2

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).

Computer Architecture. https://doi.org/10.1016/B978-0-443-15406-5.00002-X

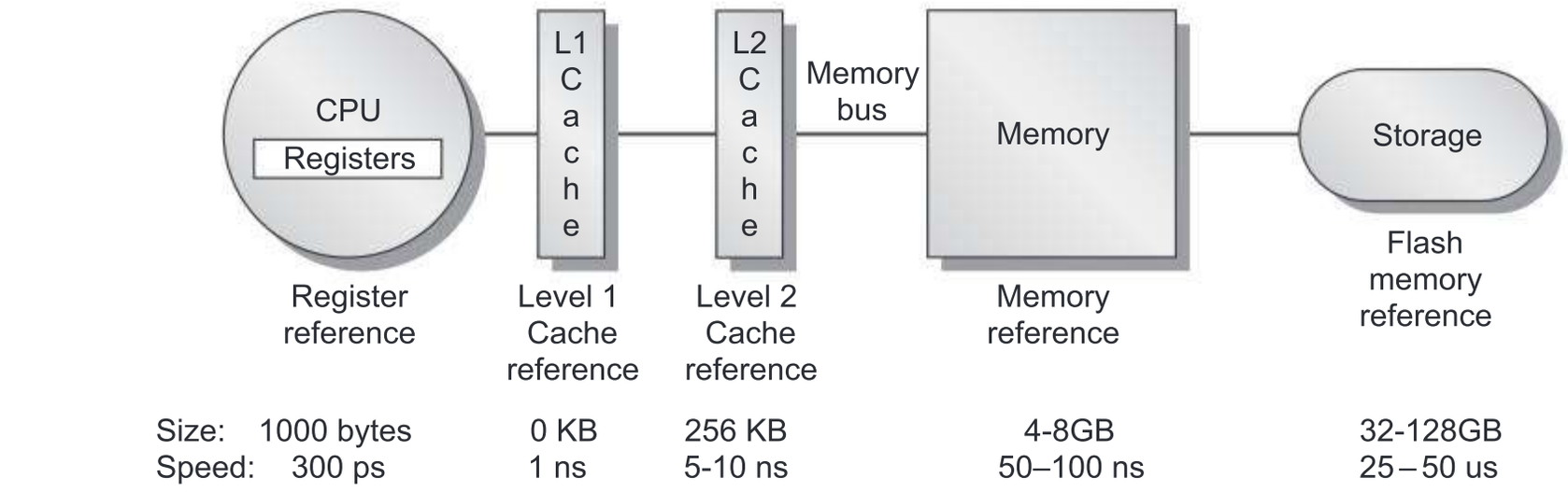
2.1 Introduction

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 trade-offs in the 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 for a given implementation technology and power budget, smaller hardware can be made faster led to hierarchies based on memories of different speeds and sizes. Figure 2.1 shows several different multilevel memory hierarchies, including typical sizes and speeds of access. As Flash and next-generation memory technologies continue to close the gap with disks in cost per bit, such technologies are likely to increasingly replace magnetic disks for secondary storage. As Figure 2.1 shows, these technologies are already used in many personal computers and increasingly in servers, where the advantages in performance, power, and density are significant.

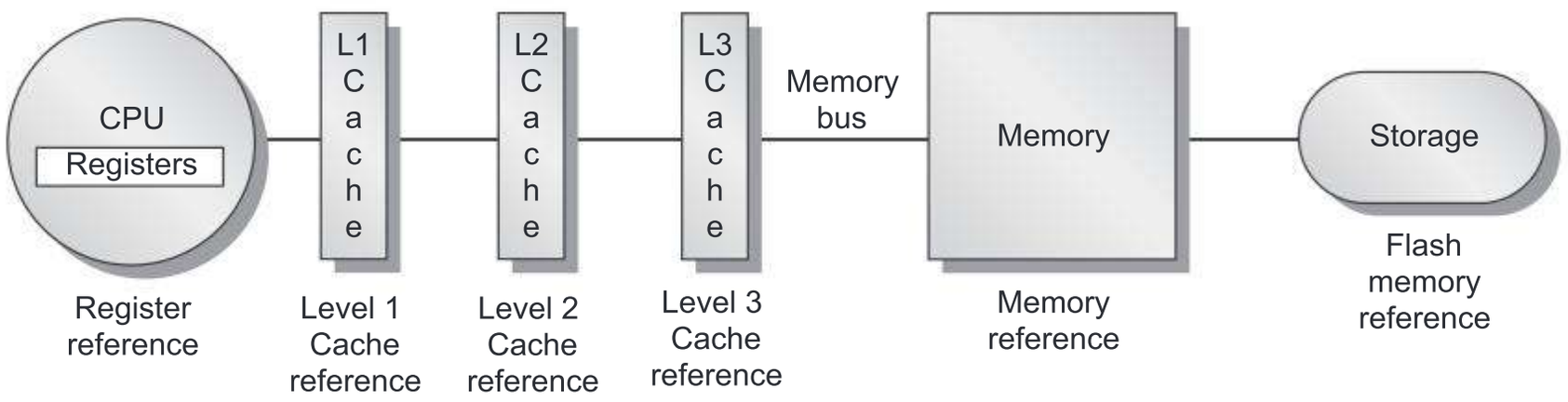
Because fast memory is more expensive, a memory hierarchy is organized into several levels—each smaller, faster, and more expensive per byte than the next lower level, which is farther from the processor. The goal is to provide a memory system with a cost per byte that is almost as low as the cheapest level of memory and a speed almost as fast as the fastest level. In most cases (but not all) the data contained in a lower level are a superset of the next higher level. This property, called the *inclusion property*, is usually maintained by the main memory in the case of caches and by secondary storage (disk or Flash) in the case of virtual memory.

The importance of the memory hierarchy has increased with advances in performance of processors. Figure 2.2 plots single core performance projections against the historical performance improvement in time to access main memory. The processor line shows the increase in memory requests per second on average (i.e., the inverse of the latency between memory references) for a single core, while the memory line shows the increase in DRAM accesses per second (i.e., the inverse of the DRAM access latency), assuming a single DRAM and a single memory bank. The reality is more complex because the processor request rate is not uniform, processors often contain multiple cores, and the memory system typically has multiple banks of DRAMs and channels. Although the gap in access time has increased significantly for many years, the lack of significant performance improvement in single processors has led to a slowdown in the growth of the gap between processors and DRAM.

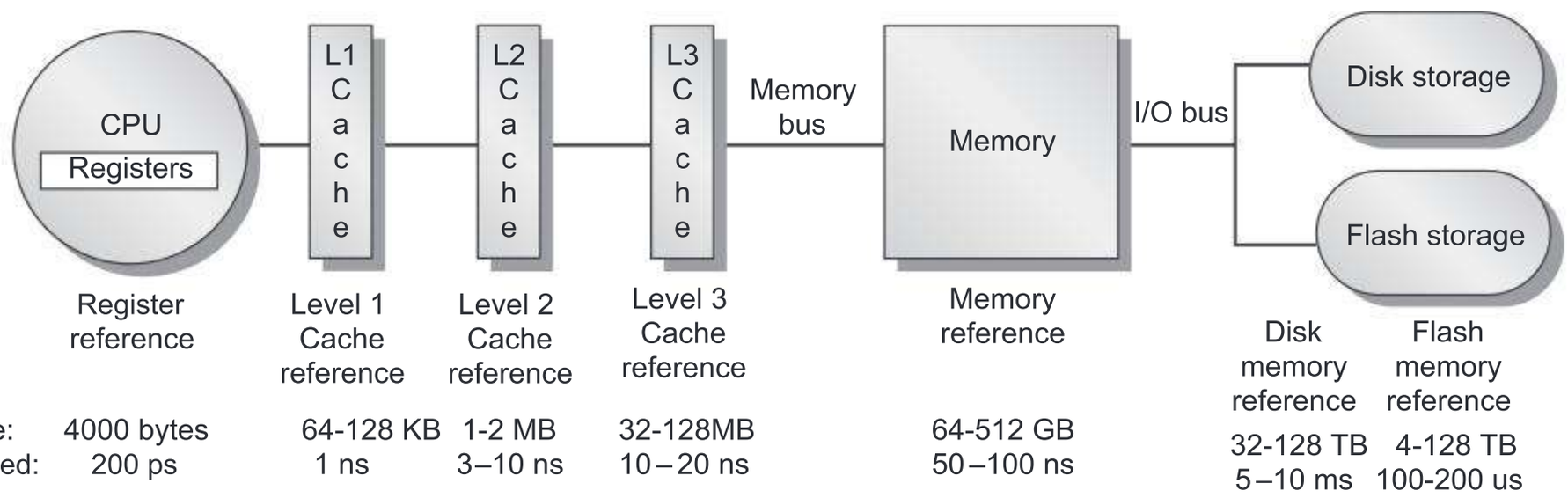
Because high-end processors have multiple cores, the bandwidth requirements are greater than for single cores. Although single-core bandwidth has grown more slowly in recent years, the gap between CPU memory demand and DRAM bandwidth continues to grow as the numbers of cores grow. A modern high-end desktop processor such as the Intel Core i9 12900 can generate four



(A) Memory hierarchy for a personal mobile device



(B) Memory hierarchy for a laptop or a desktop



(C) Memory hierarchy for server

Figure 2.1 The levels in a typical memory hierarchy in a personal mobile device (PMD), such as a cell phone or tablet (A), in a laptop or desktop computer (B), and in a server (C). 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 a factor of 10^9 from picoseconds to milliseconds in the case of magnetic disks and that the size units change by a factor of 10^{10} from thousands of bytes to tens of terabytes. If we were to add warehouse-sized computers, as opposed to just servers, the capacity scale would increase by three to six orders of magnitude. Solid-state drives (SSDs) composed of Flash are used exclusively in PMDs and heavily in both laptops and desktops. In many desktops the primary storage system is SSD, and expansion and backup are primarily hard disk drives (HDDs). Likewise, many servers mix SSDs and HDDs. Increasingly, HDDs are being used mostly for backup and archival storage.

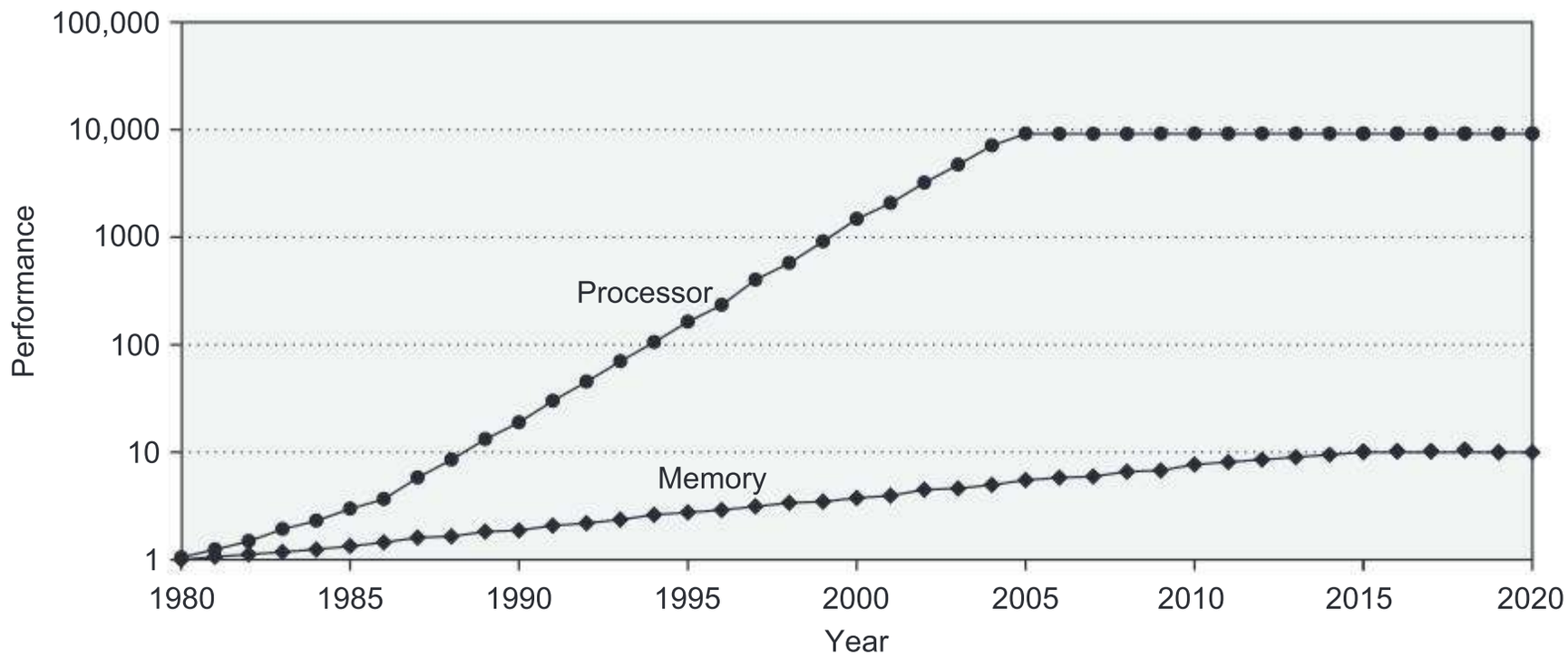


Figure 2.2 Starting with 1980 performance as a baseline, the gap in performance, measured as the difference in the time between processor memory requests (for a single processor or core) and the latency of a DRAM access, is plotted over time. In mid-2017, AMD, Intel and NVIDIA all announced chip sets using versions of HBM (High Bandwidth Memory) technology, which is not included in this graph. Note that the vertical axis must be on a logarithmic scale to record the size of the processor-DRAM performance gap. The memory baseline was 64 KiB DRAM in 1980, with a 1.07 per year performance improvement in latency until 2015 (see Figure 2.4 on page 88) and essentially no improvement in the past 5 years. The processor line assumes a 1.25 improvement per year until 1986, a 1.52 improvement until 2000, a 1.20 improvement between 2000 and 2005, and only small improvements in processor performance (on a per-core basis) between 2005 and 2015. As you can see, until 2010 memory access times in DRAM improved slowly but consistently; since 2010 the improvement in access time has reduced, as compared with the earlier periods, although there have been continued improvements in bandwidth. See Figure 1.1 in Chapter 1 for more information.

data memory references per core each clock cycle. With eight cores and an average 3 GHz clock rate, the i9 can generate a peak of 192 billion 128-bit data memory references per second, in addition to a peak instruction demand of about 48 billion 128-bit instruction references; this is a total peak demand bandwidth of 3840 GiB/s! This incredible bandwidth is achieved by multiporting and pipelining the caches; by using three levels of caches, with two private levels per core and a shared L3; and by using separate instruction and data caches at the first level. In contrast, the peak bandwidth for a single-bus DDR5 main memory, using two memory channels, is only 2% of the demand bandwidth (56 GiB/s). HBM access time is lower, and bandwidth is higher (see Sections 2.2 and 2.3).

Traditionally, designers of memory hierarchies focused on optimizing average memory access time, which is determined by the cache access time, miss rate, and miss penalty. More recently, however, power has become a major consideration. In high-end microprocessors there may be 60 MiB or more of on-chip cache, or up to 1 GiB in a multichip socket. A large second- or third-level cache will consume significant power both as leakage when not operating (called *static power*) and as active power when performing a read or write (called *dynamic power*). (See Section 2.3.) The problem is even more acute in processors within PMDs where the CPU is less aggressive, and the power budget may be 20 to 50 times

smaller. In such cases the caches can account for 25% to 50% of the total power consumption. Thus designers must consider both performance and power trade-offs, and we will examine both in this chapter.

Basics of Memory Hierarchies: A Quick Review

The increasing size and thus importance of the processor-memory 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 and their operation. The bulk of the chapter, however, describes more advanced innovations that attack the processor−memory performance gap.

When a word is not found in the cache, the word must be fetched from a lower level in the hierarchy (which may be another cache or the main memory) and placed in the cache before continuing. Multiple words, called a *block* (or *line*), are moved for efficiency reasons, and because they are likely to be needed soon due to spatial locality. Each cache block includes a *tag* to indicate 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 because the copy in the cache and memory will be identical. Caching writes is more difficult; for example, 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 in a write-back cache 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 are placed in the buffer rather than wait for full latency to write the data into memory. If the write buffer contains other modified blocks, the addresses can be checked to see if the address of the 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 Intel Core i7, among many others, uses write merging. This optimization is more helpful with a write-through cache since a write-back cache is normally writing back an entire block. Note that I/O 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 I/O register rather than use multiword writes using a single address. These side effects are typically implemented by marking the pages as requiring nonmerging write-through by the caches.

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 C's 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 were to have an infinite-sized 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 multiple blocks map to its set and accesses to the different blocks are intermingled.

Figure B.8 on page 24 shows the relative frequency of cache misses broken down by the three C's. As mentioned in Appendix B, the three C's model is conceptual, and although its insights usually hold, it is not a definitive model for explaining the cache behavior of individual references.

As we will see in Chapters 3 and 5, multithreading and multiple cores add complications for caches, both increasing the potential for capacity misses as well as adding a fourth C, for *coherency* misses due to cache flushes to keep multiple caches coherent in a multiprocessor; we will consider these issues in Chapter 5.

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

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

(This equation is often expressed in integers rather than fractions, as misses per 1000 instructions.)

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 (i.e., 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. In Chapter 3 we will see that speculative processors may execute other instructions during a miss, thereby reducing the effective miss penalty. The use of multithreading (introduced in Chapter 3) also allows a processor to tolerate misses without being forced to idle. As we will see shortly, to take advantage of such latency-tolerating techniques, we need caches that can service requests during the handling of an outstanding miss.

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

Section B.3 in Appendix B presents six basic cache optimizations, which we quickly review here. The appendix also gives quantitative examples of the benefits of these optimizations. We also comment briefly on the power implications of these trade-offs.

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. Larger blocks reduce compulsory misses, but they also increase the miss penalty. Because larger blocks lower the number of tags, they can slightly reduce static power. Larger block sizes can also increase capacity or conflict misses, especially in smaller caches. Choosing the right block size is a complex trade-off that depends on the size of cache and the miss penalty.

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. Larger caches increase both static and dynamic power.

Higher associativity to reduce miss rate—Obviously, increasing associativity reduces conflict misses. Greater associativity can come at the cost of increased hit time. As we will see shortly, associativity can also increase power consumption.

Multilevel caches to reduce miss penalty—A difficult decision is whether to make the cache hit time fast, to keep pace with the high clock rate of processors, or to make the cache large to reduce the gap between the processor accesses and main memory accesses. Adding another level of cache between the original cache and memory simplifies the decision. The first-level cache can be small enough to match a fast clock cycle time, and the second-level (or third-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. Multilevel caches are more power-efficient than a single aggregate cache. If L1 and L2 refer, respectively, to first- and second-level caches, we can redefine the average memory access time:

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

Today, all processors, from low-end PMDs to high-end server multicores, use multilevel caches, typically with two or three levels. The tradeoffs in design can be different for different cache levels due to the desire to optimize either the hit time or the miss time. For example, while the hit-time of L1 is critical, performance is less sensitive to a small increase in the hit-time of the LLC (Last Level Cache, which is typically L2 or L3). Furthermore, the miss penalty for the LLC is typically larger than that for L1; therefore an optimization that might be too costly in L1 can make sense in the LLC.

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. This choice has little effect on power consumption.

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 2.4 and B.4.) A common optimization is to use the page offset—the part that is identical in both virtual and physical addresses—to index the cache, as described in Appendix B, page B.36. This virtual index/physical tag method introduces some system complications and/or limitations on the size and structure of the L1 cache, but the advantages of removing the translation lookaside buffer (TLB) access from the critical path outweigh the disadvantages. Increasing the associativity of the L1 cache can also avoid this problem, since it reduces the number of bits being used to index the cache, and this approach is used in recent Intel and ARM processors.

Note that each of the preceding six optimizations 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 preceding material and the details in Appendix B. In the "Putting It All Together" section we examine the memory hierarchy for a microprocessor designed for a high-end desktop or smaller server, the Intel Core i9 12900, as well as one designed for use in a PMD, the Arm Cortex-53, which is the basis for the processor used in several tablets and smartphones. Within each of these classes, there is a significant diversity in approach because of the intended use of the processor.

Although the i9 has more cores and bigger caches than the Intel processors designed for mobile uses, the processors have similar architectures. A processor designed for small servers, such as the i9 series, or larger servers, such as the Intel Xeon processors, typically is running many concurrent processes, often for different users. Thus memory bandwidth becomes more important, and these processors offer larger caches and more aggressive memory systems to boost that bandwidth.

In contrast, PMDs serve one user, generally have smaller operating systems, usually less multitasking (running of several applications simultaneously), and simpler applications. PMDs must consider both performance and energy consumption, which determines battery life. Before we dive into more advanced cache organizations and optimizations, one needs to understand the various memory technologies and how they are evolving.

2.2 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)

This section describes the technologies used in a memory hierarchy, specifically in building caches and main memory. These technologies are SRAM (static random access memory, used for cache), DRAM (dynamic random access memory, used for main memory), and Flash (used for nonvolatile storage). The last of these is used as an alternative to hard disks, but because its characteristics are based on semiconductor technology, it is appropriate to include it in this section.

When a cache miss occurs, we need to move the data from the main memory as quickly as possible, which requires a high bandwidth memory (HBM). This bandwidth can be achieved by organizing the many DRAM chips that make up the main memory into multiple memory banks and ranks and by making the memory bus wider, or by doing both.

To allow memory systems to keep up with the bandwidth demands of modern processors, memory innovations started happening inside the DRAM chips themselves. This section describes the technology inside the memory chips. Before describing the technologies and options, we need to introduce some terminology.

Access time is the time between when a read is requested and when the desired word arrives. Note that with burst transfer, which both Flash and DRAM support, the time to transfer the block may be larger than the time to get the first portion of a block.

Virtually all computers since 1975 have used DRAMs for main memory and SRAMs for cache, with one to three levels integrated onto the processor chip with the CPU. Today, all PMDs and laptops and most desktops use Flash rather than disk drives. Many servers use Flash in combination with disk drives.

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 but this delay is only

seen when the next request must access the same memory bank. 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.

In earlier times most desktop and server systems used SRAM chips for their primary, secondary, or tertiary caches. Today, all three levels of caches are integrated onto the processor chip. In high-end server chips there may be as many as 60 cores and 100 MiB or more of cache with chiplets, and there may be over 100 cores and 1 GB or more of cache using stacked dies. Such systems are often configured with 256-512 GiB of DRAM per processor chip. The access times for large, third-level, on-chip caches are typically four to eight times that of a second-level cache. Even so, the L3 access time is usually at least five times faster than a DRAM access.

On-chip, cache SRAMs are normally organized with a width that matches the block size of the cache, with the tags stored in parallel to each block. This arrangement allows an entire block to be read out or written into a single cycle. This capability is particularly useful when writing data fetched after a miss into the cache or when writing back a block that must be evicted from the cache. The access time to the cache (ignoring the hit detection and selection in a set associative cache) is proportional to the number of blocks in the cache, whereas the energy consumption depends both on the number of bits in the cache (static power due to leakage) and on the number of blocks (dynamic power incurred on an access). To increase bandwidth and access time and reduce power consumption, many multilevel caches are organized in banks, a topic we explore in Section 2.3.

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 2.3 shows the basic DRAM organization. One-half of the address is sent first during 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, because the memory is organized as a rectangular matrix addressed by rows and columns.

An additional requirement of DRAM derives from the property signified by its first letter, *D*, for *dynamic*. To pack more bits per chip, DRAMs use only a single transistor, which effectively acts as a capacitor, to store a bit. This has two implications: first, the sensing wires that detect the charge must be precharged, which sets them "halfway" between a logical 0 and 1, allowing the small charge stored in the cell to cause a 0 or 1 to be detected by the sense amplifiers. On reading, a row is placed into a row buffer, where CAS signals can select a portion of the row to read out from the DRAM. Because reading a row destroys the information, it must be written back when the row is no longer needed. This write-back happens in overlapped fashion, but in early DRAMs it meant that the cycle time before a new row could be read was larger than the time to read a row and access a portion of that row.

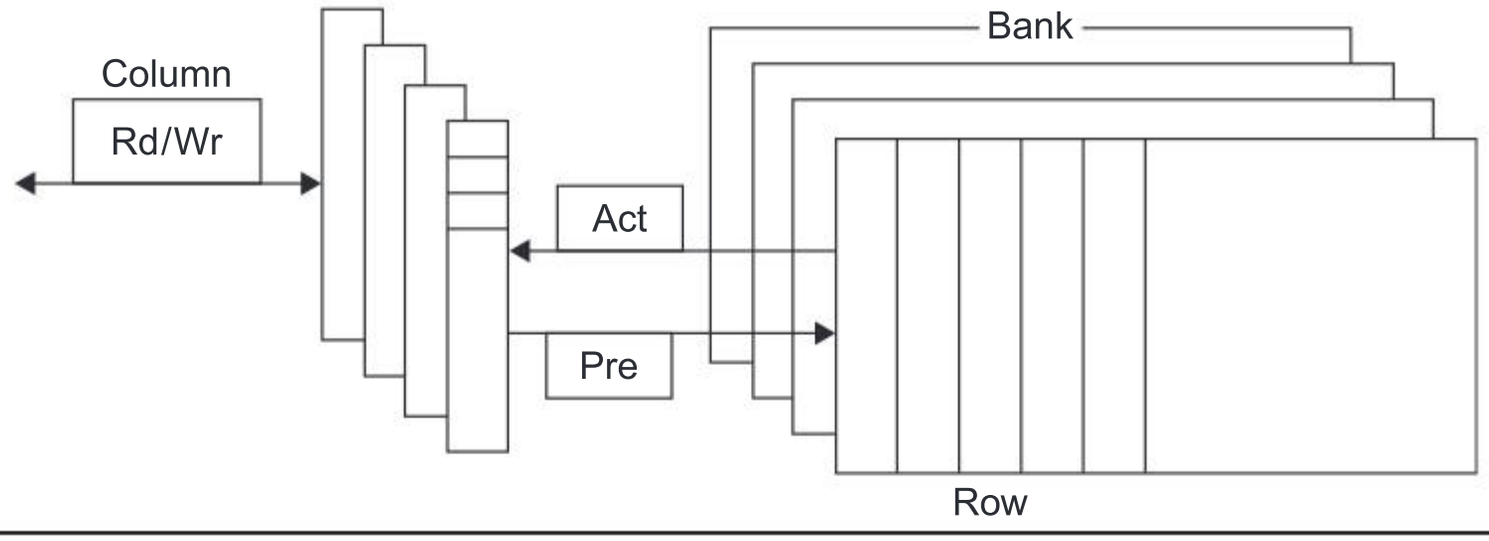


Figure 2.3 Internal organization of a DRAM. Modern DRAMs are organized in banks, up to 16 for DDR4. Each bank consists of a series of rows. Sending an Act (Activate) command opens a bank and a row and loads the row into a row buffer. When the row is in the buffer, it can be transferred by successive column addresses at whatever the width of the DRAM is (typically 4, 8, or 16 bits in DDR4) or by specifying a block transfer and the starting address. The Precharge (Pre) command closes the bank and row and readies it for a new access. Each command, as well as block transfers, are synchronized with a clock. See the next section discussing SDRAM. The row and column signals are sometimes called RAS and CAS, respectively, based on the original names of the signals.

In addition, to prevent loss of information as the charge in a cell leaks away (assuming it is not read or written), each bit must be "refreshed" periodically. Fortunately, all the bits in a row can be refreshed simultaneously just by reading that row and writing it back. Therefore every DRAM in the memory system must access every row within a certain time window, such as 64 ms. Both DRAM controllers and chips 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 a row activation and a precharge that also writes the row back (which takes roughly 2/3 of the time to get a datum because no column select is needed), and this is required for each row of the DRAM. Because 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. In fact, with SDRAMs, a DRAM controller (usually on the processor chip) tries to optimize accesses by avoiding opening new rows and using block transfer when possible. Refresh adds another unpredictable factor.

Amdahl suggested as a rule of thumb that memory capacity should grow linearly with processor speed to keep a balanced system. Thus a 1000 MIPS processor should have 1000 MiB of memory. Processor designers rely on DRAMs to supply that demand. In the past, they expected a fourfold improvement in capacity every 3 years, or 55% per year. Unfortunately, the performance of DRAMs is growing at a much slower rate. The slower performance improvements arise primarily because of smaller decreases in the row access time, which is determined by issues such as power limitations and the charge capacity (and thus the size) of an individual memory cell. Before we discuss these performance trends in more

detail, we need to describe the major changes that occurred in DRAMs starting in the mid-1990s.

Improving Memory Performance Inside a DRAM Chip: SDRAMs

Although very early DRAMs included a buffer allowing multiple column accesses to a single row, without requiring a new row access, they used an asynchronous interface, which meant that every column access and transfer involved overhead to synchronize with the controller. In the mid-1990s designers added a clock signal to the DRAM interface so that the repeated transfers would not bear that overhead, thereby creating *synchronous DRAM* (SDRAM). In addition to reducing overhead, SDRAMs allowed the addition of a burst transfer mode where multiple transfers can occur without specifying a new column address. For example, eight or more 16-bit transfers can occur without sending any new addresses by placing the DRAM in burst mode. The inclusion of such burst mode transfers has meant that there is a significant gap between the bandwidth for a stream of random accesses and access to a block of data.

To overcome the problem of getting more bandwidth from the memory as DRAM density increased, DRAMS were also made wider. Initially, they offered a four-bit transfer mode; in 2024 DDR DRAMs had up to 4, 8, 16, or 32 bit buses. In the early 2000s a further innovation was introduced: *double data rate* (DDR), which allows a DRAM to transfer data both on the rising and the falling edge of the memory clock, thereby doubling the peak data rate.

Finally, SDRAMs introduced *banks* to help with power management, improve access time, and allow interleaved and overlapped accesses to different banks. Access to different banks can be overlapped with each other, and each bank has its own row buffer, allowing multiple rows in different banks to be open at once. Creating multiple banks inside a DRAM effectively adds another segment to the address, which now consists of bank number, row address, and column address. When an address is sent that designates a new row for a bank, that row must be opened, incurring an additional delay. The management of banks and row buffers is completely handled by modern memory control interfaces so that when a subsequent access specifies an open row in a bank, the access can happen quickly, sending only the column address.

To initiate a new access, the DRAM controller sends a bank and row number (called *Activate* in SDRAMs and formerly called RAS—row select). That command opens the row and reads the entire row into a buffer. A column address can then be sent, and the SDRAM can transfer one or more data items, depending on whether it is a single item request or a burst request. Before accessing a new row, the bank must be precharged. If the next row is in the same bank, then the precharge delay is seen; however, if the row is in another bank, closing the row and precharging can overlap with accessing the new row. In SDRAMs each of these command cycles requires an integral number of clock cycles.

From 1980 to 1995, DRAMs scaled with Moore's law, doubling capacity every 18 months (or by a factor of 4 in 3 years). From the mid-1990s to 2010, capacity increased more slowly, with roughly 26 months between a doubling. It took 6 years (2010–2016) for capacity to double again, and the next doubling happened at the end of 2024! Figure 2.4 shows the capacity and access time for various generations of DDR SDRAMs. From DDR1 to DDR3, access times improved by a factor of about 3, or about 7% per year. DDR4 and DDR5 improve power and bandwidth over DDR3 but have similar access latency.

As Figure 2.4 shows, DDR is a sequence of standards. DDR2 lowers power from DDR1 by dropping the voltage from 2.5 to 1.8 V and offers higher clock rates: 266, 333, and 400 MHz. DDR3 drops voltage to 1.5 V and has a maximum clock speed of 800 MHz. DDR4, which shipped in volume in early 2016 but was expected in 2014, drops the voltage to 1.2 V and has a maximum expected clock rate of 1600 MHz. DDR5, which uses a 1.1 V, did not reach production quantities until 2022. DDR6 looks like it will not be available until 2025.

With the introduction of DDR, memory designers are increasingly focused on bandwidth, because improvements in access time are difficult. Wider DRAMs, burst transfers, and DDR all contributed to rapid increases in memory bandwidth. DRAMs are commonly sold on small boards called *dual inline memory modules* (DIMMs) that contain 4–16 DRAM chips and that are normally organized to be 8 bytes wide (+ ECC) for desktop and server systems. When DDR SDRAMs are packaged as DIMMs, they are confusingly labeled by the peak *DIMM* bandwidth. Therefore the DIMM name PC3200 comes from 200 MHz × 2 × 8 bytes, or 3200 MiB/s; it is populated with DDR SDRAM chips. Sustaining the confusion, the chips themselves are labeled with *the number of bits per second* rather than their clock rate, so a 200 MHz DDR chip is called a DDR400. Figure 2.5 shows the relationships' I/O clock rate, transfers per second per chip, chip bandwidth, chip name, DIMM bandwidth, and DIMM name.

			Best case access time (no precharge)			Precharge needed
Production year	Chip size	DRAM type	RAS time (ns)	CAS time (ns)	Total (ns)	Total (ns)
2000	256M bit	DDR1	21	21	42	63
2002	512M bit	DDR1	15	15	30	45
2004	1G bit	DDR2	15	15	30	45
2006	2G bit	DDR2	10	10	20	30
2010	4G bit	DDR3	13	13	26	39
2016	8G bit	DDR4	13	13	26	39
2021	8G bit	DDR5	13	13	26	39

Figure 2.4 Capacity and access times for DDR SDRAMs by year of production. Access time is for a random memory word and assumes a new row must be opened. If the row is in a different bank, we assume the bank is precharged; if the row is not open, then a precharge is required, and the access time is longer. As the number of banks has increased, the ability to hide the precharge time has also increased. DDR4 SDRAMs were initially expected in 2014 but did not begin production until early 2016. Note that DDR5 initially had the same size as DDR4, but in 2024, it has four times the density per chip.

Standard	I/O clock rate	M transfers/s	DRAM name	MiB/s/DIMM	DIMM name
DDR1	133	266	DDR266	2128	PC2100
DDR1	150	300	DDR300	2400	PC2400
DDR1	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
DDR4	1333	2666	DDR4-2666	21,300	PC21300
DDR5	2400	4800	DD5-4800	38,400	PC38400

Figure 2.5 Clock rates, bandwidth, and names of DDR DRAMS and DIMMs in 2024. 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. DDR5 saw significant first use in 2022.

Reducing Power Consumption in SDRAMs

Power consumption in dynamic memory chips consists of both dynamic power used in a read or write and static or standby power; both depend on the operating voltage. In DDR5 SDRAMs the operating voltage has dropped to 1.1 V, slightly improving power consumption over DDR4 and providing about a 25% reduction over DDR3. The addition of banks also reduced power because only the row in a single bank is read.

In addition to these changes, all recent SDRAMs support a power-down mode, which is entered by telling the DRAM to ignore the clock. Power-down mode disables the SDRAM, except for internal automatic refresh (without which entering power-down mode for longer than the refresh time will cause the contents of memory to be lost). Figure 2.6 shows the power consumption for three situations in a 2 GB DDR3 SDRAM. The exact delay required to return from low power mode depends on the SDRAM, but a typical delay is 200 SDRAM clock cycles. There are also low-power DDRs (LPDDR1-5) designed for use in portable devices. LPDDRs have lower supply voltage and faster clocks but allow fewer memory chips per channel that are directly connected to the CPU (versus in a DIMM).

Graphics data RAMs (GDRAMs GDDR1-6) are based on SDRAMs but offered wider interfaces and higher clock rates (and thus 2-5 times the bandwidth). The higher clock rate limits the number of GDRAMs per memory channel and requires a direct connection, like LPDDRs. In 2024 GDRAMs are being replaced by newer designs using HBM, which is the topic of the next section.

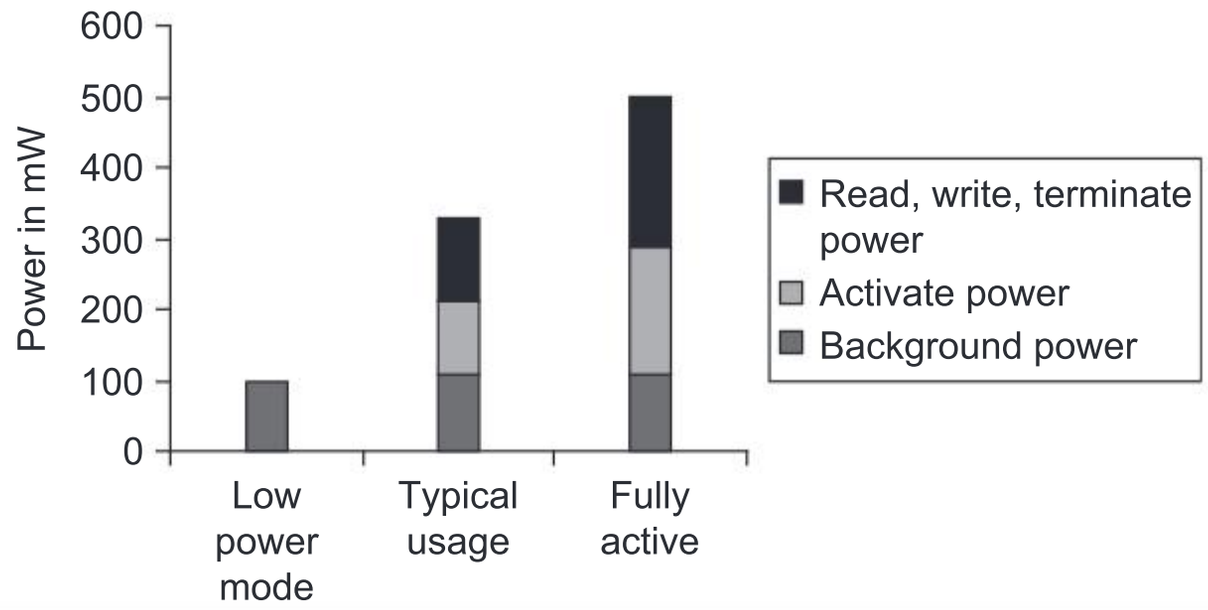


Figure 2.6 Power consumption for a DDR3 SDRAM operating under three conditions: low-power (shutdown) mode, typical system mode (DRAM is active 30% of the time for reads and 15% for writes), and fully active mode, where the DRAM is continuously reading or writing. Reads and writes assume bursts of eight transfers. These data are based on a Micron 1.5 V 2 GB DDR3-1066. Proportional savings occur with DDR4 and DDR5 SDRAMs, which also require 30% to 40% less power due to the lower operating voltage. Recent DDR5 DRAMs can handle power management at the DIMM level and offer deep shutdown modes with further power savings.

Although these enhancements have reduced the power consumption in DRAMs, the cost of going off-chip dominates both the access time and the power consumption. For example, in Figure 2.1 we see that the access time for an off-chip DRAM memory is 50 to 100 times longer than a small L1 cache and 5 to 10 times longer than the access time to a large L3. Energy scales similarly: off-chip DRAM accesses cost roughly 100 times more energy than to access a small L1 and 10 to 20 times the energy to access a large L3. The combination of the longer access time and larger energy consumption for off-chip access motivates the use of large LLCs and techniques to optimize their performance.

Packaging Innovation: Stacked or Embedded DRAMs or SRAMs

The most recent innovation in DRAMs is a packaging innovation rather than a circuit innovation. It places multiple DRAMs in stacks embedded within the same package as the processor. (Embedded DRAM also is used to refer to designs that place DRAM on the processor chip.) Placing the DRAM and processor in the same package lowers access latency (by shortening the delay between the DRAMs and the processor) and potentially increases bandwidth by allowing more and faster connections between the processor and DRAM. This technology is called *high bandwidth memory (HBM)*.

The primary use of this technology is to stack DRAMs, which are abutted with the CPU, in a single package using a substrate (interposer) containing the connections. Assuming the CPU has multiple memory channels, there could be four stacks (one per side), each with up to eight stacked DRAMs. With special versions of SDRAMs and four stacks, such a package could contain 32 GiB of

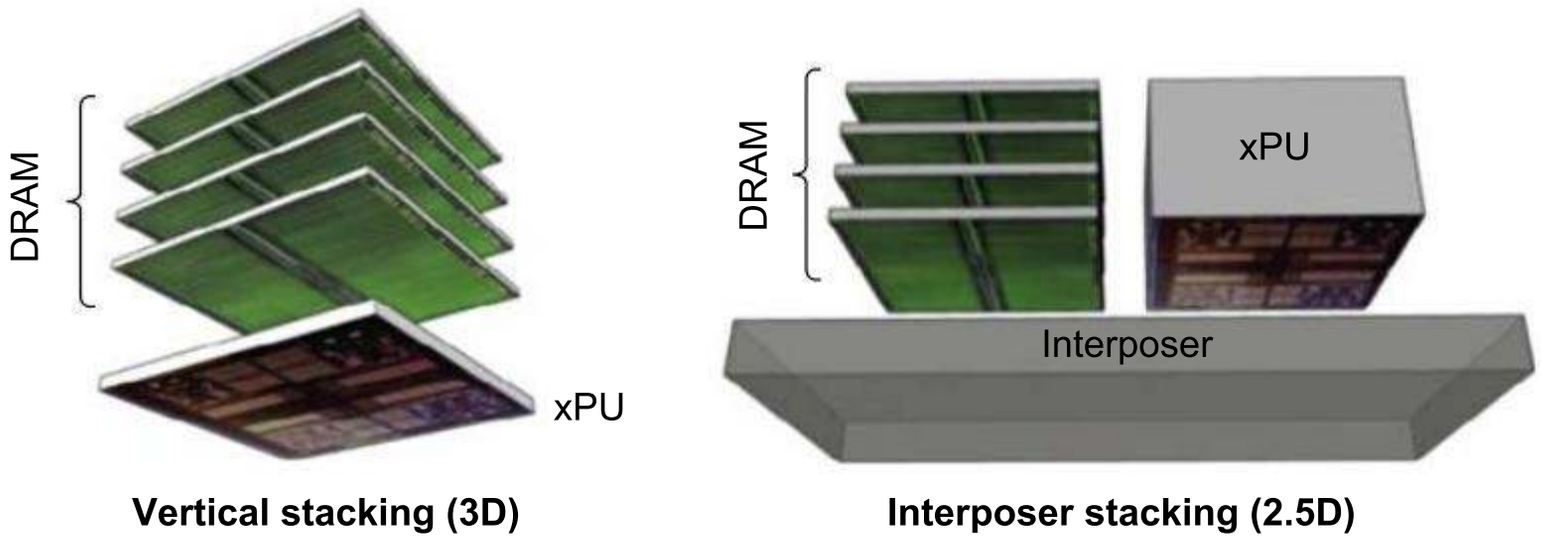


Figure 2.7 Two forms of die stacking. The 2.5D form is widely available and can accommodate up to four stacks with 8 DRAMs each. 3D stacking is currently used in the AMD Genoa chip set to stack a single SRAM on each die, supporting up to 1 GB of L3 cache on a 96-core chiplet.

memory and have data transfer rates of 4 TB/s. Because the chips must be specifically manufactured to stack, most early uses have been in high-end servers, GPUs, and DSAs (domain-specific architectures). Figure 2.7 shows the interposer approach and an alternative structure that stacks a chip directly on top of the processor. Heat removal and electrical noise are challenges in 3D structure shown in Figure 2.7, and, to date, the only use of vertical stacking is to place an SRAM used for L3 on top of the processor.

In some applications it may be possible to internally package enough DRAM to satisfy the needs of the application. For example, a version of an NVIDIA GPU used as a node in a special-purpose cluster design is being developed using HBM, and it is likely that HBM will become a successor to GDDR5 for higher-end applications. In some cases it may be possible to use HBM as main memory, although the cost limitations and heat removal issues currently rule out this technology for some embedded applications. In the next section we consider the possibility of using HBM as an additional level of cache as well as an additional main memory unit.

Flash Memory

Flash memory is a type of EEPROM (electronically erasable programmable read-only memory), which is normally read-only but can be erased. The other key property of Flash memory is that it holds its contents without any power. We focus on NAND Flash, which has a higher density than NOR Flash and is more suitable for large-scale nonvolatile memories; the drawback is that access is sequential, and writing is slower, as we will explain.

Flash is used as the secondary storage in PMDs and most desktops in the same manner that a disk functions in server. In addition, because most PMDs have a limited amount of DRAM, Flash may also act as a level of the memory hierarchy, to a much greater extent than it might have to do in a desktop or server with a main memory that might be 10–100 times larger.

Flash uses a very different architecture and has different properties than standard DRAM. The most important differences are:

1. Reads to Flash are sequential and read an entire page, which can be 512 bytes, 2 KiB, or 4 KiB. NAND Flash has a long delay to access the first byte of a random address (about 25 μS) but can supply the remainder of a page at about 40 MiB/s. By comparison, a DDR5 SDRAM takes about 40 ns to the first byte and can transfer the rest of the row at 5 GiB/s. Comparing the time to transfer 2 KiB, NAND Flash takes about 75 μS, while DDR SDRAM takes less than 450 ns, making Flash more than 150 times slower. Compared to magnetic disk, however, a 2 KiB read from Flash is 300 to 500 times faster. From these numbers, we can see why Flash is not a candidate to replace DRAM for main memory but has replaced magnetic disk in many systems since the cost per bit is close especially at smaller capacities.
2. Flash memory must be erased (thus the name flash for the "flash" erase process) before it is overwritten, and it is erased in blocks rather than individual bytes or words. This requirement means that when data must be written to Flash, an entire block must be assembled, either as new data or by merging the data to be written and the rest of the block's contents. For writing, Flash is about 1500 times slower than SDRAM, and about 8–15 times as fast as a magnetic disk.
3. Flash memory is nonvolatile (i.e., it keeps its contents even when power is not applied) and draws significantly less power when not reading or writing (from less than half in standby mode to zero when completely inactive).
4. Flash memory limits the number of times that any given block can be written, typically at least 100,000. By ensuring uniform distribution of written blocks throughout the memory, a system can maximize the lifetime of a Flash memory system. This technique, called *write leveling*, is handled by Flash memory controllers.
5. High-density NAND Flash is cheaper than SDRAM but more expensive than disks: roughly $0.12-0.20/GiB for Flash, $5 to $10/GiB for SDRAM, and $0.02-0.05/GiB for magnetic disks. In the past 5 years Flash has decreased in cost at a rate that is almost twice as fast as that of magnetic disks.

Like DRAM, Flash chips include redundant blocks to allow chips with small numbers of defects to be used; the remapping of blocks is handled in the Flash chip. Flash controllers handle page transfers, provide caching of pages, and handle write leveling.

The rapid improvements in high-density Flash have been critical to the development of low-power PMDs and laptops, but they have also significantly changed both desktops, which increasingly use solid-state disks, and large servers, which often combine disk and Flash-based storage.

Phase-Change Memory Technology

Phase-change memory (PCM) has been an active research area for decades. The technology typically uses a small heating element to change the state of a bulk substrate between its crystalline form and an amorphous form, which have different resistive properties. Each bit corresponds to a crosspoint in a two-dimensional network that overlays the substrate. Reading is done by sensing the resistance between an x and y point (thus the alternative name *memristor*), and writing is accomplished by applying a current to change the phase of the material. The absence of an active device (such as a transistor) should lead to lower costs and greater density than that of NAND Flash.

In 2017, Micron and Intel began delivering Xpoint memory chips based on this technology. PCM has two major advantages over Flash: better performance for random writes (since the erase step is unneeded) and potentially better longevity for writes. Read latency is slightly better than Flash. For reasons discussed in Fallacies and Pitfalls, PCM has not gathered momentum and has largely been shelved, at least for now.

Enhancing Dependability in Memory Systems

Large caches and main memories significantly increase the possibility of errors occurring both during the fabrication process and dynamically during operation. Errors that arise from a change in circuitry and are repeatable are called *hard errors* or *permanent faults*. Hard errors can occur during fabrication, as well as from a circuit change during operation (e.g., failure of a Flash memory cell after many writes). All DRAMs, Flash memory, and most SRAMs are manufactured with spare rows so that a small number of manufacturing defects can be accommodated by programming the replacement of a defective row by a spare row. Dynamic errors, which are changes to a cell's contents, not a change in the circuitry, are called *soft errors* or *transient faults*. Such errors, without some form of error detection, would become silent data errors, as discussed in Chapter 1.

Dynamic errors can be detected by parity bits and detected and fixed using error correcting codes (ECCs). Because instruction caches are read-only, parity suffices. In larger data caches and in main memory ECC is used to allow errors to be both detected and corrected. Parity requires only one bit of overhead to detect a single error in a sequence of bits. Because a multibit error would be undetected with parity (it would be a silent error), the number of bits protected by a parity bit must be limited. One parity bit per 8 data bits is a typical ratio. ECC can detect two errors and correct a single error with a cost of 8 bits of overhead per 64 data bits. ECC became standard in DDR5.

In very large systems the possibility of multiple errors as well as complete failure of a single memory chip becomes significant. Chipkill was introduced by IBM to solve this problem, and many very large systems, such as IBM servers and the Google Clusters, use this technology. (Intel calls their version SDDC.) Similar in nature to the RAID approach used for disks, Chipkill distributes the

data and ECC information so that the complete failure of a single memory chip can be handled by supporting the reconstruction of the missing data from the remaining memory chips. Using an analysis by IBM and assuming a 10,000-processor server with 4 GiB per processor yields the following rates of unrecoverable errors in three years of operation:

- Parity only: About 90,000, or one unrecoverable (or undetected) failure every 17 minutes.
- ECC only: About 3500, or about one undetected or unrecoverable failure every 7.5 hours.
- Chipkill: About one undetected or unrecoverable failure every 2 months.

Another way to look at this is to find the maximum number of servers (each with 4 GiB) that can be protected while achieving the same error rate as demonstrated for Chipkill. For parity, even a server with only one processor will have an unrecoverable error rate higher than a 10,000-server Chipkill-protected system. For ECC, a 17-server system would have about the same failure rate as a 10,000-server Chipkill system. Therefore Chipkill is a requirement for the 50,000−100,000 servers in warehouse-scale computers (see Section 6.8 of Chapter 6).

2.3 Ten Advanced Performance Optimizations for Memory Hierarchies

The preceding average memory access time formula gives us three metrics for cache optimizations: hit time, miss rate, and miss penalty. Given the recent trends, we add cache bandwidth and power consumption to this list. We can classify the 10 advanced memory system optimizations we examine into 5 categories based on these metrics:

1. *Reducing the hit time and power consumption*—Smaller caches can reduce both hit time and power consumption but at the cost of a reduced hit rate. Multiple banks can also be used to reduce hit time and power, especially in larger L2 and L3 caches.
2. *Increasing cache bandwidth*—Pipelined L1 caches, multibanked caches, and nonblocking caches. These techniques have varying impacts on power consumption. They may also reduce the miss penalty if the memory system can support multiple outstanding misses.
3. *Reducing the miss penalty*—Critical word first and merging write buffers. These optimizations have little impact on power. Using multiple memory banks and HBM memory can also reduce miss penalties by shortening the latency to access a block from memory, both for single block accesses and multiple parallel accesses.

4. *Reducing the miss rate*—Compiler optimizations and associativity. Obviously, any improvement at compile time improves power consumption. Using set associativity, while slightly increasing hit time, reduces the miss rate. Optimizing the replacement policy for set associative caches may also reduce miss rate.
5. *Reducing the miss penalty or miss rate via parallelism*—Hardware prefetching and compiler prefetching. These optimizations generally increase power consumption, primarily because of prefetched data that are unused.

In general, the hardware complexity increases as we go through these optimizations. In addition, several of the optimizations require sophisticated compiler technology, and one depends on using HBM. We will conclude with a summary of the implementation complexity and the performance benefits of the 10 techniques presented in Figure 2.17 on page 113. Because some of these are straightforward, we cover them briefly; others require more description.

First Optimization: Pipelined L1 Caches With Virtual Indexing and Set Associativity

Pipelining L1 allows a higher clock cycle, at the cost of slightly increased latency. For example, the pipeline for the instruction cache access for Intel Pentium processors in the mid-1990s took 1 clock cycle; for the Pentium Pro through Pentium III in the mid-1990s through 2000, it took 2 clock cycles; and for the Pentium 4, which became available in 2000, and the current Intel i7 and i9, it takes 4 clock cycles. Pipelining the instruction cache effectively increases the number of pipeline stages, leading to a greater penalty on mispredicted branches. Correspondingly, pipelining the data cache leads to more clock cycles between issuing a load and using the data (see Chapter 3). Today, all processors use some pipelining of L1, if only for the simple case of separating the access and hit detection, and many high-speed processors have three or more levels of cache pipelining.

Once caches were pipelined, it was easier to implement set associativity with little or no penalty on hit time, thus increasing the hit rate of L1 caches. Larger levels of associativity also avoid the aliasing problem that arises when caches are virtually indexed.

A typical pipeline for such a cache access might break the cache access/hit detection into four steps:

1. Address decode to get the address to select the word line in the cache.
2. Access the I or D cache SRAM.
3. Compare the tags for each element in the set against the physical tag coming from the TLB. Determine hit or miss and the element of the set if a hit.
4. Multiplex the appropriate cache block, align the word with the accessed component of the block, and transmit to the CPU.

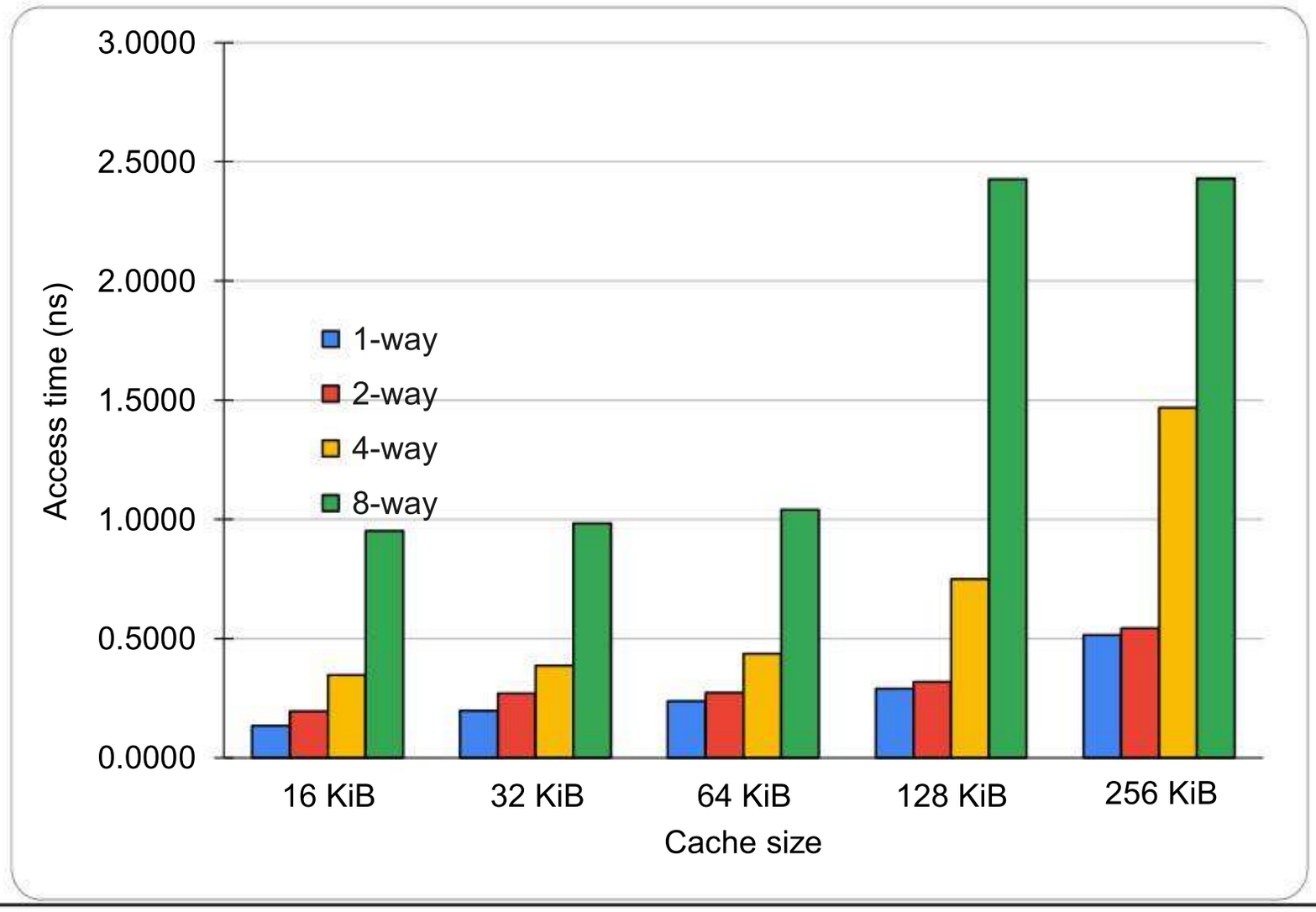


Figure 2.8 Estimated access times generally increase as cache size and associativity are increased These data come from a new CACTI model for a FIN-FET—based SRAM with 14 nm, 0.8 V technology (see Ravipati, et. al., [2022] for details). This data assumes a single bank and 64-byte blocks. The assumptions about cache layout and the complex trade-offs between interconnect delays (that change with the number of ways) and the cost of tag checks and multiplexing lead to high access times, especially for 8-way set associativity. The H-tree used to lay out the cache blocks probably increases the estimated access times for larger associativities, which require more blocks to be connected to the H-tree.

One approach to determining the impact on hit time and power consumption in advance of building a chip is to use CAD tools. CACTI is a program to estimate the access time and energy consumption of alternative cache structures on CMOS microprocessors within 10% of more detailed CAD tools. For a given minimum feature size, CACTI estimates the hit time of caches as a function of cache size, associativity, number of read/write ports, and more complex parameters. Figure 2.8 shows the estimated impact on hit time as cache size and associativity are varied. Of course, these estimates depend on technology as well as the size of the cache, and CACTI must be carefully aligned with the technology; Figure 2.8 shows the relative tradeoffs for one technology.

Example Using the data in Figure B.8 in Appendix B and Figure 2.8, determine whether a 32 KiB four-way set associative L1 cache has a faster memory access time than a 32 KiB two-way set associative L1 cache. Assume the miss penalty to L2 is 15 times the access time for the faster L1 cache. Ignore misses beyond L2. Which has the faster average memory access time?

Answer Let the access time for the two-way set associative cache be 1. Then, for the two-way cache,

$$\begin{aligned}\text{Average memory access time}_{2-\text{way}} &= \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}\\ &= 1 + 0.038 \times 15 = 1.38\end{aligned}$$

For the four-way cache, the access time is 1.4 times longer. The elapsed time of the miss penalty is 15/1.4 = 10.1. Assume 10 for simplicity:

$$\begin{aligned}\text{Average memory access time}_{4-\text{way}} &= \text{Hit time}_{2-\text{way}} \times 1.4 + \text{Miss rate} \times \text{Miss penalty}\\ &= 1.4 + 0.037 \times 10 = 1.77\end{aligned}$$

Clearly, the higher associativity looks like a bad trade-off; however, because cache access in modern processors is often pipelined, the exact impact on the clock cycle time is difficult to assess, and the higher degree of associativity may not impact cycle time. The tradeoff is even more complex in the presence of other cache optimizations.

Energy consumption is also a consideration in choosing both the cache size and associativity, as Figure 2.9 shows. The increase in energy of going from 2-way to 4-way ranges from 1.1 to 1.7 times depending on the cache size.

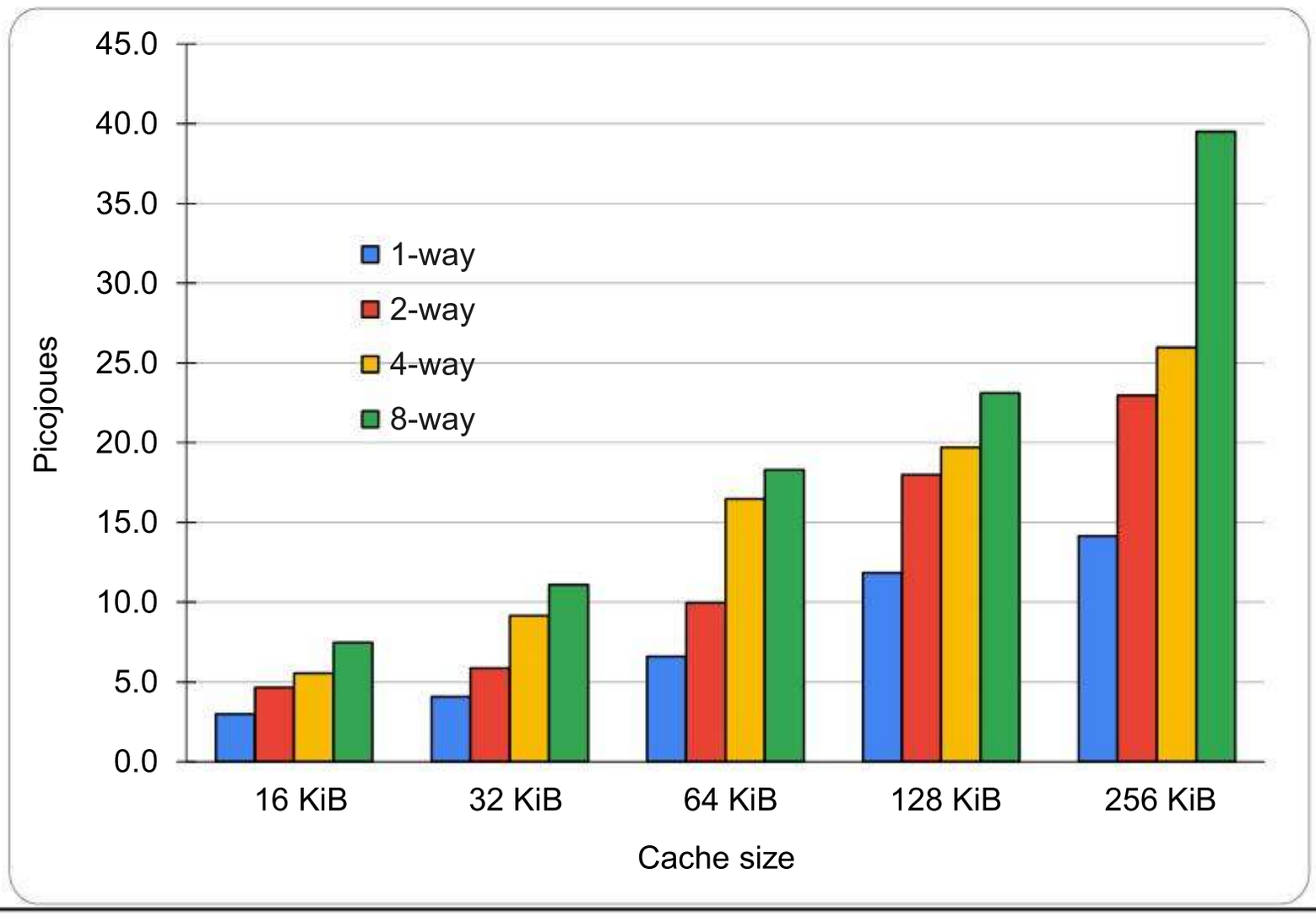


Figure 2.9 Energy consumption per read increases as cache size and associativity are increased. As in the previous figure, CACTI is used for the modeling with the same technology parameters (see Ravipati, et al [2022]). The large penalty for eight-way set associative caches is due to the cost of reading out eight tags and the corresponding data in parallel. Similarly to the previous figure, the H-tree used to lay out the cache blocks probably increases the estimated energy required for larger associativity.

As energy consumption has become critical, designers have focused on ways to reduce the energy needed for cache access. In addition to associativity, the other key factor in determining the energy used in a cache access is the number of blocks in the cache because it determines the number of "rows" that are accessed. A designer could reduce the number of rows by increasing the block size (holding total cache size constant), but this could increase the miss rate, especially in smaller L1 caches.

Way prediction is an approach that uses associativity but 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 in that clock cycle only a single tag comparison is performed in parallel with reading the cache data. A miss results in checking the other blocks for matches in the next clock cycle. Simulations suggest that set prediction accuracy is more than 90% for a two-way set associative cache and 80% for a four-way set associative cache, with better accuracy on I-caches than D-caches. Way prediction yields lower average memory access time for a two-way set associative cache if it is at least 10% faster, which is quite likely. Way prediction was first used in the MIPS R10000 in the mid-1990s. It is popular in processors that use two-way set associativity and was used in several ARM processors, which have four-way set associative caches. For very fast processors, it may be challenging to implement the one-cycle stall that is critical to keeping the way prediction penalty small.

Another way to achieve some of the advantages of set-associativity is the use of victim caches. Victim caches are small caches that hold the contents of a few recently ejected lines. Jouppi (1990) showed that small victim caches (8 entries) could significantly reduce the miss rate for a small direct-mapped L1. With today's larger L1s and L2s, higher L1 associativity and prefetching outperform victim caches. Some recent processors use the LLC effectively as a victim cache, as we will see later.

Second Optimization: Multiple Banks and Ports to Increase the Bandwidth of L1 D-Caches

As processors increased the maximum number of instructions they could execute in a clock cycle, designers needed to increase the number of loads and stores that the D-cache could accept in a cycle. Early multiple-issue processors could execute one load and one store per clock, and more recent processors can execute two loads and two stores every clock. This requires the data cache to accept four references per clock cycle! (As discussed in Section 2.5, the instruction cache uses lookahead and prefetching, relying on spatial locality and branch prediction, to keep up with the increased instruction fetch demand.)

Although it makes sense to build multiple read and write ports on the register file, this solution is more expensive in the larger memory of a cache. Hence most designers opt for using multiple memory banks, possibly with limited multiporting.

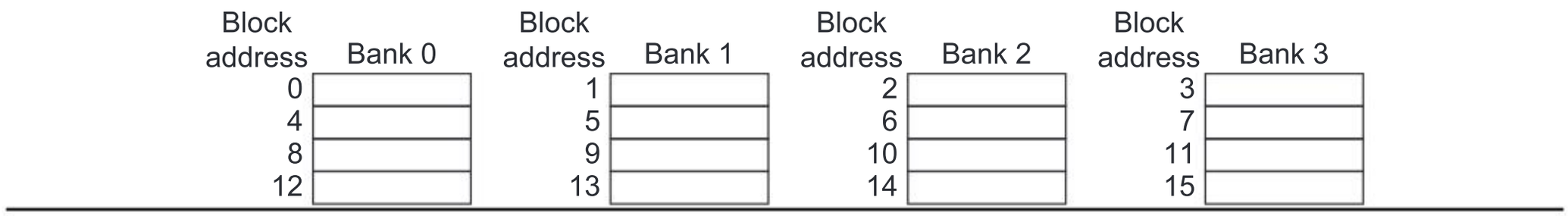


Figure 2.10 Four-way interleaved cache banks using block addressing. Assuming 64 bytes per block, each of these addresses would be multiplied by 64 to get byte addressing.

Banks were originally used to improve performance of main memory and are now used inside modern DRAM chips as well as within caches. The Intel Core i7 has four banks in the L1 data cache (to support up to two memory accesses per clock), while the i9 is dual ported and has eight banks to handle the peak of four memory references per clock.

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, which is 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 2.10 shows this interleaving.

You might think at first that even with banking there would be significant conflicts between references in the data cache, but several circumstances reduce the conflicts. First, the processor cannot sustain four memory references/clock very often; other pipeline stalls and cache misses interfere. Second, even with four references arising at the data cache, data locality means it is likely that two or more of the references are to the same cache block and can be collapsed as a single cache access.

Example Find the probability of a collision assuming eight banks and four references, where 1, 2, or 3 references are to the same block.

Answer If all four references arising at the data cache are to separate blocks, the probability of no collisions with eight banks is given by the probability that references 2, 3, and 4 do not reference the same bank as an earlier one:

$$\frac{7}{8} \times \frac{6}{8} \times \frac{5}{8} = 41\%.$$

If any two references are to the same cache block (and two are distinct) the probability is

$$\frac{7}{8} \times \frac{6}{8} = 66\%.$$

If only two banks are referenced, the probability of a collision drops to 12.5% (7/8). When a collision occurs at a bank, one of the memory references is delayed while the others proceed.

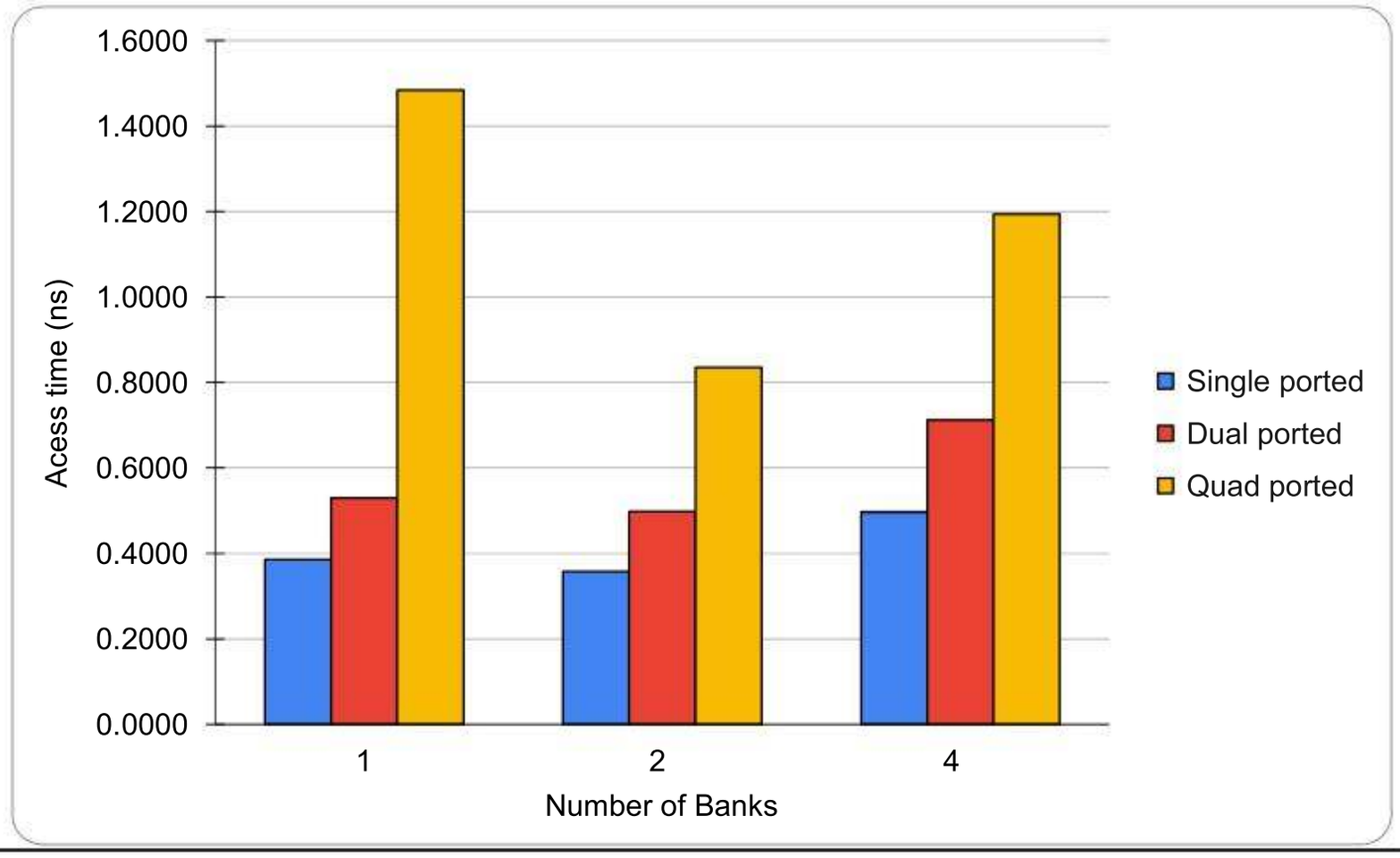


Figure 2.11 CACTI estimated access times for a 32 KiB, 4-way set associative cache with 32 B blocks varying the number of banks and ports.

Of course, any cycle in which there are not four memory references ready for the cache also lowers the probability of a collision. Dual porting also increases the bandwidth without the need to resolve conflicts. A four-banked cache with dual ports (on all four banks) has a much lower probability of a collision since three references to the same bank are needed for a collision.

Dual porting always increases access time, while the impact of multiple banks depends on other factors such as associativity and cache size. How do these possibilities for increasing L1 cache bandwidth compare in terms of access time? Figure 2.11 shows the CACTI estimated access time for 1, 2, and 4 banks with 1, 2, and 4 ports.

Example For simplicity, assume any collision has the effect of adding 1 ns to the cache access time (even if there are four collisions). Using the data from Figure 2.11 and assuming four simultaneous references, which of the following designs has the lowest access time:

- Single ported, 4 banks
- Dual ported, 2 banks (both banks are dual ported)
- Quad ported

Answer First, consider he single-ported case; the probably of a collision is $1 - \frac{3}{4} \times \frac{2}{4} \times \frac{1}{4} = 90\%$.

$$Access\ time_{single\ ported} = 0.5\ ns + 0.9 * 1\ ns = 1.4\ ns.$$

For the dual-ported case, the probability of a collision is 37.5%. The interested reader can verify this by examining all 16 possible combinations of accesses and seeing that only 6 of them have 3 hits to the same bank.

$$Access\ time_{dual\ ported} = 0.5\ ns + .375 * 1\ ns = 0.875\ ns.$$

Finally, the quad-ported case has one bank and an access time of 1.5ns. For these parameters (and penalties for a conflict) the dual-ported, 2-bank solution is best.

Third Optimization: Reducing the Miss Rate With Better Replacement Policies

With direct map caches, the replacement policy is trivial since a block can only go one place. With greater associativity, the cache system must choose which element in a set to replace when a miss occurs. Choosing wisely may reduce the miss rate especially with more associativity, since greater associativity provides more choices for which block to replace. Choosing wisely, however, requires some additional bookkeeping for each cache block. For the L1 cache, this is difficult, since the duty cycle of the cache is so high. Thus L1 caches often use random replacement, or a simple approximation. For L2 and L3 caches, the duty cycle is lower, the miss penalty is higher, and thus an optimized replacement strategy makes sense.

In fact, there is an optimal policy, called Belady's MIN policy. MIN replaces the block that will be used furthest in the future. Unfortunately, as Yogi Barra once observed: "It's tough to make predictions, especially about the future." Nonetheless, MIN provides both a way of thinking about the design of replacement strategies, as well as a benchmark against which we can measure a practical algorithm. Another commonly proposed scheme is LRU: Least Recently Used, which replaces the block that was used furthest in the past. Accurate LRU is hard to implement because it requires keeping an ordering among all the blocks in a given set.

Instead, pseudo-LRU (or approximate LRU) schemes that require simpler updates are used. One simple pseudo-LRU scheme is NRU (Not Recently Used). NRU keeps one bit per set element, which indicates that this block was not recently used, and that bit is set to off whenever the corresponding element is accessed. When a miss occurs, the block chosen for replacement is one of those whose NRU bit is on, and then all the NRU bits are set to on, except for the one associated with the newly replaced block. If multiple blocks in a set have the

NRU bit on (meaning all or none were recently referenced), a random element in the set is replaced. NRU is simple to implement because once a reference is matched to a set only one bit needs to be written. NRU has been used in the L2 and LLC of several earlier Intel processors and is often used for L1 caches.

Another inexpensive way to approximate LRU is a generalization of the NRU scheme using more bits per block organized as a counter to try to predict the block that will be used furthest in the future. For example, we can use 2 bits per block operating as a saturating counter, which has been called the *recently referenced prediction counter*. On a cache hit, the counter is set to 0 for that block. On a cache miss, if any block has a counter value of 3, we replace that block and set the counter to 2. The value 2 represents a block that may or may not be reused shortly. If no block has a counter set to 3, all the counters in the set are incremented, until at least one block has a counter of 3, and that block is then replaced. The key insight behind this scheme is that it attempts to distinguish between streaming accesses that use a block just once and those that use it repeatedly.

Jaleel et al., [2010] simulate a variety of such reference prediction replacement schemes, including NRU, LRU, and the 2-bit recently referenced scheme. For a 2 MB 16-way L2 (roughly what is in an Intel i7 single core), the NRU scheme performs about 1% worse than LRU, while the recently referenced, 2-bit predictor performs about 5% better than LRU. In a multicore, multiprogrammed setting with four cores and an 8 MB LLC, the improvement for the 2-bit scheme is 7%, likely because the multiprogrammed activity causes more single use access to the LLC, and the 2-bit scheme handles these streaming accesses better than a 1-bit scheme. We can extend this scheme to a n-bit counter: as the value of n grows the scheme approximates a true LRU scheme. By weighting the counter, as the 2-bit scheme does, we can favor those blocks that have at least one hit after a miss and are thus less likely to be single-access streamed data.

There have been many proposals for more sophisticated replacement policies that attempt to choose more wisely than an LRU-based scheme, essentially trying to model working set behavior, as is done for virtual memory by the operating system. Some of these studies show significant improvements and may be candidates for implementing in future LLCs, if they can be implemented with sufficiently low overhead and still outperform that weighted approximate LRU schemes.

Fourth Optimization: Multibanked L2 and L3 Caches to Decrease Power and Latency, and Increase Bandwidth

In L1 data caches multiple banks are used primarily to get greater load/store bandwidth; they also provide greater refill bandwidth for nonblocking cache misses, an optimization we cover shortly. In the larger L2 and L3 caches multiple banks are a way to reduce power consumption (as they did for DRAM), reduce latency, and increase bandwidth for responding to nonblocking L1 or L2 misses. The L2 in the Intel Core i9 has eight banks, while Arm Cortex processors have

used L2 caches with 1−4 banks. Intel multicores distribute the LLC among the cores, so the number of banks grows with the core count.

Fifth Optimization: Nonblocking Caches to Increase Cache Bandwidth

For pipelined computers that allow out-of-order execution (discussed in Chapter 3), 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* increases the potential benefits of such a scheme by allowing the data cache to continue to supply cache hits during a miss. This "hit under miss" optimization reduces the effective miss penalty by being helpful during a miss instead of stalling the processor completely. 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; all high-performance processors (such as the recent Intel and AMD processors) usually support both, whereas many lower-end processors provide only limited nonblocking support in L2.

To examine the effectiveness of nonblocking caches in reducing the cache miss penalty, Farkas and Jouppi (1994) did a study assuming 8 KiB caches with a 14-cycle miss penalty (appropriate for the early 1990s). They observed a reduction in the effective miss penalty of 20% for the SPECINT92 benchmarks and 30% for the SPECFP92 benchmarks when allowing one hit under miss. Li et al. (2011) updated this study to use a multilevel cache, more modern assumptions about miss penalties, and the larger and more demanding SPECCPU2006 benchmarks. The study was done assuming a model based on a single core of an Intel i7 (which can generate two data memory requests per clock at peak speed) running the SPECCPU2006 benchmarks. Figure 2.12 shows the reduction in data cache access latency when allowing 1, 2, and 64 hits under a miss; the caption describes further details of the memory system. The larger caches and the addition of an L3 cache since the earlier study have reduced the benefits, with the SPECINT2006 benchmarks showing an average reduction in cache latency of about 9% and the SPECFP2006 benchmarks about 12.5%.

In a nonblocking cache a cache miss does not necessarily stall the processor. Thus it is difficult to judge the impact of any single miss and to calculate the average memory access time. The effective miss penalty is not the sum of the miss times but the nonoverlapped time that the processor is stalled. The benefit of nonblocking caches 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 can hide much of the miss penalty of an L1 data cache miss that hits in the L2 cache but are less capable of hiding

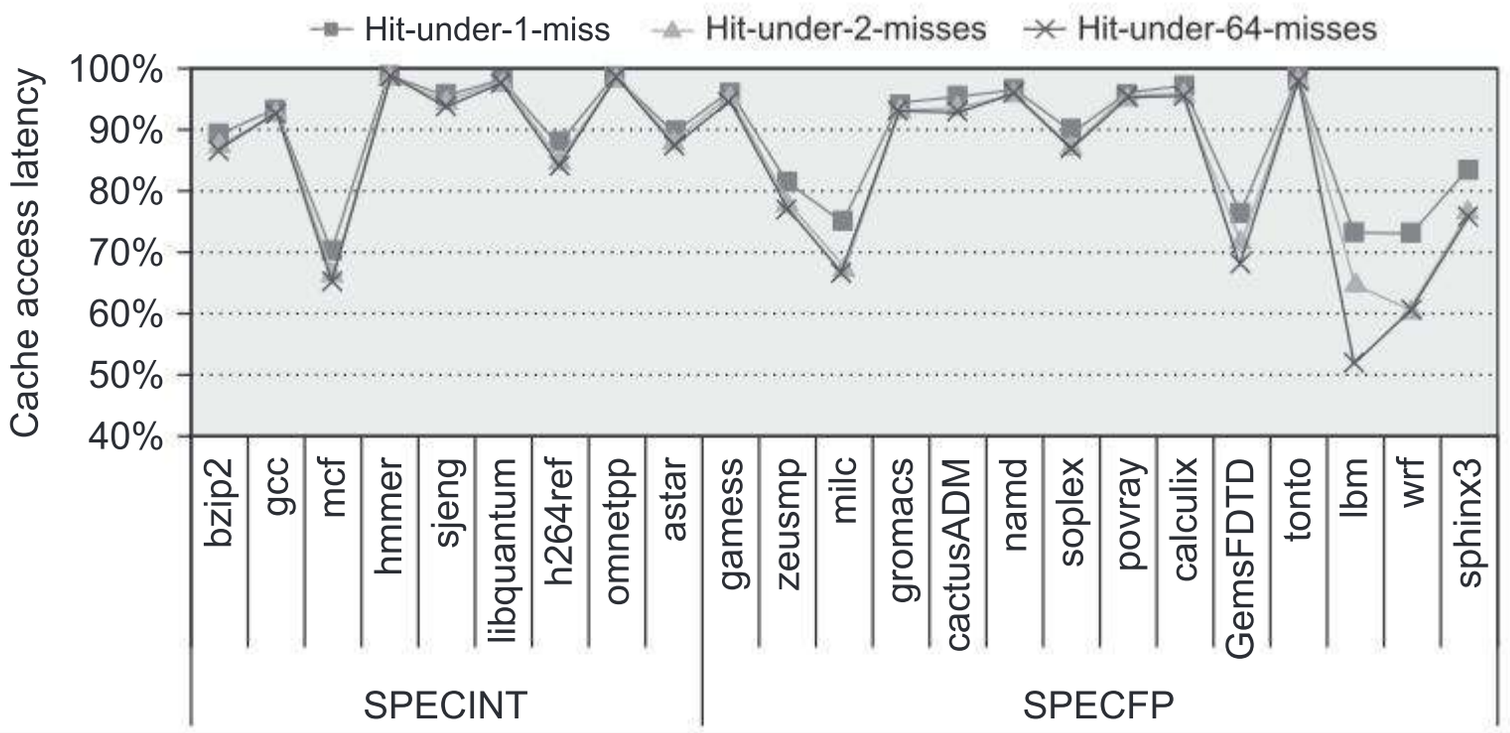


Figure 2.12 The effectiveness of a nonblocking cache is evaluated by allowing 1, 2, or 64 hits under a cache miss with 9 SPECINT (on the left) and 9 SPECFP (on the right) benchmarks. The data memory system modeled after the Intel i7 consists of a 32 KiB L1 cache with a four-cycle access latency. The L2 cache (shared with instructions) is 256 KiB with a 10-clock cycle access latency. The L3 is 2 MiB and a 36-cycle access latency. All the caches are eight-way set associative and have a 64-byte block size. Allowing one hit under miss reduces the miss penalty by 9% for the integer benchmarks and 12.5% for the floating point. Allowing a second hit improves these results to 10% and 16%, and allowing 64 results in little additional improvement.

most of the penalty for a lower-level cache miss. Deciding how many outstanding misses to support depends on a variety of factors:

- The temporal and spatial locality in the miss stream, which determines whether a miss will initiate a new access to a lower-level cache or to memory.
- The bandwidth of the responding memory or cache.
- If the processor allows more outstanding misses at the lowest level of the cache (where the miss time is the longest) it must support at least that many misses at a higher level, because the miss must begin at the highest-level cache.
- The latency of the memory system.

The following simplified example illustrates the key idea.

Example Assume a main memory access time of 36 ns and a memory system capable of a sustained transfer rate of 16 GiB/s. If the block size is 64 bytes, what is the maximum number of outstanding misses we need to support, assuming that we can maintain the peak bandwidth given the request stream and that accesses never conflict? If the probability of a reference colliding with one of the previous four is 50%, and we assume that the access has to wait until the earlier access completes, estimate the number of maximum outstanding references. For simplicity, ignore the time between misses.

Answer In the first case, assuming that we can maintain the peak bandwidth, the memory system can support $(16 \times 10)^9/64 = 250$ million references per second. Because each reference takes 36 ns, we can support $250 \times 10^6 \times 36 \times 10^{-9} = 9$ references. If the probability of a collision is greater than 0, then we need more outstanding references, because we cannot start work on those colliding references; the memory system needs more independent references, not fewer! To approximate, we can simply assume that half the memory references do not have to be issued to the memory. This means that we must support twice as many outstanding references, or 18.

In Li, Chen, Brockman, and Jouppi's study they found that the reduction in CPI for the integer programs was about 7% for one hit under miss and about 12.7% for 64. For the floating-point programs, the reductions were 12.7% for one hit under miss and 17.8% for 64. These reductions track closely the reductions in the data cache access latency shown in Figure 2.12.

Implementing a Nonblocking Cache

Although nonblocking caches have the potential to improve performance, they are nontrivial to implement. Two initial types of challenges arise: arbitrating contention between hits and misses and tracking outstanding misses so that we know when loads or stores can proceed. Consider the first problem. In a blocking cache misses cause the processor to stall and no further accesses to the cache will occur until the miss is handled. In a nonblocking cache, however, hits can collide with misses returning from the next level of the memory hierarchy. If we allow multiple outstanding misses, which almost all recent processors do, it is even possible for misses to collide. These collisions must be resolved, often by giving priority to hits over misses and then by ordering colliding misses (when they occur).

The second problem arises because we need to track multiple outstanding misses. In a blocking cache we always know which miss is returning, because only one can be outstanding. In a nonblocking cache this is rarely true. At first glance, you might think that misses always return in order so that a simple queue could be kept matching a returning miss with the longest outstanding request. Consider, however, a miss that occurs in L1. It may generate either a hit or miss in L2; if L2 is also nonblocking, then the order in which misses are returned to L1 will not necessarily be the same as the order in which they originally occurred. Multicore and other multiprocessor systems that have nonuniform cache access times (called NUCA) also create this complication.

When a miss returns, the processor must know which load or store caused the miss so that instruction can now go forward, and it must know where in the cache the data should be placed (as well as the setting of tags for that block). In recent processors this information is kept in a set of registers, typically called the *Miss Status Handling Registers (MSHRs)*. If we allow *n* outstanding misses, there will

be *n* MSHRs, each holding the information about where a miss goes in the cache and the value of any tag bits for that miss, as well as the information indicating which load or store caused the miss (in the next chapter, you will see how this is tracked). Thus, when a miss occurs, we allocate an MSHR for handling that miss, enter the appropriate information about the miss, and tag the memory request with the index of the MSHR. The memory system uses that tag when it returns the data, allowing the cache system to transfer the data and tag information to the appropriate cache block and "notify" the load or store that generated the miss that the data is now available and that it can resume operation. Nonblocking caches clearly require extra logic and thus have some cost in energy. It is difficult, however, to assess their energy costs exactly because they may reduce stall time, thereby decreasing execution time and resulting energy consumption.

In addition to the preceding issues, multiprocessor memory systems, whether within a single chip or on multiple chips, must also deal with complex implementation issues related to memory coherency and consistency. Furthermore, because cache misses are no longer atomic (because the request and response are split and may be interleaved among multiple requests), there are possibilities for deadlock. We will return to these issues in Chapter 5, and Section I.7 in online Appendix I deals with these issues in more detail.

Sixth 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 because 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.

However, 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. 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. For example, for SPECint2006 running on the i7 6700, which uses early restart and critical word first, there is more than one reference made to a block with an outstanding miss (1.23 references on average with a range from 0.5 to 3.0). We explore the performance of an advanced memory hierarchy in more detail in Section 2.6.

Seventh 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 next are found in many modern compilers.

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 in which 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. For example, if `x` is a two-dimensional array of size [5000,100] allocated so that `x[i,j]` and `x[i,j + 1]` are adjacent (an order called row major because the array is laid out by rows), then the two pieces of the following code show how the accesses can be optimized:

```
/* 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 two-dimensional *blocks*. The goal is to maximize accesses to the data loaded into the cache before the data are replaced. The following code example, 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 2.13 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.)

```
/* 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;
           };
```

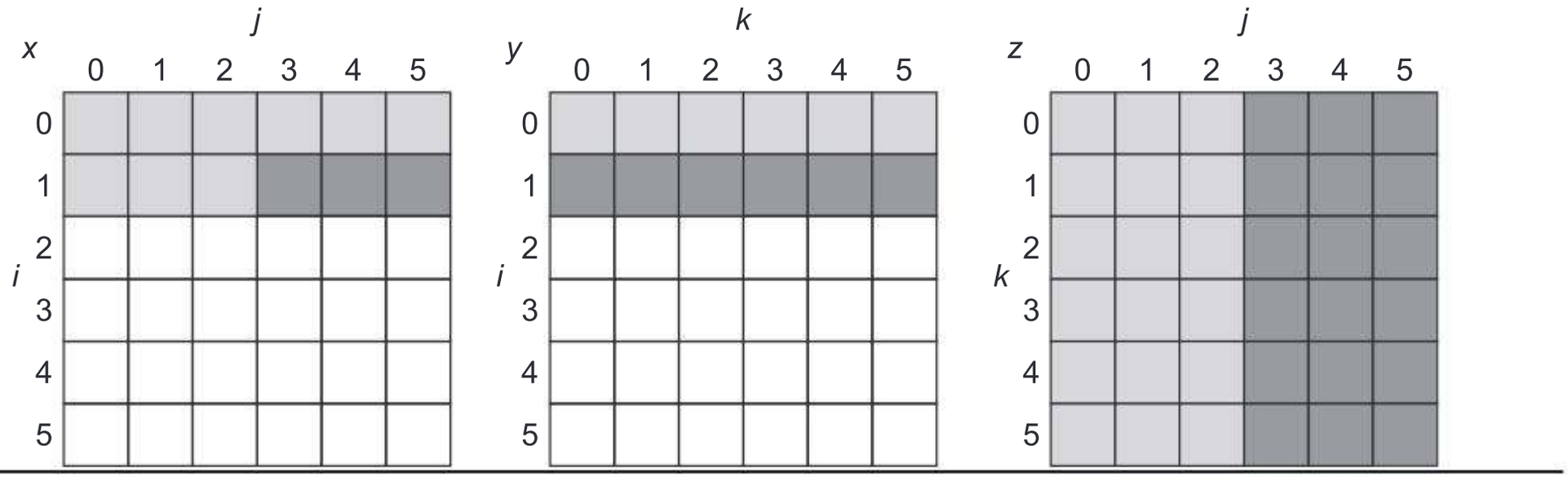


Figure 2.13 A snapshot of the three arrays `x`, `y`, **and** `z` **when** $N = 6$ **and** `i = 1`**.** The age of accesses to the array elements is indicated by shade: white means not yet touched, light means older accesses, and dark means newer accesses. The elements of `y` and `z` are read repeatedly to calculate new elements of `x`. The variables `i`, `j`, and `k` are shown along the rows or columns used to access the arrays.

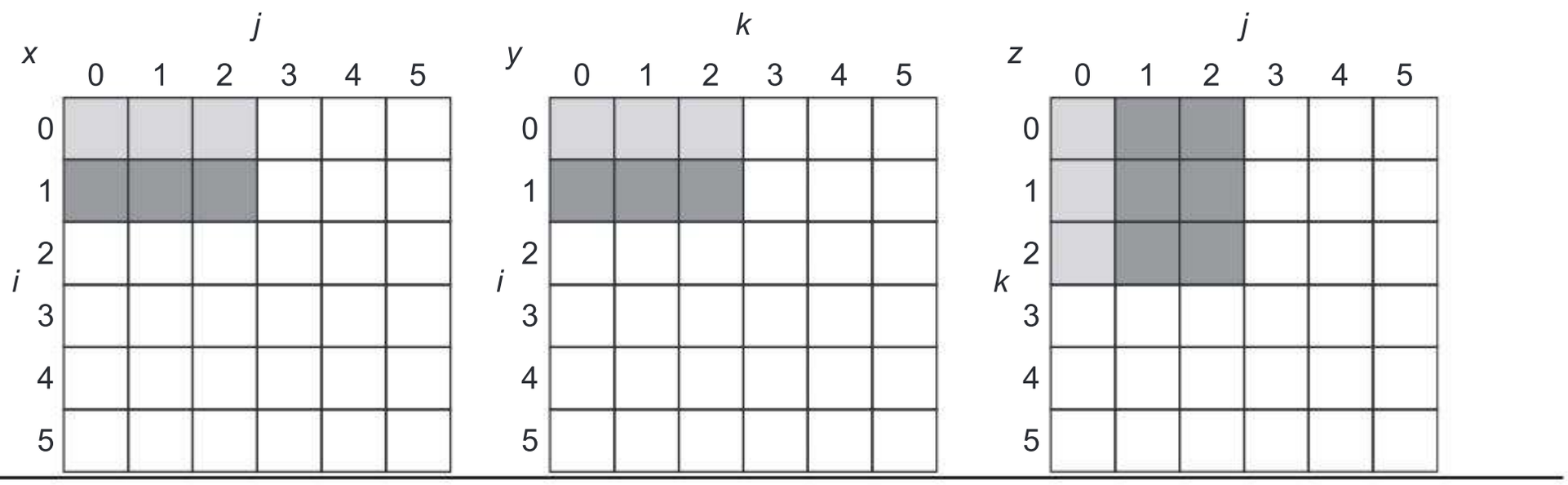


Figure 2.14 The age of accesses to the arrays x, y, and z when $B = 3$**.** Note that, in contrast to Figure 2.13, a smaller number of elements are accessed.

Figure 2.14 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 an approximate factor of `B`. Therefore blocking exploits a combination of spatial and temporal locality, because `y` benefits from spatial locality and `z` benefits from temporal locality. Although our example uses a square block (BxB), we could also use a rectangular block, which would be necessary if the matrix were not square.

We have aimed at reducing cache misses, but 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.

As we shall see in Section 4.8 of Chapter 4, cache blocking is necessary to get good performance from cache-based processors running applications using matrices as the primary data structure.

Eighth 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 in 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 KiB four-way set associative caches, one for instructions and the other for data.

The Intel Core i7 and i9 support hardware prefetching into both L1 and L2, with the most common case of prefetching being accessing the next line. In applications with low L1 and L2 miss rates prefetching cannot offer much improvement. Even with significant L1 or L2 miss rates, prefetches must be used carefully, since those "bad" prefetches might replace useful cache blocks and will consume cache or memory bandwidth. The Xeon Skylake-SP uses a hardware prefetch very similar to Sapphire Rapids; it has four data prefetchers: L1 Data cache unit prefetcher *(DCUI),* L1 Data cache instruction pointer stride prefetcher *(DCUP),* L2 Data cache spatial prefetcher *(L2A),* and L2 Data cache streamer *(L2P)*. In the more memory-intensive SPECCPU2017 benchmarks the L2P prefetcher is the most important, generating 70% of the CPI improvement. Notice that the L2 prefetchers will fetch from either L3 or memory and place the result directly into L2 (and not in L3 if the prefetch went to memory).

Figure 2.15 shows the overall performance improvement (measured as reduction in CPI) for a subset of SPECCPU2017 programs when hardware prefetching is turned on for a Skylake-SP, a multicore Xeon processor. Notice that the results vary widely, from two benchmarks that perform slightly worse, to five that get an improvement of about 10% or less, to two benchmarks that more than double performance. Figure 2.15 also shows the reduction in MPKI (Misses Per K Instructions) for L3, which correlates highly with the reduction in CPI.

Prefetches are not counted as misses. Hence, when the L2P prefetcher fetches into the L2 cache, it can eliminate both L2 misses that hit in L3 and L2 misses that miss in L3. Because L3 misses are roughly four times as long as L2 misses, the reduction in L3 misses generates most of the performance improvement, as

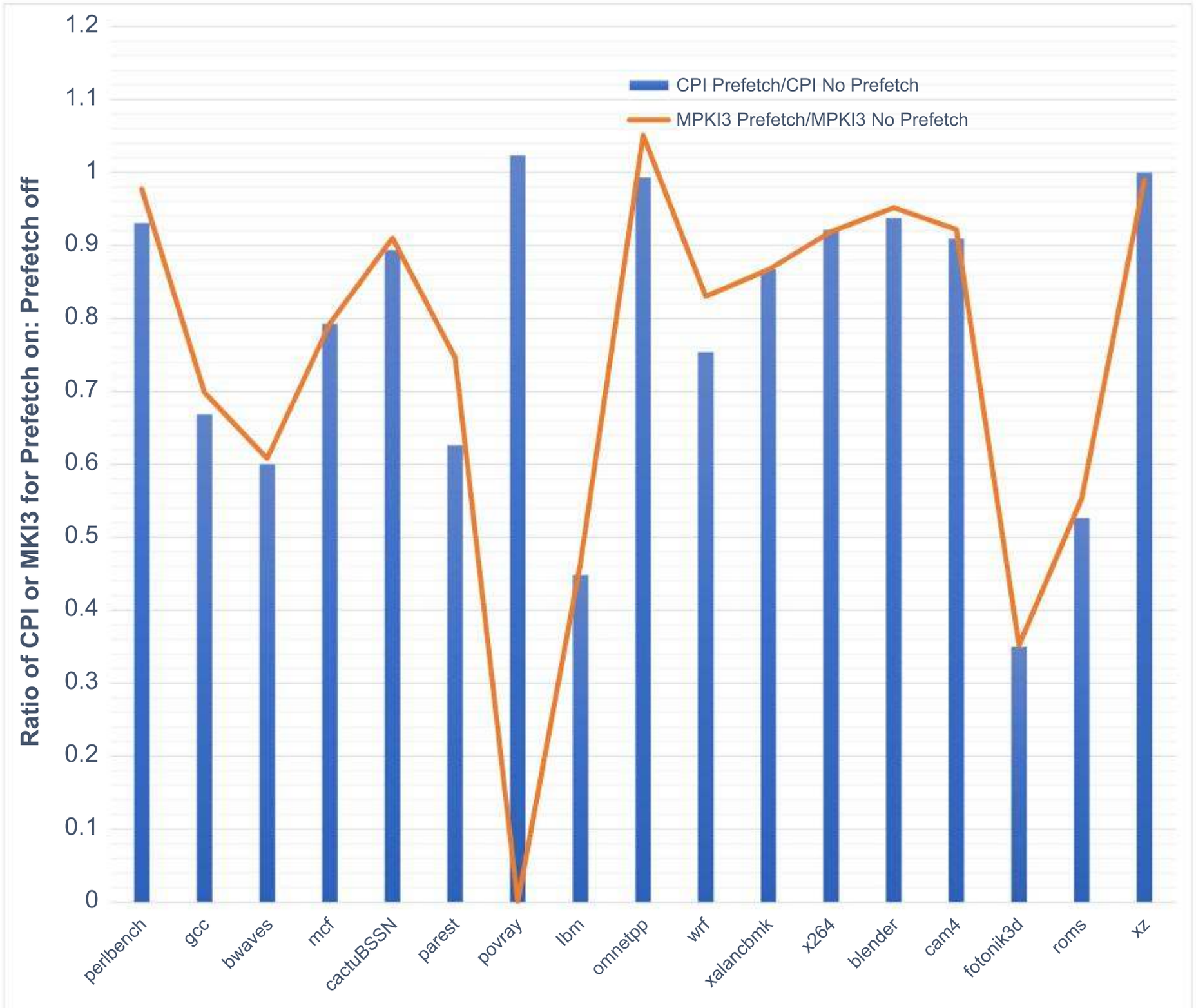


Figure 2.15 Ratio of the CPI and MKPI3 (MPKI for L3) on Skylake-SP with prefetching turned on and off with a 7 MiB L3 (4-way set associative) and the 15 most intensive memory benchmarks in SPECCPU2017. The data collected is for the most memory-intensive input for each benchmark (when there are multiple inputs). Note that except for povray the CPI improvement closely tracks the reduction in MKPI3; in the case of povray the MKPI3 is close to 0, so all the benefit comes from prefetches from L3 into L2. In addition to being the most effective overall, the L2P prefetcher generates 90% of the gain for the seven most improved benchmarks. The data in this figure comes from Navarro-Torres, et. al. [2019].

Figure 2.15 shows. Many or all the L3 misses that are eliminated would occur if the prefetches were removed, but because the prefetch scheme is not perfect, it can also cause some extraneous misses. We can see this by looking at the total data traffic requested by L3 with and without prefetching. With prefetching turned on the L3 cache requests 19% more data than with prefetching turned off. In a few cases (e.g., bwaves and lbm), the prefetcher appears highly accurate and L3 traffic grows by less than 1%. When prefetching works well, its impact on power is negligible (or even slightly positive). When prefetched data are not used or useful data are displaced, prefetching will have a significant negative impact on power.

Ninth 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:

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

Either of these can be *faulting* or *nonfaulting*; that is, the address does or does not cause an exception for virtual address faults and protection violations. Nonfaulting prefetches simply turn into no-ops if they would normally result in an exception, 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 because 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 H) 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 following code, 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 KiB 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 because 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. Because `a` has 3 rows and 100 columns, its accesses will lead to $3 \times (100/2)$, or 150 misses.

The array `b` does not benefit from spatial locality because the accesses are not in the order in which 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 because of `b` will be for `b[j + 1][0]` accesses when $i = 0$, and also the first access to `b[j][0]` when $j = 0$. Because `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 for 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 preceding code 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 preceding example. 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. Because 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. Now 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 of 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.

Many processors support instructions for cache prefetch, and high-end processors (such as the Intel Core i7 and i9) often also do some type of automated prefetch in hardware.

Tenth Optimization: Multiple Memory Buses and Memory Modules to Increase Bandwidth

High-speed processors on the desktop (e.g., Intel i9 and AMD Ryzen), in servers (Intel Xeon and IBM Power series), and domain-specific processors for graphics or deep learning often use multiple memory channels to increase the available bandwidth to memory. Because multiple memory channels allow independent accesses, they can also prevent one access from stalling another, effectively reducing the miss penalty.

Once we have multiple memory buses, we can consider structuring a system to use a combination of SDRAM and HBM, by designing two different memory channels with different bus structures. This mix allows us to utilize the speed and bandwidth advantages of HBM memory (which requires a different signaling convention from normal SDRAM) without limiting the total size of memory to a few HBM stacks. Using different types of memory in this fashion does lead to a NUMA (NonUniform Memory Architecture) and taking advantage of this capability requires that critical data be placed in the faster HBM memory. For domain-specific processors, this is usually done under software control and typically requires a domain-specific language to guide the software in allocating and moving data between memories (see Chapter 7).

The alternative to an explicitly managed NUMA is to use the HBM memory as a cache, which the newest Intel and AMD processors also support.

An HBM cache (essentially a new L4 or LLC) could be 10 or more times larger than the largest on-chip L3 caches. Using such large DRAM-based caches, however, raises an issue: where do the tags reside? The answer depends on the number of tags. Suppose we were to use a 64B block size; then a 1 GiB L4 cache requires 96 MiB of tags—roughly as much SRAM as a large multicore has for all its caches! Increasing the block size can reduce the number of tags, but very large blocks can generate a higher miss rate, especially for conflict and consistency misses.

We could have a smaller block size and more tags, if we stored the tags for the LLC in the HBM, rather than on chip. At first glance, this seems unworkable, because it requires two accesses to DRAM for each L4 access: one for the tags and one for the data itself. Loh and Hill (2011) proposed a clever solution to this problem (which we call the L-H scheme): place the tags and the data in the same row in the HBM SDRAM. Although opening the row (and eventually closing it) takes a large amount of time, the CAS latency to access a different part of the row is about one-third of the new row access time. Thus we can access the tag portion of the block first, and if it is a hit, then use a column access to choose the correct word.

Qureshi and Loh (2012) proposed an improvement called an *alloy cache* that reduces the hit time. An *alloy cache* molds the tag and data together and uses a direct mapped cache structure. This allows the LLC access time to be reduced to a single HBM cycle by directly indexing the HBM cache and doing a burst

transfer of both the tag and data. The alloy cache reduces hit time by more than a factor of 2 versus the L-H scheme, in return for an increase in the miss rate by a factor of 1.1 to 1.2.

Unfortunately, in both schemes, misses require two full DRAM accesses: one to get the initial tag and a follow-on access to the main memory (which is even slower). Figure 2.16 shows the speedup obtained using 10 memory-intensive SPECCPU2006 benchmarks run in SPECRate fashion. The alloy cache adds a small miss predictor to try to avoid the initial HBM access when it is predicted to miss. With this addition, the alloy cache approach outperforms the L-H scheme and even the impractical SRAM tags, because the combination of a fast access time for the miss predictor and good misprediction results leads to a shorter time to predict a miss and thus a lower miss penalty. The alloy cache performs close to the Ideal case, an LLC with perfect miss prediction and minimal hit time.

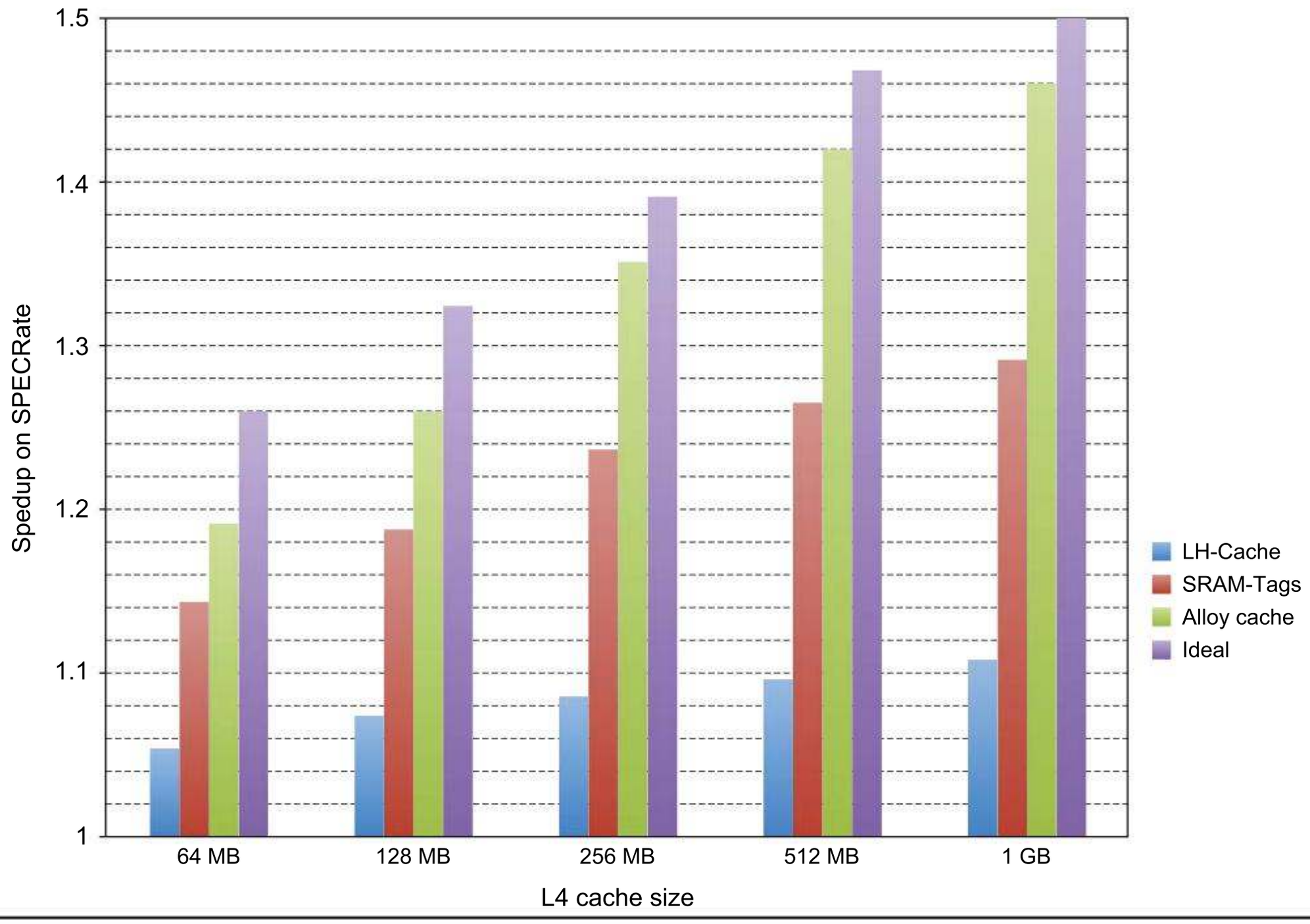


Figure 2.16 Performance speedup running the SPECrate benchmark for the Loh-Hill scheme, an SRAM tag scheme, an alloy cache, and an ideal LLC (Ideal); a speedup of 1 indicates no improvement with the L4 LLC, and a speedup of 2 would be achievable if L4 were perfect and took no access time. The 10 most memory-intensive SPECCPI2006 benchmarks are used, with each benchmark run eight times. The accompanying miss prediction scheme is used. The Ideal case assumes that only the 64-byte block requested in LLC needs to be accessed and transferred and that prediction accuracy for LLC is perfect (i.e., all misses are known at zero cost).

HBM is likely to have widespread use in a variety of different configurations, from containing the entire memory system for some high-performance, special-purpose systems to using a portion of the memory for designs with multiple memory buses to use as an LLC cache.

Memory Hierarchy 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 2.17 summarizes these

Technique	Hit time	Band-width	Miss penalty	Miss rate	Power consumption	Hardware cost/ complexity	Comment
Small and simple caches	+			–	+	0	Trivial; widely used
Way-predicting caches	+				+	1	Used in Pentium 4
Pipelined & banked caches	–	+				1	Widely used
Nonblocking caches		+	+			3	Widely used
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, but many compilers handle common linear algebra calculations
Hardware prefetching of instructions and data			+	+	–	2 instr., 3 data	Most provide prefetch instructions; modern high-end processors also automatically prefetch in hardware
Compiler-controlled prefetching			+	+		3	Needs nonblocking cache; possible instruction overhead; in many CPUs
HBM as additional level of cache	+/–		–	+	+	3	Depends on new packaging technology. Effects depend heavily on hit rate improvements

Figure 2.17 Summary of advanced cache optimizations showing impact on cache performance, power consumption, 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.

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.

2.4 Virtual Memory and Protection

Section B.4 in Appendix B describes the key concepts in virtual memory. Recall that virtual memory allows the physical memory to be treated as a cache of secondary storage (which may be either hard disk or solid state). Virtual memory moves pages between the two levels of the memory hierarchy, just as caches move blocks between levels. Likewise, TLBs act as caches on the page table, eliminating the need to do a memory access every time an address is translated. Virtual memory also provides separation between processes that share one physical memory but have separate virtual address spaces. Readers should ensure that they understand both functions of virtual memory before continuing.

In this section we focus on additional issues in protection and privacy between processes sharing the same processor. Security and privacy are two of the most vexing challenges for information technology in 2025. Electronic burglaries, often involving lists of credit card numbers, are announced regularly, and it's widely believed that many more go unreported. Of course, such problems arise from programming errors that allow a cyberattack to access data it should be unable to access. Programming errors are a fact of life, and with modern complex software systems, they occur with significant regularity. Therefore both researchers and practitioners are looking for improved 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 how side-channel attacks can leak information across protection boundaries.

Protection via Virtual Memory

Page-based virtual memory, including a TLB that caches page table entries, is the primary mechanism that protects processes from each other. Sections B.4 and B.5 in Appendix B review virtual memory, including a detailed description of protection via segmentation and paging in the 80x86. This section acts as a quick review; if it's too quick, please refer to the denoted Appendix B sections.

Multiprogramming, where several programs running concurrently share a computer, has 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*.

1. 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 a minimum, the architecture must do the following: 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 a user/supervisor mode bit, 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 A 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 KiB, 16 KiB, or larger, 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 (PTE). 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. Because 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 address. This special address translation cache is traditionally referred to as a *translation lookaside buffer* or 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. TLB misses can be handled by reading the address information from the page tables. As page table structures became more complex to deal with much larger address spaces, processors introduced hardware to handle TLB misses, in a way analogous to, but more complex than the handling of cache misses. Appendix L covers the topic of more advanced address translation.

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. Because 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. As the next section shows, however, the memory system can leak protected information, even if it correctly enforces functional separation. Chapter 6 introduces techniques (including virtual machines and enclaves) that enable programs to protect highly sensitive data even from an intentionally malicious operating system. Unfortunately, the side-channel attacks described next can sometimes circumvent these more powerful protection mechanisms.

Side-Channel Attacks on the Memory System

Side channel memory attacks all work by perturbing something in the memory system and observing the effects by monitoring performance, using timers or performance counters provided by the processor. Side-channel attacks work because multiple independent processes share hardware that is not part of the instruction set state (e.g., caches) and thus are not protected by the normal mechanisms that virtual memory and the operating system use to protect processes from one another.

Consider a simple piece of code:

```
if (x <= 0) {access P}
else {access Q}
```

Suppose we want to spy on this code using a side-channel attack to determine if x <= 0. There are three steps to this process:

1. Prime the cache: in this step the spy ensures that the cache lines where the two parts of the if-then-else would be cached are written with information from the spy.
2. Let the victim's code run once, which will cause a miss for either the code in the then-clause or the else-clause.

3. Probe the cache by accessing the P and Q, measuring whether an access hits or misses (this can be done by an accurate timer or a performance counter that counts cache misses). If x <=0 then we know that the original cache lines corresponding to P will have been replaced, and if the spy accesses those lines, it will encounter a miss. Knowing which cache lines hit or miss tells us which code path the victim took and hence leaks information about the value of x.

Although this simple example has code that differentiates the key facts we want to know (namely x <=0), there are many ways in which information about data values or execution paths can be deduced from program behavior. The spy may need many repetitions of a prime-probe sequence to get detailed information on the memory contents of another process, but these can be executed quite quickly. The key to this type of attack is that the cache is shared between the spy and victim, even if the virtual memory system prevents direct access by the spy to the victim's address space.

For the prime-probe attack to work, we need to know the location of the victim's code in memory, and we need to ensure that the victim's code is run between the prime and probe step with no intervening actions. If any other code was executed, the cache could be polluted, reducing the effectiveness of the attack significantly. Because switching between the spy and the victim requires overhead (a process switch), the cache could become polluted, reducing the effectiveness of the attack. If the spy and victim are running on separate cores in the same multicore and sharing the LLC, the attack is easier and more effective. As we will see in the next chapter, multithreading makes such attacks even more effective, since it allows the spy and victim to share hardware resources more tightly.

There are several mitigations to this type of side-channel attack. The mitigations dramatically reduce the probability that information can be leaked to the spy. It is virtually impossible to stop all side-channel attacks if any resources are shared, but we can lower the leakage rate significantly, making it practically impossible to spy on the victim. Here are some steps that make it much harder to mount a side-channel attack:

- Randomize the allocation of pages to make it hard to figure out where in the cache a code segment will appear. This solution helps with physically addressed caches, but not virtually addressed caches. Unfortunately, some OS code may be locked down in physical address locations. Alternatively, highly sensitive information can be copied before use in a random location. The key is to break the linkage of cache behavior to known code or data.
- Obfuscate timing. Critical code segments that contain highly sensitive information can insert random, short delays during their execution. An approach like this was used to protect against one of the earliest side-channel password attacks in the 1970s.
- Eliminating any implicit sharing by flushing the caches on a context switch; unfortunately, this solution must be applied to the LLC and is quite costly.

- Partition resources. If the LLC is the place where sharing occurs, we could partition the LLC to have separate segments for processes that are simultaneously active. This approach may have significant performance costs, especially for applications with large memory footprints. Most processors targeted at server environments support cache partitioning.

As we will see in the next chapter the introduction of speculation and multithreading dramatically increases the bandwidth of a side-channel attack (by perhaps a factor of 100), making hardware or software solutions mandatory.

2.5 Cross-Cutting Issues: The Design of Memory Hierarchies

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

Autonomous Instruction Fetch Units

Many processors with out-of-order execution and even some with simply deep pipelines decouple the instruction fetch (and sometimes initial decode), using a separate instruction fetch unit (see Chapter 3). Typically, the instruction fetch unit accesses the instruction cache to fetch an entire block before decoding it into individual instructions; such a technique is particularly useful when the instruction length varies and is required in a processor that can issue more than one instruction per clock. Because the instruction cache is accessed in blocks, it no longer makes sense to compare miss rates to processors that access the instruction cache once per instruction. In addition, the instruction fetch unit may prefetch blocks into the L1 cache; these prefetches may generate additional misses but may actually reduce the total miss penalty incurred (as we saw earlier). Many processors also include data prefetching, which may increase the data cache miss rate, even while decreasing the total data cache miss penalty.

Special Instruction Caches

One of the biggest challenges in superscalar processors is to supply the instruction bandwidth. For designs that translate the instructions into microoperations, such as the most recent Arm and i7 processors, instruction bandwidth demands and branch misprediction penalties can be reduced by keeping a small cache of recently translated instructions. We explore this technique in greater depth in the next chapter.

Speculation and Memory Access

One of the major techniques used in advanced pipelines is speculation, whereby an instruction is tentatively executed before the processor knows whether it is

really needed. Such techniques rely on branch prediction, which if incorrect requires that the speculated instructions are flushed from the pipeline. There are two separate issues in a memory system supporting speculation: protection and performance. With speculation, the processor may generate memory references, which will never be used because the instructions were the result of incorrect speculation. Those references, if executed, could generate protection exceptions. Obviously, such faults should occur only if the instruction is actually executed. In the next chapter we will see how such "speculative exceptions" are resolved. Because a speculative processor may generate accesses to both the instruction and data caches, and subsequently not use the results of those accesses, speculation may increase the cache miss rates. As with prefetching, however, such speculation may lower the total cache miss penalty. The use of speculation, like the use of prefetching, makes it misleading to compare miss rates to those seen in processors without speculation, even when the instruction set architecture (ISA) and cache structures are otherwise identical. Finally, speculation is one of the capabilities that can dramatically increase the leakage rate of a side-channel attack.

Coherency of Cached Data

Data can be found in memory and in the cache. As long as the processor is the sole component 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 we will see, both 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 for 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 issues of coherency and performance for shared data are major topics in Chapter 5.

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 are 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 why processors used write-through.) However, today write-

through is usually found only in first-level data caches backed by an L2 cache that uses write-back. The use of write buffers and write merging create additional complications for memory-mapped I/O. For example, addresses of I/O registers *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 I/O register rather than use multiword writes using a single address. These side effects are typically implemented by marking the pages as requiring nonmerging write-through by the caches.

I/O input requires some extra attention. 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 noncacheable, 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 of these approaches can also be used for output with write-back caches. For high performance in data centers additional methods of integrating I/O into the memory hierarchy may be needed. We explore this topic in Chapter 6.

Protection via Virtual Machines

An idea related to virtual memory that is almost as old is virtual machines (VMs). They were first developed in the late 1960s, and they have remained an important part of mainframe computing over the years. VMs provide a higher level of protection among programs sharing a computer by allowing them to run separate versions of the operating system that are kept independent. VMs are a key technology in today's large-scale data centers and are covered in detail in Chapter 6. VMs often introduce shadow page tables, which increases the cost of a TLB miss since it requires a more complex process to find the correct address mapping. As Appendix L discusses, designers may include special hardware to handle these more complex steps incurred in a TLB miss.

2.6 Putting It All Together: Memory Hierarchies in the ARM Cortex-A53 and Intel Core i9 12900

This section reveals the ARM Cortex-A53 (hereafter called the A53) and Intel Core i9 12900 (hereafter called i9) memory hierarchies and shows the performance of their components on a set of single-threaded benchmarks. We examine the Cortex-A53 first because it has a simpler memory system; we go into more detail for the i9, tracing out a memory reference in detail. This section presumes that readers are familiar with the organization of a two-level cache hierarchy using virtually indexed caches. The basics of such a memory system are explained in detail in Appendix B, and readers who are uncertain of the organization of such a system are strongly advised to review the Opteron example in Appendix B.

Once they understand the organization of the Opteron, the brief explanation of the A53 system, which is similar, will be easy to follow.

The ARM Cortex-A53

The Cortex-A53 is a configurable core that supports the ARMv8A ISA, which includes both 32-bit and 64-bit modes. The Cortex-A53 is delivered as an IP (intellectual property) core. IP cores are the dominant form of technology delivery in the embedded, PMD, and related markets; billions of ARM, MIPS, and RISC-V processors have been created from these IP cores. Note that IP cores are different from the cores in the Intel i7 or AMD Athlon multicores. An IP core (which may itself be a multicore) is designed to be incorporated with other logic (thus it is the core of a chip), including application-specific processors (such as an encoder or decoder for video), I/O interfaces, and memory interfaces, and then fabricated to yield a processor optimized for a particular application. For example, the Cortex-A53 IP core is used in a variety of tablets and smartphones; it is designed to be highly energy-efficient, a key criterion in battery-based PMDs. The A53 core is very popular and capable of being configured with multiple cores per chip for use in high-end PMDs; our discussion here focuses on a single core.

Generally, IP cores come in two flavors. *Hard cores* are optimized for a particular semiconductor vendor and are black boxes with external (but still on-chip) interfaces. Hard cores typically allow parametrization only of logic outside the core, such as L2 cache sizes, and the IP core cannot be modified. *Soft cores* are usually delivered in a form that uses a standard library of logic elements. A soft core can be compiled for different semiconductor vendors and can also be modified, although extensive modifications are very difficult because of the complexity of modern-day IP cores. In general, hard cores provide higher performance and smaller die area, while soft cores allow retargeting to other vendors and can be more easily modified.

The Cortex-A53 can issue two instructions per clock at clock rates up to 1.3 GHz. It supports both a two-level TLB and a two-level cache; Figure 2.18 summarizes the organization of the memory hierarchy. On a miss, the critical word is returned first, and the processor can continue while the miss completes; a memory system with up to four banks can be supported. For a D-cache of 32 KiB and a page size of 4 KiB, each physical page could map to two different cache addresses (since the cache index is 7 bits and the block offset is 6, which is one more than the page offset); such aliases are avoided by hardware detection on a miss as in Section B.3 of Appendix B. Figure 2.19 shows how the 32-bit virtual address is used to index the TLB and the caches, assuming 32 KiB primary caches and a 1 MiB secondary cache with 16 KiB page size.

Performance of the Cortex-A53 Memory Hierarchy

The memory hierarchy of the Cortex-A8 was measured with 32 KiB primary caches and a 1 MiB L2 cache running the SPECInt2006 benchmarks. The

Structure	Size	Organization	Typical miss penalty (clock cycles)
Instruction MicroTLB	10 entries	Fully associative	2
Data MicroTLB	10 entries	Fully associative	2
L2 Unified TLB	512 entries	4-way set associative	20
L1 Instruction cache	8–64 KiB	2-way set associative; 64-byte block	13
L1 Data cache	8–64 KiB	2-way set associative; 64-byte block	13
L2 Unified cache	128 KiB to 2 MiB	16-way set associative; LRU	124

Figure 2.18 The memory hierarchy of the Cortex A53 includes multilevel TLBs and caches. A page map cache keeps track of the location of a physical page for a set of virtual pages; it reduces the L2 TLB miss penalty. The L1 caches are virtually indexed and physically tagged; both the L1 D cache and L2 use a write-back policy defaulting to allocate on write. Replacement policy is LRU approximation in all the caches. Miss penalties to L2 are higher if both a micro-TLB and L1 miss occur. The L2 to main memory bus is 64—128 bits wide, and the miss penalty is larger for the narrow bus.

instruction cache miss rates for these SPECInt2006 are very small even for just the L1: close to zero for most and under 1% for all of them. This low rate probably results from the computationally intensive nature of the SPECCPU programs and the two-way set associative cache that eliminates most conflict misses.

Figure 2.20 shows the data cache results, which have significant L1 and L2 miss rates. The L1 rate varies by a factor of 75, from 0.5% to 37.3%, with a median miss rate of 2.4%. The global L2 miss rate varies by a factor of 180, from 0.05% to 9.0%, with a median of 0.3%. MCF, which is known as a cache buster, sets the upper bound and significantly affects the mean. Remember that the L2 global miss rate is significantly lower than the L2 local miss rate; for example, the median L2 stand-alone miss rate is 15.1% versus the global miss rate of 0.3%.

Using these miss penalties

Using these miss penalties in Figures 2.18 and 2.21 shows the average penalty per data access. Although the L1 miss rates are about seven times higher than the L2 miss rate, the L2 penalty is 9.5 times as high, leading to L2 misses slightly dominating the benchmarks that stress the memory system. In the next chapter we will examine the impact of the cache misses on overall CPI.

The Intel Core i9

The i9 supports the x86-64 ISA, a 64-bit extension of the 80x86 architecture. The i9 12900 (which uses the Alder Lake microarchitecture) is a *big.little* design meaning that it combines two types of cores: 8 performance cores (P-cores) and 8 efficiency cores (E-cores). P-cores have a higher clock rate and support multithreading (see Chapter 3). In this chapter we focus on the memory system design and performance from the viewpoint of a single P-core. The pipeline and

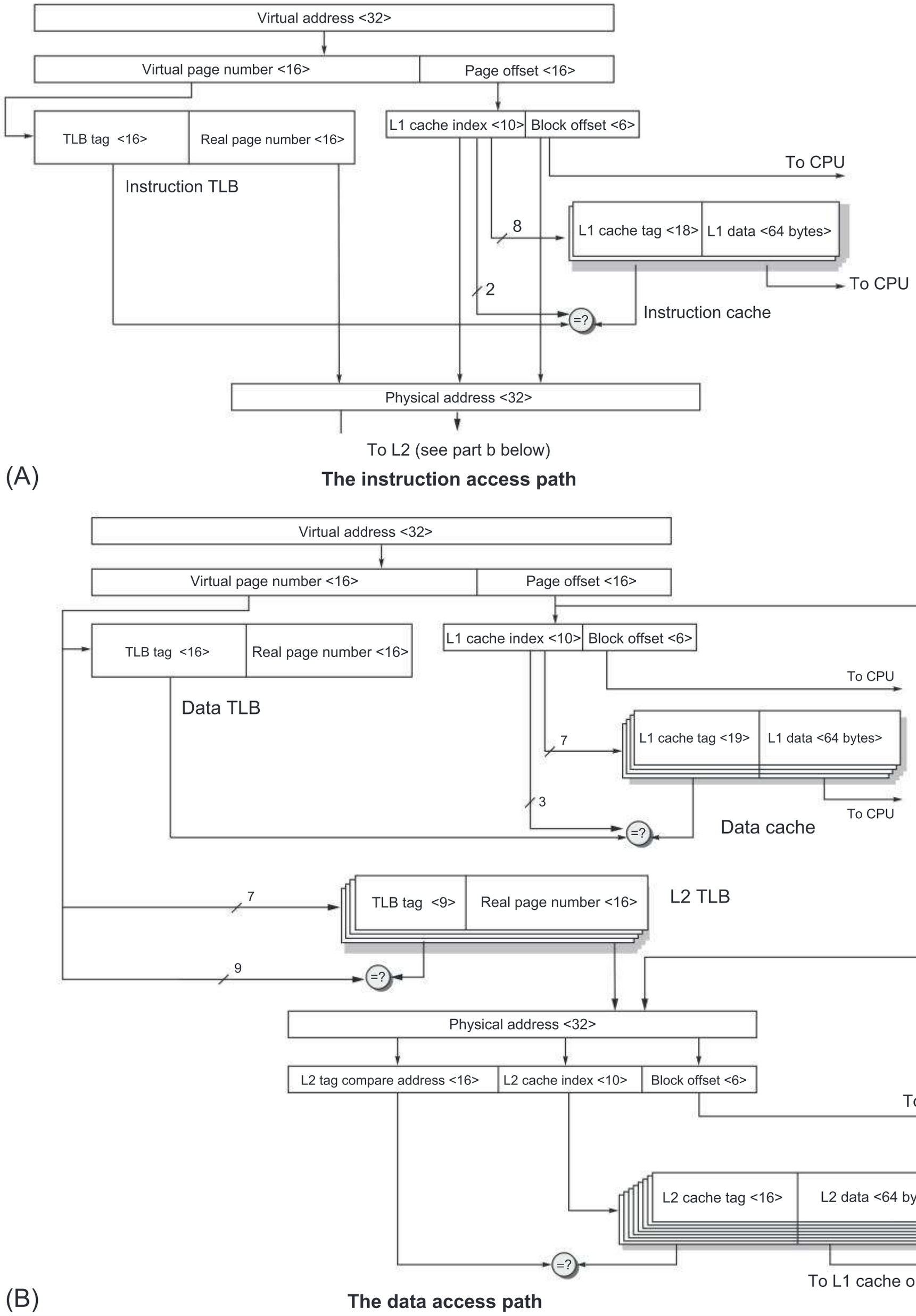


Figure 2.19 The virtual address, physical, and data blocks for the ARM Cortex-A53 caches and TLBs, assuming 32-bit addresses. The top half (A) shows the instruction access; the bottom half (B) shows the data access, including L2. The TLB (instruction or data) is fully associative, each with 10 entries, using a 64 KiB page in this example. The L1 I-cache is two-way set associative, with 64-byte blocks and 32 KiB capacity; the L1 D-cache is 32 KiB, four-way set associative, and with 64-byte blocks. The L2 TLB is 512 entries and four-way set associative. The L2 cache is 16-way set associative with 64-byte blocks and 128 cKiB to 2 MiB capacity; a 1 MiB L2 is shown. This figure doesn't show the valid bits and protection bits for the caches and TLB.

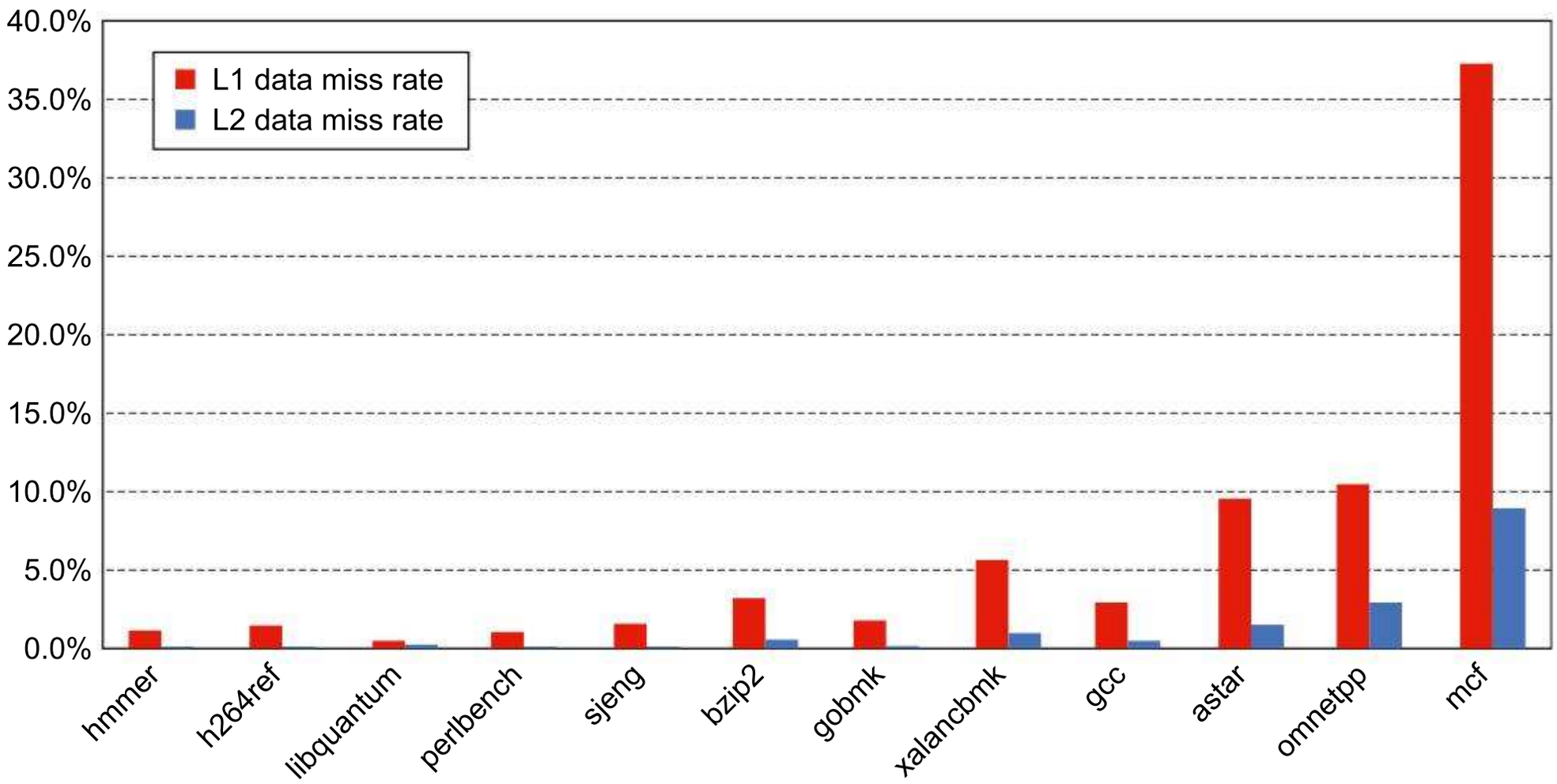


Figure 2.20 The data miss rate for ARM with a 32 KiB L1 and the global data miss rate for a 1 MiB L2 using the SPECInt2006 benchmarks are significantly affected by the applications. Applications with larger memory footprints tend to have higher miss rates in both L1 and L2. Note that the L2 rate is the global miss rate that is counting all references, including those that hit in L1. MCF is known as a cache buster.

multithread performance of the i9 is explored in Chapter 3, while the multiprocessor performance is examined in Chapter 5.

Each core in an i9 can execute up to four 80x86 instructions per clock cycle, using a multiple-issue, dynamically scheduled, 16-stage pipeline, which we describe in detail in Chapter 3. In 2024 the fastest i9 had a clock rate for a single P-core of 5.1 GHz (in Turbo Boost mode). Of course, the i9 cannot operate with multiple P-cores in turbo boost mode without overheating.

The i9 can support two memory channels, each consisting of a separate set of DIMMs, and each of which can transfer in parallel using either DDR4 or DDR5. The maximum memory bandwidth with DDR5-4800 is 77 GB/s.

The i9 uses 48-bit virtual addresses and 36-bit physical addresses, yielding a maximum physical memory of 36 GiB. Memory management is handled with a two-level TLB (see Appendix B, Section B.4), summarized in Figure 2.22.

Figure 2.23 summarizes the i9's three-level cache hierarchy. The first-level caches are virtually indexed and physically tagged (see Appendix B, Section B.3), while the L2 and L3 caches are physically indexed. Compared to earlier generations of the Skylake microarchitecture, the most recent high-end processors opted for a significant increase in the size of L2 (2—4 times) and a smaller increase in the size of L3. This decision means that the total size of L2 and the total of L3 are closer in size, leading to the decision to make L3 noninclusive. Some versions of the i9 will support a fourth-level cache using HBM packaging.

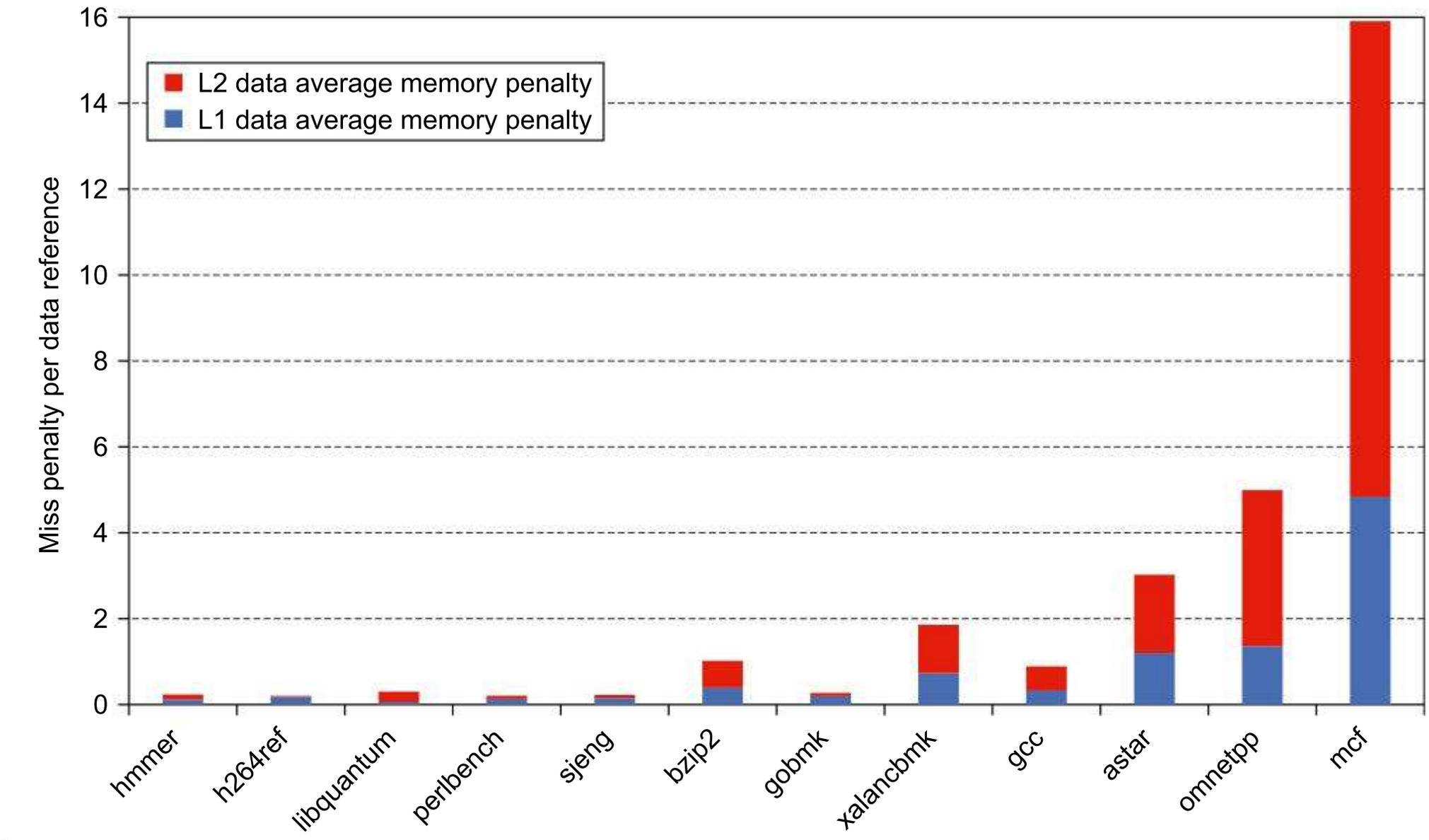


Figure 2.21 The average memory access penalty per data memory reference coming from L1 and L2 is shown for the A53 processor when running SPECInt2006. Although the miss rates for L1 are significantly higher, the L2 miss penalty, which is more than five times higher, means that the L2 misses can contribute significantly.

Characteristic	Instruction TLB	Data DLB	Second-level TLB
Entries	256 4KiB pages 32 2/4MiB pages	96 4KiB pages 32 2/4MiB pages 8 1GiB pages	2048 entries for 4KiB, 2MiB, and 1 GiB pages
Associativity	8-way	6-way, 4-way, 8-way	16-way
Replacement	Pseudo-LRU	Pseudo-LRU	Pseudo-LRU
Access latency	**1** cycle	**1** cycle	8 cycle
Miss	8 cycles	8 cycles	Tens to hundreds of cycles to access page table

Figure 2.22 Characteristics of the i9's TLB structure, which has separate multiple first-level instruction and data TLBs, both backed by a joint second-level TLB. The first-level TLBs support the standard 4 KiB page size, as well as having separate TLBs for other page sizes. The data TLB also includes separate small TLBs for loads and stores. The i9 has the ability to handle four L2 TLB misses in parallel using a hardware-based page table walker. See the online Appendix L for more discussion of multilevel TLBs and support for multiple page sizes, as well as hardware TLB miss handlers.

Characteristic	L1 I-cache	L1 D-cache	L2	L2
Size	32 KiB	48 KiB; dual ported; 8 banks	1.25 MiB per core	30 MiB shared by all cores; not inclusive
Associativity	8-way	6-way	10-way	15-way
Access latency	4 cycles	5 cycles	15 cycles	50 cycles
Replacement scheme	Pseudo-LRU	Pseudo-LRU	Weighted n-bit LRU	Weighted n-bit LRU

Figure 2.23 Characteristics of the three-level cache hierarchy in the i9 P-core. All three caches use write back and a block size of 64 bytes. The L1 and L2 caches are separate for each core, whereas the L3 cache is shared among the cores on a chip. L3 is physically distributed into multiple banks, adjacent to the cores, and a simple hashing function is used to assign addresses to L3 banks. All three caches are nonblocking and allow multiple outstanding writes. A merging write buffer is used for the L1 cache, which holds data in the event that the line is not present in L1 when it is written. (i.e., an L1 write miss does not cause the line to be allocated.) L3 is not inclusive. This choice allows the amount of data in the combined L2s and L3 to be greater than the size of L3, which is not true if L3 is inclusive. (Noninclusivity has implications for cache coherence, which we explore in Chapter 5.) Replacement is by variants on Pseudo-LRU and approximate LRU, using one-bit (a form of NRU) in L1 and a multibit scheme in L2 and L3.

Figure 2.24 is labeled with the steps of an access to the memory hierarchy, assuming 4KiB page size. First, the PC is sent to the instruction cache. The instruction cache index is

$$2^{\text{Index}} = \frac{\text{Cache size}}{\text{Block size} \times \text{Set associativity}} = \frac{32\text{ K}}{64 \times 8} = 64 = 2^6$$

or 6 bits. The page frame of the instruction's address ($36 = 48 - 12$ bits) is sent to the instruction TLB (step 1). At the same time, the 12-bit page offset from the virtual address is sent to the instruction cache (step 2). Notice that for the eight-way associative instruction cache, 12 bits are needed for the cache address: 6 bits to index the cache plus 6 bits of block offset for the 64-byte block, so no aliases are possible. Earlier versions of the i7 used a four-way set associative I-cache, meaning that a block corresponding to a virtual address could be in two different places in the cache, because the corresponding physical address could have either a 0 or 1 in this location. For instructions this did not pose a problem because even if an instruction appeared in the cache in two different locations, the two versions must be the same. If such duplication, or aliasing, of data is allowed, the cache must be checked when the page map is changed, which is an infrequent event. Note that a very simple use of page coloring (see Appendix B, Section B.3) can eliminate the possibility of these aliases. If even-address virtual pages are mapped to even-address physical pages (and the same for odd pages), then these aliases can never occur because the low-order bit in the virtual and physical page numbers will be identical.

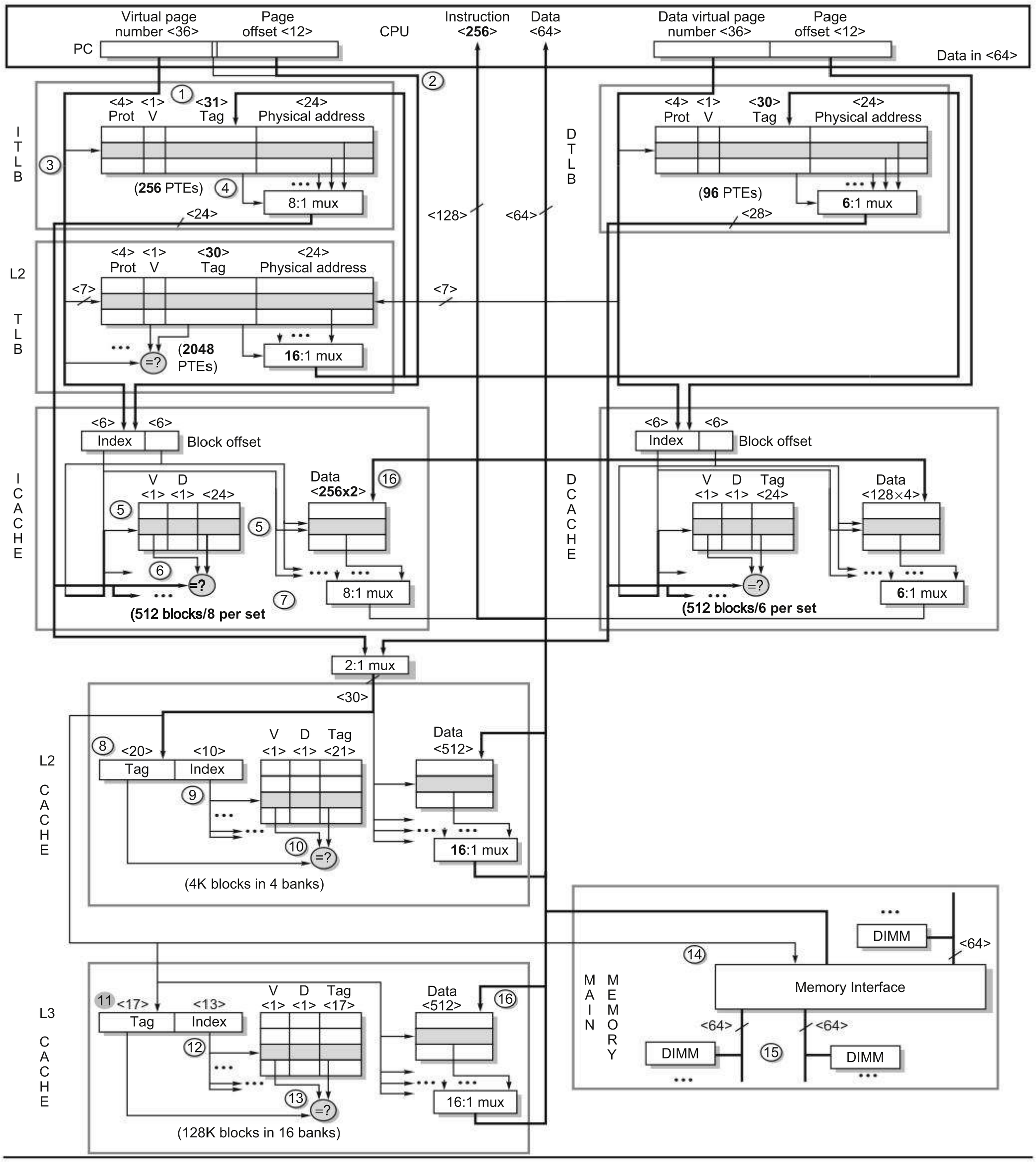


Figure 2.24 The Intel i9 memory hierarchy and the steps in both instruction and data access. We show only reads and only the primary TLBs for 4KiB pages. Writes are similar, except that misses are handled by simply placing the data in a write buffer, because the L1 cache is not write-allocated. There are several DTLBs as well as support for large page sizes both in the data and instruction TLBs.

The instruction TLB is accessed 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 because of an access violation.

An instruction TLB miss first goes to the L2 TLB (or STLB), which contains 2048 PTEs of 4 KiB page sizes and is 16-way set associative. It takes 8 clock cycles to load the L1 TLB from the L2 TLB, which leads to the 9-cycle miss penalty including the initial clock cycle to access the L1 TLB. If the L2 TLB misses, a hardware algorithm is used to walk the page table and update the TLB entry. Sections L.5 and L.6 of online Appendix L describe page table walkers and page structure caches. In the worst case the page is not in memory, and the operating system gets the page from secondary storage. Because 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 all eight banks of the instruction cache (step 5). The instruction cache tag is 36 bits − 6 bits (index) − 6 bits (block offset), or 24 bits. The four tags and valid bits are compared to the physical page frame from the instruction TLB (step 6). Because the i9 expects 32 bytes each instruction fetch, an additional bit is used from the 6-bit block offset to select the appropriate 32 bytes. Therefore 6 + 1 or 87 bits are used to send 32 bytes of instructions to the processor. The L1 cache is pipelined, and the latency of a hit is 4 clock cycles (step 7). A miss goes to the second-level cache.

As mentioned earlier, the instruction cache is virtually addressed and physically tagged. Because the second-level caches are physically addressed, the physical page address from the TLB is composed with the page offset to make an address to access the L2 cache. The L2 index is

$$2^{\text{Index}} = \frac{\text{Cache size}}{\text{Block size} \times \text{Set associativity}} = \frac{1.25\text{ M}}{64 \times 10} = 2^{11}$$

so the 30-bit block address (36-bit physical address − 6-bit block offset) is divided into a 19-bit tag and an 11-bit index (step 8). Once again, the index and tag are sent to the core's L2 cache (step 9), which are compared in parallel. If one matches and is valid (step 10), it returns the block in sequential order after the initial 15-cycle latency at a rate of 8 bytes per clock cycle.

If the L2 cache misses, the L3 cache is accessed. For an i9-12900 with 8 P-cores and 8 E-cores, L3 is 30MiB in 8 banks. The bank is hashed to reduce clashes and 3 bits from the hash are used to select the bank, and only that bank will be accessed. The index size within each bank is

$$2^{\text{Index}} = \frac{\frac{\text{Cache size}}{8}}{\text{Block size} \times \text{Set associativity}} = \frac{3.75\text{ M}}{64 \times 15} = 4096 = 2^{12}$$

The 12-bit index (step 11) is sent to all 15 ways of the selected bank of L3 (step 12). The L3 tag, which is $36 - (12 + 6) = 18$ bits, is compared against the physical address from the TLB (step 13). If a hit occurs, the block is returned after an initial latency of 50 clock cycles, at a rate of 16 bytes per clock, and placed into both L1 and L2. If L3 misses, a memory access is initiated.

The i9 has two (or more in Xeon server processors) independent memory channels, which can be connected to HBM memory or standard DIMMs. The total latency of the miss that is serviced by main memory is approximately 50 processor cycles to determine that an L3 miss has occurred, plus the DRAM latency for the critical instructions. For a single-bank DDR5-4800 SDRAM and 4.0 GHz CPU, the DRAM latency is about 40 ns or 160 clock cycles to the first 16 bytes, leading to a total miss penalty of about 200 clock cycles. The memory controller fills the remainder of the 64-byte cache block at a rate of 32 bytes per I/O bus clock cycle, which takes another 2.5 ns or 10 clock cycles. On a miss, the missing block is written into L2 and L1; it is not inserted into L3 (recall that the LLC is noninclusive).

The LLC's function is primarily to hold blocks that are ejected from L2. Ignoring sharing by multiple cores, L2 and L3 will hold different blocks, meaning that the total cache data is equal to the sum of the size of L2 and L3, rather than just the size of L3. Because the L2 and L3 are both write-back caches, any miss can lead to an old block being written back to L3 or memory, respectively. The i9 has a merging write buffer that writes back dirty cache lines when the next level in the cache is unused for a read. The write buffer is checked on a miss to see if the cache line exists in the buffer; if so, the miss is filled from the 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 miss causes the block to be placed in a write buffer because the L1 cache does not allocate the block on a write miss. On a hit, the store does not update the L1 (or L2) 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.

The i9 also supports prefetching for L1 and L2 from the next level in the hierarchy. The i9 prefetcher is similar to the i7 prefetcher, whose performance we looked at earlier.

Performance of the i9 memory system

We evaluate the performance of the i9 cache structure using the SPECINT2017 benchmarks (SPECINTRate). The complexity of the i9 pipeline, with its use of an autonomous instruction fetch unit, speculation, and both instruction and data prefetch, makes it hard to compare cache performance against simpler processors. As mentioned earlier, processors that use prefetch can generate cache accesses independent of the memory accesses performed by the program. A cache access that is generated because of an actual instruction access or data access is sometimes called a *demand access* to distinguish it from a *prefetch access*. Demand accesses can come from both speculative instruction fetches and speculative

data accesses, some of which are subsequently canceled (see Chapter 3 for a detailed description of speculation and instruction graduation). A speculative processor generates at least as many misses as an in-order nonspeculative processor, and typically more. Because of these complexities, it makes less sense to talk about the miss rate; instead, we use the now common measurement of misses per kilo instructions (MPKI), where the numerator is misses and the denominator is instructions that commit. MPKI can be given for different cache levels: hence MPKID1 (first-level data cache) and MKPI2 and MKPI3, the misses in L2 and L3, respectively.

In addition to demand misses, there are effective misses generated by instruction and data prefetches. If the prefetcher is highly effective, then eliminating the prefetches would generate other misses. Issuing the prefetch and getting a "prefetch miss" can allow the processor to run faster, if it avoids the demand miss later. The prefetcher in an i9 P-core is an extension of the Skylake prefetcher, whose performance we considered in the section on hardware prefetching starting on page 23.

The i9's instruction fetch unit attempts to fetch 32 bytes every cycle, which complicates comparing instruction cache miss rates because multiple instructions are fetched every cycle. In fact, the entire 64-byte cache line is read, and subsequent 32-byte fetches do not require additional accesses. Thus misses are tracked only based on 64-byte blocks. The 32 KiB, eight-way set associative instruction cache leads to a very low instruction miss rate for the SPECintRate programs. If, for simplicity, we measure the miss rate of SPECintRate as the number of misses for a 64-byte block divided by the number of instructions that complete, the miss rates are all under 1% except for one benchmark (XALANCBMK). Because a 64-byte block typically contains 16–20 instructions, the effective miss rate per instruction is much lower, depending on the degree of spatial locality in the instruction stream.

Figure 2.25 shows the L1 data cache, L2, and L3 MPKI rates for the SPECCPUINT2017 benchmarks. Notice that four of the benchmarks have significant L1D misses exceeding 20 instructions per thousand (recall that the L1 prefetchers in Skylake were not very effective); those same four benchmarks also have significant L2 misses of 10 or more per thousand instructions, despite the large L2 cache of 1.25 MiB.

What Figure 2.25 does not indicate is the potential impact of misses. Of course, this is virtually impossible to measure accurately in a processor with nonblocking caches and out-of-order execution since overlapping misses have lower cost and the processor may be able to stay busy especially during short L1 misses. To give a rough estimate of the potential impact of L1, L2, and L3 misses, Figure 2.26 shows the MPKI for L1D, L2, and L3 multiplied by the average latency of a miss to the next level in the hierarchy. This is not a measurement of the cost but instead gives an indication of the potential importance of misses at various levels. The benchmark mcf is widely known as a cache-buster and has significant L3 misses. The other memory-intensive benchmarks show a balance of impact between L1, L2, and L3 misses. Recall that a number of the floating-point SPEC benchmarks

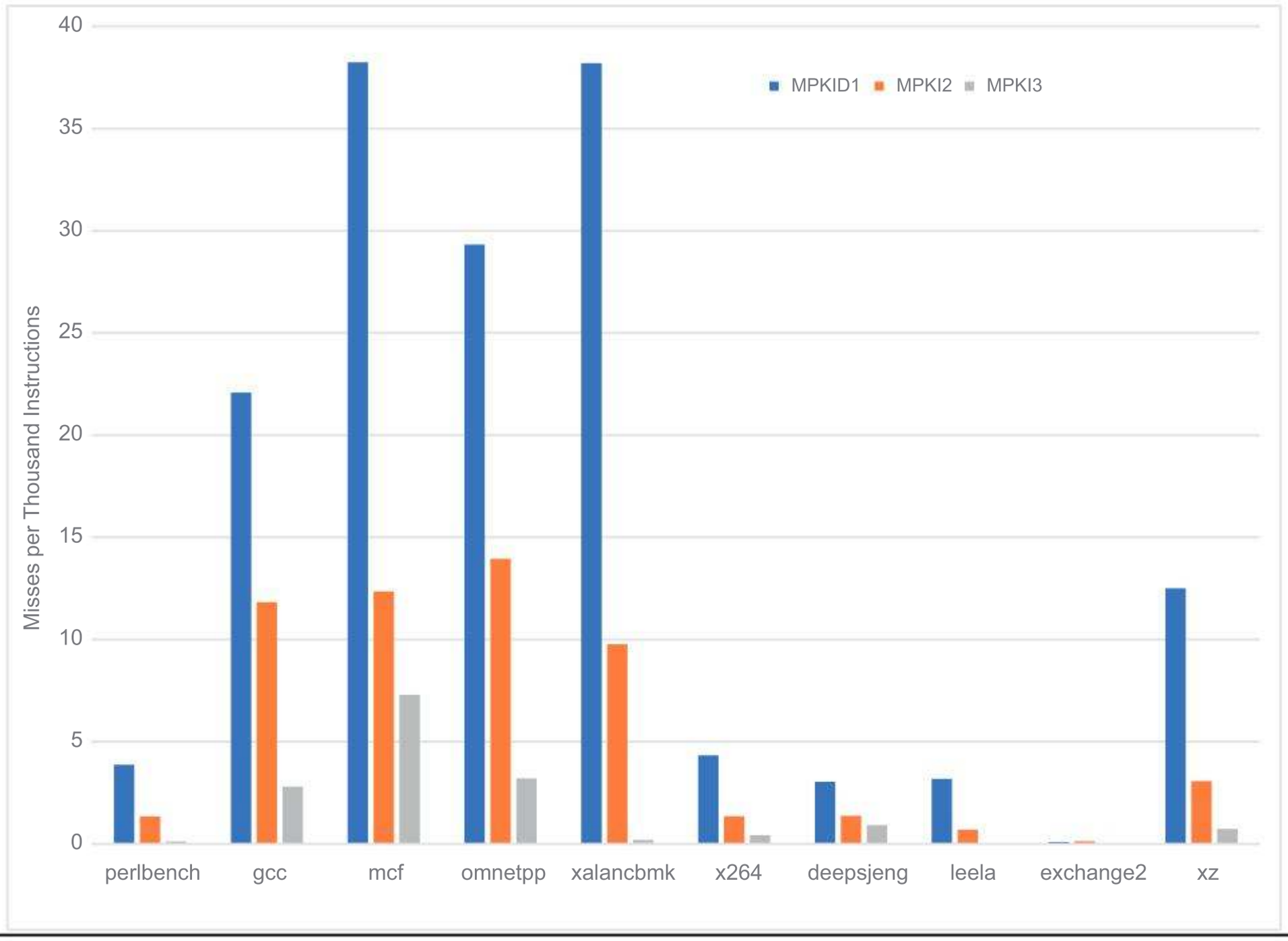


Figure 2.25 The MPKI for the L1 D-cache, L2, and L3 are shown for the SPECCPU2017 integer benchmarks. This is for the i9-12900 memory system. L2 and L3 handle L1I misses, but few of these get past L2. Prefetching is enabled, and—as shown earlier—significantly reduces the L3 demand misses that would otherwise come from L2.

also have high cache miss rates, as we discussed earlier when examining prefetching on the i7. In the next chapter we will examine the relationship between the i9 CPI and cache misses, as well as other pipeline effects.

2.7 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 2.27 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 KiB cache are 9, 2, or

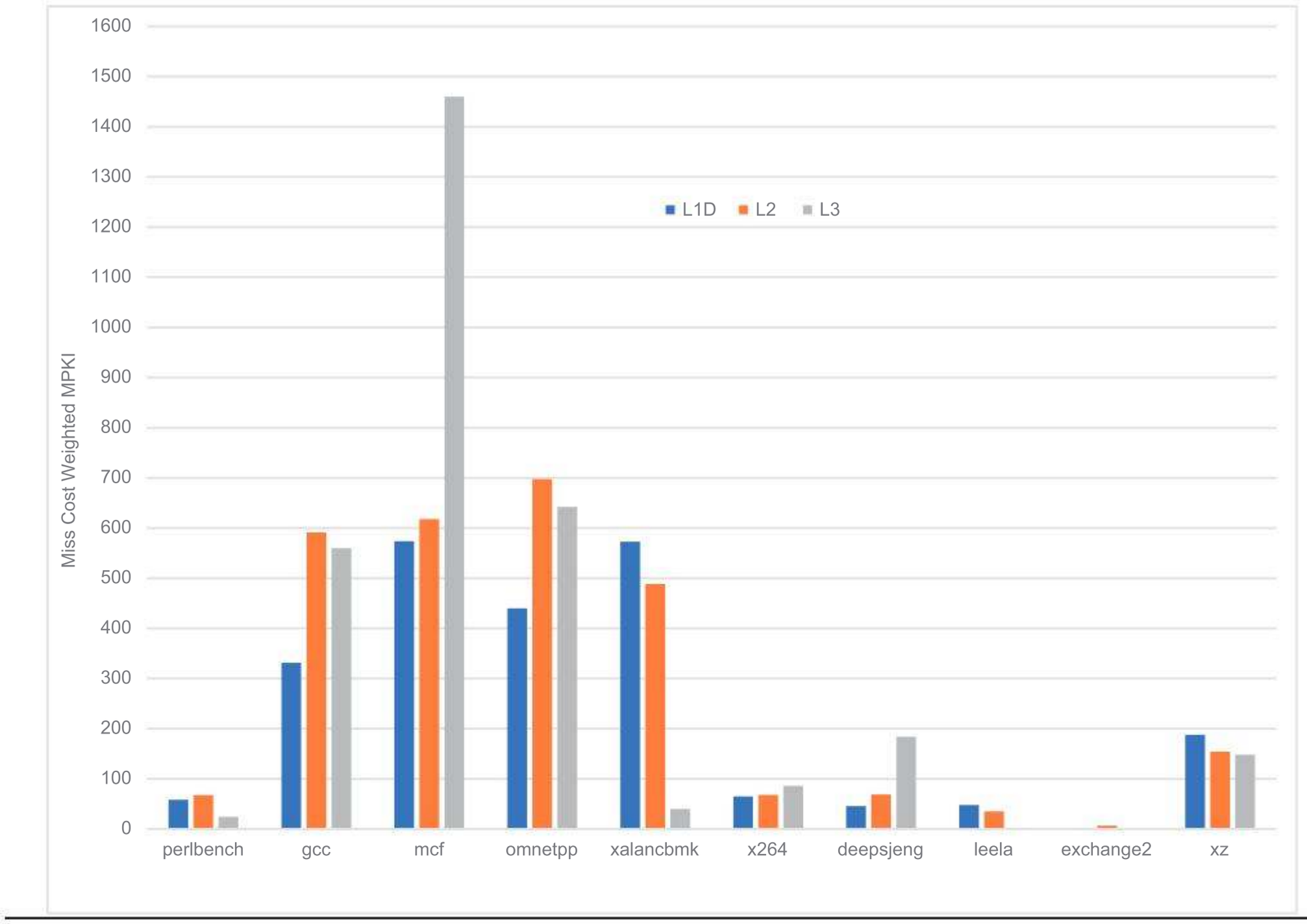


Figure 2.26 MPKI for L1D, L2, and L3 weighted by the average miss penalty. The miss penalty weighting is average and ignores contention but also ignores potential overlap with instructions.

90, and the instruction misses per thousand instructions for a 4 KiB cache are 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 SPECCPU programs. Clearly, generalizing cache performance from one program to another is unwise. As Figure 2.23 reminds us, there is a great deal of variation, and even predictions about the relative miss rates of integer and floating-point—intensive programs can be wrong, as mcf and sphnix3 remind us!

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

Pitfall 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.

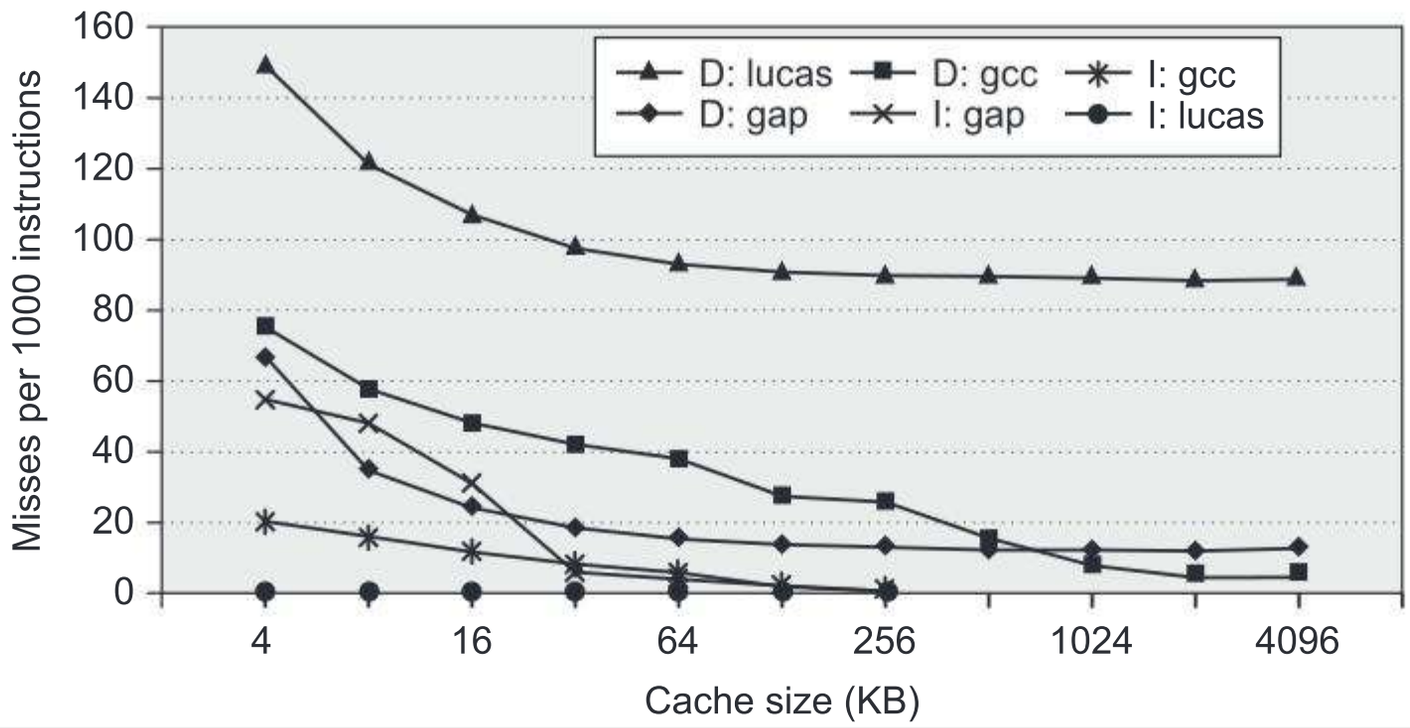


Figure 2.27 Instruction and data misses per 1000 instructions as cache size varies from 4 KiB to 4096 KiB. Instruction misses for gcc are 30,000—40,000 times larger than for lucas, and conversely, data misses for lucas are 2—60 times larger than for gcc. The programs gap, gcc, and lucas are from the SPEC2000 benchmark suite.

Figure 2.28 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.

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 design an HBM behind the cache for such applications. We will revisit this pitfall in Chapters 4 and 5.

Pitfall *Introducing a memory technology that "fits" between two existing technologies but does not offer clear advantages in either speed or price.*

There have been attempts to introduce new memory technologies either to replace an existing technology or to "fit" between two existing technologies (DRAM and disk, e.g.,). Most of these attempts have failed because the new technology did not offer a significant and sustainable advantage. Sometimes, the new technology was cheaper per bit but much slower, or sometimes, they were faster but more costly. The phase change memory (called Xpoint by Micron and Intel) seems to be the latest technology to encounter this pitfall, but there are many earlier ones, including charge-coupled devices, bubble memory, and optical disks. Flash, in contrast, is the single big success. Why did Flash win: it was nonvolatile and considerably faster than disks (especially for reads) and over time was able to close much of the difference in cost/bit. By 2022 Flash had largely replaced hard disks, except for large archival storage.

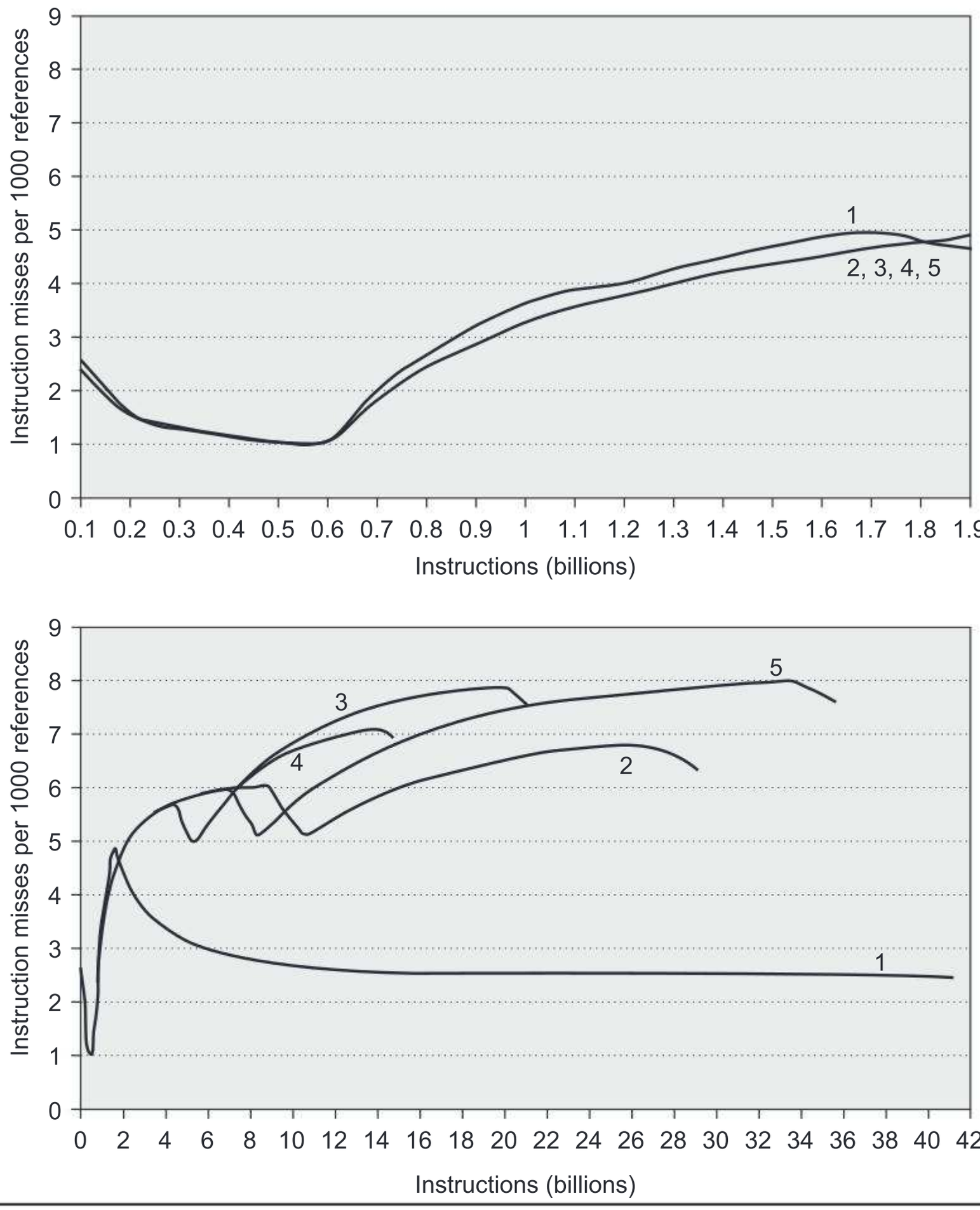


Figure 2.28 Instruction misses per 1000 references for five inputs to the perl benchmark in SPEC2000. There is little variation in misses and little difference between the five inputs for the first 1.9 billion instructions. Running to completion shows how misses vary over the life of the program and 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 being two-way 64 KiB with LRU, and a unified 1 MiB direct-mapped L2 cache.

2.8 Concluding Remarks: Looking Ahead

Over the past thirty years there have been several predictions of the eminent [sic] cessation of the rate of improvement in computer performance. Every such prediction was wrong. They were wrong because they hinged on unstated assumptions that were overturned by subsequent events. So, for example, the failure to foresee the move from discrete components to integrated circuits led to a prediction that the speed of light would limit computer speeds to several orders of magnitude slower than they are now. Our prediction of the memory wall is probably wrong too but it suggests that we have to start thinking "out of the box."

Wm. A. Wulf and Sally A. McKee,
Hitting the Memory Wall: Implications of the Obvious,
Department of Computer Science, University of Virginia (December 1994).
This paper introduced the term memory wall.

The possibility of using a memory hierarchy dates to the earliest days of general-purpose digital computers in the late 1940s and early 1950s. Virtual memory was introduced in research computers in the early 1960s and into IBM mainframes in the 1970s. Caches appeared around the same time. The basic concepts have been expanded and enhanced over time to help close the access time gap between main memory and processors, but the basic concepts remain. More recently, a variety of factors are converging to make the memory wall, which was overcome for 25 years by advanced memory hierarchies, a major limit in increasing performance.

One trend that is causing a significant change in the design of memory hierarchies is a continued slowdown in both density and access time of DRAMs. In the past 15 years both these trends have been observed and have been even more obvious over the past 5 years. While some increases in DRAM bandwidth have been achieved, decreases in access time have come much more slowly and vanished between DDR5 and DDR4. The end of Dennard scaling as well as a slowdown in Moore's law both contributed to this situation. The trenched capacitor design used in DRAMs is also limiting its ability to scale. It may well be the case that packaging technologies such as stacked memory will be the dominant source of improvements in DRAM access bandwidth and latency. More recently, SRAM has also shown less performance improvement with new semiconductor technologies, compared to logic.

Thus we see a variety of factors that will hinder the performance of memory hierarchies, including:

- No latency improvement and smaller density improvements in DRAM.
- Smaller improvements in SRAM access time and little improvement in power consumption per bit.

- Diminishing returns from either larger LLCs or more levels of cache.
- A growth in applications that stream very large data sets and have lower temporal and spatial locality (including both graphics and deep neural networks; see Chapters 4 and 7).

Together, these factors increase the effect of a memory wall, which was held at bay for 25 years. As we will see in Chapters 4 and 7, the best solution for some applications may be a different memory organization, an observation that Cray implemented in an early supercomputer designed 50 years ago!

Independently of improvements in DRAM and SRAM, Flash memory has been playing a much larger role. In PMDs Flash has dominated for 15 years and became the standard for laptops almost 10 years ago. In the past few years most desktops have shipped with Flash as the primary secondary storage. Flash's potential advantage over DRAMs, specifically the absence of a per-bit transistor to control writing, is also its Achilles heel. Flash must use bulk erase-rewrite cycles that are considerably slower. As a result, although Flash has become the fastest-growing form of secondary storage, DDR DRAMs still dominate main memory.

Although phase-change materials as a basis for memory have been around for a while, they have never been serious competitors, either for magnetic disks or for Flash. The technology appears to have several advantages over Flash, including the elimination of the slow erase-to-write cycle and greater longevity in terms, but so far it has not been a significant commercial success.

For some years, a variety of predictions have been made about the coming memory wall (see previously cited quote and paper), which would lead to serious limits on processor performance. Fortunately, the extension of caches to multiple levels (from 2 to 4), more sophisticated refill and prefetch schemes, greater compiler and programmer awareness of the importance of locality, and tremendous improvements in DRAM bandwidth (a factor of over 200 times since the mid-1990s) have helped keep the memory wall at bay. In recent years the combination of access time constraints on the size of L1 (which is limited by the clock cycle) and energy-related limitations on the size of L2 and L3 have raised new challenges. The evolution of the i7 and i9 processor cores over about a dozen years illustrates this: the L1 I-cache is the same size and D-cache has grown by only 50%! The more aggressive use of prefetching is an attempt to overcome the inability to increase L2 and L3.

In addition to schemes relying on multilevel caches, the introduction of out-of-order pipelines with multiple outstanding misses has allowed available instruction-level parallelism to hide the memory latency remaining in a cache-based system. The introduction of multithreading and more thread-level parallelism takes this a step further by providing more parallelism and thus more latency-hiding opportunities. It is likely that the use of instruction- and thread-level parallelism will be a more important tool in hiding whatever memory delays are encountered in modern multilevel cache systems.

One idea that periodically arises is the use of programmer-controlled scratchpad or other high-speed visible memories, which we will see are used in GPUs and domain-specific architectures (see Chapter 7). Such ideas have never made the mainstream in general-purpose processors for several reasons. First, they break the memory model by introducing address spaces with different behavior. Second, unlike compiler-based or programmer-based cache optimizations (such as prefetching), memory transformations with scratchpads must completely handle the remapping from main memory address space to the scratchpad address space. This makes such transformations more difficult and limited in applicability. In GPUs (see Chapter 4) and domain-specific architectures (see Chapter 7), where local scratchpad memories are heavily used, the burden of managing them currently falls on the programmer and the compiler. For domain-specific software systems that can use such memories, the performance gains are very significant. HBM technologies are the main working memories in graphics and similar systems. Domain-specific architectures have become more important in overcoming the limitations arising from the end of Dennard's law and the slowdown in Moore's law (see Chapter 7), leading to scratchpad memories and vector-like register sets becoming more popular.

The implications of the end of Dennard's law affect both DRAM and processor technology. There is a slightly greater impact on DRAMs, due to their unique structure, but both logic and memory are affected by power limits and the slowdown in Moore's law. New innovations in computer architecture and in related software that together increase performance and efficiency will be key to continuing the performance improvements seen over the past 50 years.

2.9 Historical Perspectives and References

In Section M.3 (available online) we examine the history of caches, virtual memory, and VMs. IBM plays a prominent role in the history of all three. References for further reading are included.

Case Studies and Exercises by Rajeev Balasubramonian, Norman P. Jouppi, Naveen Muralimanohar, and Sheng Li

Case Study 1: 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; this concept is illustrated here:

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];
 }
}
```

2.1. [10/15/15/12/20] <2.3> Assume that both the input and output matrices are stored in the row major order (*row major order* means that consecutive elements of a row occupy consecutive addresses in memory). Assume that you are executing a 256·256 double-precision transpose on a processor with a 16 KB fully associative (don't worry about cache conflicts) least recently used (LRU) replacement L1 data cache with 64-byte blocks. Recall that a double-precision value occupies 8 bytes. Assume that the L1 cache misses or prefetches require 16 cycles and always hit in the L2 cache, and that the L2 cache can process a request every 2 processor cycles. Assume that each iteration of the preceding inner loop requires 4 cycles if the data are present in the L1 cache. Assume that the cache has a write-allocate fetch-on-write policy for write misses. Unrealistically, assume that writing back dirty cache blocks requires 0 cycles. For the preceding simple implementation, this execution order would be nonideal for the output matrix; however, applying a loop interchange optimization would create a nonideal order for the input matrix. Because loop interchange is not sufficient to improve its performance, it must be blocked instead.

a. [10] <2.3> What should be the minimum size of the cache to take advantage of blocked execution?
b. [15] <2.3> How do the relative number of misses in the blocked and unblocked versions compare in the preceding minimum-sized cache?
c. [15] <2.3> Write code to perform a transpose with a block size parameter B that uses $B \cdot B$ blocks.
d. [12] <2.3> What is the minimum associativity required of the L1 cache for consistent performance independent of both arrays' position in memory?
e. [20] <2.3> Try out blocked and nonblocked 256·256 matrix transpositions on a computer. How closely do the results match your expectations based on what you know about the computer's memory system? Explain any discrepancies if possible.

2.2. [10] <2.3> Assume you are designing a hardware prefetcher for the preceding *unblocked* matrix transposition code. 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. For example, you can insert software prefetches in C code using the __builtin_prefetch( ) function. Assume prefetches write directly into the cache and that there is no "pollution" (overwriting data that must be used before the data that are prefetched). For best performance given a nonunit stride prefetcher, in the steady state of the inner loop, how many prefetches must be outstanding at a given time?

2.3. [15/20] <2.3> With software prefetching, it is important to be careful to have the prefetches occur in time for use but also to minimize the number of outstanding prefetches 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. [15] <2.3> Create a blocked version of the matrix transpose with software prefetching.
b. [20] <2.3> Estimate and compare the performance of the blocked and unblocked transpose codes both with and without software prefetching.

Case Study 2: Putting It All Together: Highly Parallel Memory Systems

Concepts illustrated by this case study

- Cross-Cutting Issues: The Design of Memory Hierarchies
- Memory Hierarchy Parameters in Modern Processors

The program in Figure 2.29 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 2.29 shows 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 have to be changed 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 `.csv` 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. Note that the code can take several minutes to complete on a modern machine. The code in Figure 2.29 was derived from a program written by Andrea Dusseau at the University of California-Berkeley

```
#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 */
  time_t ltime;
  time(&ltime);
  return (double) ltime;
}

int label(int i) { /* generate text labels */
  if (i<1e3) printf("%ldB,",i);
  else if (i<1e6) printf("%ldK,",i/1024);
  else if (i<1e9) printf("%ldM,",i/1048576);
  else printf("%ldG,",i/1073741824);
  return 0;
}

int main(int argc, char* 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 for 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();
sec0 = get_seconds();
while (sec0 == lastsec) {
  sec0 = get_seconds();
}

/* 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 20 seconds of execution */
  for (i=stride;i!=0;i=i-1) { /* keep samples same */
       nextstep = 0;
```

Figure 2.29 C program for evaluating the memory system.

```
        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 number of while iterations */
sec0 = get_seconds(); /* start timer */
do { /* repeat until same number of 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 = number of iterations above */
  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 2.29 continued

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 there are versions available that are compatible with Linux and Microsoft Visual C++ platforms at this link: https://users.cs.utah.edu/~rajeev/HP-code/. The preceding program assumes that the program addresses track physical addresses, which is true on the few machines that use virtually addressed caches. In general, virtual addresses tend to follow physical addresses shortly after rebooting, so you may need to reboot the machine to get smooth lines in your results. To answer the following questions, 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 that the size of a second-level cache block is greater than or equal to the size of a block in a first-level cache.

2.4. [12/12/12/10/12] <2.6> An example of the output of the program is plotted in Figure 2.30; the x-axis lists the size of the array that is exercised. Using the sample program results in Figure 2.30:

a. [12] <2.6> What are the overall size and block size of the second-level cache?
b. [12] <2.6> What is the miss penalty of the second-level cache?

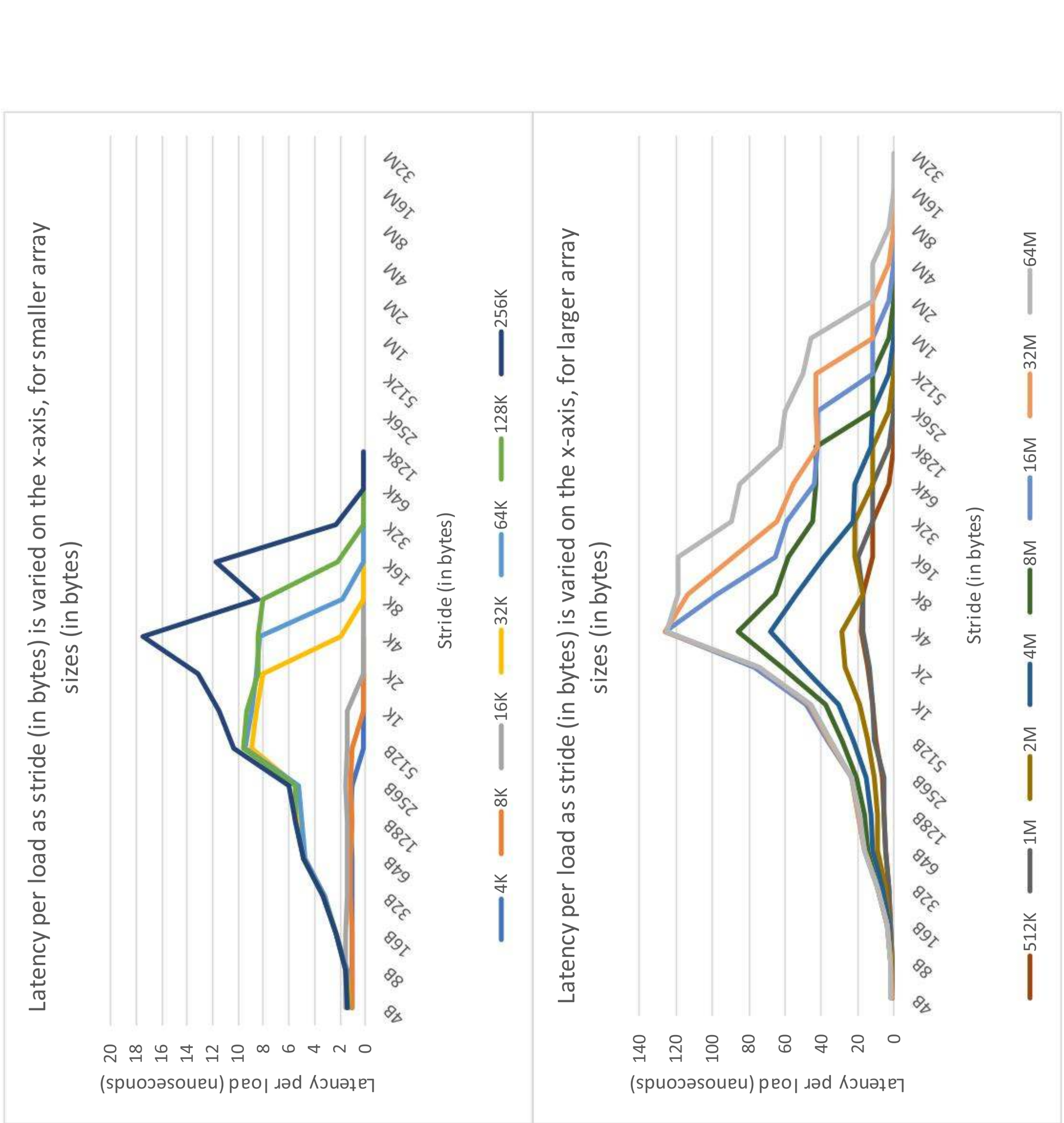


Figure 2.30 Sample output from the code in Figure 2.29.

c. [12] <2.6> What is the associativity of the second-level cache?
d. [10] <2.6> What is the size of the main memory?
e. [12] <2.6> What is the paging time if the page size is 4 KB?

2.5. [12/15/15/20] <2.6> If necessary, modify the code in Figure 2.29 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. [12] <2.6> What is the system page size?
b. [15] <2.6> How many entries are there in the TLB?
c. [15] <2.6> What is the miss penalty for the TLB?
d. [20] <2.6> What is the associativity of the TLB?

2.6. [20/20] <2.6> 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 2.29 and run multiple copies at the same time. Can you determine:

a. [20] <2.6> How many actual processors are in your computer system and how many system processors are just additional multithreaded contexts?
b. [20] <2.6> How many memory controllers does your system have?

2.7. [20] <2.6> Can you think of a way to test some of the characteristics of an instruction cache using a program? *Hint*: The compiler may generate a large number of nonobvious instructions from a piece of code. Try to use simple arithmetic instructions of known length in your instruction set architecture (ISA).

Case Study 3: Studying the Impact of Various Memory System Organizations

Concepts illustrated by this case study

- DDR3/DDR4 Memory Systems
- Impact of Ranks, Banks, Row Buffers on Performance and Power
- DRAM Timing Parameters
- DRAM Scheduling Policies

A processor chip typically supports a few DDR3 or DDR4 memory channels. We will focus on a single memory channel in this case study and explore how its performance and power are impacted by varying several parameters. Recall that the channel is populated with one or more DIMMs. Each DIMM supports one or more ranks—a rank is a collection of DRAM chips that work in unison to service a single command issued by

the memory controller. For example, a rank may be composed of 16 DRAM chips, where each chip deals with a 4-bit input or output on every channel clock edge. Each such chip is referred to as a x4 (by four) chip. In other examples a rank may be composed of 8x8 chips or 4x16 chips—note that in each case a rank can handle data that are being placed on a 64-bit memory channel. A rank is itself partitioned into 8 (DDR3) or 16 (DDR4) banks. Each bank has a row buffer that essentially remembers the last row read out of a bank. Here's an example of a typical sequence of memory commands when performing a read from a bank:

(i) The memory controller issues a Precharge command to get the bank ready to access a new row. The precharge is completed after time tRP.
(ii) The memory controller then issues an Activate command to read the appropriate row out of the bank. The activation is completed after time tRCD and the row is deemed to be part of the row buffer.
(iii) The memory controller can then issue a column-read or CAS command that places a specific subset of the row buffer on the memory channel. After time CL, the first 64 bits of the data burst are placed on the memory channel. A burst typically includes eight 64-bit transfers on the memory channel, performed on the rising and falling edges of 4 memory clock cycles (referred to as transfer time).
(iv) If the memory controller wants to then access data in a different row of the bank, referred to as a row buffer miss, it repeats steps (i) to (iii). For now, we will assume that after CL has elapsed, the Precharge in step (i) can be issued; in some cases an additional delay must be added, but we will ignore that delay here. If the memory controller wants to access another block of data in the same row, referred to as a row buffer hit, it simply issues another CAS command. Two back-to-back CAS commands have to be separated by at least 4 cycles so that the first data transfer is complete before the second data transfer can begin.

Note that a memory controller can issue commands to different banks in successive cycles so that it can perform many memory reads/writes in parallel and it is not sitting idle waiting for tRP, tRCD, and CL to elapse in a single bank. For the subsequent questions, assume that tRP = tRCD = CL = 13 ns and that the memory channel frequency is 1 GHz, that is, a transfer time of 4 ns.

2.8. [10] <2.2> What is the read latency experienced by a memory controller on a row buffer miss?

2.9. [10] <2.2> What is the latency experienced by a memory controller on a row buffer hit?

2.10. [10] <2.2> If the memory channel supports only one bank and the memory access pattern is dominated by row buffer misses, what is the utilization of the memory channel?

2.11. [15] <2.2> Assuming a 100% row buffer miss rate, what is the minimum number of banks that the memory channel should support in order to achieve 100% memory channel utilization?

2.12. [10] <2.2> Assuming a 50% row buffer miss rate, what is the minimum number of banks that the memory channel should support in order to achieve 100% memory channel utilization?

2.13. [15] <2.2> Assume that we are executing an application with four threads and the threads exhibit zero spatial locality, that is, a 100% row buffer miss rate. Every 200 ns, each of the four threads simultaneously inserts a read operation into the memory controller queue. What is the average memory latency experienced if the memory channel supports only one bank? What if the memory channel supported four banks?

2.14. [10] <2.2> From these questions, what have you learned about the benefits and downsides of growing the number of banks?

2.15. [20] <2.2> Now let's turn our attention to memory power. Download a copy of the Micron power calculator from this link: https://www.micron.com/sales-support/design-tools/dram-power-calculator

While the link has spreadsheets for various DDR technologies, we will focus on a DDR3 analysis here. The spreadsheet is preconfigured to estimate the power dissipation for several DRAM chips manufactured by Micron. In the "DDR3 Config" tab of the spreadsheet select the following configuration: an x8 DDR3 2Gb DRAM chip, with speed grade -093 and Fast precharge exit mode. Then, go to the "System Config" tab and set the workload's behavior as follows: page hit rate (row buffer hit rate) of 50%, read data bus utilization of 45%, and write data bus utilization of 25% (use the default settings for other parameters). Click on the "Summary" tab to see the power breakdown in a single DRAM chip under these chip and workload conditions. These spreadsheets are periodically updated by Micron as technology improves. The snapshot in summer 2024 shows that the DRAM chip consumes 229 mW, with the following breakdown: 45 mW in Activate operations, 145 mW in CAS operations, and 39 mW of background power. Next, click on the "System Config" tab again. Modify the read/write traffic and the row buffer hit rate and observe how that changes the power profile. For example, what is the decrease in power when channel utilization is 35% (25% reads and 10% writes), or when row buffer hit rate is increased to 80%?

2.16. [20] <2.2> In the default configuration a rank consists of eight x8 2 Gb DRAM chips. A rank can also be comprised of 16 x 4 chips or 4 x 16 chips. You can also vary the capacity of each DRAM chip—1 Gb, 2 Gb, and 4 Gb. These selections can be made in the "DDR3 Config" tab of the Micron power calculator. Tabulate the total power consumed for each rank organization. What is the most power-efficient approach to constructing a rank of a given capacity?

2.17. [20]<2.2> Consider two different memory controllers for a DDR-based memory system—as discussed earlier, one uses an open-page policy and the second uses a close-page policy. In an open-page policy, if there is no pending request to a memory bank, the row buffer is left open in hopes of future row buffer hits. In a close-page policy, if there is no pending request to a memory bank, a precharge is issued to get the bank ready for a future access to a different row (in anticipation of limited spatial locality). In both policies, if there are pending requests, the memory controller prioritizes row buffer hits; if there are no row buffer hits, it follows a first-come, first-serve policy. Below is a sequence of block reads to a bank, specified by the DRAM row that they access (X, Y, Z) and the cycle at which the request is enqueued at the memory controller. Specify the cycle at which each command is issued by the memory controller. Assume that Precharge, Activate, and Column-Read all take 10 cycles. Assume that the bank is already precharged when the first request shows up at cycle 20.

Memory Transaction	*Cycle when Enqueued*	*Command Sequence with Open-Page Policy*	*Command Sequence with Close-Page Policy*
Read X	20		
Read X	35		
Read X	90		
Read Y	95		
Read X	100		
Read X	160		

Exercises

Several of the exercises require the use of a cache simulator. The most capable, publicly available simulator is CACTI (Version 7.0), but it uses older technology (32nm). This does not fundamentally affect the exercises, although the results will not match newer SRAM and DRAM technologies. CACTI 7.0 is available on GitHub (https://github.com/topanitanw/Cacti-7.0). If you want to explore the exercises with more up-to-date technology (14nm), try using FT-CACTI, which updates the CATCI 7.0 models to a newer technology. FT-CACTI is also available on GitHub (https://github.com/marg-tools/FN-CACTI).

2.18. [12/12/15] <2.3> The following questions investigate the impact of small and simple caches using CACTI 7.0 or FT-CACTI.

a. [12] <2.3> Compare the access times of 64 KB caches with 64-byte blocks and a single bank. What are the relative access times of two-way and four-way set associative caches compared to a direct mapped organization?

b. [12] <2.3> Compare the access times of four-way set associative caches with 64-byte blocks and a single bank. What are the relative access times of 32 and 64 KB caches compared to a 16 KB cache?
c. [15] <2.3> For a 64 KB cache, find the cache associativity between 1 and 8 with the lowest average memory access time given that misses per instruction for a certain workload suite is 0.00664 for direct-mapped, 0.00366 for two-way set associative, 0.000987 for four-way set associative, and 0.000266 for eight-way set associative cache. Overall, there are 0.3 data references per instruction. Assume cache misses take 10 ns in all models. To calculate the hit time in cycles, assume the cycle time output using CACTI, which corresponds to the maximum frequency a cache can operate without any bubbles in the pipeline.

2.19. [15] <2.3> You have been asked to investigate the relative performance of a banked versus pipelined L1 data cache for a new microprocessor. Assume a 64 KB two-way set associative cache with 64-byte blocks. This pipelined cache requires four pipestages to access and perform hit detection and set selection. A banked implementation would consist of two 32 KB two-way set associative banks. Use CACTI or FT-CACTI and assume the best supported. Compare the area and total dynamic read energy per access of the pipelined design versus the banked design. State which takes up less area and which requires more power, and explain why that might be.

2.20. [12/15] <2.3> Consider the usage of critical word first and early restart on L2 cache misses. Assume a 2 MB L2 cache with 64-byte blocks and a refill path that is 16 bytes wide. Assume that 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 120 cycles, each additional 16-byte block 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 L2 cache and the requested data to the L1 cache.

a. [12] <2.3> How many cycles would it take to service an L2 cache miss with and without critical word first and early restart?
b. [15] <2.3> Do you think critical word first and early restart would be more important for L1 caches or L2 caches, and what factors would contribute to their relative importance?

2.21. [12/12] <2.3> You are designing a write buffer between a write-through L1 cache and a write-back L2 cache. The L2 cache write data bus is 16 B wide and can perform a write to an independent cache address every four processor cycles.

a. [12] <2.3> How many bytes wide should each write buffer entry be?
b. [15] <2.3> 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 64-bit stores if all other instructions could be issued in parallel with the stores and the blocks are present in the L2 cache?

c. [15] <2.3> What would the effect of possible L1 misses be on the number of required write buffer entries for systems with blocking and nonblocking caches?

2.22. [20] <2.1, 2.2, 2.3> A cache acts as a filter. For example, for every 1000 instructions of a program, an average of 20 memory accesses may exhibit low enough locality that they cannot be serviced by a 2 MB cache. The 2 MB cache is said to have an MPKI (misses per thousand instructions) of 20, and this will be largely true regardless of the smaller caches that precede the 2 MB cache. Assume the following cache/latency/MPKI values: 32 KB/1/100, 128 KB/2/80, 512 KB/4/50, 2 MB/8/40, and 8 MB/16/10. Assume that accessing the off-chip memory system requires 200 cycles on average. For the following cache configurations, calculate the average time spent accessing the cache hierarchy. What do you observe about the downsides of a cache hierarchy that is too shallow or too deep?

a. 32 KB L1; 8 MB L2; off-chip memory
b. 32 KB L1; 512 KB L2; 8 MB L3; off-chip memory
c. 32 KB L1; 128 KB L2; 2 MB L3; 8 MB L4; off-chip memory

2.23. [15] <2.1, 2.2, 2.3> Consider a 16 MB 16-way L3 cache that is shared by two programs, A and B. There is a mechanism in the cache that monitors cache miss rates for each program and allocates 1−15 ways to each program such that the overall number of cache misses is reduced. Assume that program A has an MPKI of 100 when it is assigned 1 MB of the cache. Each additional 1 MB assigned to program A reduces the MPKI by 1. Program B has an MPKI of 50 when it is assigned 1 MB of cache; each additional 1 MB assigned to program B reduces its MPKI by 2. What is the best allocation of ways to programs A and B?

2.24. [20] <2.1, 2.6> You are designing a PMD and optimizing it for low energy. The core, including a 16 KB L1 data cache, consumes 1 W whenever it is not in hibernation. If the core has a perfect L1 cache hit rate, it achieves an average CPI of 1 for a given task, that is, 1000 cycles to execute 1000 instructions. Each additional cycle accessing the L2 and beyond adds a stall cycle for the core. Based on the following specifications, what is the size of L2 cache that achieves the lowest energy for the PMD (core, L1, L2, memory) for that given task?

a. The core frequency is 1 GHz, and the L1 has an MPKI of 100.
b. A 256 KB L2 has a latency of 10 cycles, an MPKI of 20, and a background power of 0.2 W, and each L2 access consumes 0.5 nJ.
c. A 1 MB L2 has a latency of 20 cycles, an MPKI of 10, and a background power of 0.8 W, and each L2 access consumes 0.7 nJ.
d. The memory system has an average latency of 100 cycles and a background power of 0.5 W, and each memory access consumes 35 nJ.

2.25. [20] <2.2> Memory systems following DDR standards transfer 64 bits of data on every memory clock edge. To tolerate a single corrupted bit in every transfer (either from a random soft error or a hard error in some DRAM pin or wire), a reliable memory system attaches an 8-bit code to every 64-bit data transfer. This is typically a Hamming code used for single error correction and double error detection (SECDED). However, some hardware infrastructures need to provide even higher levels of reliability, for example, they may want to provide correction guarantees even if an entire DRAM chip and all its output bits fail (referred to as Chipkill). Can you design a simple memory system that can provide correct data even if one DRAM chip fails? What are the drawbacks to providing this higher reliability? While you may not be an expert in coding theory, you are likely familiar with parity and cyclic redundancy check (CRC) that are used in implementing RAID storage.

2.26. [20] <2.4> This chapter describes side channels, using shared cache access and prime-probe as the primary example. Can you enumerate a few other side channels in the memory hierarchy, how an attacker might exploit them, and how a processor architect might defend against those attacks?

2.27. [20] <2.3> Consider the following three 4-way set-associative L2 cache designs. (i) Parallel tag/data access: a 4-cycle 20 pJ tag lookup is performed in parallel with a 20-cycle 1000 pJ data array access; (ii) serial tag/data access: a 4-cycle 20 pJ tag lookup is first performed, which is followed by a 20-cycle 250 pJ data array access to a single way; and (iii) way-predicted tag/data access: a 1-cycle 5 pJ way predictor is first accessed, followed by parallel tag/data access to the predicted way (4 and 20 cycles for tag and data lookup, 5 pJ tag lookup and 250 pJ data array access). In case of a way misprediction a parallel tag/data access across the other three ways is then performed (4 and 20 cycles for tag and data lookup, 15 pJ tag lookup and 750 pJ data array access). Rank these three designs in terms of performance and energy. Assume a way prediction accuracy of 80%.

2.28. [20] <2.3> The chapter introduces the possibility of a large L4 DRAM cache, culminating in the design of the “Alloy Cache.” To alleviate the high cost of DRAM access, the Alloy Cache was organized as a direct-mapped cache. Conventional wisdom states that direct-mapped caches can suffer from a high rate of conflict misses. Do you think this will be a problem for the Alloy Cache, and if yes, how can this problem be alleviated? Further, discuss the trade-offs in using a large data block size within the Alloy Cache. What are the implications of using a data block size of, say, 4KB?

2.29. 20] <2.3> Consider a baseline Alloy Cache design where it takes 30 cycles to access the tag, followed by an additional 4 cycles to receive the data. If a miss is discovered after the tag access, a memory access is initiated, which on average takes 100 cycles. Since it takes so long to discover an Alloy Cache miss, a 1-cycle hit/miss predictor is introduced in parallel with the Alloy Cache access. If a hit is predicted, the operation sequence is exactly as the baseline. If a miss is predicted, the memory access is initiated in parallel with the Alloy Cache lookup.

Calculate the average latency in the Alloy Cache and beyond for both of these designs. Assume that the hit/miss predictor has the following profile: (i) correctly predicts a hit for 80% of all accesses, (ii) correctly predicts a miss for 5% of all accesses, (iii) incorrectly predicts a hit for 5% of all accesses, and (iv) incorrectly predicts a miss for 10% of all accesses.

2.30. [10/10] <2.1, 2.2, 2.3> The ways of a set can be viewed as a priority list, ordered from high priority to low priority. Every time the set is touched, the list can be reorganized to change block priorities. With this view, cache management policies can be decomposed into three subpolicies: Insertion, Promotion, and Victim Selection. Insertion defines where newly fetched blocks are placed in the priority list. Promotion defines how a block's position in the list is changed every time it is touched (a cache hit). Victim Selection defines which entry of the list is evicted to make room for a new block when there is a cache miss.

a. Can you frame the LRU cache policy in terms of the Insertion, Promotion, and Victim Selection subpolicies?
b. Can you define other Insertion and Promotion policies that may be competitive and worth exploring further?

2.31. [15] <2.1, 2.3> In a processor that is running multiple programs the last-level cache is typically shared by all the programs. This leads to interference, where one program's behavior and cache footprint can impact the cache available to other programs. First, this is a problem from a quality-of-service (QoS) perspective, where the interference leads to a program receiving fewer resources and lower performance than promised, say, by the operator of a cloud service. Second, this is a problem in terms of privacy. Based on the interference it sees, a program can infer the memory access patterns of other programs, using a side-channel attack. What policies can you add to your last-level cache so that the behavior of one program is immune to the behavior of other programs sharing the cache?

2.32. [15] <2.3> A large multimegabyte L3 cache can take tens of cycles to access because of the long wires that have to be traversed. For example, it may take 20 cycles to access a 16 MB L3 cache. Instead of organizing the 16 MB cache such that every access takes 20 cycles, we can organize the cache so that it is an array of smaller cache banks. Some of these banks may be closer to the processor core, while others may be farther. This leads to NonUniform Cache Access (NUCA), where 2 MB of the cache may be accessible in 8 cycles, the next 2 MB in 10 cycles, and so on until the last 2 MB is accessed in 22 cycles. What new policies can you introduce to maximize performance in a NUCA cache?

2.33. [10/10/10] <2.2> Consider a desktop system with a processor connected to a 2 GB DRAM with *error-correcting code (ECC)*. Assume that there is only one memory channel of width 72 bits (64 bits for data and 8 bits for ECC).

a. [10] <2.2> How many DRAM chips are on the DIMM if 1 Gb DRAM chips are used, and how many data I/Os must each DRAM have if only one DRAM connects to each DIMM data pin?

b. [10] <2.2> What burst length is required to support 32 B L2 cache blocks?
c. [10] <2.2> Calculate the peak bandwidth for DDR2-667 and DDR2-533 DIMMs for reads from an active page excluding the ECC overhead.

2.34. [20] <2.3> What are the main challenges in accurately tracking LRU information in a last-level cache?

2.35. [20] <2.3> A transistor is constantly leaking some current between power supply and ground. The energy consumed within a processor is classified as either dynamic energy, because of current flow when transistors switch, or as leakage energy, because of leakage current flow for every transistor every cycle. Dynamic energy is a function of switching activity, while leakage energy is a function of the number of transistors. More than half the area of a modern microprocessor is occupied by caches; they therefore are a dominant contributor to overall leakage energy. One technique to reduce leakage energy is to gate off the power supply to the transistor so that it no longer leaks. There is a catch: if you turn off power supply to the transistors in a cache block, the data stored in that cache block is lost. Note that if a cache block consumes X units of leakage energy in a cycle, fetching the cache block from memory consumes 1000X units of dynamic energy. Leverage the above technique and design a mechanism for the last level cache that significantly reduces overall system energy consumption while minimally impacting performance.

2.36. [12] <2.2> You are provisioning a server with 16-core 3 GHz CMP that can execute a workload with an overall CPI of 2.0 (assuming that L2 cache miss refills are not delayed). The L2 cache line size is 32 bytes. Assuming the system uses DDR4-2666 DIMMs, how many independent memory channels should be provided so that the system is not limited by memory bandwidth if the bandwidth required is sometimes twice the average? The workloads incur, on average, 6.67 L2 misses per 1 K instructions.

2.37. [15] <2.2> Consider a processor that has four memory channels. Should consecutive memory blocks be placed in the same bank, or should they be placed in different banks on different channels?

2.38. [15] <2.2> Whenever a computer is idle, we can either put it on standby (where DRAM is still active) or we can let it hibernate. Assume that, to hibernate, we have to copy just the contents of DRAM to a nonvolatile medium such as Flash. If reading or writing a cache line of size 64 bytes to Flash requires 2.56 J and DRAM requires 0.5 nJ, and if idle power consumption for DRAM is 1.6 W (for 8 GB), how long should a system be idle to benefit from hibernating? Assume a main memory of size 8 GB.

2.39. [15] <2.2, 2.7> As discussed in Section 2.7, the Intel i7 processor has an aggressive prefetcher. What are potential disadvantages of designing a prefetcher that is extremely aggressive?

This page intentionally left blank