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. Locality occurs in time (*temporal locality*) and in space (*spatial locality*). This principle, plus the guideline that smaller hardware can be made 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 per byte almost as low as the cheapest level of memory and speed almost as fast as the fastest level.

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. Figure 5.2 plots processor performance projections against the historical performance improvement in time to access main memory. Clearly, computer architects must try to close the processor-memory gap.

The increasing size and thus importance of this gap led to the migration of the basics of memory hierarchy into undergraduate courses in computer architecture, and even to courses in operating systems and compilers. Thus, we'll start with a quick review of caches. The bulk of the chapter, however, describes more advanced innovations that address the processor-memory performance gap.

When a word is not found in the cache, the word must be fetched from the memory and placed in the cache before continuing. Multiple words, called a

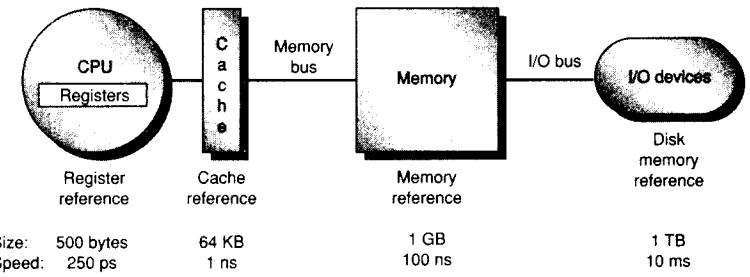

Figure 5.1 The levels in a typical memory hierarchy in embedded, desktop, and server computers. As we move farther away from the processor, the memory in the level below becomes slower and larger. Note that the time units change by factors of 10—from picoseconds to milliseconds—and that the size units change by factors of 1000—from bytes to terabytes.

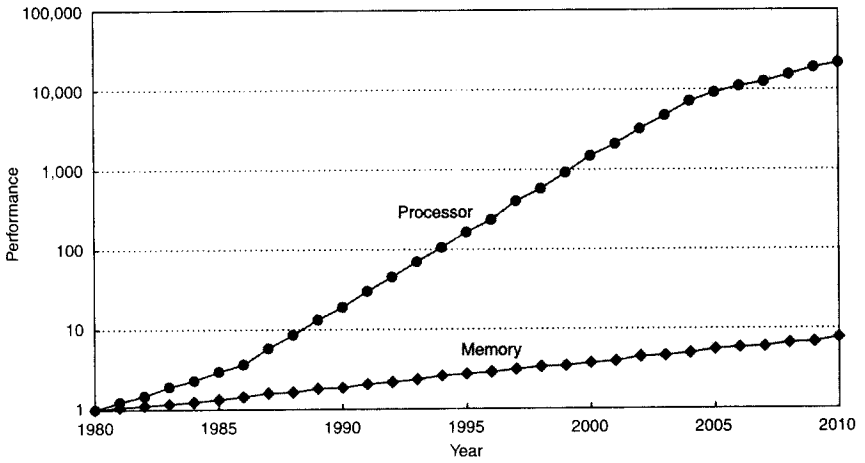

Figure 5.2 Starting with 1980 performance as a baseline, the gap in performance between memory and processors is plotted over time. Note that the vertical axis must be on a logarithmic scale to record the size of the processor-DRAM performance gap. The memory baseline is 64 KB DRAM in 1980, with a 1.07 per year performance improvement in latency (see Figure 5.13 on page 313). The processor line assumes a 1.25 improvement per year until 1986, and a 1.52 improvement until 2004, and a 1.20 improvement thereafter; see Figure 1.1 in Chapter 1.

block (or *line*), are moved for efficiency reasons. Each cache block includes a *tag* to see which memory address it corresponds to.

A key design decision is where blocks (or lines) can be placed in a cache. The most popular scheme is *set associative*, where 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. Finding a block consists of first mapping the block address to the set, and then searching the set—usually in parallel—to find the block. The set is chosen by the address of the data:

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

If there are n blocks in a set, the cache placement is called *n-way set associative*. The end points of set associativity have their own names. A *direct-mapped* cache has just one block per set (so a block is always placed in the same location), and a *fully associative* cache has just one set (so a block can be placed anywhere).

Caching data that is only read is easy, since the copy in the cache and memory will be identical. Caching writes is more difficult: how can the copy in the cache and memory be kept consistent? There are two main strategies. A *write-through* cache updates the item in the cache *and* writes through to update main memory. A *write-back* cache only updates the copy in the cache. When the block is about to be replaced, it is copied back to memory. Both write strategies can use a *write buffer* to allow the cache to proceed as soon as the data is placed in the buffer rather than wait the full latency to write the data into memory.

One measure of the benefits of different cache organizations is miss rate. *Miss rate* is simply the fraction of cache accesses that result in a miss—that is, the number of accesses that miss divided by the number of accesses.

To gain insights into the causes of high miss rates, which can inspire better cache designs, the three Cs model 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. Compulsory misses are those that occur even if you had an infinite cache.
- *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 not fully associative, conflict misses (in addition to compulsory and capacity misses) will occur because a block may be discarded and later retrieved if conflicting blocks map to its set.

Figures C.8 and C.9 on pages C-23 and C-24 show the relative frequency of cache misses broken down by the "three C's." (Chapter 4 adds a fourth C, for *Coherency* misses due to cache flushes to keep multiple caches coherent in a multiprocessor; we won't consider those here.)

Alas, miss rate can be a misleading measure for several reasons. Hence, some designers prefer measuring *misses per instruction* rather than misses per memory reference (miss rate). These two are related:

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

(It is often reported as misses per 1000 instructions to use integers instead of fractions.) For speculative processors, we only count instructions that commit.

The problem with both measures is that they don't factor in the cost of a miss. A better measure 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 and *Miss penalty* is the time to replace the block from memory (that is, the cost of a miss). 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. For example, in Chapter 2 we saw that speculative processors may execute other instructions during a miss, thereby reducing the effective miss penalty.

If this material is new to you, or if this quick review moves too quickly, see Appendix C. It covers the same introductory material in more depth and includes examples of caches from real computers and quantitative evaluations of their effectiveness.

Section C.3 in Appendix C also presents six basic cache optimizations, which we quickly review here. The appendix also gives quantitative examples of the benefits of these optimizations.

1. *Larger block size to reduce miss rate*—The simplest way to reduce the miss rate is to take advantage of spatial locality and increase the block size. Note that larger blocks also reduce compulsory misses, but they also increase the miss penalty.
2. *Bigger caches to reduce miss rate*—The obvious way to reduce capacity misses is to increase cache capacity. Drawbacks include potentially longer hit time of the larger cache memory and higher cost and power.
3. *Higher associativity to reduce miss rate*—Obviously, increasing associativity reduces conflict misses. Greater associativity can come at the cost of increased hit time.
4. *Multilevel caches to reduce miss penalty*—A difficult decision is whether to make the cache hit time fast, to keep pace with the increasing clock rate of processors, or to make the cache large, to overcome the widening gap between the processor and main memory. Adding another level of cache between the original cache and memory simplifies the decision (see Figure 5.3). The first-level cache can be small enough to match a fast clock cycle time, yet the second-level cache can be large enough to capture many accesses that would go to main memory. The focus on misses in second-level caches leads to larger blocks, bigger capacity, and higher associativity. If L1 and L2 refer, respectively, to first- and second-level caches, we can redefine the average memory access time:

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

5. *Giving priority to read misses over writes to reduce miss penalty*—A write buffer is a good place to implement this optimization. Write buffers create hazards because they hold the updated value of a location needed on a read miss—that is, a read-after-write hazard through memory. One solution is to check the contents of the write buffer on a read miss. If there are no conflicts, and if the memory system is available, sending the read before the writes reduces the miss penalty. Most processors give reads priority over writes.
6. *Avoiding address translation during indexing of the cache to reduce hit time*—Caches must cope with the translation of a virtual address from the processor to a physical address to access memory. (Virtual memory is covered in Sections 5.4 and C.4.) Figure 5.3 shows a typical relationship between caches, translation lookaside buffers (TLBs), and virtual memory. A common optimization is to use the page offset—the part that is identical in both virtual and physical addresses—to index the cache. The virtual part of the address is translated while the cache is read using that index, so the tag match can use physical addresses. This scheme allows the cache read to begin immediately, and yet the tag comparison still uses physical addresses. The drawback of this *virtually indexed, physically tagged* optimization is that the size of the page limits the size of the cache. For example, a direct-mapped cache can be no bigger than the page size. Higher associativity can keep the cache index in the physical part of the address and yet still support a cache larger than a page.

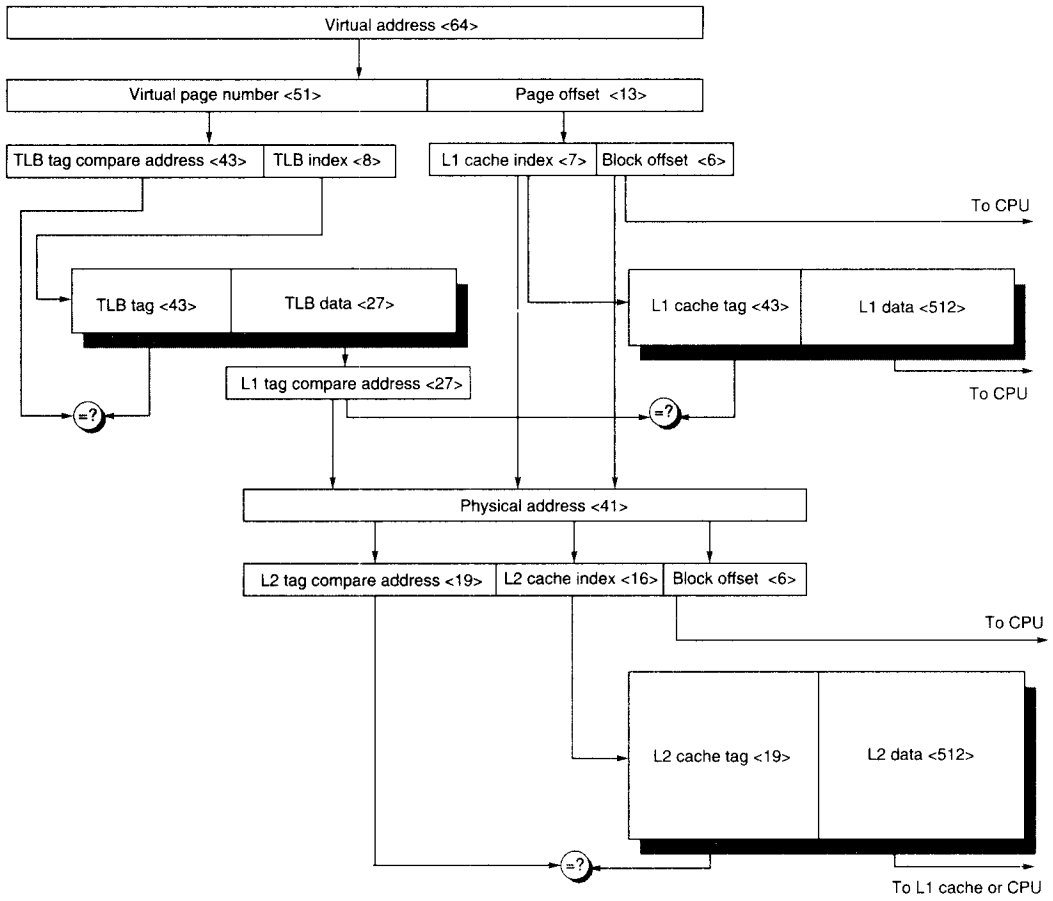

Figure 5.3 The overall picture of a hypothetical memory hierarchy going from virtual address to L2 cache access. The page size is 8 KB. The TLB is direct mapped with 256 entries. The L1 cache is a direct-mapped 8 KB, and the L2 cache is a direct-mapped 4 MB. Both use 64-byte blocks. The virtual address is 64 bits and the physical address is 40 bits. The primary difference between this figure and a real memory hierarchy, as in Figure 5.18 on page 327, is higher associativity for caches and TLBs and a smaller virtual address than 64 bits.

For example, doubling associativity while doubling the cache size maintains the size of the index, since it is controlled by this formula:

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

A seemingly obvious alternative is to just use virtual addresses to access the cache, but this can cause extra overhead in the operating system.

Note that each of these six optimizations above has a potential disadvantage that can lead to increased, rather than decreased, average memory access time.

The rest of this chapter assumes familiarity with the material above, including Figure 5.3. To put cache ideas into practice, throughout this chapter (and Appendix C) we show examples from the memory hierarchy of the AMD Opteron microprocessor. Toward the end of the chapter, we evaluate the impact of this hierarchy on performance using the SPEC2000 benchmark programs.

The Opteron is a microprocessor designed for desktops 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 have hundreds of users running potentially dozens of applications simultaneously. These characteristics result in more context switches, which effectively increase miss rates. Thus, desktop computers are concerned more with average latency from the memory hierarchy, whereas server computers are also concerned about memory bandwidth.

5.2 Eleven Advanced Optimizations of Cache Performance

The average memory access time formula above gives us three metrics for cache optimizations: hit time, miss rate, and miss penalty. Given the popularity of superscalar processors, we add cache bandwidth to this list. Hence, we group 11 advanced cache optimizations into the following categories:

- Reducing the hit time: small and simple caches, way prediction, and trace caches
- Increasing cache bandwidth: pipelined caches, multibanked caches, and nonblocking caches
- Reducing the miss penalty: critical word first and merging write buffers
- Reducing the miss rate: compiler optimizations
- Reducing the miss penalty or miss rate via parallelism: hardware prefetching and compiler prefetching

We will conclude with a summary of the implementation complexity and the performance benefits of the 11 techniques presented (Figure 5.11 on page 309).

First Optimization: Small and Simple Caches to Reduce Hit Time

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. Smaller hardware can be faster, so a small cache can help the hit time. It is also critical to keep an L2 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. One 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 lower-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.

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 for three generations of AMD microprocessors: K6, Athlon, and Opteron. The emphasis is on fast clock rate while hiding L1 misses with dynamic execution and using L2 caches to avoid going to memory.

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 5.4 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 time for direct mapped is 1.2–1.5 times faster than two-way set associative; two-way is 1.02–1.11 times faster than four-way; and four-way is 1.0–1.08 times faster than fully associative.

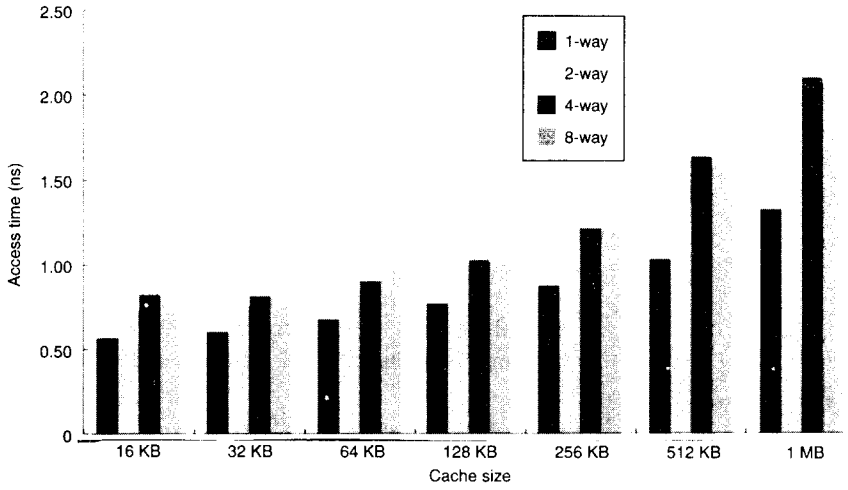

Figure 5.4 Access times as size and associativity vary in a CMOS cache. These data are based on the CACTI model 4 4.0 by Tarjan, Thoziyoor, and Jouppi [2006]. They assumed 90 nm feature size, a single bank, and 64-byte blocks. The median ratios of access time relative to the direct-mapped caches are 1.32, 1.39, and 1.43 for 2-way, 4-way, and 8-way associative caches, respectively.

Example Assume that the hit time of a two-way set-associative first-level data cache is 1.1 times faster than a four-way set-associative cache of the same size. The miss rate falls from 0.049 to 0.044 for an 8 KB data cache, according to Figure C.8 in Appendix C. Assume a hit is 1 clock cycle and that the cache is the critical path for the clock. Assume the miss penalty is 10 clock cycles to the L2 cache for the two-way set-associative cache, and that the L2 cache does not miss. Which has the faster average memory access time?

Answer For the two-way cache:

$$\begin{aligned}\text{Average memory access time}_{2\text{-way}} &= \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}\\ &= 1 + 0.049 \times 10 = 1.49\end{aligned}$$

For the four-way cache, the clock time is 1.1 times longer. The elapsed time of the miss penalty should be the same since it's not affected by the processor clock rate, so assume it takes 9 of the longer clock cycles:

$$\begin{aligned}\text{Average memory access time}_{4\text{-way}} &= \text{Hit time} \times 1.1 + \text{Miss rate} \times \text{Miss penalty}\\ &= 1.1 + 0.044 \times 9 = 1.50\end{aligned}$$

If it really stretched the clock cycle time by a factor of 1.1, the performance impact would be even worse than indicated by the average memory access time, as the clock would be slower even when the processor is not accessing the cache.

Despite this advantage, since many processors take at least 2 clock cycles to access the cache, L1 caches today are often at least two-way associative.

Second Optimization: Way Prediction to Reduce Hit Time

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 way, or block within the set of the *next* cache access. This prediction means the multiplexor is set early to select the desired block, and only a single tag comparison is performed that clock cycle in parallel with reading the cache data. A miss results in checking the other blocks for matches in the next clock cycle.

Added to each block of a cache are block predictor bits. The bits select which of the blocks to try on the *next* cache access. If the predictor is correct, the cache access latency is the fast hit time. If not, it tries the other block, changes the way predictor, and has a latency of one extra clock cycle. Simulations suggested set prediction accuracy is in excess of 85% for a two-way set, so way prediction saves pipeline stages more than 85% of the time. Way prediction is a good match to speculative processors, since they must already undo actions when speculation is unsuccessful. The Pentium 4 uses way prediction.

Third Optimization: Trace Caches to Reduce Hit Time

A challenge in the effort to find lots of instruction-level parallelism is to find enough instructions every cycle without use dependencies. To address this challenge, blocks in a *trace cache* contain dynamic traces of the executed instructions rather than static sequences of instructions as determined by layout in memory. Hence, the branch prediction is folded into the cache and must be validated along with the addresses to have a valid fetch.

Clearly, trace caches have much more complicated address-mapping mechanisms, as the addresses are no longer aligned to power-of-two multiples of the word size. However, they can better utilize long blocks in the instruction cache. Long blocks in conventional caches may be entered in the middle from a branch and exited before the end by a branch, so they can have poor space utilization. The downside of trace caches is that conditional branches making different choices result in the same instructions being part of separate traces, which each occupy space in the trace cache and lower its space efficiency.

Note that the trace cache of the Pentium 4 uses decoded micro-operations, which acts as another performance optimization since it saves decode time.

Many optimizations are simple to understand and are widely used, but a trace cache is neither simple nor popular. It is relatively expensive in area, power, and complexity compared to its benefits, so we believe trace caches are likely a one-time innovation. We include them because they appear in the popular Pentium 4.

Fourth Optimization: Pipelined Cache Access to Increase Cache Bandwidth

This optimization is simply to pipeline cache access so that the effective latency of a first-level cache hit can be multiple clock cycles, giving fast clock cycle time and high bandwidth but slow hits. For example, the pipeline for the Pentium took 1 clock cycle to access the instruction cache, for the Pentium Pro through Pentium III it took 2 clocks, and for the Pentium 4 it takes 4 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 Chapter 2).

Fifth Optimization: Nonblocking Caches to Increase Cache Bandwidth

For pipelined computers that allow out-of-order completion (Chapter 2), the processor need not stall on a data cache miss. For example, the processor could continue fetching instructions from the instruction cache while waiting for the data cache to return the missing data. A *nonblocking cache* or *lockup-free cache* escalates the potential benefits of such a scheme by allowing the data cache to continue to supply cache hits during a miss. This "hit under miss" optimization

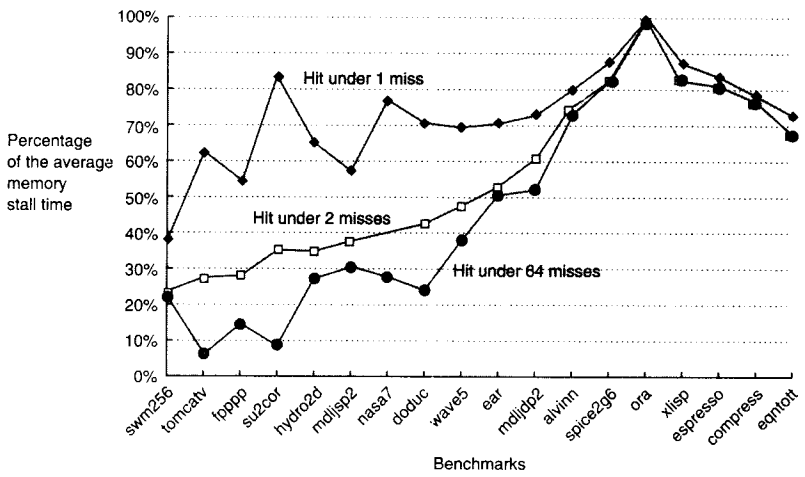

Figure 5.5 Ratio of the average memory stall time for a blocking cache to hit-under-miss schemes as the number of outstanding misses is varied for 18 SPEC92 programs. The hit-under-64-misses line allows one miss for every register in the processor. The first 14 programs are floating-point programs: the 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]. Although it may be a good model for L1 misses to L2 caches, it would be interesting to redo this experiment with SPEC2006 benchmarks and modern assumptions on miss penalty.

reduces the effective miss penalty by being helpful during a miss instead of ignoring the requests of the processor. 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.

Figure 5.5 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. As pointed out in Chapter 3, the number of simultaneous outstanding misses limits achievable instruction-level parallelism in programs.

Example 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 and the cache described in Figure 5.5, which we assume has a L2 cache.

Answer The numbers for Figure 5.5 were based on a miss penalty of 16 clock cycles assuming an L2 cache. Although the programs are older and this is low for a miss penalty, let's stick with it for consistency. (To see how well it would work on modern programs and miss penalties, we'd need to redo this experiment.) 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 for two-way are thus 1.71/1.84 or 93% of direct-mapped cache. The caption of Figure 5.5 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

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

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

The memory stalls of two-way are thus 0.96/1.18 or 81% of direct-mapped cache. The caption of Figure 5.5 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 using this data.

The real difficulty with performance evaluation of nonblocking caches is that a cache miss does not necessarily stall the processor. In this case, it is difficult to judge the impact of any single miss, and hence difficult to calculate the average memory access time. The effective miss penalty is not the sum of the misses but the nonoverlapped time that the processor is stalled. The benefit of nonblocking caches is complex, as it depends upon the miss penalty when there are multiple misses, the memory reference pattern, and how many instructions the processor can execute with a miss outstanding.

In general, out-of-order processors are capable of hiding much of the miss penalty of an L1 data cache miss that hits in the L2 cache, but are not capable of hiding a significant fraction of an L2 cache miss.

Sixth Optimization: Multibanked Caches to Increase Cache Bandwidth

Rather than treat the cache as a single monolithic block, we can divide it into independent banks that can support simultaneous accesses. Banks were origi-

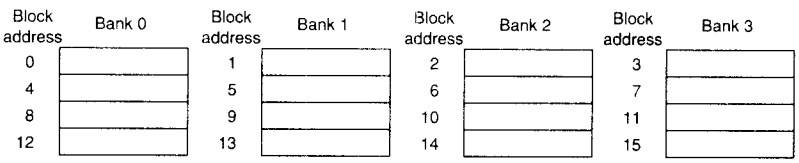

Figure 5.6 Four-way interleaved cache banks using block addressing. Assuming 64 bytes per blocks, each of these addresses would be multiplied by 64 to get byte addressing.

nally used to improve performance of main memory and are now used inside modern DRAM chips as well as with caches. The L2 cache of the AMD Opteron has two banks, and the L2 cache of the Sun Niagara has four banks.

Clearly, banking works best when the accesses naturally spread themselves across the banks, so the mapping of addresses to banks affects the behavior of the memory system. A simple mapping that works well is to spread the addresses of the block sequentially across the banks, called *sequential interleaving*. For example, if there are four banks, bank 0 has all blocks whose address modulo 4 is 0; bank 1 has all blocks whose address modulo 4 is 1; and so on. Figure 5.6 shows this interleaving.

Seventh Optimization: Critical Word First and Early Restart to Reduce Miss Penalty

This technique is based on the observation that the processor 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 processor. Here are two specific strategies:

- *Critical word first*—Request the missed word first from memory and send it to the processor as soon as it arrives; let the processor continue execution while filling the rest of the words in the block.
- *Early restart*—Fetch the words in normal order, but as soon as the requested word of the block arrives, send it to the processor and let the processor continue execution.

Generally, these techniques only benefit designs with large cache blocks, since the benefit is low unless blocks are large. Note that caches normally continue to satisfy accesses to other blocks while the rest of the block is being filled.

Alas, given spatial locality, there is a good chance that the next reference is to the rest of the block. Just as with nonblocking caches, the miss penalty is not simple to calculate. When there is a second request in critical word first, the effective miss penalty is the nonoverlapped time from the reference until the second piece arrives.

Example Let's assume a computer has a 64-byte cache block, an L2 cache that takes 7 clock cycles to get the critical 8 bytes, and then 1 clock cycle per 8 bytes + 1 extra clock cycle to fetch the rest of the block. (These parameters are similar to the AMD Opteron.) Without critical word first, it's 8 clock cycles for the first 8 bytes and then 1 clock per 8 bytes for the rest of the block. 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 read data 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 7 clock cycles for critical word first, and without critical word first it takes 8 + (8 – 1) x 1 or 15 clock cycles for the processor to read a full cache block. Thus, for one word, the answer is 15 versus 7 clock cycles. The Opteron issues two loads per clock cycle, so it takes 8/2 or 4 clocks to issue the loads. Without critical word first, it would take 19 clock cycles to load and read the full block. With critical word first, it's 7 + 7 x 1 + 1 or 15 clock cycles to read the whole block, since the loads are overlapped in critical word first. For the full block, the answer is 19 versus 15 clock cycles.

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.

Eighth Optimization: Merging Write Buffer to Reduce Miss Penalty

Write-through caches rely on write buffers, as all stores must be sent to the next lower level of the hierarchy. 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 processor's perspective; the processor 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 a valid write buffer entry. If so, the new data are combined with that entry. *Write merging* is the name of this optimization. The Sun Niagara processor, among many others, uses write merging.

If the buffer is full and there is no address match, the cache (and processor) 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. Skadron and Clark [1997] found that about 5% to 10% of performance was lost due to stalls in a four-entry write buffer.

The optimization also reduces stalls due to the write buffer being full. Figure 5.7 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

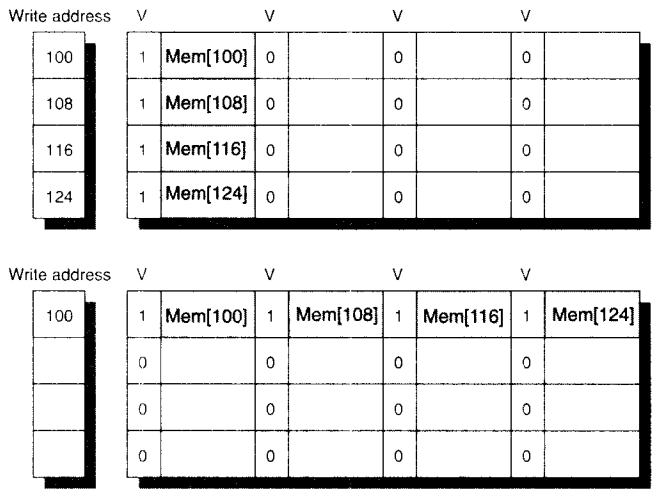

Figure 5.7 To illustrate write merging, the write buffer on top does not use it while the write buffer on the bottom does. The four writes are merged into a single buffer entry with write merging; without it, the buffer is full even though three-fourths of each entry is wasted. The buffer has four entries, and each entry holds four 64-bit words. The address for each entry is on the left, with a valid bit (V) indicating whether the next sequential 8 bytes in this entry are occupied. (Without write merging, the words to the right in the upper part of the figure would only be used for instructions that wrote multiple words at the same time.)

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.

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

In a write-back cache, the block that is replaced is sometimes called the *victim*. Hence, the AMD Opteron calls its write buffer a *victim buffer*. The write victim buffer or victim buffer contains the dirty blocks that are discarded from a cache because of a miss. Rather than stall on a subsequent cache miss, the contents of the buffer are checked on a miss to see if they have the desired data before going to the next lower-level memory. This name makes it sounds like another optimization called a *victim cache*. In contrast, the victim cache can include any blocks discarded from the cache on a miss, whether they are dirty or not [Jouppi 1990].

While the purpose of the write buffer is to allow the cache to proceed without waiting for dirty blocks to write to memory, the goal of a victim cache is to reduce the impact of conflict misses. Write buffers are far more popular today than victim caches, despite the confusion caused by the use of "victim" in their title.

Ninth Optimization: Compiler Optimizations to Reduce Miss Rate

Thus far, our techniques have required changing the hardware. This next 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. The optimizations presented below are found in many modern compilers.

Code and Data Rearrangement

Code can easily be rearranged without affecting correctness; for example, reordering the procedures of a program might reduce instruction miss rates by reducing conflict misses [McFarling 1989]. 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. If the compiler knows that a branch is likely to be taken, it can improve spatial locality by changing the sense of the branch and swapping the basic block at the branch target with the basic block sequentially after the branch. *Branch straightening* is the name of this optimization.

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—the cause of most misses in scientific codes—can be changed to operate on all data in a cache block rather than blindly striding through arrays in the order that the programmer wrote 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 they are stored. Assuming the arrays do not fit in the cache, this technique reduces misses by improving spatial locality; reordering maximizes use of data in a cache block before they are 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.

Blocking

This optimization improves temporal locality to reduce misses. 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 loop iteration. Such orthogonal accesses mean that transformations such as loop interchange still leave plenty of room for improvement.

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 5.8 gives a snapshot of the accesses to the three arrays. A dark shade indicates a recent access, a light shade indicates an older access, and white means not yet accessed.

The number of capacity misses clearly depends on N and the size of the cache. If it can hold all three N-by-N matrices, then all is well, provided there are no cache conflicts. If the cache can hold one N-by-N matrix and one row of N, then at least the ith row of y and the array z may stay in the cache. Less than that and misses may occur for both x and z. In the worst case, there would be $2N^3 + N^2$ 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.)

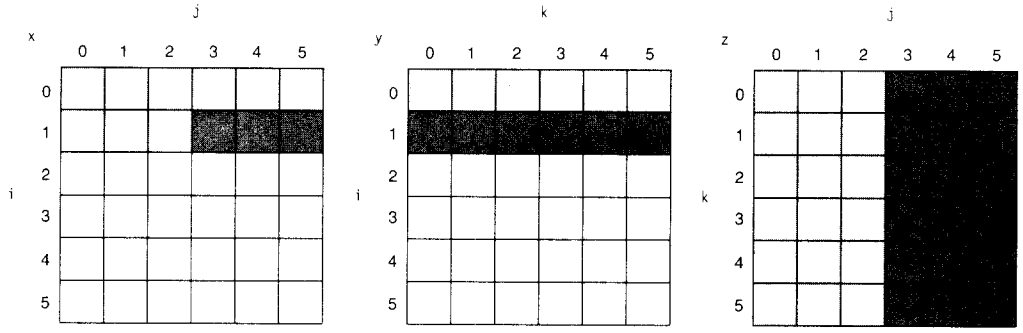

Figure 5.8 A snapshot of the three arrays x, y, and z when $N = 6$ and i = 1. The age of accesses to the array elements is indicated by shade: white means not yet touched, light means older accesses, and dark means newer accesses. Compared to Figure 5.9, elements of y and z are read repeatedly to calculate new elements of x. The variables i, j, and k are shown along the rows or columns used to access the arrays.

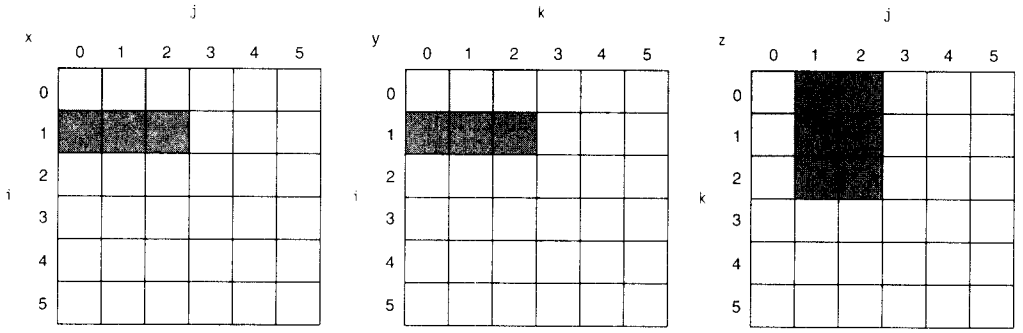

Figure 5.9 The age of accesses to the arrays x, y, and z when $B = 3$. Note in contrast to Figure 5.8 the smaller number of elements accessed.

```
/* 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 5.9 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.

Tenth Optimization: Hardware Prefetching of Instructions and Data to Reduce Miss Penalty or Miss Rate

Nonblocking caches effectively reduce the miss penalty by overlapping execution with memory access. Another approach is to prefetch items before the processor requests them. 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.

A similar approach can be applied to data accesses [Jouppi 1990]. Palacharla and Kessler [1994] looked at a set of scientific programs and considered multiple 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 Intel Pentium 4 can prefetch data into the second-level cache from up to eight streams from eight different 4 KB pages. Prefetching is invoked if there are two successive L2 cache misses to a page, and if the distance between those cache blocks is less than 256 bytes. (The stride limit is 512 bytes on some models of the Pentium 4.) It won't prefetch across a 4 KB page boundary.

Figure 5.10 shows the overall performance improvement for a subset of SPEC2000 programs when hardware prefetching is turned on. Note that this figure includes only 2 of 12 integer programs, while it includes the majority of the SPEC floating-point programs.

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.

Eleventh Optimization: Compiler-Controlled Prefetching to Reduce Miss Penalty or Miss Rate

An alternative to hardware prefetching is for the compiler to insert prefetch instructions to request data before the processor needs it. There are two flavors of prefetch:

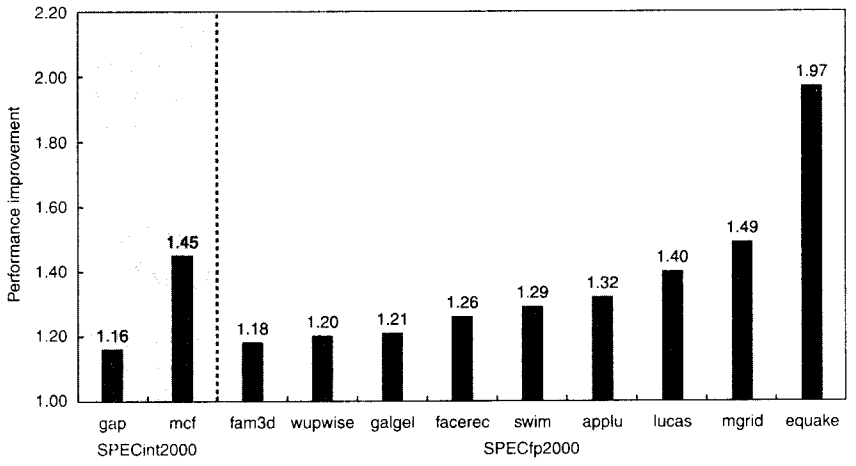

Figure 5.10 Speedup due to hardware prefetching on Intel Pentium 4 with hardware prefetching turned on for 2 of 12 SPECint2000 benchmarks and 9 of 14 SPECfp2000 benchmarks. Only the programs that benefit the most from prefetching are shown; prefetching speeds up the missing 15 SPEC benchmarks by less than 15% [Singhal 2004].

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

Either of these can be *faulting* or *nonfaulting;* that is, the address does or does not cause an exception for virtual address faults and protection violations. Using this terminology, a normal load instruction could be considered a "faulting register prefetch instruction." Nonfaulting prefetches simply turn into no-ops if they would normally result in an exception, which is what we want.

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 nonfaulting cache prefetches. This section assumes nonfaulting cache prefetch, also called *nonbinding* prefetch.

Prefetching makes sense only if the processor can proceed while prefetching the data; 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 (see Appendix G) or unrolls many times to prefetch data for a future iteration.

Issuing prefetch instructions incurs an instruction overhead, however, so compilers must take care 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 since 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 for a plus 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 already be in the cache, or we will pay the miss penalty of the first few elements of a or b. Nor will we worry about suppressing the prefetches at the end of the loop that try to prefetch beyond the end of a (a[i][100] . . . a[i][106]) and the end of b (b[101][0] . . . b[107][0]). If these were faulting prefetches, we could not take this luxury. Let's assume that the miss penalty is so large we need to start prefetching at least, say, seven iterations in advance. (Stated alternatively, we assume prefetching has no benefit until the eighth iteration.) We underline the changes to the code above needed to add prefetching.

```
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[100][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 ([7/2]) for elements `a[0][0]`, `a[0][1]`, . . . , `a[0][6]` in the first loop (spatial locality reduces misses to 1 per 16-byte cache block)
- 4 misses ([7/2]) for elements `a[1][0]`, `a[1][1]`, . . . , `a[1][6]` in the second loop
- 4 misses ([7/2]) 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.

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 10 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.

Cache Optimization Summary

The techniques to improve hit time, bandwidth, miss penalty, and miss rate generally affect the other components of the average memory access equation as well as the complexity of the memory hierarchy. Figure 5.11 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.

Technique	Hit time	Band-width	Miss pen-alty	Miss rate	Hardware cost/ complexity	Comment
Small and simple caches	+			–	0	Trivial; widely used
Way-predicting caches	+				1	Used in Pentium 4
Trace caches	+				3	Used in Pentium 4
Pipelined cache access	–	+			1	Widely used
Nonblocking caches		+	+		3	Widely used
Banked caches		+			1	Used in L2 of Opteron and Niagara
Critical word first and early restart			+		2	Widely used
Merging write buffer			+		1	Widely used with write through
Compiler techniques to reduce cache misses				+	0	Software is a challenge; some computers have compiler option
Hardware prefetching of instructions and data			+	+	2 instr., 3 data	Many prefetch instructions; Opteron and Pentium 4 prefetch data
Compiler-controlled prefetching			+	+	3	Needs nonblocking cache; possible instruction overhead; in many CPUs

Figure 5.11 Summary of 11 advanced cache optimizations showing impact on cache performance and complexity. 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.

5.3 Memory Technology and Optimizations

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

Maurice Wilkes
Memoirs of a Computer Pioneer (1985)

Main memory is the next level down in the hierarchy. Main memory satisfies the demands of caches and serves as the I/O interface, as it is the destination of input as well as the source for output. Performance measures of main memory emphasize both latency and bandwidth. 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 multiprocessors and I/O. Chapter 4 discusses the relationship of main memory and multiprocessors, and Chapter 6 discusses the relationship of main memory and I/O.

Although caches benefit from low-latency memory, it is generally easier to improve memory bandwidth with new organizations than it is to reduce latency. The popularity of second-level caches, and their larger block sizes, makes main memory bandwidth 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 processor-DRAM performance gap, but simply making caches larger or adding more levels of caches cannot eliminate the gap. Innovations in main memory are needed as well.

In the past, the innovation was how to organize the many DRAM chips that made up the main memory, such as multiple memory banks. Higher bandwidth is available using memory banks, by making memory and its bus wider, or doing both.

Ironically, as capacity per memory chip increases, there are fewer chips in the same-sized memory system, reducing chances for innovation. For example, a 2 GB main memory takes 256 memory chips of 64 Mbit (16M × 4 bits), easily organized into 16 64-bit-wide banks of 16 memory chips. However, it takes only 16 256M × 4-bit memory chips for 2 GB, making one 64-bit-wide bank the limit. Since computers are often sold and benchmarked with small, standard memory configurations, manufacturers cannot rely on very large memories to get bandwidth. This shrinking number of chips in a standard configuration shrinks the importance of innovations at the board level.

Hence, memory innovations are now happening inside the DRAM chips themselves. This section describes the technology inside the memory chips and those innovative, internal organizations. Before describing the technologies and 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.

Virtually all desktop or server computers since 1975 used DRAMs for main memory, and virtually all use SRAMs for cache, our first topic.

SRAM Technology

The first letter of SRAM stands for *static*. The dynamic nature of the circuits in DRAM requires 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 don't need to refresh and so the access time is very close to the cycle time. SRAMs typically use six transistors per bit to prevent the information from being disturbed when read. SRAM needs only minimal power to retain the charge in standby mode.

SRAM designs are concerned with speed and capacity, while in DRAM designs the emphasis is on cost per bit and capacity. For memories designed in comparable technologies, the capacity of DRAMs is roughly 4–8 times that of SRAMs. The cycle time of SRAMs is 8–16 times faster than DRAMs, but they are also 8–16 times as expensive.

DRAM Technology

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 5.12 shows the basic DRAM organization. One-half of the address is sent first, called the *row access strobe* (RAS). The other half of the address, sent during the *column access strobe* (CAS), follows it. These names come from the internal chip organization, since the memory is organized as a rectangular matrix addressed by rows and columns.

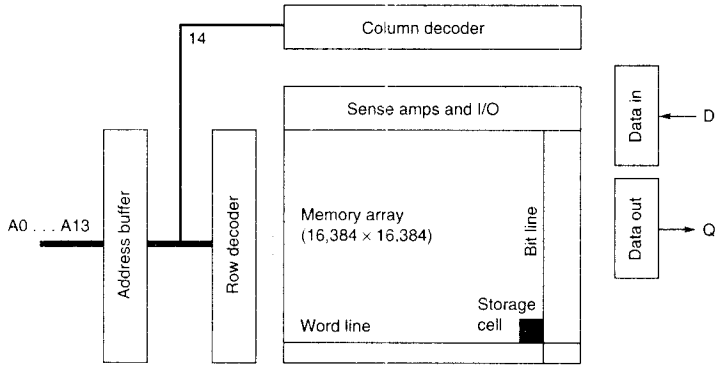

Figure 5.12 Internal organization of a 64M bit DRAM. DRAMs often use banks of memory arrays internally and select between them. For example, instead of one 16,384 × 16,384 memory, a DRAM might use 256 1024 × 1024 arrays or 16 2048 × 2048 arrays.

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 destroys the information, so it must be restored. This is one reason the DRAM cycle time is much longer than the access time. In addition, to prevent loss of information when a bit is not read or written, the 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 ms. Memory controllers include hardware to refresh the DRAMs periodically.

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 less than 5% of the total time.

So far we have presented main memory as if it operated like a Swiss train, consistently delivering the goods exactly according to schedule. Refresh belies that analogy, since 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 processor speed to keep a balanced system, so that a 1000 MIPS processor should have 1000 MB of memory. Processor designers rely on DRAMs to supply that demand: In the past, they expected a fourfold improvement in capacity every three years, or 55% per year. Unfortunately, the performance of DRAMs is growing at a much slower rate. Figure 5.13 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.

Although we have been talking about individual chips, DRAMs are commonly sold on small boards called *dual inline memory modules* (DIMMs). DIMMs typically contain 4–16 DRAMs, and they are normally organized to be 8 bytes wide (+ ECC) 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. DRAMs obeyed Moore's Law for 20 years, bringing out a new chip with four times the capacity every three years. Due to a slowing in demand for DRAMs, since 1998 new chips only double capacity every two years. In 2006, this new slower pace shows signs of further deceleration.

Improving Memory Performance inside a DRAM Chip

As Moore's Law continues to supply more transistors and as the processor-memory gap increases pressure on memory performance, the ideas of the previous section have made their way inside the DRAM chip. Generally, innovation

Year of introduction	Chip size	Row access strobe (RAS)		Column access strobe (CAS)/ data transfer time (ns)	Cycle time (ns)
		Slowest DRAM (ns)	Fastest DRAM (ns)		
1980	64K bit	180	150	75	250
1983	256K bit	150	120	50	220
1986	1M bit	120	100	25	190
1989	4M bit	100	80	20	165
1992	16M bit	80	60	15	120
1996	64M bit	70	50	12	110
1998	128M bit	70	50	10	100
2000	256M bit	65	45	7	90
2002	512M bit	60	40	5	80
2004	1G bit	55	35	5	70
2006	2G bit	50	30	2.5	60

Figure 5.13 Times of fast and slow DRAMs with each generation. (Cycle time is defined on page 310.) Performance improvement of row access time is about 5% per year. The improvement by a factor of 2 in column access in 1986 accompanied the switch from NMOS DRAMs to CMOS DRAMs.

has led to greater bandwidth, sometimes 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—16K bits for 256M bits, 64K bits for 1G bits, 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 1–4M bits. Thus, if you were to examine a 1G bit DRAM under a microscope, you might see 512 memory arrays, each of 2M bits, on the chip. This large number of arrays internally presents the opportunity to provide much higher bandwidth off chip.

To improve bandwidth, there has 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 1024–2048 bits for each access.

Conventional DRAMs had an asynchronous interface to the memory controller, and hence every transfer involved overhead to synchronize with the controller. The second major change was to add a clock signal to the DRAM interface, so that the repeated transfers would not bear that overhead. *Synchronous DRAM*

Standard	Clock rate (MHz)	M transfers per second	DRAM name	MB/sec /DIMM	DIMM name
DDR	133	266	DDR266	2128	PC2100
DDR	150	300	DDR300	2400	PC2400
DDR	200	400	DDR400	3200	PC3200
DDR2	266	533	DDR2-533	4264	PC4300
DDR2	333	667	DDR2-667	5336	PC5300
DDR2	400	800	DDR2-800	6400	PC6400
DDR3	533	1066	DDR3-1066	8528	PC8500
DDR3	666	1333	DDR3-1333	10,664	PC10700
DDR3	800	1600	DDR3-1600	12,800	PC12800

Figure 5.14 Clock rates, bandwidth, and names of DDR DRAMS and DIMMs in 2006. Note the numerical relationship between the columns. The third column is twice the second, and the fourth uses the number from the third column in the name of the DRAM chip. The fifth column is eight times the third column, and a rounded version of this number is used in the name of the DIMM. Although not shown in this figure, DDRs also specify latency in clock cycles. The name DDR400 CL3 means that memory delays 3 clock cycles of 5 ns each—the clock period a 200 MHz clock—before starting to deliver the request data. The exercises explore these details further.

(SDRAM) is the name of this optimization. SDRAMs typically also had a programmable register to hold the number of bytes requested, and hence can send many bytes over several cycles per request.

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* (DDR). To supply data at these high rates, DDR SDRAMs activate multiple banks internally.

The bus speeds for these DRAMs are also 133–200 MHz, but these DDR DIMMs are confusingly labeled by the peak *DIMM* bandwidth. Hence, the DIMM name PC2100 comes from 133 MHz × 2 × 8 bytes or 2100 MB/sec. Sustaining the confusion, the chips themselves are labeled with *the number of bits per second* rather than their clock rate, so a 133 MHz DDR chip is called a DDR266. Figure 5.14 shows the relationship between clock rate, transfers per second per chip, chip name, DIMM bandwidth, and DIMM name.

Example Suppose you measured a new DDR3 DIMM to transfer at 16000 MB/sec. What do you think its name will be? What is the clock rate of that DIMM? What is your guess of the name of DRAMs used in that DIMM?

Answer A good guideline is to assume that DRAM marketers picked names with the biggest numbers. The DIMM name is likely PC16000. The clock rate of the DIMM is

$$\begin{aligned}\text{Clock rate} \times 2 \times 8 &= 16000\\ \text{Clock rate} &= 16000/16\\ \text{Clock rate} &= 1000\end{aligned}$$

or 1000 MHz and 2000 M transfers per second, so the DRAM name is likely to be DDR3-2000.

DDR is now a sequence of standards. DDR2 lowers power by dropping the voltage from 2.5 volts to 1.8 volts and offers higher clock rates: 266 MHz, 333 MHz, and 400 MHz. DDR3 drops voltage to 1.5 volts and has a maximum clock speed of 800 MHz.

In each of these three cases, the advantage of such optimizations is that they add a small amount of logic to exploit the high potential internal DRAM bandwidth, adding little cost to the system while achieving a significant improvement in bandwidth.

5.4 Protection: Virtual Memory and Virtual Machines

A virtual machine is taken to be an efficient, isolated duplicate *of the real machine. We explain these notions through the idea of a* virtual machine monitor *(VMM). . . . a VMM has three essential characteristics. First, the VMM provides an environment for programs which is essentially identical with the original machine; second, programs run in this environment show at worst only minor decreases in speed; and last, the VMM is in complete control of system resources.*

Gerald Popek and Robert Goldberg
"Formal requirements for virtualizable third generation architectures,"
Communications of the ACM (July 1974)

Security and privacy are two of the most vexing challenges for information technology in 2006. Electronic burglaries, often involving lists of credit card numbers, are announced regularly, and it's widely believed that many more go unreported. Hence, both researchers and practitioners are looking for new ways to make computing systems more secure. Although protecting information is not limited to hardware, in our view real security and privacy will likely involve innovation in computer architecture as well as in systems software.

This section starts with a review of the architecture support for protecting processes from each other via virtual memory. It then describes the added protection provided from virtual machines, the architecture requirements of virtual machines, and the performance of a virtual machine.

Protection via Virtual Memory

Page-based virtual memory, including a translation lookaside buffer that caches page table entries, is the primary mechanism that protects processes from each

other. Sections C.4 and C.5 in Appendix C review virtual memory, including a detailed description of protection via segmentation and paging in the 80x86. This subsection acts as a quick review; refer to those sections if it's too quick.

Multiprogramming, where several programs running concurrently would share a computer, led to demands for protection and sharing among programs and 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. At any instant, it must be possible to switch from one process to another. This exchange is called a *process switch* or *context switch*.

The operating system and architecture join forces to allow processes to share the hardware yet not interfere with each other. To do this, the architecture must limit what a process can access when running a user process yet allow an operating system process to access more. At the minimum, the architecture must do the following:

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 or a *supervisor* process.
2. Provide a portion of the processor state that a user process can use but not write. This state includes an user/supervisor mode bit(s), an exception enable/disable bit, and memory protection information. Users are prevented from writing this state because the operating system cannot control user processes if users can give themselves supervisor privileges, disable exceptions, or change memory protection.
3. Provide mechanisms whereby the processor 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 processor is placed in supervisor mode. The return to user mode is like a subroutine return that restores the previous user/supervisor mode.
4. Provide mechanisms to limit memory accesses to protect the memory state of a process without having to swap the process to disk on a context switch.

Appendix C describes several memory protection schemes, but by far the most popular is adding protection restrictions to each page of virtual memory. Fixed-sized pages, typically 4 KB or 8 KB long, are mapped from the virtual address space into physical address space via a page table. The protection restrictions are included in each page table entry. The protection restrictions might determine whether a user process can read this page, whether a user process can write to this page, and whether code can be executed from this page. In addition, a process can neither read nor write a page if it is not in the page table. Since only the OS can update the page table, the paging mechanism provides total access protection.

Paged virtual memory 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 would be far too dear. The solution is to 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 lookaside buffer* (TLB).

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 address, protection field, valid bit, and usually a use bit and a dirty bit. The operating system changes 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.

Assuming the computer faithfully obeys the restrictions on pages and maps virtual addresses to physical addresses, it would seem that we are done. Newspaper headlines suggest otherwise.

The reason we're not done is that we depend on the accuracy of the operating system as well as the hardware. Today's operating systems consist of tens of millions of lines of code. Since bugs are measured in number per thousand lines of code, there are thousands of bugs in production operating systems. Flaws in the OS have led to vulnerabilities that are routinely exploited.

This problem, and the possibility that not enforcing protection could be much more costly than in the past, has led some to look for a protection model with a much smaller code base than the full OS, such as Virtual Machines.

Protection via Virtual Machines

An idea related to virtual memory that is almost as old is Virtual Machines (VM). They were first developed in the late 1960s, and they have remained an important part of mainframe computing over the years. Although largely ignored in the domain of single-user computers in the 1980s and 1990s, they have recently gained popularity due to

- the increasing importance of isolation and security in modern systems,
- the failures in security and reliability of standard operating systems,
- the sharing of a single computer among many unrelated users, and
- the dramatic increases in raw speed of processors, which makes the overhead of VMs more acceptable.

The broadest definition of VMs includes basically all emulation methods that provide a standard software interface, such as the Java VM. We are interested in VMs that provide a complete system-level environment at the binary instruction set architecture (ISA) level. Although some VMs run different ISAs in the VM from the native hardware, we assume they always match the hardware. Such VMs are called (Operating) *System Virtual Machines*. IBM VM/370, VMware ESX Server, and Xen are examples. They present the illusion that the users of a VM

have an entire computer to themselves, including a copy of the operating system. A single computer runs multiple VMs and can support a number of different operating systems (OSes). On a conventional platform, a single OS "owns" all the hardware resources, but with a VM, multiple OSes all share the hardware resources.

The software that supports VMs is called a *virtual machine monitor* (VMM) or *hypervisor;* the VMM is the heart of Virtual Machine technology. The underlying hardware platform is called the *host*, and its resources are shared among the *guest* VMs. The VMM determines how to map virtual resources to physical resources: A physical resource may be time-shared, partitioned, or even emulated in software. The VMM is much smaller than a traditional OS; the isolation portion of a VMM is perhaps only 10,000 lines of code.

In general, the cost of processor virtualization depends on the workload. User-level processor-bound programs, such as SPEC CPU2006, have zero virtualization overhead because the OS is rarely invoked so everything runs at native speeds. I/O-intensive workloads generally are also OS-intensive, which execute many system calls and privileged instructions that can result in high virtualization overhead. The overhead is determined by the number of instructions that must be emulated by the VMM and how slowly they are emulated. Hence, when the guest VMs run the same ISA as the host, as we assume here, the goal of the architecture and the VMM is to run almost all instructions directly on the native hardware. On the other hand, if the I/O-intensive workload is also *I/O-bound*, the cost of processor virtualization can be completely hidden by low processor utilization since it is often waiting for I/O (as we will see later in Figures 5.15 and 5.16).

Although our interest here is in VMs for improving protection, VMs provide two other benefits that are commercially significant:

1. *Managing software*. VMs provide an abstraction that can run the complete software stack, even including old operating systems like DOS. A typical deployment might be some VMs running legacy OSes, many running the current stable OS release, and a few testing the next OS release.
2. *Managing hardware*. One reason for multiple servers is to have each application running with the compatible version of the operating system on separate computers, as this separation can improve dependability. VMs allow these separate software stacks to run independently yet share hardware, thereby consolidating the number of servers. Another example is that some VMMs support migration of a running VM to a different computer, either to balance load or to evacuate from failing hardware.

Requirements of a Virtual Machine Monitor

What must a VM monitor do? It presents a software interface to guest software, it must isolate the state of guests from each other, and it must protect itself from guest software (including guest OSes). The qualitative requirements are

- Guest software should behave on a VM exactly as if it were running on the native hardware, except for performance-related behavior or limitations of fixed resources shared by multiple VMs.
- Guest software should not be able to change allocation of real system resources directly.

To "virtualize" the processor, the VMM must control just about everything—access to privileged state, address translation, I/O, exceptions and interrupts—even though the guest VM and OS currently running are temporarily using them.

For example, in the case of a timer interrupt, the VMM would suspend the currently running guest VM, save its state, handle the interrupt, determine which guest VM to run next, and then load its state. Guest VMs that rely on a timer interrupt are provided with a virtual timer and an emulated timer interrupt by the VMM.

To be in charge, the VMM must be at a higher privilege level than the guest VM, which generally runs in user mode; this also ensures that the execution of any privileged instruction will be handled by the VMM. The basic requirements of system virtual machines are almost identical to those for paged virtual memory listed above:

- At least two processor modes, system and user.
- A privileged subset of instructions that is available only in system mode, resulting in a trap if executed in user mode. All system resources must be controllable only via these instructions.

(Lack of) Instruction Set Architecture Support for Virtual Machines

If VMs are planned for during the design of the ISA, it's relatively easy to both reduce the number of instructions that must be executed by a VMM and how long it takes to emulate them. An architecture that allows the VM to execute directly on the hardware earns the title *virtualizable,* and the IBM 370 architecture proudly bears that label.

Alas, since VMs have been considered for desktop and PC-based server applications only fairly recently, most instruction sets were created without virtualization in mind. These culprits include 80x86 and most RISC architectures.

Because the VMM must ensure that the guest system only interacts with virtual resources, a conventional guest OS runs as a user mode program on top of the VMM. Then, if a guest OS attempts to access or modify information related to hardware resources via a privileged instruction—for example, reading or writing the page table pointer—it will trap to the VMM. The VMM can then effect the appropriate changes to corresponding real resources.

Hence, if any instruction that tries to read or write such sensitive information traps when executed in user mode, the VMM can intercept it and support a virtual version of the sensitive information as the guest OS expects.

In the absence of such support, other measures must be taken. A VMM must take special precautions to locate all problematic instructions and ensure that they behave correctly when executed by a guest OS, thereby increasing the complexity of the VMM and reducing the performance of running the VM.

Sections 5.5 and 5.7 give concrete examples of problematic instructions in the 80x86 architecture.

Impact of Virtual Machines on Virtual Memory and I/O

Another challenge is virtualization of virtual memory, as each guest OS in every VM manages its own set of page tables. To make this work, the VMM separates the notions of *real* and *physical memory* (which are often treated synonymously), and makes real memory a separate, intermediate level between virtual memory and physical memory. (Some use the terms *virtual memory, physical memory,* and *machine memory* to name the same three levels.) The guest OS maps virtual memory to real memory via its page tables, and the VMM page tables map the guests' real memory to physical memory. The virtual memory architecture is specified either via page tables, as in IBM VM/370 and the 80x86, or via the TLB structure, as in many RISC architectures.

Rather than pay an extra level of indirection on every memory access, the VMM maintains a *shadow page table* that maps directly from the guest virtual address space to the physical address space of the hardware. By detecting all modifications to the guest's page table, the VMM can ensure the shadow page table entries being used by the hardware for translations correspond to those of the guest OS environment, with the exception of the correct physical pages substituted for the real pages in the guest tables. Hence, the VMM must trap any attempt by the guest OS to change its page table or to access the page table pointer. This is commonly done by write protecting the guest page tables and trapping any access to the page table pointer by a guest OS. As noted above, the latter happens naturally if accessing the page table pointer is a privileged operation.

The IBM 370 architecture solved the page table problem in the 1970s with an additional level of indirection that is managed by the VMM. The guest OS keeps its page tables as before, so the shadow pages are unnecessary. AMD has proposed a similar scheme for their Pacifica revision to the 80x86.

To virtualize the TLB architected in many RISC computers, the VMM manages the real TLB and has a copy of the contents of the TLB of each guest VM. To pull this off, any instructions that access the TLB must trap. TLBs with Process ID tags can support a mix of entries from different VMs and the VMM, thereby avoiding flushing of the TLB on a VM switch. Meanwhile, in the background, the VMM supports a mapping between the VMs' virtual Process IDs and the real Process IDs.

The final portion of the architecture to virtualize is I/O. This is by far the most difficult part of system virtualization because of the increasing number of I/O devices attached to the computer *and* the increasing diversity of I/O device types. Another difficulty is the sharing of a real device among multiple VMs, and yet

another comes from supporting the myriad of device drivers that are required, especially if different guest OSes are supported on the same VM system. The VM illusion can be maintained by giving each VM generic versions of each type of I/O device driver, and then leaving it to the VMM to handle real I/O.

The method for mapping a virtual to physical I/O device depends on the type of device. For example, physical disks are normally partitioned by the VMM to create virtual disks for guest VMs, and the VMM maintains the mapping of virtual tracks and sectors to the physical ones. Network interfaces are often shared between VMs in very short time slices, and the job of the VMM is to keep track of messages for the virtual network addresses to ensure that guest VMs receive only messages intended for them.

An Example VMM: The Xen Virtual Machine

Early in the development of VMs, a number of inefficiencies became apparent. For example, a guest OS manages its virtual to real page mapping, but this mapping is ignored by the VMM, which performs the actual mapping to physical pages. In other words, a significant amount of wasted effort is expended just to keep the guest OS happy. To reduce such inefficiencies, VMM developers decided that it may be worthwhile to allow the guest OS to be aware that it is running on a VM. For example, a guest OS could assume a real memory as large as its virtual memory so that no memory management is required by the guest OS.

Allowing small modifications to the guest OS to simplify virtualization is referred to as *paravirtualization*, and the open source Xen VMM is a good example. The Xen VMM provides a guest OS with a virtual machine abstraction that is similar to the physical hardware, but it drops many of the troublesome pieces. For example, to avoid flushing the TLB, Xen maps itself into the upper 64 MB of the address space of each VM. It allows the guest OS to allocate pages, just checking to be sure it does not violate protection restrictions. To protect the guest OS from the user programs in the VM, Xen takes advantage of the four protection levels available in the 80x86. The Xen VMM runs at the highest privilege level (0), the guest OS runs at the next level (1), and the applications run at the lowest privilege level (3). Most OSes for the 80x86 keep everything at privilege levels 0 or 3.

For subsetting to work properly, Xen modifies the guest OS to not use problematic portions of the architecture. For example, the port of Linux to Xen changed about 3000 lines, or about 1% of the 80x86-specific code. These changes, however, do not affect the application-binary interfaces of the guest OS.

To simplify the I/O challenge of VMs, Xen recently assigned privileged virtual machines to each hardware I/O device. These special VMs are called *driver domains*. (Xen calls its VMs "domains.") Driver domains run the physical device drivers, although interrupts are still handled by the VMM before being sent to the appropriate driver domain. Regular VMs, called *guest domains*, run simple virtual device drivers that must communicate with the physical device drivers in the driver domains over a channel to access the physical I/O hardware. Data are sent between guest and driver domains by page remapping.

Figure 5.15 compares the relative performance of Xen for six benchmarks. According to these experiments, Xen performs very close to the native performance of Linux. The popularity of Xen, plus such performance results, led standard releases of the Linux kernel to incorporate Xen's paravirtualization changes.

A subsequent study noticed that the experiments in Figure 5.15 were based on a single Ethernet network interface card (NIC), and the single NIC was a performance bottleneck. As a result, the higher processor utilization of Xen did not affect performance. Figure 5.16 compares TCP receive performance as the number of NICs increases from 1 to 4 for native Linux and two configurations of Xen:

1. *Xen privileged VM only (driver domain).* To measure the overhead of Xen without the driver VM scheme, the whole application is run inside the single privileged driver domain.
2. *Xen guest VM + privileged VM.* In the more natural setting, the application and virtual device driver run in the guest VM (guest domain), and the physical device driver runs in the privileged driver VM (driver domain).

Clearly, a single NIC is a bottleneck. Xen driver VM peaks at 1.9 Gbits/sec with 2 NICs while native Linux peaks at 2.5 Gbits/sec with 3 NICs. For guest VMs, the peak receive rate drops under 0.9 Gbits/sec.

After removing the NIC bottleneck, a different Web server workload showed that driver VM Xen achieves less than 80% of the throughput of native Linux, while guest VM + driver VM drops to 34%.

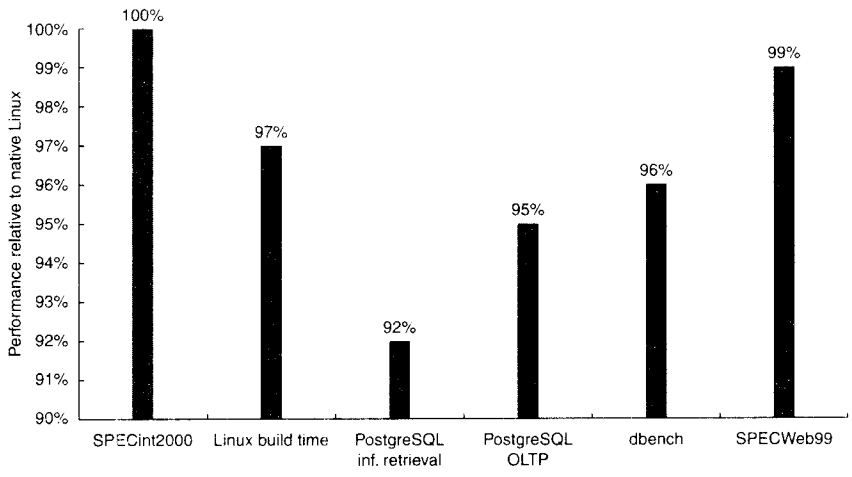

Figure 5.15 Relative performance for Xen versus native Linux. The experiments were performed on a Dell 2650 dual processor 2.4 GHz Xeon server with 2 GB RAM, one Broadcom Tigon 3 Gigabit Ethernet NIC, a single Hitachi DK32EJ 146 GB 10K RPM SCSI disk, and running Linux version 2.4.21 [Barham et al. 2003; Clark et al. 2004].

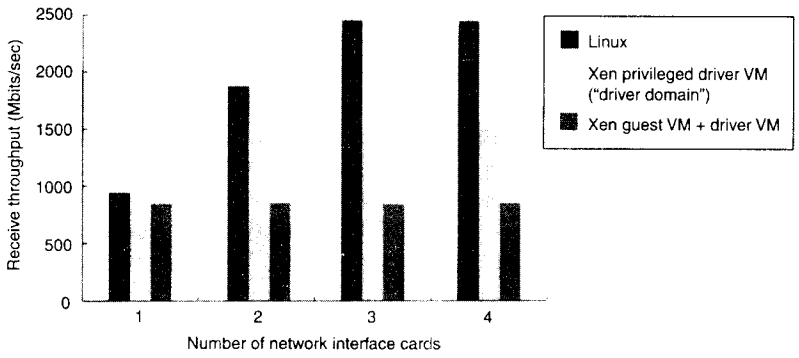

Figure 5.16 TCP receive performance in Mbits/sec for native Linux versus two configurations of Xen. Guest VM + driver VM is the conventional configuration [Menon et al. 2005]. The experiments were performed on a Dell PowerEdge 1600SC running a 2.4 GHz Xeon server with 1 GB RAM, and four Intel Pro-1000 Gigabit Ethernet NIC, running Linux version 2.6.10 and Xen version 2.0.3.

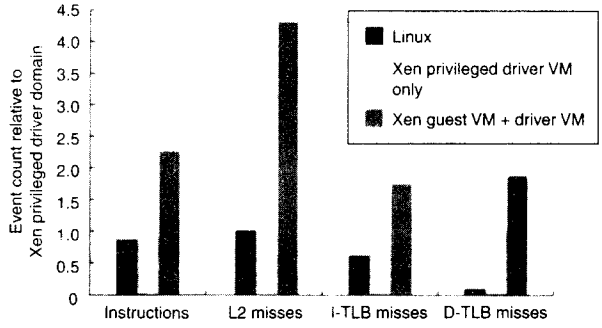

Figure 5.17 Relative change in instructions executed, L2 cache misses, and I-TLB and D-TLB misses of native Linux versus two configurations of Xen for a Web workload [Menon et al. 2005]. Higher L2 and TLB misses come from the lack of support in Xen for superpages, globally marked PTEs, and gather DMA [Menon 2006].

Figure 5.17 explains this drop in performance by plotting the relative change in instructions executed, L2 cache misses, and instruction and data TLB misses for native Linux and the two Xen configurations. Data TLB misses per instruction are 12–24 times higher for Xen than for native Linux, and this is the primary reason for the slowdown for the privileged driver VM configuration. The higher TLB misses are because of two optimizations that Linux uses that Xen does not: superpages and marking page table entries as global. Linux uses superpages for part of its kernel space, and using 4 MB pages obviously lowers TLB misses versus using 1024 4 KB pages. PTEs marked global are not flushed on a context switch, and Linux uses them for its kernel space.

In addition to higher D-TLB misses, the more natural guest VM + driver VM configuration executes more than twice as many instructions. The increase is due to page remapping and page transfer between the driver and guest VMs and due to communication between the two VMs over a channel. This is also the reason for the lower receive performance of guest VMs in Figure 5.16. In addition, the guest VM configuration has more than four times as many L2 caches misses. The reason is Linux uses a zero-copy network interface that depends on the ability of the NIC to do DMA from different locations in memory. Since Xen does not support "gather DMA" in its virtual network interface, it can't do true zero-copy in the guest VM, resulting in more L2 cache misses.

While future versions of Xen may be able to incorporate support for superpages, globally marked PTEs, and gather DMA, the higher instruction overhead looks to be inherent in the split between guest VM and driver VM.

5.5 Crosscutting Issues: The Design of Memory Hierarchies

This section describes three topics discussed in other chapters that are fundamental to memory hierarchies.

Protection and Instruction Set Architecture

Protection is a joint effort of architecture and operating systems, but architects had to modify some awkward details of existing instruction set architectures when virtual memory became popular. For example, to support virtual memory in the IBM 370, architects had to change the successful IBM 360 instruction set architecture that had been announced just six years before. Similar adjustments are being made today to accommodate virtual machines.

For example, the 80x86 instruction POPF loads the flag registers from the top of the stack in memory. One of the flags is the Interrupt Enable (IE) flag. If you run the POPF instruction in user mode, rather than trap it simply changes all the flags except IE. In system mode, it does change the IE. Since a guest OS runs in user mode inside a VM, this is a problem, as it expects to see a changed IE.

Historically, IBM mainframe hardware and VMM took three steps to improve performance of virtual machines:

1. Reduce the cost of processor virtualization
2. Reduce interrupt overhead cost due to the virtualization
3. Reduce interrupt cost by steering interrupts to the proper VM without invoking VMM

IBM is still the gold standard of virtual machine technology. For example, an IBM mainframe ran thousands of Linux VMs in 2000, while Xen ran 25 VMs in 2004 [Clark et al. 2004].

In 2006, new proposals by AMD and Intel try to address the first point, reducing the cost of processor virtualization (see Section 5.7). It will be interesting how many generations of architecture and VMM modifications it will take to address all three points, and how long before virtual machines of the 21st century will be as efficient as the IBM mainframes and VMMs of the 1970s.

Speculative Execution and the Memory System

Inherent in processors 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 protection exceptions were taken, but 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 because again unnecessary stalls could overwhelm the benefits of speculation. Hence, these processors must be matched with nonblocking caches.

In reality, the penalty of an L2 miss is so large that compilers normally only speculate on L1 misses. Figure 5.5 on page 297 shows that for some well-behaved scientific programs the compiler can sustain multiple outstanding L2 misses to cut the L2 miss penalty effectively. Once again, for this to work, the memory system behind the cache must match the goals of the compiler in number of simultaneous memory accesses.

I/O and Consistency of Cached Data

Data can be found in memory and in the cache. As long as one processor is the sole device changing or reading the data and the cache stands between the processor and memory, there is little danger in the processor seeing the old or *stale* copy. As mentioned in Chapter 4, multiple processors and I/O devices raise the opportunity for copies to be inconsistent and to read the wrong copy.

The frequency of the cache coherency problem is different for multiprocessors than I/O. Multiple data copies are a rare event for I/O—one to be avoided whenever possible—but 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 *I/O cache coherency* 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 processor see the same data. The difficulty in this approach is that it interferes with the processor and can cause the processor 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.

The goal for the I/O system in a computer with a cache is to prevent the stale-data problem while interfering 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 is usually found today only in first-level data caches backed by an L2 cache that uses write back.

Input requires some extra work. The software solution is to guarantee that no blocks of the input buffer are in the cache. A page containing the buffer can be marked as noncachable, and the operating system can always input to such a page. Alternatively, the operating system can flush the buffer addresses from the cache before the input occurs. A hardware solution is to check the I/O addresses on input to see if they are in the cache. 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.

5.6 Putting It All Together: AMD Opteron Memory Hierarchy

This section unveils the AMD Opteron memory hierarchy and shows the performance of its components for the SPEC2000 programs. The Opteron is an out-of-order execution processor that fetches up to three 80x86 instructions per clock cycle, translates them into RISC-like operations, issues three of them per clock cycle, and it has 11 parallel execution units. In 2006, the 12-stage integer pipeline yields a maximum clock rate of 2.8 GHz, and the fastest memory supported is PC3200 DDR SDRAM. It uses 48-bit virtual addresses and 40-bit physical addresses. Figure 5.18 shows the mapping of the address through the multiple levels of data caches and TLBs, similar to the format of Figure 5.3 on page 292.

We are now ready to follow the memory hierarchy in action: Figure 5.19 is labeled with the steps of this narrative. First, the PC is sent to the instruction cache. It is 64 KB, two-way set associative with a 64-byte block size and LRU replacement. The cache index is

$$2^{\text{Index}} = \frac{\text{Cache size}}{\text{Block size} \times \text{Set associativity}} = \frac{64\text{K}}{64 \times 2} = 512 = 2^9$$

or 9 bits. It is virtually indexed and *physically* tagged. Thus, the page frame of the instruction's data address is sent to the instruction TLB (step 1) at the same time the 9-bit index (plus an additional 2 bits to select the appropriate 16 bytes) from the virtual address is sent to the data cache (step 2). The fully associative TLB simultaneously searches all 40 entries to find a match between the address and a valid PTE (steps 3 and 4). In addition to translating the address, the TLB checks to see if the PTE demands that this access result in an exception due to an access violation.

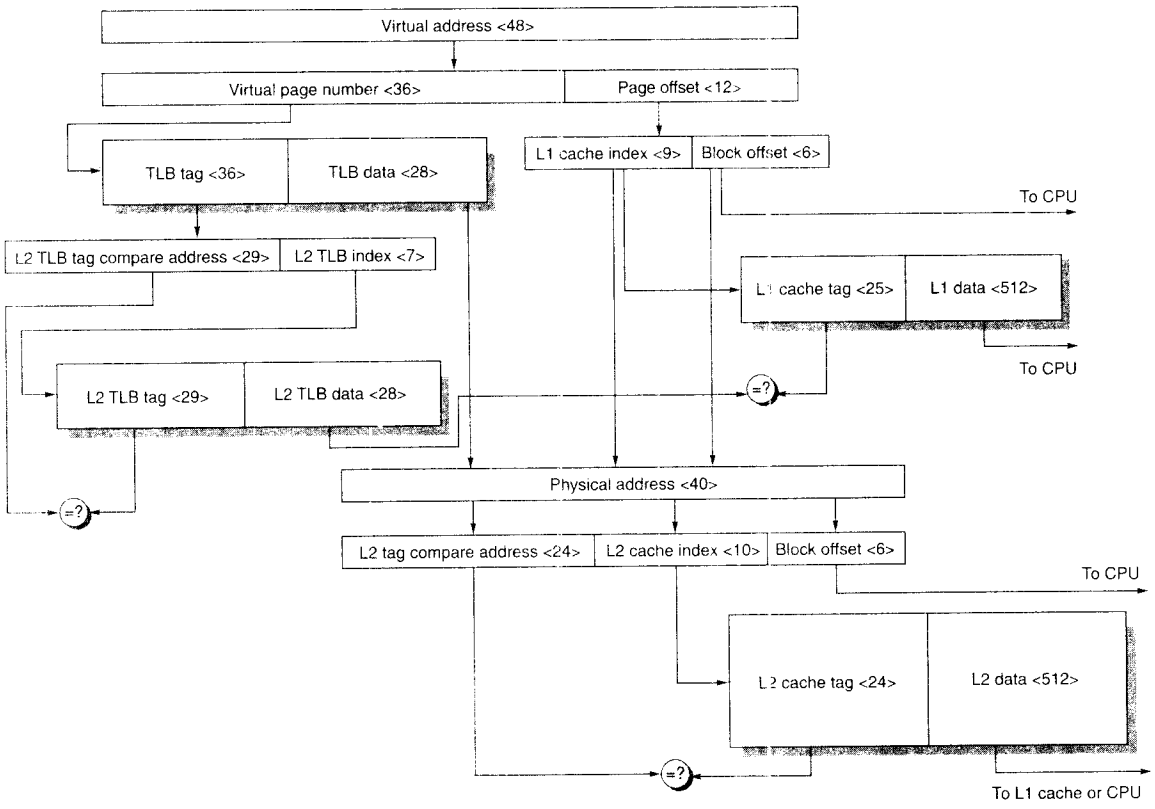

Figure 5.18 The virtual address, physical address, indexes, tags, and data blocks for the AMD Opteron caches and TLBs. Since the instruction and data hierarchies are symmetric, we only show one. The L1 TLB is fully associative with 40 entries. The L2 TLB is 4-way set associative with 512 entries. The L1 cache is 2-way set associative with 64-byte blocks and 64 KB capacity. The L2 cache is 16-way set associative with 64-byte blocks and 1 MB capacity. This figure doesn't show the valid bits and protection bits for the caches and TLBs, as does Figure 5.19.

An I TLB miss first goes to the L2 I TLB, which contains 512 PTEs of 4 KB page sizes and is four-way set associative. It takes 2 clock cycles to load the L1 TLB from the L2 TLB. The traditional 80x86 TLB scheme flushes all TLBs if the page directory pointer register is changed. In contrast, Opteron checks for changes to the actual page directory in memory and flushes only when the data structure is changed, thereby avoiding some flushes.

In the worst case, the page is not in memory, and the operating system gets the page from disk. Since millions of instructions could execute during a page fault, the operating system will swap in another process if one is waiting to run. Otherwise, if there is no TLB exception, the instruction cache access continues.

The index field of the address is sent to both groups of the two-way set-associative data cache (step 5). The instruction cache tag is 40 – 9 bits (index) – 6 bits (block offset) or 25 bits. The four tags and valid bits are compared to the

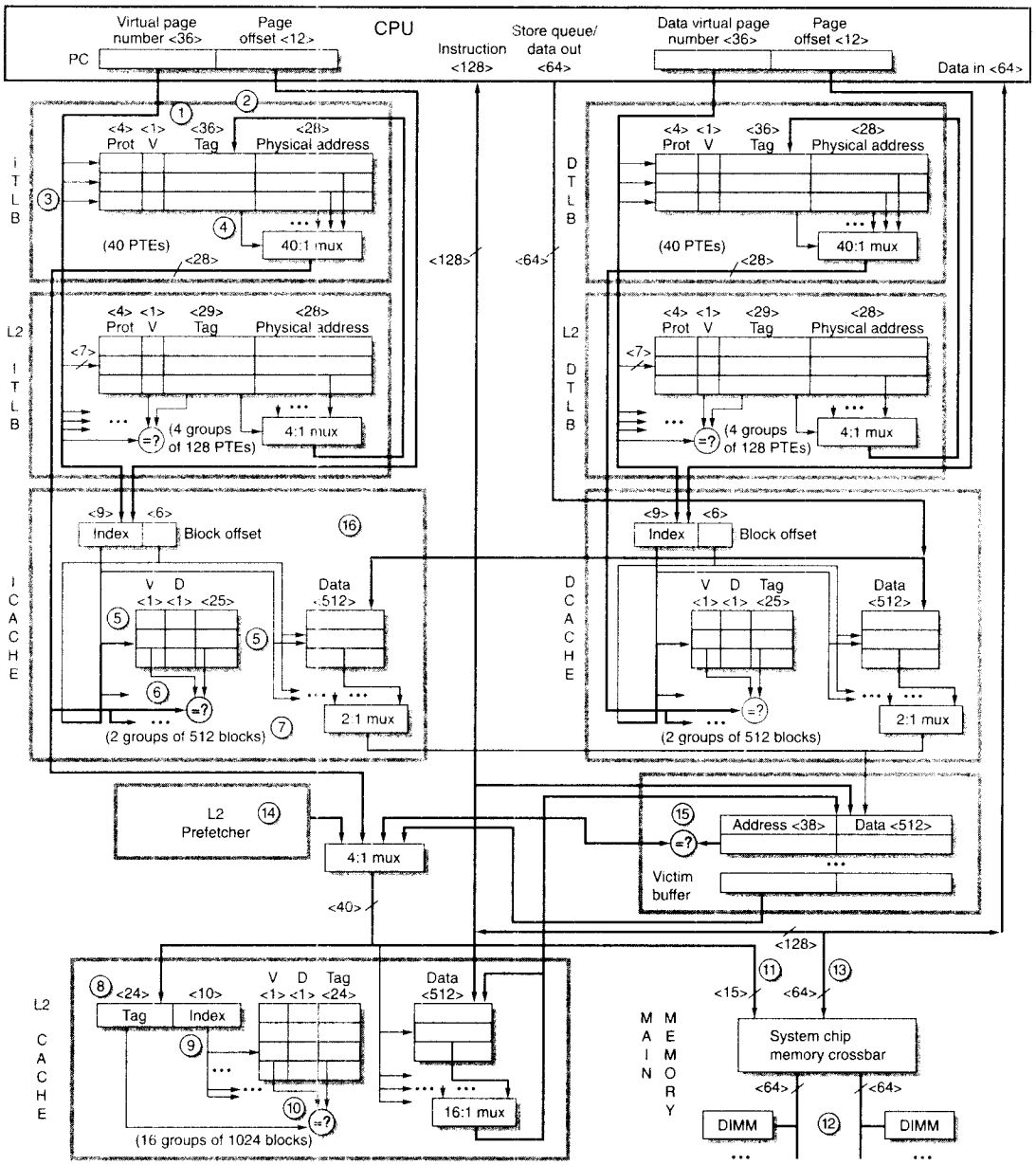

Figure 5.19 The AMD Opteron memory hierarchy. The L1 caches are both 64 KB, 2-way set associative with 64-byte blocks and LRU replacement. The L2 cache is 1 MB, 16-way set associative with 64-byte blocks, and pseudo LRU replacement. The data and L2 caches use write back with write allocation. The L1 instruction and data caches are virtually indexed and physically tagged, so every address must be sent to the instruction or data TLB at the same time as it is sent to a cache. Both TLBs are fully associative and have 40 entries, with 32 entries for 4 KB pages and 8 for 2 MB or 4 MB pages. Each TLB has a 4-way set associative L2 TLB behind it, with 512 entities of 4 KB page sizes. Opteron supports 48-bit virtual addresses and 40-bit physical addresses.

physical page frame from the Instruction TLB (step 6). As the Opteron expects 16 bytes each instruction fetch, an additional 2 bits are used from the 6-bit block offset to select the appropriate 16 bytes. Hence, 9 + 2 or 11 bits are used to send 16 bytes of instructions to the processor. The L1 cache is pipelined, and the latency of a hit is 2 clock cycles. A miss goes to the second-level cache *and* to the memory controller, to lower the miss penalty in case the L2 cache misses.

As mentioned earlier, the instruction cache is 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 instruction cache tags are examined for synonyms in parallel with the L2 cache tags during an L2 lookup. As the minimum page size is 4 KB or 12 bits and the cache index plus block offset is 15 bits, the cache must check 2^3 or 8 blocks per way for synonyms. Opteron uses the redundant snooping tags to check all synonyms in 1 clock cycle. If it finds a synonym, the offending block is invalidated and refetched from memory. This guarantees that a cache block can reside in only one of the 16 possible data cache locations at any given time.

The second-level cache tries to fetch the block on a miss. The L2 cache is 1 MB, 16-way set associative with 64-byte blocks. It uses a pseudo-LRU scheme by managing eight pairs of blocks LRU, and then randomly picking one of the LRU pair on a replacement. The L2 index is

$$2^{\text{Index}} = \frac{\text{Cache size}}{\text{Block size} \times \text{Set associativity}} = \frac{1024\text{K}}{64 \times 16} = 1024 = 2^{10}$$

so the 34-bit block address (40-bit physical address – 6-bit block offset) is divided into a 24-bit tag and a 10-bit index (step 8). Once again, the index and tag are sent to all 16 groups of the 16-way set associative data cache (step 9), which are compared in parallel. If one matches and is valid (step 10), it returns the block in sequential order, 8 bytes per clock cycle. The L2 cache also cancels the memory request that the L1 cache sent to the controller. An L1 instruction cache miss that hits in the L2 cache costs 7 processor clock cycles for the first word.

The Opteron has an exclusion policy between the L1 caches and the L2 cache to try to better utilize the resources, which means a block is in L1 or L2 caches but not in both. Hence, it does not simply place a copy of the block in the L2 cache. Instead, the only copy of the new block is placed in the L1 cache. The old L1 block is sent to the L2 cache. If a block knocked out of the L2 cache is dirty, it is sent to the write buffer, called the victim buffer in the Opteron.

In the last chapter, we showed how inclusion allows all coherency traffic to affect only the L2 cache and not the L1 caches. Exclusion means coherency traffic must check both. To reduce interference between coherency traffic and the processor for the L1 caches, the Opteron has a duplicate set of address tags for coherency snooping.

If the instruction is not found in the secondary cache, the on-chip memory controller must get the block from main memory. The Opteron has dual 64-bit memory channels that can act as one 128-bit channel, since there is only one memory controller and the same address is sent on both channels (step 11). Wide

transfers happen when both channels have identical DIMMs. Each channel supports up to four DDR DIMMs (step 12).

Since the Opteron provides single-error correction/double-error detection checking on data cache, L2 cache, buses, and main memory, the data buses actually include an additional 8 bits for ECC for every 64 bits of data. To reduce the chances of a second error, the Opteron uses idle cycles to remove single-bit errors by reading and rewriting damaged blocks in the data cache, L2 cache, and memory. Since the instruction cache and TLBs are read-only structures, they are protected by parity, and reread from lower levels if a parity error occurs.

The total latency of the instruction miss that is serviced by main memory is approximately 20 processor cycles plus the DRAM latency for the critical instructions. For a PC3200 DDR SDRAM and 2.8 GHz CPU, the DRAM latency is 140 processor cycles (50 ns) to the first 16 bytes. The memory controller fills the remainder of the 64-byte cache block at a rate of 16 bytes per memory clock cycle. With 200 MHz DDR DRAM, that is three more clock edges and an extra 7.5 ns latency, or 21 more processor cycles with a 2.8 GHz processor (step 13).

Opteron has a prefetch engine associated with the L2 cache (step 14). It looks at patterns for L2 misses to consecutive blocks, either ascending or descending, and then prefetches the next line into the L2 cache.

Since the second-level cache is a write-back cache, any miss can lead to an old block being written back to memory. The Opteron places this "victim" block into a victim buffer (step 15), as it does with a victim dirty block in the data cache. The buffer allows the original instruction fetch read that missed to proceed first. The Opteron sends the address of the victim out the system address bus following the address of the new request. The system chip set later extracts the victim data and writes it to the memory DIMMs.

The victim buffer is size eight, so many victims can be queued before being written back either to L2 or to memory. The memory controller can manage up to 10 simultaneous cache block misses—8 from the data cache and 2 from the instruction cache—allowing it to hit under 10 misses, as described in Appendix C. The data cache and L2 cache check the victim buffer for the missing block, but it stalls until the data is written to memory and then refetched. The new data are loaded into the instruction cache as soon as they arrive (step 16). Once again, because of the exclusion property, the missing block is not loaded into the L2 cache.

If this initial instruction is a load, the data address is sent to the data cache and data TLBs, acting very much like an instruction cache access since the instruction and data caches and TLBs are symmetric. One difference is that the data cache has two banks so that it can support two loads or stores simultaneously, as long as they address different banks. In addition, a write-back victim can be produced on a data cache miss. The victim data are extracted from the data cache simultaneously with the fill of the data cache with the L2 data and sent to the victim buffer.

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, since the policy is to allocate on writes. The store does not update the cache until later, after it is known to be nonspeculative. During this time the store resides in a load-store queue, part of the out-of-order control mechanism of the processor. It can hold up to 44 entries and supports speculative forwarding results to the execution unit. The data cache is ECC protected, so a read-modify-write operation is required to update the data cache on stores. This is accomplished by assembling the full block in the load/store queue and always writing the entire block.

Performance of the Opteron Memory Hierarchy

How well does the Opteron work? The bottom line in this evaluation is the percentage of time lost while the processor 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 processor like the Opteron, 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.

Figure 5.20 shows the CPI and various misses per 1000 instructions for a benchmark similar to TPC-C on a database and the SPEC2000 programs. Clearly, most of the SPEC2000 programs do not tax the Opteron memory hierarchy, with mcf being the exception. (SPEC nicknamed it the "cache buster" because of its memory footprint size and its access patterns.) The average SPEC I cache misses per instruction is 0.01% to 0.09%, the average D cache misses per instruction are 1.34% to 1.43%, and the average L2 cache misses per instruction are 0.23% to 0.36%. The commercial benchmark does exercise the memory hierarchy more, with misses per instruction of 1.83%, 1.39%, and 0.62%, respectively.

How do the real CPIs of Opteron compare to the peak rate of 0.33, or 3 instructions per clock cycle? The Opteron completes on average 0.8–0.9 instructions per clock cycle for SPEC2000, with an average CPI of 1.15–1.30. For the database benchmark, the higher miss rates for caches and TLBs yields a CPI of 2.57, or 0.4 instructions per clock cycle. This factor of 2 slowdown in CPI for TPC-C-like benchmarks suggests that microprocessors designed in servers see heavier demands on the memory systems than do microprocessors for desktops. Figure 5.21 estimates the breakdown between the base CPI of 0.33 and the stalls for memory and for the pipeline.

Figure 5.21 assumes none of the memory hierarchy misses are overlapped with the execution pipeline or with each other, so the pipeline stall portion is a lower bound. Using this calculation, the CPI above the base that is attributable to memory averages about 50% for the integer programs (from 1% for eon to 100% for vpr) and about 60% for the floating-point programs (from 12% for sixtrack to 98% for applu). Going deeper into the numbers, about 50% of the memory CPI (25% overall) is due to L2 cache misses for the integer programs and L2 represents about 70% of the memory CPI for the floating-point programs (40% overall). As mentioned earlier, L2 misses are so long that it is difficult to hide them with extra work.

		Misses per 1000 instructions						
Benchmark	Avg CPI	Icache	Dcache	L2	ITLB L1	DTLB L1	ITLB L2	DTLB L2
TPC-C-like	2.57	18.34	13.89	6.18	3.25	9.00	0.09	1.71
SPECint2000 total	1.30	0.90	14.27	3.57	0.25	12.47	0.00	1.06
164.gzip	0.86	0.01	16.03	0.10	0.01	11.06	0.00	0.09
175.vpr	1.78	0.02	23.36	5.73	0.01	50.52	0.00	3.22
176.gcc	1.02	1.94	19.04	0.90	0.79	4.53	0.00	0.19
181.mcf	13.06	0.02	148.90	103.82	0.01	50.49	0.00	26.98
186.crafty	0.72	3.15	4.05	0.06	0.16	18.07	0.00	0.01
197.parser	1.28	0.08	14.80	1.34	0.01	11.56	0.00	0.65
252.eon	0.82	0.06	0.45	0.00	0.01	0.05	0.00	0.00
253.perlbmk	0.70	1.36	2.41	0.43	0.93	3.51	0.00	0.31
254.gap	0.86	0.76	4.27	0.58	0.05	3.38	0.00	0.33
255.vortex	0.88	3.67	5.86	1.17	0.68	15.78	0.00	1.38
256.bzip2	1.00	0.01	10.57	2.94	0.00	8.17	0.00	0.63
300.twolf	1.85	0.08	26.18	4.49	0.02	14.79	0.00	0.01
SPECfp2000 total	1.15	0.08	13.43	2.26	0.01	3.70	0.00	0.79
168.wupwise	0.83	0.00	6.56	1.66	0.00	0.22	0.00	0.17
171.swim	1.88	0.01	30.87	2.02	0.00	0.59	0.00	0.41
172.mgrid	0.89	0.01	16.54	1.35	0.00	0.35	0.00	0.25
173.applu	0.97	0.01	8.48	3.41	0.00	2.42	0.00	0.13
177.mesa	0.78	0.03	1.58	0.13	0.01	8.78	0.00	0.17
178.galgel	1.07	0.01	18.63	2.38	0.00	7.62	0.00	0.67
179.art	3.03	0.00	56.96	8.27	0.00	1.20	0.00	0.41
183.equake	2.35	0.06	37.29	3.30	0.00	1.20	0.00	0.59
187.facerec	1.07	0.01	9.31	3.94	0.00	1.21	0.00	0.20
188.ammp	1.19	0.02	16.58	2.37	0.00	8.61	0.00	3.25
189.lucas	1.73	0.00	17.35	4.36	0.00	4.80	0.00	3.27
191.fma3d	1.34	0.20	11.84	3.02	0.05	0.36	0.00	0.21
200.sixtrack	0.63	0.03	0.53	0.16	0.01	0.66	0.00	0.01
301.apsi	1.17	0.50	13.81	2.48	0.01	10.37	0.00	1.69

Figure 5.20 CPI and misses per 1000 instructions for running a TPC-C-like database workload and the SPEC2000 benchmarks on the AMD Opteron. Since the Opteron uses an out-of-order instruction execution, the statistics are calculated as the number of misses per 1000 instructions successfully committed.

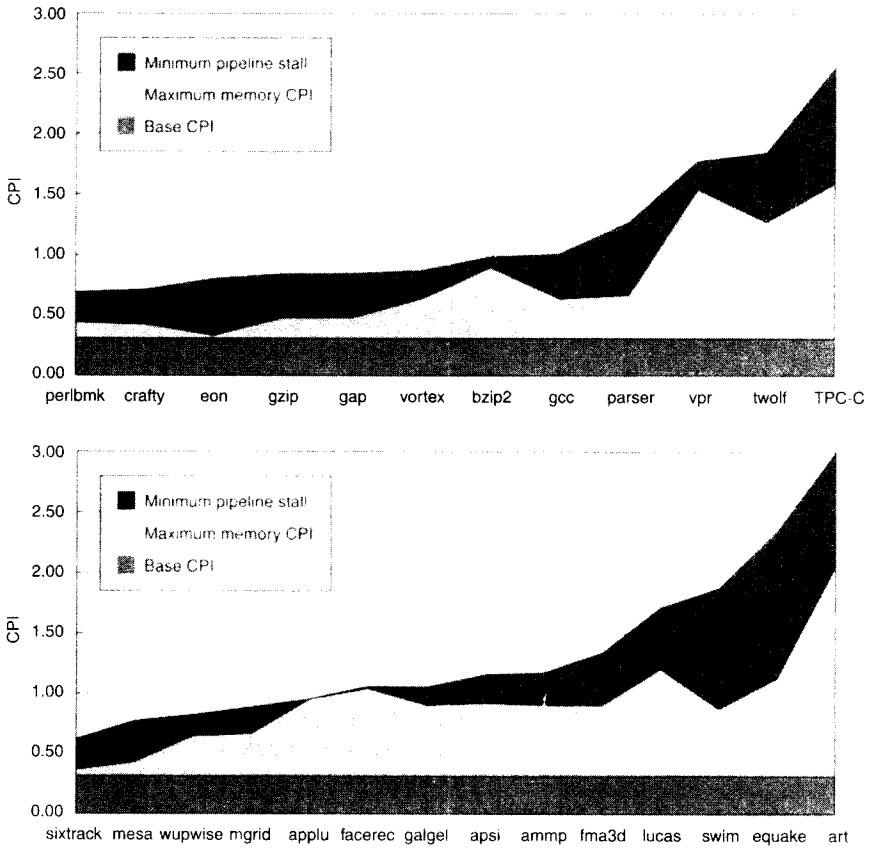

Figure 5.21 Area plots that estimate CPI breakdown into base CPI, memory stalls, and pipeline stalls for SPECint2000 programs (plus a TPC-C-like benchmark) on the top and SPECfp2000 on the bottom. They are sorted from lowest to highest overall CPI. We estimated the memory CPI by multiplying the misses per instruction at the various levels by their miss penalties, and subtracted it and the base CPI from the measured CPI to calculate the pipeline stall CPI. The L2 miss penalty is 140 clock cycles, and all other misses hit in the L2 cache. This estimate assumes no overlapping of memory and execution, so the memory portion is high, as some of it is surely overlapped with pipeline stalls and with other memory accesses. Since it would overwhelm the rest of the data with its CPI of 13.06, mcf is not included. Memory misses must be overlapped in mcf; otherwise the CPI would grow to 18.53.

Finally, Figure 5.22 compares the miss rates of the data caches and the L2 caches of Opteron to the Intel Pentium 4, showing the ratio of the misses per instruction for 10 SPEC2000 benchmarks. Although they are executing the same programs compiled for the same instruction set, the compilers and resulting code

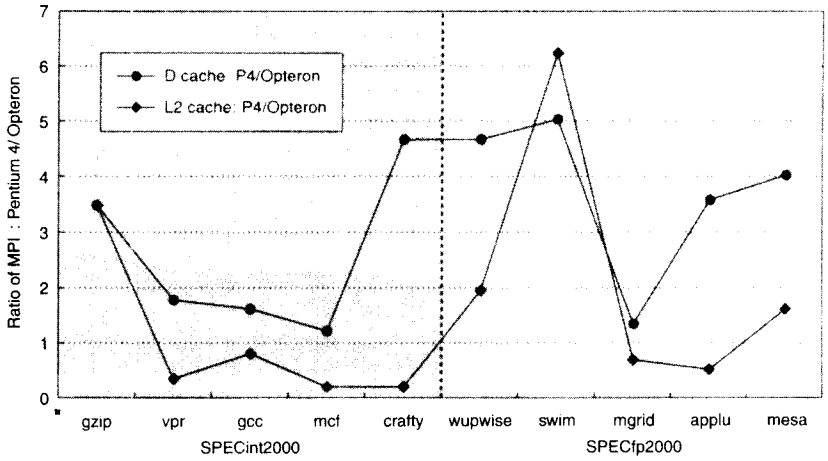

Figure 5.22 Ratio of misses per instruction for Pentium 4 versus Opteron. Bigger means a higher miss rate for Pentium 4. The 10 programs are the first 5 SPECint2000 and the first 5 SPECfp2000. (The two processors and their memory hierarchies are described in the table in the text.) The geometric mean of the ratio of performance of the 5 SPECint programs on the two processors is 1.00 with a standard deviation of 1.42; the geometric mean of the performance of the 5 SPECfp programs suggests Opteron is 1.15 times faster, with a standard deviation of 1.25. Note the clock rate for the Pentium 4 was 3.2 GHz in these experiments; higher-clock-rate Pentium 4s were available but not used in this experiment. Figure 5.10 shows that half of these programs benefit significantly from the prefetching hardware of the Pentium 4: mcf, wupwise, swim, mgrid, and applu.

sequences are different as are the memory hierarchies. The following table summarizes the two memory hierarchies:

Processor	Pentium 4 (3.2 GHz)	Opteron (2.8 GHz)
Data cache	8-way associative, 16 KB, 64-byte block	2-way associative, 64 KB, 64-byte block
L2 cache	8-way associative, 2 MB, 128-byte block, inclusive of D cache	16-way associative, 1 MB, 64-byte block, exclusive of D cache
Prefetch	8 streams to L2	1 stream to L2

Although the Pentium 4 has much higher associativity, the four times larger data cache of Opteron has lower L1 miss rates. The geometric mean of the ratios of L1 miss rates is 2.25 and geometric standard deviation is 1.75 for the five SPECint2000 programs; they are 3.37 and 1.72 for the five SPECfp2000 programs (see Chapter 1 to review geometric means and standard deviations).

With twice the L2 block size and L2 cache capacity and more aggressive prefetching, the Pentium 4 usually has fewer L2 misses per instruction. Surprisingly, the Opteron L2 cache has fewer on 4 of the 10 programs. This variability is reflected in the means and high standard deviations: the geometric mean and standard deviation of the ratios of L2 miss rates is 0.50 and 3.45 for the integer programs and 1.48 and 2.66 for the floating-point programs. As mentioned earlier, this nonintuitive result could simply be the consequence of using different compilers and optimizations. Another possible explanation is that the lower memory latency and higher memory bandwidth of the Opteron helps the effectiveness of its hardware prefetching, which is known to reduce misses on many of these floating-point programs. (See Figure 5.10 on page 306.)

5.7 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 we were limited here not by lack of warnings, but by lack of space!

Fallacy *Predicting cache performance of one program from another.*

Figure 5.23 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. Commercial programs such as databases will have significant miss rates even in large second-level caches, which is generally not the case for the SPEC programs. Clearly, generalizing cache performance from one program to another is unwise.

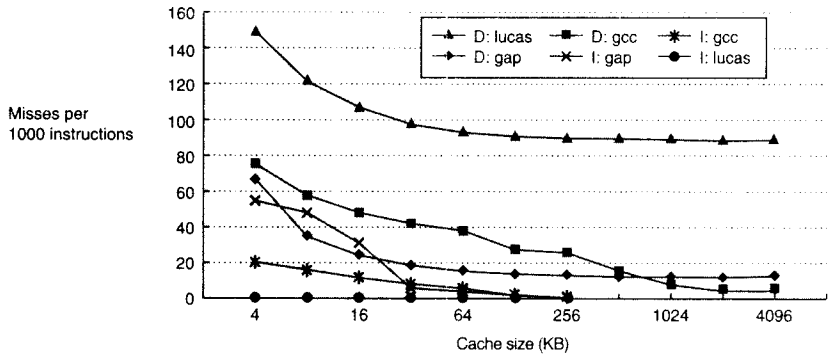

Figure 5.23 Instruction and data misses per 1000 instructions as cache size varies from 4 KB to 4096 KB. Instruction misses for gcc are 30,000 to 40,000 times larger than lucas, and conversely, data misses for lucas are 2 to 60 times larger than gcc. The programs gap, gcc, and lucas are from the SPEC2000 benchmark suite.

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 5.24 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 the 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 a 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 minute:

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

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

Pitfall *Overemphasizing memory bandwidth in DRAMs.*

Several years ago, a startup named RAMBUS innovated on the DRAM interface. Its product, Direct RDRAM, offered up to 1.6 GB/sec of bandwidth from a single DRAM. When announced, the peak bandwidth was eight times faster than individual conventional SDRAM chips. Figure 5.25 compares prices of various versions of DRAM and RDRAM, in memory modules and in systems.

PCs do most memory accesses through a two-level cache hierarchy, so it was unclear how much benefit is gained from high bandwidth without also improving memory latency. According to Pabst [2000], when comparing PCs with 400 MHz DRDRAM to PCs with 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 an 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.

One measure of the RDRAM cost is die size; it had about a 20% larger die for the same capacity compared to SDRAM. DRAM designers use redundant rows and columns to improve yield significantly 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. Using the evaluation in Figure 5.25, in 2000 the price was about a factor of 2–3 higher for RDRAM. In 2006, the ratio is not less.

RDRAM was at its strongest in small memory systems that need high bandwidth, such as a Sony Playstation.

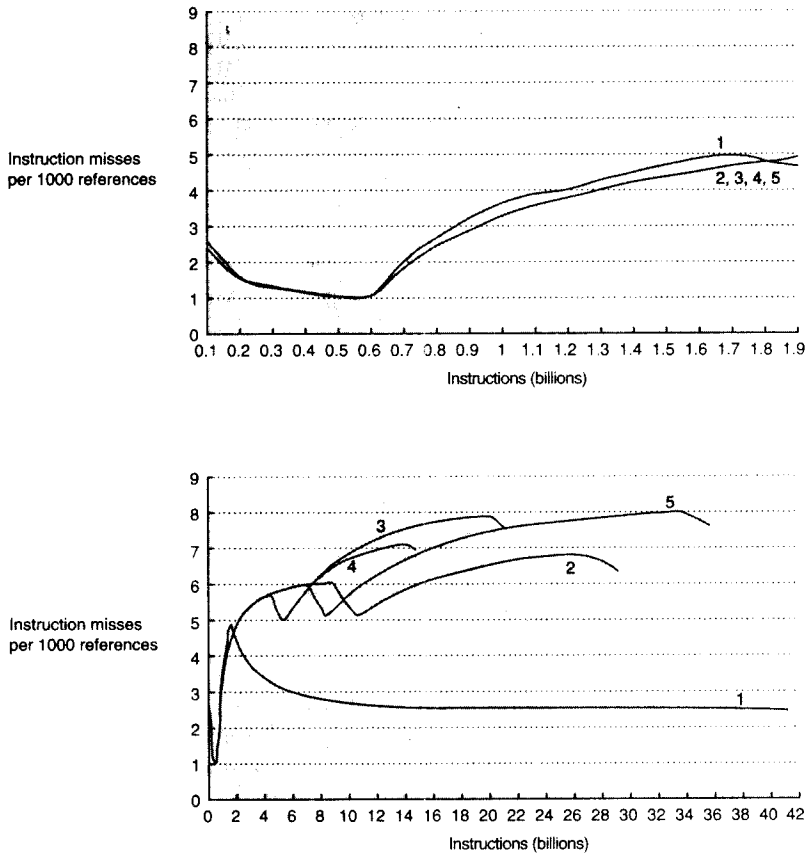

Figure 5.24 Instruction misses per 1000 references for five inputs to perl benchmark from SPEC2000. There is little variation in misses and little difference between the five inputs for the first 1.9 billion instructions. Running to completion shows how misses vary over the life of the program and how 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 completion, which takes 16–41 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 two-way 64 KB with LRU, and a unified 1 MB direct-mapped L2 cache.

Pitfall *Not 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 must go to main memory. The architect must

	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	$1519	$2139	$620	$1559	$2269	$710
DRDRAM PC700	$725	$826	$1689	$3009	$1320	$1789	$3409	$1620
Price ratio DRDRAM/SDRAM	4.1	3.2	1.1	1.4	2.1	1.1	1.5	2.3

Figure 5.25 Comparison of price of SDRAM versus DRDRAM in memory modules and in systems in 2000. DRDRAM memory modules cost about a factor of 4 more without ECC and 3 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 CD-ROM, 17-inch monitor, and Microsoft Windows 95/98 and Office 98. The module prices were the lowest found at pricewatch.com in June 2000. By September 2005, 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 2005 Dell did not offer systems whose only difference was type of DRAMs; hence, we stick with the comparison from 2000.)

design a high bandwidth memory behind the cache for such applications. As an extreme example, the NEC SX7 offers up to 16,384 interleaved SDRAM memory banks. It is a vector computer that doesn't rely on data caches for memory performance (see Appendix F). Figure 5.26 shows the top 10 results from the Stream benchmark as of 2005, which measures bandwidth to copy data [McCalpin 2005]. Not surprisingly, the NEC SX7 has the top ranking.

Only four computers rely on data caches for memory performance, and their memory bandwidth is a factor of 7–25 slower than the NEC SX7.

Pitfall *Implementing a virtual machine monitor on an instruction set architecture that wasn't designed to be virtualizable.*

Many architects in the 1970s and 1980s weren't careful to make sure that all instructions reading or writing information related to hardware resource information were privileged. This laissez faire attitude causes problems for VMMs for all of these architectures, including the 80x86, which we use here as an example.

Figure 5.27 describes the 18 instructions that cause problems for virtualization [Robin and Irvine 2000]. The two broad classes are instructions that

- read control registers in user mode that reveals that the guest operating system is running in a virtual machine (such as POPF mentioned earlier), and
- check protection as required by the segmented architecture but assume that the operating system is running at the highest privilege level.

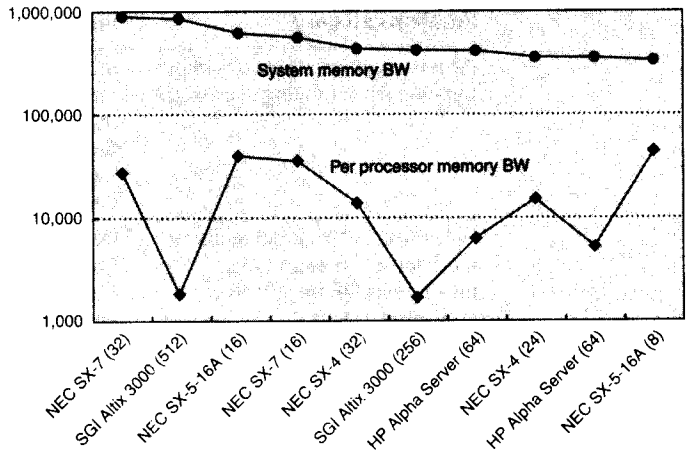

Figure 5.26 Top 10 in memory bandwidth as measured by the untuned copy portion of the stream benchmark [McCalpin 2005]. The number of processors is shown in parentheses. Two are cache-based clusters (SGI), two are cache-based SMPs (HP), but most are NEC vector processors of different generations and number of processors. Systems use between 8 and 512 processors to achieve higher memory bandwidth. System bandwidth is bandwidth of all processors collectively. Processor bandwidth is simply system bandwidth divided by the number of processors. The STREAM benchmark is a simple synthetic benchmark program that measures sustainable memory bandwidth (in MB/sec) for simple vector kernels. It specifically works with data sets much larger than the available cache on any given system.

Virtual memory is also challenging. Because the 80x86 TLBs do not support process ID tags, as do most RISC architectures, it is more expensive for the VMM and guest OSes to share the TLB: each address space change typically requires a TLB flush.

Virtualizing I/O is also a challenge for the 80x86, in part because it both supports memory-mapped I/O and has separate I/O instructions, but more importantly, because there is a very large number and variety of types of devices and device drivers of PCs for the VMM to handle. Third-party vendors supply their own drivers, and they may not properly virtualize. One solution for conventional VM implementations is to load real device drivers directly into the VMM.

To simplify implementations of VMMs on the 80x86, both AMD and Intel have proposed extensions to the architecture. Intel's VT-x provides a new execution mode for running VMs, an architected definition of the VM state, instructions to swap VMs rapidly, and a large set of parameters to select the circumstances where a VMM must be invoked. Altogether, VT-x adds 11 new instructions for the 80x86. AMD's Pacifica makes similar proposals.

After turning on the mode that enables VT-x support (via the new `VMXON` instruction), VT-x offers four privilege levels for the guest OS that are lower in priority than the original four. VT-x captures all the state of a virtual machine in

Problem category	Problem 80x86 instructions
Access sensitive registers without trapping when running in user mode	Store global descriptor table register (SGDT) Store local descriptor table register (SLDT) Store interrupt descriptor table register (SIDT) Store machine status word (SMSW) Push flags (PUSHF, PUSHFD) Pop flags (POPF, POPFD)
When accessing virtual memory mechanisms in user mode, instructions fail the 80x86 protection checks	Load access rights from segment descriptor (LAR) Load segment limit from segment descriptor (LSL) Verify if segment descriptor is readable (VERR) Verify if segment descriptor is writable (VERW) Pop to segment register (POP CS, POP SS, . . .) Push segment register (PUSH CS, PUSH SS, . . .) Far call to different privilege level (CALL) Far return to different privilege level (RET) Far jump to different privilege level (JMP) Software interrupt (INT) Store segment selector register (STR) Move to/from segment registers (MOVE)

Figure 5.27 Summary of 18 80x86 instructions that cause problems for virtualization [Robin and Irvine 2000]. The first five instructions of the top group allow a program in user mode to read a control register, such as a descriptor table registers, without causing a trap. The pop flags instruction modifies a control register with sensitive information, but fails silently when in user mode. The protection checking of the segmented architecture of the 80x86 is the downfall of the bottom group, as each of these instructions checks the privilege level implicitly as part of instruction execution when reading a control register. The checking assumes that the OS must be at the highest privilege level, which is not the case for guest VMs. Only the MOVE to segment register tries to modify control state, and protection checking foils it as well.

the Virtual Machine Control State (VMCS), and then provides atomic instructions to save and restore a VMCS. In addition to critical state, the VMCS includes configuration information to determine when to invoke the VMM, and then specifically what caused the VMM to be invoked. To reduce the number of times the VMM must be invoked, this mode adds shadow versions of some sensitive registers and adds masks that check to see whether critical bits of a sensitive register will be changed before trapping. To reduce the cost of virtualizing virtual memory, AMD's Pacifica adds an additional level of indirection, called *nested page tables*. It makes shadow page tables unnecessary.

It is ironic that AMD and Intel are proposing a new mode. If operating systems like Linux or Microsoft Windows start using that mode in their kernel, the new mode would cause performance problems for the VMM since it would be about 100 times too slow! Nevertheless, the Xen organization plans to use VT-x to allow it to support Windows as a guest OS.

5.8 Concluding Remarks

Figure 5.28 compares the memory hierarchy of microprocessors aimed at desktop and server 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.

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,

MPU	AMD Opteron	Intel Pentium 4	IBM Power 5	Sun Niagara
Instruction set architecture	80x86 (64b)	80x86	PowerPC	SPARC v9
Intended application	desktop	desktop	server	server
CMOS process (nm)	90	90	130	90
Die size (mm^2)	199	217	389	379
Instructions issued/clock	3	3 RISC ops	8	1
Processors/chip	2	1	2	8
Clock rate (2006)	2.8 GHz	3.6 GHz	2.0 GHz	1.2 GHz
Instruction cache per processor	64 KB, 2-way set associative	12000 RISC op trace cache (~96 KB)	64 KB, 2-way set associative	16 KB, 1-way set associative
Latency L1 I (clocks)	2	4	1	1
Data cache per processor	64 KB, 2-way set associative	16 KB, 8-way set associative	32 KB, 4-way set associative	8 KB, 1-way set associative
Latency L1 D (clocks)	2	2	2	1
TLB entries (I/D/L2 I/L2 D)	40/40/512/512	128/54	1024/1024	64/64
Minimum page size	4 KB	4 KB	4 KB	8 KB
On-chip L2 cache	2 x 1 MB, 16-way set associative	2 MB, 8-way set associative	1.875 MB, 10-way set associative	3 MB, 2-way set associative
L2 banks	2	1	3	4
Latency L2 (clocks)	7	22	13	22 I, 23 D
Off-chip L3 cache	—	—	36 MB, 12-way set associative (tags on chip)	—
Latency L3 (clocks)	—	—	87	—
Block size (L1I/L1D/L2/L3, bytes)	64	64/64/128/—	128/128/128/256	32/16/64/—
Memory bus width (bits)	128	64	64	128
Memory bus clock	200 MHz	200 MHz	400 MHz	400 MHz
Number of memory buses	1	1	4	4

Figure 5.28 Memory hierarchy and chip size of desktop and server microprocessors in 2005.

not in inventing new techniques. The increasingly fast processors 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 few technological "sweet spots" in balancing cost, performance, power, and complexity: Missing a target wastes performance, power, hardware, design time, debug time, or possibly all five. Architects hit a target by careful, quantitative analysis.

5.9 Historical Perspective and References

In Section K.6 on the companion CD we examine the history of caches, virtual memory, and virtual machines. IBM plays a prominent role in the history of all three. References for further reading are included.

Case Studies with Exercises by Norman P. Jouppi

Case Study 1: Optimizing Cache Performance via Simple Hardware

Concepts illustrated by this case study

- Small and Simple Caches
- Way Prediction
- Pipelined Caches
- Banked Caches
- Merging Write Buffers
- Critical Word First and Early Restart
- Nonblocking Caches
- Calculating Impact of Cache Performance on Simple In-Order Processors

Imagine (unrealistically) that you are building a simple in-order processor that has a CPI of 1 for all nondata memory access instructions. In this case study we will consider the performance implications of small and simple caches, way-predicting caches, pipelined cache access, banked caches, merging write buffers, and critical word first and early restart. Figure 5.29 shows SPEC2000 data miss ratios (misses per 1000 instructions) with the harmonic mean of the full execution of all benchmarks.

CACTI is a tool for estimating the access and cycle time, dynamic and leakage power, and area of a cache based on its lithographic feature size and cache organization. Of course there are many different possible circuit designs for a

D-cache misses/inst: 2,521,022,899,870 data refs (0.32899--/inst);
1,801,061,937,244 D-cache 64-byte block accesses (0.23289--/inst)

Size	Direct	2-way LRU	4-way LRU	8-way LRU	Full LRU
1 KB	0.0863842--	0.0697167--	0.0634309--	0.0563450--	0.0533706--
2 KB	0.0571524--	0.0423833--	0.0360463--	0.0330364--	0.0305213--
4 KB	0.0370053--	0.0260286--	0.0222981--	0.0202763--	0.0190243--
8 KB	0.0247760--	0.0155691--	0.0129609--	0.0107753--	0.0083886--
16 KB	0.0159470--	0.0085658--	0.0063527--	0.0056438--	0.0050068--
32 KB	0.0110603--	0.0056101--	0.0039190--	0.0034628--	0.0030885--
64 KB	0.0066425--	0.0036625--	0.0009874--	0.0002666--	0.0000106--
128 KB	0.0035823--	0.0002341--	0.0000109--	0.0000058--	0.0000058--
256 KB	0.0026345--	0.0000092--	0.0000049--	0.0000051--	0.0000053--
512 KB	0.0014791--	0.0000065--	0.0000029--	0.0000029--	0.0000029--
1 MB	0.0000090--	0.0000058--	0.0000028--	0.0000028--	0.0000028--

Figure 5.29 SPEC2000 data miss ratios (misses per 1000 instructions) [Cantin and Hill 2003].

given cache organization, and many different technologies for a given lithographic feature size, but CACTI assumes a "generic" organization and technology. Thus it may not be accurate for a specific cache design and technology in absolute terms, but it is fairly accurate at quantifying the relative performance of different cache organizations at different feature sizes. CACTI is available in an online form at *http://quid.hpl.hp.com:9081/cacti/*. Assume all cache misses take 20 cycles.

5.1 [12/12/15/15] <5.2> The following questions investigate the impact of small and simple caches using CACTI, and assume a 90 nm (0.09 μm) technology.

a. [12] <5.2> Compare the access times of 32 KB caches with 64-byte blocks and a single bank. What is the relative access times of two-way and four-way set associative caches in comparison to a direct-mapped organization?

b. [12] <5.2> Compare the access times of two-way set-associative caches with 64-byte blocks and a single bank. What is the relative access times of 32 KB and 64 KB caches in comparison to a 16 KB cache?

c. [15] <5.2> Does the access time for a typical level 1 cache organization increase with size roughly as the capacity in bytes B, the square root of B, or the log of B?

d. [15] <5.2> Find the cache organization with the lowest average memory access time given the miss ratio table in Figure 5.29 and a cache access time budget of 0.90 ns. What is this organization, and does it have the lowest miss rate of all organizations for its capacity?

5.2 [12/15/15/10] <5.2> You are investigating the possible benefits of a way-predicting level 1 cache. Assume that the 32 KB two-way set-associative single-banked level 1

data cache is currently the cycle time limiter. As an alternate cache organization you are considering a way-predicted cache modeled as a 16 KB direct-mapped cache with 85% prediction accuracy. Unless stated otherwise, assume a mispredicted way access that hits in the cache takes one more cycle.

a. [12] <5.2> What is the average memory access time of the current cache versus the way-predicted cache?

b. [15] <5.2> If all other components could operate with the faster way-predicted cache cycle time (including the main memory), what would be the impact on performance from using the way-predicted cache?

c. [15] <5.2> Way-predicted caches have usually only been used for instruction caches that feed an instruction queue or buffer. Imagine you want to try out way prediction on a data cache. Assume you have 85% prediction accuracy, and subsequent operations (e.g., data cache access of other instructions, dependent operations, etc.) are issued assuming a correct way prediction. Thus a way misprediction necessitates a pipe flush and replay trap, which requires 15 cycles. Is the change in average memory access time per load instruction with data cache way prediction positive or negative, and how much is it?

d. [10] <5.2> As an alternative to way prediction, many large associative level 2 caches serialize tag and data access, so that only the required data set array needs to be activated. This saves power but increases the access time. Use CACTI's detailed Web interface for a 0.090 μm process 1 MB four-way set-associative cache with 64-byte blocks, 144 bits read out, 1 bank, only 1 read/write port, and 30-bit tags. What are the ratio of the total dynamic read energies per access and ratio of the access times for serializing tag and data access in comparison to parallel access?

5.3 [10/12/15] <5.2> You have been asked to investigate the relative performance of a banked versus pipelined level 1 data cache for a new microprocessor. Assume a 64 KB two-way set-associative cache with 64 B blocks. The pipelined cache would consist of two pipe stages, similar to the Alpha 21264 data cache. A banked implementation would consist of two 32 KB two-way set-associative banks. Use CACTI and assume a 90 nm (0.09 μm) technology in answering the following questions.

a. [10] <5.2> What is the cycle time of the cache in comparison to its access time, and how many pipe stages will the cache take up (to two decimal places)?

b. [12] <5.2> What is the average memory access time if 20% of the cache access pipe stages are empty due to data dependencies introduced by pipelining the cache and pipelining more finely doubles the miss penalty?

c. [15] <5.2> What is the average memory access time of the banked design if there is a memory access each cycle and a random distribution of bank accesses (with no reordering) and bank conflicts cause a one-cycle delay?

5.4 [12/15] <5.2> Inspired by the usage of critical word first and early restart on level 1 cache misses, consider their use on level 2 cache misses. Assume a 1 MB L2 cache with 64-byte blocks and a refill path that is 16 bytes wide. Assume the L2 can be written with 16 bytes every 4 processor cycles, the time to receive the first 16-byte block from the memory controller is 100 cycles, each additional 16 B from main memory requires 16 cycles and data can be bypassed directly into the read port of the L2 cache. Ignore any cycles to transfer the miss request to the level 2 cache and the requested data to the level 1 cache.

a. [12] <5.2> How many cycles would it take to service a level 2 cache miss with and without critical word first and early restart?

b. [15] <5.2> Do you think critical word first and early restart would be more important for level 1 caches or level 2 caches, and what factors would contribute to their relative importance?

5.5 [10/12] <5.2> You are designing a write buffer between a write-through level 1 cache and a write-back level 2 cache. The level 2 cache write data bus is 16 bytes wide and can perform a write to an independent cache address every 4 processor cycles.

a. [10] <5.2> How many bytes wide should each write buffer entry be?

b. [12] <5.2> What speedup could be expected in the steady state by using a merging write buffer instead of a nonmerging buffer when zeroing memory by the execution of 32-bit stores if all other instructions could be issued in parallel with the stores and the blocks are present in the level 2 cache?

Case Study 2: Optimizing Cache Performance via Advanced Techniques

Concepts illustrated by this case study

- Nonblocking Caches
- Compiler Optimizations for Caches
- Software and Hardware Prefetching
- Calculating Impact of Cache Performance on More Complex Processors

The transpose of a matrix interchanges its rows and columns and is illustrated below:

$$\begin{bmatrix} A11 & A12 & A13 & A14 \\ A21 & A22 & A23 & A24 \\ A31 & A32 & A33 & A34 \\ A41 & A42 & A43 & A44 \end{bmatrix} \Rightarrow \begin{bmatrix} A11 & A21 & A31 & A41 \\ A12 & A22 & A32 & A42 \\ A13 & A23 & A33 & A43 \\ A14 & A24 & A34 & A44 \end{bmatrix}$$

Here is a simple C loop to show the transpose:

```
for (i = 0; i < 3; i++) {
   for (j = 0; j < 3; j++) {
      output[j][i] = input[i][j];
   }
}
```

Assume both the input and output matrices are stored in the row major order (row major order means row index changes fastest). Assume you are executing a 256×256 double-precision transpose on a processor with a 16 KB fully associative (so you don't have to worry about cache conflicts) LRU replacement level 1 data cache with 64-byte blocks. Assume level 1 cache misses or prefetches require 16 cycles, always hit in the level 2 cache, and the level 2 cache can process a request every 2 processor cycles. Assume each iteration of the inner loop above requires 4 cycles if the data is present in the level 1 cache. Assume the cache has a write-allocate fetch-on-write policy for write misses. Unrealistically assume writing back dirty cache blocks require 0 cycles.

5.6 [10/15/15] <5.2> For the simple implementation given above, this execution order would be nonideal for the input matrix. However, applying a loop interchange optimization would create a nonideal order for the output matrix. Because loop interchange is not sufficient to improve its performance, it must be blocked instead.

a. [10] <5.2> What block size should be used to completely fill the data cache with one input and output block?

b. [15] <5.2> How do the relative number of misses of the blocked and unblocked versions compare if the level 1 cache is direct mapped?

c. [15] <5.2> Write code to perform a transpose with a block size parameter B that uses $B \times B$ blocks.

5.7 [12] <5.2> Assume you are redesigning a hardware prefetcher for the *unblocked* matrix transposition code above. The simplest type of hardware prefetcher only prefetches sequential cache blocks after a miss. More complicated "nonunit stride" hardware prefetchers can analyze a miss reference stream, and detect and prefetch nonunit strides. In contrast, software prefetching can determine nonunit strides as easily as it can determine unit strides. Assume prefetches write directly into the cache and no *pollution* (overwriting data that needs to be used before the data that is prefetched). In the steady state of the inner loop, what is the performance (in cycles per iteration) when using an ideal nonunit stride prefetcher?

5.8 [15/15] <5.2> Assume you are redesigning a hardware prefetcher for the *unblocked* matrix transposition code as in Exercise 5.7. However, in this case we evaluate a simple two-stream sequential prefetcher. If there are level 2 access slots available, this prefetcher will fetch up to 4 sequential blocks after a miss and place them in a stream buffer. Stream buffers that have empty slots obtain access to the level 2 cache on a round-robin basis. On a level 1 miss, the stream buffer

that has least recently supplied data on a miss is flushed and reused for the new miss stream.

a. [15] <5.2> In the steady state of the inner loop, what is the performance (in cycles per iteration) when using a simple two-stream sequential prefetcher assuming performance is limited by prefetching?

b. [15] <5.2> What percentage of prefetches are useful given the level 2 cache parameters?

5.9 [12/15] <5.2> With software prefetching it is important to be careful to have the prefetches occur in time for use, but also minimize the number of outstanding prefetches, in order to live within the capabilities of the microarchitecture and minimize cache pollution. This is complicated by the fact that different processors have different capabilities and limitations.

a. [12] <5.2> Modify the unblocked code above to perform prefetching in software.

b. [15] <5.2> What is the expected performance of unblocked code with software prefetching?

Case Study 3: Main Memory Technology and Optimizations

Concepts illustrated by this case study

- Memory System Design: Latency, Bandwidth, Cost, and Power
- Calculating Impact of Memory System Performance

Using Figure 5.14, consider the design of a variety of memory systems. Assume a chip multiprocessor with eight 3 GHz cores and directly attached memory controllers (i.e., integrated northbridge) as in the Opteron. The chip multiprocessor (CMP) contains a single shared level 2 cache, with misses from that level going to main memory (i.e., no level 3 cache). A sample DDR2 SDRAM timing diagram appears in Figure 5.30. t_{RCD} is the time required to activate a row in a bank, while the CAS latency (CL) is the number of cycles required to read out a column in a row.

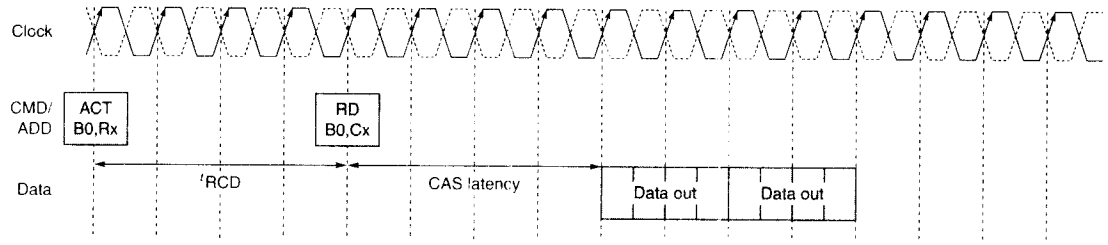

Figure 5.30 DDR2 SDRAM timing diagram.

Assume the RAM is on a standard DDR2 DIMM with ECC, having 72 data lines. Also assume burst lengths of 8 which read out 8 bits per data line, or a total of 32 bytes from the DIMM. Assume the DRAMs have a 1 KB page size, 8 banks, t_{RCD} = CL * Clock_frequency, and Clock_frequency = Transfers_per_second/2. The on-chip latency on a cache miss through levels 1 and 2 and back not including the DRAM access is 20 ns. Assume a DDR2-667 1 GB DIMM with CL = 5 is available for $130 and a DDR2-533 1 GB DIMM with CL = 4 is available for $100. (See *http://download.micron.com/pdf/technotes/ddr2/TN4702.pdf* for more details on DDR2 memory organization and timing.)

5.10 [10/10/10/12/12] <5.3> Assume the system is your desktop PC and only one core on the CMP is active. Assume there is only one memory channel.

a. [10] <5.3> How many DRAMs are on the DIMM if 512 Mbit DRAMs are used, and how many data I/Os must each DRAM have if only one DRAM connects to each DIMM data pin?

b. [10] <5.3> What burst length is required to support 32-byte versus 64-byte level 2 cache blocks?

c. [10] <5.3> What is the peak bandwidth ratio between the DIMMs for reads from an active page?

d. [12] <5.3> How much time is required from the presentation of the activate command until the last requested bit of data from the DRAM transitions from valid to invalid for the DDR2-533 1 GB CL = 4 DIMM?

e. [12] <5.3> What is the relative latency when using the DDR2-533 DIMM of a read requiring a bank activate versus one to an already open page, including the time required to process the miss inside the processor?

5.11 [15] <5.3> Assume just one DIMM is used in a system, and the rest of the system costs $800. Consider the performance of the system using the DDR2-667 and DDR2-533 DIMMs on a workload with 3.33 level 2 misses per 1K instructions, and assume all DRAM reads require an activate. Assume all 8 cores are active with the same workload. What is the cost divided by the performance of the whole system when using the different DIMMs assuming only one level 2 miss is outstanding at a time and an in-order core with a CPI of 1.5 not including level 2 cache miss memory access time?

5.12 [12] <5.3> You are provisioning a server based on the system above. All 8 cores on the CMP will be busy with an overall CPI of 2.0 (assuming level 2 cache miss refills are not delayed). What bandwidth is required to support all 8 cores running a workload with 6.67 level 2 misses per 1K instructions, and optimistically assuming misses from all cores are uniformly distributed in time?

5.13 [12] <5.3> A large amount (more than a third) of DRAM power can be due to page activation (see *http://download.micron.com/pdf/technotes/ddr2/TN4704.pdf* and *http://www.micron.com/systemcalc*). Assume you are building a system with 1 GB of memory using either 4-bank 512 Mbit × 4 DDR2 DRAMs or 8-bank 1 Gbit × 8 DRAMs, both with the same speed grade. Both use a page size of 1 KB. Assume DRAMs that are not active are in precharged standby and dissipate negligible

power. Assume the time to transition from standby to active is not significant. Which type of DRAM would be expected to result in lower power? Explain why.

Case Study 4: Virtual Machines

Concepts illustrated by this case study

- Capabilities Provided by Virtual Machines
- Impact of Virtualization on Performance
- Features and Impact of Architectural Extensions to Support Virtualization

Intel and AMD have both created extensions to the x86 architecture to address the shortcomings of the x86 for virtualization. Intel's solution is called VT-x (Virtualization Technology x86) (see IEEE [2005] for more information on VT-x), while AMD's is called Secure Visual Machine (SVM). Intel has a corresponding technology for the Itanium architecture called VT-i. Figure 5.31 lists the early performance of various system calls under native execution, pure virtualization, and paravirtualization for LMbench using Xen on an Itanium system with times measured in microseconds (courtesy of Matthew Chapman of the University of New South Wales).

5.14 [10/10/10/10/10] <5.4> Virtual machines have the potential for adding many beneficial capabilities to computer systems, for example, resulting in improved total cost of ownership (TCO) or availability. Could VMs be used to provide the following capabilities? If so, how could they facilitate this?

a. [10] <5.4> Make it easy to consolidate a large number of applications running on many old uniprocessor servers onto a single higher-performance CMP-based server?

b. [10] <5.4> Limit damage caused by computer viruses, worms, or spyware?

c. [10] <5.4> Higher performance in memory-intensive applications with large memory footprints?

d. [10] <5.4> Dynamically provision extra capacity for peak application loads?

e. [10] <5.4> Run a legacy application on old operating systems on modern machines?

5.15 [10/10/12/12] <5.4> Virtual machines can lose performance from a number of events, such as the execution of privileged instructions, TLB misses, traps, and I/O. These events are usually handled in system code. Thus one way of estimating the slowdown when running under a VM is the percentage of application execution time in system versus user mode. For example, an application spending 10% of its execution in system mode might slow down by 60% when running on a VM.

a. [10] <5.4> What types of programs would be expected to have larger slowdowns when running under VMs?

Benchmark	Native	Pure	Para
Null call	0.04	0.96	0.50
Null I/O	0.27	6.32	2.91
Stat	1.10	10.69	4.14
Open/close	1.99	20.43	7.71
Install sighandler	0.33	7.34	2.89
Handle signal	1.69	19.26	2.36
Fork	56.00	513.00	164.00
Exec	316.00	2084.00	578.00
Fork + exec sh	1451.00	7790.00	2360.00

Figure 5.31 Early performance of various system calls under native execution, pure virtualization, and paravirtualization.

b. [10] <5.4> If slowdowns were linear as a function of system time, given the slowdown above, how much slower would a program spending 30% of its execution in system time be expected to run?

c. [12] <5.4> What is the mean slowdown of the functions in Figure 5.31 under pure and para virtualization?

d. [12] <5.4> Which functions in Figure 5.31 have the smallest slowdowns? What do you think the cause of this could be?

5.16 [12] <5.4> Popek and Goldberg's definition of a virtual machine said that it would be indistinguishable from a real machine except for its performance. In this exercise we'll use that definition to find out if we have access to native execution on a processor or are running on a virtual machine. The Intel VT-x technology effectively provides a second set of privilege levels for the use of the virtual machine. What would happen to relative performance of a virtual machine if it was running on a native machine or on another virtual machine given two sets of privilege levels as in VT-x?

5.17 [15/20] <5.4> With the adoption of virtualization support on the x86 architecture, virtual machines are actively evolving and becoming mainstream. Compare and contrast the Intel VT-x and AMD Secure Virtual Machine (SVM) virtualization technologies. Information on AMD's SVM can be found in *http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24593.pdf.*

a. [15] <5.4> How do VT-x and SVM handle privilege-sensitive instructions?

b. [20] <5.4> What do VT-x and SVM do to provide higher performance for memory-intensive applications with large memory footprints?

Case Study 5: Putting It All Together: Highly Parallel Memory Systems

Concept illustrated by this case study

- Understanding the Impact of Memory System Design Tradeoffs on Machine Performance

The program in Figure 5.32 can be used 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. Figure 5.32 is the code in C. The first part is a procedure that uses a standard 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 results are output in .csn file format to facilitate importing into spreadsheets. 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. Running the program in single-user mode or at least without other active applications will give more consistent results. The code in Figure 5.32 was derived from a program written by Andrea Dusseau of U.C. Berkeley and was based on a detailed description found in Saavedra-Barrera [1992]. It has been modified to fix a number of issues with more modern machines and to run under Microsoft Visual C++.

The program shown in Figure 5.32 assumes that program addresses track physical addresses, which is true on the few machines 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 machine in order to get smooth lines in your results. To answer the exercises, assume that the sizes of all components of the memory hierarchy are powers of 2. Assume that the size of the page is much larger than the size of a block in a second-level cache (if there is one), 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. An example of the output of the program is plotted in Figure 5.33, with the key listing the size of the array that is exercised.

5.18 [10/12/12/12/12] <5.6> Using the sample program results in Figure 5.33:

a. [10] <5.6> How many levels of cache are there?

b. [12] <5.6> What are the overall size and block size of the first-level cache?

c. [12] <5.6> What is the miss penalty of the first-level cache?

d. [12] <5.6> What is the associativity of the first-level cache?

e. [12] <5.6> What effects can you see when the size of the data used in the array is equal to the size of the second-level cache?

```
#include "stdafx.h"
#include <stdio.h>
#include <time.h>
#define ARRAY_MIN (1024) /* 1/4 smallest cache */
#define ARRAY_MAX (4096*4096) /* 1/4 largest cache */
int x[ARRAY_MAX]; /* array going to stride through */

double get_seconds() { /* routine to read time in seconds */
    __time64_t ltime;
    _time64( &ltime );
    return (double) ltime;
}
int label(int i) {/* generate text labels */
    if (i<1e3) printf("%1dB,",i);
    else if (i<1e6) printf("%1dK,",i/1024);
    else if (i<1e9) printf("%1dM,",i/1048576);
    else printf("%1dG,",i/1073741824);
    return 0;
}
int _tmain(int argc, _TCHAR* argv[]) {
int register nextstep, i, index, stride;
int csize;
double steps, tsteps;
double loadtime, lastsec, sec0, sec1, sec; /* timing variables */

/* Initialize output */
printf(" ,");
for (stride=1; stride <= ARRAY_MAX/2; stride=stride*2)
    label(stride*sizeof(int));
printf("\n");

/* Main loop for each configuration */
for (csize=ARRAY_MIN; csize <= ARRAY_MAX; csize=csize*2) {
    label(csize*sizeof(int)); /* print cache size this loop */
    for (stride=1; stride <= csize/2; stride=stride*2) {

        /* Lay out path of memory references in array */
        for (index=0; index < csize; index=index+stride)
            x[index] = index + stride; /* pointer to next */
        x[index-stride] = 0; /* loop back to beginning */

        /* Wait for timer to roll over */
        lastsec = get_seconds();
        do sec0 = get_seconds(); while (sec0 == lastsec);

        /* Walk through path in array for twenty seconds */
        /* This gives 5% accuracy with second resolution */
        steps = 0.0; /* number of steps taken */
        nextstep = 0; /* start at beginning of path */
        sec0 = get_seconds(); /* start timer */
        do { /* repeat until collect 20 seconds */
            for (i=stride;i!=0;i=i-1) { /* keep samples same */
                nextstep = 0;
                do nextstep = x[nextstep]; /* dependency */
                while (nextstep != 0);
            }
            steps = steps + 1.0; /* count loop iterations */
            sec1 = get_seconds(); /* end timer */
        } while ((sec1 - sec0) < 20.0); /* collect 20 seconds */
        sec = sec1 - sec0;

        /* Repeat empty loop to loop subtract overhead */
        tsteps = 0.0; /* used to match no. while iterations */
        sec0 = get_seconds(); /* start timer */
        do { /* repeat until same no. iterations as above */
            for (i=stride;i!=0;i=i-1) { /* keep samples same */
                index = 0;
                do index = index + stride;
                while (index < csize);
            }
            tsteps = tsteps + 1.0;
            sec1 = get_seconds(); /* - overhead */
        } while (tsteps<steps); /* until = no. iterations */
        sec = sec - (sec1 - sec0);
        loadtime = (sec*1e9)/(steps*csize);
        /* write out results in .csv format for Excel */
        printf("%4.1f,", (loadtime<0.1) ? 0.1 : loadtime);
      }; /* end of inner for loop */
      printf("\n");
    }; /* end of outer for loop */
    return 0;
}
```

Figure 5.32 C program for evaluating memory systems.

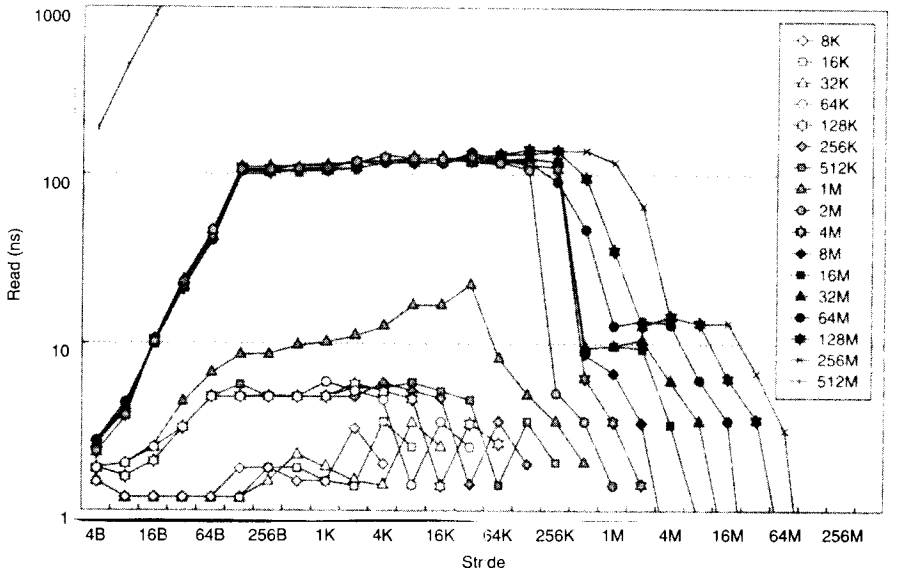

Figure 5.33 Sample results from program in Figure 5.32.

5.19 [15/20/25] <5.6> Modify the code in Figure 5.32 to measure the following system characteristics. 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.

a. [15] <5.6> Is the L1 cache write-through or write-back?

b. [20] <5.6> Is the memory system blocking or nonblocking?

c. [25] <5.6> For a nonblocking memory system, how many outstanding memory references can be supported?

5.20 [25/25] <5.6> In multiprocessor memory systems, lower levels of the memory hierarchy may not be able to be saturated by a single processor, but should be able to be saturated by multiple processors working together. Modify the code in Figure 5.32, and run multiple copies at the same time. Can you determine:

a. [25] <5.6> How much bandwidth does a shared level 2 or level 3 cache (if present) provide?

b. [25] <5.6> How much bandwidth does the main memory system provide?

5.21 [30] <5.6> Since instruction-level parallelism can also be effectively exploited on in-order superscalar processors and VLIWs with speculation, one important reason for building an out-of-order (OOO) superscalar processor is the ability to tolerate unpredictable memory latency caused by cache misses. Hence, you can think about hardware supporting OOO issue as being part of the memory system! Look at the floorplan of the Alpha 21264 in Figure 5.34 to find the

relative area of the integer and floating-point issue queues and mappers versus the caches. The queues schedule instructions for issue, and the mappers rename register specifiers. Hence these are necessary additions to support OOO issue. The 21264 only has level 1 data and instruction caches on chip, and they are both 64 KB two-way set associative. Use an OOO superscalar simulator such as Simplescalar *(www.cs.wisc.edu/~mscalar/simplescalar.html)* on memory-intensive benchmarks to find out how much performance is lost if the area of the issue queues and mappers is used for additional level 1 data cache area in an in-order superscalar processor, instead of OOO issue in a model of the 21264. Make sure the other aspects of the machine are as similar as possible to make the comparison fair. Ignore any increase in access or cycle time from larger caches and effects of the larger data cache on the floorplan of the chip. (Note this comparison will not be totally fair, as the code will not have been scheduled for the in-order processor by the compiler.)

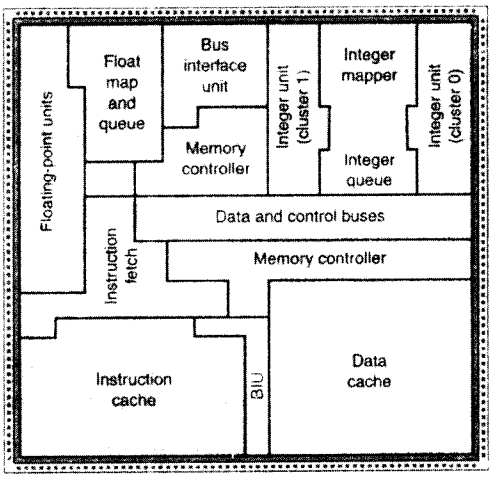

Figure 5.34 Floorplan of the Alpha 21264 [Kessler 1999].