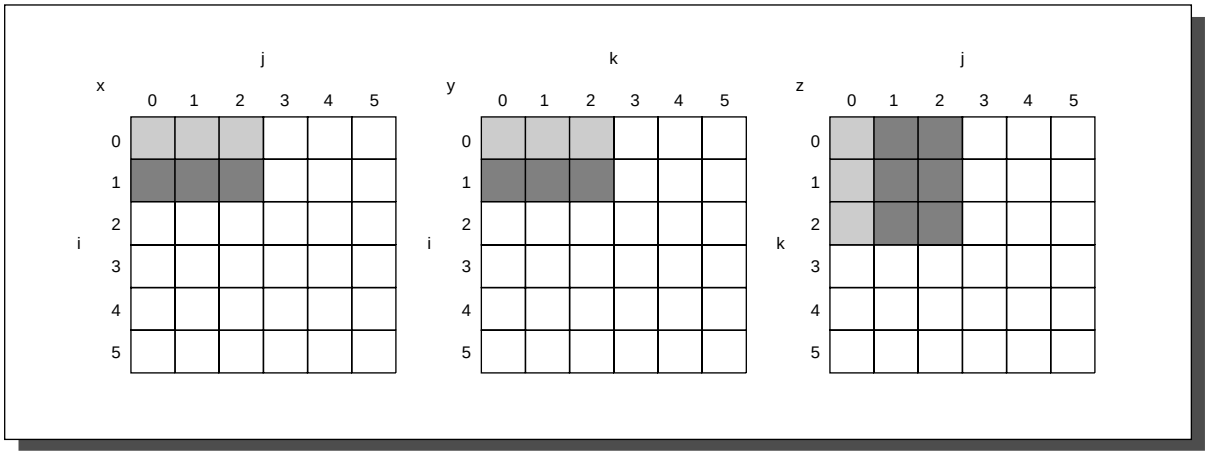

FIGURE 5.19 The age of accesses to the arrays x, y, and z. Note in contrast to Figure 5.18 the smaller number of elements accessed.

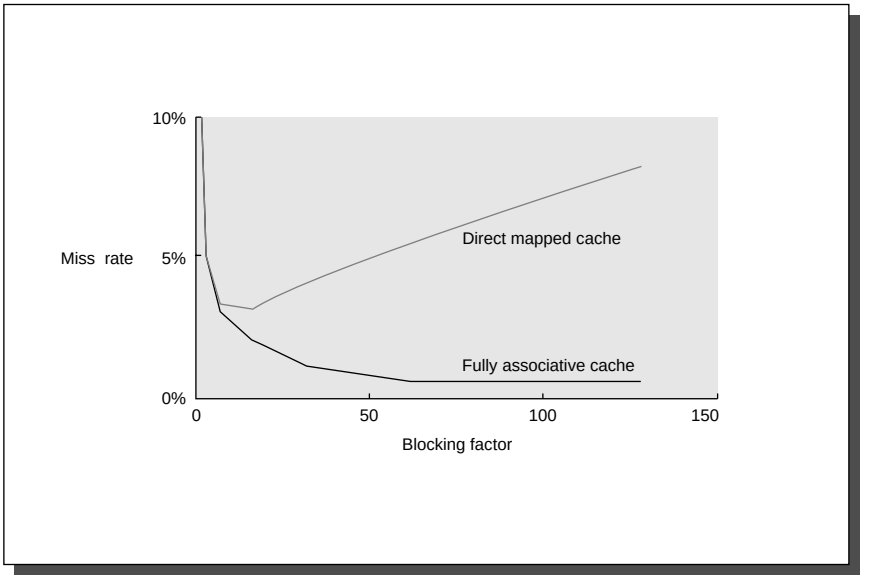

FIGURE 5.20 The impact of conflict misses in caches that aren't fully associative on block size. For example, Lam, Rothberg, and Wolf [1991] found one case where a blocking factor of 24 had a fifth the number of misses of a blocking factor of 48, despite both fitting into the cache.

Now that we have spent more than 20 pages on techniques that reduce cache misses, it is time to look at reducing the next component of average memory access time.

5.4 Reducing Cache Miss Penalty

Reducing cache misses has been the traditional focus of cache research, but the cache performance formula assures us that improvements in miss penalty can be just as beneficial as improvements in miss rate. Moreover, Figure 5.1 shows that technology trends have improved the speed of processors faster than DRAMs, making the relative cost of miss penalties increase over time. We give five optimizations here to address this problem. Perhaps the most interesting optimization is the final one, which adds another level of cache to reduce miss penalty.

First Miss Penalty Reduction Technique: Giving Priority to Read Misses over Writes

With a write-through cache the most important improvement is a write buffer (page 380) of the proper size (see the pitfall on page 470 in section 5.11). Write buffers, however, do complicate memory accesses in that they might hold the updated value of a location needed on a read miss.

EXAMPLE Look at this code sequence:

```
SW 512(R0),R3          ; M[512] ← R3   (cache index 0)
LW R1,1024(R0)         ; R1 ← M[1024]  (cache index 0)
LW R2,512(R0)          ; R2 ← M[512]   (cache index 0)
```

Assume a direct-mapped, write-through cache that maps 512 and 1024 to the same block, and a four-word write buffer. Will the value in R2 always be equal to the value in R3?

ANSWER Using the terminology from Chapter 3, this is a read-after-write data hazard in memory. Let's follow a cache access to see the danger. The data in R3 are placed into the write buffer after the store. The following load uses the same cache index and is therefore a miss. The second load instruction tries to put the value in location 512 into register R2; this also results in a miss. If the write buffer hasn't completed writing to location 512 in memory, the read of location 512 will put the old, wrong value into the cache block, and then into R2. Without proper precautions, R3 would not be equal to R2! ■

The simplest way out of this dilemma is for the read miss to wait until the write buffer is empty. A write buffer of a few words in a write-through cache will almost always have data in the buffer on a miss, thereby increasing the read miss penalty. The designers of the MIPS M/1000 estimated that waiting for a four-word buffer to empty would have increased the average read miss penalty by a factor of 1.5. The alternative is to check the contents of the write buffer on a read miss, and if there are no conflicts and the memory system is available, let the read miss continue.

The cost of writes by the processor in a write-back cache can also be reduced. Suppose a read miss will replace a dirty memory block. Instead of writing the dirty block to memory, and then reading memory, we could copy the dirty block to a buffer, then read memory, and *then* write memory. This way the CPU read, for which the processor is probably waiting, will finish sooner. Similar to the situation above, if a read miss occurs, the processor can either stall until the buffer is empty or check the addresses of the words in the buffer for conflicts.

Second Miss Penalty Reduction Technique: Sub-block Placement for Reduced Miss Penalty

Suppose you are designing a cache that must fit on the chip. You may find that your tags are too large, either because they don't fit on the chip or because they are too slow. A simple solution is to go to large blocks, which reduces tag storage without decreasing the amount of information you can store in the cache. Of course the miss rate will likely improve, but the increase in miss penalty could make the larger blocks a bad decision.

One solution is called *sub-block placement*. A valid bit is added to units smaller than the full block, called *sub-blocks*. Only a single sub-block need be read on a miss. The valid bits specify some parts of the block as valid and some parts as invalid, so a match of the tag doesn't mean the word is necessarily in the cache, as the valid bit for that word must also be on. Figure 5.21 gives an example. Clearly sub-blocks will have a smaller miss penalty than full blocks.

Figure 5.21 shows the reduction in tag storage; if the valid bits had to be replaced by full tags, there would be much more memory dedicated to tags, which is the reason sub-block placement was invented.

Third Miss Penalty Reduction Technique: Early Restart and Critical Word First

The first two techniques require extra hardware to reduce miss penalty, but not this third technique. It is based on the observation that the CPU needs just one word of the block at a time. This strategy is impatience: Don't wait for the full block to be loaded before sending the requested word and restarting the CPU. Here are two specific strategies:

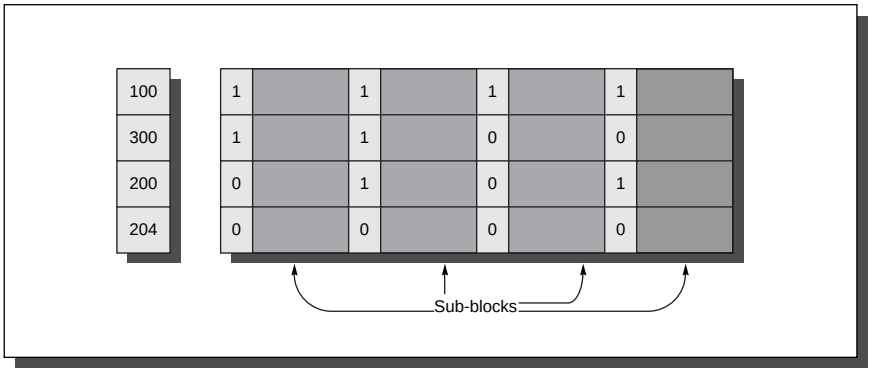

FIGURE 5.21 In this example there are four sub-blocks per block in a direct-mapped cache. Sub-blocks can be thought of as an extra level of addressing beyond the address tag. In the first block (top), all the valid bits are on, equivalent to the valid bit being on for a block in a normal cache. In the last block (bottom), the opposite is true; no valid bits are on. In the second block, locations 300 and 301 are valid and will be hits, while locations 302 and 303 will be misses. For the third block, locations 201 and 203 are hits. If, instead of this organization, there were 16 blocks the size of the sub-block, 16 tags would be needed instead of 4. Note that for caches with sub-block placement, a block can no longer be defined as the minimum unit transferred between cache and memory. For such caches a block is defined as the unit of information associated with an address tag.

- *Early restart*—As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution.
- *Critical word first*—Request the missed word first from memory and send it to the CPU as soon as it arrives; let the CPU continue execution while filling the rest of the words in the block. Critical-word-first fetch is also called *wrapped* fetch and *requested word first*.

Generally these techniques only benefit designs with very large cache blocks, since the benefit is low unless blocks are large.

EXAMPLE Let's assume a machine has a 32-byte cache block and the memory system takes five clock cycles to fetch bytes over a 16-byte wide path to memory, as in the case of the Alpha AXP 21064. Calculate the average miss penalty for critical word first, assuming that there will be no other accesses to the other half of the block until it is completely fetched. Then calculate assuming the following instruction reads data from the other half of the block.

ANSWER The average miss penalty is five clock cycles for critical word first. For back-to-back reads of both halves of the cache block, only one cycle is saved since the pipeline will only move one instruction further until it must stall on the missing data. ■

As this example illustrates, the benefits of critical word first and early restart depend on the size of the block and the likelihood of another access to the portion of the block that has not yet been fetched.

The next technique takes overlap between the CPU and cache miss penalty even further to reduce the average miss penalty.

Fourth Miss Penalty Reduction Technique: Nonblocking Caches to Reduce Stalls on Cache Misses

Early restart still waits for the requested word to arrive before the CPU can continue execution. For pipelined machines that allow out-of-order completion using a scoreboard or Tomasulo-style control (section 4.2 in Chapter 4), the CPU need not stall on a cache miss. For example, the CPU could continue fetching instructions from the instruction cache while waiting for the data cache to return the missing data. A *nonblocking cache* or *lockup-free cache* escalates the potential benefits of such a scheme by allowing the data cache to continue to supply cache hits during a miss. This "hit under miss" optimization reduces the effective miss penalty by being helpful during a miss instead of ignoring the requests of the CPU. A subtle and complex option is that the cache may further lower the effective miss penalty if it can overlap multiple misses: a "hit under multiple miss" or "miss under miss" optimization. The second option is beneficial only if the memory system can service multiple misses (see page 434). Be aware that hit under miss significantly increases the complexity of the cache controller as there can be multiple outstanding memory accesses.

Figure 5.22 shows the average time in clock cycles for cache misses for an 8-KB data cache as the number of outstanding misses is varied. Floating-point programs benefit from increasing complexity, while integer programs get almost all of the benefit from a simple hit-under-one-miss scheme.

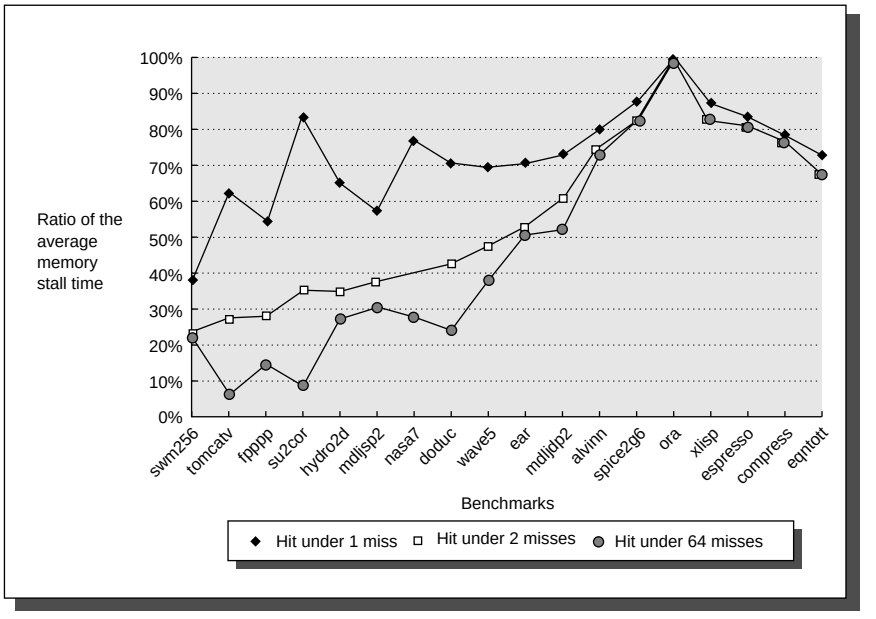

FIGURE 5.22 Ratio of the average memory stall time for a blocking cache to hit-under-miss schemes as the number of outstanding misses is varied for 18 SPEC92 programs. The hit-under-64-misses line allows one miss for every register in the machine. The first 14 programs are floating-point programs: the average for hit under 1 miss is 76%, for 2 misses is 51%, and for 64 misses is 39%. The final four are integer programs, and the three averages are 81%, 78%, and 78%, respectively. These data were collected for an 8-KB direct-mapped data cache with 32-byte blocks and a 16-clock-cycle miss penalty. These data were generated using the VLIW Multiflow Compiler, which scheduled loads away from use [Farkas and Jouppi 1994].

EXAMPLE For the cache described in Figure 5.22, which is more important for floating-point programs: two-way set associativity or hit under one miss? What about for integer programs? Assume the following average miss rates for 8-KB data caches: 11.4% for floating-point programs with a direct-mapped cache, 10.7% for these programs with a two-way set-associative cache, 7.4% for integer programs with a direct-mapped cache, and 6.0% for integer programs with a two-way set-associative cache. Assume the average memory stall time is just the product of the miss rate and the miss penalty.

ANSWER The numbers for Figure 5.22 were based on a miss penalty of 16 clock cycles. Although this is low for a miss penalty, let's stick with it for consistency. For floating-point programs the average memory stall times are

$$\text{Miss rate}_{\text{DM}} \times \text{Miss penalty} = 11.4\% \times 16 = 1.84$$

$$\text{Miss rate}_{\text{2-way}} \times \text{Miss penalty} = 10.7\% \times 16 = 1.71$$

The memory stalls of two-way are thus 1.71/1.84 or 93% of direct-mapped cache. The caption of Figure 5.22 says hit under one miss reduces the average memory stall time to 76% of a blocking cache, so for floating-point programs the direct-mapped data cache supporting hit under one miss gives better performance than a two-way set-associative cache that blocks on a miss.

For integer programs the calculation is

$$\text{Miss rate}_{\text{DM}} \times \text{Miss penalty} = 7.4\% \times 16 = 1.18$$

$$\text{Miss rate}_{\text{2-way}} \times \text{Miss penalty} = 6.0\% \times 16 = 0.96$$

The memory stalls of two-way are thus 0.96/1.18 or 81% of direct-mapped cache. The caption of Figure 5.22 says hit under one miss reduces the average memory stall time to 81% of a blocking cache, so the two options give about the same performance for integer programs. One potential advantage of hit under miss is that it cannot affect the hit time, as associativity can. ■

Fifth Miss Penalty Reduction Technique: Second-Level Caches

The first four techniques to reduce miss penalty have impact on the CPU. This final technique ignores the CPU, concentrating on the interface between the cache and main memory.

The performance gap between processors and memory leads the architect to this question: Should I make the cache faster to keep pace with the speed of CPUs, or make the cache larger to overcome the widening gap between the CPU and main memory? One answer is, both. By adding another level of cache between the original cache and memory, the first-level cache can be small enough to match the clock cycle time of the fast CPU, while the second-level cache can be large enough to capture many accesses that would go to main memory, thereby lessening the effective miss penalty.

While the concept of adding another level in the hierarchy is straightforward, it complicates performance analysis. Definitions for a second level of cache are

not always straightforward. Let's start with the definition of *average memory access time* for a two-level cache. Using the subscripts L1 and L2 to refer, respectively, to a first-level and a second-level cache, the original formula is

$$\text{Average memory access time} = \text{Hit time}_{L1} + \text{Miss rate}_{L1} \times \text{Miss penalty}_{L1}$$

and

$$\text{Miss penalty}_{L1} = \text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2}$$

so

$$\text{Average memory access time} = \text{Hit time}_{L1} + \text{Miss rate}_{L1} \times (\text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2})$$

In this formula, the second-level miss rate is measured on the leftovers from the first-level cache. To avoid ambiguity, these terms are adopted here for a two-level cache system:

- *Local miss rate*—The number of misses in the cache divided by the total number of memory accesses to this cache; this is Miss rate_{L2} above for the second-level cache.
- *Global miss rate*—The number of misses in the cache divided by the total number of memory accesses generated by the CPU; using the terms above, the global miss rate of the second-level cache is $\text{Miss rate}_{L1} \times \text{Miss rate}_{L2}$.

This local miss rate is large because the first-level cache skims the cream of the memory accesses, and this is why the global miss rate is the more useful measure: it indicates what fraction of the memory accesses that leave the CPU go all the way to memory.

EXAMPLE Suppose that in 1000 memory references there are 40 misses in the first-level cache and 20 misses in the second-level cache. What are the various miss rates?

ANSWER The miss rate (either local or global) for the first-level cache is 40/1000 or 4%. The local miss rate for the second-level cache is 20/40 or 50%. The global miss rate of the second-level cache is 20/1000 or 2%. ■

Note that these formulas are for combined reads and writes, assuming a write-back first-level cache. Obviously, a write-through first-level cache will send *all* writes to the second level, not just the misses, and a write buffer would be used.

Figures 5.23 and 5.24 show how miss rates and relative execution time change with the size of a second-level cache for one design. From these figures we can gain two insights. The first is that the global cache miss rate is very similar to the

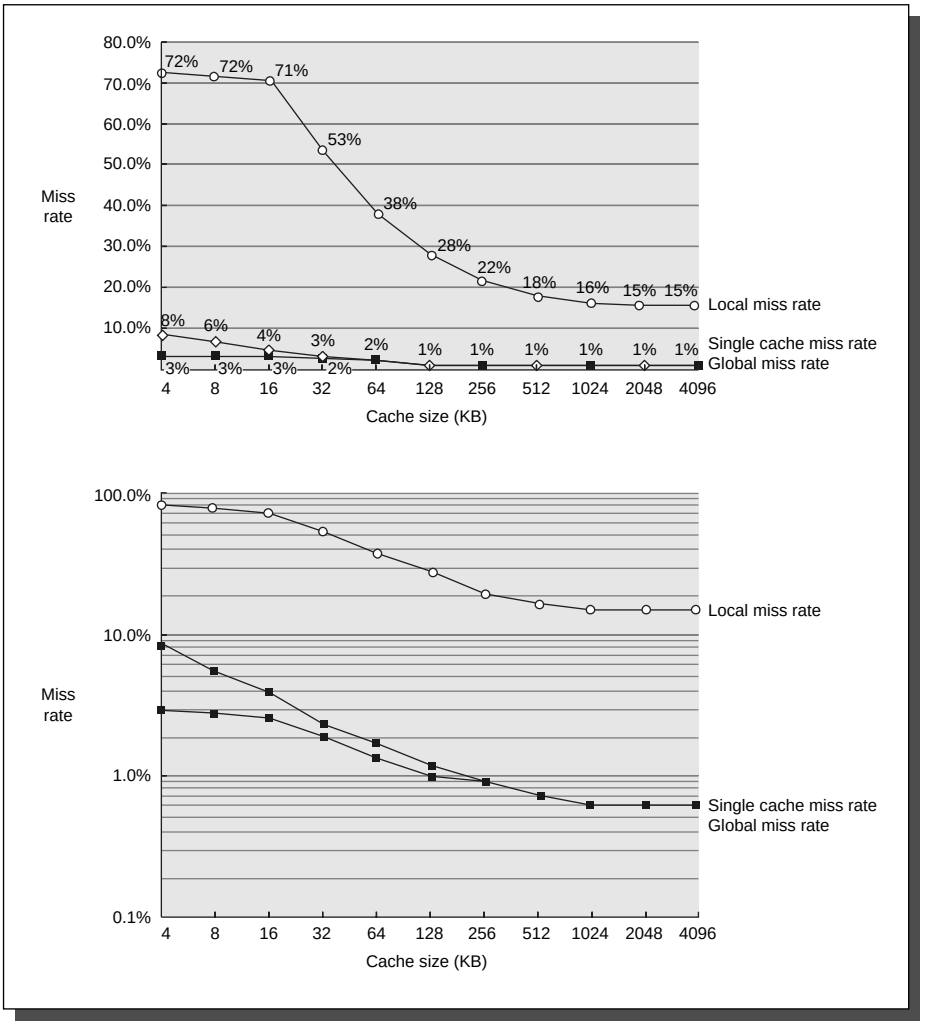

FIGURE 5.23 Miss rates versus cache size for reads and writes. The top graph shows the results plotted on a linear scale as we have done with earlier figures, while the bottom graph shows the results plotted on a log scale. As miss rates shrink, the log scale makes the differences easier to follow. The miss rate of a single-level cache versus size is plotted against the local miss rate and global miss rate of a second-level cache using a 32-KB first-level cache. Second-level caches *smaller* than the 32-KB first level make little sense, as reflected in the high miss rates. After 256 KB the single cache and global miss rates are virtually identical. Przybylski [1990] used four traces from the VAX system and four user programs from the MIPS R2000 that were randomly interleaved to duplicate the effect of process switches.

single cache miss rate of the second-level cache, provided that the second-level cache is much larger than the first-level cache. Hence our intuition and knowledge about the first-level caches apply. The second insight is that the local cache

rate is *not* a good measure of secondary caches; it is a function of the miss rate of the first-level cache, and hence can vary by changing the first-level cache. Thus, the global cache miss rate should be used when evaluating second-level caches.

With these definitions in place, we can consider the parameters of second-level caches. The foremost difference between the two levels is that the speed of the first-level cache affects the clock rate of the CPU, while the speed of the second-level cache only affects the miss penalty of the first-level cache. Thus, we can consider many alternatives in the second-level cache that would be ill chosen for the first-level cache. There are but two questions for the design of the second-level cache: Will it lower the average memory access time portion of the CPI, and how much does it cost?

The initial decision is the size of a second-level cache. Since everything in the first-level cache is likely to be in the second-level cache, the second-level cache should be much bigger than the first. If second-level caches are just a little bigger, the local miss rate will be high. This observation inspires design of huge second-level caches—the size of main memory in older computers! Large size means that the second-level cache may have practically no capacity misses, leaving a few compulsory and conflict misses for our attention. One question is whether set associativity makes more sense for second-level caches.

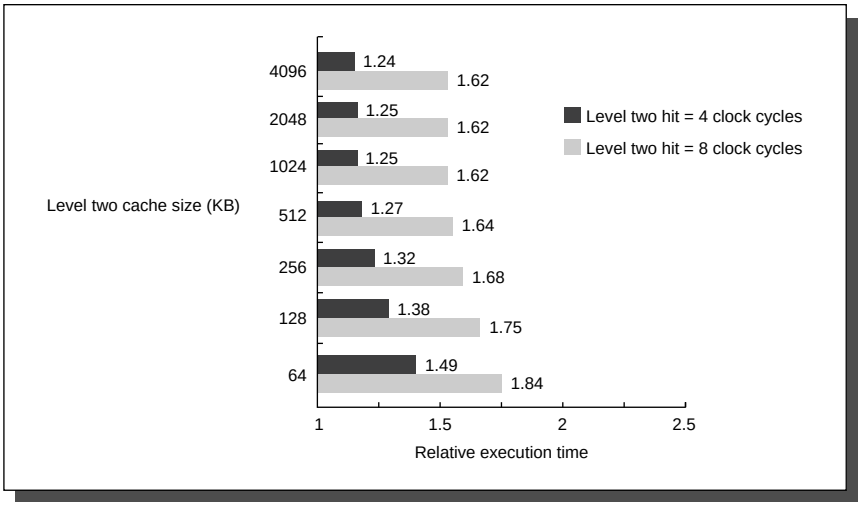

FIGURE 5.24 Relative execution time by second-level cache size. Przybylski [1990] collected these data using a 32-KB first-level write-back cache, varying the size of the second-level cache. The two bars are for different clock cycles for a level two cache hit. The reference execution time of 1.00 is for a 4096-KB second-level cache with a one-clock-cycle latency on a second-level hit. These data were collected the same way as in Figure 5.23.

EXAMPLE Given the data below, what is the impact of second-level cache associativity on the miss penalty?

- Two-way set associativity increases hit time by 10% of a CPU clock cycle
- Hit time$_{L2}$ for direct mapped = 10 clock cycles
- Local miss rate$_{L2}$ for direct mapped = 25%
- Local miss rate$_{L2}$ for two-way set associative = 20%
- Miss penalty$_{L2}$ = 50 clock cycles

ANSWER For a direct-mapped second-level cache, the first-level cache miss penalty is

$$\text{Miss penalty}_{1\text{- way L2}} = 10 + 25\% \times 50 = 22.5 \text{ clock cycles}$$

Adding the cost of associativity increases the hit cost only 0.1 clock cycles, making the new first-level cache miss penalty

$$\text{Miss penalty}_{2\text{- way L2}} = 10.1 + 20\% \times 50 = 20.1 \text{ clock cycles}$$

In reality, second-level caches are almost always synchronized with the first-level cache and CPU. Accordingly, the second-level hit time must be an integral number of clock cycles. If we are lucky, we can shave the second-level hit time to 10 cycles; if not, we can round up to 11 cycles. Either choice is an improvement over the direct-mapped second-level cache:

$$\text{Miss penalty}_{2\text{- way L2}} = 10 + 20\% \times 50 = 20.0 \text{ clock cycles}$$

$$\text{Miss penalty}_{2\text{- way L2}} = 11 + 20\% \times 50 = 21.0 \text{ clock cycles}$$

■

Now we can reduce the miss penalty by reducing the miss rate of the second-level caches using techniques from section 5.3. Higher associativity or pseudo-associativity (page 398) are worth considering because they have small impact on the second-level hit time and because so much of the average access time is due to misses in the second-level cache. Although the larger size of the second-level cache eliminates conflict misses by distributing data over more blocks, it also eliminates most of the capacity misses; thus the *percentage* of conflict misses is still significant in direct-mapped second-level caches.

Another approach to reducing misses is increasing block size in second-level caches. Increasing block size can increase conflict misses with small caches since there may not be enough places to put data, therefore increasing miss rate. Because this is not an issue in large second-level caches, and because memory

access time is relatively longer, block sizes of 64 bytes, 128 bytes, and even occasionally 256 bytes are popular. Figure 5.25 shows the variation in execution time as the second-level block size changes for a relatively narrow memory bus of 32 bits.

Another consideration concerns whether all data in the first-level cache are always in the second-level cache. If so, the second-level cache is said to have the *multilevel inclusion property.* Inclusion is desirable because consistency between I/O and caches (or between caches in a multiprocessor) can be determined just by checking the second-level cache (see section 8.7).

The drawback to this natural inclusion is that the lower average memory access times can suggest smaller blocks for the smaller first-level cache and larger blocks for the larger second-level cache. Inclusion can still be maintained with more work on a second-level miss: The second-level cache must invalidate all first-level blocks that map onto the second-level block to be replaced, causing a slightly higher first-level miss rate. It can also cause unneeded cache invalidates. Inclusion escalates in complexity when combined with performance optimizations, such as a nonblocking secondary cache.

Finally, although a novice might design the first- and second-level caches independently, the designer of the first-level cache has a simpler job given a second-level cache to back up the first. It is less of a gamble to use a write through, for example, if there is a write-back cache at the next level to act as a backstop for repeated writes.

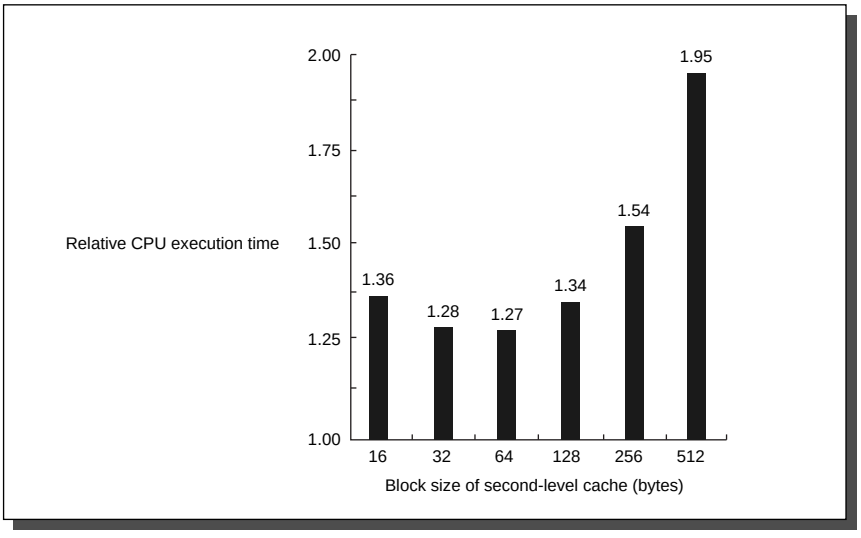

FIGURE 5.25 Relative execution time by block size for a two-level cache. Przybylski [1990] collected these data using a 512-KB second-level cache. These data were collected the same way as in Figure 5.23. The path to memory was basically 32 bits wide in this study: one clock cycle to send the address, six clock cycles to access the data, and one word per clock cycle to transfer the data.

Summarizing the second-level cache considerations, the essence of cache design is balancing fast hits and few misses. Most optimizations that help one hurt the other. For second-level caches, there are many fewer hits than in the first-level cache, so the emphasis shifts to fewer misses. This insight leads to larger caches with higher associativity and larger blocks.

5.5 Reducing Hit Time

Now that we have examined ways to improve cache performance by reducing misses (in section 5.3) and by reducing miss penalty (in section 5.4), we are ready to reduce the third component of the average memory access time.

Hit time is critical because it affects the clock rate of the processor; on many machines today the cache access time limits the clock cycle rate, even for machines that take multiple clock cycles to access the cache. Hence a fast hit time is multiplied in importance beyond the average memory access time formula because it helps everything. This section gives two general techniques and then one optimization for write hits.

First Hit Time Reduction Technique: Small and Simple Caches

A time-consuming portion of a cache hit is using the index portion of the address to read the tag memory and then compare it to the address. Our guideline from Chapter 1 suggests that smaller hardware is faster, and a small cache certainly helps the hit time. It is also critical to keep the cache small enough to fit on the same chip as the processor to avoid the time penalty of going off-chip. Some designs strike a compromise by keeping the tags on-chip and the data off-chip, promising a fast tag check, yet providing the greater capacity of separate memory chips. The second suggestion is to keep the cache simple, such as using direct mapping (see page 396). A main benefit of direct-mapped caches is that the designer can overlap the tag check with the transmission of the data. This effectively reduces hit time. Hence the pressure of a fast clock cycle encourages small and simple cache designs for first-level caches.

Second Hit Time Reduction Technique: Avoiding Address Translation During Indexing of the Cache

Even a small and simple cache must cope with the translation of a virtual address from the CPU to a physical address to access memory. As described below in section 5.7, processors treat main memory as just another level of the memory hierarchy, and thus the address of the virtual memory that exists on disk must be mapped onto the main memory.

The guideline of making the common case fast suggests that we use virtual addresses for the cache, since hits are much more common than misses. Such caches are termed *virtual caches*, with *physical cache* used to identify the traditional cache that uses physical addresses. Virtual addressing eliminates address translation time from a cache hit. Then why doesn't everyone build virtually addressed caches? One reason is that every time a process is switched, the virtual addresses refer to different physical addresses, requiring the cache to be flushed. Figure 5.26 shows the impact on miss rates of this flushing. One solution is to

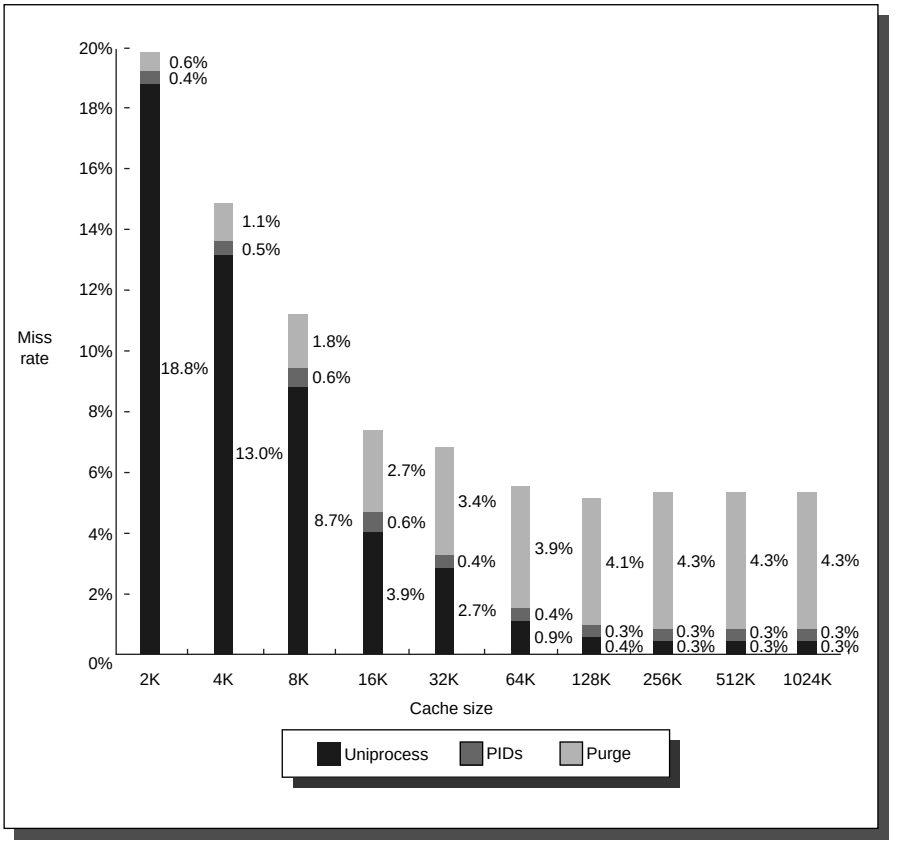

FIGURE 5.26 Miss rate versus virtually addressed cache size of a program measured three ways: without process switches (uniprocess), with process switches using a process-identifier tag (PIDs), and with process switches but without PIDs (purge). PIDs increase the uniprocess absolute miss rate by 0.3% to 0.6% and save 0.6% to 4.3% over purging. Agarwal [1987] collected these statistics for the Ultrix operating system running on a VAX, assuming direct-mapped caches with a block size of 16 bytes. Note that the miss rate goes up from 128K to 256K. Such nonintuitive behavior can occur in caches because changing size changes the mapping of memory blocks onto cache blocks, which can change the conflict miss rate.

increase the width of the cache address tag with a *process-identifier tag* (PID). If the operating system assigns these tags to processes, it only need flush the cache when a PID is recycled; that is, the PID distinguishes whether or not the data in the cache are for this program. Figure 5.26 shows the improvement in miss rates by using PIDs to avoid cache flushes.

Another reason why virtual caches are not more popular is that operating systems and user programs may use two different virtual addresses for the same physical address. These duplicate addresses, called *synonyms* or *aliases*, could result in two copies of the same data in a virtual cache; if one is modified, the other will have the wrong value. With a physical cache this wouldn't happen, since the accesses would first be translated to the same physical cache block. Hardware solutions, called *anti-aliasing*, guarantee every cache block a unique physical address.

Software can make this problem much easier by forcing aliases to share some address bits. The version of UNIX from Sun Microsystems, for example, requires all aliases to be identical in the last 18 bits of their addresses; this restriction is called *page coloring*. Note that page coloring is simply set-associative mapping applied to virtual memory: the 4-KB (2^{12}) pages are mapped using 64 (2^6) sets to ensure that the physical and virtual addresses match in the last 18 bits. This restriction means a direct-mapped cache that is 2^{18} (256K) bytes or smaller can never have duplicate physical addresses for blocks.

The final area of concern with virtual addresses is I/O. I/O typically uses physical addresses and thus would require mapping to virtual addresses to interact with a virtual cache. (The impact of I/O on caches is further discussed below in section 5.9.)

Another technique to get fast hits is to break address translation and cache access into separate pipeline stages, giving fast cycle time and slow hits. This increases the number of pipeline stages for a memory access, leading to greater penalty on mispredicted branches and more clock cycles between the issue of the load and the use of the data (see section 3.9).

One alternative to get the best of both virtual and physical caches is to use the page offset—the part unaffected by address translation—to index the cache while sending the virtual part to be translated. This alternative allows the comparison to be with physical addresses and yet overlap the time to read the tags with address translation. The limitation of this virtually indexed, physically tagged alternative is that a direct-mapped cache can be no bigger than the page size. This is an advantage of the 8-KB caches of the Alpha AXP 21064; the minimum page size is 8 KB, so the 8-bit index can be taken from the physical part of the address.

One way to keep the index small enough to be taken from the physical part of the address and still have a large cache is to use high associativity. Recall that the size of the index is controlled by this formula:

$$2^{\text{index}} = \frac{\text{Cache size}}{\text{Block size} \times \text{Set associativity}}$$

The IBM 3033 cache, as an extreme example, is 16-way set associative, even though studies show there is little benefit to miss rates above eight-way set associativity. This high associativity allows a 64-KB cache to be addressed with a physical index despite the limitation of 4-KB pages in the IBM architecture. Figure 5.27 shows the relationship of index to page offset.

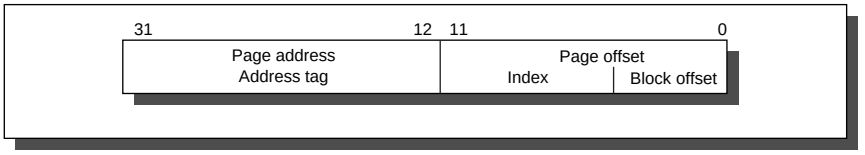

FIGURE 5.27 Relationship of index field and page offset in the IBM 3033 cache. The 4-KB page means the last 12 bits of the address are not translated, and hence some of it can be used to index the cache.

One alternative to higher associativity is for the operating system to implement page coloring by guaranteeing that the last few bits of the virtual and physical page address are identical. Such cooperating allows a larger index than first with the page offset and still compares physical addresses.

Another alternative to higher associativity is to have a small piece of hardware that guesses the mapping of the last few bits of virtual address bits to physical address. This might be a small table that uses a hashing function on the virtual address. This guess is used with the physical portion of the address to index the cache, with the translated address used to match the tag selected by this hybrid index. If the tag matches, we have a hit. If the tag doesn't match, either the data were not in the cache or we had a bad guess of the mapping of the last few bits of virtual address. The cache would presumably retry with the correct index to decide whether the access was a hit or a real miss.

Keeping caches small and simple and techniques to avoid delays of address translation will make both read hits and write hits faster. The next subsection concentrates only on writes.

Third Hit Time Reduction Technique: Pipelining Writes for Fast Write Hits

Write hits usually take longer than read hits because the tag must be checked before writing the data; otherwise the wrong address would be written. One technique, used by the Alpha AXP 21064 and other machines, pipelines the writes. Figure 5.28 shows the hardware organization of pipelined writes. First, tags and data are split so that they can be addressed independently. On a write, the cache compares the tag with the current write address, as usual. The difference comes

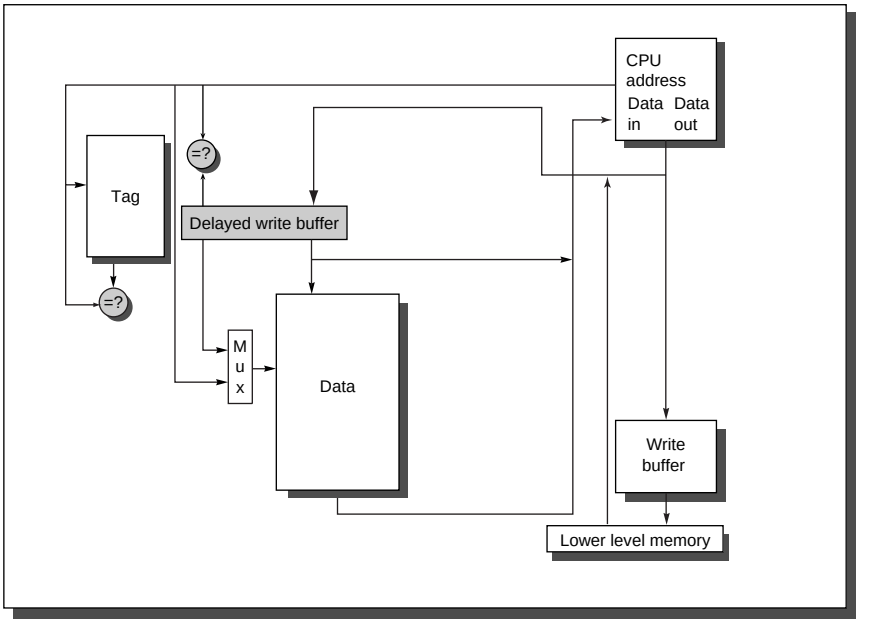

FIGURE 5.28 The hardware organization of pipelined writes. It is possible to find the desired data in the delayed write buffer. In that case, either the write buffer supplies the newer data or the write buffer could complete and then the new data are read from the cache.

with the write to the data portion of the cache that occurs during the tag comparison; it must be using some other address since the current write address is still being checked. The trick is that the cache uses the address and data from the *previous* write, which has already been determined to be a hit. Thus the logical pipeline is between writes—the second stage of the write occurs during the first stage of the next write (or during a cache miss). Therefore, writes can be performed back to back at one per clock cycle because the CPU does not have to wait for the tag check before writing. Reads play no part in this pipeline since they already operate in parallel with the tag check, and so no help is needed.

Cache Optimization Summary

The techniques in sections 5.3 to 5.5 to improve miss rate, miss penalty, and hit time generally impact the other components of the average memory access equation as well as the complexity of the memory hierarchy. Figure 5.29 summarizes these techniques and estimates the impact on complexity, with + meaning that the technique improves the factor, – meaning it hurts that factor, and blank meaning it has no impact. Note that few techniques help more than one category, and none help all three.

Technique	Miss rate	Miss penalty	Hit time	Hardware complexity	Comment
Larger block size	+	–		0	Trivial; RS/6000 550 uses 128
Higher associativity	+		–	1	e.g., MIPS R10000 is 4-way
Victim caches	+			2	Similar technique in HP 7200
Pseudo-associative caches	+			2	Used in L2 of MIPS R10000
Hardware prefetching of instructions and data	+			2	Data are harder to prefetch; tried in a few machines; Alpha 21064
Compiler-controlled prefetching	+			3	Needs nonblocking cache too; several machines support it
Compiler techniques to reduce cache misses	+			0	Software is challenge; some machines give compiler option
Giving priority to read misses over writes		+		1	Trivial for uniprocessor, and widely used
Subblock placement		+		1	Used primarily to reduce tags
Early restart and critical word first		+		2	Used in MIPS R10000, IBM 620
Nonblocking caches		+		3	Used in Alpha 21064, R10000
Second-level caches		+		2	Costly hardware; harder if block size L1 ≠ L2; widely used
Small and simple caches	–		+	0	Trivial; widely used
Avoiding address translation during indexing of the cache			+	2	Trivial if small cache; used in Alpha 21064
Pipelining writes for fast write hits			+	1	Used in Alpha 21064

FIGURE 5.29 Summary of cache optimizations and impact on the three aspects of cache performance and on cache complexity. + means that the technique improves the factor, – means it hurts that factor, and blank means it has no impact. The complexity measure is subjective, with 0 being the easiest and 3 being a challenge.

5.6 Main Memory

… the one single development that put computers on their feet was the invention of a reliable form of memory, namely, the core memory. … Its cost was reasonable, it was reliable and, because it was reliable, it could in due course be made large. [p. 209]

Maurice Wilkes, *Memoirs of a Computer Pioneer* (1985)

Main memory is the next level down in the hierarchy. Main memory satisfies the demands of caches and serves as the I/O interface, as it is the destination of input as well as the source for output. Performance measures of main memory emphasize both latency and bandwidth. (Memory bandwidth is the number of bytes read or written per unit time.) Traditionally, main memory latency (which affects the cache miss penalty) is the primary concern of the cache, while main memory bandwidth is the primary concern of I/O. With the popularity of second-level caches and their larger block sizes, main memory bandwidth becomes important to caches as well. In fact, cache designers may take advantage of the high memory bandwidth by increasing block size. The relationship of main memory and I/O is discussed in Chapter 6.

Memory Technology

Memory latency is traditionally quoted using two measures—access time and cycle time. *Access time* is the time between when a read is requested and when the desired word arrives, while *cycle time* is the minimum time between requests to memory. One reason that cycle time is greater than access time is that the memory needs the address lines to be stable between accesses.

As early DRAMs grew in capacity, the cost of a package with all the necessary address lines was an issue. The solution was to multiplex the address lines, thereby cutting the number of address pins in half. One half of the address is sent first, called the *row access strobe* or *RAS*. It is followed by the other half of the address, sent during the *column access strobe* or *CAS*. These names come from the internal chip organization, since the memory is organized as a rectangular matrix addressed by rows and columns.

An additional requirement of DRAM derives from the property signified by its first letter, *D*, for *dynamic*. DRAMs use only a single transistor to store a bit, but reading that bit can disturb the information. To prevent loss of information, each bit must be "refreshed" periodically. Fortunately, all the bits in a row can be refreshed simultaneously just by reading that row. Hence every DRAM in the memory system must access every row within a certain time window, such as 8 milliseconds. Memory controllers include hardware to periodically refresh the DRAMs.

This requirement means that the memory system is occasionally unavailable because it is sending a signal telling every chip to *refresh*. The time for a refresh is typically a full memory access (RAS and CAS) for each row of the DRAM. Since the memory matrix in a DRAM is conceptually square, the number of steps in a refresh is usually the square root of the DRAM capacity. DRAM designers try to keep time spent refreshing to be less than 5% of the total time.

In contrast to DRAMs are SRAMs—the first letter standing for *static*. The dynamic nature of the circuits in DRAM require data to be written back after being read, hence the difference between the access time and the cycle time as well as the need to refresh. SRAMs use four to six transistors per bit to prevent the information from being disturbed when read. Thus, unlike DRAMs, there is no difference between access time and cycle time, and there is no need to refresh SRAM. In DRAM designs the emphasis is on capacity, while SRAM designs are

concerned with both speed *and* capacity. (Because of this concern, SRAM address lines are not multiplexed.) For memories designed in comparable technologies, the capacity of DRAMs is roughly 4 to 8 times that of SRAMs. The cycle time of SRAMs is 8 to 16 times faster than DRAMs, but they are also 8 to 16 times as expensive.

The main memory of virtually every computer sold since 1975 is composed of semiconductor DRAMs (and virtually all caches use SRAM); the exception that proves the rule is Cray supercomputers such as the C-90, which use SRAM for main memory.

Amdahl suggested a rule of thumb that memory capacity should grow linearly with CPU speed to keep a balanced system (see section 1.4), and CPU designers rely on DRAMs to supply that demand: they expect a four-fold improvement in capacity every three years in the base technology, or 60% per year. Unfortunately, the performance of DRAMs is growing at a much slower rate. Figure 5.30 shows a performance improvement in row access time of about 22% per generation, or 7% per year.

		Row access strobe (RAS)			
Year of introduction	**Chip size**	**Slowest DRAM**	**Fastest DRAM**	**Column access strobe (CAS)**	**Cycle time**
1980	64 Kbit	180 ns	150 ns	75 ns	250 ns
1983	256 Kbit	150 ns	120 ns	50 ns	220 ns
1986	1 Mbit	120 ns	100 ns	25 ns	190 ns
1989	4 Mbit	100 ns	80 ns	20 ns	165 ns
1992	16 Mbit	80 ns	60 ns	15 ns	120 ns
1995	64 Mbit	65 ns	50 ns	10 ns	90 ns

FIGURE 5.30 Times of fast and slow DRAMs with each generation. The improvement by a factor of two in column access accompanied the switch from NMOS DRAMs to CMOS DRAMs. With three years per generation, the performance improvement of row access time is about 7% per year. Data in the last row represent predicted performance for 64-Mbit DRAMs.

As we saw in Figure 5.1 on page 374, the CPU-DRAM performance gap is clearly a problem today—Amdahl's Law warns us what will happen if we ignore one portion of the computation while trying to speed up the rest. The previous sections describe what can be done with cache organization to reduce this performance gap, but simply making caches larger or adding more levels of caches may not be a cost-effective way to eliminate the gap. Innovative organizations of main memory are needed as well. In the next section we examine techniques for organizing memory to improve bandwidth, concluding with techniques especially for DRAMs.

Organizations for Improving Main Memory Performance

Although caches are interested in low latency memory, it is generally easier to improve memory bandwidth with new organizations than it is to reduce latency. Caches benefit from bandwidth improvement by allowing each cache block size to increase without a large increase in the miss penalty.

Let's illustrate these organizations with the case of satisfying a cache miss. Assume the performance of the basic memory organization is

- 4 clock cycles to send the address
- 24 clock cycles for the access time per word
- 4 clock cycles to send a word of data

Given a cache block of four words, the miss penalty is $4 \times (4 + 24 + 4)$ or 128 clock cycles, with a memory bandwidth of one-eighth byte (16/128) per clock cycle.

Figure 5.31 shows some of the options to faster memory systems. The next four solutions assume generic memory, either DRAM or SRAM. DRAM-specific solutions form the last subsection.

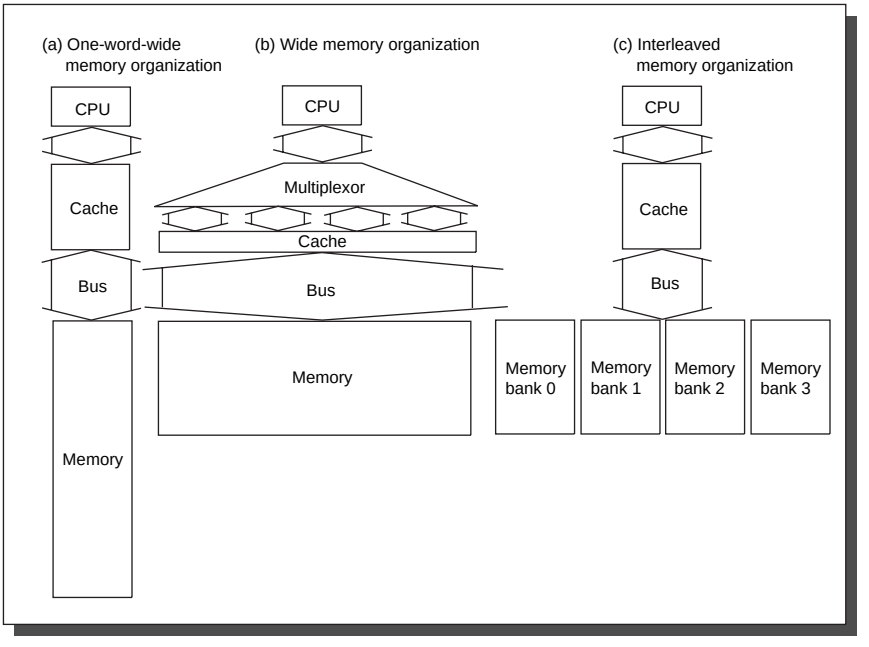

FIGURE 5.31 Three examples of bus width, memory width, and memory interleaving to achieve higher memory bandwidth. (a) is the simplest design, with everything the width of one word; (b) shows a wider memory, bus, and cache; while (c) shows a narrow bus and cache with an interleaved memory.

The simplest approach to increasing memory bandwidth, then, is to make the memory wider; we examine this first.

First Technique for Higher Bandwidth: Wider Main Memory

First-level caches are often organized with a physical width of one word because most CPU accesses are that size. Systems without second-level caches often design main memory to match the width of the cache. Doubling or quadrupling the width of the cache and the memory will therefore double or quadruple the memory bandwidth. With a main memory width of two words, the miss penalty in our example would drop from 4×32 or 128 clock cycles to 2×32 or 64 clock cycles. At four words wide the miss penalty is just 1×32 clock cycles. The bandwidth is then one-quarter byte per clock cycle at two words wide and one-half byte per clock cycle when the memory is four words wide.

There is cost in the wider connection between the CPU and memory, typically called a memory *bus*. CPUs will still access the cache a word at a time, so there now needs to be a multiplexer between the cache and the CPU—and that multiplexer may be on the critical timing path. Second-level caches can help since the multiplexing can be between first- and second-level caches, not on the critical path. Another drawback is that since main memory is traditionally expandable by the customer, the minimum increment is doubled or quadrupled when the width is doubled or quadrupled. Finally, memories with error correction have difficulties with writes to a portion of the protected block (e.g., a write of a byte); the rest of the data must be read so that the new error correction code can be calculated and stored when the data are written. If the error correction is done over the full width, the wider memory will increase the frequency of such "read-modify-write" sequences because more writes become partial block writes. Many designs of wider memory have separate error correction every 32 bits since most writes are that size.

One example of wide main memory is the Alpha AXP 21064 whose second-level cache, memory bus, and memory are all 256 bits wide. To allow customers to purchase small amounts of memory without sacrificing width, DEC sells older generations of DRAM for small memories as well as current DRAMs for the larger memory systems (see section 5.10).

Second Technique for Higher Bandwidth: Simple Interleaved Memory

Increasing width is one way to improve bandwidth, but another is to take advantage of the potential parallelism of having many DRAMs in a memory system. Memory chips can be organized in banks to read or write multiple words at a time rather than a single word. In general, the purpose of interleaved memory is to try to take advantage of the potential memory bandwidth of *all* the DRAMs in the

system; in contrast, most memory systems activate only the DRAMs containing the needed words.

The banks are often one word wide so that the width of the bus and the cache need not change, but sending addresses to several banks permits them all to read simultaneously. Figure 5.31(c) shows this organization. For example, sending an address to four banks (with access times shown on page 430) yields a miss penalty of $4 + 24 + 4 \times 4$ or 44 clock cycles, giving a bandwidth of about 0.4 bytes per clock cycle. Banks are also valuable on writes. Although back-to-back writes would normally have to wait for earlier writes to finish, banks allow one clock cycle for each write, provided the writes are not destined to the same bank. Such a memory organization is especially important for write through.

The mapping of addresses to banks affects the behavior of the memory system. The example above assumes the addresses of the four banks are interleaved at the word level—bank 0 has all words whose address modulo 4 is 0, bank 1 has all words whose address modulo 4 is 1, and so on. Figure 5.32 shows this interleaving. This mapping is referred to as the *interleaving factor*; *interleaved memory* normally means banks of memory that are word interleaved. This interleaving optimizes sequential memory accesses. A cache read miss is an ideal match to word-interleaved memory, as the words in a block are read sequentially. Write-back caches make writes as well as reads sequential, getting even more efficiency from word-interleaved memory.

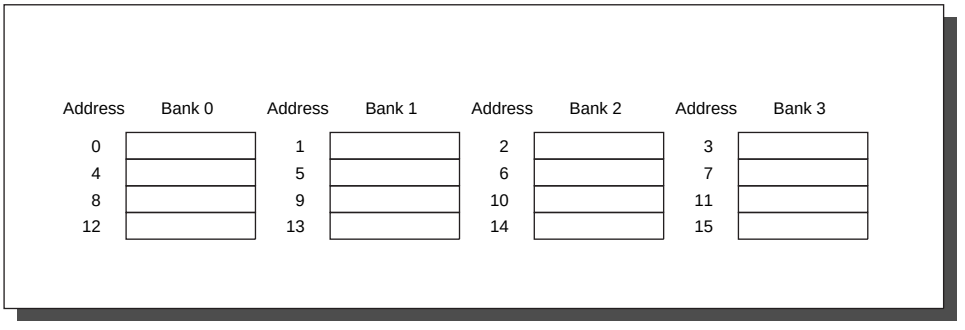

FIGURE 5.32 Four-way interleaved memory. This example assumes word addressing: with byte addressing and four bytes per word, each of these addresses would be multiplied by four.

EXAMPLE What can interleaving and a wide memory buy? Consider the following description of a machine and its cache performance:

Block size = 1 word

Memory bus width = 1 word

Miss rate = 3%

Memory accesses per instruction = 1.2

Cache miss penalty = 32 cycles (as above)

Average cycles per instruction (ignoring cache misses) = 2

If we change the block size to two words, the miss rate falls to 2%, and a four-word block has a miss rate of 1%. What is the improvement in performance of interleaving two ways and four ways versus doubling the width of memory and the bus, assuming the access times on page 430?

ANSWER The CPI for the base machine using one-word blocks is

$$2 + (1.2 \times 3\% \times 32) = 3.15$$

Since the clock cycle time and instruction count won't change in this example, we can calculate performance improvement by just comparing CPI.

Increasing the block size to two words gives the following options:

32-bit bus and memory, no interleaving = $2 + (1.2 \times 2\% \times 2 \times 32) = 3.54$

32-bit bus and memory, interleaving = $2 + (1.2 \times 2\% \times (4 + 24 + 8)) = 2.86$

64-bit bus and memory, no interleaving = $2 + (1.2 \times 2\% \times 1 \times 32) = 2.77$

Thus, doubling the block size slows down the straightforward implementation (3.54 versus 3.15), while interleaving or wider memory is 1.10 or 1.14 times faster, respectively. If we increase the block size to four, the following is obtained:

32-bit bus and memory, no interleaving = $2 + (1.2 \times 1\% \times 4 \times 32) = 3.54$

32-bit bus and memory, interleaving = $2 + (1.2 \times 1\% \times (4 + 24 + 16)) = 2.53$

64-bit bus and memory, no interleaving = $2 + (1.2 \times 1\% \times 2 \times 32) = 2.77$

Again, the larger block hurts performance for the simple case, although the interleaved 32-bit memory is now fastest—1.25 times faster versus 1.14 for the wider memory and bus. ■

This subsection has shown that interleaved memory is logically a wide memory, except that accesses to banks are staged over time to share internal resources—the bus in this example.

How many banks should be included? One metric, used in vector computers (Appendix B), is as follows:

Number of banks ≥ Number of clock cycles to access word in bank

The memory system goal is to deliver information from a new bank each clock cycle for sequential accesses. To see why this formula holds, imagine there were fewer banks than clock cycles to access a word in a bank; say, 8 banks with an access time of 10 clock cycles. After 10 clock cycles the CPU could get a word from bank 0, and then bank 0 would begin fetching the next desired word as the CPU received the following 7 words from the other 7 banks. At clock cycle 18 the CPU would be at the door of bank 0, waiting for it to supply the next word. The CPU would have to wait until clock cycle 20 for the word to appear. Hence we want more banks than clock cycles to access a bank to avoid waiting.

We will discuss conflicts on nonsequential accesses to banks in the following subsections. For now, we note that having many banks reduces the chance of these bank conflicts.

Ironically, as capacity per memory chip increases, there are fewer chips in the same-sized memory system, making multiple banks much more expensive. For example, a 64-MB main memory takes 512 memory chips of 1 M × 1 bit, easily organized into 16 banks of 32 memory chips. But it takes only eight 64-M × 1-bit memory chips for 64 MB, making one bank the limit. Even though the Amdahl/Case rule of thumb for balanced computer systems recommends increasing memory capacity with increasing CPU performance, many manufacturers will want to have a small memory option in the baseline model. This shrinking number of DRAMs is the main disadvantage of interleaved memory banks. DRAMs organized with wider paths, such as 16 M × 4 bits or 8 M × 8 bits, will postpone this weakness.

A second disadvantage of memory banks is again the difficulty of main memory expansion. Either the memory system must support multiple generations of DRAM, as in the DEC 3000 model 800, or the minimum increment will be to, say, double main memory.

Third Technique for Higher Bandwidth: Independent Memory Banks

The original motivation for memory banks was higher memory bandwidth by interleaving sequential accesses. This hardware is not much more difficult since the banks can share address lines with a memory controller, enabling each bank to use the data portion of the memory bus. A generalization of interleaving is to allow multiple independent accesses, where multiple memory controllers allow banks (or sets of word-interleaved banks) to operate independently. Each bank needs separate address lines and possibly a separate data bus. For example, an input device may use one controller and one bank, the cache read may use another, and a cache write may use a third. Nonblocking caches (page 414) allow the CPU to proceed beyond a cache miss, potentially allowing multiple cache misses to be serviced simultaneously. Such a design only makes sense with memory banks; otherwise the multiple reads will be serviced by a single memory port and get only a small benefit of overlapping access with transmission. Multiprocessors

that share a common memory provide further motivation for memory banks (see Chapter 8).

Thus the term *memory bank* has potentially two conflicting definitions. We use the term *superbank* to mean all memory active on one block transfer and the term *bank* for the portion within a superbank that is word interleaved. Figure 5.33 shows this relationship. If there is no confusion, we'll just use the shorter term *bank* to mean a collection of memory.

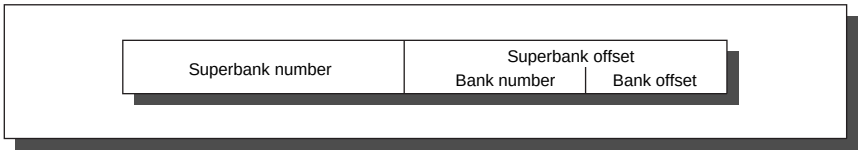

FIGURE 5.33 The relationship of superbanks and banks.

Fourth Technique for Higher Bandwidth: Avoiding Memory Bank Conflicts

If the memory system is being designed to support multiple independent requests—as in the case of miss-under-miss caches, direct memory access I/O that can read data from noncontiguous addresses ("gather") or write data to noncontiguous addresses ("scatter"), multiprocessors (see Chapter 8), or vector computers (see Appendix B)—the effectiveness of the system will depend on the frequency that independent requests will go to different banks. Sequential accesses work well with traditional interleaving, as do any accesses that differ by an odd number. The problem is when this difference between addresses is an even number. One solution, used by larger computers, is to statistically reduce the chances by having many banks; the NEC SX/3, for instance, has up to 128 banks.

The problem with such a solution is that data memory references are not random, and may go to the same bank no matter how many banks are provided. Suppose we have 128 memory banks, interleaved on a word basis, and execute this code:

```
int x[256][512];
       for (j = 0; j < 512; j = j+1)
              for (i = 0; i < 256; i = i+1)
                     x[i][j] = 2 * x[i][j];
```

Since the 512 is an even multiple of 128, all the elements of a column will be in the same memory bank and code will stall on data cache misses no matter how sophisticated a CPU or memory system.

There are both software and hardware solutions to the bank conflict problem. The compiler could do the loop interchange optimization (see page 407) to avoid accessing the same bank. A simpler solution would be for the programmer or the compiler to expand the size of the array so that it is not a power of two, thereby forcing the addresses above to go to different banks.

Before describing a hardware solution, let's review how addressing of banks works. The mapping of an address to a location in a memory bank can be expressed as two problems:

$$\text{Bank number} = \text{Address MOD Number of banks}$$

$$\text{Address within bank} = \lfloor \text{Address} / \text{Number of banks} \rfloor$$

Traditional memory systems keep both the number of banks and the amount of memory per bank a power of two to make this calculation trivial.

One hardware solution to reduce the number of bank conflicts is to have a prime number of banks! Such a number would seem to demand more hardware to perform a complex calculation: the modulo and the division mentioned above. Furthermore, this complex calculation would lengthen each memory access.

Fortunately, there are several hardware schemes to calculate modulo quickly, especially if the prime number of memory banks is one less than a power of two (see Exercise 5.10). In this case division can be replaced by the following simple calculation:

$$\text{Address within bank} = \text{Address MOD Number of words in bank}$$

Since the number of words in a bank is very likely a power of two, we have replaced division by a prime number by bit selection.

The proof of this simplification is based on the *Chinese Remainder Theorem*. This 2000-year-old observation states that as long as two sets of integers a_i and b_i follow these rules:

$$b_i = x \text{ MOD } a_i,\ 0 \le b_i < a_i,\ 0 \le x < a_0 \times a_1 \times a_2 \times \ldots$$

and that a_i and a_j are co-prime if $i \neq j$, then the integer x has only one solution of each pair of integers a_i and b_i (two integers are *co-prime* if they have no common prime number as a factor). The Chinese Remainder Theorem guarantees that there is no ambiguity with this mapping of addresses to banks because the following conditions hold:

- Bank number = Address MOD Number of banks ($b_0 = x$ MOD a_0).
- Address within bank = Address MOD Number of words in bank ($b_1 = x$ MOD a_1).
- Bank number < Number of banks ($0 \le b_0 < a_0$).
- Address within a bank < Number of words in bank ($0 \le b_1 < a_1$).
- Address < Number of banks × Number of words in a bank ($0 \le x < a_0 \times a_1$).

- The number of banks and the number of words in a bank are co-prime (a_0 and a_1 are co-prime).

The first two conditions above are simply the definition of the mapping. The next three conditions are trivially true because an N-word address goes from 0 to $N-1$. The last condition is true since the number of banks is a prime number greater than two and the number of words in a bank is a power of two.

Figure 5.34 shows three memory modules, each with eight words, showing the traditional sequentially interleaved mapping of addresses on the left and the new mapping on the right.

Address within bank	Memory bank					
	Sequentially interleaved			Modulo interleaved		
	0	1	2	0	1	2
0	0	1	2	0	16	8
1	3	4	5	9	1	17
2	6	7	8	18	10	2
3	9	10	11	3	19	11
4	12	13	14	12	4	20
5	15	16	17	21	13	5
6	18	19	20	6	22	14
7	21	22	23	15	7	23

FIGURE 5.34 Three memory banks with sequentially interleaved addressing on the left, requiring a division as part of addressing of the word within a module, and the new mapping, which requires only modulo to a power of two. For example, address 5 is mapped to the second word of memory bank 2 on the left and to the sixth word of memory bank 2 on the right.

Fifth Technique for Higher Bandwidth: DRAM-Specific Interleaving

Thus far we have seen four techniques that improve memory bandwidth: wider memory, interleaved memory, banked memory, and bank conflict avoidance. These techniques work with any memory technology, and have been used or discussed since before DRAMs were invented. This section presents techniques that take advantage of the nature of DRAMs.

As mentioned earlier, DRAM access is divided into row access and column access. DRAMs must buffer a row of bits inside the DRAM for the column access, and this row is usually the square root of the DRAM size—8 Kbits for 64 Mbits, 16 Kbits for 256 Mbits, and so on. To improve performance, all DRAMs come with timing signals that allow repeated accesses to the buffer without another row access time. There are three versions for this optimization:

- *Nibble mode*—The DRAM can supply three extra bits from sequential locations for every row access strobe.
- *Page mode*—The buffer acts like a SRAM; by changing column address, random bits can be accessed in the buffer until the next row access or refresh time.
- *Static column*—Very similar to page mode, except that it's not necessary to toggle the column access strobe line every time the column address changes.

Starting with the 1-Mbit generation, most DRAMs can perform any of the three options, with the optimization selected at the time the die is packaged by choosing which pads to wire up. These operations change the definition of cycle time for DRAMs. Figure 5.35 shows the traditional cycle time plus the fastest speed between accesses in the optimized mode.

	Row access				
Chip size	**Slowest DRAM**	**Fastest DRAM**	**Column access**	**Cycle time**	**Optimized time nibble, page, static column**
64 Kbits	180 ns	150 ns	75 ns	250 ns	150 ns
256 Kbits	150 ns	120 ns	50 ns	220 ns	100 ns
1 Mbits	120 ns	100 ns	25 ns	190 ns	50 ns
4 Mbits	100 ns	80 ns	20 ns	165 ns	40 ns
16 Mbits	80 ns	60 ns	15 ns	120 ns	30 ns
64 Mbits	65 ns	50 ns	10 ns	90 ns	25 ns

FIGURE 5.35 DRAM cycle time for the optimized accesses. This figure is the same as Figure 5.30 (page 429), with a column added to show the optimized cycle time for the three modes. Starting with the 1-Mbit DRAM, optimized cycle time is about four times faster than unoptimized cycle time. It is so much faster that page mode was renamed *fast page mode*. The optimized cycle time is the same no matter which of the three optimized modes is selected.

The advantage of such optimizations is that they use the circuitry already on the DRAMs, adding little cost to the system while achieving almost a fourfold improvement in bandwidth. For example, nibble mode was designed to take advantage of the same program behavior as interleaved memory. The chip reads 4 bits at a time internally, supplying 4 bits externally in the time of four optimized cycles. Unless the bus transfer time is faster than the optimized cycle time, the cost of four-way interleaved memory is only more complicated timing control. Page mode and static column could also be used to get even higher interleaving with slightly more complex control. DRAMs also tend to have weak tristate buffers, implying traditional interleaving with more memory chips must include buffer chips for each memory bank.

Recently new breeds of DRAMs have been produced that further optimize the interface between the DRAM and CPU. One example is from RAMBUS. This company takes the standard DRAM core and provides a new interface, making a single chip act more like a memory system than a memory component. RAMBUS has dropped RAS/CAS, replacing it with a bus that allows other accesses over the bus between the sending of the address and return of the data. (Such a bus is called a *packet-switched bus* or *split-transaction* bus, described in Chapters 6 and 7.) This bus allows a single chip to act as a memory bank. A chip can return a variable amount of data from a single request, and even perform its own refresh. RAMBUS offers a byte-wide interface, and a clock signal so that the chip can be tightly synchronized to the CPU clock. Once the address pipeline is full, a single chip can deliver one byte every 2 ns.

Most main memory systems use techniques such as page mode to reduce the CPU-DRAM performance gap. Unlike traditional interleaved memories, there are no disadvantages using such a mode as DRAMs scale upward in capacity. On the other hand, the new breed of DRAMs such as RAMBUS might cost a premium of, say, 20% per megabyte over traditional DRAMs to provide the greater bandwidth. The marketplace will determine whether the more radical DRAMs such as RAMBUS will become popular for main memory, or whether the price premium restricts them to niche markets.

One example niche market is computer graphics, where a DRAM with a fast serial output line is used to drive displays. This special DRAM is called a *video RAM* or *VRAM*; RAMBUS is challenging VRAMs in this market.

5.7 Virtual Memory

… a system has been devised to make the core drum combination appear to the programmer as a single level store, the requisite transfers taking place automatically.

Kilburn et al. [1962]

At any instant in time computers are running multiple processes, each with its own address space. (Processes are described in the next section.) It would be too expensive to dedicate a full-address-space worth of memory for each process, especially since many processes use only a small part of their address space. Hence, there must be a means of sharing a smaller amount of physical memory among many processes. One way to do this, *virtual memory*, divides physical memory into blocks and allocates them to different processes. Inherent in such an approach must be a *protection* scheme that restricts a process to the blocks belonging only to that process. Most forms of virtual memory also reduce the time to start a program, since not all code and data need be in physical memory before a program can begin.

Although virtual memory is essential for current computers, sharing is not the reason virtual memory was invented. If a program became too large for physical memory, it was the programmer's job to make it fit. Programmers divided programs into pieces, then identified the pieces that were mutually exclusive, and loaded or unloaded these *overlays* under user program control during execution. The programmer ensured that the program never tried to access more physical main memory than was in the machine and that the proper overlay was loaded at the proper time. As one can well imagine, this responsibility eroded programmer productivity. Virtual memory was invented to relieve programmers of this burden; it automatically manages the two levels of the memory hierarchy represented by main memory and secondary storage. Figure 5.36 shows the mapping of virtual memory to physical memory for a program with four pages.

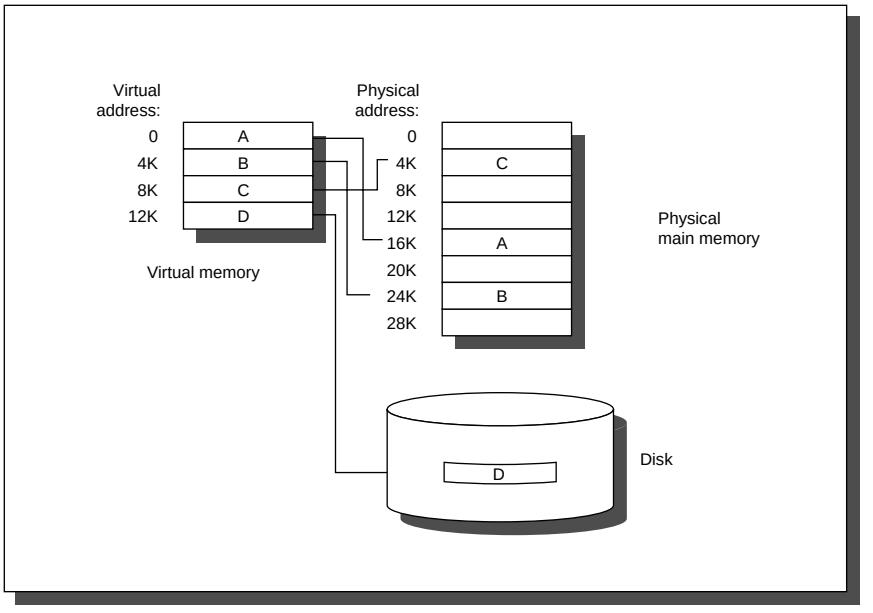

FIGURE 5.36 The logical program in its contiguous virtual address space is shown on the left: it consists of four pages A, B, C, and D. The physical location of three of the blocks is physical memory and one is located on disk.

In addition to sharing protected memory space and automatically managing the memory hierarchy, virtual memory also simplifies loading the program for execution. Called *relocation*, this mechanism allows the same program to run in any location in physical memory. The program in Figure 5.36 can be placed anywhere in physical memory or disk just by changing the mapping between them. (Prior to the popularity of virtual memory, machines would include a relocation

register just for that purpose.) An alternative to a hardware solution would be software that changed all addresses in a program each time it was run.

Several general memory-hierarchy terms from Chapter 1 apply to virtual memory, while some other terms are different. *Page* or *segment* is used for block, and *page fault* or *address fault* is used for miss. With virtual memory, the CPU produces *virtual addresses* that are translated by a combination of hardware and software to *physical addresses*, which access main memory. This process is called *memory mapping* or *address translation*. Today, the two memory-hierarchy levels controlled by virtual memory are DRAMs and magnetic disks. Figure 5.37 shows a typical range of memory-hierarchy parameters for virtual memory.

Parameter	First-level cache	Virtual memory
Block (page) size	16–128 bytes	4096–65,536 bytes
Hit time	1–2 clock cycles	40–100 clock cycles
Miss penalty	8–100 clock cycles	700,000–6,000,000 clock cycles
(Access time)	(6–60 clock cycles)	(500,000–4,000,000 clock cycles)
(Transfer time)	(2–40 clock cycles)	(200,000–2,000,000 clock cycles)
Miss rate	0.5–10%	0.00001– 0.001%
Data memory size	0.016–1MB	16–8192 MB

FIGURE 5.37 Typical ranges of parameters for caches and virtual memory. Virtual memory parameters represent increases of 10 to 100,000 times over cache parameters.

There are further differences between caches and virtual memory beyond those quantitative ones mentioned in Figure 5.37:

- Replacement on cache misses is primarily controlled by hardware, while virtual memory replacement is primarily controlled by the operating system; the longer miss penalty means it's more important to make a really good decision and also that the operating system can afford to get involved and spend more time deciding what to replace.
- The size of the processor address determines the size of virtual memory, but the cache size is independent of the processor address size.
- In addition to acting as the lower-level backing store for main memory in the hierarchy, secondary storage is also used for the file system that is not normally part of the address space; most of secondary storage is in fact taken up by the file system.

Virtual memory also encompasses several related techniques. Virtual memory systems can be categorized into two classes: those with fixed-size blocks, called *pages*, and those with variable-size blocks, called *segments*. Pages are fixed at 4096 to 65,536 bytes, while segment size varies. The largest segment supported

on any machine ranges from 2^{16} bytes up to 2^{32} bytes; the smallest segment is 1 byte. Figure 5.38 shows how the two approaches might divide code and data.

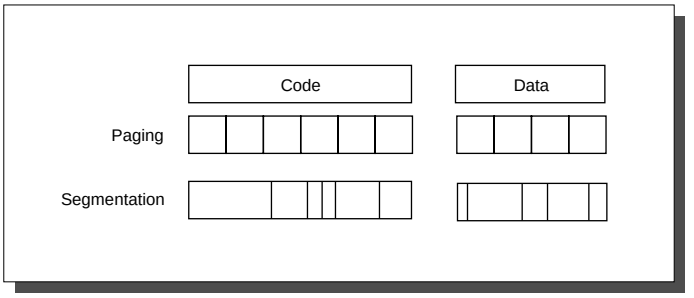

FIGURE 5.38 Example of how paging and segmentation divide a program.

The decision to use paged virtual memory versus segmented virtual memory affects the CPU. Paged addressing has a single fixed-size address divided into page number and offset within a page, analogous to cache addressing. A single address does not work for segmented addresses; the variable size of segments requires one word for a segment number and one word for an offset within a segment, for a total of two words. An unsegmented address space is simpler for the compiler.

The pros and cons of these two approaches have been well documented in operating systems textbooks; Figure 5.39 summarizes the arguments. Because of

	Page	**Segment**
Words per address	One	Two (segment and offset)
Programmer visible?	Invisible to application programmer	May be visible to application programmer
Replacing a block	Trivial (all blocks are the same size)	Hard (must find contiguous, variable-size, unused portion of main memory)
Memory use inefficiency	Internal fragmentation (unused portion of page)	External fragmentation (unused pieces of main memory)
Efficient disk traffic	Yes (adjust page size to balance access time and transfer time)	Not always (small segments may transfer just a few bytes)

FIGURE 5.39 Paging versus segmentation. Both can waste memory, depending on the block size and how well the segments fit together in main memory. Programming languages with unrestricted pointers require both the segment and the address to be passed. A hybrid approach, called *paged segments*, shoots for the best of both worlds: segments are composed of pages, so replacing a block is easy, yet a segment may be treated as a logical unit.

the replacement problem (the third line of the figure), few machines today use pure segmentation. Some machines use a hybrid approach, called *paged segments*, in which a segment is an integral number of pages. This simplifies replacement because memory need not be contiguous, and the full segments need not be in main memory. A more recent hybrid is for a machine to offer multiple page sizes, with the larger sizes being powers of two times the smallest page size. The Alpha AXP 21064, for example, allows 8 KB, 64 KB ($2^3 \times 8$ KB), 512 KB ($2^6 \times 8$ KB), and 4096 KB ($2^9 \times 8$ KB) to act as a single page.

We are now ready to answer the four memory-hierarchy questions for virtual memory.

Q1: Where can a block be placed in main memory?

The miss penalty for virtual memory involves access to a rotating magnetic storage device and is therefore quite high. Given the choice of lower miss rates or a simpler placement algorithm, operating systems designers normally pick lower miss rates because of the exorbitant miss penalty. Thus, operating systems allow blocks to be placed anywhere in main memory. According to the terminology in Figure 5.2 (page 376), this strategy would be labeled fully associative.

Q2: How is a block found if it is in main memory?

Both paging and segmentation rely on a data structure that is indexed by the page or segment number. This data structure contains the physical address of the block. For segmentation, the offset is added to the segment's physical address to obtain the final physical address. For paging, the offset is simply concatenated to this physical page address (see Figure 5.40).

This data structure, containing the physical page addresses, usually takes the form of a *page table.* Indexed by the virtual page number, the size of the table is the number of pages in the virtual address space. Given a 28-bit virtual address, 4-KB pages, and 4 bytes per page table entry, the size of the page table would be 256 KB. To reduce the size of this data structure, some machines apply a hashing function to the virtual address so that the data structure need only be the length of the number of *physical* pages in main memory; this number could be much smaller than the number of virtual pages. Such a structure is called an *inverted page table*. Using the example above, a 64-MB physical memory would only need 128 KB (8×64 MB/4 KB) for an inverted page table; the extra 4 bytes per page table entry is for the virtual address.

To reduce address translation time, computers use a cache dedicated to these address translations, called a *translation look-aside buffer,* or simply *translation buffer.* They are described in more detail shortly.

Q3: Which block should be replaced on a virtual memory miss?

As mentioned above, the overriding operating system guideline is minimizing page faults. Consistent with this guideline, almost all operating systems try to

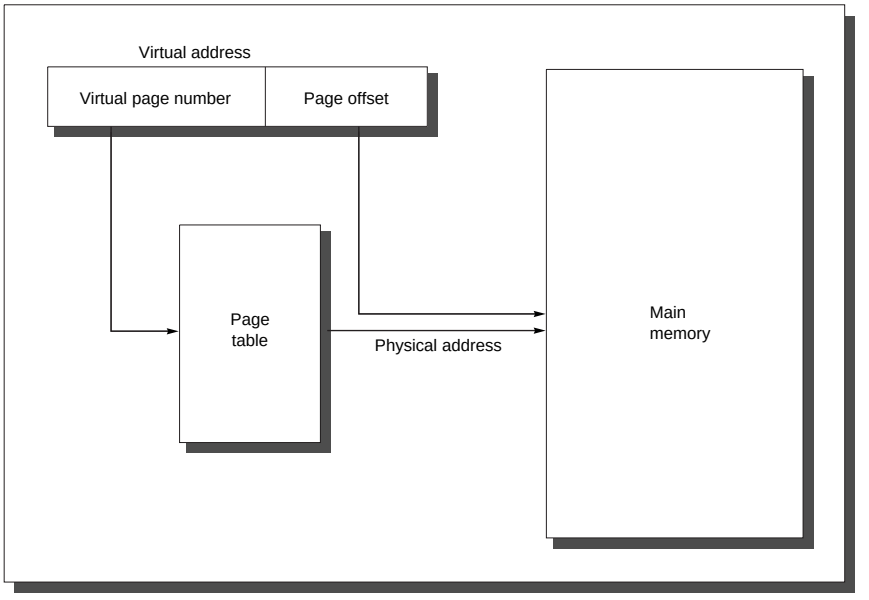

FIGURE 5.40 The mapping of a virtual address to a physical address via a page table.

replace the least-recently used (LRU) block, because that is the one least likely to be needed. To help the operating system estimate LRU, many machines provide a *use bit* or *reference bit*, which is set whenever a page is accessed. The operating system periodically clears the use bits and later records them so it can determine which pages were touched during a particular time period. By keeping track in this way, the operating system can select a page that is among the least-recently referenced.

Q4: What happens on a write?

The level below main memory contains rotating magnetic disks that take millions of clock cycles to access. Because of the great discrepancy in access time, no one has yet built a virtual memory operating system that can write through main memory straight to disk on every store by the CPU. (This remark should not be interpreted as an opportunity to become famous by being the first to build one!) Thus, the write strategy is always write back. Since the cost of an unnecessary access to the next-lower level is so high, virtual memory systems usually include a dirty bit so that the only blocks written to disk are those that have been altered since they were loaded from the disk.

Techniques for Fast Address Translation

Page tables are usually so large that they are stored in main memory, and sometimes paged themselves. This means that every memory access logically takes at least twice as long, with one memory access to obtain the physical address and a second access to get the data. This cost is far too dear.

One remedy is to remember the last translation, so that the mapping process is skipped if the current address refers to the same page as the last one. A more general solution is to again rely on the principle of locality; if the accesses have locality, then the *address translations* for the accesses must also have locality. By keeping these address translations in a special cache, a memory access rarely requires a second access to translate the data. This special address translation cache is referred to as a *translation look-aside buffer* or TLB, also called a *translation buffer* or TB.

A TLB entry is like a cache entry where the tag holds portions of the virtual address and the data portion holds a physical page frame number, protection field, valid bit, and usually a use bit and dirty bit. To change the physical page frame number or protection of an entry in the page table, the operating system must make sure the old entry is not in the TLB; otherwise, the system won't behave properly. Note that this dirty bit means the corresponding *page* is dirty, not that the address translation in the TLB is dirty nor that a particular block in the data cache is dirty.

Figure 5.41 shows the Alpha AXP 21064 data TLB organization, with each step of a translation labeled. The TLB uses fully associative placement; thus, the translation begins (steps 1 and 2) by sending the virtual address to all tags. Of course, the tag must be marked valid to allow a match. At the same time, the type of memory access is checked for a violation (also in step 2) against protection information in the TLB.

For reasons similar to those in the cache case, there is no need to include the 13 bits of the Alpha AXP 21064 page offset in the TLB. The matching tag sends the corresponding physical address through the 32:1 multiplexer (step 3). The page offset is then combined with the physical page frame to form a full 34-bit physical address (step 4).

As mentioned on page 422, one architectural challenge stems from the difficulty of combining caches with virtual memory. Small caches can restrict the index to the page offset so that the index can proceed immediately. While the cache address tags are being read, the virtual portion of the address (the page frame address) is sent to the TLB to be translated. The address comparison is then between the physical address from the TLB and the cache tag; hence the cache index is virtual but the tags are physical.

Address translation can easily be on the critical path determining the clock cycle of the processor, since even in the simplest cache the TLB values must be read and compared. Thus the TLB is usually smaller and faster than the cache-address-tag memory, so that simultaneous TLB reading does not stretch the cache

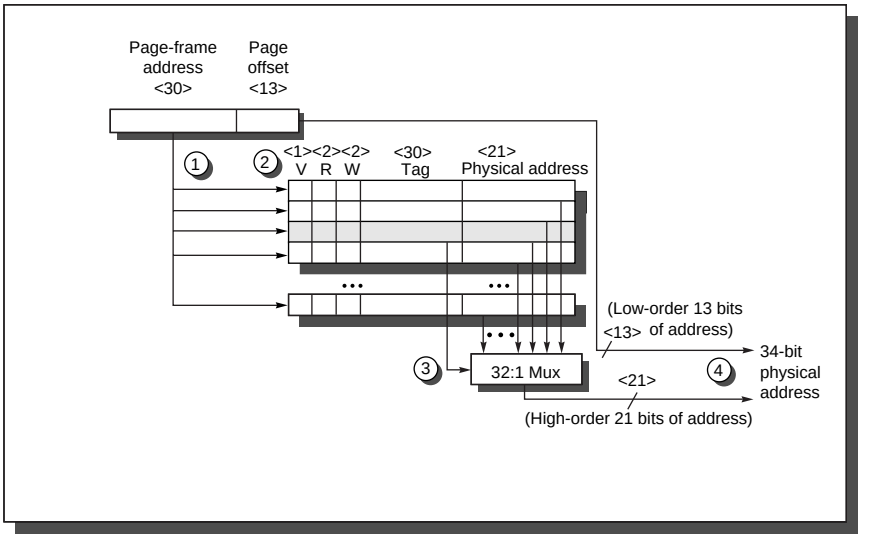

FIGURE 5.41 Operation of the Alpha AXP 21064 data TLB during address translation. The four steps of a TLB hit are shown as circled numbers. The three left fields of an entry are valid (V), read permissions (R), and write permissions (W). Note that there is no specific reference, use bit, or dirty bit. Hence, a page replacement algorithm such as LRU must rely on disabling reads and writes occasionally to record reads and writes to pages to measure usage and whether or not pages are dirty. The advantage of these omissions is that the TLB need not be written during normal memory accesses.

hit time. For example, in the Alpha AXP 21064, the data TLB has 32 blocks and the data cache has 256 blocks. Because of its critical nature, TLB access is sometimes pipelined.

Selecting a Page Size

The most obvious architectural parameter is the page size. Choosing the page is a question of balancing forces that favor a larger page size versus those favoring a smaller size. The following favor a larger size:

- The size of the page table is inversely proportional to the page size; memory (or other resources used for the memory map) can therefore be saved by making the pages bigger.
- As mentioned on page 424 in section 5.5, a larger page size simplifies fast cache hit times.
- Transferring larger pages to or from secondary storage, possibly over a network, is more efficient than transferring smaller pages.

- The number of TLB entries are restricted, so a larger page size means that more memory can be mapped efficiently, thereby reducing the number of TLB misses.

It is for this final reason that recent microprocessors have decided to support multiple page sizes; for some programs, TLB misses can be as significant on CPI as the cache misses.

The main motivation for a smaller page size is conserving storage. A small page size will result in less wasted storage when a contiguous region of virtual memory is not equal in size to a multiple of the page size. The term for this unused memory in a page is *internal fragmentation*. Assuming that each process has three primary segments (text, heap, and stack), the average wasted storage per process will be 1.5 times the page size. This is negligible for machines with megabytes of memory and page sizes in the range of 4 KB to 8 KB. Of course, when the page sizes become very large (more than 32 KB), lots of storage (both main and secondary) may be wasted, as well as I/O bandwidth. A final concern is process start-up time; many processes are small, so larger page sizes would lengthen the time to invoke a process.

5.8 Protection and Examples of Virtual Memory

The invention of multiprogramming, where a computer would be shared by several programs running concurrently, led to new demands for protection and sharing among programs. These are closely tied to virtual memory in computers today, and so we cover the topic here along with two examples of virtual memory.

Multiprogramming leads to the concept of a *process*. Metaphorically, a process is a program's breathing air and living space—that is, a running program plus any state needed to continue running it. Time-sharing is a variation of multiprogramming that shares the CPU and memory with several interactive users at the same time, giving the illusion that all users have their own machines. Thus, at any instant it must be possible to switch from one process to another. This is called a *process switch* or *context switch*.

A process must operate correctly whether it executes continuously from start to finish, or is interrupted repeatedly and switched with other processes. The responsibility for maintaining correct process behavior is shared by the computer designer, who must ensure that the CPU portion of the process state can be saved and restored, and the operating system designer, who must guarantee that processes do not interfere with each others' computations. The safest way to protect the state of one process from another would be to copy the current information to disk. But a process switch would then take seconds—far too long for a time-sharing environment. This problem is solved by operating systems partitioning main memory so that several different processes have their state in memory at the same time. This means that the operating system designer needs help from the

computer designer to provide protection so that one process cannot modify another. Besides protection, the computers also provide for sharing of code and data between processes, to allow communication between processes or to save memory by reducing the number of copies of identical information.

Protecting Processes

The simplest protection mechanism is a pair of registers that checks every address to be sure that it falls between the two limits, traditionally called *base* and *bound*. An address is valid if

$$\text{Base} \leq \text{Address} \leq \text{Bound}$$

In some systems the address is considered an unsigned number that is always added to the base, so the limit test is just

$$(\text{Base} + \text{Address}) \leq \text{Bound}$$

If user processes are allowed to change the base and bounds registers, then users can't be protected from each other. The operating system, however, must be able to change the registers so that it can switch processes. Hence, the computer designer has three more responsibilities in helping the operating system designer protect processes from each other:

1. Provide at least two modes, indicating whether the running process is a user process or an operating system process. This latter process is sometimes called a *kernel* process, a *supervisor* process, or an *executive* process.
2. Provide a portion of the CPU state that a user process can use but not write. This includes the base/bound registers, a user/supervisor mode bit(s), and the exception enable/disable bit. Users are prevented from writing this state because the operating system cannot control user processes if users can change the address range checks, give themselves supervisor privileges, or disable exceptions.
3. Provide mechanisms whereby the CPU can go from user mode to supervisor mode and vice versa. The first direction is typically accomplished by a *system call*, implemented as a special instruction that transfers control to a dedicated location in supervisor code space. The PC is saved from the point of the system call, and the CPU is placed in supervisor mode. The return to user mode is like a subroutine return that restores the previous user/supervisor mode.

Base and bound constitute the minimum protection system, while virtual memory offers a more fine-grained alternative to this simple model. As we have seen, the CPU address must go through a mapping from virtual to physical address. This mapping provides the opportunity for the hardware to check further

for errors in the program or to protect processes from each other. The simplest way of doing this is to add permission flags to each page or segment. For example, since few programs today intentionally modify their own code, an operating system can detect accidental writes to code by offering read-only protection to pages. This page-level protection can be extended by adding user/kernel protection to prevent a user program from trying to access pages that belong to the kernel. As long as the CPU provides a read/write signal and a user/kernel signal, it is easy for the address translation hardware to detect stray memory accesses before they can do damage. Such reckless behavior simply interrupts the CPU and invokes the operating system.

Processes are thus protected from one another by having their own page tables, each pointing to distinct pages of memory. Obviously, user programs must be prevented from modifying their page tables or protection would be circumvented.

Protection can be escalated, depending on the apprehension of the computer designer or the purchaser. Rings added to the CPU protection structure expand memory access protection from two levels (user and kernel) to many more. Like a military classification system of top secret, secret, confidential, and unclassified, concentric *rings* of security levels allow the most trusted to access anything, the second most trusted to access everything except the innermost level, and so on until the "civilian" programs, which are the least trusted and, hence, have the most limited range of accesses. There may also be restrictions on what pieces of memory can contain code—execute protection—and even on the entrance point between the levels. The Intel Pentium protection structure, which uses rings, is described later in this section. It is not clear today whether rings are an improvement in practice over the simple system of user and kernel modes.

As the designer's apprehension escalates to trepidation, these simple rings may not suffice. Restricting the freedom given a program in the inner sanctum requires a new classification system. Instead of a military model, the analogy of this system is to keys and locks: A program can't unlock access to the data unless it has the key. For these keys, or *capabilities*, to be useful, the hardware and operating system must be able to explicitly pass them from one program to another without allowing a program itself to forge them. Such checking requires a great deal of hardware support if time for checking keys is to be kept low.

A Paged Virtual Memory Example: The Alpha AXP Memory Management and the 21064 TLB

The Alpha AXP architecture uses a combination of segmentation and paging, providing protection while minimizing page table size. The 64-bit address space is first divided into three segments: *seg0* (bits 63 – 41 = 0...00), *kseg* (bits 63 – 41 = 0...01), and *seg1* (bits 63 to 41 = 1...11). kseg is reserved for the operating system kernel, has uniform protection for the whole space, and does not use memory management. User processes use seg0, which is mapped into pages with individual protection. Figure 5.42 shows the layout of seg0 and seg1. seg0 grows from

address 0 upward, while seg1 grows downward to 0. Many systems today use some such combination of predivided segments and paging. This approach provides many advantages: segmentation divides the address space and conserves page table space, while paging provides virtual memory, relocation, and protection.

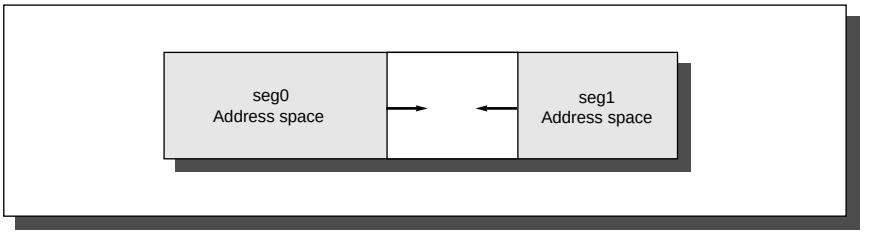

FIGURE 5.42 The organization of seg0 and seg1 in the Alpha. User processes live in seg0, while seg1 is used for portions of the page tables. seg0 includes a downward growing stack, text and data, and an upward growing heap.

Even with this division, the size of page tables for the 64-bit address space is alarming. Hence the Alpha uses a three-level hierarchical page table to map the address space to keep the size reasonable. The addresses for each of these page tables come from three "level" fields, labeled level1, level2, and level3. Figure 5.43 shows address translation in the Alpha AXP. Address translation starts with adding the level1 address field to the page table base register and then reading memory from this location to get the base of the second-level page table. The level2 address field is in turn added to this newly fetched address, and memory is accessed again to determine the base of the third page table. The level3 address field is added to this base address, and memory is read using this sum to (finally) get the physical address of the page being referenced. This address is concatenated with the page offset to get the full physical address. Each page table in the Alpha AXP architecture is constrained to fit within a single page, so all page table addresses are physical addresses that need no further translation.

The Alpha uses a 64-bit *page table entry* (*PTE*) in each of these page tables. The first 32 bits contain the physical page frame number, and the other half includes the following five protection fields:

- *Valid*—Says that the page frame number is valid for hardware translation
- *User read enable*—Allows user programs to read data within this page
- *Kernel read enable*—Allows the kernel to read data within this page
- *User write enable*—Allows user programs to write data within this page
- *Kernel write enable*—Allows the kernel to write data within this page

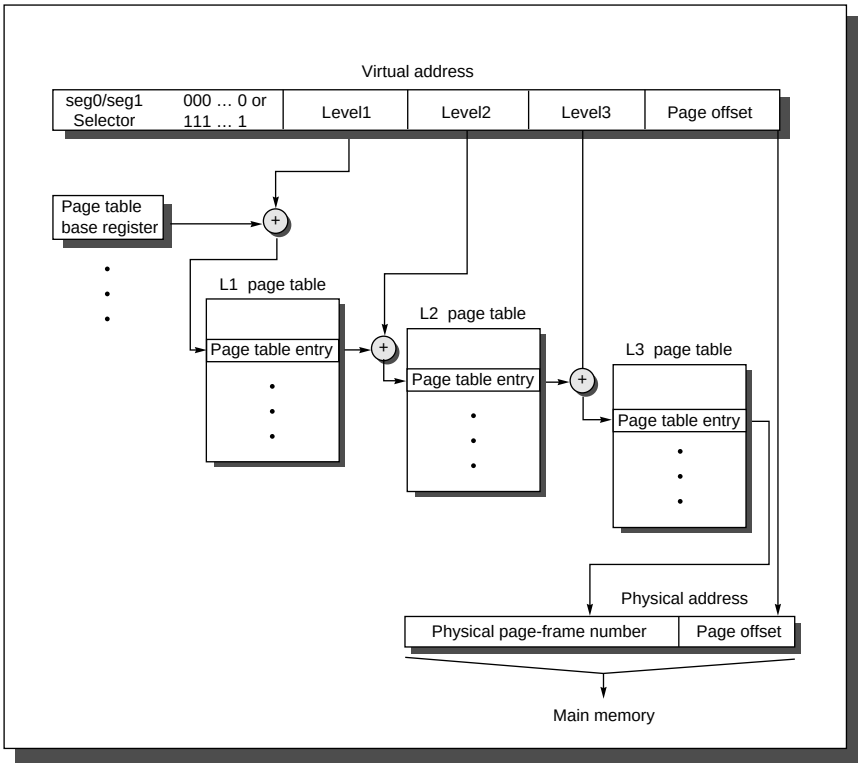

FIGURE 5.43 The mapping of an Alpha virtual address. Each page table is exactly one page long, so each level field is n bits wide where 2^n = page size/8. The Alpha AXP architecture document allows the page size to grow from 8 KB in the current implementations to 16 KB, 32 KB, or 64 KB in the future. The virtual address for each page size grows from the current 43 bits to 47, 51, or 55 bits and the maximum physical address size grows from the current 41 bits to 45, 47, or 48 bits. The 21064 uses 8-KB pages, but it implements just 34 bits of the maximum 41-bit physical address possible in this scheme.

In addition, the PTE has fields reserved for systems software to use as it pleases. Since the Alpha goes through three levels of tables on a TLB miss, there are three potential places to check protection restrictions. The Alpha obeys only the third-level PTE, checking the first two only to be sure the valid bit is set.

Since the PTEs are 8 bytes long, the page tables are exactly one page long, and the Alpha AXP 21064 has 8-KB pages, each page table has 1024 PTEs. Each of the three level fields are 10 bits long and the page offset is 13 bits, which leaves $64 - (3 \times 10 + 13)$ or 21 bits to be defined. If this is a seg0 address, the most-significant bit is a 0, and for seg1 the two most-significant bits are 11_{two}. Alpha requires all bits to the left of the level1 field to be identical. For seg0 these 21 bits

are all zeros and for seg1 they are all ones. This means the 21064 virtual addresses are really 43 bits long instead of the full 64 bits found in registers. The physical addresses would appear to be 32 + 13 or 45 bits, but Alpha AXP architecture requires that the physical address be smaller than the virtual address. The 21064 saves space on the chip by further limiting the physical address to 34 bits.

The maximum virtual address and physical address is then tied to the page size. The architecture document allows for the Alpha to expand the minimum page size from 8 KB up to 64 KB, thereby increasing the virtual address to 3×13 + 16 or 55 bits and the maximum physical address to 32 + 16 or 48 bits; it will be interesting to see whether or not operating systems accommodate such expansion plans over the life of the Alpha.

While we have explained translation of legal addresses, what prevents the user from creating illegal address translations and getting into mischief? The page tables themselves are protected from being written by user programs. Thus, the user can try any virtual address, but by controlling the page table entries the operating system controls what physical memory is accessed. Sharing of memory between processes is accomplished by having a page table entry in each address space point to the same physical memory page.

The first implementation of this architecture was the Alpha AXP 21064, which employs two TLBs to reduce address translation time, one for instruction accesses and another for data accesses. Figure 5.44 shows the key parameters of each TLB. The Alpha allows the operating system to tell the TLB that contiguous sequences of pages can act as one: the options are 8, 64, and 512 times the minimum page size. Thus the variable page size of a PTE mapping makes the match more challenging, as the size of the space being mapped in the PTE also must be checked to determine the match. Figure 5.41 above describes the data TLB.

Parameter	Description
Block size	1 PTE (8 bytes)
Hit time	1 clock cycle
Miss penalty (average)	20 clock cycles
TLB size	Instruction: 8 PTE for 8-KB pages, 4 PTE for 4-MB pages (96 bytes total) Data: 32 PTE for 8-KB, 64-KB, 512-KB, or 4-MB pages (256 bytes total)
Block selection	Random, but not last used
Write strategy	(Not applicable)
Block placement	Fully associative

FIGURE 5.44 Memory-hierarchy parameters of the Alpha AXP 21064 TLB.

Memory management in the Alpha 21064 is typical of most computers today, relying on page-level address translation and correct operation of the operating system to provide safety to multiple processes sharing the computer. The primary difference is that Alpha has extended the virtual address beyond 32 bits. In the next section we see a protection scheme for individuals who want to trust the operating system as little as possible.

A Segmented Virtual Memory Example: Protection in the Intel Pentium

The second system is the most dangerous system a man ever designs. . . . The general tendency is to over-design the second system, using all the ideas and frills that were cautiously sidetracked on the first one.

F. P. Brooks, Jr., *The Mythical Man-Month* (1975)

The original 8086 used segments for addressing, yet it provided nothing for virtual memory or for protection. Segments had base registers but no bound registers and no access checks, and before a segment register could be loaded the corresponding segment had to be in physical memory. Intel's dedication to virtual memory and protection is evident in the successors to the 8086, with a few fields extended to support larger addresses. This protection scheme is elaborate, with many details carefully designed to try to avoid security loopholes. The next few pages highlight a few of the Intel safeguards; if you find the reading difficult, imagine the difficulty of implementing them!

The first enhancement is to double the traditional two-level protection model: the Pentium has four levels of protection. The innermost level (0) corresponds to Alpha kernel mode and the outermost level (3) corresponds to Alpha user mode. The Pentium has separate stacks for each level to avoid security breaches between the levels. There are also data structures analogous to Alpha page tables that contain the physical addresses for segments, as well as a list of checks to be made on translated addresses.

The Intel designers did not stop there. The Pentium divides the address space, allowing both the operating system and the user access to the full space. The Pentium user can call an operating system routine in this space and even pass parameters to it while retaining full protection. This safe call is not a trivial action, since the stack for the operating system is different from the user's stack. Moreover, the Pentium allows the operating system to maintain the protection level of the *called* routine for the parameters that are passed to it. This potential loophole in protection is prevented by not allowing the user process to ask the operating system to access something indirectly that it would not have been able to access itself. (Such security loopholes are called *Trojan horses*.)

The Intel designers were guided by the principle of trusting the operating system as little as possible, while supporting sharing and protection. As an example

of the use of such protected sharing, suppose a payroll program writes checks and also updates the year-to-date information on total salary and benefits payments. Thus, we want to give the program the ability to read the salary and year-to-date information, and modify the year-to-date information but not the salary. We shall see the mechanism to support such features shortly. In the rest of this subsection, we will look at the big picture of the Pentium protection and examine its motivation.

Adding Bounds Checking and Memory Mapping

The first step in enhancing the Intel processor was getting the segmented addressing to check bounds as well as supply a base. Rather than a base address, as in the 8086, segment registers in the Pentium contain an index to a virtual memory data structure called a *descriptor table*. Descriptor tables play the role of page tables in the Alpha. On the Pentium the equivalent of a page table entry is a *segment descriptor*. It contains fields found in PTEs:

- A *present bit*—equivalent to the PTE valid bit, used to indicate this is a valid translation
- A *base field*—equivalent to a page frame address, containing the physical address of the first byte of the segment
- An *access bit*—like the reference bit or use bit in some architectures that is helpful for replacement algorithms
- An *attributes field*—specifies the valid operations and protection levels for operations that use this segment

There is also a *limit field*, not found in paged systems, which establishes the upper bound of valid offsets for this segment. Figure 5.45 shows examples of Pentium segment descriptors.

Pentium provides an optional paging system in addition to this segmented addressing, where the upper portion of the 32-bit address selects the segment descriptor and the middle portion is used as an index into the page table selected by the descriptor. We describe below the protection system that does not rely on paging.

Adding Sharing and Protection

To provide for protected sharing, half of the address space is shared by all processes and half is unique to each process, called *global address space* and *local address space*, respectively. Each half is given a descriptor table with the appropriate name. A descriptor pointing to a shared segment is placed in the global descriptor table, while a descriptor for a private segment is placed in the local descriptor table.

A program loads a Pentium segment register with an index to the table *and* a bit saying which table it desires. The operation is checked according to the

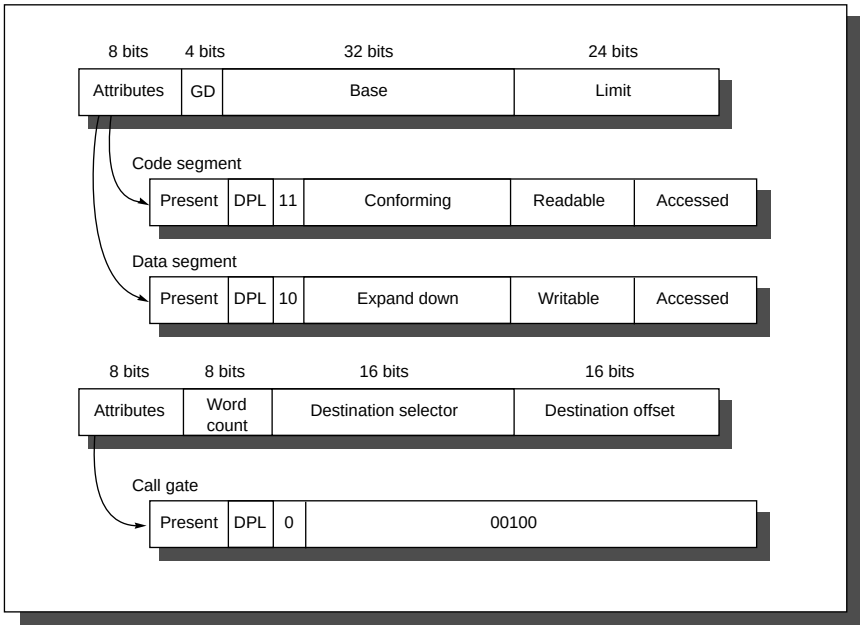

FIGURE 5.45 The Pentium segment descriptors are distinguished by bits in the attributes field. *Base*, *limit, present*, *readable*, and *writable* are all self-explanatory. D gives the default addressing size of the instructions: 16 bits or 32 bits. G gives the granularity of the segment limit: 0 means in bytes and 1 means in 4-KB pages. G is set to 1 when paging is turned on to set the size of the page tables. DPL means *descriptor privilege level*—this is checked against the code privilege level to see if the access will be allowed. *Conforming* says the code takes on the privilege level of the code being called rather than the privilege level of the caller; it is used for library routines. The *expand-down field* flips the check to let the base field be the high-water mark and the limit field be the low-water mark. As one might expect, this is used for stack segments that grow down. *Word count* controls the number of words copied from the current stack to the new stack on a call gate. The other two fields of the call gate descriptor, *destination selector* and *destination offset*, select the descriptor of the destination of the call and the offset into it, respectively. There are many more than these three segment descriptors in the Pentium.

attributes in the descriptor, the physical address being formed by adding the offset in the CPU to the base in the descriptor, provided the offset is less than the limit field. Every segment descriptor has a separate 2-bit field to give the legal access level of this segment. A violation occurs only if the program tries to use a segment with a lower protection level in the segment descriptor.

We can now show how to invoke the payroll program mentioned above to update the year-to-date information without allowing it to update salaries. The program could be given a descriptor to the information that has the writable field clear, meaning it can read but not write the data. A trusted program can then be supplied that will only write the year-to-date information and is given a descrip-

tor with the writable field set (Figure 5.45). The payroll program invokes the trusted code using a code segment descriptor with the conforming field set. This means the called program takes on the privilege level of the code being called rather than the privilege level of the caller. Hence, the payroll program can read the salaries and call a trusted program to update the year-to-date totals, yet the payroll program cannot modify the salaries. If a Trojan horse exists in this system, to be effective it must be located in the trusted code whose only job is to update the year-to-date information. The argument for this style of protection is that limiting the scope of the vulnerability enhances security.

Adding Safe Calls from User to OS Gates and Inheriting Protection Level for Parameters

Allowing the user to jump into the operating system is a bold step. How, then, can a hardware designer increase the chances of a safe system without trusting the operating system or any other piece of code? The Pentium approach is to restrict where the user can enter a piece of code, to safely place parameters on the proper stack, and to make sure the user parameters don't get the protection level of the called code.

To restrict entry into others' code, the Pentium provides a special segment descriptor, or *call gate*, identified by a bit in the attributes field. Unlike other descriptors, call gates are full physical addresses of an object in memory; the offset supplied by the CPU is ignored. As stated above, their purpose is to prevent the user from randomly jumping anywhere into a protected or more-privileged code segment. In our programming example, this means the only place the payroll program can invoke the trusted code is at the proper boundary. This restriction is needed to make conforming segments work as intended.

What happens if caller and callee are "mutually suspicious," so that neither trusts the other? The solution is found in the word count field in the bottom descriptor in Figure 5.45. When a call instruction invokes a call gate descriptor, the descriptor copies the number of words specified in the descriptor from the local stack onto the stack corresponding to the level of this segment. This allows the user to pass parameters by first pushing them onto the local stack. The hardware then safely transfers them onto the correct stack. A return from a call gate will pop the parameters off both stacks and copy any return values to the proper stack. Note that this model is incompatible with the current practice of passing parameters in registers.

This scheme still leaves open the potential loophole of having the operating system use the user's address, passed as parameters, with the operating system's security level, instead of with the user's level. The Pentium solves this problem by dedicating 2 bits in every CPU segment register to the *requested protection level*. When an operating system routine is invoked, it can execute an instruction that sets this 2-bit field in all address parameters with the protection level of the user that called the routine. Thus, when these address parameters are loaded into

the segment registers, they will set the requested protection level to the proper value. The Pentium hardware then uses the requested protection level to prevent any foolishness: No segment can be accessed from the system routine using those parameters if it has a more-privileged protection level than requested.

Summary: Protection on the Alpha versus the Pentium

If the Pentium protection model looks harder to build than the Alpha model, that's because it is. This effort must be especially frustrating for the Pentium engineers, since few customers use the elaborate protection mechanism. Also, the fact that the protection model is a mismatch for the simple paging protection of UNIX means it will be used only by someone writing an operating system especially for this computer. NT from Microsoft is the best candidate, but only time will tell whether the performance cost of such protection is justified for a personal computer operating system.

One wild card is the increasing popularity of the Internet, where virtually any machine can become an information provider, and hence almost anyone could access the desktop computer. This openness leads to extraordinary sharing of information, but it also gives a powerful opportunity for malicious behavior.

We conclude this section with questions rather than answers: Will the considerable protection engineering effort, which must be borne by each generation of the 80x86 family, be put to good use? Will it prove any safer in practice than its paging system? Will the popularity of the Internet lead to demands of increased support for protection in all computers?

5.9 Crosscutting Issues in the Design of Memory Hierarchies

This section describes four topics discussed in other chapters that are fundamental to memory-hierarchy design.

Superscalar CPU and Number of Ports to the Cache

One complexity of the advanced designs of Chapter 4 is that multiple instructions can be issued within a single clock cycle. Clearly, if there is not sufficient peak bandwidth from the cache to match the peak demands of the instructions, there is little benefit to designing such parallelism in the processor. As mentioned above, similar reasoning applies to CPUs that want to continue executing instructions on a cache miss: clearly the memory hierarchy must also be nonblocking or the CPU benefits little.

For example, the IBM RS/6000 Power 2 model 900 can issue up to six instructions per clock cycle, and its data cache can supply two 128-bit accesses per clock cycle. The RS/6000 does this by making the instruction cache and data cache wide and by making two reads to the data cache each clock cycle, certainly likely to be the critical path in the 71.5-MHz machine.

Speculative Execution and the Memory System

Inherent in CPUs that support speculative execution or conditional instructions is the possibility of generating invalid addresses that would not occur without speculative execution. Not only would this be incorrect behavior if exceptions were taken, the benefits of speculative execution would be swamped by false exception overhead. Hence the memory system must identify speculatively executed instructions and conditionally executed instructions and suppress the corresponding exception.

By similar reasoning, we cannot allow such instructions to cause the cache to stall on a miss, for again unnecessary stalls could overwhelm the benefits of speculation. Hence these CPUs must be matched with nonblocking caches (see page 414).

Compiler Optimization: Instruction-Level Parallelism versus Reducing Cache Misses

Sometimes the compiler must choose between improving instruction-level parallelism and improving cache performance. For example, the code below,

```
for (i = 0; i < 512; i = i+1)
       for (j = 1; j < 512; j = j+1)
              x[i][j] = 2 * x[i][j-1];
```

accesses the data in the order they are stored, thereby minimizing cache misses. Unfortunately, the dependency limits parallel execution. Unrolling the loop shows this dependency:

```
for (i = 0; i < 512; i = i+1)
       for (j = 1; j < 512; j = j+4){
              x[i][j]   = 2 * x[i][j-1];
              x[i][j+1] = 2 * x[i][j];
              x[i][j+2] = 2 * x[i][j+1];
              x[i][j+3] = 2 * x[i][j+2];
              };
```

Each of the last three statements has a RAW dependency on the prior statement. We can improve parallelism by interchanging the two loops:

```
for (j = 1; j < 512; j = j+1)
       for (i = 0; i < 512; i = i+1)
                     x[i][j] = 2 * x[i][j-1];
```

Unrolling the loop shows this parallelism:

```
for (j = 1; j < 512; j = j+1)
       for (i = 0; i < 512; i = i+4) {
              x[i][j]   = 2 * x[i][j-1];
              x[i+1][j] = 2 * x[i+1][j-1];
              x[i+2][j] = 2 * x[i+2][j-1];
              x[i+3][j] = 2 * x[i+3][j-1];
              };
```

Now all four statements in the loop are independent! Alas, increasing parallelism leads to accesses that hop through memory, reducing spatial locality and cache hit rates.

I/O and Consistency of Cached Data

Because of caches, data can be found in memory and in the cache. As long as the CPU is the sole device changing or reading the data and the cache stands between the CPU and memory, there is little danger in the CPU seeing the old or *stale* copy. I/O devices give the opportunity for other devices to cause copies to be inconsistent or for other devices to read the stale copies. Figure 5.46 illustrates the problem, generally referred to as the *cache-coherency* problem.

The question is this: Where does the I/O occur in the computer—between the I/O device and the cache or between the I/O device and main memory? If input puts data into the cache and output reads data from the cache, both I/O and the CPU see the same data, and the problem is solved. The difficulty in this approach is that it interferes with the CPU. I/O competing with the CPU for cache access will cause the CPU to stall for I/O. Input will also interfere with the cache by displacing some information with the new data that is unlikely to be accessed by the CPU soon. For example, on a page fault the CPU may need to access a few words in a page, but a program is not likely to access every word of the page if it were loaded into the cache. Given the integration of caches onto the same integrated circuit, it is also difficult for that interface to be visible.

The goal for the I/O system in a computer with a cache is to prevent the stale-data problem while interfering with the CPU as little as possible. Many systems, therefore, prefer that I/O occur directly to main memory, with main memory

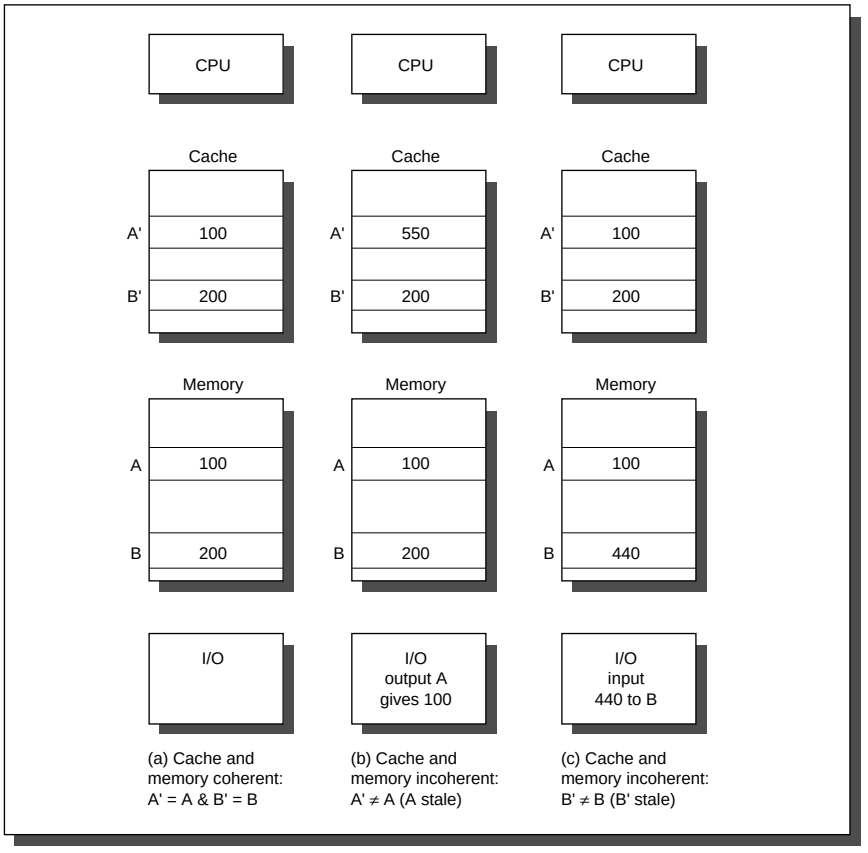

FIGURE 5.46 The cache-coherency problem. A' and B' refer to the cached copies of A and B in memory. (a) shows cache and main memory in a coherent state. In (b) we assume a write-back cache when the CPU writes 550 into A. Now A' has the value but the value in memory has the old, stale value of 100. If an output used the value of A from memory, it would get the stale data. In (c) the I/O system inputs 440 into the memory copy of B, so now B' in the cache has the old, stale data.

acting as an I/O buffer. If a write-through cache is used, then memory has an up-to-date copy of the information, and there is no stale-data issue for output. (This is a reason many machines use write through.) Input requires some extra work. The software solution is to guarantee that no blocks of the I/O buffer designated for input are in the cache. In one approach, a buffer page is marked as noncachable; the operating system always inputs to such a page. In another approach, the operating system flushes the buffer addresses from the cache after the input occurs. A hardware solution is to check the I/O addresses on input to see if they are in the cache; to avoid slowing down the cache to check addresses, sometimes a duplicate set of tags are used to allow checking of I/O addresses in parallel with processor cache accesses. If there is a match of I/O addresses in the cache, the

cache entries are invalidated to avoid stale data. All these approaches can also be used for output with write-back caches. More about this is found in Chapter 6.

The cache-coherency problem applies to multiprocessors as well as I/O. Unlike I/O, where multiple data copies are a rare event—one to be avoided whenever possible—a program running on multiple processors will want to have copies of the same data in several caches. Performance of a multiprocessor program depends on the performance of the system when sharing data. The protocols to maintain coherency for multiple processors are called *cache-coherency protocols*, and are described in Chapter 8.

5.10 Putting It All Together: The Alpha AXP 21064 Memory Hierarchy

Thus far we have given glimpses of the Alpha AXP 21064 memory hierarchy; this section unveils the full design and shows the performance of its components for the SPEC92 programs. Figure 5.47 gives the overall picture of this design.

Let's really start at the beginning, when the Alpha is turned on. Hardware on the chip loads the instruction cache from an external PROM. This initialization allows the 8-KB instruction cache to omit a valid bit, for there are always valid instructions in the cache; they just might not be the ones your program is interested in. The hardware does clear the valid bits in the data cache. The PC is set to the kseg segment so that the instruction addresses are not translated, thereby avoiding the TLB.

One of the first steps is to update the instruction TLB with valid page table entries (PTEs) for this process. Kernel code updates the TLB with the contents of the appropriate page table entry for each page to be mapped. The instruction TLB has eight entries for 8-KB pages and four for 4-MB pages. (The 4-MB pages are used by large programs such as the operating system or data bases that will likely touch most of their code.) A miss in the TLB invokes the Privileged Architecture Library (PAL code) software that updates the TLB. PAL code is simply machine language routines with some implementation-specific extensions to allow access to low-level hardware, such as the TLB. PAL code runs with exceptions disabled, and instruction accesses are not checked for memory management violations, allowing PAL code to fill the TLB.

Once the operating system is ready to begin executing a user process, it sets the PC to the appropriate address in segment seg0.

We are now ready to follow memory hierarchy in action: Figure 5.47 is labeled with the steps of this narrative. The page frame portion of this address is sent to the TLB (step 1), while the 8-bit index from the page offset is sent to the direct-mapped 8-KB (256 32-byte blocks) instruction cache (step 2). The fully associative TLB simultaneously searches all 12 entries to find a match between the address and a valid PTE (step 3). In addition to translating the address, the TLB checks to see if the PTE demands that this access result in an exception. An exception might occur if either this access violates the protection on the page or if

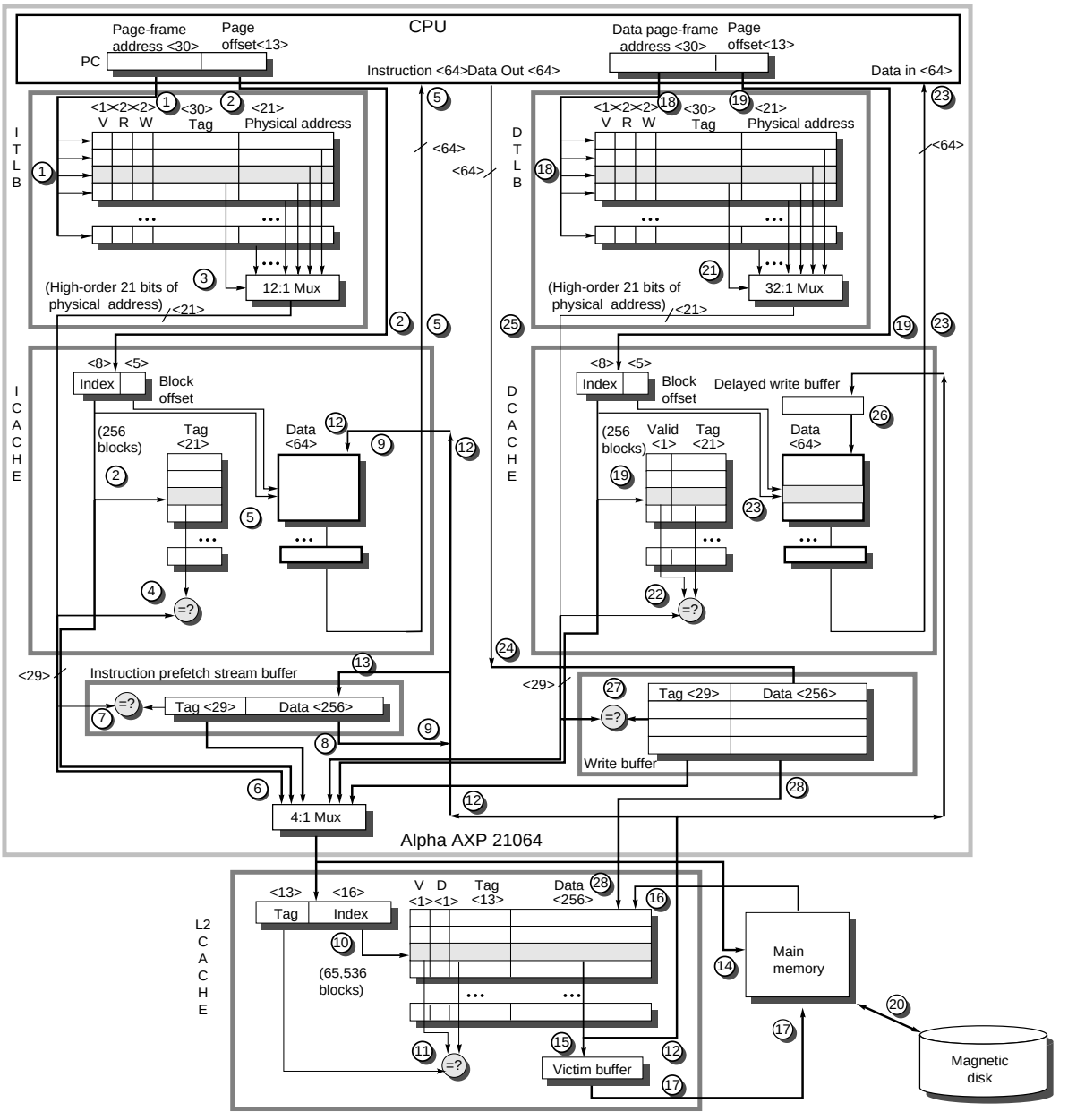

FIGURE 5.47 The overall picture of the Alpha AXP 21064 memory hierarchy. Individual components can be seen in greater detail in Figures 5.5 (page 381), 5.28 (page 426), and 5.41 (page 446). While the data TLB has 32 entries, the instruction TLB has just 12.

the page is not in main memory. If there is no exception, and if the translated physical address matches the tag in the instruction cache (step 4), then the proper 8 bytes of the 32-byte block are furnished to the CPU using the lower bits of the page offset (step 5), and the instruction stream access is done.

A miss, on the other hand, simultaneously starts an access to the second-level cache (step 6) and checks the prefetch instruction stream buffer (step 7). If the desired instruction is found in the stream buffer (step 8), the critical 8 bytes are sent to the CPU, the full 32-byte block of the stream buffer is written into the instruction cache (step 9), and the request to the second-level cache is canceled. Steps 6 to 9 take just a single clock cycle.

If the instruction is not in the prefetch stream buffer, the second-level cache continues trying to fetch the block. The 21064 microprocessor is designed to work with direct-mapped second-level caches from 128 KB to 8 MB with a miss penalty between 3 and 16 clock cycles. For this section we use the memory system of the DEC 3000 model 800 Alpha AXP. It has a 2-MB (65,536 32-byte blocks) second-level cache, so the 29-bit block address is divided into a 13-bit tag and a 16-bit index (step 10). The cache reads the tag from that index and if it matches (step 11), the cache returns the critical 16 bytes in the first 5 clock cycles and the other 16 bytes in the next 5 clock cycles (step 12). The path between the first- and second-level cache is 128 bits wide (16 bytes). At the same time, a request is made for the next sequential 32-byte block, which is loaded into the instruction stream buffer in the next 10 clock cycles (step 13).

The instruction stream does not rely on the TLB for address translation. It simply increments the physical address of the miss by 32 bytes, checking to make sure that the new address is within the same page. If the incremented address crosses a page boundary, then the prefetch is suppressed.

If the instruction is not found in the secondary cache, the translated physical address is sent to memory (step 14). The DEC 3000 model 800 divides memory into four memory mother boards (MMB), each of which contains two to eight SIMMs (single inline memory modules). The SIMMs come with eight DRAMs for information plus one DRAM for error protection per side, and the options are single- or double-sided SIMMs using 1-Mbit, 4-Mbit, or 16-Mbit DRAMs. Hence the memory capacity of the model 800 is 8 MB ($4 \times 2 \times 8 \times 1 \times 1/8$) to 1024 MB ($4 \times 8 \times 8 \times 16 \times 2/8$), always organized 256 bits wide. The average time to transfer 32 bytes from memory to the secondary cache is 36 clock cycles after the processor makes the request. The second-level cache loads this data 16 bytes at a time.

Since the second-level cache is a write-back cache, any miss can lead to some old block being written back to memory. The 21064 places this "victim" block into a victim buffer to get out of the way of new data (step 15). The new data are loaded into the cache as soon as they arrive (step 16), and then the old data are written from the victim buffer (step 17). There is a single block in the victim buffer, so a second miss would need to stall until the victim buffer empties.

Suppose this initial instruction is a load. It will send the page frame of its data address to the data TLB (step 18) at the same time as the 8-bit index from the page offset is sent to the data cache (step 19). The data TLB is a fully associative cache containing 32 PTEs, each of which represents page sizes from 8 KB to 4 MB. A TLB miss will trap to PAL code to load the valid PTE for this address. In the worst case, the page is not in memory, and the operating system gets the page from disk (step 20). Since millions of instructions could execute during a page fault, the operating system will swap in another process if there is something waiting to run.

Assuming that we have a valid PTE in the data TLB (step 21), the cache tag and the physical page frame are compared (step 22), with a match sending the desired 8 bytes from the 32-byte block to the CPU (step 23). A miss goes to the second-level cache, which proceeds exactly like an instruction miss.

Suppose the instruction is a store instead of a load. The page frame portion of the data address is again sent to the data TLB and the data cache (steps 18 and 19), which checks for protection violations as well as translates the address. The physical address is then sent to the data cache (steps 21 and 22). Since the data cache uses write through, the store data are simultaneously sent to the write buffer (step 24) and the data cache (step 25). As explained on page 425, the 21064 pipelines write hits. The data address of this store is checked for a match, and at the same time the data from the previous write hit are written to the cache (step 26). If the address check was a hit, then the data from this store are placed in the write pipeline buffer. On a miss, the data are just sent to the write buffer since the data cache does not allocate on a write miss.

The write buffer takes over now. It has four entries, each containing a whole cache block. If the buffer is full, then the CPU must stall until a block is written to the second-level cache. If the buffer is not full, the CPU continues and the address of the word is presented to the write buffer (step 27). It checks to see if the word matches any block already in the buffer so that a sequence of writes can be stitched together into a full block, thereby optimizing use of the write bandwidth between the first- and second-level cache.

All writes are eventually passed on to the second-level cache. If a write is a hit, then the data are written to the cache (step 28). Since the second-level cache uses write back, it cannot pipeline writes: a full 32-byte block write takes 5 clock cycles to check the address and 10 clock cycles to write the data. A write of 16 bytes or less takes 5 clock cycles to check the address and 5 clock cycles to write the data. In either case the cache marks the block as dirty.

If the access to the second-level cache is a miss, the victim block is checked to see if it is dirty; if so, it is placed in the victim buffer as before (step 15). If the new data are a full block, then the data are simply written and marked dirty. A partial block write results in an access to main memory since the second-level cache policy is to allocate on a write miss.

Performance of the 21064 Memory Hierarchy

How well does the 21064 work? The bottom line in this evaluation is the percentage of time lost while the CPU is waiting for the memory hierarchy. The major components are the instruction and data caches, instruction and data TLBs, and the secondary cache. Figure 5.48 shows the percentage of the execution time

	CPI							Miss rates		
Program	**I cache**	**D cache**	**L2**	**Total cache**	**Instr. issue**	**Other stalls**	**Total CPI**	**I cache**	**D cache**	**L2**
TPC-B (db1)	0.57	0.53	0.74	1.84	0.79	1.67	4.30	8.10%	41.00%	7.40%
TPC-B (db2)	0.58	0.48	0.75	1.81	0.76	1.73	4.30	8.30%	34.00%	6.20%
AlphaSort	0.09	0.24	0.50	0.83	0.70	1.28	2.81	1.30%	22.00%	17.40%
Avg comm	0.41	0.42	0.66	1.49	0.75	1.56	3.80	5.90%	32.33%	10.33%
espresso	0.06	0.13	0.01	0.20	0.74	0.57	1.51	0.84%	9.00%	0.33%
li	0.14	0.17	0.00	0.31	0.75	0.96	2.02	2.04%	9.00%	0.21%
eqntott	0.02	0.16	0.01	0.19	0.79	0.41	1.39	0.22%	11.00%	0.55%
compress	0.03	0.30	0.04	0.37	0.77	0.52	1.66	0.48%	20.00%	1.19%
sc	0.20	0.18	0.04	0.42	0.78	0.85	2.05	2.79%	12.00%	0.93%
gcc	0.33	0.25	0.02	0.60	0.77	1.14	2.51	4.67%	17.00%	0.46%
Avg SPECint92	0.13	0.20	0.02	0.35	0.77	0.74	1.86	1.84%	13.00%	0.61%
spice	0.01	0.68	0.02	0.71	0.83	0.99	2.53	0.21%	36.00%	0.43%
doduc	0.16	0.26	0.00	0.42	0.77	1.58	2.77	2.30%	14.00%	0.11%
mdljdp2	0.00	0.31	0.01	0.32	0.83	2.18	3.33	0.06%	28.00%	0.21%
wave5	0.04	0.39	0.04	0.47	0.68	0.84	1.99	0.57%	24.00%	0.89%
tomcatv	0.00	0.42	0.04	0.46	0.67	0.79	1.92	0.06%	20.00%	0.89%
ora	0.00	0.10	0.00	0.10	0.72	1.25	2.07	0.05%	7.00%	0.10%
alvinn	0.03	0.49	0.00	0.52	0.62	0.25	1.39	0.38%	18.00%	0.01%
ear	0.01	0.15	0.00	0.16	0.65	0.24	1.05	0.11%	9.00%	0.01%
mdljsp2	0.00	0.09	0.00	0.09	0.80	1.67	2.56	0.05%	5.00%	0.11%
swm256	0.00	0.24	0.01	0.25	0.68	0.37	1.30	0.02%	13.00%	0.32%
su2cor	0.03	0.74	0.01	0.78	0.66	0.71	2.15	0.41%	43.00%	0.16%
hydro2d	0.01	0.54	0.01	0.56	0.69	1.23	2.48	0.09%	32.00%	0.32%
nasa7	0.01	0.68	0.02	0.71	0.68	0.64	2.03	0.19%	37.00%	0.25%
fpppp	0.52	0.17	0.00	0.69	0.70	0.97	2.36	7.42%	7.00%	0.01%
Avg SPECfp92	0.06	0.38	0.01	0.45	0.71	0.98	2.14	0.85%	20.93%	0.27%

FIGURE 5.48 Percentage of execution time due to memory latency and miss rates for three commercial programs and the SPEC92 benchmarks (see Chapter 1) running on the Alpha AXP 21064 in the DEC 3000 model 800. The first two commercial programs are pieces of the TP1 benchmark and the last is a sort of 100-byte records in a 100-MB database.

due to the memory hierarchy for the SPEC92 programs and three commercial programs. The three commercial programs tax the memory much more heavily, with secondary cache misses alone responsible for 20% to 28% of the execution time.

Figure 5.48 also shows the miss rates for each component. The SPECint92 programs have about a 2% instruction miss rate, a 13% data cache miss rate, and a 0.6% second-level cache miss rate. For SPECfp92 the averages are 1%, 21%, and 0.3%, respectively. The commercial workloads really exercise the memory hierarchy; the averages of the three miss rates are 6%, 32%, and 10%. Figure 5.49 shows the same data graphically. This figure makes clear that the primary performance limits of the superscalar 21064 are instruction stalls, which result from branch mispredictions, and the other category, which includes data dependencies.

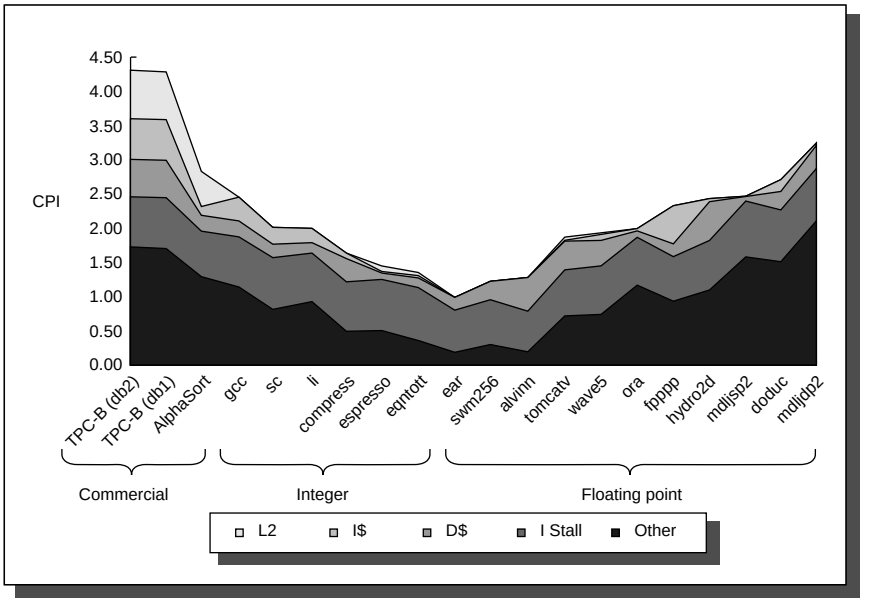

FIGURE 5.49 Graphical representation of the data in Figure 5.48, with programs in each of the three classes sorted by total CPI.

5.11 Fallacies and Pitfalls

As the most naturally quantitative of the computer architecture disciplines, memory hierarchy would seem to be less vulnerable to fallacies and pitfalls. Yet the authors were limited here not by lack of warnings, but by lack of space!

Pitfall: Too small an address space.

Just five years after DEC and Carnegie Mellon University collaborated to design the new PDP-11 computer family, it was apparent that their creation had a fatal flaw. An architecture announced by IBM six years *before* the PDP-11 was still thriving, with minor modifications, 25 years later. And the DEC VAX, criticized for including unnecessary functions, has sold 100,000 units since the PDP-11 went out of production. Why?

The fatal flaw of the PDP-11 was the size of its addresses as compared to the address sizes of the IBM 360 and the VAX. Address size limits the program length, since the size of a program and the amount of data needed by the program must be less than $2^{\text{address size}}$. The reason the address size is so hard to change is that it determines the minimum width of anything that can contain an address: PC, register, memory word, and effective-address arithmetic. If there is no plan to expand the address from the start, then the chances of successfully changing address size are so slim that it normally means the end of that computer family. Bell and Strecker [1976] put it like this:

There is only one mistake that can be made in computer design that is difficult to recover from—not having enough address bits for memory addressing and memory management. The PDP-11 followed the unbroken tradition of nearly every known computer. [p. 2]

A partial list of successful machines that eventually starved to death for lack of address bits includes the PDP-8, PDP-10, PDP-11, Intel 8080, Intel 8086, Intel 80186, Intel 80286, Motorola AMI 6502, Zilog Z80, CRAY-1, and CRAY X-MP. A few companies already offer computers with 64-bit flat addresses, and the authors expect that the rest of the industry will offer 64-bit address machines before the third edition of this book!

Fallacy: Predicting cache performance of one program from another.

Figure 5.50 shows the instruction miss rates and data miss rates for three programs from the SPEC92 benchmark suite as cache size varies. Depending on the program, the data miss rate for a direct-mapped 4-KB cache is either 28%, 12%, or 8%, and the instruction miss rate for a direct-mapped 1-KB cache is either 10%, 3%, or 0%. Figure 5.48 on page 465 shows that commercial programs such as databases will have significant miss rates even in a 2-MB second-level cache, which is not the case for the SPEC92 programs. Clearly it is not safe to generalize cache performance from one of these programs to another.

Nor is it safe to generalize cache measurements from one architecture to another. Figure 5.48 for the DEC Alpha with 8-KB caches running gcc shows miss rates of 17% for data and 4.67% for instructions, yet the DEC MIPS machine running the same program and cache size measured in Figure 5.48 suggests 10% for data and 4% for instructions.

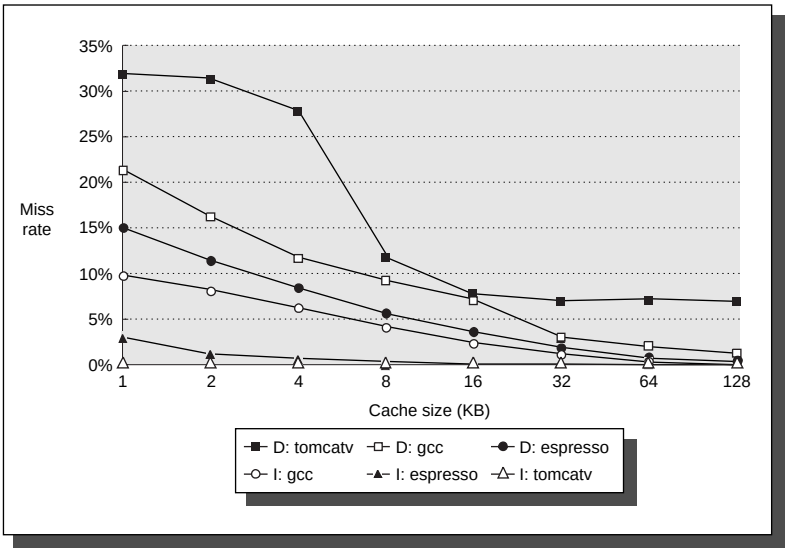

FIGURE 5.50 Instruction and data miss rates for direct-mapped caches with 32-byte blocks for running three programs for DEC 5000 as cache size varies from 1 KB to 128 KB. The programs espresso, gcc, and tomcatv are from the SPEC92 benchmark suite.

Pitfall: Simulating enough instructions to get accurate performance measures of the memory hierarchy.

There are really two pitfalls here. One is trying to predict performance of a large cache using a small trace, and the other is that a program's locality behavior is not constant over the run of the entire program. Figure 5.51 shows the cumulative average memory access time for four programs over the execution of billions of instructions. For these programs, the average memory access times for the first billion instructions executed is very different from their average memory access times for the second billion. While two of the programs need to execute half of the total number of instructions to get a good estimate of the average memory access time, SOR needs to get to the three-quarters mark, and TV needs to finish completely before the accurate measure appears.

The first edition of this book included another example of this pitfall. The compulsory miss ratios were erroneously high (e.g., 1%) because of tracing too few memory accesses. A program with an infinite cache miss ratio of 1% running on a machine accessing memory 10 million times per second would touch hundreds of megabytes of new memory every minute:

$$\frac{10{,}000{,}000 \text{ accesses}}{\text{Second}} \times \frac{0.01 \text{ misses}}{\text{Access}} \times \frac{32 \text{ bytes}}{\text{Miss}} \times \frac{60 \text{ seconds}}{\text{Minute}} = \frac{192{,}000{,}000 \text{ bytes}}{\text{Minute}}$$

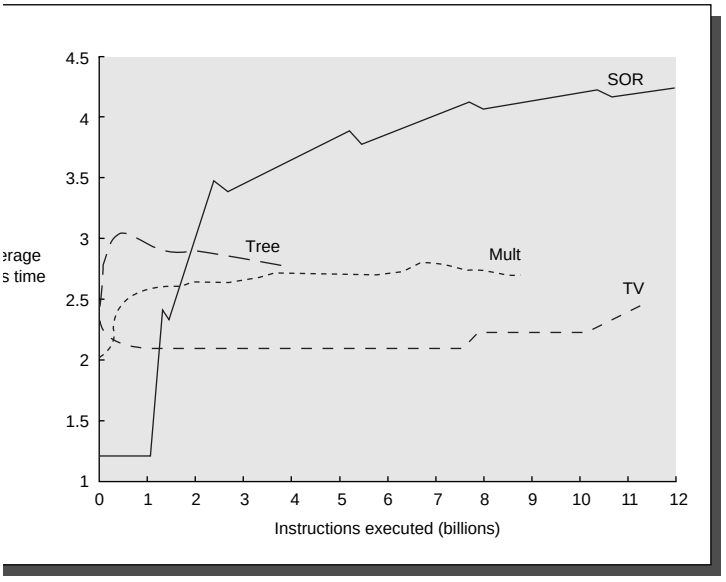

FIGURE 5.51 Average memory access times for four programs over execution time of billions of instructions. The assumed memory hierarchy was a 4-KB instruction cache and 4-KB data cache with 16-byte blocks, and a 512-KB second-level cache with 128-byte blocks using the Titan RISC instruction set. The first-level data cache is write through with a four-entry write buffer, and the second-level cache is write back. The miss penalty for the first-level cache to second-level cache is 12 clock cycles, and the miss penalty from the second-level cache to main memory is 200 clock cycles. SOR is a FORTRAN program for successive over-relaxation, Tree is a Scheme program that builds and searches a tree, Mult is a multi-programmed workload consisting of six smaller programs, and TV is a Pascal program for timing verification of VLSI circuits. (This figure taken from Figure 3-5 on page 276 of the paper by Borg, Kessler, and Wall [1990].)

Data on typical page fault rates and process sizes do not support the conclusion that memory is touched at this rate.

Pitfall: Ignoring the impact of the operating system on the performance of the memory hierarchy.

Figure 5.52 shows the memory stall time due to the operating system spent on three large workloads. About 25% of the stall time is either spent in misses in the operating system or results from misses in the application programs because of interference with the operating system.

	Misses		Time						
			% time due to appl. misses	% time due directly to OS misses					% time OS misses & appl. conflicts
Workload	% in appl	% in OS	Inherent appl. misses	OS conflicts w. appl.	OS instr misses	Data misses for migration	Data misses in block operations	Rest of OS misses	
Pmake	47%	53%	14.1%	4.8%	10.9%	1.0%	6.2%	2.9%	25.8%
Multipgm	53%	47%	21.6%	3.4%	9.2%	4.2%	4.7%	3.4%	24.9%
Oracle	73%	27%	25.7%	10.2%	10.6%	2.6%	0.6%	2.8%	26.8%

FIGURE 5.52 Misses and time spent in misses for applications and operating system. Collected on Silicon Graphics POWER station 4D/340, a multiprocessor with four 33-MHz R3000 CPUs running three application workloads under a UNIX System V—Pmake: a parallel compile of 56 files; Multipgm: the parallel numeric program MP3D running concurrently with Pmake and five-screen edit session; and Oracle: running a restricted version of the TP-1 benchmark using the Oracle database. Each CPU has a 64-KB instruction cache and a two-level data cache with 64 KB in the first level and 256 KB in the second level; all caches are direct mapped with 16-byte blocks. Data from Torrellas, Gupta, and Hennessy [1992].

Pitfall: Basing the size of the write buffer on the speed of memory and the average mix of writes.

This seems like a reasonable approach:

$$\text{Write buffer size} = \frac{\text{Memory references}}{\text{Clock cycle}} \times \text{Write percentage} \times \text{Clock cycles to write memory}$$

If there is one memory reference per clock cycle, 10% of the memory references are writes, and writing a word of memory takes 10 cycles, then a one-word buffer is added ($1 \times 10\% \times 10 = 1$). Calculating for the Alpha AXP 21064,

$$\text{Write buffer size} = \frac{1.36 \text{ memory references}}{2.0 \text{ clock cycles}} \times 0.1 \text{ writes} \times \frac{15 \text{ clock cycles}}{\text{Write}} = 1.0$$

Thus, a one-word buffer seems sufficient.

The pitfall is that when writes come close together, the CPU must stall until the prior write is completed. Hence the calculation above says that a one-word buffer would be utilized 100% of the time. Queuing theory tells us if utilization is close to 100%, then writes will normally stall the CPU.

The proper question to ask is how large a buffer is needed to keep utilization low so that the buffer rarely fills, thereby keeping CPU write stall time low. The impact of write buffer size can be established by simulation or estimated with a queuing model.

5.12 Concluding Remarks

The difficulty of building a memory system to keep pace with faster CPUs is underscored by the fact that the raw material for main memory is the same as that found in the cheapest computer. It is the principle of locality that saves us here—its soundness is demonstrated at all levels of the memory hierarchy in current computers, from disks to TLBs. Figure 5.53 summarizes the attributes of the memory-hierarchy examples described in this chapter.

	TLB	First-level cache	Second-level cache	Virtual memory
Block size	4–8 bytes (1 PTE)	4–32 bytes	32–256 bytes	4096–16,384 bytes
Hit time	1 clock cycle	1–2 clock cycles	6–15 clock cycles	10–100 clock cycles
Miss penalty	10–30 clock cycles	8–66 clock cycles	30–200 clock cycles	700,000–6,000,000 clock cycles
Miss rate (local)	0.1–2%	0.5–20%	15–30%	0.00001–0.001%
Size	32–8192 bytes (8–1024 PTEs)	1–128 KB	256 KB–16 MB	16–8192 MB
Backing store	First-level cache	Second-level cache	Page-mode DRAM	Disks
Q1: block placement	Fully associative or set associative	Direct mapped	Direct mapped or set associative	Fully associative
Q2: block identification	Tag/block	Tag/block	Tag/block	Table
Q3: block replacement	Random	N.A. (direct mapped)	Random	≈ LRU
Q4: write strategy	Flush on a write to page table	Write through or write back	Write back	Write back

FIGURE 5.53 Summary of the memory-hierarchy examples in this chapter.

Yet the design decisions at these levels interact, and the architect must take the whole system view to make wise decisions. The primary challenge for the memory-hierarchy designer is in choosing parameters that work well together, not in inventing new techniques. The increasingly fast CPUs are spending a larger fraction of time waiting for memory, which has led to new inventions that have increased the number of choices: variable page size, pseudo-associative caches, and cache-aware compilers weren't found in the first edition of this book. Fortunately, there tends to be a technological "sweet spot" in balancing cost, performance, and complexity: missing the target wastes performance, hardware, design time, debug time, or possibly all four. Architects hit the target by careful, quantitative analysis.

5.13 Historical Perspective and References

While the pioneers of computing knew of the need for a memory hierarchy and coined the term, the automatic management of two levels was first proposed by Kilburn et al. [1962] and demonstrated with the Atlas computer at the University of Manchester. This was the year *before* the IBM 360 was announced. While IBM planned for its introduction with the next generation (System/370), the operating system TSS wasn't up to the challenge in 1970. Virtual memory was announced for the 370 family in 1972, and it was for this machine that the term "translation look-aside buffer" was coined [Case and Padegs 1978]. The only computers today without virtual memory are a few supercomputers, embedded processors, and older personal computers.

Both the Atlas and the IBM 360 provided protection on pages, and the GE 645 was the first system to provide paged segmentation. The Intel 80286, the first 80x86 to have the protection mechanisms described on pages 453 to 457, was inspired by the Multics protection software that ran on the GE 645. Over time, machines evolved more elaborate mechanisms. The most elaborate mechanism was *capabilities*, which reached its highest interest in the late 1970s and early 1980s [Fabry 1974; Wulf, Levin, and Harbison 1981]. Wilkes [1982], one of the early workers on capabilities, had this to say:

Anyone who has been concerned with an implementation of the type just described [capability system], or has tried to explain one to others, is likely to feel that complexity has got out of hand. It is particularly disappointing that the attractive idea of capabilities being tickets that can be freely handed around has become lost ...

Compared with a conventional computer system, there will inevitably be a cost to be met in providing a system in which the domains of protection are small and frequently changed. This cost will manifest itself in terms of additional hardware, decreased runtime speed, and increased memory occupancy. It is at present an open question whether, by adoption of the capability approach, the cost can be reduced to reasonable proportions.

Today there is little interest in capabilities either from the operating systems or the computer architecture communities, although there is growing interest in protection and security.

Bell and Strecker [1976] reflected on the PDP-11 and identified a small address space as the only architectural mistake that is difficult to recover from. At the time of the creation of PDP-11, core memories were increasing at a very slow rate, and the competition from 100 other minicomputer companies meant that DEC might not have a cost-competitive product if every address had to go

through the 16-bit datapath twice, hence the architect's decision to add just 4 more address bits than the predecessor of the PDP-11. The architects of the IBM 360 were aware of the importance of address size and planned for the architecture to extend to 32 bits of address. Only 24 bits were used in the IBM 360, however, because the low-end 360 models would have been even slower with the larger addresses in 1964. Unfortunately, the architects didn't reveal their plans to the software people, and the expansion effort was foiled by programmers who stored extra information in the upper 8 "unused" address bits. Virtually every machine since then, including the Alpha AXP, will check to make sure the unused bits stay unused, and trap if the bits have the wrong value.

A few years after the Atlas paper, Wilkes published the first paper describing the concept of a cache [1965]:

The use is discussed of a fast core memory of, say, 32,000 words as slave to a slower core memory of, say, one million words in such a way that in practical cases the effective access time is nearer that of the fast memory than that of the slow memory. [p. 270]

This two-page paper describes a direct-mapped cache. While this is the first publication on caches, the first implementation was probably a direct-mapped instruction cache built at the University of Cambridge. It was based on tunnel diode memory, the fastest form of memory available at the time. Wilkes states that G. Scarott suggested the idea of a cache memory.

Subsequent to that publication, IBM started a project that led to the first commercial machine with a cache, the IBM 360/85 [Liptay 1968]. Gibson [1967] describes how to measure program behavior as memory traffic as well as miss rate and shows how the miss rate varies between programs. Using a sample of 20 programs (each with 3 million references!), Gibson also relied on average memory access time to compare systems with and without caches. This was over 25 years ago, and yet many used miss rates until recently.

Conti, Gibson, and Pitkowsky [1968] describe the resulting performance of the 360/85. The 360/91 outperforms the 360/85 on only 3 of the 11 programs in the paper, even though the 360/85 has a slower clock cycle time (80 ns versus 60 ns), smaller memory interleaving (4 versus 16), and a slower main memory (1.04 μsec versus 0.75 μsec). This paper was also the first to use the term "cache." Strecker [1976] published the first comparative cache design paper examining caches for the PDP-11. Smith [1982] later published a thorough survey paper, using the terms "spatial locality" and "temporal locality"; this paper has served as a reference for many computer designers. While most studies have relied on simulations, Clark [1983] used a hardware monitor to record cache misses of the VAX-11/780 over several days. Hill [1987] proposed the three C's used in section 5.3 to explain cache misses. One of the first papers on nonblocking caches is by Kroft [1981].

This chapter relies on the measurements of SPEC92 benchmarks collected by Gee et al. [1993] for DEC 5000s. There are several other papers used in this chapter that are cited in the captions of the figures that use the data: Borg, Kessler, and Wall [1990]; Farkas and Jouppi [1994]; Jouppi [1990]; Lam, Rothberg, and Wolf [1991]; Mowry, Lam, and Gupta [1992]; Lebeck and Wood [1994]; and Torrellas, Gupta, and Hennessy [1992]. For more details on prime numbers of memory modules, read Gao [1993]; for more on pseudo-associative caches, see Agarwal and Pudar [1993]. Caches remain an active area of research.

The Alpha AXP architecture is described in detail by Bhandarkar [1995] and by Sites [1992], and a good source of data on implementations is the *Digital Technical Journal*, issue no. 4 of 1992, which is dedicated to articles on Alpha.

References

AGARWAL, A. [1987]. *Analysis of Cache Performance for Operating Systems and Multiprogramming*, Ph.D. Thesis, Stanford Univ., Tech. Rep. No. CSL-TR-87-332 (May).

AGARWAL, A. AND S. D. PUDAR [1993]. "Column-associative caches: A technique for reducing the miss rate of direct-mapped caches," 20th Annual Int'l Symposium on Computer Architecture ISCA '20, San Diego, Calif., May 16–19. *Computer Architecture News* 21:2 (May), 179–90.

BAER, J.-L. AND W.-H. WANG [1988]. "On the inclusion property for multi-level cache hierarchies," *Proc. 15th Annual Symposium on Computer Architecture* (May–June), Honolulu, 73–80.

BELL, C. G. AND W. D. STRECKER [1976]. "Computer structures: What have we learned from the PDP-11?," *Proc. Third Annual Symposium on Computer Architecture* (January), Pittsburgh, 1–14.

BHANDARKAR, D. P. [1995]. *Alpha Architecture Implementations*, Digital Press, Newton, Mass.

BORG, A., R. E. KESSLER, AND D. W. WALL [1990]. "Generation and analysis of very long address traces," *Proc. 17th Annual Int'l Symposium on Computer Architecture* (Cat. No. 90CH2887–8), Seattle, May 28–31, IEEE Computer Society Press, Los Alamitos, Calif., 270–9.

CASE, R. P. AND A. PADEGS [1978]. "The architecture of the IBM System/370," *Communications of the ACM* 21:1, 73–96. Also appears in D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples* (1982), McGraw-Hill, New York, 830–855.

CLARK, D. W. [1983]. "Cache performance of the VAX-11/780," *ACM Trans. on Computer Systems* 1:1, 24–37.

CONTI, C., D. H. GIBSON, AND S. H. PITKOWSKY [1968]. "Structural aspects of the System/360 Model 85, Part I: General organization," *IBM Systems J.* 7:1, 2–14.

CRAWFORD, J. H. AND P. P. GELSINGER [1987]. *Programming the 80386*, Sybex, Alameda, Calif.

FABRY, R. S. [1974]. "Capability based addressing," *Comm. ACM* 17:7 (July), 403–412.

FARKAS, K. I. AND N. P. JOUPPI [1994]. "Complexity/performance tradeoffs with non-blocking loads," *Proc. 21st Annual Int'l Symposium on Computer Architecture*, Chicago (April).

GAO, Q. S. [1993]. "The Chinese remainder theorem and the prime memory system," 20th Annual Int'l Symposium on Computer Architecture ISCA '20, San Diego, May 16–19, 1993. *Computer Architecture News* 21:2 (May), 337–40.

GEE, J. D., M. D. HILL, D. N. PNEVMATIKATOS, AND A. J. SMITH [1993]. "Cache performance of the SPEC92 benchmark suite," *IEEE Micro* 13:4 (August), 17–27.

GIBSON, D. H. [1967]. "Considerations in block-oriented systems design," *AFIPS Conf. Proc.* 30, SJCC, 75–80.

HANDY, J. [1993]. *The Cache Memory Book*, Academic Press, Boston.

HILL, M. D. [1987]. *Aspects of Cache Memory and Instruction Buffer Performance*, Ph.D. Thesis, University of Calif. at Berkeley, Computer Science Division, Tech. Rep. UCB/CSD 87/381 (November).

HILL, M. D. [1988]. "A case for direct mapped caches," *Computer* 21:12 (December), 25–40.

JOUPPI, N. P. [1990]. "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," *Proc. 17th Annual Int'l Symposium on Computer Architecture* (Cat. No. 90CH2887–8), Seattle, May 28–31, 1990. IEEE Computer Society Press, Los Alamitos, Calif., 364–73.

KILBURN, T., D. B. G. EDWARDS, M. J. LANIGAN, AND F. H. SUMNER [1962]. "One-level storage system," *IRE Trans. on Electronic Computers* EC-11 (April) 223–235. Also appears in D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples* (1982), McGraw-Hill, New York, 135–148.

KROFT, D. [1981]. "Lockup-free instruction fetch/prefetch cache organization," *Proc. Eighth Annual Symposium on Computer Architecture* (May 12–14), Minneapolis, 81–87.

LAM, M. S., E. E. ROTHBERG, AND M. E. WOLF [1991]. "The cache performance and optimizations of blocked algorithms," Fourth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems, Santa Clara, Calif., April 8–11. *SIGPLAN Notices* 26:4 (April), 63–74.

LEBECK, A. R. AND D. A. WOOD [1994]. "Cache profiling and the SPEC benchmarks: A case study," *Computer* 27:10 (October), 15–26.

LIPTAY, J. S. [1968]. "Structural aspects of the System/360 Model 85, Part II: The cache," *IBM Systems J.* 7:1, 15–21.

MCFARLING, S. [1989]. "Program optimization for instruction caches," *Proc. Third Int'l Conf. on Architectural Support for Programming Languages and Operating Systems* (April 3–6), Boston, 183–191.

MOWRY, T. C., S. LAM, AND A. GUPTA [1992]. "Design and evaluation of a compiler algorithm for prefetching," Fifth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V), Boston, October 12–15 , *SIGPLAN Notices* 27:9 (September), 62–73.

PALACHARLA, S. AND R. E. KESSLER [1994]. "Evaluating stream buffers as a secondary cache replacement," *Proc. 21st Annual Int'l Symposium on Computer Architecture*, Chicago, April 18–21, IEEE Computer Society Press, Los Alamitos, Calif., 24–33.

PRZYBYLSKI, S. A. [1990]. *Cache Design: A Performance-Directed Approach*, Morgan Kaufmann Publishers, San Mateo, Calif.

PRZYBYLSKI, S. A., M. HOROWITZ, AND J. L. HENNESSY [1988]. "Performance tradeoffs in cache design," *Proc. 15th Annual Symposium on Computer Architecture* (May–June), Honolulu, 290–298.

SAAVEDRA-BARRERA, R. H. [1992]. *CPU Performance Evaluation and Execution Time Prediction Using Narrow Spectrum Benchmarking*, Ph.D. Dissertation, University of Calif., Berkeley (May).

SAMPLES, A. D. AND P. N. HILFINGER [1988]. "Code reorganization for instruction caches," Tech. Rep. UCB/CSD 88/447 (October), University of Calif., Berkeley.

SITES, R. L. (ED.) [1992]. *Alpha Architecture Reference Manual*, Digital Press, Burlington, Mass.

SMITH, A. J. [1982]. "Cache memories," *Computing Surveys* 14:3 (September), 473–530.

SMITH, J. E. AND J. R. GOODMAN [1983]. "A study of instruction cache organizations and replacement policies," *Proc. 10th Annual Symposium on Computer Architecture* (June 5–7), Stockholm, 132–137.

STRECKER, W. D. [1976]. "Cache memories for the PDP-11?," *Proc. Third Annual Symposium on Computer Architecture* (January), Pittsburgh, 155–158.

TORRELLAS, J., A. GUPTA, AND J. HENNESSY [1992]. "Characterizing the caching and synchronization performance of a multiprocessor operating system," Fifth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V), Boston, October 12–15, *SIGPLAN Notices* 27:9 (September), 162–174.

WANG, W.-H., J.-L. BAER, AND H. M. LEVY [1989]. "Organization and performance of a two-level virtual-real cache hierarchy," *Proc. 16th Annual Symposium on Computer Architecture* (May 28–June 1), Jerusalem, 140–148.

WILKES, M. [1965]. "Slave memories and dynamic storage allocation," *IEEE Trans. Electronic Computers* EC-14:2 (April), 270–271.

WILKES, M. V. [1982]. "Hardware support for memory protection: Capability implementations," *Proc. Symposium on Architectural Support for Programming Languages and Operating Systems* (March 1–3), Palo Alto, Calif., 107–116.

WULF, W. A., R. LEVIN, AND S. P. HARBISON [1981]. *Hydra/C.mmp: An Experimental Computer System*, McGraw-Hill, New York.

EXERCISES

5.1 [15/15/12/12] <5.1,5.2> Let's try to show how you can make *unfair* benchmarks. Here are two machines with the same processor and main memory but different cache organizations. Assume the miss time is 10 times a cache hit time for both machines. Assume writing a 32-bit word takes 5 times as long as a cache hit (for the write-through cache) and that writing a whole 32-byte block takes 10 times as long as a cache-read hit (for the write-back cache). The caches are unified; that is, they contain both instructions and data.

Cache A: 128 sets, two elements per set, each block is 32 bytes, and it uses write through and no-write allocate.

Cache B: 256 sets, one element per set, each block is 32 bytes, and it uses write back and does allocate on write misses.

a. [15] <1.5,5.2> Describe a program that makes machine A run as much faster as possible than machine B. (Be sure to state any further assumptions you need, if any.)

b. [15] <1.5,5.2> Describe a program that makes machine B run as much faster as possible than machine A. (Be sure to state any further assumptions you need, if any.)

c. [12] <1.5,5.2> Approximately how much faster is the program in part (a) on machine A than machine B?

d. [12] <1.5,5.2> Approximately how much faster is the program in part (b) on machine B than on machine A?

5.2 [15/10/12/12/12/12/12/12/12/12/12] <5.3,5.4> In this exercise, we will run a program to evaluate the behavior of a memory system. The key is having accurate timing and then having the program stride through memory to invoke different levels of the hierarchy. Below is the code in C for UNIX systems. The first part is a procedure that uses a standard UNIX utility to get an accurate measure of the user CPU time; this procedure may need to change to work on some systems. The second part is a nested loop to read and write memory at different strides and cache sizes. To get accurate cache timing, this code is repeated many times. The third part times the nested loop overhead only so that it can be subtracted from overall measured times to see how long the accesses were. The last part prints the time per access as the size and stride varies. You may need to change `CACHE_MAX` depending on the

question you are answering and the size of memory on the system you are measuring. The code below was taken from a program written by Andrea Dusseau of U.C. Berkeley, and was based on a detailed description found in Saavedra-Barrera [1992].

```
#include <stdio.h>
#include <sys/times.h>
#include <sys/types.h>
#include <time.h>
#define CACHE_MIN (1024) /* smallest cache */
#define CACHE_MAX (1024*1024) /* largest cache */
#define SAMPLE 10 /* to get a larger time sample */
#ifndef CLK_TCK
#define CLK_TCK 60 /* number clock ticks per second */
#endif
int x[CACHE_MAX]; /* array going to stride through */

double get_seconds() { /* routine to read time */
    struct tms rusage;
    times(&rusage); /* UNIX utility: time in clock ticks */
    return (double) (rusage.tms_utime)/CLK_TCK;
}
void main() {
int register i, index, stride, limit, temp;
int steps, tsteps, csize;
double sec0, sec; /* timing variables */

for (csize=CACHE_MIN; csize <= CACHE_MAX; csize=csize*2)
    for (stride=1; stride <= csize/2; stride=stride*2) {
        sec = 0; /* initialize timer */
        limit = csize-stride+1; /* cache size this loop */

        steps = 0;
            do { /* repeat until collect 1 second */
        sec0 = get_seconds(); /* start timer */
        for (i=SAMPLE*stride;i!=0;i=i-1) /* larger sample */
          for (index=0; index < limit; index=index+stride)
                x[index] = x[index] + 1; /* cache access */
        steps = steps + 1; /* count while loop iterations */
        sec = sec + (get_seconds() - sec0);/* end timer */
        } while (sec < 1.0); /* until collect 1 second */

        /* Repeat empty loop to subtract loop overhead */
        tsteps = 0; /* used to match no. while iterations */
            do { /* repeat until same no. iterations as above */
        sec0 = get_seconds(); /* start timer */
        for (i=SAMPLE*stride;i!=0;i=i-1) /* larger sample */
          for (index=0; index < limit; index=index+stride)
                temp = temp + index; /* dummy code */
        tsteps = tsteps + 1; /* count while iterations */
        sec = sec - (get_seconds() - sec0);/* - overhead */
        } while (tsteps<steps); /* until = no. iterations */

        printf("Size:%7d Stride:%7d read+write:%l4.0f ns\n",
            csize*sizeof(int), stride*sizeof(int), (double)
        sec*1e9/(steps*SAMPLE*stride*((limit-1)/stride+1)));
    }; /* end of both outer for loops */
}
```

The program above assumes that program addresses track physical addresses, which is true on the few machines that use virtually addressed caches. In general, virtual addresses tend to follow physical addresses shortly after rebooting, so you may need to reboot the machine in order to get smooth lines in your results.

To answer the questions below, assume that the sizes of all components of the memory hierarchy are powers of 2.

a. [15] <5.3,5.4> Plot the experimental results with elapsed time on the y-axis and the memory stride on the x-axis. Use logarithmic scales for both axes, and draw a line for each cache size.

b. [10] <5.3,5.4> How many levels of cache are there?

c. [12] <5.3,5.4> What is the size of the first-level cache? Block size? *Hint*: Assume the size of the page is much larger than the size of a block in a secondary cache (if any), and the size of a second-level cache block is greater than or equal to the size of a block in a first-level cache.

d. [12] <5.3,5.4> What is the size of the second-level cache (if any)? Block size?

e. [12] <5.3,5.4> What is the associativity of the first-level cache? Second-level cache?

f. [12] <5.3,5.4> What is the page size?

g. [12] <5.3,5.4> How many entries are in the TLB?

h. [12] <5.3,5.4> What is the miss penalty for the first-level cache? Second-level?

i. [12] <5.3,5.4> What is the time for a page fault to secondary memory? *Hint*: A page fault to magnetic disk should be measured in milliseconds.

j. [12] <5.3,5.4> What is the miss penalty for the TLB?

k. [12] <5.3,5.4> Is there anything else you have discovered about the memory hierarchy from these measurements?

5.3 [10/10/10] <5.2> Figure 5.54 shows the output from running the program in Exercise 5.2 on a SPARCstation 1+, which has a single unified cache.

a. [10] <5.2> What is the size of the cache?

b. [10] <5.2> What is the block size of the cache?

c. [10] <5.2> What is the miss penalty for the first-level cache?

5.4 [15/15] <5.2> You purchased an Acme computer with the following features:

- 95% of all memory accesses are found in the cache.
- Each cache block is two words, and the whole block is read on any miss.
- The processor sends references to its cache at the rate of 10^9 words per second.
- 25% of those references are writes.
- Assume that the memory system can support 10^9 words per second, reads or writes.
- The bus reads or writes a single word at a time (the memory system cannot read or write two words at once).

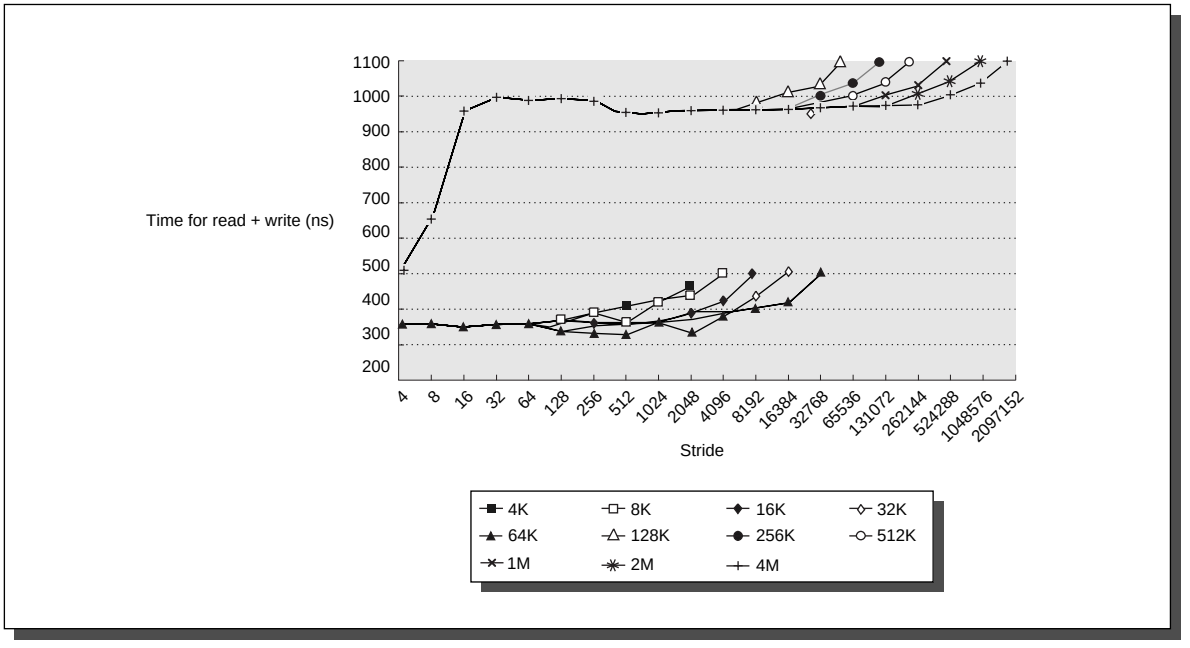

FIGURE 5.54 Results of running program in Exercise 5.2 on a SPARCstation 1+.

- Assume at any one time, 30% of the blocks in the cache have been modified.
- The cache uses write allocate on a write miss.

You are considering adding a peripheral to the system, and you want to know how much of the memory system bandwidth is already used. Calculate the percentage of memory system bandwidth used on the average in the two cases below. Be sure to state your assumptions.

a. [15] <5.2> The cache is write through.

b. [15] <5.2> The cache is write back.

5.5 [15/15] <5.5> One difference between a write-through cache and a write-back cache can be in the time it takes to write. During the first cycle, we detect whether a hit will occur, and during the second (assuming a hit) we actually write the data. Let's assume that 50% of the blocks are dirty for a write-back cache. For this question, assume that the write buffer for write through will never stall the CPU (no penalty). Assume a cache read hit takes 1 clock cycle, the cache miss penalty is 50 clock cycles, and a block write from the cache to main memory takes 50 clock cycles. Finally, assume the instruction cache miss rate is 0.5% and the data cache miss rate is 1%.

a. [15] <5.5> Using statistics for the average percentage of loads and stores from DLX in Figure 2.26 on page 105, estimate the performance of a write-through cache with a two-cycle write versus a write-back cache with a two-cycle write for each of the programs.

b. [15] <5.5> Do the same comparison, but this time assume the write-through cache pipelines the writes, as described on page 425, so that a write hit takes just one clock cycle.

5.6 [20] <5.3> Improve on the compiler prefetch Example found on page 401: Try to eliminate both the number of extraneous prefetches and the number of non-prefetched cache misses. Calculate the performance of this refined version using the parameters in the Example.

5.7 [15/12] <5.3> The Example evaluation of a pseudo-associative cache on page 399 assumed that on a hit to the slower block the hardware swapped the contents with the corresponding fast block so that subsequent hits on this address would all be to the fast block. Assume that if we don't swap, a hit in the slower block takes just one extra clock cycle instead of two extra clock cycles.

a. [15] <5.3> Derive a formula for the average memory access time using the terminology for direct-mapped and two-way set-associative caches as given on page 399.

b. [12] <5.3> Using the formula from part (a), recalculate the average memory access times for the two cases found on page 399 (2-KB cache and 128-KB cache). Which pseudo-associative scheme is faster for the given configurations and data?

5.8 [15/20/15] <5.7> If the base CPI with a perfect memory system is 1.5, what is the CPI for these cache organizations? Use Figure 5.9 (page 391):

- 16-KB direct-mapped unified cache using write back.
- 16-KB two-way set-associative unified cache using write back.
- 32-KB direct-mapped unified cache using write back.

Assume the memory latency is 40 clocks, the transfer rate is 4 bytes per clock cycle and that 50% of the transfers are dirty. There are 32 bytes per block and 20% of the instructions are data transfer instructions. There is no write buffer. Add to the assumptions above a TLB that takes 20 clock cycles on a TLB miss. A TLB does not slow down a cache hit. For the TLB, make the simplifying assumption that 0.2% of all references aren't found in TLB, either when addresses come directly from the CPU or when addresses come from cache misses.

a. [15] <5.3> Compute the effective CPI for the three caches assuming an ideal TLB.

b. [20] <5.3> Using the results from part (a), compute the effective CPI for the three caches with a real TLB.

c. [15] <5.3> What is the impact on performance of a TLB if the caches are virtually or physically addressed?

5.9 [10] <5.4> What is the formula for average access time for a three-level cache?

5.10 [15/15] <5.6> The section on avoiding bank conflicts by having a prime number of memory banks mentioned that there are techniques for fast modulo arithmetic, especially when the prime number can be represented as $2^N - 1$. The idea is that by understanding the laws of modulo arithmetic we can simplify the hardware. The key insights are the following:

1. Modulo arithmetic obeys the laws of distribution:

   $((a \text{ modulo } c) + (b \text{ modulo } c)) \text{ modulo } c = (a + b) \text{ modulo } c$
   $((a \text{ modulo } c) \times (b \text{ modulo } c)) \text{ modulo } c = (a \times b) \text{ modulo } c$

2. The sequence 2^0 modulo $2^N - 1$, 2^1 modulo $2^N - 1$, 2^2 modulo $2^N - 1$, . . . is a repeating pattern 2^0, 2^1, 2^2, and so on for powers of 2 less than 2^N. For example, if $2^N - 1 = 7$, then

   2^0 modulo 7 = 1
   2^1 modulo 7 = 2
   2^2 modulo 7 = 4
   2^3 modulo 7 = 1
   2^4 modulo 7 = 2
   2^5 modulo 7 = 4

3. Given a binary number a, the value of (a mod 7) can be expressed as

   $a_i \times 2^i + \ldots + a_2 \times 2^2 + a_1 \times 2^1 + a_0 \times 2^0$ modulo 7 =
   $((a_0 + a_3 + \ldots) \times 1 + (a_1 + a_4 + \ldots) \times 2 + (a_2 + a_5 + \ldots) \times 4)$ modulo 7

   where $i = \log_2 a$ and $a_j = 0$ for $j > i$

   This is possible because 7 is a prime number of the form $2^N - 1$. Since the multiplications in the expression above are by powers of two, they can be replaced by binary shifts (a very fast operation).

4. The address is now small enough to find the modulo by looking it up in a read-only memory (ROM) to get the bank number.

Finally, we are ready for the questions.

a. [15] <5.6> Given $2^N - 1$ memory banks, what is the approximate reduction in size of an address that is M bits wide as a result of the intermediate result in step 3 above? Give the general formula, and then show the specific case of $N = 3$ and $M = 32$.

b. [15] <5.6> Draw the block structure of the hardware that would pick the correct bank out of seven banks given a 32-bit address. Assume that each bank is 8 bytes wide. What is the size of the adders and ROM used in this organization?

5.11 [25/10/15] <5.6> The CRAY X-MP instruction buffers can be thought of as an instruction-only cache. The total size is 1 KB, broken into four blocks of 256 bytes per block. The cache is fully associative and uses a first-in, first-out replacement policy. The access time on a miss is 10 clock cycles, with the transfer time of 64 bytes every clock cycle. The X-MP takes 1 clock cycle on a hit. Use the cache simulator to determine the following:

a. [25] <5.6> Instruction miss rate.

b. [10] <5.6> Average instruction memory access time measured in clock cycles.

c. [15] <5.6> What does the CPI of the CRAY X-MP have to be for the portion due to instruction cache misses to be 10% or less?

5.12 [25] <5.6> Traces from a single process give too high estimates for caches used in a multiprocess environment. Write a program that merges the uniprocess DLX traces into a single reference stream. Use the process-switch statistics in Figure 5.26 (page 423) as the average process-switch rate with an exponential distribution about that mean. (Use the number of clock cycles rather than instructions, and assume the CPI of DLX is 1.5.) Use the cache simulator on the original traces and the merged trace. What is the miss rate for each, assuming a 64-KB direct-mapped cache with 16-byte blocks? (There is a process-identified tag in the cache tag so that the cache doesn't have to be flushed on each switch.)

5.13 [25] <5.6> One approach to reducing misses is to prefetch the next block. A simple but effective strategy, found in the Alpha 21064, is when block *i* is referenced to make sure block *i* + 1 is in the cache, and if not, to prefetch it. Do you think automatic prefetching is more or less effective with increasing block size? Why? Is it more or less effective with increasing cache size? Why? Use statistics from the cache simulator and the traces to support your conclusion.

5.14 [20/25] <5.6> Smith and Goodman [1983] found that for a *small instruction* cache, a cache using direct mapping could consistently outperform one using fully associative with LRU replacement.

a. [20] <5.6> Explain why this would be possible. (*Hint*: You can't explain this with the three C's model because it ignores replacement policy.)

b. [25] <5.6> Use the cache simulator to see if their results hold for the traces.

5.15 [30] <5.7> Use the cache simulator and traces to calculate the effectiveness of a four-bank versus eight-bank interleaved memory. Assume each word transfer takes one clock on the bus and a random access is eight clocks. Measure the bank conflicts and memory bandwidth for these cases:

a. <5.7> No cache and no write buffer.

b. <5.7> A 64-KB direct-mapped write-through cache with four-word blocks.

c. <5.7> A 64-KB direct-mapped write-back cache with four-word blocks.

d. <5.7> A 64-KB direct-mapped write-through cache with four-word blocks but the "interleaving" comes from a page-mode DRAM.

e. <5.7> A 64-KB direct-mapped write-back cache with four-word blocks but the "interleaving" comes from a page-mode DRAM.

5.16 [25/25/25] <5.7> Use a cache simulator and traces to calculate the effectiveness of early restart and out-of-order fetch. What is the distribution of first accesses to a block as block size increases from 2 words to 64 words by factors of two for the following:

a. [25] <5.7> A 64-KB instruction-only cache?

b. [25] <5.7> A 64-KB data-only cache?

c. [25] <5.7> A 128-KB unified cache?

Assume direct-mapped placement.

5.17 [25/25/25/25/25/25] <5.2> Use a cache simulator and traces with a program you write yourself to compare the effectiveness of these schemes for fast writes:

a. [25] <5.2> One-word buffer and the CPU stalls on a data-read cache miss with a write-through cache.

b. [25] <5.2> Four-word buffer and the CPU stalls on a data-read cache miss with a write-through cache.

c. [25] <5.2> Four-word buffer and the CPU stalls on a data-read cache miss only if there is a potential conflict in the addresses with a write-through cache.

d. [25] <5.2> A write-back cache that writes dirty data first and then loads the missed block.

e. [25] <5.2> A write-back cache with a one-block write buffer that loads the miss data first and then stalls the CPU on a clean miss if the write buffer is not empty.

f. [25] <5.2> A write-back cache with a one-block write buffer that loads the miss data first and then stalls the CPU on a clean miss only if the write buffer is not empty and there is a potential conflict in the addresses.

Assume a 64-KB direct-mapped cache for data and a 64-KB direct-mapped cache for instructions with a block size of 32 bytes. The CPI of the CPU is 1.5 with a perfect memory system and it takes 14 clocks on a cache miss and 7 clocks to write a single word to memory.

5.18 [25] <5.4> Using the UNIX pipe facility, connect the output of one copy of the cache simulator to the input of another. Use this pair to see at what cache size the global miss rate of a second-level cache is approximately the same as a single-level cache of the same capacity for the traces provided.

5.19 [Discussion] <5.7> Second-level caches now contain several megabytes of data. Although new TLBs provide for variable length pages to try to map more memory, most operating systems do not take advantage of them. Does it make sense to miss the TLB on data that are found in a cache? How should TLBs be reorganized to avoid such misses?

5.20 [Discussion] <5.7> Some people have argued that with increasing capacity of memory storage per chip, virtual memory is an idea whose time has passed, and they expect to see it dropped from future computers. Find reasons for and against this argument.

5.21 [Discussion] <5.7> So far, few computer systems take advantage of the extra security available with gates and rings found in a CPU like the Intel Pentium. Construct some scenario whereby the computer industry would switch over to this model of protection.

5.22 [Discussion] <5.12> Many times a new technology has been invented that is expected to make a major change to the memory hierarchy. For the sake of this question, let's suppose that biological computer technology becomes a reality. Suppose biological memory technology has the following unusual characteristic: It is as fast as the fastest semiconductor DRAMs and it can be randomly accessed, but its per byte costs are the same as magnetic disk memory. It has the further advantage of not being any slower no matter how big it is. The only drawback is that you can only write it once, but you can read it many times. Thus it is called a *WORM* (write once, read many) memory. Because of the way it is manufactured, the WORM memory module can be easily replaced. See if you can come up with several new ideas to take advantage of WORMs to build better computers using "biotechnology."

5.23 [Discussion] <3,4,5> Chapters 3 and 4 showed how execution time is being reduced by pipelining and by superscalar and VLIW organizations: even floating-point operations may account for only a fraction of a clock cycle in total execution time. On the other hand, Figure 5.1 on page 374 shows that the memory hierarchy is increasing in importance. The research on algorithms, data structures, operating systems, and even compiler optimizations were done in an era of simpler machines, with no pipelining or caches. Classes and textbooks may still reflect those simpler machines. What is the impact of the changes in computer architecture on these other fields? Find examples where textbooks suggest the solution appropriate for old machines but inappropriate for modern machines. Talk to people in other fields to see what they think about these changes.