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 hierarchies based on memories of different speeds and sizes. Figure 5.1 shows a multilevel memory hierarchy, including typical sizes and speeds of access.

Since fast memory is expensive, a memory hierarchy is organized into several levels—each smaller, faster, and more expensive per byte than the next lower 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.

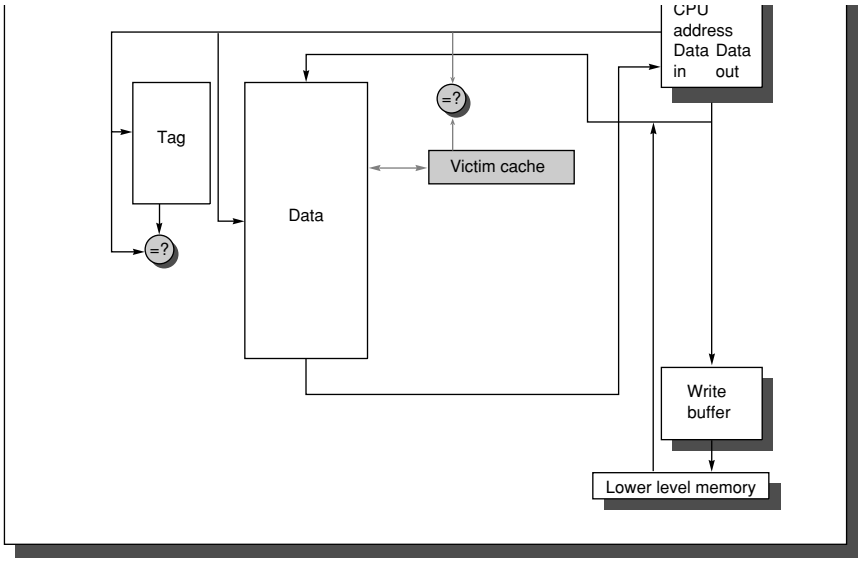

FIGURE 5.1 These are the levels in a typical memory hierarchy in embedded, desktop, and server computers. As we move farther away from the CPU, the memory in the level below becomes slower and larger. Note that the time units change by factors of ten–from picoseconds to milliseconds–and that the size units change by factors of a thousand–from bytes to terabytes. Figure 5.3 shows more parameters for desktops and small servers.

Note that each level maps addresses from a slower, 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 2001 many come with two levels of caches on the chip. As noted in Chapter 1, microprocessor performance improved 55% per year since 1987, and 35% per year until 1986. Figure 5.2 plots CPU performance projections against the historical performance improvement in time to access main memory. Clearly, there is a processor-memory performance gap that computer architects must try to close.

This chapter describes the many ideas invented to overcome the processor-memory performance gap. 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 21264 microprocessor. Toward the end of the chapter we evaluate the impact of these levels on performance using the SPEC95 benchmark programs.

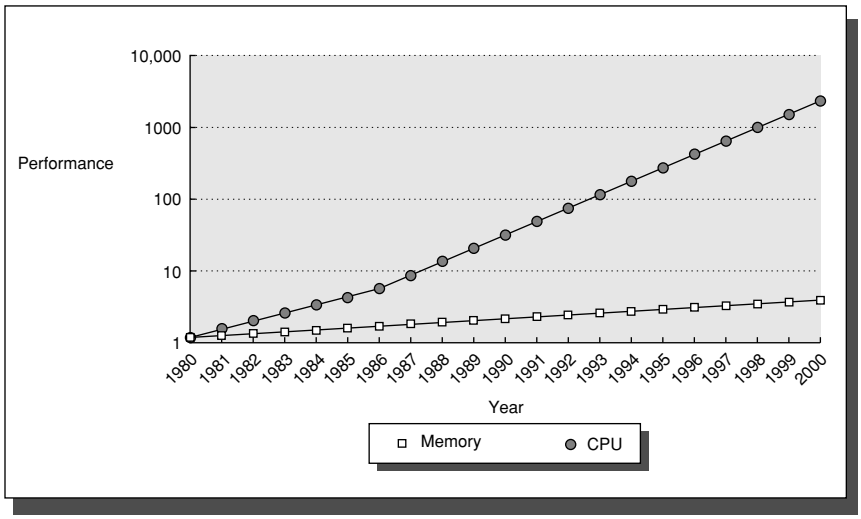

FIGURE 5.2 Starting with 1980 performance as a baseline, the gap in performance between memory and CPUs are plotted over time. Note that the vertical axis must be on a logarithmic scale to record the size of the CPU-DRAM performance gap. The memory baseline is 64-KB DRAM in 1980, with three years to the next generation and a 7% per year performance improvement in latency (see Figure 5.30 on page 444). The CPU line assumes a 1.35 improvement per year until 1986, and a 1.55 improvement thereafter. **<<NOTE: Artist continue the X-axis to 2005: DRAM should be 5.4 in 2005, and CPU should be 25,000 in 2005. This means the Y-axis needs to be extended to 100,000 to accommodate.>>**

The 21264 is a microprocessor designed for desktop and servers. Even these two related classes of computers have different concerns in a memory hierarchy. Desktop computers are primarily running one application at a time on top of an operating system for a single user, whereas server computers may typically have hundreds of users potentially running potentially dozens of applications simultaneously. These characteristics result in more context switches, which effectively increases compulsory miss rates. Thus, desktop computers are concerned more with average latency from the memory hierarchy whereas server computers are also concerned about memory bandwidth. Although protection is important on desktop computers to deter programs from clobbering each other, server computers must prevent one user from accessing another's data, and hence the importance of protection escalates. Server computers also tend to be much larger, with more memory and disk storage, and hence often run much larger applications. In 2001 virtually all servers can be purchased as multiprocessors with hundreds of disks, which places even greater bandwidth demands on the memory hierarchy.

The memory hierarchy of the embedded computers is often quite different from that of the desktop and server. First, embedded computers are often used in real-time applications, and hence programmers must worry about worst case per-

formance. This concern is problematic for caches that improve average case performance, but can degrade worst case performance; we'll mention some techniques to address this later in the chapter. Second, embedded applications are often concerned about power and battery life. The best way to save power is to have less hardware. Hence, embedded computers may not chose hardware-intensive optimizations in the quest of better memory hierarchy performance, as would most desktop and server computers. Third, embedded applications are typically only running one application and use a very simple operating system, if they one at all. Hence, the protection role of the memory hierarchy is often diminished. Finally, the main memory itself may be quite small–less than one megabyte–and there is often no disk storage.

This chapter is a tour of the general principles of memory hierarchy using the desktop as the generic example, but we will take detours to point out where the memory hierarchy of servers and embedded computers diverge from the desktop. Towards the end of the chapter we will pause for two views of the memory hierarchy in addition to the Alpha 21264: the Sony Playstation 2 and the Sun Fire 6800 server. Our first stop is a review.

5.2 Review of 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)

This section is a quick review of cache basics, covering the following 36 terms:

cache	*cache hit*	*cache miss*	*block*
virtual memory	*page*	*page fault*	*page fault*
memory stall cycles	*miss penalty*	*miss rate*	*address trace*
direct mapped	*fully associative*	*n-way set associative*	*set*
valid bit	*dirty bit*	*least-recently used*	*random replacement*
block address	*block offset*	*tag field*	*index*
write through	*write back*	*write allocate*	*no-write allocate*
instruction cache	*data cache*	*unified cache*	*write buffer*
average memory access time	*hit time*	*misses per instruction*	*write stall*

Readers who know the meaning of such terms should skip to "An Example: The Alpha 21264 Data Cache" on page 387, or even further to section 5.3 on page 390 about cache performance. (If this review goes too quickly, you might want to look at Chapter 7 in *Computer Organization and Design*, which we wrote for readers with less experience.)

For those interested in a review, two particularly important levels of the memory hierarchy are cache and virtual memory.

Cache is the name 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 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.

When the CPU finds a requested data item in the cache, it is called a *cache hit*. When the CPU does not find a data item it needs in the cache, a *cache miss* occurs. A fixed-size collection of data containing the requested word, called a *block*, is retrieved from the main memory and placed into the cache. *Temporal locality* tells us that we are likely to need this word again in the near future, so it is useful to place it in the cache where it can be accessed quickly. Because of *spatial locality*, there is high probability that the other data in the block will be needed soon.

The time required for the cache miss depends on both the latency and bandwidth of the memory. Latency determines the time to retrieve the first word of the block, and bandwidth determines the time to retrieve the rest of this block. A cache miss is handled by hardware and causes processors following in-order execution to pause, or stall, until the data are available.

Similarly, not all objects referenced by a program need to reside in main memory. If the computer has *virtual memory*, then some objects may reside on disk. The address space is usually broken into fixed-size blocks, called *pages*. At any time, each page resides either in main memory or on disk. When the CPU references an item within a page that is not present in the cache or main memory, a *page fault* occurs, and the entire page is moved from the disk to main memory. Since page faults take so long, they are handled in software and the CPU is not stalled. The CPU usually switches to some other task while the disk access occurs. The cache and main memory have the same relationship as the main memory and disk.

Figure 5.3 shows the range of sizes and access times of each level in the memory hierarchy for computers ranging from high-end desktops to low-end servers.

Cache Performance Review

Because of locality and the higher speed of smaller memories, a memory hierarchy can substantially improve performance. One method to evaluate cache performance is to expand our CPU execution time equation from Chapter 1. We now account for the number of cycles during which the CPU is stalled waiting for a memory access, which we call the *memory stall cycles*. The performance is then the product of the clock cycle time and the sum of the CPU cycles and the memory stall cycles:

Level	1	2	3	4
Called	Registers	Cache	Main memory	Disk storage
Typical size	< 1 KB	< 16 MB	< 16 GB	> 100 GB
Implementation technology	Custom memory with multiple ports, CMOS	On-chip or off-chip CMOS SRAM	CMOS DRAM	Magnetic disk
Access time (in ns)	0.25 -0.5	0.5 to 25	80-250	5,000,000
Bandwidth (in MB/sec)	20,000-100,000	5,000-10,000	1000-5000	20-150
Managed by	Compiler	Hardware	Operating system	Operating system/operator
Backed by	Cache	Main memory	Disk	CD or Tape

FIGURE 5.3 The typical levels in the hierarchy slow down and get larger as we move away from the CPU for a large workstation or small server. Embedded computers might have no disk storage, and much smaller memories and caches. The access times increase as we move to lower levels of the hierarchy, which makes it feasible to manage the transfer less responsively. The implementation technology shows the typical technology used for these functions. The access time is given in nanoseconds for typical values in 2001; these times will decrease over time. Bandwidth is given in megabytes per second between levels in the memory hierarchy. Bandwidth for disk storage includes both the media and the buffered interfaces.

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

This equation assumes that the CPU clock cycles include the time to handle a cache hit, and that the CPU is stalled during a cache miss. Section 5.3 re-examines this simplifying assumption.

The number of memory stall cycles depends on both the number of misses and the cost per miss, which is called the *miss penalty*:

$$\begin{aligned}\text{Memory stall cycles} &= \text{Number of misses} \times \text{Miss penalty}\\ &= \text{IC} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}\\ &= \text{IC} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss rate} \times \text{Miss penalty}\end{aligned}$$

The advantage of the last form is that the components can be easily measured. We already know how to measure IC (instruction count). Measuring the number of memory references per instruction can be done in the same fashion; every instruction requires an instruction access and we can easily decide if it also requires a data access.

Note that we calculated miss penalty as an average, but we will use it below as if it were a constant. The memory behind the cache may be busy at the time of the miss due prior memory requests or memory refresh (see section 5.9). The number of clock cycles also varies at interfaces between different clocks of the processor, bus, and memory. Thus, please remember that using a single number for miss penalty is a simplification.

The component *miss rate* is simply the fraction of cache accesses that result in a miss (i.e., number of accesses that miss divided by number of accesses). Miss rates can be measured with cache simulators that take an *address trace* of the instruction and data references, simulate the cache behavior to determine which references hit and which miss, and then report the hit and miss totals. Some microprocessors provide hardware to count the number of misses and memory references, which is a much easier and faster way to measure miss rate.

The formula above is an approximation since the miss rates and miss penalties are often different for reads and writes. Memory stall clock cycles could then be defined in terms of the number of memory accesses per instruction, miss penalty (in clock cycles) for reads and writes, and miss rate for reads and writes:

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

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

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

The miss rate is one of the most important measures of cache design, but, as we will see in later sections, not the only measure.

EXAMPLE Assume we have a computer where the clocks per instruction (CPI) is 1.0 when all memory accesses hit in the cache. The only data accesses are loads and stores, and these total 50% of the instructions. If the miss penalty is 25 clock cycles and the miss rate is 2%, how much faster would the computer be if all instructions were cache hits?

ANSWER First compute the performance for the computer that always hits:

$$\begin{aligned}\text{CPU execution time} &= (\text{CPU clock cycles} + \text{Memory stall cycles}) \times \text{Clock cycle} \\ &= (\text{IC} \times \text{CPI} + 0) \times \text{Clock cycle} \\ &= \text{IC} \times 1.0 \times \text{Clock cycle}\end{aligned}$$

Now for the computer with the real cache, first we compute memory stall cycles:

$$\begin{aligned}\text{Memory stall cycles} &= \text{IC} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss rate} \times \text{Miss penalty} \\ &= \text{IC} \times (1 + 0.5) \times 0.02 \times 25 \\ &= \text{IC} \times 0.75\end{aligned}$$

where the middle term (1 + 0.5) represents one instruction access and 0.5

data accesses per instruction. The total performance is thus

$$\begin{aligned}\text{CPU execution time}_{\text{cache}} &= (\text{IC} \times 1.0 + \text{IC} \times 0.75) \times \text{Clock cycle} \\ &= 1.75 \times \text{IC} \times \text{Clock cycle}\end{aligned}$$

The performance ratio is the inverse of the execution times:

$$\begin{aligned}\frac{\text{CPU execution time}_{\text{cache}}}{\cancel{\text{CPU execution time}}} &= \frac{1.75 \times \text{IC} \times \text{Clock cycle}}{\cancel{1.0 \times \text{IC} \times \text{Clock cycle}}} \\ &= 1.75\end{aligned}$$

The computer with no cache misses is 1.75 times faster. n

Some designers prefer measuring miss rate as *misses per instruction* rather than misses per memory reference. These two are related:

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

The latter formula is useful when you know the average number of memory accesses per instruction as it allows you to convert miss rate into misses per instruction, and vice versa. For example, we can turn the miss rate per memory reference in the example above into misses per instruction:

$$\frac{\text{Misses}}{\cancel{\text{Instruction}}} = \text{Miss rate} \times \frac{\text{Memory accesses}}{\cancel{\text{Instruction}}} = 0.02 \times 1.5 = 0.030$$

By the way, misses per instruction are often reported as misses per 1000 instructions to show integers instead of fractions. Thus, the answer above could also be expressed as 30 misses per 1000 instructions.

The advantage of misses per instruction is that it is independent of the hardware implementation. For example, the 21264 fetches about twice as many instructions as are actually committed, which can artificially reduce the miss rate if measured as misses per memory reference rather than per instruction. 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 MIPS. Thus, misses per instruction are most popular with architects working with a single computer family, although the similarity of RISC architectures allows one to give insights into others.

EXAMPLE To show equivalency between the two miss rate equations, let's redo the example above, this time assuming a miss rate per 1000 instructions of 30. What is memory stall time in terms of instruction count?

ANSWER Recomputing the memory stall cycles:

$$\begin{aligned}
\text{Memory stall cycles} &= \text{Number of misses} \times \text{Miss penalty} \\
&= \text{IC} \times \frac{\text{Misses}}{\cancel{\text{Instruction}}} \times \text{Miss penalty} \\
&= \text{IC} / 1000 \times \frac{\text{Misses}}{\cancel{\text{Instruction}} \times 1000} \times \text{Miss penalty} \\
&= \text{IC} / 1000 \times 30 \times 25 \\
&= \text{IC} / 1000 \times 750 \\
&= \text{IC} \times 0.75
\end{aligned}$$

We get the same answer as on page 379. n

Four Memory Hierarchy Questions

We continue our introduction to caches by answering the four common questions for the first level of the memory 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.

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

Figure 5.4 shows that the restrictions on where a block is placed create three categories of cache organization:

- 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

  (Block address) MOD *(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 *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,

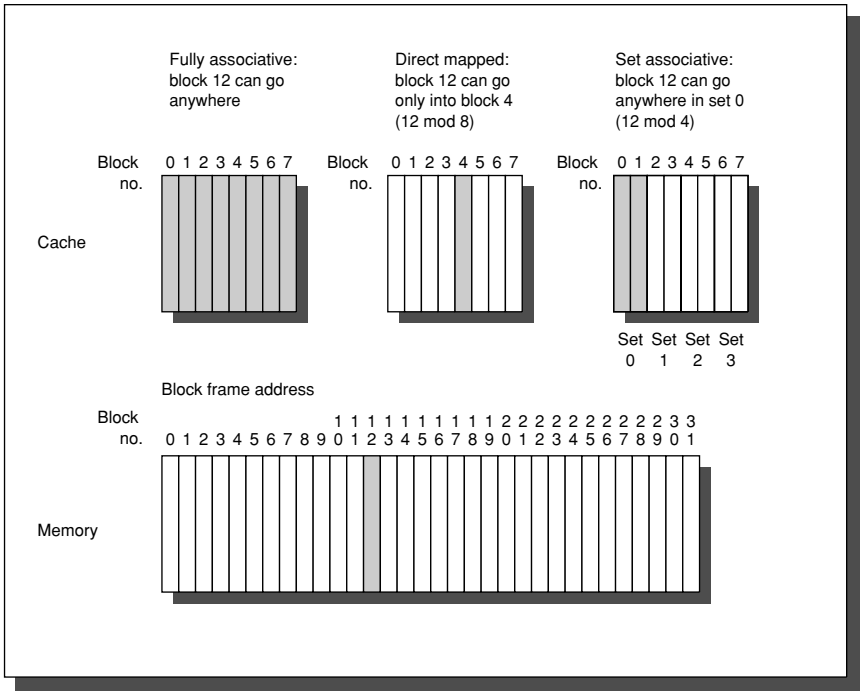

FIGURE 5.4 This example cache has eight block frames and memory has 32 blocks. The three options for caches are shown left to right. In fully associative, block 12 from the lower level can go into any of the eight block frames of the cache. With direct mapped, block 12 can only be placed into block frame 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 in block 1 of the cache. Real caches contain thousands 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.

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

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 5.5 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. Although the comparison could be made on more of the address than the tag, there is no need because of the following:

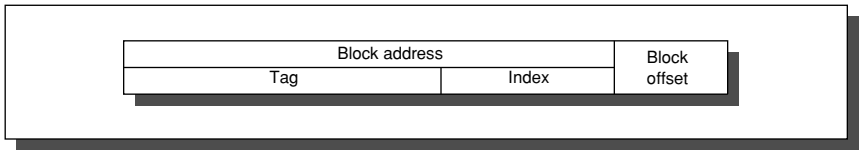

FIGURE 5.5 The three portions of an address in a set-associative or direct-mapped cache. The tag is used to check all the blocks in the set and the index is used to select the set. The block offset is the address of the desired data within the block. Fully associative caches have no index field.

- The offset should not be used in the comparison, since the entire block is present or not, and hence all block offsets result in a match by definition.
- Checking the index is 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; set 1 must have an index value of 1; and so on. This optimization saves hardware and power by reducing the width of memory size for the cache tag.

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 5.5 moves to the right with increasing associativity, with the end point 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 three 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. Relying on the past to predict the future, the block replaced is the one that has been unused for the longest time. LRU relies on a corollary of locality: If recently used blocks are likely to be used again, then a good candidate for disposal is the least-recently used block.
- *First In First Out* (FIFO)—Because LRU can be complicated to calculate, this approximates LRU by determining the *oldest* block rather than the LRU.

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 5.6 shows the difference in miss rates between LRU, random, and FIFO replacement.

	Associativity								
	Two-way			Four-way			Eight-way		
Size	LRU	Random	FIFO	LRU	Random	FIFO	LRU	Random	FIFO
16 KB	114.1	117.3	115.5	111.7	115.1	113.3	109.0	111.8	110.4
64 KB	103.4	104.3	103.9	102.4	102.3	103.1	99.7	100.5	100.3
256 KB	92.2	92.1	92.5	92.1	92.1	92.5	92.1	92.1	92.5

FIGURE 5.6 Data cache misses per 1000 instructions comparing least-recently used, random, and first-in, first-out replacement for several sizes and associativities. There is little difference between LRU and random for the largest-size cache, with LRU outpeforming the others for smaller caches. FIFO generally outperforms random in the smaller cache sizes. These data were collected for a block size of 64 bytes for the Alpha architecture using ten SPEC2000 benchmarks. Five are from SPECint2000 (gap, gcc, gzip, mcf, and perl) and five are from SPECfp2000 (applu, art, equake, lucas, and swim). We will use this computer and these benchmarks in most figures in this chapter.

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 2.32 on page 149 in Chapter 2

suggests a mix of 10% stores and 37% loads for MIPS programs, making writes 10%/(100% + 37% + 10%) or about 7% of the overall memory traffic. Of the *data cache* traffic, writes are 10%/(37% + 10%) or about 21%. 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 in desktop and server computers; just ignore the value read. Embedded's emphasis on power generally means avoiding unnecessary work, which might lead the designer to separate data read from address check so that data is not read on a miss.

Such optimism is not allowed 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; once again, embedded designers might weigh the power benefits of reading less.

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

- *Write through* —The information is written to both the block in the cache *and* to the block in the lower-level memory.
- *Write back* —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 back on a miss, since identical information to the cache is found in lower levels.

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 which are common in servers. Since write back uses the rest of the memory hierarchy and memory buses less than write through, it also saves power, making it attractive for embedded applications.

Write through is easier to implement than write back. The cache is always clean, so unlike write back read misses never result in writes to the lower level. Write through also has the advantage that the next lower level has the most current copy of the data, which simplifies data coherency. Data coherency (see sec-

tion 5.12) is important for multiprocessors and for I/O, which we examine in Chapters 6 and 7.

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 are two options on a write miss:

- *Write allocate* —The block is allocated on a write miss, followed by the write-hit actions above. In this natural option, write misses act like read misses.
- *No-write allocate*—This apparently unusual alternative is write misses do *not* affect the cache. Instead, the block is modified only in the lower level memory.

Thus, blocks stays out of the cache in no-write allocate until the program tries to read the blocks, but even blocks that are only written will still be in the cache with write allocate. Let's look at an example.

EXAMPLE Assume a fully associative write back cache with many cache entries that starts empty. Below is a sequence of five memory operations (the address is in parentheses):

Write Mem[100];
WriteMem[100];
Read Mem[200];
WriteMem[200];
WriteMem[100].

What are the number of hits and misses with using no-write allocate versus write allocate?

ANSWER For no-write allocate, the address 100 is not in the cache, and there is no allocation on write, so the first two writes will result in misses. Address 200 is also not in the cache, so the read is also a miss. The subsequent write to address 200 is a hit. The last write to 100 is still a miss. The result for no write allocate is 4 misses and 1 hit.

For write allocate, the first accesses to 100 and 200 are misses, and the rest are hits since 100 and 200 are both found in the cache. Thus, the result for write allocate is 2 misses and 3 hits. ■

Either write-miss policy could be used with write through or write back. Normally, write-back caches use write allocate, hoping that subsequent writes to that block will be captured by the cache. Write-through caches often use no-write allocate. The reasoning is that even if there are subsequent writes to that block, the writes must still go to the lower level memory, so what's to be gained?

An Example: The Alpha 21264 Data Cache

To give substance to these ideas, Figure 5.7 shows the organization of the data cache in the Alpha 21264 microprocessor that is found in the Compaq AlphaServer ES40, one of several models that use it. The cache contains 65,536 (64K) bytes of data in 64-byte blocks with two-way set-associative placement, write back, and write allocate on a write miss.

Let's trace a cache hit through the steps of a hit as labeled in Figure 5.7. (The four steps are shown as circled numbers.) As we shall see later (Figure 5.36), the 21264 processor presents a 48-bit virtual address to the cache for tag comparison, which is simultaneously translated into a 44-bit physical address. (It also optionally supports 43-bit virtual addresses with 41-bit physical addresses.)

The reason Alpha doesn't use all 64 bits of virtual address is that its designers don't think anyone needs that big of virtual address space yet, and the smaller size simplifies the Alpha virtual address mapping. The designers planned to grow the virtual address in future microprocessors.

The physical address coming into the cache is divided into two fields: the 38-bit block address and 6-bit block offset ($64 = 2^6$ and $38 + 6 = 44$). The block address is further divided into an address tag and cache index. Step 1 shows this division.

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. For the 21264 cache the set associativity is set to two, and we calculate the index as follows:

$$2^{\text{index}} = \frac{\text{Cache size}}{\cancel{\text{Block size} \times \text{Set associativity}}} = \frac{65536}{\cancel{64 \times 2}} = 512 = 2^9$$

Hence, the index is 9 bits wide, and the tag is 38 – 9 or 29 bits wide. Although that is the index needed to select the proper block, 64 bytes is much more than the CPU wants to consume at once. Hence, it makes more sense to organize the data portion of the cache memory 8 bytes wide, which is the natural data word of the 64-bit Alpha processor. Thus, in addition to 9 bits to index the proper cache block, 3 more bits from the block offset are used to index the proper 8 bytes.

Index selection is step 2 in Figure 5.7. The two tags are compared and the winner is selected. (Section 5.10 explains how the 21264 handles virtual address translation.)

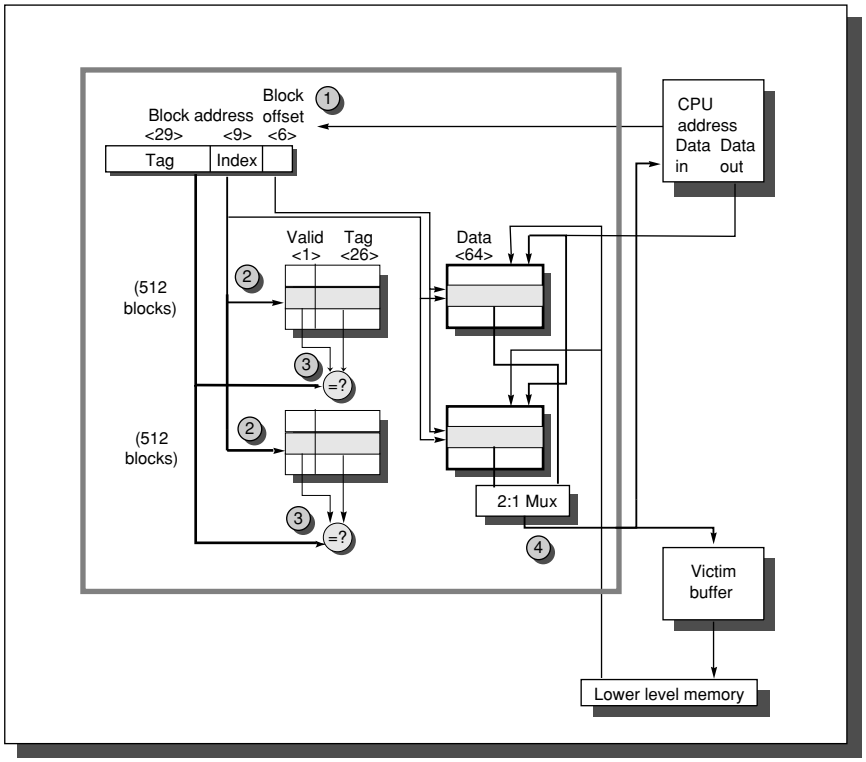

FIGURE 5.7 The organization of the data cache in the Alpha 21264 microprocessor. The 64-KB cache is two-way set associative with 64-byte blocks. The 9-bit index selects between 512 sets. The four steps of a read hit, shown as circled numbers in order of occurrence, label this organization. Three bits of the block offset join the index to supply the RAM address to select the proper 8 bytes. Thus, the cache holds two groups 4096 64-bit words, with each group containing half of the 512 sets. Although not exercised in this example, the line from lower level memory to the cache is used on a miss to load the cache. The size of address leaving the CPU is 44 bits because it is a physical address and not a virtual address. Figure 5.36 on page 454 explains how the Alpha maps from the virtual to physical for a cache access. <**<REVIZE DRAWING, including the more realistic drawing of the memory blocks as in old figure 5.10 in CA:AQA 2/e>>**

After reading the two tags from the cache, they are compared to the tag portion of the block address from the CPU. This comparison is step 3 in the figure. To be sure the tag contains valid information, the valid bit must be set or else the results of the comparison are ignored.

Assuming one tag does match, the final step is to signal the CPU to load the proper data from the cache by using the winning input from a 2:1 multiplexor. The 21264 allows three clock cycles for these four steps, so the instructions in the following two clock cycles would wait if they tried to use the result of the load.

Handling writes is more complicated than handling reads in the 21264, as it is in any cache. If the word to be written is in the cache, the first three steps are the same. Since the 21264 executes out-of-order, only after it signals that the instruction has committed and the cache tag comparison indicates a hit are the data are written to the cache.

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 signal to the processor telling it the data is not yet available, and 64 bytes are read from the next level of the hierarchy. The path to the next lower level in the 21264 is 16 bytes wide. In the 667 MHz AlphaServer ES40 it takes 2.25 ns per transfer, or 9 ns for all 64 bytes. Since the data cache is set associative, there is a choice on which block to replace. The 21264 does *round robin* selection, dedicating a bit for every two blocks to remember where to go next. Unlike LRU, which selects the block that was referenced longest ago, round robin selects the block that was filled longest ago. Round robin is easier to implement since it is only updated on a miss rather than on every hit. Replacing a block means updating the data, the address tag, the valid bit, and the round robin bit.

Since the 21264 uses write back, the old data block could have been modified, and hence it cannot simply be discarded. The 21264 keeps one dirty bit per block to record if the block was written. If the "victim" was modified, its data and address are sent to the Victim Buffer. (This structure is similar to a *write buffer* in other computers.) The 21264 has space for eight victim blocks. In parallel with other cache actions, it writes victim blocks to the next level of the hierarchy. If the Victim Buffer is full, the cache must wait.

A write miss is very similar to a read miss, since the 21264 allocates a block on a read or a write miss.

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 21264. Hence, it has a 64-KB instruction cache as well as the 64-KB data cache.

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 21264, the terms *unified* or *mixed* are applied to caches that can contain either instructions or data.)

Size	Instruction cache	Data cache	Unified cache
8 KB	8.16	44.0	63.0
16 KB	3.82	40.9	51.0
32 KB	1.36	38.4	43.3
64 KB	0.61	36.9	39.4
128 KB	0.30	35.3	36.2
256 KB	0.02	32.6	32.9

FIGURE 5.8 Miss per 1000 instructions for instruction, data, and unified caches of different sizes. The percentage of instruction references is about 78%. The data are for two-way associative caches with 64-byte blocks for the same computer and benchmarks as Figure 5.6.

Figure 5.8 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 16-KB instruction cache and 16-KB data cache should be compared to a 32-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 2.32 on page 149 suggests the split is 100%/(100% + 37% + 10%) or about 78% instruction references to (37% + 10%)/(100% + 37% + 10%) or about 22% data references. Splitting affects performance beyond what is indicated by the change in miss rates, as we shall see shortly.

5.3 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 memory access time*:

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

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, 0.25 to 1.0 nanoseconds on a hit—or in the number of clock cycles that

the CPU waits for the memory—such as a miss penalty of 75 to 100 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 assuming 47% of the instructions are data transfer instructions. Assume a hit takes 1 clock cycle and the miss penalty is 100 clock cycles. A load or store hit takes 1 extra clock cycle on a unified cache if 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 First let's convert misses per 1000 instructions into miss rates. Solving the general formula is from above, miss rate is

$$\text{Miss rate} = \frac{\frac{\text{Misses}}{\text{1000 Instructions}} / 1000}{\frac{\text{Memory accesses}}{\text{Instruction}}}$$

Since every instruction access has exactly 1 memory access to fetch the instruction, the instruction miss rate is:

$$\text{Miss rate}_{\text{16 KB Instruction}} = \frac{3.82 / 1000}{1.00} = 0.004$$

Since 47% of the instructions are data transfers, the data miss rate is:

$$\text{Miss rate}_{\text{16 KB Data}} = \frac{40.9 / 1000}{0.47} = 0.087$$

The unified miss rate needs to account for instruction and data accesses:

$$\text{Miss rate}_{\text{32 KB Unified}} = \frac{43.3 / 1000}{1.00 + 0.47} = 0.029$$

As stated above, about 78% of the memory accesses are instruction references. Thus, the overall miss rate for the split caches is

$$(78\% \times 0.004) + (22\% \times 0.087) = 0.022$$

Thus, a 32-KB unified cache has a higher effective miss rate than two16-KB caches.

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

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

Therefore, the time for each organization is

$$\begin{aligned}&\text{Average memory access time}_{\text{split}}\\&= 78\% \times (1 + 0.004 \times 100) + 22\% \times (1 + 0.087 \times 100)\\&= (78\% \times 1.38) + (22\% \times 9.70) = 1.078 + 2.134 = 3.21\\&\text{Average memory access time}_{\text{unified}}\\&= 78\% \times (1 + 0.029 \times 100) + 22\% \times (1 + 1 + 0.029 \times 100)\\&= (78\% \times 3.95) + (22\% \times 4.95) = 3.080 + 1.089 = 4.17\end{aligned}$$

Hence, the split caches in this example—which offer two memory ports per clock cycle, thereby avoiding the structural hazard—also have a better average memory access time than the single-ported unified cache. n

Average memory access time and Processor Performance

An obvious question is whether average memory access time due to cache misses predicts processor performance.

First, there are other reasons for stalls, such as contention due to I/O devices using memory. Designers often assume that all memory stalls are due to cache misses, since the memory hierarchy typically dominates other reasons for stalls. We use this simplifying assumption here, but beware to account for *all* memory stalls when calculating final performance.

Second, the answer depends also on the CPU. If we have an in-order execution CPU (See Chapter 3), then the answer is basically yes. The CPU stalls during misses, and the memory stall time is strongly correlated to average memory access time. Let's make that assumption for now, but we'll return to out-of-order CPUs in the next subsection.

As stated in the prior section, we can model CPU time as:

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

This formula 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.

We can now explore the impact of caches on performance.

EXAMPLE Let's use an in-order execution computer for the first example, such as the UltraSPARC III (see section 5.15). Assume the cache miss penalty is 100 clock cycles, and all instructions normally take 1.0 clock cycles (ignoring memory stalls). Assume the average miss rate is 2%, there is an average of 1.5 memory references per instruction, and that the average number of cache misses per 1000 instructions is 30. What is the impact on performance when behavior of the cache is included? Calculate the impact using both misses per instruction and miss rate.

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

$$\begin{aligned} \text{CPU time}_{\text{with cache}} &= \text{IC} \times (1.0 + (30 / 1000 \times 100)) \times \text{Clock cycle time} \\ &= \text{IC} \times 4.00 \times \text{Clock cycle time} \end{aligned}$$

Now calculating performance using miss rate:

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

$$\begin{aligned} \text{CPU time}_{\text{with cache}} &= \text{IC} \times (1.0 + (1.5 \times 2\% \times 100)) \times \text{Clock cycle time} \\ &= \text{IC} \times 4.00 \times \text{Clock cycle time} \end{aligned}$$

The clock cycle time and instruction count are the same, with or without a cache. Thus, CPU time increases fourfold, with CPI from 1.00 for a "perfect cache" to 4.00 with a cache that can miss. Without any memory hierarchy at all the CPI would increase again to 1.0 + 100 × 1.5 or 151—a factor of almost 40 times longer than a system with a cache! n

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 a higher memory portion of CPI.

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 computers. 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, the clock cycle time is 1.0 ns, there are 1.5 memory references per instruction, the size of both caches is 64 KB, and both have a block size of 64 bytes. One cache is direct mapped and the other is two-way set associative. Figure 5.7 on page 388 shows that for set-associative caches we must add a multiplexor 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 1.25 times to accommodate the selection multiplexor of the set-associative cache. To the first approximation, the cache miss penalty is 75 ns for either cache organization. (In practice, it is normally 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, 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%.

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

$$\text{Average memory access time}_{\text{1-way}} = 1.0 + (.014 \times 75) = 2.05 \text{ ns}$$
$$\text{Average memory access time}_{\text{2-way}} = 1.0 \times 1.25 + (.010 \times 75) = 2.00 \text{ ns}$$

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

CPU performance is

$$\text{CPU time} = \text{IC} \times \left(\text{CPI}_{\text{Execution}} + \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty} \right) \times \text{Clock cycle time}$$

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

Substituting 75 ns for (Miss penalty × Clock cycle time), the performance

of each cache organization is

$$\text{CPU time}_{\text{1-way}} = \text{IC} \times (2 \times 1.0 + (1.5 \times 0.014 \times 75)) = 3.58 \times \text{IC}$$
$$\text{CPU time}_{\text{2-way}} = \text{IC} \times (2 \times 1.0 \times 1.25 + (1.5 \times 0.010 \times 75)) = 3.63 \times \text{IC}$$

and relative performance is

$$\frac{\text{CPU time}_{\text{2-way}}}{\text{CPU time}_{\text{1-way}}} = \frac{3.63 \times \text{Instruction count}}{3.58 \times \text{Instruction count}} = \frac{3.63}{3.58} = 1.01$$

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 set-associative 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. n

Miss Penalty and Out-of-Order Execution Processors

For an out-of-order execution processor, how do you define miss penalty? Is it the full latency of the miss to memory, or is it just the "exposed" or non-overlapped latency when the processor must stall? This question does not arise in processors which stall until the data miss completes.

Let's redefine memory stalls to lead to a new definition of miss penalty as non-overlapped latency:

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

Similarly, as some out-of-order CPUs stretch the hit time, that portion of the performance equation could be divided total hit latency less overlapped hit latency. This equation could be further expanded to account for contention for memory resources in an out-of-order processor by dividing total miss latency into latency without contention and latency due to contention. Let's just concentrate on miss latency.

We now have to decide

- *length of memory latency*: what to consider as the start and the end of a memory operation in an out-of-order processor; and
- *length of latency overlap*: what is the start of overlap with for the processor (or equivalently, when do we say a memory operation is stalling the processor).

Given the complexity of out-of-order execution processors, there is no single correct definition.

Since only committed operations are seen at the retirement pipeline stage, we say a processor is stalled in a clock cycle if it does not retire the maximum possible number of instructions in that cycle. We attribute that stall to the first instruction that could not be retired. This definition is by no means foolproof. For example, applying an optimization to improve a certain stall time may not always improve execution time because another type of stall—hidden behind the targeted stall—may now be exposed.

For latency, we could start measuring from the time the memory instruction is queued in the instruction window, or when the address is generated, or when the instruction is actually sent to the memory system. Any option works as long as it is used in a consistent fashion.

EXAMPLE Let's redo the example above, but this time assuming the processor with the longer clock cycle time supports out-of-order execution yet still has a direct mapped cache. Assume 30% of the 75 ns miss penalty can be overlapped; that is, the average CPU memory stall time is now 52.5 ns.

ANSWER Average memory access time for the out-of-order computer is

$$\text{Average memory access time}_{\text{1-way,OOO}} = 1.0 \times 1.25 + (0.014 \times 52.5) = 1.99 \text{ ns}$$

The performance of the OOO cache is

$$\text{CPU time}_{\text{1-way,OOO}} = \text{IC} \times (2 \times 1.0 \times 1.25 + (1.5 \times 0.014 \times 52.5)) = 3.60 \times \text{IC}$$

Hence, despite a much slower clock cycle time and the higher miss rate of a direct mapped cache, the out-of-order computer can be slightly faster if it can hide 30% of the miss penalty. n

In summary, although the state-of-the-art in defining and measuring memory stalls for out-of-order processors is not perfect and is relatively complex, readers should be aware of the issues for they significantly affect performance.

Improving Cache Performance

To help summarize this section and to act as a handy reference, Figure 5.9 lists the cache equations in this chapter.

The increasing gap between CPU and main memory speeds shown in Figure 5.2 has attracted the attention of many architects. A bibliographic search for the years 1989 to 2001 revealed more than 5000 research papers on the subject of caches. Your authors' job was to survey all 5000 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!

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

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

$$\text{Memory stall cycles} = \text{Number of misses} \times \text{Miss penalty}$$

$$\text{Memory stall cycles} = \text{IC} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

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

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

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

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

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

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

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

$$\frac{\text{Memory stall cycles}}{\text{Instruction}} = \frac{\text{Misses}_{\text{L1}}}{\text{Instruction}} \times \text{Hit time}_{\text{L2}} + \frac{\text{Misses}_{\text{L2}}}{\text{Instruction}} \times \text{Miss penalty}_{\text{L2}}$$

FIGURE 5.9 Summary of performance equations in this chapter. The first equation calculates with cache index size, but the rest help evaluates performance. The final two equations deal with multilevel caches, is explained early in the next section. They are included here to help make the figure a useful reference.

Fortunately, this task was simplified by our long standing policy of only including ideas in this book that have made their way into commercially viable computers. In computer architecture, many ideas look much better on paper than in silicon.

The average memory access time formula gave us a framework to present the surviving cache optimizations for improving cache performance or power:

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

Hence, we organize 16 cache optimizations into four categories:

- Reducing the miss penalty (Section 5.4): multilevel caches, critical word first, read miss before write miss, merging write buffers, victim caches;
- Reducing the miss rate (Section 5.5): larger block size, larger cache size, higher associativity, pseudo-associativity, and compiler optimizations;
- Reducing the miss penalty or miss rate via parallelism (Section 5.6): nonblocking caches, hardware prefetching, and compiler prefetching;
- Reducing the time to hit in the cache (Section 5.7): small and simple caches, avoiding address translation, and pipelined cache access.

Figure 5.26 on page 436 concludes with a summary of the implementation complexity and the performance benefits of the 17 techniques presented.

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 5.2 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 increasing miss penalty. Perhaps the most interesting optimization is the first, which adds more levels of caches to reduce miss penalty.

First Miss Penalty Reduction Technique: Multi-Level Caches

Many techniques to reduce miss penalty affect the CPU. This 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. Adding another level of cache between the original cache and memory simplifies the decision. The first-level cache can be small enough to match the clock cycle time of the fast CPU. Yet the second-level cache can be large enough to capture many accesses that would go to main memory, thereby lessening the effective miss penalty.

Although 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*—This rate is simply the number of misses in a cache divided by the total number of memory accesses to this cache. As you would expect, for the first-level cache it is equal to Miss rate_{L1} and for the second-level cache it is Miss rate_{L2}.
- *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 for the first-level cache is still just Miss rate_{L1} but for the second-level cache it is Miss $\text{rate}_{L1} \times$ Miss rate_{L2}.

This local miss rate is large for second level caches because the first-level cache skims the cream of the memory accesses. 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.

Here is a place where the misses per instruction metric shines. Instead of confusion about local or global miss rates, we just expand memory stalls per instruction to add the impact of a second level cache.

Average memory stalls per instruction = Misses per $\text{instruction}_{L1} \times$ Hit time_{L2} + Misses per $\text{instruction}_{L2} \times$ Miss penalty_{L2}.

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? Assume the miss penalty from L2 cache to Memory is 100 clock cycles, the hit time of L2 cache is 10 clock cycles, the Hit time of L1 is 1 clock cycles, and there are 1.5 memory references per instruction. What is the average memory access time and average stall cycles per instruction? Ignore the impact of writes.

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%. Then

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

$= 1 + 4\% \times (10 + 50\% \times 100) = 1 + 4\% \times 60 = 3.4$ clock cycles

To see how many misses we get per instruction, we divide 1000 memory references by 1.5 memory references per instruction, which yields 667 instructions. Thus, we need to multiply the misses by 1.5 to get the number of misses per 1000 instructions. We have 40×1.5 or 60 L1 misses 20×1.5 or 30 L2 misses per 1000 instructions. For average memory stalls per instruction, assuming the misses are distributed uniformly between instructions and data:

Average memory stalls per instruction = Misses per instruction$_{L1}$ × Hit time$_{L2}$ + Misses per instruction$_{L2}$ × Miss penalty$_{L2}$
= (60/1000) × 10 + (30/1000) × 100
= 0.060 × 10 + 0.030 × 100 = 3.6 clock cycles
If we subtract the L1 hit time from AMAT and then multiplying by the average number of memory references per instruction we get the same average memory stalls per instruction:
(3.4 - 1.0) x 1.5 = 2.4 x 1.5 = 3.6 clock cycles.
As this example shows, there is less confusion with multilevel caches when calculating using misses per instruction versus miss rates. n

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 might be used.

Figures 5.10 and 5.11 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 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 two major 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! One question is whether set associativity makes more sense for second-level caches.

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

- Hit time$_{L2}$ for direct mapped = 10 clock cycles
- Two-way set associativity increases hit time by 0.1 clock cycles to 10.1 clock cycles

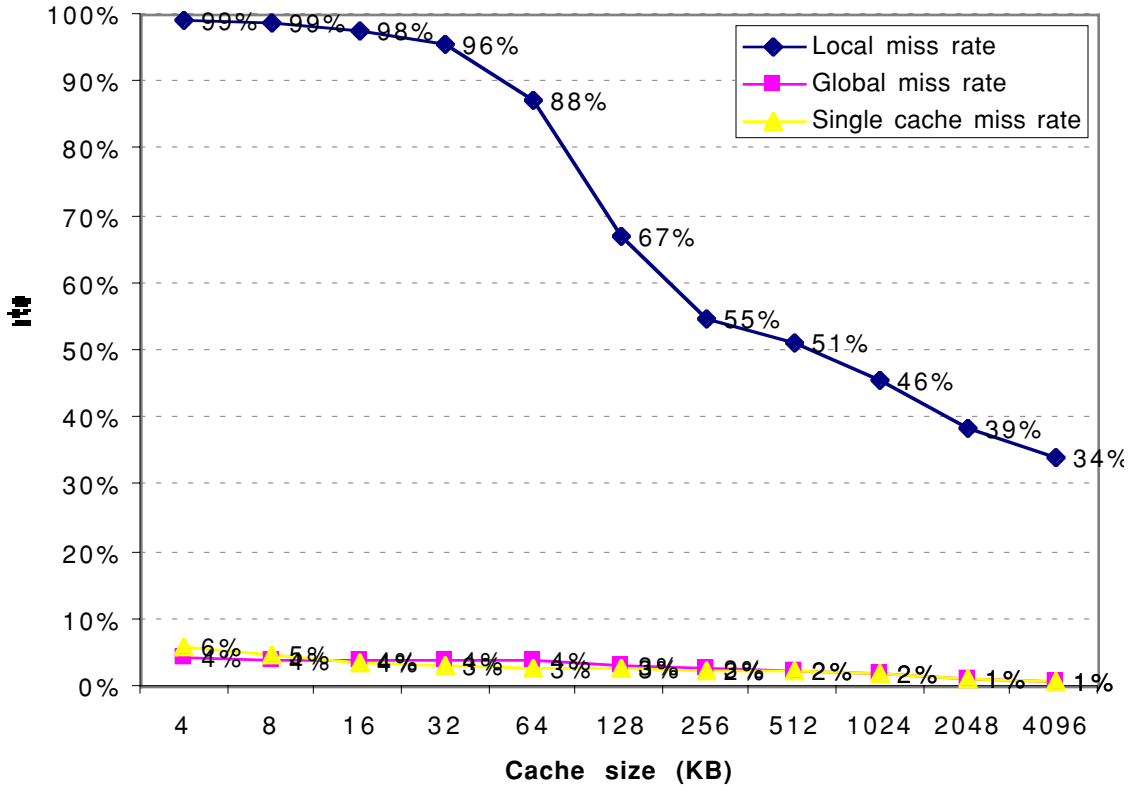

FIGURE 5.10 Miss rates versus cache size for multilevel caches. Second-level caches *smaller* than the sum of the two 64-KB first level make little sense, as reflected in the high miss rates. After 256 KB the single cache is within 10% of the global miss rates. The miss rate of a single-level cache versus size is plotted against the local miss rate and global miss rate of a second-level cache using a 32-KB first-level cache. The L2 Caches (unified) were 2-way set-associative with LRU replacement. Each had split L1instruction and data caches that were 64KB 2-way set-associative with LRU replacement. The block size for both L1 and L2 caches was 64 bytes. Data was collected for as in Figure 5.6.**<<Artist please separate numbers for graphs at bottom>>**

- Local miss rate$_{L2}$ for direct mapped = 25%
- Local miss rate$_{L2}$ for two-way set associative = 20%
- Miss penalty$_{L2}$ = 100 clock cycles

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

$$\text{Miss penalty}_{1\text{- way L2}} = 10 + 25\% \times 100 = 35.0 \text{ clock cycles}$$

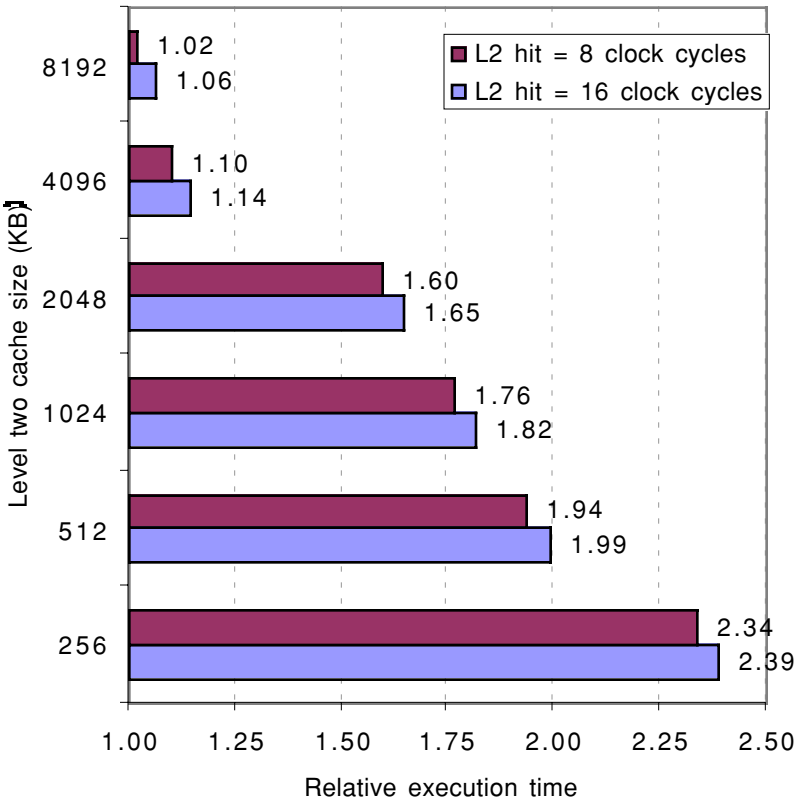

FIGURE 5.11 Relative execution time by second-level cache size. The two bars are for different clock cycles for a L2 cache hit. The reference execution time of 1.00 is for an 8192-KB second-level cache with a one-clock-cycle latency on a second-level hit. These data were collected the same way as in Figure 5.10, using a simulator to imitate the Alpha 21264.

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

$$\text{Miss penalty}_{2\text{- way L2}} = 10.1 + 20\% \times 100 = 30.1 \text{ clock cycles}$$

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 shave the second-level hit time to 10 cycles; if not, we round up to 11 cycles. Either choice is an improvement over the direct-mapped second-level cache:

$$\text{Miss penalty}_{2\text{- way L2}} = 10 + 20\% \times 100 = 30.0 \text{ clock cycles}$$

$$\text{Miss penalty}_{2\text{- way L2}} = 11 + 20\% \times 100 = 31.0 \text{ clock cycles}$$

n

Now we can reduce the miss penalty by reducing the *miss rate* of the second-level caches.

Another consideration concerns whether data in the first-level cache is in the second-level cache. *Multilevel inclusion* is the natural policy for memory hierarchies: L1 data is always present in L2. Inclusion is desirable because consistency between I/O and caches (or among caches in a multiprocessor) can be determined just by checking the second-level cache (see section 8.7).

One drawback to inclusion is that measurements can suggest smaller blocks for the smaller first-level cache and larger blocks for the larger second-level cache. For example, the Pentium 4 has 64-byte blocks in its L1 caches and 128-byte blocks in its L2 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. To avoid such problems, many cache designers keep the block size the same in all levels of caches.

However, what if the designer can only afford an L2 cache that is slightly bigger than the L1 cache? Should a significant portion of its space be used as a redundant copy of the L1 cache? In such cases a sensible opposite policy is *multilevel exclusion*: L1 data is *never* found in L2 cache. Typically, with exclusion a cache miss in L1 results in a swap of blocks between L1 and L2 instead of a replacement of a L1 block with a L2 block. This policy prevents wasting space in L2 cache. For example, the AMD Athlon chip obeys the exclusion property since it has two 64 KB first level caches and only a 256 KB L2 cache.

As these issues illustrate, 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 compatible second-level cache. 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.

The essence of all cache designs is balancing fast hits and few misses. 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 much larger caches and techniques to lower the miss rate described in section 5.5, such as higher associativity and larger blocks.

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

Multilevel caches require extra hardware to reduce miss penalty, but not this second technique. It is based on the observation that the CPU normally 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:

- *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*.
- *Early restart*—Fetch the words in normal order, but as soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution.

Generally these techniques only benefit designs with large cache blocks, since the benefit is low unless blocks are large. The problem is that given spatial locality, there is more than random chance that the next miss is to the remainder of the block. In such cases, the effective miss penalty is the time from the miss until the second piece arrives.

EXAMPLE Let's assume a computer has a 64-byte cache block, an L2 cache that takes 11 clock cycles to get the critical 8 bytes, and then 2 clock cycles per 8 bytes to fetch the rest of the block.(These parameters are similar to the AMD Athlon.) Calculate the average miss penalty for critical word first, assuming that there will be no other accesses to the rest of the block until it is completely fetched. Then calculate assuming the following instructions reads data sequentially 8 bytes at a time from the rest of the block. Compare the times with and without critical word first.

ANSWER The average miss penalty is 11 clock cycles for critical word first. The Athlon can issue two loads per clock cycle, which is faster than the L2 cache can supply data. Thus, it would take 11 + (8-1) x 2 or 25 clock cycles for the CPU to sequentially read a full cache block. Without critical word first, it would take 25 clocks cycle to load the block, and then 8/2 or 4 clocks to issue the loads, giving 29 clock cycles total. ▫

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.

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

This optimization serves reads before writes have been completed. We start with looking at complexities of a write buffer.

With a write-through cache the most important improvement is a write buffer (page 386) of the proper size. 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 R2, 512(R0)          ; 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! n

The simplest way out of this dilemma is for the read miss to wait until the write buffer is empty. 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. Virtually all desktop and server processors use the latter approach, giving reads priority over writes.

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.

Fourth Miss Penalty Reduction Technique: Merging Write Buffer

This technique also involves write buffers, this time improving their efficiency.

Write through caches rely on write buffers, as all stores must be sent to the next lower level of the hierarchy. As mentioned above, even write back caches

use a simple buffer when a block is replaced. 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 can be 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*.

If the buffer is full and there is no address match, the cache (and CPU) must wait until the buffer has an empty entry. This optimization uses the memory more efficiently since multiword writes are usually faster than writes performed one word at a time.

The optimization also reduces stalls due to the write buffer being full. Figure 5.12 shows a write buffer with and without write merging. Assume we had four entries in the write buffer, and each entry could hold four 64-bit words. Without this optimization, four stores to sequential addresses would fill the buffer at one word per entry, even though these four words when merged exactly fit within a single entry of the write buffer.

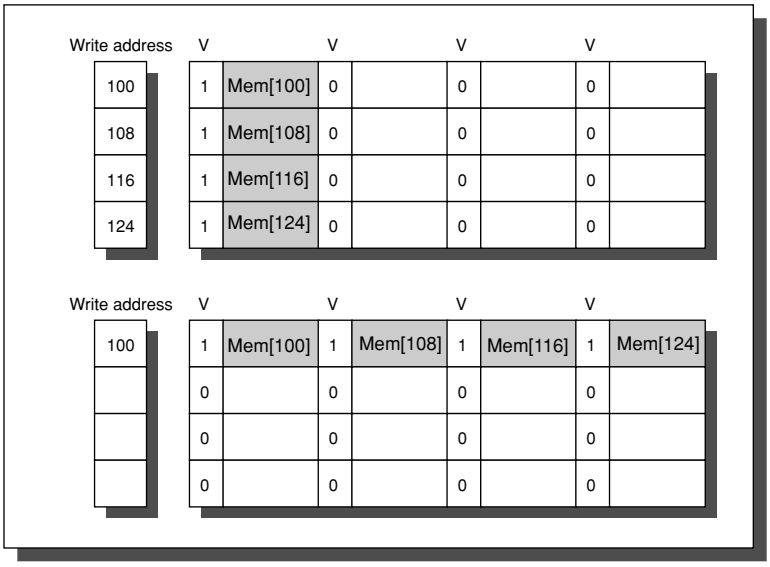

FIGURE 5.12 To illustrate write merging, the write buffer on top does not use it while the write buffer on the bottom does. The four writes are merged into a single buffer entry with write merging; without it, the buffer is full even though three-fourths of each entry is wasted. The buffer has four entries, 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 eight bytes are occupied in this entry. (Without write merging, the words to the right in the upper drawing would only be used for instructions which wrote multiple words at the same time.)

Note that input/output device registers are often mapped into the physical address space, as is the case of the 21264. These I/O addresses cannot allow write merging, as separate I/O registers may not act like an array of words in memory. For example, they may require one addresses and data word per register rather than multiword writes using a single address.

Fifth Miss Penalty Reduction Technique: Victim Caches

One approach to lower miss penalty is to remember what was discarded in case it is needed again. Since the discarded data has already been fetched, it can be used again at small cost.

Such "recycling" requires a small, fully associative cache between a cache and its refill path. Figure 5.13 shows the organization. This *victim cache* contains only blocks that 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. The AMD Athlon has a victim cache with eight entries.

Jouppi [1990] found that victim caches of one to five entries are effective at reducing misses, especially for small, direct-mapped data caches. Depending on the program, a four-entry victim cache might remove one quarter of the misses in a 4-KB direct-mapped data cache.

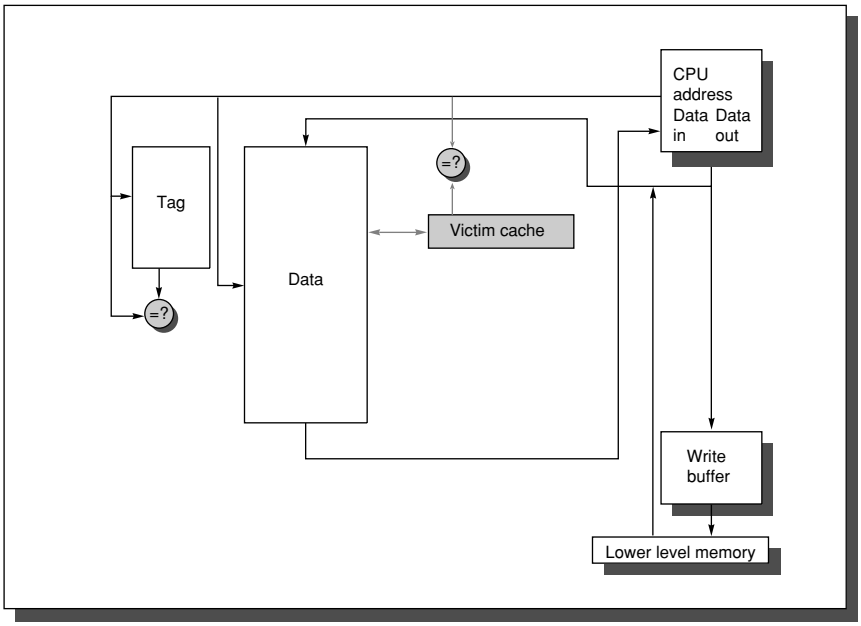

FIGURE 5.13 Placement of victim cache in the memory hierarchy. Although it reduces miss penalty, the victim cache is aimed at reducing the damage done by conflict misses, described in the next section. Jouppi [1990] found the four-entry victim cache could reduce the miss penalty for 20% to 95% of conflict misses.

Summary of Miss Penalty Reduction Techniques

The processor-memory performance gap of Figure 5.2 on page 375 determines the miss penalty, and as the gap grows so do techniques that try to close it. We present five in this section. The first technique follows the proverb "the more the merrier": assuming the principle of locality will keep applying recursively, just keep adding more levels of increasingly larger caches until you are happy. The second technique is impatience: it retrieves the word of the block that caused the miss rather than waiting for the full block to arrive. The next technique is preference. It gives priority to reads over writes since the processor generally waits for reads but continues after launching writes. The fourth technique is companionship, combining writes to sequential words into a single block to create a more efficient transfer to memory. Finally comes a cache equivalent of recycling, as a victim cache keeps a few discarded blocks available for when the fickle primary cache wants a word that it recently discarded. All these techniques help with miss penalty, but multilevel caches is probably the most important.

Testimony of the importance of miss penalty is that most desktop and server computers use the first four of optimizations. Yet most cache research has concentrated on reducing the miss rate, so that is where we go in the next section.

5.5 Reducing Miss Rate

The classical approach to improving cache behavior is to reduce miss rates, and we present five techniques to do so. To gain better insights into the causes of misses, we first 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 (in addition to compulsory 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 may be discarded and later retrieved if too many blocks map to its set. These misses are also called *collision misses* or *interference misses*. The idea is that hits in a fully associative cache which become misses in an N-way set associative cache are due to more than N requests on some popular sets.

Figure 5.14 shows the relative frequency of cache misses, broken down by the "three C's." Compulsory misses are those that occur in an infinite cache. Capacity misses are those that occur in a fully associative cache. Conflict misses are those that occur going from fully associative to 8-way associative, 4-way associative, and so on. Figure 5.15 presents the same data graphically. The top graph

Cache size	Degree associative	Total miss rate	Miss rate components (relative percent) (*Sum = 100% of total miss rate*)					
			Compulsory		Capacity		Conflict	
4 KB	1-way	0.098	0.0001	0.1%	0.070	72%	0.027	28%
4 KB	2-way	0.076	0.0001	0.1%	0.070	93%	0.005	7%
4 KB	4-way	0.071	0.0001	0.1%	0.070	99%	0.001	1%
4 KB	8-way	0.071	0.0001	0.1%	0.070	100%	0.000	0%
8 KB	1-way	0.068	0.0001	0.1%	0.044	65%	0.024	35%
8 KB	2-way	0.049	0.0001	0.1%	0.044	90%	0.005	10%
8 KB	4-way	0.044	0.0001	0.1%	0.044	99%	0.000	1%
8 KB	8-way	0.044	0.0001	0.1%	0.044	100%	0.000	0%
16 KB	1-way	0.049	0.0001	0.1%	0.040	82%	0.009	17%
16 KB	2-way	0.041	0.0001	0.2%	0.040	98%	0.001	2%
16 KB	4-way	0.041	0.0001	0.2%	0.040	99%	0.000	0%
16 KB	8-way	0.041	0.0001	0.2%	0.040	100%	0.000	0%
32 KB	1-way	0.042	0.0001	0.2%	0.037	89%	0.005	11%
32 KB	2-way	0.038	0.0001	0.2%	0.037	99%	0.000	0%
32 KB	4-way	0.037	0.0001	0.2%	0.037	100%	0.000	0%
32 KB	8-way	0.037	0.0001	0.2%	0.037	100%	0.000	0%
64 KB	1-way	0.037	0.0001	0.2%	0.028	77%	0.008	23%
64 KB	2-way	0.031	0.0001	0.2%	0.028	91%	0.003	9%
64 KB	4-way	0.030	0.0001	0.2%	0.028	95%	0.001	4%
64 KB	8-way	0.029	0.0001	0.2%	0.028	97%	0.001	2%
128 KB	1-way	0.021	0.0001	0.3%	0.019	91%	0.002	8%
128 KB	2-way	0.019	0.0001	0.3%	0.019	100%	0.000	0%
128 KB	4-way	0.019	0.0001	0.3%	0.019	100%	0.000	0%
128 KB	8-way	0.019	0.0001	0.3%	0.019	100%	0.000	0%
256 KB	1-way	0.013	0.0001	0.5%	0.012	94%	0.001	6%
256 KB	2-way	0.012	0.0001	0.5%	0.012	99%	0.000	0%
256 KB	4-way	0.012	0.0001	0.5%	0.012	99%	0.000	0%
256 KB	8-way	0.012	0.0001	0.5%	0.012	99%	0.000	0%
512 KB	1-way	0.008	0.0001	0.8%	0.005	66%	0.003	33%
512 KB	2-way	0.007	0.0001	0.9%	0.005	71%	0.002	28%
512 KB	4-way	0.006	0.0001	1.1%	0.005	91%	0.000	8%
512 KB	8-way	0.006	0.0001	1.1%	0.005	95%	0.000	4%

FIGURE 5.14 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. Figure 5.15 shows the same information graphically. Note that the 2:1 cache rule of thumb (inside front cover) is supported by the statistics in this table through 128 KB: a direct-mapped cache of size *N* has about the same miss rate as a 2-way set-associative cache of size *N*/2. Caches larger than 128 KB do not prove that rule. Note that the Capacity column is also the fully associative miss rate. Data was collected as in Figure 5.6 using LRU replacement.

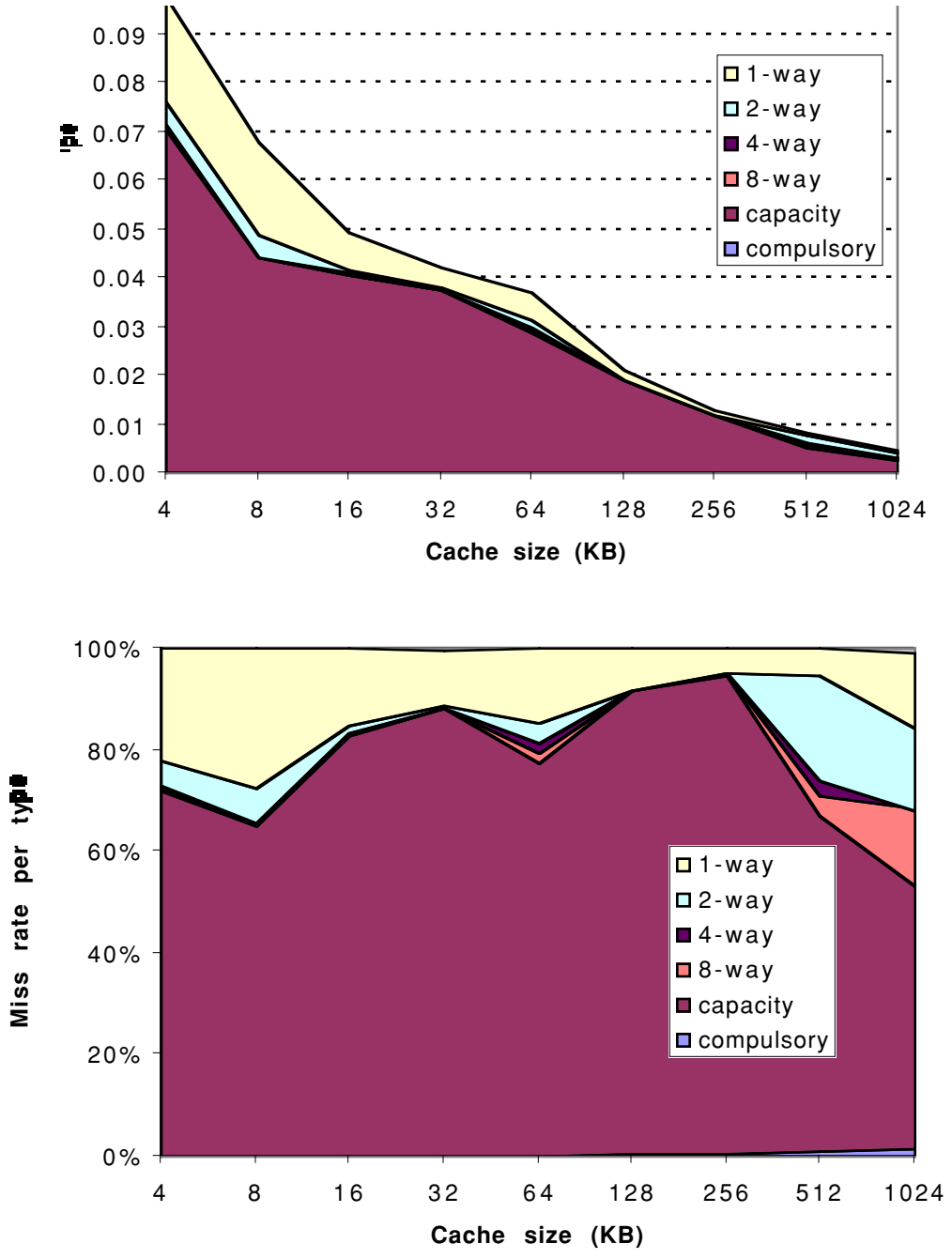

FIGURE 5.15 Total miss rate (top) and distribution of miss rate (bottom) for each size cache according to three C's for the data in Figure 5.14. The top diagram is the actual D-cache miss rates, while the bottom diagram shows percentage in each category. (Space allows the graphs to show one extra cache size than can fit in Figure 5.14.

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 of conflict misses and how they are calculated:

- *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)

As we can see from the figures, the compulsory miss rate of the SPEC2000 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 computer 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 contradicts the three C's model. (Some have proposed using an address trace to determine optimal placement to avoid placement misses from the 3 Cs model; we've not followed that advice here.)

Alas, many of the techniques that reduce miss rates also increase hit time or miss penalty. The desirability of reducing miss rates using the five 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

The simplest way to reduce miss rate is to increase the block size. Figure 5.16 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.

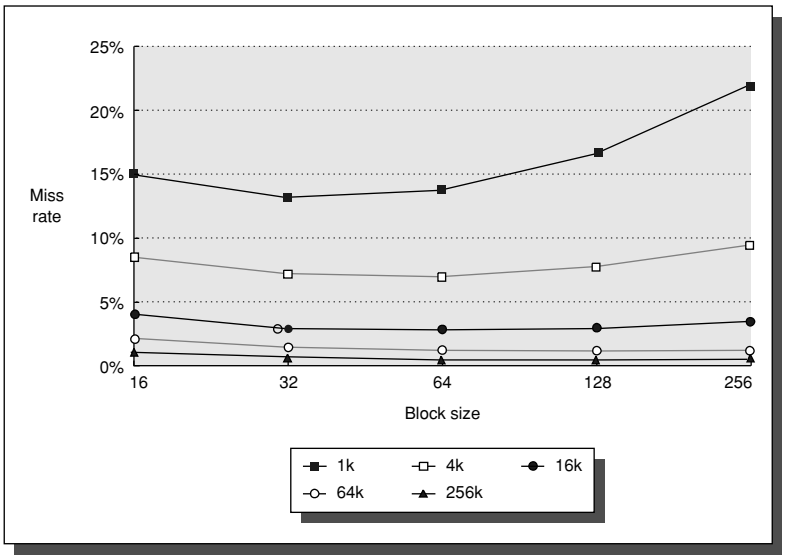

FIGURE 5.16 Miss rate versus block size for five different-sized caches. Note that miss rate actually goes up is the block size is too large relative to the cache size. Each line represents a cache of different size. Figure 5.17 shows the data used to plot these lines. Unfortunately, SPEC2000 traces would take too long if block size was included, so these data are based on SPEC92 on a DECstation 5000 (Gee et al [1993]). **<<Artist: Drop 1K from graph and legend.; then scale Y axis to 0% to 10%>>**

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. 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.

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

FIGURE 5.17 Actual miss rate versus block size for five different-sized caches in Figure 5.16. Note that for a 4-KB cache, 256-byte blocks have the highest 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 5.17 shows the actual miss rates plotted in Figure 5.16. Assume the memory system takes 80 clock cycles of overhead and then delivers 16 bytes every 2 clock cycles. Thus, it can supply 16 bytes in 82 clock cycles, 32 bytes in 84 clock cycles, and so on. Which block size has the smallest average memory access time for each cache size in Figure 5.17?

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

$$\text{Average memory access time} = 1 + (15.05\% \times 82) = 13.341 \text{ clock cycles}$$

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

$$\text{Average memory access time} = 1 + (0.49\% \times 112) = 1.549 \text{ clock cycles}$$

Figure 5.18 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 and 4-KB, and 64 bytes for the larger caches. These sizes are, in fact, popular block sizes for processor caches today.

n

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 band-

		Cache size				
Block size	**Miss penalty**	**1K**	**4K**	**16K**	**64K**	**256K**
16	82	13.341	8.027	4.231	2.673	1.894
32	84	**12.206**	**7.082**	3.411	2.134	1.588
64	88	13.109	7.160	**3.323**	**1.933**	**1.449**
128	96	16.974	8.469	3.659	1.979	1.470
256	112	25.651	11.651	4.685	2.288	1.549

FIGURE 5.18 Average memory access time versus block size for five different-sized caches in Figure 5.16. Block sizes of 32 and 64 byte dominate. The smallest average time per cache size is boldfaced.

width 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. For example, twice the miss penalty of a small block may be close to the penalty of a block twice the size. The larger number of small blocks may also reduce conflict misses. Note that Figures 5.16 and 5.18 above show the difference between selecting a block size based on minimizing miss rate versus minimizing average memory access time.

After seeing the positive and negative impact of larger block size on compulsory and capacity misses, the next two subsections look at the potential of higher capacity and higher associativity.

Second Miss Rate Reduction Technique: Larger caches

The obvious way to reduce capacity misses in Figures 5.14 and 5.15 above is to increases capacity of the cache. The obvious drawback is longer hit time and higher cost. This technique has been especially popular in off-chip caches: The size of second or third level caches in 2001 equals the size of main memory in desktop computers from the first edition of this book, only a decade before!

Third Miss Rate Reduction Technique: Higher Associativity

Figures 5.14 and 5.15 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. You can see the difference by comparing the 8-way entries to the capacity miss column in Figure 5.14, since capacity misses are calculated using fully associative cache. 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 $N/2$. This held for cache sizes less than 128 KB.

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 (see Figure 5.24 on page 431 in section 5.7.)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 higher associativity would increase the clock cycle time as listed below:

$$\text{Clock cycle time}_{\text{2-way}} = 1.36 \times \text{Clock cycle time}_{\text{1-way}}$$
$$\text{Clock cycle time}_{\text{4-way}} = 1.44 \times \text{Clock cycle time}_{\text{1-way}}$$
$$\text{Clock cycle time}_{\text{8-way}} = 1.52 \times \text{Clock cycle time}_{\text{1-way}}$$

Assume that the hit time is 1 clock cycle, that the miss penalty for the direct-mapped case is 25 clock cycles to an L2 cache that never misses, and that the miss penalty need not be rounded to an integral number of clock cycles. Using Figure 5.14 for miss rates, for which cache sizes are each of these three statements true?

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

ANSWER Average memory access time for each associativity is

$$\text{Average memory access time}_{\text{8-way}} = \text{Hit time}_{\text{8-way}} + \text{Miss rate}_{\text{8-way}} \times \text{Miss penalty}_{\text{8-way}} = 1.52 + \text{Miss rate}_{\text{8-way}} \times 25$$
$$\text{Average memory access time}_{\text{4-way}} = 1.44 + \text{Miss rate}_{\text{4-way}} \times 25$$
$$\text{Average memory access time}_{\text{2-way}} = 1.36 + \text{Miss rate}_{\text{2-way}} \times 25$$
$$\text{Average memory access time}_{\text{1-way}} = 1.00 + \text{Miss rate}_{\text{1-way}} \times 25$$

The miss penalty is the same time in each case, so we leave it as 25 clock cycles. For example, the average memory access time for a 4-KB direct-mapped cache is

$$\text{Average memory access time}_{\text{1-way}} = 1.00 + (0.133 \times 25) = 3.44$$

and the time for a 512-KB, eight-way set-associative cache is

$$\text{Average memory access time}_{\text{8-way}} = 1.52 + (0.006 \times 25) = 1.66$$

Using these formulas and the miss rates from Figure 5.14, Figure 5.19 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 8 KB for up to 4-way associativity. Starting with 16 KB, the greater hit time of larger associativity outweighs the time saved due to the reduction in misses.

	Associativity			
Cache size (KB)	One-way	Two-way	Four-way	Eight-way
4	3.44	3.25	3.22	**3.28**
8	2.69	2.58	2.55	**2.62**
16	2.23	**2.40**	**2.46**	**2.53**
32	2.06	**2.30**	**2.37**	**2.45**
64	1.92	**2.14**	**2.18**	**2.25**
128	1.52	**1.84**	**1.92**	**2.00**
256	1.32	**1.66**	**1.74**	**1.82**
512	1.20	**1.55**	**1.59**	**1.66**

FIGURE 5.19 Average memory access time using miss rates in Figure 5.14 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.

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.

n

Fourth Miss Rate Reduction Technique: Way Prediction and Pseudo-Associative Caches and

Another approach reduces conflict misses and yet maintains the hit speed of direct mapped cache. In *way-prediction*, extra bits are kept in the cache to predict the set of the *next* cache access. This prediction means the multiplexor is set early to select the desired set, and only a single tag comparison is performed that clock cycle. A miss results in checking the other sets for matches in subsequent clock cycles.

The Alpha 21264 uses way prediction in its instruction cache. (Added to each block of the instruction cache is a set predictor bit. The bit is used to select which of the two sets to try on the *next* cache access. If the predictor is correct, the instruction cache latency is one clock cycle. If not, it tries the other set, changes the set predictor, and has a latency of three clock cycles. (The latency of the 21264 data cache, which is very similar to its instruction cache, is also three clock cycles.) Simulations using SPEC95 suggested set prediction accuracy is in excess of 85%, so pseudo associativity saves pipeline stages in more than 85% of the instruction fetches.

In addition to improving performance, way prediction can reduce power for embedded applications. By only supplying power to the half of the tags that are expected to be used, the MIPS R4300 series lowers power consumption with the same benefits.

A related approach is called *pseudo-associative* or *column associative*. Accesses proceed 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, a second 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 5.20 shows the relative times. One danger would be if many fast hit times of the direct-mapped cache became slow hit times in the pseudo-associative cache. The performance would then be *degraded* by this optimization. Hence, it is important to indicate for each set which block should be the fast hit and which should be the slow one. One way is simply to make the upper one fast and swap the contents of the blocks. Another danger is that the miss penalty may become slightly longer, adding the time to check another cache entry.

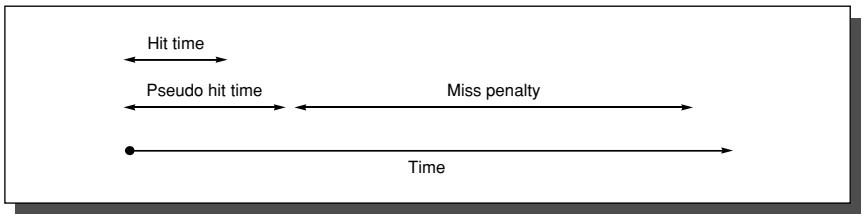

FIGURE 5.20 Relationship between a regular hit time, pseudo hit time, and miss penalty. Basically, pseudoassociativity offers a normal hit and a slow hit rather than more misses.

Fifth Miss Rate Reduction Technique: Compiler Optimizations

Thus far our techniques to reduce misses have required changes to or additions to the hardware: larger blocks, larger caches, higher associativity, or pseudo-associativity. 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 re-

ducing conflict misses. McFarling [1989] looked at using profiling information to determine likely conflicts between groups of instructions. Reordering the instructions reduced 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. Even without that feature, optimized programs on a direct-mapped cache missed less than unoptimized programs on an eight-way set-associative cache of the same size.

Another code optimization aims for better efficiency from long cache blocks. Aligning basic blocks so that the entry point is at the beginning of a cache block decreases the chance of a cache miss for sequential code.

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 two examples, transforming the C code by hand to reduce cache misses.

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. Assuming the arrays do not fit in cache, 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 one cache block before going to the next block. This optimization improves cache performance without affecting the number of instructions executed, unlike the prior example.

Blocking

This optimization 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 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, read the same N elements in a row of y repeatedly, and write one row of N elements of x. Figure 5.21 gives a

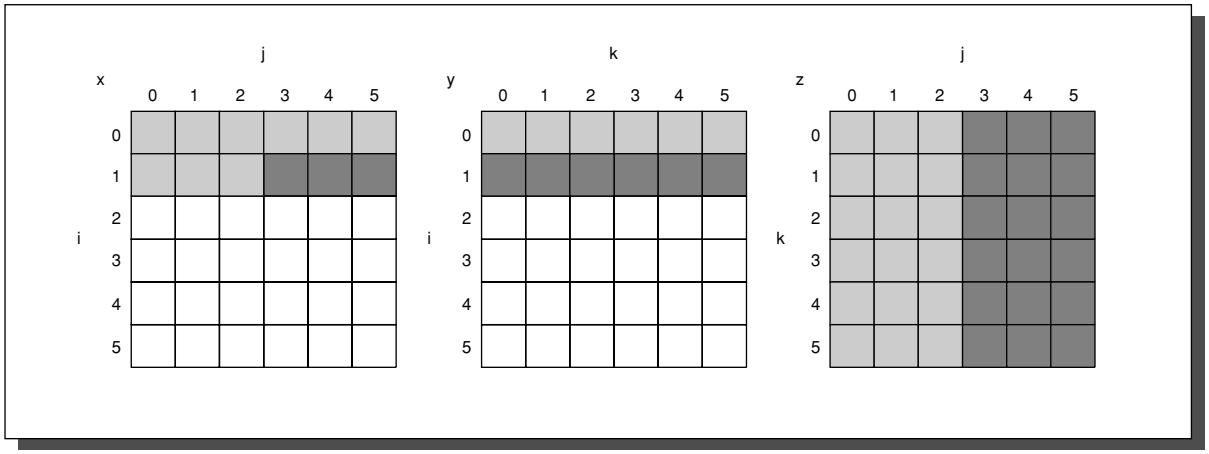

FIGURE 5.21 A snapshot of the three arrays x, y, and z when i = 1. The age of accesses to the array elements is indicated by shade: white means not yet touched, light means older accesses and dark means newer accesses. Compared to Figure 5.22, elements of y and z are read repeatedly to calculate new elements of x. 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. A dark shade indicates a recent access, a light shade indicates an older access, and white means 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 i-th 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 $2N^3 + N^2$ memory words accessed for N^3 operations.

To ensure that the elements being accessed can fit in the cache, the original code is changed to compute on a submatrix of size B by B. Two inner loops now compute in steps of size B rather than the full length 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 5.22 illustrates the accesses to the three arrays using blocking. Looking only at capacity misses, the total number of memory words accessed is $2N^3/B + N^2$. This total is an improvement by about a factor of B. Hence, 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.

Summary of Reducing Cache Miss Rate

This section first presented the three C's model of cache misses: compulsory, capacity, and conflict. This intuitive model led to three obvious optimizations: larger block size to reduce compulsory misses, larger cache size to reduce capacity misses, and higher associativity to reduce conflict misses. Since higher associativity may affect cache hit time or cache power consumption, way prediction checks only a piece of the cache for hits and then on a miss checks the rest. The final technique is the favorite of the hardware designer, leaving cache optimizations to the compiler.

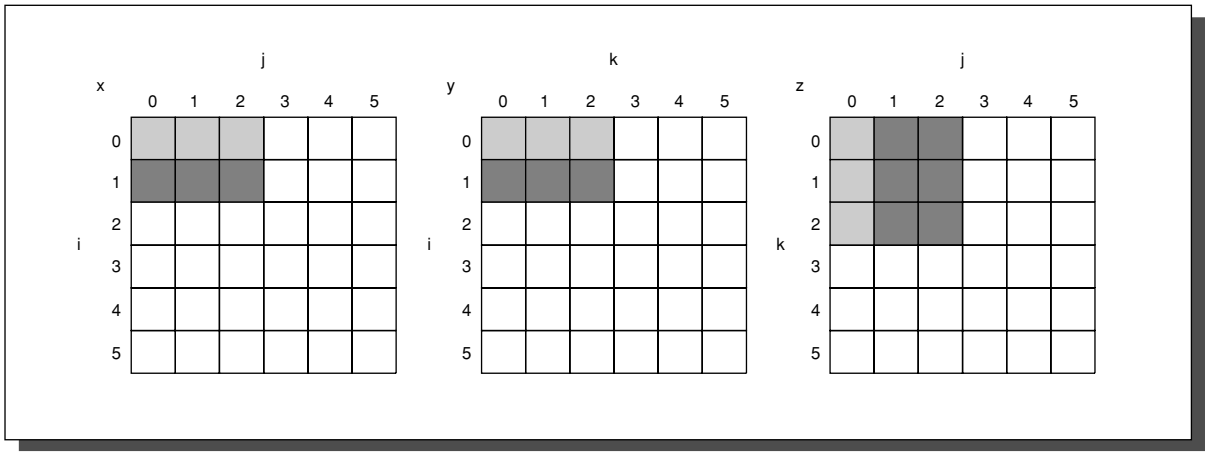

FIGURE 5.22 The age of accesses to the arrays x, y, and z. Note in contrast to Figure 5.21 the smaller number of elements accessed.

The increasing processor-memory gap has meant that cache misses are a primary cause of lower than expected performance. As a result, both algorithms and compilers are changing from the traditional focus of reducing operations to reducing cache misses.

The next section increases performance by having the processor and memory hierarchy operate in parallel, with compilers again playing a significant role in orchestrating this parallelism.

5.6 Reducing Cache Miss Penalty or Miss Rate via Parallelism

This section describes three techniques that overlap the execution of instructions with activity in the memory hierarchy. The first creates a memory hierarchy to match the out-of-order processors, but the second and third work with any type of processor. Although popular in desktop and server computers, the emphasis on efficiency in power and silicon area of embedded computers means such techniques are only found in embedded computers if they are small and reduce power.

First Miss Penalty/Rate Reduction Technique: Nonblocking Caches to Reduce Stalls on Cache Misses

For pipelined computers that allow out-of-order completion (Chapter 3), the CPU need not stall on a cache miss. For example, the CPU could continue fetching in-

structions 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 441). Be aware that hit under miss significantly increases the complexity of the cache controller as there can be multiple outstanding memory accesses.

Figure 5.23 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. Following the discussion in Chapter 3, the number of simultaneous outstanding misses limits achievable instruction level parallelism in programs.

EXAMPLE For the cache described in Figure 5.23, which is more important for floating-point programs: two-way set associativity or hit under one miss? What about 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 5.23 were based on a miss penalty of 16 clock cycles assuming an L2 cache. Although this is low for a miss penalty (we'll see how in the next subsection), 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 5.23 says hit under one miss reduces the average memory stall time to 76% of a blocking cache. Hence, 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

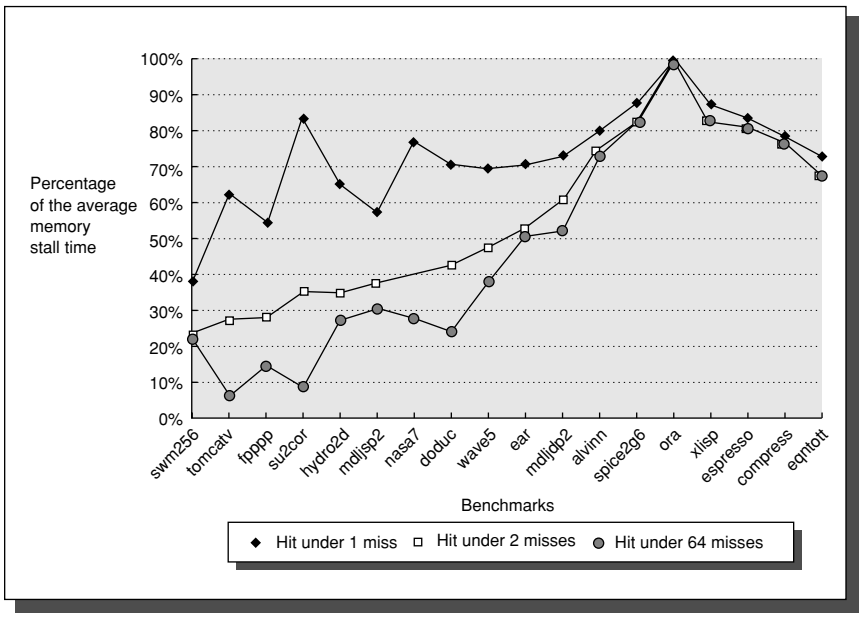

FIGURE 5.23 Ratio of the average memory stall time for a blocking cache to hit-under-miss schemes as the number of outstanding misses is varied for 18 SPEC92 programs. The hit-under-64-misses line allows one miss for every register in the processor. The first 14 programs are floating-point programs: 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, which today would imply a second level cache. These data were generated using the VLIW Multiflow Compiler, which scheduled loads away from use [Farkas and Jouppi 1994].

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

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

The memory stalls of two-way are thus 0.96/1.18 or 81% of direct-mapped cache. The caption of Figure 5.23 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 advantage of hit under miss is that it cannot affect the hit time, as associativity can. n

The real difficulty with performance evaluation of nonblocking caches is that that they imply a dynamic-issue CPU. As a cache miss does not necessarily stall the CPU as it would a static issue CPU. As mentioned on page 395, it is difficult to judge the impact of any single miss, and hence difficult to calculate the average

memory access time. As was the case of a second miss to the remaining block with critical word first above, the effective miss penalty is not the sum of the misses but the non-overlapped time that the CPU is stalled. In general, out-of-order CPUs are capable of hiding the miss penalty of an L1 data cache miss which hits in the L2 cache, but not capable of hiding a significant fraction of an L2 cache miss. Changing the program to pipeline L2 misses can help, especially to a banked memory system (see section 5.8).

Chapter 1 discusses the pros and cons of execution-driven simulation versus trace-driven simulation. Cache studies involving an out-of-order CPUs use execution-driven simulation to evaluate innovations, as avoiding a cache miss that is completely hidden by dynamic issue does not help performance.

An added complexity of multiple outstanding misses is that it is now possible for there to be more than one miss request to the same block. For example, with 64 byte blocks there could be a miss to address 1000 and then later a miss to address 1032. Thus, the hardware must check on misses to be sure it is not to block already being requested to avoid possible incoherency problems and to save time.

Second Miss Penalty/Rate Reduction Technique: Hardware Prefetching of Instructions and Data

Nonblocking caches effectively reduce the miss penalty by overlapping execution with memory access. To have value, we need a processor that can allow instructions to execute out-of-order. Another approach 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. Typically, the processor 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.

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.

The UltraSPARC III uses such a prefetch scheme. A prefetch cache remembers the address used to prefetch the data. If a load hits in prefetch cache, the block is read from the prefetch cache, and the next prefetch request is issued. It calculates the "stride" of the next prefetched block using the difference between current address and the previous address. There can be up to eight simultaneous prefetches in UltraSPARC III.

EXAMPLE What is the effective miss rate of the UltraSPARC III using instruction prefetching? How much bigger an data cache would be needed in the UltraSPARC III to match the average access time if prefetching were removed? It has an 64-KB data cache. Assume prefetching reduces data miss rate by 20%.

ANSWER We assume it takes 1 extra clock cycle if the data 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 50%. Figure 5.8 on page 390 gives the misses per 1000 instructions for an 64-KB data cache as 36.9. To convert to a miss rate, is we assume 22% data references, the rate is

$\frac{36.9}{1000 \times 22/100} = \frac{36.9}{220}$ or 16.7%. Assume the he hit time is 1 clock cycles, and the miss penalty is 15 clock cycles since UltraSPARC III has an L2 cache:

$$\text{Average memory access time}_{\text{prefetch}} = 1 + (16.7\% \times 20\% \times 1) + (16.7\% \times (1 - 20\%) \times 15) = 1 + 0.034 + 2.013 = 3.046$$

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

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

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

$$\text{Miss rate} = \frac{3.046 - 1}{15} = \frac{2.046}{15} = 13.6\%$$

Our calculation suggests that the effective miss rate of prefetching with an 64-KB cache is 13.6%. Figure 5.8 on page 390 gives the misses per 1000 instructions of a 256-KB instruction cache as 32.6, yielding a miss rate of 32.6/(22% x1000) or 14.8%,. If the prefetching reduces miss rate by 20%, then a 64 KB data cache with prefetching outperforms a 256-KB cache without it. n

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

Third Miss Penalty/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. Nonfaulting prefetches simply turn into no-ops if they would normally result in an exception. Using this terminology, a normal load instruction could be considered a "faulting register prefetch instruction."

The most effective prefetch is "semantically invisible" to a program: it doesn't change the contents of registers and memory *and* it cannot cause virtual memory faults. Most processors today offer non-faulting cache prefetches. 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 do not stall but continue to supply instructions and data while waiting for the prefetched data to return. As you would expect, the data cache for such computers is normally nonblocking.

Like hardware-controlled prefetching, the goal is to overlap execution with the prefetching of data. Loops are the important 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 by prefetching. Let's assume we have an 8-KB direct-mapped

data cache with 16-byte blocks, and it is a write-back cache that does write allocate. The elements of `a` and `b` are 8 bytes long as they are double-precision floating-point arrays. There are 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 $3 \times \left\lceil \frac{100}{2} \right\rceil$ 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 times `a` for plus a 101 times for `b`, or 251 misses.

To simplify our optimization, we will not worry about prefetching the first accesses of the loop. These may be already in the cache, or we will pay the miss penalty of first few elements of a or b. Nor we will worry about suppressing the prefetches at the end of the loop which try to prefetch beyond the end of `a` (`a[i][100]`...`a[i][106]`) and end of `b` (`b[100][0]`... `b[106][0]`). If these were faulting prefetches, we could not take this luxury. Let's assume that the miss penalty is so large we prefetch need to start at least, say, seven iterations in advance. (Stated alternatively, we assume prefetching has no benefit until the eighth iteration.)

```
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][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:

- 7 misses for elements b[0][0], b[1][0], ..., b[6][0] in the first loop;
- 4 misses ($\lceil 7/2 \rceil$) for elements a[0][0], a[0][1], ..., a[0][6] for in the first loop (spatial locality reduces misses to one per 16 byte cache block);
- 4 misses ($\lceil 7/2 \rceil$) for elements a[1][0], a[1][1], ..., a[1][6] in the second loop;
- 4 misses ($\lceil 7/2 \rceil$) for elements a[2][0], a[2][1], ..., a[2][6] in the second loop;

or a total of 19 nonprefetched misses. The cost of avoiding 232 cache misses is executing 400 prefetch instructions, 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 100 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 251×100 or 25,100 clock cycles, giving a total of 27,200 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 11×100 or 1100 clock cycles for cache misses, giving a total of 2000. The second loop executes 2×100 or 200 times, and at 8 clock cycles per iteration it takes 1600 clock cycles plus 8×100 or 800 clock cycles for cache misses. This gives a total of 2400 clock cycles. From the prior example we know that this code executes 400 prefetch instructions during the 2000 + 2400 or

4400 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 27,200/4400 or 6.2 times faster.

In addition to the nonfaulting prefetch loads, the 21264 offers prefetches to help with writes. In the example above, we prefetched `a[i][j+7]` even though we were not going to read the data. We just wanted it in the cache so that we could write over it. If an aligned cache block is being written in full, the write hint instruction tells the cache to allocate the block but do not bother loading the data, as the CPU will write over it. In the example above, if the array `a` were properly aligned and padded, such an instruction could replace the instructions prefetching `a` with the write hint instructions, thereby saving hundreds of memory accesses. Since these write hints do have side effects, care would also have to be taken not to access memory outside of the memory allocated for `a`.

Although array optimizations are easy to understand, modern programs are more likely to use pointers. Luk and Mowry [1999] have demonstrated that compiler-based prefetching can sometimes be extended to pointers as well. Of ten programs with recursive data structures, prefetching all pointers when a node is visited improved performance by 4% to 31% in half the programs. On the other hand, the remaining programs were still within 2% of their original performance. The issue is both whether prefetches are to data already in the cache and whether they occur early enough for the data to arrive by the time it is needed.

Summary of Reducing Cache Miss Penalty/ Miss Rate via Parallelism

This section first covered non-blocking caches, which enable out-of-order processors. In general such processors cache hide misses to L1 caches that hit in the L2 cache, but not a complete L2 cache miss. However, if miss under miss is supported, nonblocking caches can take advantage of more bandwidth behind the cache by having several outstanding misses operating at once for programs with sufficient instruction level parallelism.

The hardware and software prefetching techniques leverage excess memory bandwidth for performance by trying to anticipate the needs of a cache. Although speculation may not make sense for power sensitive embedded applications, it normally does for desktop and server computers. The potential success of prefetching is either lower miss penalty, or if they are started far in advance of need, reduction of the miss rate. This ambiguity of whether they help miss rate or miss penalty is one reason they are included in separate section.

Now that we have spent nearly 30 pages on techniques that reduce cache misses or miss penalty in sections 5.4 to 5.6, it is time to look at reducing the final component of average memory access time.

5.7 Reducing Hit Time

Hit time is critical because it affects the clock rate of the processor; in many processors today the cache access time limits the clock cycle rate, even for processors 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 four general techniques.

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. The second suggestion is to keep the cache simple, such as using direct mapping (see page 414). 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. For second level caches, 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.

One approach to determining the impact on hit time in advance of building a chip is to use CAD tools. CACTI is a program to estimate the access time of alternative cache structures on CMOS microprocessors within 10% of more detailed CAD tools. For a given minimum feature size, it estimates the hit time of caches as you vary cache size, associativity, and number of read/write ports. Figure 5.24 shows the estimated impact on hit time as cache size and associativity are varied. Depending on cache size, for these parameters the model suggests that hit times for direct mapped is 1.2 to 1.5 times faster than 2-way set associative; 2-way is l.02 to 1.11 times faster than 4-way; and 4-way is 1.0 to 1.08 times faster than fully associative (except for a 256 KB cache, which is 1.19 times faster).

Although the amount of on-chip cache increased with new generations of microprocessors, the size of the L1 caches has recently not increased between generations. The L1 caches are the same size between the Alpha 21264 and 21364, UltraSPARC II and III, and AMD K6 and Athlon. The L1 data cache size is actually reduced from 16 KB in Pentium III to 8 KB in Pentium 4. The emphasis recently is on fast clock time while hiding L1 misses with dynamic execution and using L2 caches to avoid going to memory.

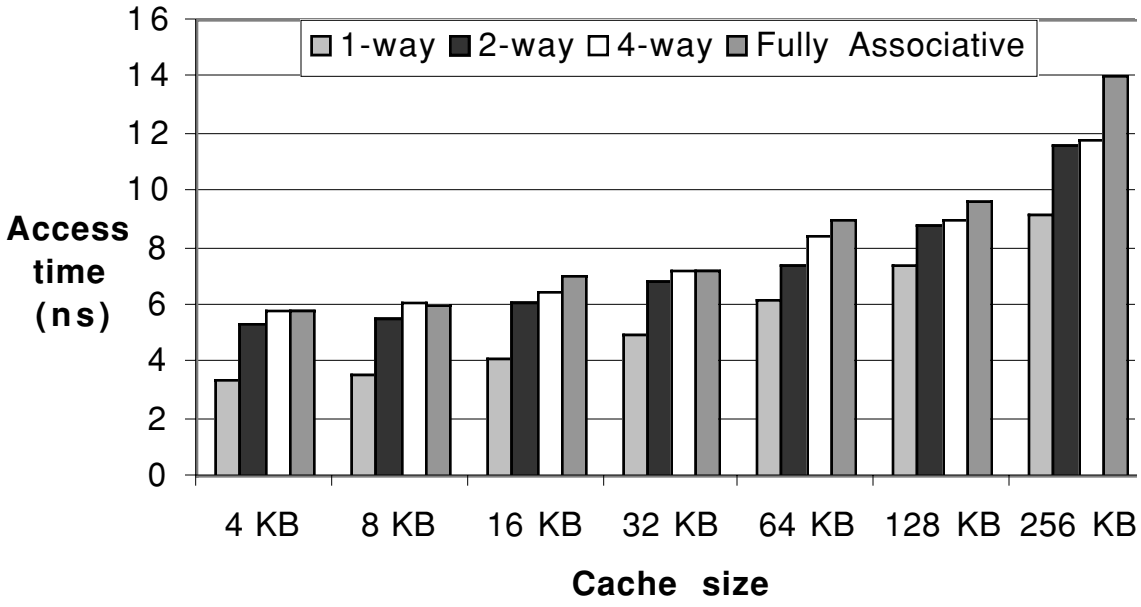

FIGURE 5.24 Access times for as size and associativity vary in a CMOS cache. These data are based on Spice runs used to validate the CACTI model 2.0 by Reinmann and Jouppi [1999]. They assumed 0.80-micron feature size, a single read/write port, 32 address bits, 64 output bits, and 32 byte blocks. The median ratios of access time relative to the direct mapped caches are 1.36, 1.44, and 1.52 for 2-way, 4-way, and 8-way associative caches, respectively.

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 5.10, 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. As we shall shortly see, it is important to distinguish two tasks: indexing the cache and the comparing addresses. Thus, the issues are whether a virtual or physical address is used to index the cache and whether a virtual or physical index is used in the tag comparison. Full virtual addressing for both index and tags eliminates address translation time from a cache hit. Then why doesn't everyone build virtually addressed caches?

One reason is protection. Page level protection is checked as part of the virtual to physical address translation, and it must be enforced no matter what. One solu-

tion is to copy the protection information from the TLB on a miss, add a field to hold it, and check it on every access to the virtually addressed cache.

Another reason is that every time a process is switched, the virtual addresses refer to different physical addresses, requiring the cache to be flushed. Figure 5.25 shows the impact on miss rates of this flushing. One solution is to 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

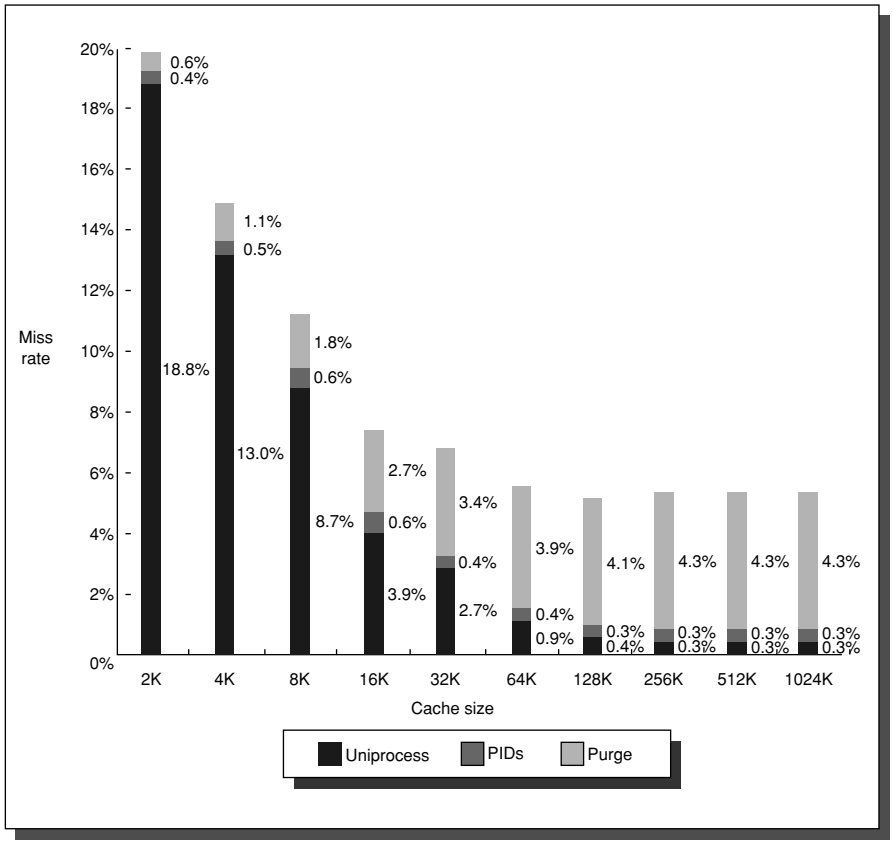

FIGURE 5.25 Miss rate versus virtually addressed cache size of a program measured three ways: without process switches (uniprocess), with process switches using a process-identifier tag (PIDs), and with process switches but without PIDs (purge). PIDs increase the uniprocess absolute miss rate by 0.3% 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.

the cache are for this program. Figure 5.25 shows the improvement in miss rates by using PIDs to avoid cache flushes.

A third 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 to the synonym problem, called *anti-aliasing*, guarantee every cache block a unique physical address. The Alpha 21264 uses a 64 KB instruction cache with an 8 KB page and two-way set associativity, hence the hardware must handle aliases involved with the 2 virtual address bits in both sets. It avoids aliases by simply checking all 8 possible locations on a miss–four entries per set–to be sure that none match the physical address of the data being fetched. If one is found, it is invalidated, so when the new data is loaded into the cache its physical address is guaranteed to be unique.

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. From the perspective of the cache, page coloring effectively increases the page offset, as software guarantees that the last few bits of the virtual and physical page address are identical

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 5.12.)

One alternative to get the best of both virtual and physical caches is to use part of the page offset—the part that is identical in both virtual and physical addresses—to index the cache. At the same time as the cache is being read using that index, the virtual part of the address is translated, and the tag match uses physical addresses.

This alternative allows the cache read to begin immediately and yet the tag comparison is still with physical addresses. The limitation of this *virtually indexed, physically tagged* alternative is that a direct-mapped cache can be no bigger than the page size. For example, in the data cache in Figure 5.7 on page 388, the index is 9 bits and the cache block offset is 6 bits. To use this trick, the virtual page size would have to be at least $2^{(9+6)}$ bytes or 32 KB. If not, a portion of the index must be translated from virtual to physical address.

Associativity can keep the index in the physical part of the address and yet still support a large cache. Recall that size of index is controlled by this formula:

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

For example, doubling associativity and doubling the cache size does not change the size of the index. The Pentium III, with 8-KB pages, avoids translation with its16-KB cache by using 2-way 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 handicap of 4-KB pages in the IBM architecture.

Third Hit Time Reduction Technique: Pipelined Cache Access

The final technique is simply to pipeline cache access so that the effective latency of a first level cache hit can be multiple clock cycles, giving fast cycle time and slow hits. For example, the pipeline for the Pentium takes one clock cycle to access the instruction cache, for the Pentium Pro through Pentium III it takes two clocks, and for the Pentium 4 it takes four clocks. This split increases the number of pipeline stages, 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).

Note that this technique in reality increases the bandwidth of instructions rather that decreasing the actual latency of a cache hit.

Fourth Hit Time Reduction Technique: Trace Caches

A challenge in the effort to find instruction level parallelism beyond four instructions per cycle is to supply enough instructions every cycle without dependencies. One solution is called a *trace cache*. Instead of limiting the instructions in a static cache block to spatial locality, a trace cache finds a dynamic sequence of instructions *including taken branches* to load into a cache block.

The name comes from the cache blocks containing dynamic traces of the executed instructions as determined by the CPU rather than containing static sequences of instructions as determined by memory. Hence, the branch prediction is folded into cache, and must be validated along with the addresses to have a valid fetch. The Intel Netburst microarchitecture, which is the foundation of the Pentium 4 and its successors, uses a trace cache.

Clearly, trace caches have much more complicated address mapping mechanisms, as the addresses are no longer aligned to power of 2 multiple of the word size. However, they have other benefits for utilization of the data portion of the instruction cache. Very long blocks in conventional caches may be entered from a taken branch, and hence the first portion of the block would occupy space in the cache might not be fetched. Similarly, such blocks may be exited by taken branches, so the last portion of the block might be wasted. Given that taken branches or jumps are one in 5 to 10 instructions, space utilization is a real problem for processors like the AMD Athlon, whose 64 byte block would likely include16 to 24 80x86 instructions. The trend towards even greater instruction issue should make the problem worse. Trace caches store instructions only from the branch entry point to the exit of the trace, thereby avoiding such header and trailer overhead.

The downside of trace caches is that they store the same instructions multiple times in the instruction cache. Conditional branches making different choices result in the same instructions being part of separate traces, which each occupy space in the cache.

Cache Optimization Summary

The techniques in sections 5.4 to 5.7 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 5.26 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. Generally no technique helps more than one category.

5.8 Main Memory and Organizations for Improving Performance

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 and multiprocessors. The relationship of main memory and multiprocessors is discussed in Chapter 6, and relationship of main memory and I/O is discussed in Chapter 7.

Technique	Miss penalty	Miss rate	Hit time	Hardware complexity	Comment
Multi-level caches	+			2	Costly hardware; harder if block size L1 ≠ L2; widely used
Critical word first and early restart	+			2	Widely used
Giving priority to read misses over writes	+			1	Trivial for uniprocessor, and widely used
Merging Write Buffer	+			1	Used with write through; in 21164. UltraSPARC III; widely used
Victim caches	+	+		2	AMD Athlon has 8 entries
Larger block size	–	+		0	Trivial; Pentium 4 L2 uses 128 bytes
Larger cache size		+	–	1	Widely used, esp. for L2 caches
Higher associativity		+	–	1	Widely used
Way-predicting caches		+		2	Used in I-cache of UltraSPARC III; D-cache of MIPS R4300 series
Pseudo-associative		+		2	Used in L2 of MIPS R10000
Compiler techniques to reduce cache misses		+		0	Software is challenge; some computers have compiler option
Nonblocking caches	+			3	Used with all out-of-order CPUs
Hardware prefetching of instructions and data	+	+		2 instr., 3 data	Many prefetch instructions; UltraSPARC III prefetches data
Compiler-controlled prefetching	+	+		3	Needs nonblocking cache too; several processors support it
Small and simple caches		–	+	0	Trivial; widely used
Avoiding address translation during indexing of the cache			+	2	Trivial if small cache; used in Alpha 21164, UltraSPARC III
Pipelined cache access			+	1	Widely used
Trace cache			+	3	Used in Pentium 4

FIGURE 5.26 Summary of cache optimizations showing impact on cache performance and complexity for the techniques in sections 5.4 to 5.7. Although generally a technique helps only one factor, prefetching can reduce misses if done sufficiently early; if not, it can reduce miss penalty. + 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.

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. 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 increase block size to take advantage of the high memory bandwidth.

The previous sections describe what can be done with cache organization to reduce this CPU-DRAM 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 this section we examine techniques for organizing memory to improve bandwidth.

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

Given a cache block of four words, and that a word is 8 bytes, the miss penalty is 4 × (4 + 56 + 4) or 256 clock cycles, with a memory bandwidth of one-eighth byte (32/256) per clock cycle. These values are our default case.

Figure 5.27 shows some of the options to faster memory systems. The next three solutions assume generic memory. The next three solutions assume generic memory technology, which we explore in the next section.

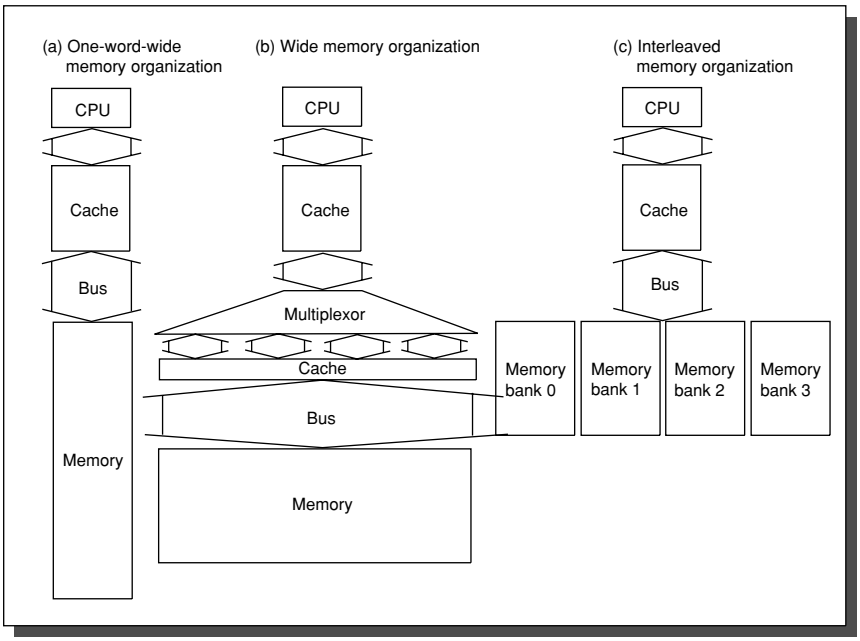

FIGURE 5.27 Three examples of bus width, memory width, and memory interleaving to achieve higher memory bandwidth. (a) is the simplest design, with everything the width of one word; (b) shows a wider memory, bus, and L2 cache with a narrow L1 cache; while (c) shows a narrow bus and cache with an interleaved 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; see Figure 5.27(a). 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 4×64 or 256 clock cycles as calculated above to 2×64 or 128 clock cycles. The reason is at twice the width we need half the memory accesses, and each takes 64 clock cycles. At four words wide the miss penalty is just 1×64 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 multiplexor between the cache and the CPU—and that multiplexor 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; see Figure 5.27(b).

Since main memory is traditionally expandable by the customer, a drawback to wide memory is that the minimum increment is doubled or quadrupled when the width is doubled or quadrupled. In addition, memories with error correction have difficulties with writes to a portion of the protected block, such as 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. (Section 5.15 describes error correction on the Sun Fire 6800 server.) 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 word since most writes are that size.

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 chips 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 chips in the system; in contrast, most memory systems activate only the chips containing the needed words. The two philosophies affect the power of the memory system,

leading to different decisions depending on the relative importance of power versus performance.

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 5.27(c) shows this organization. For example, sending an address to four banks (with access times shown on page 437) yields a miss penalty of 4 + 56 + (4 × 4) or 76 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 5.28 shows this interleaving.

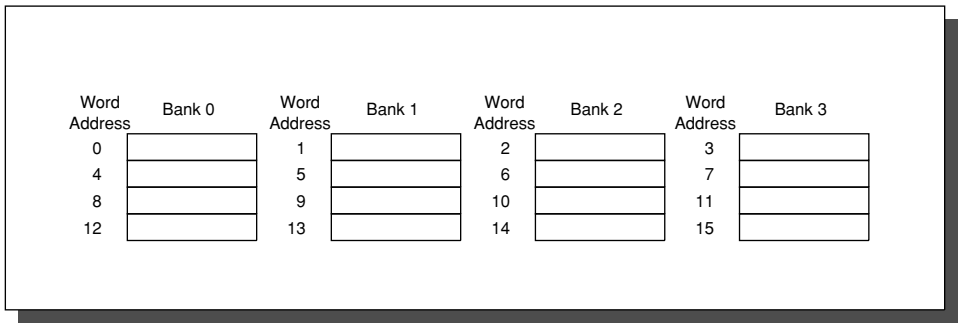

FIGURE 5.28 Four-way interleaved memory. This example assumes word addressing: with byte addressing and eight bytes per word, each of these addresses would be multiplied by eight.

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.

EXAMPLE What can interleaving and wide memory buy? Consider the following description of a computer 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 = 64 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 1.2%. 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 437?

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

$$2 + (1.2 \times 3\% \times 64) = 4.30$$

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:

$$\text{64-bit bus and memory, no interleaving} = 2 + (1.2 \times 2\% \times 2 \times 64) = 5.07$$

$$\text{64-bit bus and memory, interleaving} = 2 + (1.2 \times 2\% \times (4 + 56 + 8)) = 3.63$$

$$\text{128-bit bus and memory, no interleaving} = 2 + (1.2 \times 2\% \times 1 \times 64) = 3.54$$

Thus, doubling the block size slows down the straightforward implementation (5.07 versus 4.30), while interleaving or wider memory is 1.19 or 1.22 times faster, respectively. If we increase the block size to four, the following is obtained:

$$\text{64-bit bus and memory, no interleaving} = 2 + (1.2 \times 1.2\% \times 4 \times 64) = 5.69$$

$$\text{64-bit bus and memory, interleaving} = 2 + (1.2 \times 1.2\% \times (4 + 56 + 16)) = 3.09$$

$$\text{128-bit bus and memory, no interleaving} = 2 + (1.2 \times 1.2\% \times 2 \times 64) = 3.84$$

Again, the larger block hurts performance for the simple case (5.69 vs. 4.30), although the interleaved 64-bit memory is now fastest—1.39 times faster versus 1.22 for the wider memory and bus. n

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 memory bus in this example.

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

$$\text{Number of banks} \geq \text{Number of clock cycles to access word in bank}$$

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 64-bit 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 subsection. 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 512-MB main memory takes 256 memory chips of 4 M × 4 bit, easily organized into 16 banks of 16 memory chips. However, it takes only sixteen 64-M × 4-bit memory chips for 64 MB, making one bank the limit. Many manufacturers will want to have a small memory option in the baseline model. This shrinking number of chips is the main disadvantage of interleaved memory banks. Chips organized with wider paths, such as 16 M × 16 bits, 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 memory chips, or the memory controller changes the interleaving based on the size of physical memory, or both.

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 421) 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 6).

Independent of memory technology, higher bandwidth is available using memory banks, by making memory and its bus wider, or doing both. The next section examines the underlying memory technology.

5.9 Memory Technology

… 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)

The prior section described ways to organize memory chips; this section describes the technology inside the memory chips. Before describing the options, let's go over the performance metrics.

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.

DRAM technology

The main memory of virtually every desktop or server computer sold since 1975 is composed of semiconductor DRAMs,.

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. Figure 5.29 shows the basic DRAM organization. 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*. To pack more bits per chip, DRAMs use only a single transistor to store a bit. Reading that bit can disturb the information, however. 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.

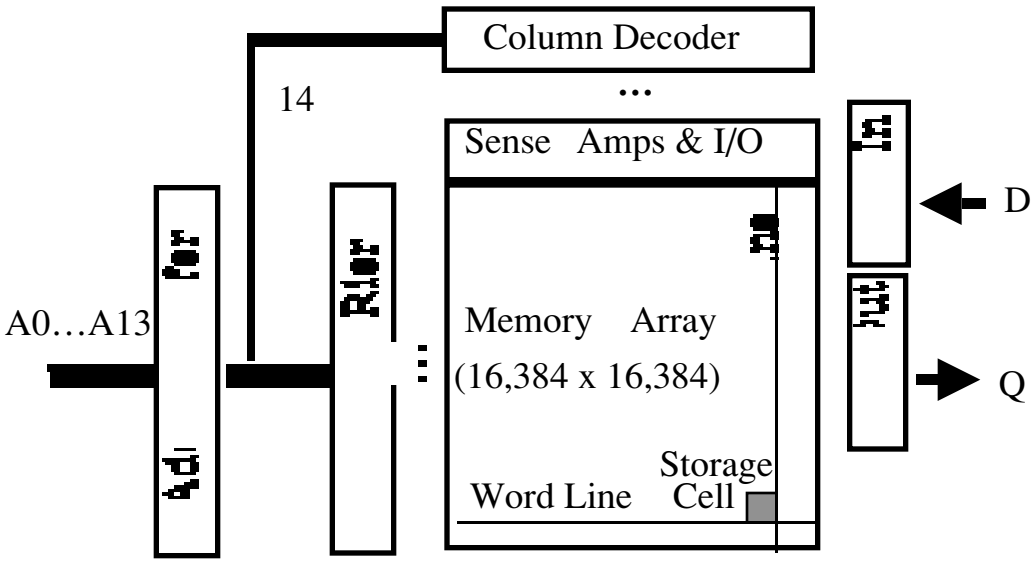

FIGURE 5.29 Internal organization of a 64-Mbit DRAM. DRAMs often use banks of memory arrays internally, and select between them. For example, instead of one 16,384 x 16,384 memory, a DRAM might use 256 1,024 x 1,024 arrays or 16 2,048 x 2,048 arrays.

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.

Earlier sections presented main memory as if operated like a Swiss train, consistently delivering the goods exactly according to schedule. Refresh belies that myth, for some accesses take much longer than others do. Thus, refresh is another reason for variability of memory latency and hence cache miss penalty.

Amdahl suggested a rule of thumb that memory capacity should grow linearly with CPU speed to keep a balanced system, so that a 1000 MIPS processor should have 1000 megabytes of memory. CPU designers rely on DRAMs to supply that demand: in the past they expected a four-fold improvement in capacity every three years, or 55% 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, which is related to latency, of about 5% per year. The CAS or Data Transfer Time, which is related to bandwidth, is growing at more than twice that rate.

Year of introduction	Chip size	Row access strobe (RAS)		Column access strobe (CAS) / Data Transfer Time	Cycle time
		Slowest DRAM	Fastest DRAM		
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
1996	64 Mbit	70 ns	50 ns	12 ns	110 ns
1998	128 Mbit	70 ns	50 ns	10 ns	100 ns
2000	256 Mbit	65 ns	45 ns	7 ns	90 ns
2002	512 Mbit	60 ns	40 ns	5 ns	80 ns

FIGURE 5.30 Times of fast and slow DRAMs with each generation. Performance improvement of row access time is about 5% per year. The improvement by a factor of two in column access accompanied the switch from NMOS DRAMs to CMOS DRAMs.

Although we have been talking about individual chips, DRAMs are commonly sold on small boards called *DIMMs* for *Dual Inline Memory Modules*. DIMMs typically contain 4 to 16 DRAMs. They are normally organized to be eight bytes wide for desktop systems.

In addition to the DIMM packaging and the new interfaces to improve the data transfer time, discussed in the following subsections, the biggest change to DRAMs has been a slowing down in capacity growth. For 20 years DRAMs obeyed Moore's Law, bringing out a new chip with four times the capacity every three years. As a result of a slowing in demand for DRAMs, since 1998 new chips only double capacity every two years. In 2001, this new slower pace shows no sign of changing.

Just as virtually all desktop or server computer since 1975 used DRAMs for main memory, virtually all use SRAM for cache, the topic of the next subsection.

SRAM Technology

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 typically use six transistors per bit to prevent the information from being disturbed when read.

This difference in refresh alone can make a difference for embedded applications. Devices often go into low power or standby mode for long periods. SRAM needs only minimal power to retain the charge in standby mode, but DRAMs must continue to be refreshed occasionally so as to not lose information.

In DRAM designs the emphasis is on cost per bit and capacity, while SRAM designs are concerned with speed and capacity. (Because of this concern, SRAM address lines are not multiplexed.). Thus, unlike DRAMs, there is no difference between access time and cycle time. 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.

Embedded Processor Memory Technology: ROM and Flash

Embedded computers usually have small memories, and most do not have a disk to act as non-volatile storage. Two memory technologies are found in embedded computers to address this problem.

The first is *Read-Only Memory* (*ROM*). ROM is programmed at time of manufacture, needing only a single transistor per bit to represent 1 or 0. ROM is used for the embedded program and for constants, often included as part of a larger chip.

In addition to being non-volatile, ROM is also non-destructible; nothing the computer can do can modify the contents of this memory. Hence, ROM also provides a level of protection to the code of embedded computers. Since address-based protection is often not enabled in embedded processors, ROM can fulfill an important role.

The second memory technology offers non-volatility but allows the memory to be modified. *Flash memory* allows the embedded device to alter nonvolatile memory after the system is manufactured, which can shorten product development. Flash memory, described in on page 498 in Chapter 7, allows reading at almost DRAM speeds but writing flash is 10 to 100 times slower. In 2001, the DRAM capacity per chip and the megabytes per dollar is about four to eight times greater than flash memory.

Improving Memory Performance in a standard DRAM Chip

As Moore's Law continues to supply more transistors and as the processor-memory gap increases pressure on memory performance, some of the ideas of the prior section have made their way inside the DRAM chip. Generally the idea has been for greater bandwidth, often at the cost of greater latency. This subsection presents techniques that take advantage of the nature of DRAMs.

As mentioned earlier, a 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.

Although presented logically as a single monolithic array of memory bits, the internal organization of DRAM actually consists of many memory modules. For a variety of manufacturing reasons, these modules are usually 1 to 4 megabits. Thus, if you were to examine a 256 Mbit DRAM under a microscope, you might see 128 2-megabit memory arrays on the chip. This large number of arrays internally presents the opportunity to provide much higher bandwidth off chip.

To improve bandwidth, there have been a variety of evolutionary innovations over time. The first was timing signals that allow repeated accesses to the row buffer without another row access time, typically called *fast page mode*. Such a buffer comes naturally, as each array will buffer 1024 to 2048 bits for each access.

The second major change is that conventional DRAMs have an asynchronous interface to the memory controller, and hence every transfer involves overhead to synchronize with the controller. The solution was to add a clock signal to the DRAM interface, so that the repeated transfers would not bear that overhead. This optimization is called *Synchronous DRAM*, abbreviated *SDRAM*. SDRAMs typically also have a programmable register to hold the number of bytes requested, and hence can send many bytes over several cycles per request.

In 2001 the bus rates are 100 MHz to 150 MHz. SDRAM DIMMs of these speeds are called PC100, PC133, and PC 150, based on the clock speed of the individual chip. Multiplying the eight-byte width of the DIMM times the clock rate, the peak speed per memory module is 800 to 1200 MB/sec.

The third major DRAM innovation to increase bandwidth is to transfer data on both the rising edge and falling edge of the DRAM clock signal, thereby doubling the peak data rate. This optimization is called *Double Data Rate*, and abbreviated *DDR*. The bus speeds for these DRAMs are also 100 to 150 MHz, but these DDR DIMMs are confusingly labeled by the peak *DIMM* bandwidth. The name PC1600 comes from100 MHz $\times$ 2 $\times$ 8 bytes or 1600 Megabytes/second, 133 MHz leads to PC2100, 150 MHz yields PC2400, and so on.

In each of the three cases the advantage of such optimizations is that they add a small amount of logic to exploit the high internal DRAM bandwidth, adding little cost to the system while achieving a significant improvement in bandwidth. Unlike traditional interleaved memories, there is no danger in using such a mode as DRAM chips increase in capacity.

Improving Memory Performance via a new DRAM Interface: RAMBUS

Recently new breeds of DRAMs have been produced that further optimize the interface between the DRAM and CPU. The company RAMBUS takes the standard DRAM core and provides a new interface, making a single chip act more like a memory system than a memory component: each chip has interleaved memory and a high speed interface. RAMBUS licenses its technology to companies that use its interface, both DRAM and microprocessor manufacturers.

The first generation RAMBUS interface 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. It is typically called *RDRAM*. (Such a bus is called a *packet-switched bus* or *split-transaction* bus, described in Chapters 7 and 8.) 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. RDRAM offered a byte-wide interface, and was one of the first DRAMs to use a clock signal, and it also transfers on both edges of its clock. Inside each chip were four banks, each with their own row buffer. To run at its 300 Mhz clock, the RAMBUS bus is limited to be no more than 4 inches long. Typically a microprocessor uses a single RAMBUS channel, so just one RDRAM is transferring at a time.

The second generation RAMBUS interface, called *Direct RDRAM* or *DRDRAM*, offers up to 1.6 GBytes/second of bandwidth from a single DRAM. Innovations in this interface include a separate row- and column-command buses instead of the conventional multiplexing; an 18-bit data bus; expanding from 4 to 16 internal banks per RDRAM to reduce bank conflicts; increasing the number of row buffers from 4 to 8; increasing the clock to 400 MHz clock; and a much more sophisticated controller on chip. Because of the separation of data, row, and column buses, three transactions can be performed simultaneously.

RAMBUS helped set the new optimistic naming trend, calling the 350 MHz part PC700, the 400 MHz part PC800, and so on. Since each chip is 2 bytes wide, the peak chip bandwidth of PC700 is 1400 MB/second, PC800 is 1600 MB/second, and so on. RAMBUS chips are not sold in DIMMs but in "RIMMs" which is similar in size but incompatible with DIMMs. RIMMs are designed to have a single RAMBUS chip on the RIMM supply the memory bandwidth needs of the computer, and are not interchangeable with DIMMs.

Comparing RAMBUS and DDR SDRAM

How does the RAMBUS interface compare in cost and performance when placed in a system? Most main memory systems already use SDRAM to get more bits per memory access, in the hope of reducing the CPU-DRAM performance gap. Since the most computers use memory in DIMM packages, which are typically at least 64-bits wide, the DIMM memory bandwidth is closer to what RAMBUS provides than you might expect when just comparing DRAM chips.

The one note of caution is that performance of cache based systems are based in part on latency to the first byte and in part on the bandwidth to deliver the rest of the bytes in the block. Although these innovations help with the latter case, none help with latency. Amdahl's Law reminds us of the limits of accelerating one piece of the problem while ignoring another part.

In addition to performance, the new breed of DRAMs such as RDRAM and DRDRAM have price a premium over traditional DRAMs to provide the greater

bandwidth since these chips are larger. The question over time is how much more. In 2001 it is factor of two; Section 5.16 has a detailed price-performance evaluation.

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.

5.10 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 protection provided by virtual memory is essential for current computers, sharing is not the reason that 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 computer, 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 5.31 shows the mapping of virtual memory to physical memory for a program with four pages.

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 5.31 can be placed any-

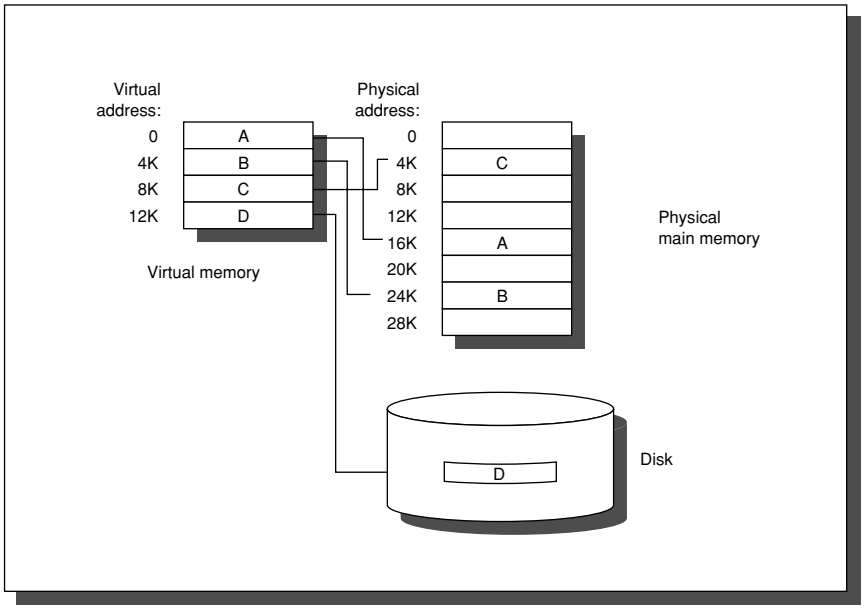

FIGURE 5.31 The logical program in its contiguous virtual address space is shown on the left. It consists of four pages A, B, C, and D. The actual location of three of the blocks is in physical main memory and the other is located on the disk.

where in physical memory or disk just by changing the mapping between them. (Prior to the popularity of virtual memory, processors 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 ideas from Chapter 1 about caches are analogous to virtual memory, although many of the 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 5.32 shows a typical range of memory-hierarchy parameters for virtual memory.

There are further differences between caches and virtual memory beyond those quantitative ones mentioned in Figure 5.32:

- 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 good decision, so the operating system can be involved and spend take 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.

Parameter	First-level cache	Virtual memory
Block (page) size	16-128 bytes	4096-65,536 bytes
Hit time	1-3 clock cycles	50-150 clock cycles
Miss penalty	8-150 clock cycles	1,000,000-10,000,000 clock cycles
(Access time)	(6-130 clock cycles)	(800,000-8,000,000 clock cycles)
(Transfer time)	(2-20 clock cycles)	(200,000-2,000,000 clock cycles)
Miss rate	0.1-10%	0.00001- 0.001%
Address mapping	25- 45 bit physical address to 14- 20 bit cache address	32-64 bit virtual address to 25-45 bit physical address

FIGURE 5.32 Typical ranges of parameters for caches and virtual memory. Virtual memory parameters represent increases of 10 to 1,000,000 times over cache parameters. Normally first level caches contain at most 1 megabyte of data while physical memory contains 32 megabytes to 1 terabyte.

- 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. In fact, the file system occupies most of secondary storage. It is not normally in the address space.

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 typically fixed at 4096 to 65,536 bytes, while segment size varies. The largest segment supported on any processor ranges from 2^{16} bytes up to 2^{32} bytes; the smallest segment is 1 byte. Figure 5.33 shows how the two approaches might divide code and data.

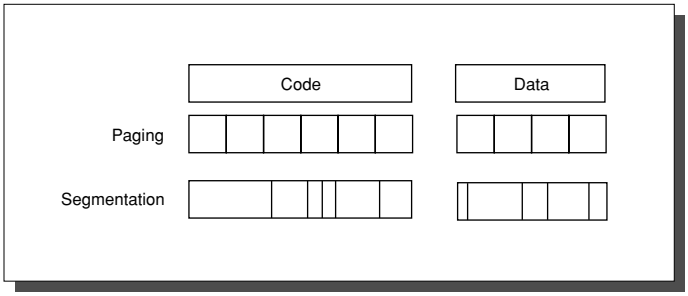

FIGURE 5.33 Example of how paging and segmentation divide a program.

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 re-

quires 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 5.34 summarizes the arguments. Because of the replacement problem (the third line of the figure), few computers today use pure segmentation. Some computers 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 computer to offer multiple page sizes, with the larger sizes being powers of two times the smallest page size. The IBM 405CR embedded processor, for example, allows 1 KB, 4 KB ($2^2 \times 1$ KB), 16 KB ($2^4 \times 1$ KB), 64 KB ($2^6 \times 1$ KB), 256 KB ($2^8 \times 1$ KB), 1024 KB ($2^{10} \times 1$ KB), and 4096 KB ($2^{12} \times 1$ KB) to act as a single page.

	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 5.34 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.

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 5.4 (page 382), 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 5.35).

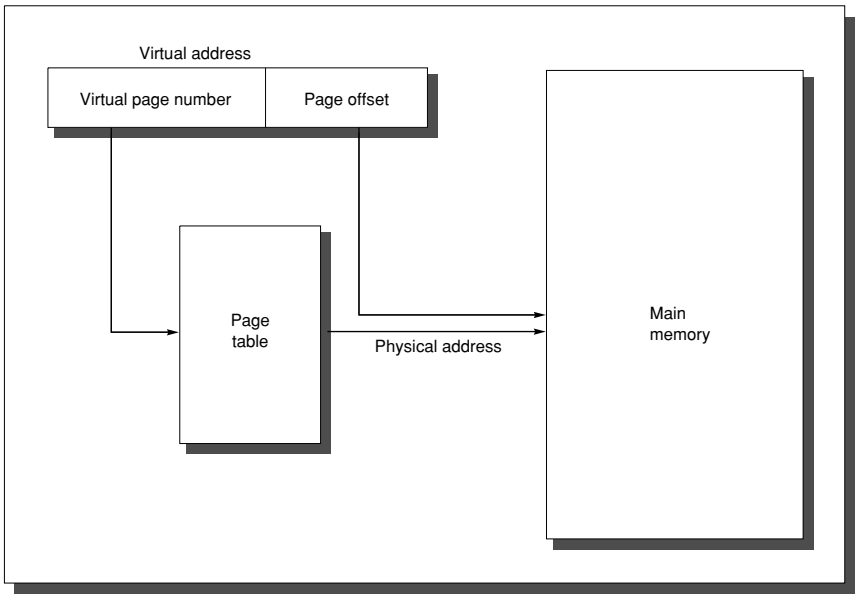

FIGURE 5.35 The mapping of a virtual address to a physical address via a page table.

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 32-bit virtual address, 4-KB pages, and 4 bytes per page table entry, the size of the page table would be $(2^{32}/2^{12}) \times 2^2 = 2^{22}$ or 4 MB.

To reduce the size of this data structure, some computers apply a hashing function to the virtual address. The hash allows the data structure to 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 512-MB physical memory would only need 1 MB (8×512 MB/4 KB) for an inverted page table; the extra 4 bytes per page table entry is for the virtual address. The HP/Intel IA-64 covers both bases by offering both traditional pages tables *and* inverted page tables, leaving the choice of mechanism to the operating system programmer.

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 replace the least-recently used (LRU) block, because if the past predicts the future, that is the one less likely to be needed.

To help the operating system estimate LRU, many processors provide a *use bit* or *reference bit*, which is logically set whenever a page is accessed. (To reduce work, it is actually set only on a translation buffer miss, which is described shortly.) 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 writes through main memory 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. It allows blocks to be written to disk only if they have been altered since being read 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. Paging 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. The operating system resets these bits by changing the value in the page table and then invalidating the corresponding TLB entry. When the entry is reloaded from the page table, the TLB gets an accurate copy of the bits.

Figure 5.36 shows the Alpha 21264 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.

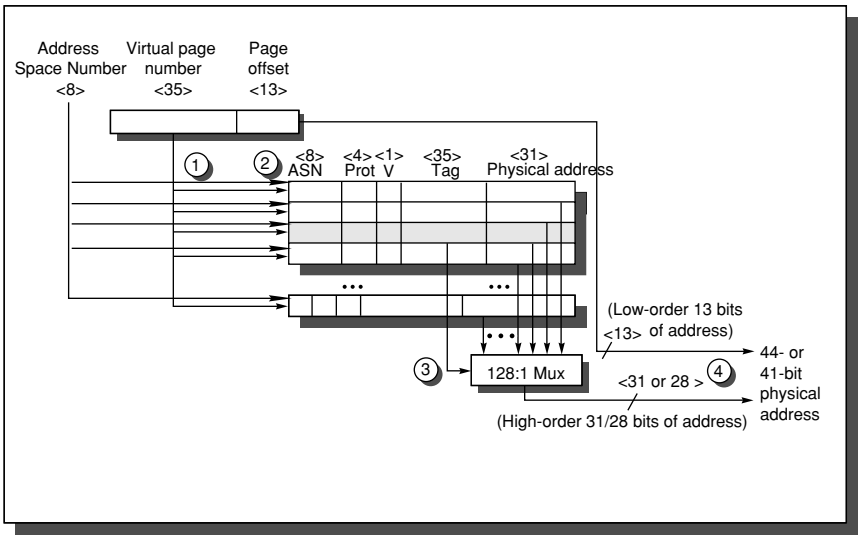

FIGURE 5.36 Operation of the Alpha 21264 data TLB during address translation. The four steps of a TLB hit are shown as circled numbers. The Address Space Number (ASN) is used like a Process ID for virtual caches, in that the TLB is not flushed on a context switch, only when ASNs are recycled. The next fields of an entry are protection permissions (Prot) and the valid bit (V). 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 nor during a TLB miss. Alpha 21264 has an option of either 44-bit or 41-bit physical addresses. This TLB has 128 entries.

To reduce TLB misses due to context switches, each entry has an 8-bit Address Space Number or ASN, which plays the same role as a Process ID number

mentioned in Figure 5.25 on page 432. If the context switching returns to the process with the same ASN, it can still match the TLB. Thus, the process ASN and the PTE ASN must also match for a valid tag.

For reasons similar to those in the cache case, there is no need to include the 13 bits of the Alpha 21264 page offset in the TLB. The matching tag sends the corresponding physical address through effectively a 128:1 multiplexor (step 3). The page offset is then combined with the physical page frame to form a full physical address (step 4). The address size is 44 or 41 bits depending on a physical address mode bit (see section 5.11).

Address translation can easily be on the critical path determining the clock cycle of the processor, so the 21264 uses a virtually addressed instruction cache, thus the TLB is only accessed during an instruction cache miss.

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 433 in section 5.7, a larger page size can allow larger caches with 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 amount is negligible for computers with hundreds of megabytes of memory and page sizes 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 a large page size would lengthen the time to invoke a process

Summary of Virtual Memory and Caches

With virtual memory, TLBs, first level caches, and second levels caches all mapping portions of the virtual and physical address space, it can get confusing what bits go where. Figure 5.37 gives a hypothetical example going from a 64-bit virtual address to a 41 bit physical address with two levels of cache. This L1 cache is virtually indexed, physically tagged since both the cache size and the page size are 8 KB. The L2 cache is 4 MB. The block size for both is 64 bytes.

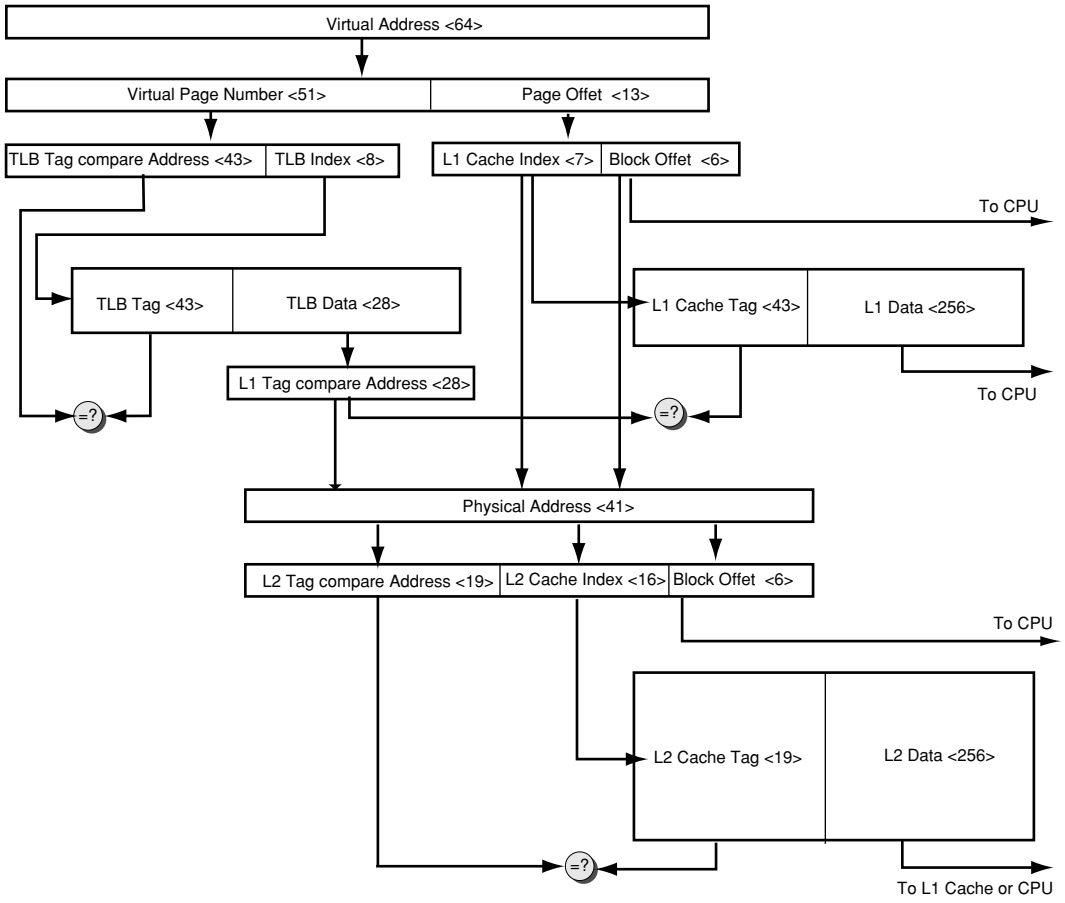

FIGURE 5.37 The overall picture of an hypothetical memory hierarchy going from virtual address to L2 cache access. The page size is 8 KB. The TLB is direct mapped with 256 entries. The L1 cache is a direct-mapped 8 KB and the L2 cache is a direct-mapped 4 MB. Both using 64 byte blocks. The virtual address is 64 bits and the physical address is 41 bits. The primary difference between this simple figure and a real cache, as in Figure 5.43 on page 472 , is replication of pieces of this figure.

First, the 64-bit virtual address is logically divided into a virtual page number and page offset. The former is sent to the TLB to be translated into a physical address, and the latter is sent to the L1 cache act as an index. If the TLB match is a hit, then the physical page number is sent to the L1 cache tag to check for a match. If it matches, its a L1 cache hit. The block offset then selects the word for the CPU.

If the L1 cache check results in a miss, the physical address is then used to try the L2 cache. The middle portion of the physical address is used as an index to the 4 MB L2 cache. The resulting L2 Cache Tag is compared to the upper part of the physical address to check for a match. If it matches, we have a L2 cache hit, and the data is sent to the CPU, which uses the block offset to select the desired word. On an L2 miss, the physical address is then used to get the block from memory.

Although this is a simple example, the major difference between this drawing and a real cache is replication. First, there is only one L1 cache. When there are two L1 caches, the top half of the diagram is duplicated. Note this would lead to two TLBs, which is typical. Hence, one cache and TLB is for instructions, driven from the PC, and one cache and TLB is for data, driven from the effective address. The second simplification is that all the caches and TLBs are direct mapped. If any were N-way set associative, then we would replicate each set of tag memory, comparators, and data memory N times and connect data memories with a N:1 multiplexor to select a hit. Of course, if the total cache size remained the same, the cache index would also shrink by N bits according to the formula in Figure 5.9 on page 397 .

5.11 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 demands 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 computers. Thus, at any instant it must be possible to switch from one process to another. This exchange 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 designers of

the computer and the operating system. The computer designer must ensure that the CPU portion of the process state can be saved and restored. The operating system designer 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. However, 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 division 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 state 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 be mapped 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. The "civilian" programs 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 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 Memory Management and the 21264 TLB

The Alpha architecture uses a combination of segmentation and paging, providing protection while minimizing page table size. With 48-bit virtual addresses, the 64-bit address space is first divided into three segments: *seg0* (bits 63 - 47 = 0...00), *kseg* (bits 63 - 46 = 0...10), and *seg1* (bits 63 to 46 = 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 5.38 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.

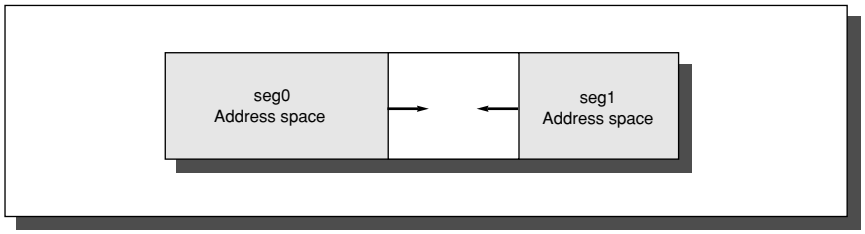

FIGURE 5.38 The organization of seg0 and seg1 in the Alpha. User processes live in seg0, while seg1 is used for portions of the page tables. seg0 includes a downward growing stack, text and data, and an upward growing heap.

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. Figure 5.39 shows address translation in the Alpha. The addresses for each of these page tables come from three "level" fields, labeled level1, level2, and level3. 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 ar-

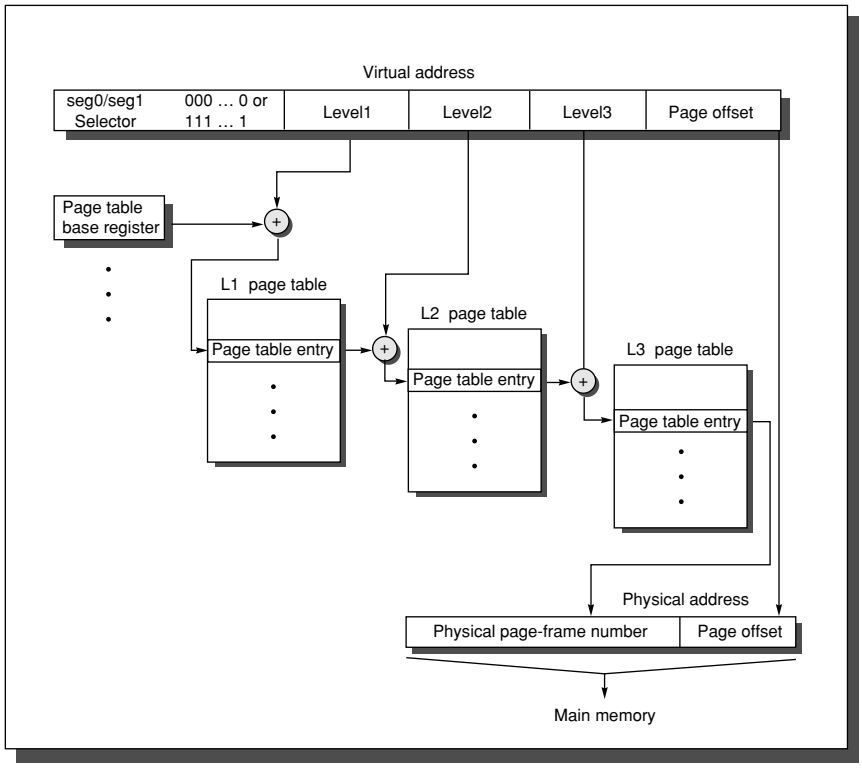

FIGURE 5.39 The mapping of an Alpha virtual address. This figure/description shows the 21264's virtual memory implementation with 3 page table levels, which supports an effective physical address size of 41 bits (allowing access up to 2^40 bytes of memory and 2^40 I/O addresses). Each page table is exactly one page long, so each level field is n bits wide where 2^n = page size/8. The Alpha 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 original 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 21264 also can support a 4-level page table structure (with a level 0 page table field in virtual address bits 43-52) that can allow full access to its 44-bit physical address and 48-bit virtual address while keeping page sizes to 8 KB. That mode is not depicted here. In addition, the size depends on the operating system. VMS does not require KSEG like UNIX does, so VMS could reach the entire 44-bit physical address space with 3-level page tables. The 41-bit physical address restriction comes from the fact that some operating systems need the KSEG section.

chitecture is constrained to fit within a single page. The first three levels (0, 1, and 2) use physical addresses that need no further translation, but Level 3 is mapped virtually. These normally hit the TLB, but if not, the table is accessed a second time with physical addresses.

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

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 bottom-level PTE, checking the others 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 21264 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. This derivation leaves $64 - (3 \times 10 + 13)$ or 21 bits to be defined. If this is a seg0 address, the most-significant bit of the level 1 field is a 0, and for seg1 the two most-significant bits of the level 1 field 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 restriction means the 21264 virtual addresses are really much shorter than the full 64 bits found in registers.

The maximum virtual address and physical address is then tied to the page size. The original 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 $3 \times 13 + 16$ or 55 bits and the maximum physical address to $32 + 16$ or 48 bits. In fact, the upcoming 21364 supports both. It will be interesting to see whether or not operating systems accommodate such expansion plans.

Although 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 Alpha 21264 employs two TLBs to reduce address translation time, one for instruction accesses and another for data accesses. Figure 5.40 shows the important parameters. 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 5.36 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	Same for Instruction and Data TLBs: 128 PTEs per TLB, each of which can map 1, 8, 64, or 512 pages
Block selection	Round robin
Write strategy	(Not applicable)
Block placement	Fully associative

FIGURE 5.40 Memory-hierarchy parameters of the Alpha 21264 TLB.

Memory management in the Alpha 21264 is typical of most desktop or server computers today, relying on page-level address translation and correct operation of the operating system to provide safety to multiple processes sharing the computer. 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 (today called IA-32), 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 IA-32 has separate stacks for each level to avoid security breaches between the levels. There are also data structures analogous to Alpha page tables that con-

tain 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 IA-32 divides the address space, allowing both the operating system and the user access to the full space. The IA-32 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 IA-32 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 IA-32 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 IA-32 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 IA-32 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 5.41 shows examples of IA-32 segment descriptors.

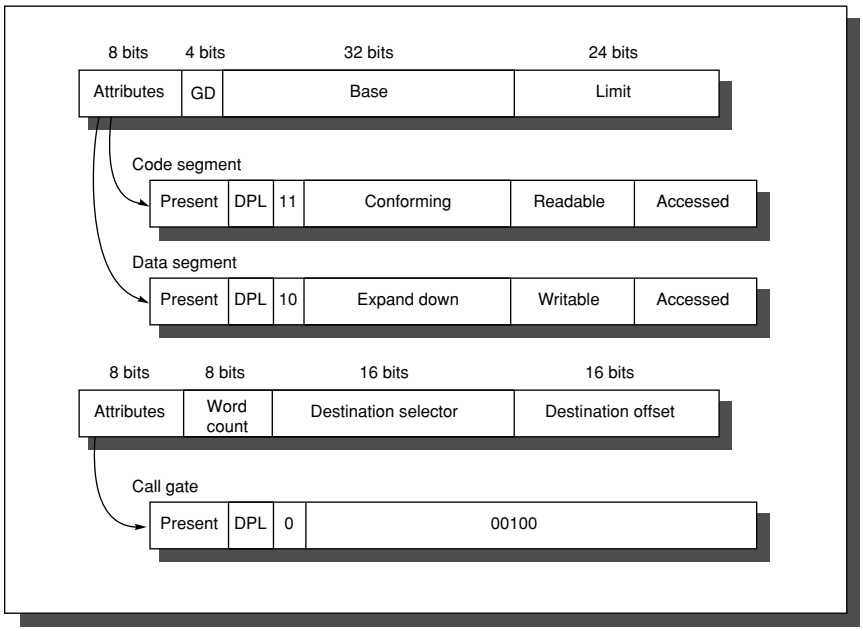

FIGURE 5.41 The IA-32 segment descriptors are distinguished by bits in the attributes field. *Base*, *limit, present*, *readable*, and *writable* are all self-explanatory. D gives the default addressing size of the instructions: 16 bits or 32 bits. G gives the granularity of the segment limit: 0 means 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 IA-32 protection model.

IA-32 provides an optional paging system in addition to this segmented addressing. The upper portion of the 32-bit address selects the segment descriptor and the middle portion is 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 IA-32 segment register with an index to the table *and* a bit saying which table it desires. The operation is checked according to the 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. It is given a descriptor with the writable field set (Figure 5.41). The payroll program invokes the trusted code using a code segment descriptor with the conforming field set. This setting 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 IA-32 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 IA-32 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 5.41. 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 copying 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 IA-32 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 IA-32 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 IA-32

If the IA-32 protection model looks harder to build than the Alpha model, that's because it is. This effort must be especially frustrating for the IA-32 engineers, since few customers use the elaborate protection mechanism. In addition, the fact that the protection model is a mismatch for the simple paging protection of UNIX-like systems means it will be used only by someone writing an operating system especially for this computer.

In the last edition we wondered whether the popularity of the Internet would lead to demands for increased support for security, and hence put this elaborate protection model to good use. Despite widely documented security breaches and the ubiquity of this architecture, no one has proposed a new operating system to leverage the 80x86 protection features.

5.12 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 Chapters 3 and 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.

Some processors increase complexity of instruction fetch by allowing instructions to be issued to be found on any boundary instead of, say, a multiple of four words. 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 cannot benefit.

For example, the UltraSPARC III fetches up to 4 instructions per clock cycle, and executes up to 4, with up to 2 being loads or stores. Hence, the instruction cache must deliver 128 bits per clock cycle and the data cache must support two 64-bit accesses per clock cycle.

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 421).

In reality, the penalty of the an L2 miss is so large that compilers normally only speculate on L1 misses. Figure 5.23 on page 423 shows that for some well-behaved scientific programs the compiler can sustain multiple outstanding L2 misses (“miss under miss”) so as to effectively cut the L2 miss penalty. Once again, for this to work the memory system behind the cache must match the desires of the compiler in number of simultaneous memory accesses.

Combining the Instruction Cache with Instruction Fetch and Decode Mechanisms

With Moore’s Law continuing to offer more transistors and increasing demands for instruction level parallelism and clock rate, increasingly the instruction cache and first part of instruction execution are merging (see Chapter 3).

The leading example is the Netburst microarchitecture of the Pentium 4 and its successors. Not only does it use a trace cache (see page 434), which combines branch prediction with instruction fetch, it stores the internal RISC operations (see Chapter 3) in the trace cache. Hence, cache hits save 5 of 25 pipeline stages for decoding and translation. The downside of caching decoded instructions is impact on die size. It appears on the die that the 12000 RISC operations in the trace cache take equivalent of 96 KB of SRAM, which suggests that the RISC operations are about 64-bits long. 80x86 instructions would surely be two to three times more efficient.

Embedded computers also have bigger instructions in the cache, but for another reason. Given the importance of code size for such applications, several keep a compressed version of the instruction in main memory and then expand to the full size in the instruction cache (see page 130 in Chapter 2.)

Embedded Computer Caches and Real Time Performance

As mentioned before, embedded computers often are placed in real time environments where a set of tasks must be completed every time period. In such situations performance variability is of more concern than average case performance. Since caches were invented to improve average case performance at the cost of greater variability, they would seem to be a problem for real time computing.

In practice, instruction caches are widely used in embedded computers since most code has predictable behavior. Data caches then are the real issue.

To cope with that challenge, some embedded computers allow a portion of the cache to be "locked down." That is, a portion of the cache acts like a small scratchpad memory under program control. In a set associative data cache, one block of an entry would be locked down while the others could still buffer accesses to main memory. If it was direct mapped, then every address that maps onto that locked down block would result in a miss and later is passed to the CPU.

Embedded Computer Caches and Power

Although caches were invented to reduce memory access time, they also save power. It is much more power efficient to access on chip memory than it is to drive the pins of the chip, drive the memory bus, activate the external memory chips and then make the return trip.

To further improve power efficiency of caches on chip, some of the optimizations in sections 5.4 to 5.7 are reoriented for power. For example, the MIPS 4300 uses way prediction to only power half of the address checking hardware for its two-way set associative cache.

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 5.42 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

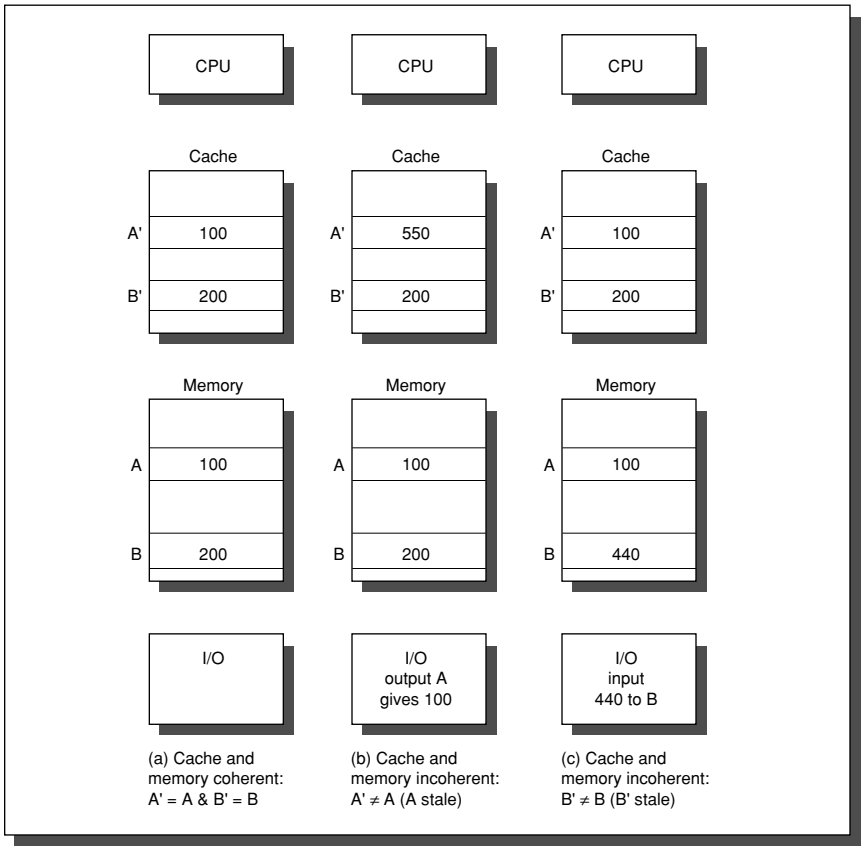

FIGURE 5.42 The cache-coherency problem. A' and B' refer to the cached copies of A and B in memory. (a) shows cache and main memory in a coherent state. In (b) we assume a write-back cache when the CPU writes 550 into A. Now A' has the value but the value in memory has the old, stale value of 100. If 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.

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 may also interfere with the cache by displacing some information with new data that is unlikely to be accessed 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 acting as an I/O buffer. If a write-through cache were used, then memory would have an up-to-date copy of the information, and there would be no stale-data issue for output. (This benefit is a reason processors used write through.) Alas, write-through usually found only today in first level data caches backed by a L2 cache which uses write back. Even embedded caches avoid write through for reasons of power efficiency

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, a duplicate set of tags may be 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 7.

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 6.

5.13 Putting It All Together: Alpha 21264 Memory Hierarchy

Thus far we have given glimpses of the Alpha 21264 memory hierarchy; this section unveils the full design and shows the performance of its components for the SPEC95 programs. Figure 5.43 gives the overall picture of this design. The 21264 is an out-of-order execution processor that fetches up to four instructions per clock cycle and executes up to six instructions per clock cycle. It uses either a 48-bit virtual address and a 44-bit physical address or 43-bit virtual address and 41-bit physical; thus far, all systems just use 41 bits. In either case, Alpha halves the physical address space, with the lower half for memory addresses and the upper half for I/O addresses. For the rest of this section, we assume use of the 43-bit virtual address and the 41-bit physical address.

Let's really start at the beginning, when the Alpha is turned on. Hardware on the chip loads the instruction cache serially from an external PROM. This initialization fills up to 64-KB worth of instructions (16K instructions) into the cache. The same serial interface (and PROM) also loads configuration information that specifies L2 cache speed/timing, system port speed/timing, and much other infor-

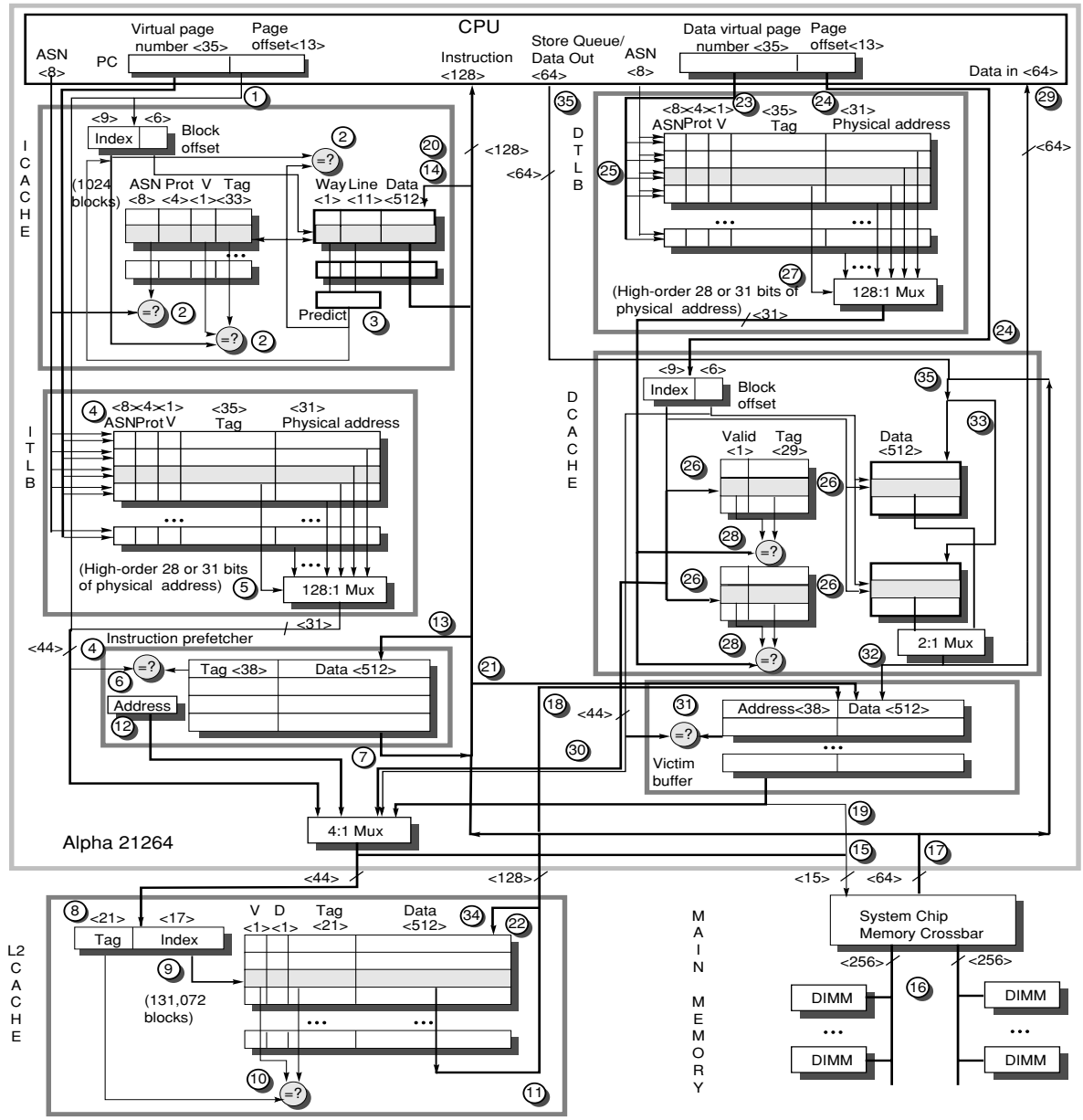

FIGURE 5.43 The overall picture of the Alpha 21264 memory hierarchy. Individual components can be seen in greater detail in Figures 5.7 (page 388) and 5.36 (page 454). The instruction cache is virtually indexed and tagged, but the data cache has virtual index but physical tags. Hence, every data address must be sent to the data TLB at the same time as it is sent to the data cache. Both the instruction and data TLB's have 128 entries. Each TLB entry can map a page of size 8KB, 64KB, 512KB, or 4MB. The 21264 supports a 48-bit or 43-bit virtual address and a 44-bit or 41-bit physical address.

mation necessary for the 21264 to communicate with the external logic. This code completes the remainder of the processor and system initialization.

The preloaded instructions execute in Privileged Architecture Library (PAL) mode. The software executed in PAL mode 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 the instruction addresses are not translated. Since PAL code avoids the TLB, instruction accesses are not checked for memory management violations.

One of the first steps is to update the instruction TLB with valid page table entries (PTEs) for this process. Kernel code updates the appropriate page table entry (in memory) for each page to be mapped. A miss in the TLB is handled by PAL code, since normal code that relies on the TLB cannot change 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 5.43 is labeled with the steps of this narrative. First, a 12-bit address is sent to the 64-KB instruction cache, along with a 35-bit page number. An 8-bit Address Space Number (ASN) is also sent, for the same purpose as using ASN's in the TLB (step 1).The instruction cache is virtually indexed and virtually tagged, so instruction TLB translations are only required on cache misses. As mentioned in section 5.5, the Instruction cache uses way prediction, so a 1-bit way predict bit is prepended to the 9-bit index. The effective index is then 10 bits, similar to a 64-KB direct mapped cache with 1024 blocks. Thus, the effective instruction cache index is 10 bits (see page 387), and the instruction cache tag is then 48 – 9 bits (actual index) – 6 bits (block offset) or 33 bits. As the 21264 expects 4 instructions (16 bytes) each instruction fetch, an additional 2 bits is used from the 6-bit block offset to select the appropriate 16 bytes. Hence, 10 + 2 or 12 bits to read 16 bytes of instructions.

To reduce latency, the instruction cache includes two mechanisms to begin early access of the next block. As mentioned in section 5.5, the way predicting cache relies on a 1-bit field for every 16 bytes to predict which of two sets will be used next, offering the hit time of direct mapped with miss rate of two-way associativity. It also includes 11 bits to predict the next group of 16 bytes to be read. This field is loaded with the address of the next sequential group on a cache miss, and updated to a nonsequential address by the dynamic branch predictor. These two techniques are called *way prediction* and *line prediction*.

Thus, the index field of the PC is compared to the predicted block address, the tag field of the PC is compared to the address from the tag portion of the cache, and the 8-bit process ASN to the tag ASN field (step 2). The valid bit is also checked. If any field has the wrong value, it is a miss. On a hit in the instruction cache, the proper fetch block is supplied, and the next way and line prediction is loaded to read the next block (step 3). There is also a protection field in the tag, to

ensure that instruction fetch does not violate protection barriers. The instruction stream access is now done.

An instruction cache miss causes a simultaneous check of the instruction TLB and the instruction prefetcher (step 4). The fully associative TLB simultaneously searches all 128 entries to find a match between the address and a valid PTE (step 5). In addition to translating the address, the TLB checks to see if the PTE demands that this access result in an exception, and if the address space number of the processor matches the address space number in the field. An exception might occur if either this access violates the protection on the page or if the page is not in main memory. If the desired instruction address is found in the instruction prefetcher (step 6), the instructions are (eventually) supplied directly by the prefetcher (step 7). Otherwise, if there is no TLB exception, an access to the second-level cache is started (step 8).

In the case of a complete miss, the second-level cache continues trying to fetch the block. The 21264 microprocessor is designed to work with direct-mapped second-level caches from 1MB to 16 MB. For this section we use the memory system of the 667 Mhz Compaq AlphaserverES40, a shared memory system with from 1 to 4 processors. It has a 444 Mhz, 8-MB direct-mapped, second-level cache. (The data rate is 444 Mhz; the L2 SRAM parts use the double data rate technique of DRAMs, so they are clocked at only half that rate.)The L2 index is

$$2^{\text{index}} = \frac{\text{Cache size}}{\cancel{\text{Block size} \times \text{Set associativity}}} = \frac{8192K}{\cancel{64 \times 1}} = 128K = 2^{17}$$

so the 35-bit block address (41-bit physical address – 6-bit block offset) is divided into a 18-bit tag and a 17-bit index (step 9).The cache controller reads the tag from that index and if it matches and is valid (step 10), it returns the critical 16 bytes (step 11), with the remaining 48 bytes of the cache block supplied 16 bytes per 2.25 ns. The 21264 L2 interface does not require that the L2 cache clock be an integer multiple of the processor clock, so it can be loaded faster than 3.00 ns that you might expect from a 667 MHz processor. At the same time, a request is made for the next sequential 64-byte block (step 12), which is loaded into the instruction prefetcher in the next 6 clock cycles (step 13). Each miss can cause a prefetch of up to 4 cache blocks. An instruction cache miss costs approximately 15 CPU cycles (22 ns), depending on clock alignments.

By the way, the instruction prefetcher does not rely on the TLB for address translation. It simply increments the physical address of the miss by 64 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. To save time, the prefetched instructions are passed around the CPU and then written to the instruction cache while the instructions execute in the CPU (step 14).

If the instruction is not found in the secondary cache, the physical address command is sent to the ES40 system chip set via four consecutive transfer cycles

on a narrow, 15-bit outbound address bus (step 15). The address and command use the address bus for 8 CPU cycles. The ES40 connects the microprocessor to memory via a crossbar to one of two 256-bit memory busses to service the request (step 16). Each bus contains a variable number of DIMMs (dual inline memory modules). The size and number of DIMMs can vary to give a total of 32 GB of memory in the 667 Mhz ES40. Since the 21264 provides single error correction/double error detection checking on data cache (see section 5.15), L2 cache, busses, and main memory, the data busses actually include an additional 32-bits for ECC bits.

Although the crossbar has two 256-bit buses, the path to the microprocessor is much narrower. 64 data bits. Thus, the 21264 has two off-chip paths: 128 data bits for the L2 cache and 64 data bits for memory crossbar. Separate paths allows a point-to-point connection and hence a high clock rate interface for both the L2 cache and the crossbar.

The total latency of the instruction miss that is serviced by main memory is approximately 130 CPU cycles for the critical instructions. The system logic fills the remainder of the 64-byte cache block at a rate of 8 bytes per 2 CPU cycles (step 17).

Since the second-level cache is a write-back cache, any miss can lead to an old block being written back to memory. The 21264 places this "victim" block into a victim file (step 18), as it does with a victim dirty block in the data cache, to get out of the way of new data when the new cache reference determined first read the L2 cache; that is, the original instruction fetch read that missed (step 8). The 21264 sends out the address of the victim out the system address bus following the address of the new request (step 19). The system chip set later extracts the victim data and writes it to the memory DIMMs.

The new data are loaded into the instruction cache as soon as they arrive (step 20). It also goes into a (L1) victim buffer (step 21), and is later written to the L2 cache (step 22). The victim buffer is of size 8, so many victims can be queued before being written back either to the L2 or to memory. The 21264 can also manage up to 8 simultaneous cache block misses, allowing it to hit under 8 misses as described in section 5.4.

If this initial instruction is a load, the data addresses is also sent to the data cache. It is 64 KB, 2-way set-associative, and write-back with a 64-byte block size. Unlike the instruction cache, the data cache is virtually indexed and *physically* tagged. Thus, the page frame of the instruction's data address is sent to the data TLB (step 23) at the same time as the (9+3)-bit index from the virtual address is sent to the data cache (step 24). The data TLB is a fully associative cache containing 128 PTEs (step 25), 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, just as before. Since millions of instructions could execute during a page fault, the operating system will swap in another process if one is waiting to run.

The index field of the address is sent of both sets of the data cache (step 26). Assuming that we have a valid PTE in the data TLB (step 27), the two tags and valid bits are compared to the physical page frame (step 28), with a match sending the desired 8 bytes to the CPU (step 29). A miss goes to the second-level cache, which proceeds similar to an instruction miss (Step 30), except that it must check the victim buffer first to be sure the block is not there (Step 31).

As mentioned in section 5.7, the data cache can be virtually addressed and physically tagged. On a miss, the cache controller must check for a synonym (two different virtual addresses that reference the same physical address). Hence, the data cache tags are examined in parallel with the L2 cache tags during an L2 lookup. As the minimum page size is 8 KB or 13 bits and the cache index plus block offset is 15 bits, the cache must check 2^2 or 4 blocks per set for synonyms. If it finds a synonym, the offending block is invalidated. This guarantees that a cache block can reside in one of the 8 possible data cache locations at any given time.

A write back victim can be produced on a data cache miss. The victim data is extracted from the data cache simultaneously with the fill of the data cache with the L2 data, and sent to the victim buffer (Step 32). I

In the case of an L2 miss, the fill data from the system is written directly into the (L1) data cache (step 33). The L2 is written only with L1 victims (step 34). They appear either because they were modified by the CPU, or because they had been loaded from memory directly into the data cache but not yet written into the L2 cache.

Suppose the instruction is a store instead of a load. When the store issues it does a data cache lookup just like a load. A store miss causes the block to be filled into the data cache very much as with a load miss. The store does not update the cache until later, after it is known to be non-speculative. During this time the store resides in a store queue, part of the out-of-order control mechanism of the CPU. Stores write from the store queue into the data cache on idle cache cycles (step 35). The data cache is ECC protected, so a read-modify-write operation is required to update the data cache on stores.

Performance of the 21264 Memory Hierarchy

How well does the 21264 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. Alas, in an out-of-order execution processors like the 21264 it is very hard to isolate the time waiting for memory, since a memory stall for one instruction may be completely hidden by successful completion of a later instruction.

How well does out-or-order perform compared in in-order? Figure 5.44 shows relative performance for SPECint2000 benchmarks for the out-of-order 21264 and its predecessor, the in-order Alpha 21164. The clock rates are similar in the

figure, but other differences in addition include the on-chip caches (two 64 KB L1 caches vs. two 8 KB L1 caches plus one 96 KB L2 cache). The miss rate for the 21164 on-chip L2 cache is also plotted in the figure along with the miss rate

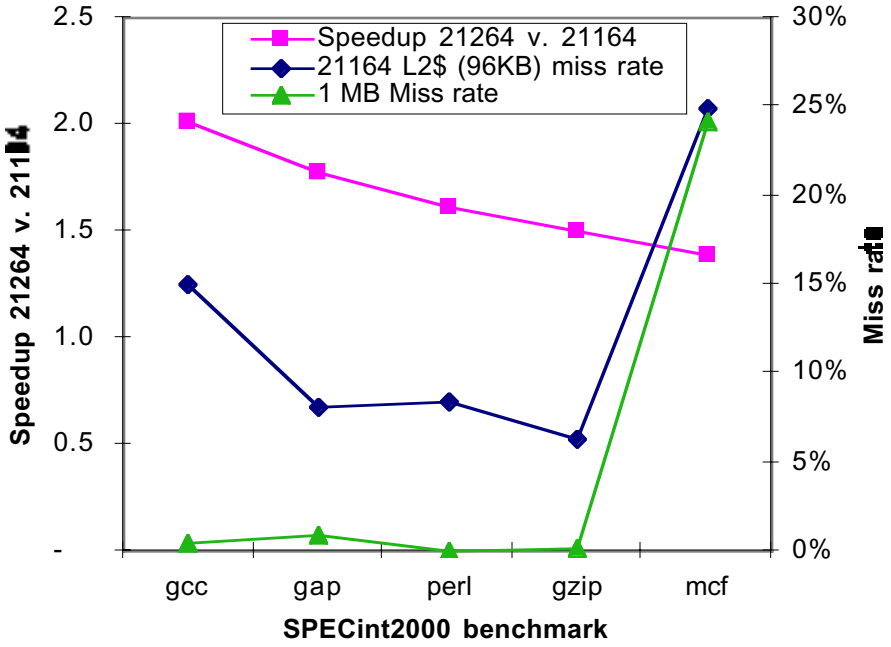

FIGURE 5.44 Alpha 21264/21164 Performance speedup vs. miss rates for SPECint2000. The left axis shows the speedup of the out-of-order 21264 is greatest with the highest miss rate of the 21164 L2 cache (right axis) as long as the access is a hit in the 21264 L2 cache. If it misses the L2 cache of the 21264, out-of-order execution is not as helpful. The 21264 is running at 500 MHz and the earlier 21164 is running at 533MHz.

of a 1MB cache. Figure 5.44 shows that speedup generally tracks its miss rate; the higher the 21164 miss rate, the higher the speedup of the 21264 over the 21164. The only exception is MCF, which is also the only program with a high miss rate for the a 1MB cache.

This result is likely explained by the 21264's ability to continue to execute during caches misses which stall the 21164 but hit in the L2 cache of the 21264. If the miss also misses in the L2 cache, then the 21264 must also stall, hence the lower speedup for MCF.

Figure 5.45 shows the CPI and various misses per 1000 instructions for a benchmark similar to TPC-C on a commercial database and the SPEC95 programs. Clearly, the SPEC95 programs do not tax the 21264 memory hierarchy, with instruction cache misses per instruction of 0.001% to 0.343% and second-level cache misses per instruction of 0.001% to 1%. The commercial benchmark

Program	CPI	Cache misses per 1000 instructions		TLB misses per 1000 instructions
		I cache	L2 cache	D TLB
TPC-C like	2.23	11.15	7.30	1.21
go	0.58	0.53	0.00	0.00
m88ksim	0.38	0.16	0.04	0.01
gcc	0.63	3.43	0.25	0.30
compress	0.70	0.00	0.40	0.00
li	0.49	0.07	0.00	0.01
ijpeg	0.49	0.03	0.02	0.01
perl	0.56	1.66	0.09	0.26
vortex	0.58	1.19	0.63	1.98
Avg. SPECint95	0.55	0.88	0.18	0.03
tomcatv	0.52	0.01	5.16	0.12
swim	0.40	0.00	5.99	0.10
su2cor	0.59	0.03	1.64	0.11
hydro2d	0.64	0.01	0.46	0.19
mgrid	0.44	0.02	0.05	0.10
applu	0.94	0.01	10.20	0.18
turb3d	0.44	0.01	1.60	0.10
apsi	0.67	0.05	0.01	0.04
fpppp	0.52	0.13	0.00	0.00
wave5	0.74	0.07	1.72	0.89
Avg. SPECfp95	0.59	0.03	2.68	0.09

FIGURE 5.45 CPI and misses per 1000 instructions for a running a TPC-C like database workload and the SPEC95 benchmarks (see Chapter 1) on the Alpha 21264 in the Compaq ES40. In addition to the worse miss rates shown here, the TPC-C like benchmark also has a branch misprediction rate of about 19 per 1000 instructions retired. This rate is 1.7 times worse than the average SPECint95 program and 25 times worse than the average SPECfp95. Since the 21264 uses an out-of-order instruction execution, the statistics are calculated as the number of misses per 1000 instructions successfully committed. Cvetnovic and Kessler [2000] collected this data, but unfortunately did not include miss rates of the L1 data cache or data TLB. Note that their hardware performance monitor could not isolate the benefits of successful hardware prefetching to the Instruction Cache. Hence, compulsory misses are likely very low.

does exercise the memory hierarchy more, with misses per instruction of 1.1% and 0.7%, respectively.

How do the CPIs compare to the peak rate of 0.25, or 4 instructions per clock cycle? For SPEC95 the 21264 completes almost 2 instructions per clock cycle, with an average CPI of 0.55 to 0.59. For the database benchmark, the combination of higher miss rates for caches and TLBs and a higher branch misprediction

rate (not shown) yield a CPI of 2.23, or less than one instruction every two clock cycles. This factor of four slowdown in CPI suggest that microprocessors designed to be used in servers may see much heavier demands on the memory systems than do microprocessors for desktops.

5.14 Another View: The Emotion Engine of the Sony Playstation 2

Desktop computers and severs rely on the memory hierarchy to reduce average access time to relatively static data, but there are embedded applications where data is often a continuos stream. In such applications there is still spatial locality, but temporal locality is much more limited.

To give another look at memory performance beyond the desktop, this section examines the microprocessor at the heart of Sony Playstation 2. As we shall see, the steady stream of graphics and audio demanded by electronic games leads to a different approach to memory design. The style is high bandwidth via many dedicated independent memories.

Figure 5.46 shows the 3Cs for the MP3 decode kernel. Compared to SPEC2000 results in Figure 5.15 on page 410, much smaller caches capture the misses for multimedia applications. Hence, we expect small caches.

Figure 5.47 shows a block diagram of the Sony Playstation 2 (PS2). Not surprisingly for a game machine, there are interfaces for video, sound, and a DVD player. Surprisingly, there are two standard computer I/O busses, USB and IEEE 1394, a PCMCIA slot as found in portable PCs, and a modem. These additions suggest Sony has suggest greater plans for the PS2 beyond traditional games. Although it appears the that I/O processor (IOP) simply handles the I/O devices and the game console, it includes a 34 MHz MIPS processor which also acts as the emulation computer to run games for earlier Sony Playstations. It also connects to a standard PC audio card to provide the sound for the games.

Thus, one challenge for the memory system of this embedded application is to act as source or destination for the extensive number of I/O devices. The PS2 designers met this challenge with two PC800 (400 MHz) DRDRAM chips using two channels, offering 32 MB of storage and a peak memory bandwidth of 3.2 MB/second (see section 5.8).

What's left in the figure is basically two big chips: the Graphics Synthesizer and the Emotion Engine.

The Graphics Synthesizer takes rendering commands from the Emotion Engine in what are commonly called *display lists*. These are lists of 32-bit commands that tell the renderer what shape to use and where to place them, plus what colors and textures to fill them.

This chip also has the highest bandwidth portion of the memory system. By using embedded DRAM on the Graphics Synthesizer, the chip contains the full video buffer *and* has a 2048-bit wide interface so that pixel filling is not a bottleneck. This embedded DRAM greatly reduces the bandwidth demands on the

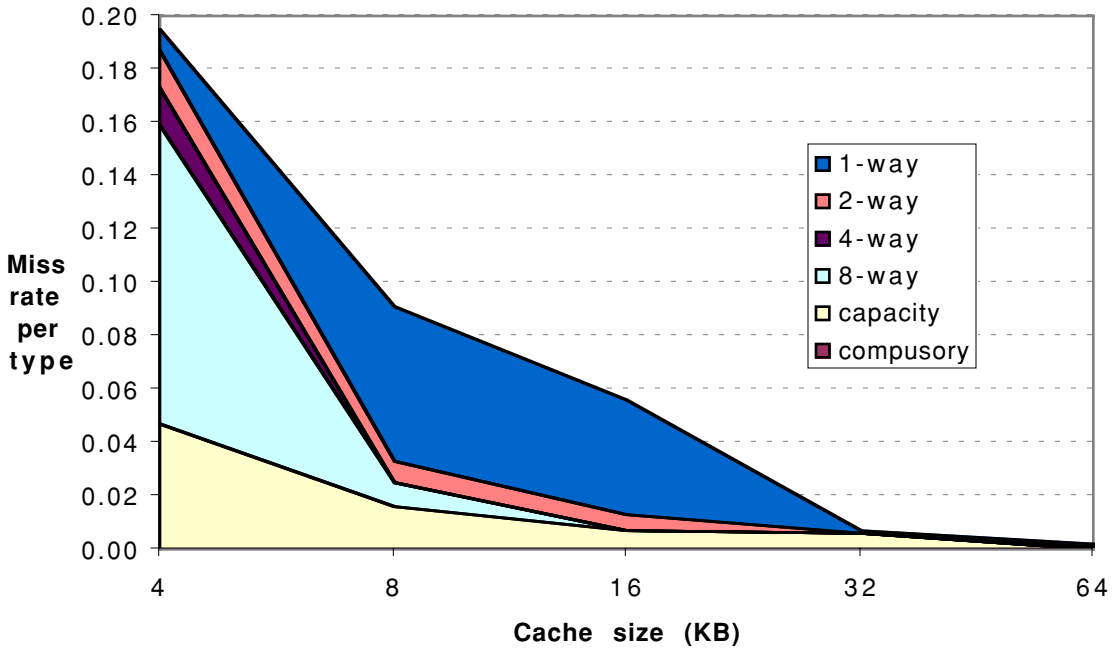

FIGURE 5.46 Three Cs for MPEG3 decode. A 2-way set associative 16-KB data cache has a total miss rate of 0.013, compared to in 0.041 in Figure 5.14 on page 409. The compulsory misses are too small to see on the graph. From Hughes et al [2001].

DRDRAM. It illustrates a common technique found in embedded applications: separate memories dedicated to individual functions to inexpensively achieve greater memory bandwidth for the entire system.

The remaining large chip is the Emotion Engine, and its job is to accept inputs from the IOP and create the display lists of a video game to enable 3D video transformations in real time. A major insight shaped the design of the Emotion Engine: generally in a racing car game there are foreground objects that are constantly changing and background objects that change less in reaction to the events, although the background can be most of the screen. This observation led to a split of responsibilities.

The CPU works with VPU0 as a tightly-coupled coprocessor, in that every VPU0 instruction is a standard MIPS coprocessor instruction, and the addresses are generated by the MIPS CPU. VPU0 is called a vector processor, but it is similar to a 128-bit, SIMD extensions for multimedia found in several desktop processors (see Chapter 2).

VPU1, in contrast, fetches its own instructions and data and acts in parallel with CPU-VPU0, acting more like a traditional vector unit. With this split, the

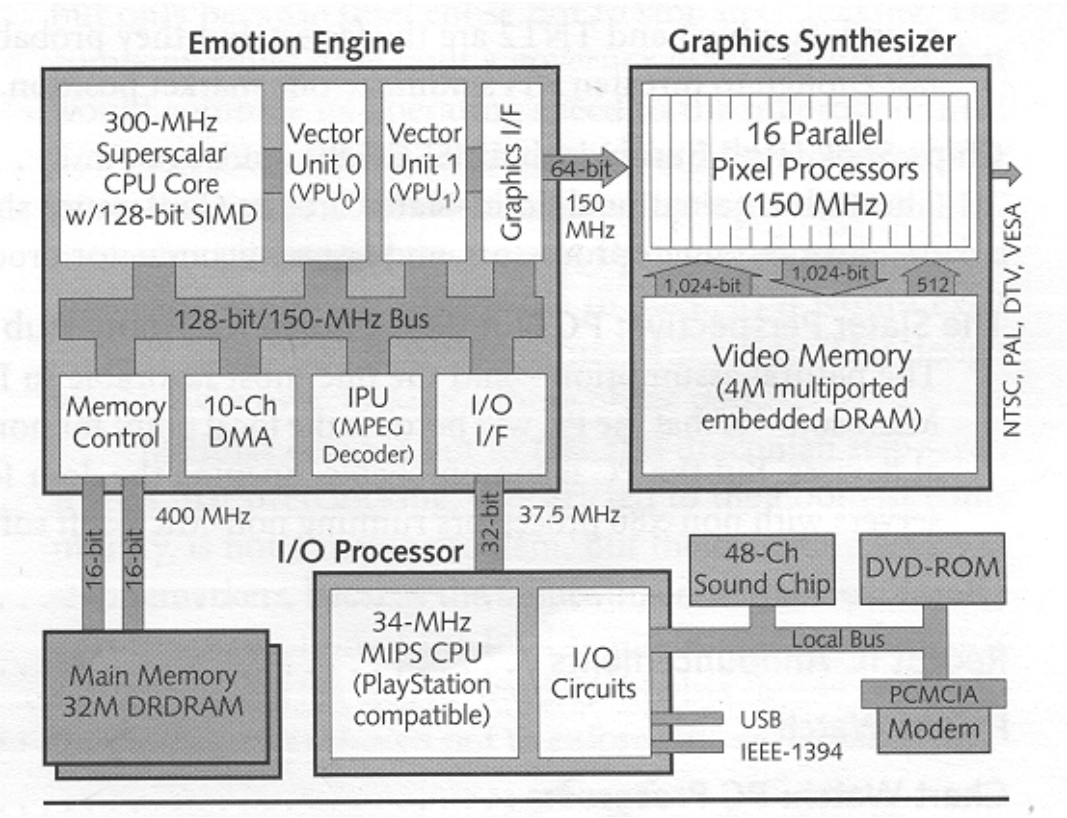

FIGURE 5.47 Block diagram of the Sony Playstation 2. The 10 DMA channels orchestrate the transfers between all the small memories on the chip, which when completed all head towards the Graphics Interface so as to be rendered by the Graphics Synthesizer. The Graphics Synthesized uses DRAM on-chip to provide an entire frame buffer plus graphics processors to perform the rendering desired based on the display commands given from the Emotion Engine. The Embedded DRAM allows 1024-bit transfers between the pixel processors and the display buffer. The Supescalar CPU is a 64-bit MIPS III with two-instruction issue, and comes with a 2-way set associative, 16-KB instruction cache, a 2-way set associative 8-KB data cache, and 16 KB of scatchpad memory. It has been extended with 128-bit SIMD instructions for multimedia applications (see Chapter 2). Vector Unit 0 is a primarily a DSP-like coprocessor for the CPU (see Chapter 2), which can an operator on 128-bit registers in SIMD manner between 8 bits and 32 bits per word. It has 4 KB of instruction memory and 4 KB of data memory. Vector Unit 1 has similar functions to VPU0, but it normally operates independently of the CPU, and contains 16 KB of instruction memory and 16 KB of data memory. All three units can communicate over the 128-bit system bus, but there are is also a 128-bit dedicated path between the CPU and VPU0 and a 128-bit dedicated path between VPU1 and the Graphics Interface. Although VPU0 and VPU1 have identical microarchitectures, the differences in memory size and units to which they have direct connections affect the roles that they take in a game. At 0.25- micron line widths, the Emotion Engine chip uses 13.5M transistors and 225 mm^2 and the Graphics Synthesizer is 279 mm^2. To put this in perspective, the Alpha 21264 microprocessor in 0.25-micron technology is about 160 mm^2 and uses 15M transistors. (This figure is based on a Figure 1 in “Sony’s Emotionally Charged Chip,” Microprocessor Report, 13:5.)

more flexible CPU-VPU0 handles the foreground action and the VPU1 handles the background. Both deposit their resulting display lists into the Graphics Interface to send the lists to the Graphics Synthesizer.

Thus, the programmer of the Emotion Engine thus has three processors sets to chose from to implement his program: the traditional 64-bit MIPS architecture including a floating point unit, the MIPS architecture extended with multimedia instructions (VPU0), and an independent vector processor (VPU1). To accelerate MPEG decoding, the is another coprocessor (Image Processing Unit) that can act independent of the other two.

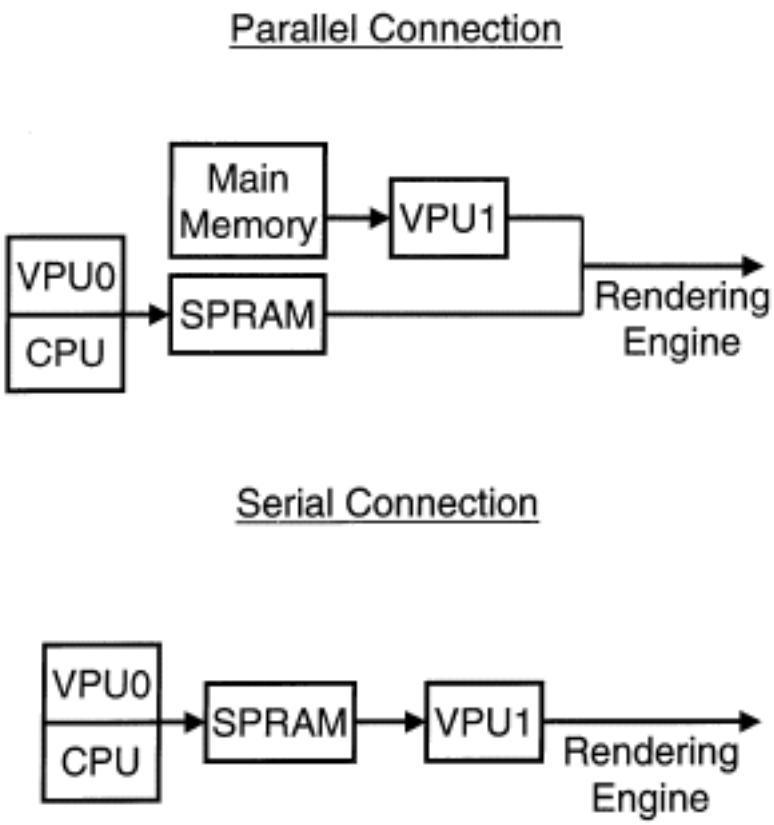

FIGURE 5.48 Two modes of using Emotion Engine organization. The first mode divides the work between the two units, and then allows the Graphics Interface to properly merge the display lists. The second more uses CPU/VPU0 as a filter of what to send to VPU1, which then does all the display lists. Its up to the programmer to chose between serial and parallel data flow. SPRAM is the scratchpad memory. **<<Redraw with Serial on top, Parallel on Bottom; can be much smaller>>**

With this split of function, the question then is how to connect the units together, how to make the data flow between units, and how to provide the memory

bandwidth needed by all these units. As mentioned above, the Emotion Engine designers chose many dedicated memories. The CPU has a 16-KB scatchpad memory (SPRAM) in addition to a 16-KB instruction cache and an 8-KB data cache. VPU0 has a 4-KB instruction memory and a 4-KB data memory, and VPU1 has a 16-KB instruction memory and a 16-KB data memory. Note that these are four *memories*, not caches of a larger memory elsewhere. In each memory the latency is just one clock cycle. VPU1 has more memory than VPU0 because it creates the bulk of the display lists and because it largely acts independently.

The programmer organizes all memories as two double buffers, one pair for the incoming DMA data and one pair for the outgoing DMA data. The programmer then uses the various processors to transform the data from the input buffer to the output buffer. To keep the data flowing among the units, the programmer next sets up the 10 DMA channels, taking care to meet the real time deadline for realistic animation of 15 frame per second.

Figure 5.48 shows that this organization supports two main operating modes: serial where CPU/VPU0 act as a preprocessor on what to give VPU1 for it to create for the Graphics Interface using the scratchpad memory as the buffer, and parallel where both the CPU/VPU0 and VPU1 create display lists. The display lists and the Graphics Synthesizer have multiple context identifiers to distinguish the parallel display lists to produce a coherent final image.

All units in the Emotion Engine are linked by a common 150 Mhz, 128-bit wide bus. To offer greater bandwidth, there are also two dedicated buses: a 128-bit path between the CPU and VPU0, and a 128-bit path between VPU1 and the Graphics Interface. The programmer also chooses which bus to use when setting up the DMA channels.

Taking the big picture, if a server-oriented designer had been given the problem we might see a single common bus with many local caches and cache coherent mechanism to keep data consistent. In contrast, the Playstation 2 followed the tradition of embedded designers and has at least nine distinct memory modules. To keep the data flowing in real time from memory to the display the PS2 uses dedicated memories, dedicated buses, and DMA channels. Coherency is the responsibility of the programmer, and given the continuous flow from main memory to the graphics interface and the real time requirements, programmer controlled coherency works well for this application.

5.15 Another View: The Sun Fire 6800 Server

The Sun Fire 6800 is a mid range multiprocessor server with particular attention paid to the memory system. The emphasis of this server is cost-performance for both commercial computing, running data base applications such data warehousing and data mining, as well as high performance computing. This server also includes special features to improve availability and maintainability.

Given these goals, what should be the size of the caches? Looking at the SPEC2000 results in Figure 5.17 on page 413 suggests miss rates of 0.5% for a 1-MB data cache, with infinitesimal instruction cache miss rates at those sizes. It would seem that a 1-MB off chip cache should be sufficient.

Commercial workloads, however, get very different results. Figure 5.49 shows the impact on the off chip cache for an Alpha 21164 server running commercial workloads. Unlike the results for SPEC2000, commercial workloads running database applications have significant misses just for instructions with a 1-megabyte cache. The reason is that code size of commercial databases is measured in millions of lines of code, unlike any SPEC2000 benchmark. Second, note that capacity and conflict misses remain significant until cache size becomes 4 to 8 megabyte. Note that even compulsory misses lead to a measurable causes higher CPI; this is because servers often run many processes, which results in many context switches and thus more compulsory misses. Finally, there is a new category of misses in a multiprocessor, and these are due to having to keep all the caches of a multiprocessor coherent, a problem mentioned in section 5.12. These are sometimes called *coherency misses*, adding a fourth C to our three C model from section 5.5. Chapter 6 explains a good deal more about coherence or sharing traffic in multiprocessors. The data suggests that commercial workloads need considerably bigger off-chip caches than do SPEC2000.

Figure 5.50 shows the essential characteristics of the Sun Fire 6800 that the designers selected. Note the 8 MB L2 cache, which is in response to the commercial needs.

The microprocessor that drives this server is the UltraSPARC III. One striking feature of the chip is the number of pins: 1368 in a Ball Grid Array. Figure 5.51 shows how one chip could use so many pins. The L2 caches bus operates at 200 MHz, local memory at 75 MHz, and the rest operate at 150 MHz. The combination of UltraSPARC III and the data switch yields a peak bandwidth to off chip memory of 11 Gbytes/second.

Note that the several wide buses include error correction bits in Figure 5.51. Error correction codes enable buses and memories to both detect and correct errors. The idea is to calculate and storage parity over different subsets of the bits in the protected word. When parity does not match it indicates an error. By looking at which of the overlapping subsets have a parity error and which don't, its possible to determine the bit that failed. The Sun Fire ECC was also designed to detect any pair of bit errors, and also to detect if a whole DRAM chip failed turning all the bits of an 8-bit wide chip to zero. Such codes are generally classified as *Single Error Correcting/Double Error Detecting* (*SEC/DED*). The UltraSPARC sends these ECC bits between the memory chips and the microprocessor, so that errors that occur on the high-speed buses are also detected and corrected.

In addition to several wide busses for memory bandwidth, the designers of UltraSPARC were concerned about latency. Hence, the chip includes a DRAM controller on the chip, which they claim saved 80 ns of latency. The result is 220 ns to the local memory and 280 ns to the non-local memory. (This server supports non-uniform memory access shared memory, described in Chapter 6.) Since

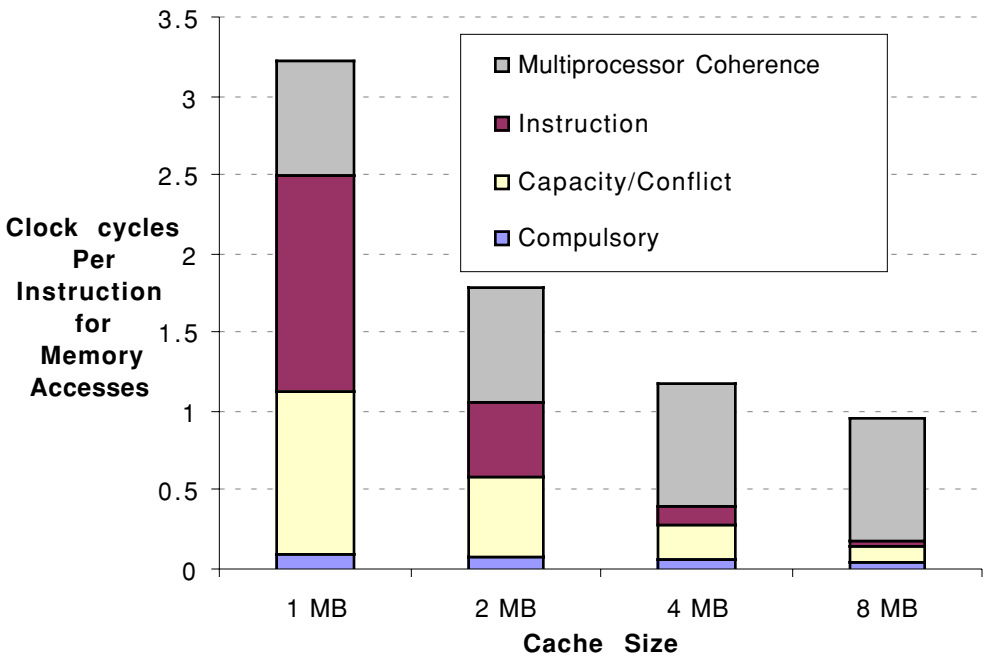

FIGURE 5.49 Clock cycles per instruction for memory accesses versus off-chip cache size for a four-processor server. Note how much higher the performance impact is for large caches than for the SPEC2000 programs in Figure 5.15 on page 410. The workload includes the Oracle commercial database engine for the online transaction processing (OLTP) and decision support systems and the AltaVista search engine for the Web index search. This data was collected by Barroso et al [1998] using the Alpha 21164 microprocessor.

memory is connected directly to the processor to lower latency, its size is a function of the number of processors. The limit in 2001 is 8 GB per processor.

For similar latency reasons, UltraSPARC also includes the tags for the L2 cache on chip. The designers claim this saved 10 clock cycles off a miss. At a total of almost 90 KB of tag, it is comparable in size to the data cache.

The on-chip caches are both 4-way set associative, with the instruction cache being 32 KB and the data cache being 64 KB. The block size for these caches is 32 bytes. To reduce latency to the data cache, it combines an address adder with the word line decoder. This combination largely eliminates the adder's latency. Compared to UltraSPARC II at the same cache size and clock rate, sum-addressed memory reduced latency from three to two clock cycles.

The L1 data cache uses write through (no write allocate) and the L2 cache uses write back (write allocate). Both L1 caches provide parity to detect errors; since the data cache is write through, there is always a redundant copy of the data elsewhere, so parity errors only require prefetching the good data. The memory system behind the cache supports to up 15 outstanding memory accesses.

Processors	2 to 24
Processors	2 to 24 UltraSPARC III processors
Processor Clock rate	900 MHz
Pipeline	14 stages
Superscalar	4-issue, 4-way sustained
L1 I cache	32 KB, 4-way set associative (S.A.), pseudorandom replacement
L1 I cache latency	2 clocks
L1 D cache	64 KB, 4-way SA; write through, no write allocate, pseudorandom replacement
L1 D cache latency	2 clocks
L1 I/D miss penalty	20 ns (15 to 18 clock cycles, depending on clock rate)
L2 cache	8 MB, direct mapped; write back, write allocate, multilevel inclusion
L2 miss penalty	220 to 280 ns (198 to 252 clock cycles, depending whether memory is local)
Write Cache	2 KB, 4-way SA, LRU, 64 byte block, no write allocate
Prefetch Cache	2 KB, 4-way SA, LRU, 64 byte block
Block size	32 bytes
Processor Address space	64 bit
Maximum Memory	8 GB/processor, or up to 192 GB total
System Bus, peak speed	Sun Fire Interconnect, 9.6 GB/sec
I/O cards	up to 8 66-MHz, 64-bit PCI, 24 33-MHz 64-bit PCI
Domains	1 to 4
Processor Power	70 W at 750 MHz
Processor Package	1368 pin flip-chip ceramic Ball Grid Array
Processor Technology	29 M transistors (75% SRAM cache), die size is 217 mm^2; 0.15 micron, 7-layer CMOS

FIGURE 5.50 Technical summary of Sun Fire 6800 server and UltraSPARC III microprocessor.

Between the two levels of cache is a 2-KB write cache. The write cache act as a write buffer and merges writes to consecutive locations. It keeps a bit per byte to indicate if it is valid and does not read the block from the L2 cache when the block is allocated. Often the entire block is written, there by avoiding a read access to the L2 cache. The designers claim more than 90% of the time UltraSPARC III can merge a store into an existing dirty block of the write cache. The write cache is also a convenient place to calculate ECC.

UltraSPARC III handles address translation with multiple levels of on-chip TLBs, with the smaller ones being fastest. A cache access starts with a virtual address selecting four blocks and four microtags which checks 8 bits of the virtual address to select the set. In parallel with the cache access the 64-bit virtual address is translated to the 43-bit physical address using two TLBs: a 16 entry fully

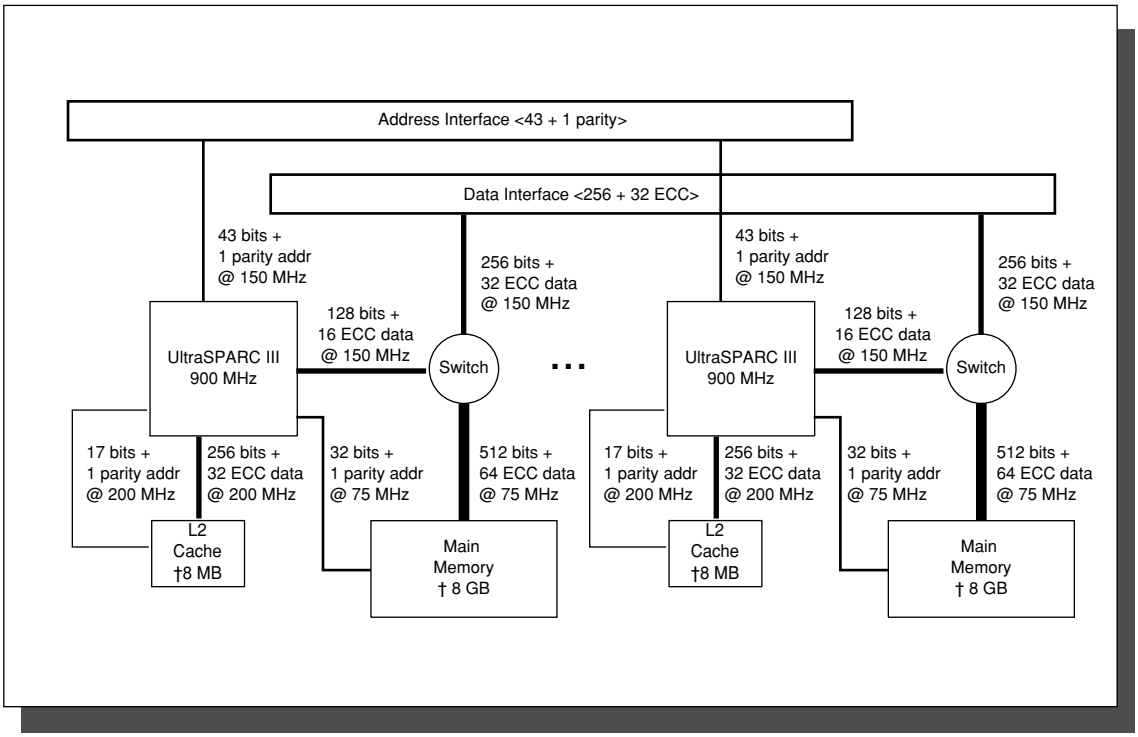

FIGURE 5.51 Sun Fire 6800 server block diagram. Note the large number of wide memory paths per processors. Up to 24 processors can be connected. Up to 12 share a single Sun Fire Data Interconnect. With more than 12, a second data interconnect is added. The number of separate paths to different memories are 256 data pins + 32 bits of error correction code (ECC) and 17 address bits + 1 bit parity for the off-chip L2 cache (200 MHz); 43 address pins + 1 bit of parity for addresses to external main memory; 32 address pins + 1 bit of parity for addresses to local main memory; 128 data pins + 16 bits of ECC to a data switch (150 MHz); 256 data pins + 32 bits of ECC between the data switch and the data interconnect (150 MHz); and 512 data pins + 64 bits of ECC between the data switch and local memory (75 MHz).

associative cache and a 128 entry 4-way associative cache. The physical address is then compared to the cache full tag, and only if they match is a cache hit allowed to proceed.

To get even more memory performance, UltraSPARC III also has a data prefetch cache, essentially the same as the streaming buffers described in section 5.6. It supports up to eight prefetch requests initiated either by hardware or by software. The prefetch cache remembers the address used to prefetch the data. If a load hits in prefetch cache, it automatically prefetches next load address. It calculates the stride using the current address and the previous address. Each prefetch entry is 64 bytes, and it is filled in 14 clock cycles from the L2 cache. Loads from the prefetch cache complete at the rate of 2 every 3 clock cycles versus 2 every 4 clock cycles from the data cache. This multiported memory can support two 8-byte reads and one 16-byte write every clock cycle.

In addition to prefetching data, UltraSPARC III has a small instruction prefetch buffer of 32 bytes that tries to stay one block ahead of the instruction decoder. On an instruction cache miss, two blocks are requested: one for the instruction cache and one for the instruction prefetch buffer. The buffer is then used to fill the cache if a sequential access also misses. In addition to parity and ECC to help with dependability, Sun Fire 6800 server offers up to four dynamic system domains. This option allows the computer to be divided into quarters or halves, with each piece running its own version of the operating system independently. Thus, a hardware or software failure in one piece does not affect applications running on the rest of the computer. The dynamic portion of the name means the reconfiguration can occur without rebooting the system.

To help diagnose problems, every UltraSPARC III has an 8-bit ``back door'' bus that runs independently from the main buses. If the system bus has an error, processors can still boot and run diagnostic programs over this back door bus to diagnose the problem.

Among the other availability features of the 6800 is a redundant path between the processors. Each system has two networks to connect the 24 processors together so that if one fails the system still works. Similarly, each Sun Fire interconnect has two crossbar chips to link it to the processor board, so that one crossbar chip can fail and yet the board can still communicate. There are also dual redundant system controllers that monitors server operation, and so they are able to notify administrators when problems are detected. Administrators can use the controllers to remotely initiate diagnostics and corrective actions.

In summary, the Sun Fire 6800 server and its processor pay much greater attention to dependability, memory latency and bandwidth, and system scalability than do desktop computers and processors.

5.16 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!

Fallacy: Predicting cache performance of one program from another.

Figure 5.14 on page 409 shows the instruction miss rates and data miss rates for three programs from the SPEC2000 benchmark suite as cache size varies. Depending on the program, the data misses per thousand instructions for a 4096-KB cache is 9, 2, or 90, and the instruction misses per thousand instructions for a 4-KB cache is 55, 19, or 0.0004. Figure 5.45 on page 478 shows that commercial programs such as databases will have significant miss rates even in a 8-MB second-level cache, which is generally not the case for the SPEC programs. Similarly, MPEG3 decode in Figure 5.46 on page 480 fits entirely in a 64 KB data cache,

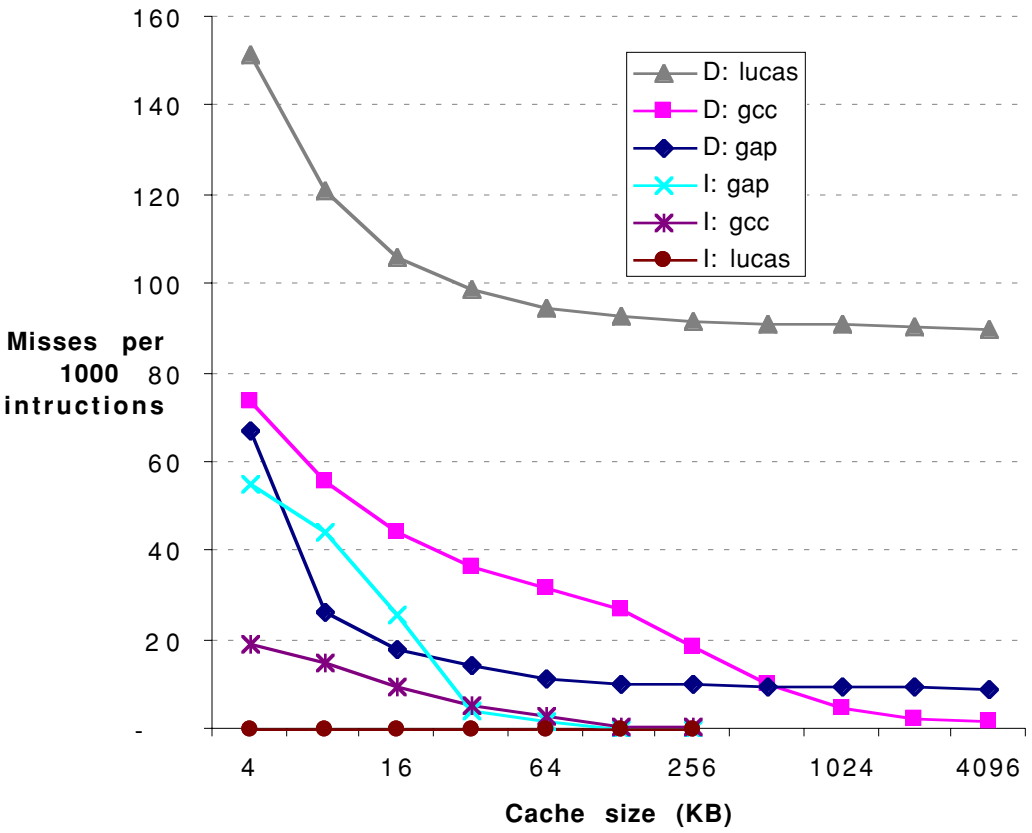

FIGURE 5.52 Instruction and data misses per thousand instructions for as cache size varies from 4 KB to 4096 KB. Instruction misses for gcc are 30,000 to 40,000 times larger than lucas, and conversely, data misses for lucas are 2 to 60 times larger than gcc. The programs gap, gcc, and lucas are from the SPEC2000 benchmark suite. These data are from the same experiment as in Figure 5.10.

while SPEC doesn't get such low miss rates until 1024 KB. Clearly, generalizing cache performance from one programs to another is unwise.

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

There are really three pitfalls here. One is trying to predict performance of a large cache using a small trace. Another is that a program's locality behavior is not constant over the run of the entire program. The third is that a program's locality behavior may vary depending on the input.

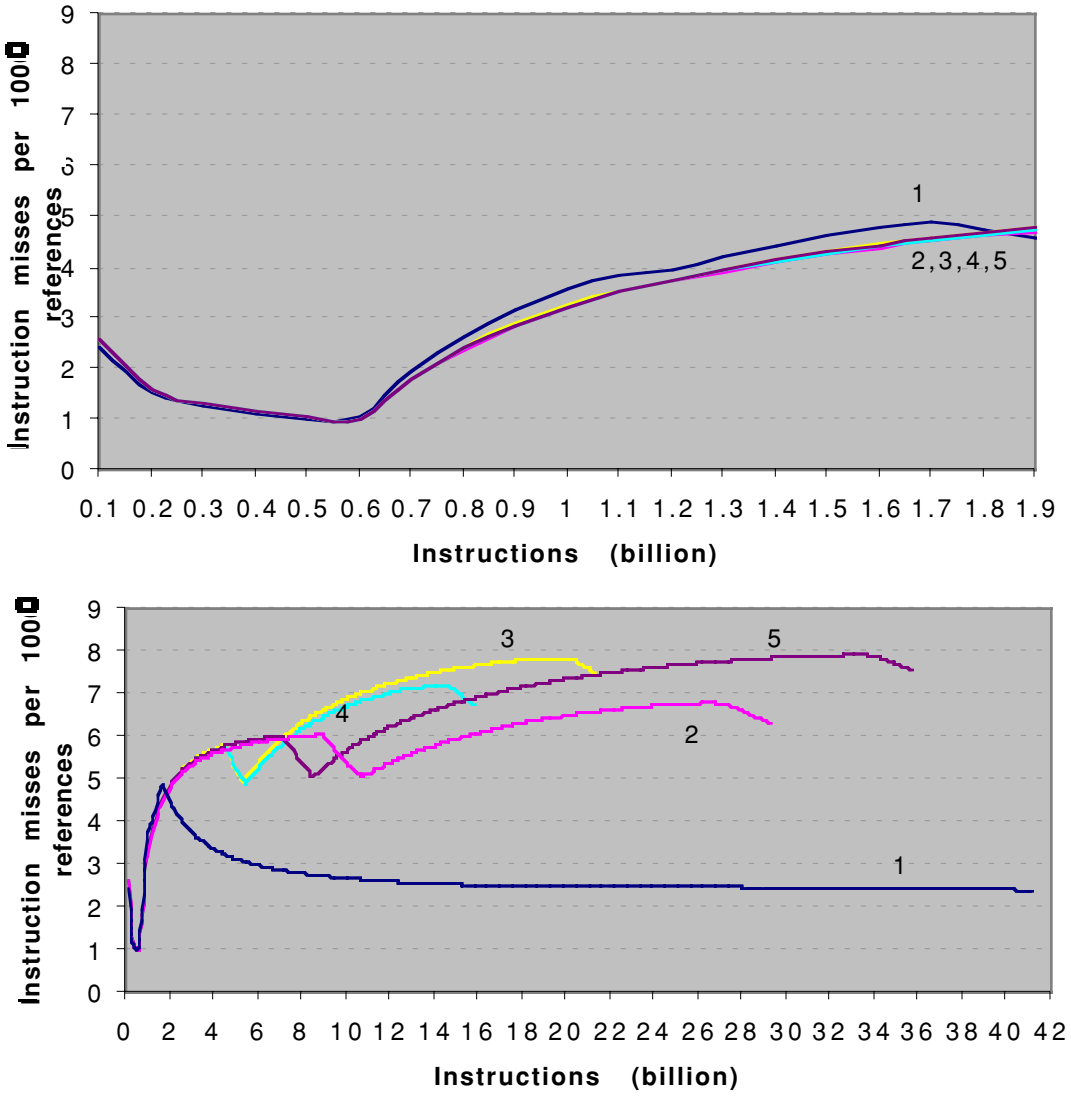

FIGURE 5.53 Instruction misses per 1000 references for five inputs to perl benchmark from SPEC2000. There is little variation in misses and little difference between the five inputs for the first 1.9 billion instructions, but running to completion shows how misses vary over the life of the program and how they depend on the input. The top graph shows the running average misses for the first 1.9 billion instructions, which starts at about 2.5 and ends at about 4.7 misses per 1000 references for all five inputs. The bottom graph shows the running average misses to run to competition, which takes 16 to 41 billion instructions depending on the input. After the first 1.9 billion instructions, the misses per 1000 references vary from 2.4 to 7.9 depending on the input. The simulations were for the Alpha processor using separate L1 caches for instructions and data, each 2-way 64KB with LRU, and a unified1MB direct-mapped L2 cache.

Figure 5.53 shows the cumulative average instruction misses per thousand instructions for five inputs to a single SPEC2000 program. For these inputs, the average memory rate for the first 1.9 billion instructions is very different from their average miss rate for the rest of the execution.

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 compulsory cache miss ratio of 1% running on a computer accessing memory 10 million times per second (at the time of the first edition) would access hundreds of megabytes of memory per second:

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

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

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, sold millions of units after the PDP-11 went out of production. Why?

The fatal flaw of the PDP-11 was the size of its addresses (16 bits) as compared to the address sizes of the IBM 360 (24 to 31 bits) and the VAX (32 bits). 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 computers 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.

Even the venerable 80x86 line is showing danger signs, with Intel justifying migration to IA-64 in part to provide a larger flat address space than 32 bits, and AMD proposing its own 64-bit address extension called x86-64.

As we expected, by this third edition every desktop and server microprocessor manufacturer offers computers with 64-bit flat addresses. DSPs and embedded applications, however, may yet be condemned to repeat history as memories grow and desired functions multiply.

Pitfall: Emphasizing memory bandwidth in DRAMs versus memory latency.

Direct RDRAM offers up to 1.6 GBytes/second of bandwidth from a single DRAM. When announced, the peak bandwidth was 8 times faster than individual conventional SDRAM chips.

PCs do most memory accesses through a two-level cache hierarchy, so its unclear how much benefit is gained from high bandwidth without also improving memory latency. According to Pabst [2000], when comparing PCs with 400 MHz DRDRAMs to PCs 133 MHz SDRAM, for office applications they had identical average performance. For games, DRDRAM was 1% to 2% faster. For professional graphics applications, it was 10% to 15% faster. The tests used a 800 MHz Pentium III (which integrates a 256-KB L2 cache), chip sets that support a 133 MHz system bus, and 128 MB of main memory.

	Modules		Dell XPS PCs					
ECC?	**No ECC**	**ECC**	**No ECC**			**ECC**		
Label	**DIMM**	**RIMM**	**A**	**B**	**B - A**	**C**	**D**	**D - C**
Memory or System?	**DRAM**		**System**		**DRAM**	**System**		**DRAM**
Memory Size (MB)	256	256	128	512	384	128	512	384
SDRAM PC100	$175	$259	$1,519	$2,139	$620	$1,559	$2,269	$710
DRDRAM PC700	$725	$826	$1,689	$3,009	$1,320	$1,789	$3,409	$1,620
Price Ratio DRDRAM/SDRAM	4.1	3.2	1.1	1.4	2.1	1.1	1.5	2.3

FIGURE 5.54 Comparison of price of SDRAM v. DRDRAM in memory modules and in systems in 2000. DRDRAM memory modules cost about a factor of four more without ECC and three more with ECC. Looking at the cost of the extra 384 MB of memory in PCs in going from 128 MB to 512 MB, DRDRAM costs twice as much. Except for differences in bandwidths of the DRAMs, the systems were identically configured. The Dell XPS PCs were identical except for memory: 800 MHz Pentium III, 20 GB ATA disk, 48X CDROM, 17" monitor, and Microsoft Windows 95/98 and Office 98. The module prices were the lowest found at pricewatch.com in June 2000. By September 2001 PC800 DRDRAM cost $76 for 256 MB while PC100 to PC150 SDRAM cost $15 to $23, or about a factor of 3.3 to 5.0 less expensive. (In September 2001 Dell did not offer systems whose only difference was type of DRAMs, hence we stick with the comparison from 2000.)

One measure of the RDRAM cost is about a 20% larger die for the same capacity compared to SDRAM. DRAM designers use redundant rows and columns to significantly improve yield on the memory portion of the DRAM, so a much

larger interface might have a disproportionate impact on yield. Yields are a closely guarded secret, but prices are not. Figure 5.54 compares prices of various versions of DRAM, in memory modules and in systems. Using this evaluation, in 2000 the price is about a factor of two to three higher for RDRAM.

RDRAM is at its strongest in small memory systems that need high bandwidth. The low cost of the Sony Playstation 2, for example, limits the amount of memory in the system to just two chips, yet its graphics has an appetite for high memory bandwidth. RDRAM is at its weakest in servers, where the large number of DRAM chips needed in even the minimal memory configuration make it easy to achieve high bandwidth with ordinary DRAMs.

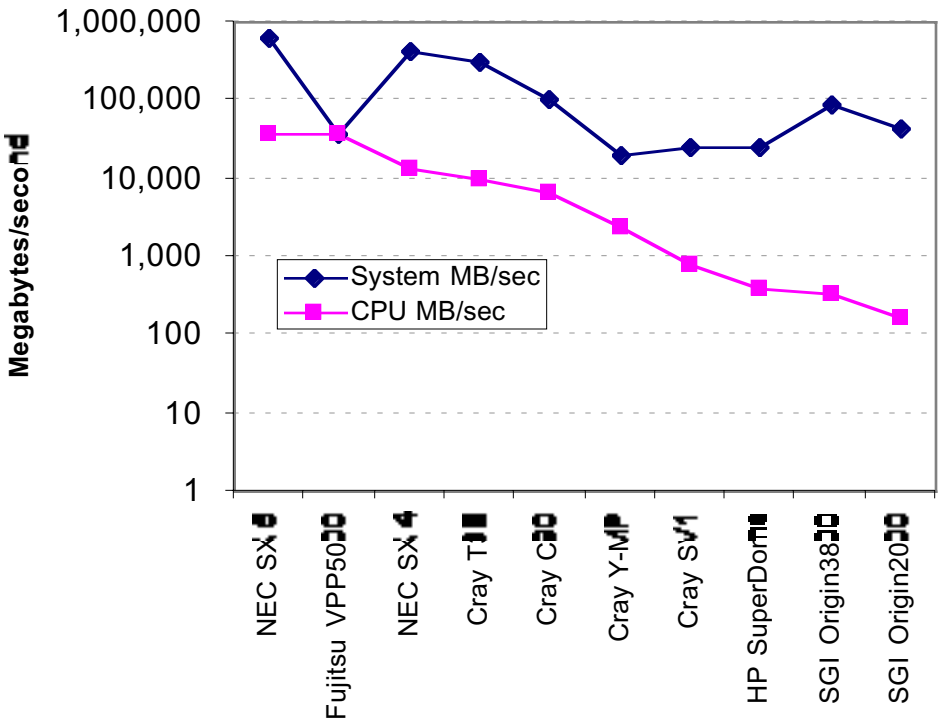

FIGURE 5.55 Top 10 in memory bandwidth as measured by the copy portion of the stream benchmark [McCalpin 2001]. Note that the last three computers are the only cache-based systems on the list, and that six of the top seven are vector computers. Systems use between 8 and 256 processors to achieve higher memory bandwidth. System bandwidth is bandwidth of all CPUs collectively. CPU bandwidth is simply system bandwidth divided by the number of CPUs. The STREAM benchmark is a simple synthetic benchmark program that measures sustainable memory bandwidth (in MB/s) for simple vector kernels. It is specifically works with data sets much larger than the available cache on any given system.

Pitfall: Delivering high memory bandwidth in a cache-based system.

Caches help with average cache memory latency but may not deliver high memory bandwidth to an application that needs it. Figure 5.55 shows the top ten results from the Stream benchmark as of 2001, which measures bandwidth to copy data [McCalpin 2001].The NEC SX 5 memory offers up to 16384 SDRAM memory banks to achieve its top ranking.

Only three computers rely on data caches, and they are the three slowest of the top ten, about a factor of a hundred slower than the fastest processor. Stated another way, a processor from 1988–the Cray YMP–still has a factor of 10 in memory bandwidth advantage over cache based processors from 2001.

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

Figure 5.56 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 5.56 Misses and time spent in misses for applications and operating system. The operating system adds about 25% to the execution time of the application. 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. 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. Data from Torrellas, Gupta, and Hennessy [1992].

Pitfall: Relying on the operating systems to change the page size over time.

The Alpha architects had an elaborate plan to grow the architecture over time by growing its page size, even building it into the size of its virtual address. When it

became time to grow page sizes with later Alphas, the operating system designers balked and the virtual memory system was revised to grow the address space while maintaining the 8-KB page.

Architects of other computers noticed very high TLB miss rates, and so added multiple, larger page sizes to the TLB. The hope was that operating systems programmers would allocate an object to the largest page that made sense, thereby preserving TLB entries. After a decade of trying, most operating systems use these "superpages" only for handpicked functions: mapping the display memory or other I/O devices, or using very large pages for the database code.

5.17 Concluding Remarks

Figure 5.57 compares the memory hierarchy of microprocessors aimed at desktop, server, and embedded applications. The L1 caches are similar across applications, with the primary differences being L2 cache size, die size, processor clock rate, and instructions issued per clock.

In contrast to showing the state of the art in a given year, Figure 5.56 shows evolution over a decade of the memory hierarchy of Alpha microprocessors and systems. The primary change between the Alpha 21064 and 21364 is the hundredfold increase in on-chip cache size, which tries to compensate for the sixfold increase in main memory latency as measured in instructions.

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. One question is whether increasing scale breaks any of our assumptions. Are L3 caches bigger than prior main memories a cost-effective solution? Do 8 KB pages makes sense with terabyte main memories?

The design decisions at all 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: prefetching, cache-aware compilers, and increasing page size. Fortunately, there tends to be a technological "sweet spot" in balancing cost, performance, power, and complexity: missing the target wastes performance, power, hardware, design time, debug time, or possibly all five. Architects hit the target by careful, quantitative analysis.

MPU	AMD Athlon	Intel Pentium III	Intel Pentium 4	IBM Power-PC 405CR	Sun UltraSPARC III
Instruction set architecture	80x86	80x86	80x86	PowerPC	SPARC v9
Intended application	desktop	desktop, server	desktop	embedded core	server
CMOS Process	0.18	0.18	0.18	0.25	0.15
Die size (mm^2)	128	106 to 385	217	37	210
Instructions issued/clock	3	3	3 RISC ops	1	4
Clock Rate (2001)	1400 MHz	900 - 1200 MHz	2000 MHz	266 MHz	900 MHz
Instruction Cache	64 KB, 2-way S.A.	16 KB, 2-way S.A.	12000 RISC op trace cache (~96 KB)	16 KB, 2-way S.A.	32 KB, 4-way S.A.
Latency (clocks)	3	3	4	1	2
Data Cache	64 KB, 2-way S.A.	16 KB, 2-way S.A.	8 KB, 4-way S.A.	8 KB, 2-way S.A.	64 KB, 4-way S.A.
Latency (clocks)	3	3	2	1	2
TLB entries (I/D/L2 TLB)	280/288	32/64	128	4/8/64	128/512
Min. page size	8 KB	8 KB	8 KB	1 KB	8 KB
On Chip L2 Cache	256 KB, 16-way S.A.	256 - 2048 KB, 8-way S.A.	256 KB, 8-way S.A.	--	--
Off Chip L2 Cache	--	--	--	--	8MB, 1-wayS.A.
Latency (clocks)	11	7	?	--	15
Block Size (L1/L2, bytes)	64	32	64/128	32	32
Memory bus width (bits)	64	64	64	64	128
Memory bus clock	133 MHz	133 MHz	400 MHz	133 MHz	150 MHz

FIGURE 5.57 Desktop, embedded and server microprocessors in 2001. From a memory hierarchy perspective, the primary differences between applications is L2 cache. There is no L2 cache for embedded, 256 KB on chip for desktop, and servers use 2MB on chip or 8 MB off chip. The processor clock rates also vary: 266 MHz for embedded, 900 MHz for servers, and 1200 to 2000 MHz for desktop. The Intel Pentium III includes the Xeon chip set for multiprocessor servers. It has the same processor core as the standard Pentium III, but a much larger on-chip L2 cache (up to 2 MB) and die size (385 mm^2) but a slower clock rate (900 MHz).

CPU	21064	21164	21264	21364
CMOS Process Feature Size	0.68	0.50	0.35	0.18
Clock Rate (Initial)	200	300	525	~ 1000
1st System Ship Date	3000 / 800	8400 5/300	ES40	2002-2003?
CPI gcc (SPECInt92/95)	2.51	1.27	0.63	~ 0.6
Instruction Cache	8 KB, 1-way	8 KB, 1-way	64 KB, 2-way	64 KB, 2-way
Latency (clocks)	2	2	2 or 3	2 or 3
Data Cache	8 KB, 1-way, Write Through	8 KB, 1-way, Write Through	64 KB, 2-way, Write Back	64 KB, 2-way, Write Back
Latency	2	2	3	3
Write/Victim Buffer	4 blocks	6 blocks	8 blocks	32 blocks
Block Size (bytes, all caches)	32	32	32 or 64	64
Virtual/Physical Address Size	43/34	43/40	48/44 or 43/41	48/44 or 43/41
Page size	8 KB	8 KB	8 KB	8 KB or 64 KB
Instruction TLB	12 entry, F.A	48 entry, F.A	128 entry, F.A	128 entry, F.A
Data TLB	32 entry, F.A	64 entry, F.A	128 entry, F.A	128 entry, F.A
Path Width Off Chip (bits)	128	128	128 to L2, 64 to memory	128?
On Chip Unified L2 Cache	---	96 KB, 3-way, Write Back	---	1536 KB, 6-way, Write Back
Latency (clocks)	---	7	---	12
Off Chip Unified L2 or L3 Cache	2 MB, 1-way, Write Back	4 MB, 1-way, Write Back	8 MB, 1-way, Write Back	---
Latency (clocks)	5	12	16	---
Memory size	.008 - 1 GB	0.125 - 14 GB	0.5 - 32 GB	0.5 - 4 GB / proc.
Latency (clocks)	68	80	122	~ 90
Latency (instructions)	27	63	194	~ 150

FIGURE 5.58 Four generations of Alpha microprocessors and systems. Instruction latency was calculated by dividing the latency in clock cycles by average CPI for SPECint programs. The 21364 integrates a large on-chip cache and a memory controller to connect directly to DRDRAM chips, thereby significantly lowering memory latency. The large on-chip cache and low latency to memory make an off-chip cache unnecessary. A network allows processors to access non-local memory with non-uniform access times (see Chapter 6): 30 clocks per network hop, so 120 clocks in the nearest group of 4 and 200 in a group of 16. Memory latency in instructions is calculated by dividing clocks by average CPI.

5.18 Historical Perspective and References

Although 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]. It was demonstrated with the Atlas computer at the University of Manchester. This computer appeared the year *before* the IBM 360 was announced. Although 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 computer 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 earlier Burroughs computers provided virtual memory using segmentation, similar to the segmented address scheme of the Intel 8086. The 80286, the first 80x86 to have the protection mechanisms described on pages 463 to 467, was inspired by the Multics protection software that ran on the GE 645. Over time, computers 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, despite 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. In addition, 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 only 4 more address bits than found in 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 programmers who stored extra information in the upper 8 "unused" address bit foiled the expansion effort. (Apple made a similar mistake 20 years later with the 24-bit address in the Motorola 68000, which required a procedure to later determine "32-bit clean" programs for the Macintosh when later 68000s used the full 32-bit virtual address.) Virtually every computer since then, 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. Although 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 computer 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 precedent is more than 30 years old, and yet many used miss rates until the early 1990s.

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), less memory interleaving (4 versus 16), and a slower main memory (1.04 microsecond versus 0.75 microsecond). This paper was also the first to use the term "cache."

Others soon expanded the cache literature. 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.

Although most studies relied on simulations, Clark [1983] used a hardware monitor to record cache misses of the VAX-11/780 over several days. Clark and Emer [1985] later compared simulations and hardware measurements for translations.

Hill [1987] proposed the three C's used in section 5.5 to explain cache misses. Jouppi [1998] retrospectively says that Hill's three Cs model led directly to his invention of the victim cache to take advantage of faster direct map caches and yet avoid most of the cost of conflict misses. Ugumar and Abraham' [1993] argue that the baseline cache for the three C's model should use optimal replacement; this eliminates the anomalies of LRU-based miss classification, and allows conflict misses to be broken down into those caused by mapping and those caused by a non-optimal replacement algorithm.

One of the first papers on nonblocking caches is by Kroft [1981]. Kroft [1998] later explained that he was the first to design a computer with a cache at Control Data Corporation, and when using old concepts for new mechanisms, he hit upon the idea of allowing his two-ported cache to continue to service other accesses on a miss.

Baer and Wang [1988] did one of the first examinations of multilevel inclusion property. Wang, Baer, and Levy [1989] then produced an early paper on performance evaluation of multilevel caches. Later, Jouppi and Wilton [1994] proposed multilevel exclusion for multilevel caches on chip.

In addition to victim caches, Jouppi [1990] also examined prefetching via streaming buffers. His work was extended by Farkas et al [1995] to that streaming buffers work well with non-blocking loads and speculative execution for in-order processors, and later Farkas et al [1997] showed that while out-of-order processors can tolerate unpredictable latency better, they still benefit. They also refined memory bandwidth demands of stream buffers.

Proceedings of the Symposium on Architectural Support for Compilers and Operating Systems (ASPLOS) and the International Computer Architecture Symposium (ISCA) from the 1990s are filled with papers on caches. (In fact some wags claimed ISCA really stood for the International *Cache* Architecture Symposium.)

This chapter relies on the measurements of SPEC2000 benchmarks collected by Cantin and Hill [2001]. There are several other papers used in this chapter that are cited in the captions of the figures that use the data: Agarwal and Pudar [1993]; Barroso, Gharachorloo, and Bugnion [1998]; Farkas and Jouppi [1994]; Jouppi [1990]; Lam, Rothberg, and Wolf [1991]; Mowry, Lam, and Gupta [1992]; Lebeck and Wood [1994]; McCalpin [2001]; and Torrellas, Gupta, and Hennessy [1992].

The Alpha architecture is described in detail by Bhandarkar [1995] and by Sites [1992], and sources of information on the 21264 are Compaq [1999], Cvetanovic and Kessler [2000], and Kessler[1999]. Two Emotion Engine references are Kunimatsu et al [2000] and Oka and Suzuoki [1999]. Information on the Sun Fire 6800 server is found primarily on Sun's web site, but Horel and Lauterbach [1999] and Heald, R. *et al* [2000] published detailed information on UltraSPARC III.

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.

Barroso, L.A., Gharachorloo, K. and E. Bugnion [1998]. "Memory System Characterization of Commercial Workloads," Proceedings 25th International Symposium on Computer Architecture, Barcelona (July), 3-14.

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*, Seattle, May 28–31, 270–9.

CABTIN, J. F. AND M. D.HILL, [2001]. *Cache Performance for Selected SPEC CPU2000 Benchmark*s, http://www.jfred.org/cache-data.html, (June).

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.

D. W. CLARK and J. S. EMER [1985], "Performance of the VAX-11/780 Translation Buffer: Simulation and Measurement," *ACM Trans. on Computer Systems*, 3, 1 (February 1985),31-62.

COMPAQ COMPUTER CORPORATION [1999] *Compiler Writer's Guide for the Alpha 21264* Order Number EC-RJ66A-TE, June, 112 pages. http://www1.support.compaq.com/alpha-tools/documentation/current/21264_EV67/ec-rj66a-te_comp_writ_gde_for_alpha21264.pdf

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.

CVETANOVIC, Z. and R.E. KESSLER [2000] "Performance Analysis of the Alpha 21264-based Compaq ES40 System." *Proc. 27th Annual Int'l Symposium on Computer Architecture*, Vancouver, Canada, June 10-14, IEEE Computer Society Press, 192-202.

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

Farkas, K.I., P. Chow, N.P. Jouppi,; Z. Vranesic [1997]. "Memory-system design considerations for dynamically-scheduled processors."24th Annual International Symposium on Computer Architecture. Denver, CO, USA, 2-4 June, 133-43.

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

Farkas, K.I., N.P.Jouppi, and P. Chow [1995]. "How useful are non-blocking loads, stream buffers and speculative execution in multiple issue processors?," Proceedings. First IEEE Symposium on High-Performance Computer Architecture, Raleigh, NC, USA, 22-25 Jan.,78-89.

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.

Heald, R. et al [2000]. " A third-generation SPARC V9 64-b microprocessor," *IEEE Journal of Solid-State Circuits*, 35:11 (Nov), 1526-38.

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.

Horel, T. and G. Lauterbach [1999]. "UltraSPARC-III: designing third-generation 64-bit performance," *IEEE Micro*, 19:3 (May-June), 73-85.

Hughes, C.J.; Kaul, P.; Adve, S.V.; Jain, R.; Park, C.; Srinivasan, J. [2001]. "Variability in the execution of multimedia applications and implications for architecture, " *Proc. 28th Annual International Symposium on Computer Architecture*, Goteborg, Sweden, 30 June-4 July , 254-65.

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*, 364–73.

JOUPPI, N. P. [1998]. "Retrospective: Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers."*25 Years of the International Symposia on Computer Architecture (Selected Papers)*. ACM, 71-73.

Jouppi, N.P. and S.J.E. Wilton [1994]. "Trade-offs in two-level on-chip caching. Proceedings the 21st Annual International Symposium on Computer Architecture, Chicago, IL, USA, (18-21 April).34-45.

KESSLER, R.E. [1999] "The Alpha 21264 microprocessor." *IEEE Micro*, vol.19, (no.2), March-April, 24-36.

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.

KROFT, D. [1998]. "Retrospective: Lockup-Free Instruction Fetch/Prefetch Cache Organization."*25 Years of the International Symposia on Computer Architecture (Selected Papers)*. ACM, 20-21.

KUNIMATSU, A., N. IDE, T. SATO, et al. [2000] "Vector unit architecture for emotion synthesis." *IEEE Micro*, vol.20, (no.2), IEEE, March-April, 40-7.

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.

LUK, C.-K. and T.C MOWRY[1999]. "Automatic compiler-inserted prefetching for pointer-based applications." *IEEE Transactions on Computer*s, vol.48, (no.2), IEEE, Feb. p.134-41.

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.

MCCALPIN, J.D. [2001]. *STREAM: Sustainable Memory Bandwidth in High Performance Computers* http://www.cs.virginia.edu/stream/.

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.

OKA, M. and M. SUZUOKI. [1999] "Designing and programming the emotion engine." *IEEE Micro*, vol.19, (no.6), Nov.-Dec., 20-8.

PABST, T. [2000], "Performance Showdown at 133 MHz FSB - The Best Platform for Coppermine," http://www6.tomshardware.com/mainboard/00q1/000302/.

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, 24–33.

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

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

REINMAN, G. and N. P. JOUPPI. [1999]. "Extensions to CACTI," http://research.compaq.com/wrl/people/jouppi/CACTI.html.

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.

STOKES, J. [2000], "Sound and Vision: A Technical Overview of the Emotion Engine," http://arstechnica.com/reviews/1q00/playstation2/ee-1.html.

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

Sugumar, R.A. and S.G. Abraham [1993]. "Efficient simulation of caches under optimal replacement with applications to miss characterization." 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems, Santa Clara, CA, USA, 17-21 May, p.24-35.

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

- Jouppi comments, 2nd pass: drop prime number exercises; mention that "L2 misses can be overlapped on 21264 when doing block copies; e.g., overlap reading of source block, writing of destination block. Stream benchmark uses this; put 21064 on the web, then still use exercise 5.11.
- Reinhart comments, 2nd pass: "In the prefetching section, it might be interesting to compare & contrast standard prefetches with the speculative load support found in IA-64 (or maybe this would make a good exercise)."
- Technical reviewer, 2nd pass: Suggests deriving components of accesses to arrays X, Y, and Z before and after in blocking example on page 419 to show how came up with "$2N^3 + N^2$ memory words accessed for N^3 operations" before blocking and "total number of memory words accessed is $2N^3/B + N^2$" after blocking. (See page 10 of technical review)
- The CACTI program mentioned in the section allows people to design caches and compare hit times as well as miss rates; I would propose projects that look at both, or possible use an existing exercise and look at the cache access times in different technologies: 0.18, 0.13, 0.10 to see how/if the trade-offs differ. Also, see how access times differ with different sized caches.
- Jouppi said his teaching experience at Stanford was that it was important to show a simple in-order CPU so that simple memory performance questions could be answered using spreadsheets before being exposed to the OOO simulators and traces. Thus he suggests saving the 21064 design in some detail and then include a series of exercises about it so that they can get the ideas here first. Perhaps just include a section in the exercises that describes the 21064 caches, and keep some of the old exercises?
- To really account for the impact of cache misses on OOO computers, there needs to be an OOO instruction simulator to go with the cache. A popular one is Simple Scalar from Doug Berger, originally from Wisconsin. Another is RSIM from Rice. I'd add the URLs and propose exercises which use them.
- Add a discussion question: given an Out-of-order CPU, how do you quantiatively evaluate memory options? Why is it different from an in-order CPU?
- Thus these exercises need to be sorted into whether to assume CPU in-order, and can use spreadsheet since it stalls, or out-of-order, and use a simulator since you cannot account for overlap. Perhaps can simply look at L2 cache

misses, assuming L1 are overlapped?

- One advanced exercise (level [25]) is to study the impact of out of order execution on temporal locality in an L1 data cache. The hypothesis is that there may be more conflict misses due to OOO execution, and that hence multiway set associativity may be more impact for OOO than for in-order CPUs. The idea would be to vary out-of-orderness in terms of the size of the load and store queues and to look at different miss rates as you vary associativity, and see if the CPU execution model makes much difference.
- Some of these examples with a short miss time (e.g., next one where miss is 10X a hit), state that there is a second level cache and for purposes of this equation assume that it doesn't miss, hence it makes sense to talk about "small" miss penalties. Real misses all the way to memory should be no less than 100 clock cycles
- Mark Hill and Norm Jouppi think we should emphasize "misses per 1000 instructions" vs. "miss rate" in this edition, so we need several exercises comparing them. I intend to replace some of the figures, or show it both ways, so this should be supported by exercises using MPI. This is especially true for L2 caches, which is much less confusing.
- Next is a great exercise. I'd just change/modernize the numbers so the answers are different. Perhaps make up another one that is similar, just to have some that aren't in the answer book?

5.1 [15/15/12/12] <5.1,5.2> Let's try to show how you can make *unfair* benchmarks. Here are two computers with the same processor and main memory but different cache organizations. Assume the miss time is 10 times a cache hit time for both computers. 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 computer A run as much faster as possible than computer B. (Be sure to state any further assumptions you need, if any.)

b. [15] <1.5,5.2> Describe a program that makes computer B run as much faster as possible than computer 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 computer A than computer B?

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

- Next is another interesting exercise. Just update graph example. 21264 would be great, if you had access to it.

5.2 [15/10/12/12/12/12/12/12/12/12/12] <5.5,5.4> 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 computers that use virtually addressed caches such as the Alpha 21264. In general, virtual addresses tend to follow physical addresses shortly after rebooting, so you may need to reboot the computer 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] <5.5,5.4> 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] <5.5,5.4> How many levels of cache are there?

c. [12] <5.5,5.4> 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] <5.5,5.4> What is the size of the second-level cache (if any)? Block size?

e. [12] <5.5,5.4> What is the associativity of the first-level cache? Second-level cache?

f. [12] <5.5,5.4> What is the page size?

g. [12] <5.5,5.4> How many entries are in the TLB?

h. [12] <5.5,5.4> What is the miss penalty for the first-level cache? Second-level?

i. [12] <5.5,5.4> 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] <5.5,5.4> What is the miss penalty for the TLB?

k. [12] <5.5,5.4> Is there anything else you have discovered about the memory hierarchy from these measurements?

- Replace Figure below with a newer computer, perhaps by getting results from someone who has run this on a recent computer, or worst case from the paper by Saveerda and Smith.

5.3 [10/10/10] <5.2> Figure 5.59 shows the output from running the program in Exercise 5.2 on a SPARCstation 1+, which has a single unified cache.

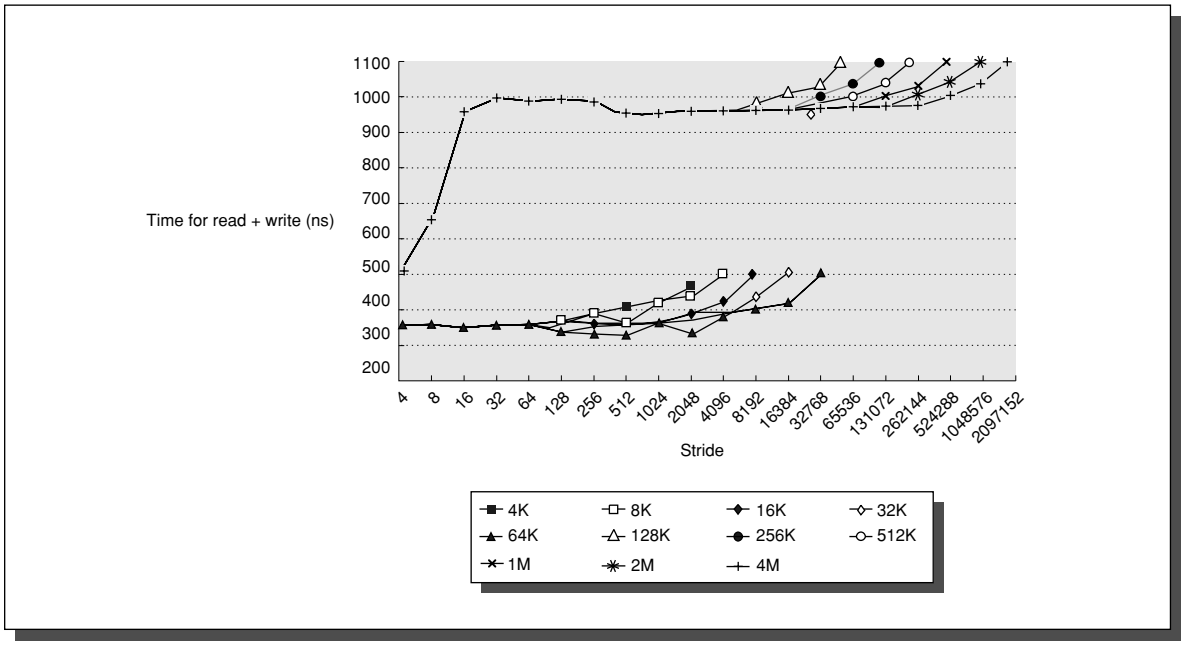

FIGURE 5.59 Results of running program in Exercise 5.2 on a SPARCstation 1+.

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).

- 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] <5.7> 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] <5.7> Using statistics for the average percentage of loads and stores from MIPS in Figure 2.32 on page 149, 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] <5.7> Do the same comparison, but this time assume the write-through cache pipelines the writes, so that a write hit takes just one clock cycle.

5.6 [20] <5.5> Improve on the compiler prefetch Example found on page 425: 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.

- The following example isn't there any more

5.7 [15/12] <5.5> 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] <5.5> 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] <5.5> Using the formula from part (a), recalculate the average memory access times for the two cases found on page 399 (8-KB cache and 256-KB cache). Which pseudo-associative scheme is faster for the given configurations and data?

- Perhaps next is a good in-order v. out-of-order exercise?

5.8 [15/20/15] <5.10> If the base CPI with a perfect memory system is 1.5, what is the CPI for these cache organizations? Use Figure 5.14 (page 409):

- 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] <5.5> Compute the effective CPI for the three caches assuming an ideal TLB.

b. [20] <5.5> Using the results from part (a), compute the effective CPI for the three caches with a real TLB.

c. [15] <5.5> 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?

- prime number of memory banks were dropped for the 3/e. Thus we must modify Exercise 5.10 which refers to the prime number of banks, at least to explain the ideas. Perhaps even add exercises, including an explanation of the ideas dropped from the second edition by using text from the second edition?

5.10 [15/15] <5.8> 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

   $2^0 \text{ modulo } 7 = 1$
   $2^1 \text{ modulo } 7 = 2$
   $2^2 \text{ modulo } 7 = 4$
   $2^3 \text{ modulo } 7 = 1$
   $2^4 \text{ modulo } 7 = 2$
   $2^5 \text{ modulo } 7 = 4$

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

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

   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 multiplica-

tions 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] <5.8> 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] <5.8> 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?

n Old, so drop?

5.11 [25/10/15] <5.8> 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] <5.8> Instruction miss rate.

b. [10] <5.8> Average instruction memory access time measured in clock cycles.

c. [15] <5.8> 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?

n The next 5 exercises all refer to traces. Since we no longer have traces readily available, these exercises should be changed to work with something like Burger's simulator running a program and producing addresses vs. an address trace, unless there are some traces left online someplace?

5.12 [25] <5.8> 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 5.25 (page 432) 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] <5.8> 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] <5.8> 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] <5.8> 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] <5.8> Use the cache simulator to see if their results hold for the traces.

5.15 [30] <5.10> 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. <5.10> No cache and no write buffer.

b. <5.10> A 64-KB direct-mapped write-through cache with four-word blocks.

c. <5.10> A 64-KB direct-mapped write-back cache with four-word blocks.

d. <5.10> A 64-KB direct-mapped write-through cache with four-word blocks but the "interleaving" comes from a page-mode DRAM.

e. <5.10> 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] <5.10> 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] <5.10> A 64-KB instruction-only cache?

b. [25] <5.10> A 64-KB data-only cache?

c. [25] <5.10> 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] <5.10> 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] <5.10> 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] <5.10> 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] <5.17> 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 5.2 on page 375 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 computers, with no pipelining or caches. Classes and textbooks may still reflect those simpler computers. What is the impact of the changes in computer architecture on these other fields? Find examples where textbooks suggest the solution appropriate for old computers but inappropriate for modern computers. Talk to people in other fields to see what they think about these changes.