Sixth Edition

John L. Hennessy | David A. Patterson

COMPUTER ARCHITECTURE

A Quantitative Approach

The goal for the I/O system in a computer with a cache is to prevent the stale data problem while interfering as little as possible. Many systems therefore prefer that I/O occur directly to main memory, with main memory acting as an I/O buffer. If a write-through cache were used, then memory would have an up-to-date copy of the information, and there would be no stale data issue for output. (This benefit is a reason processors used write through.) However, today write through is usually found only in first-level data caches backed by an L2 cache that uses write back.

Input requires some extra work. The software solution is to guarantee that no blocks of the input buffer are in the cache. A page containing the buffer can be marked as noncachable, and the operating system can always input to such a page. Alternatively, the operating system can flush the buffer addresses from the cache before the input occurs. A hardware solution is to check the I/O addresses on input to see if they are in the cache. If there is a match of I/O addresses in the cache, the cache entries are invalidated to avoid stale data. All of these approaches can also be used for output with write-back caches.

Processor cache coherency is a critical subject in the age of multicore processors, and we will examine it in detail in Chapter 5.

2.6 Putting It All Together: Memory Hierarchies in the ARM Cortex-A53 and Intel Core i7 6700

This section reveals the ARM Cortex-A53 (hereafter called the A53) and Intel Core i76700 (hereafter called i7) 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 i7, 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

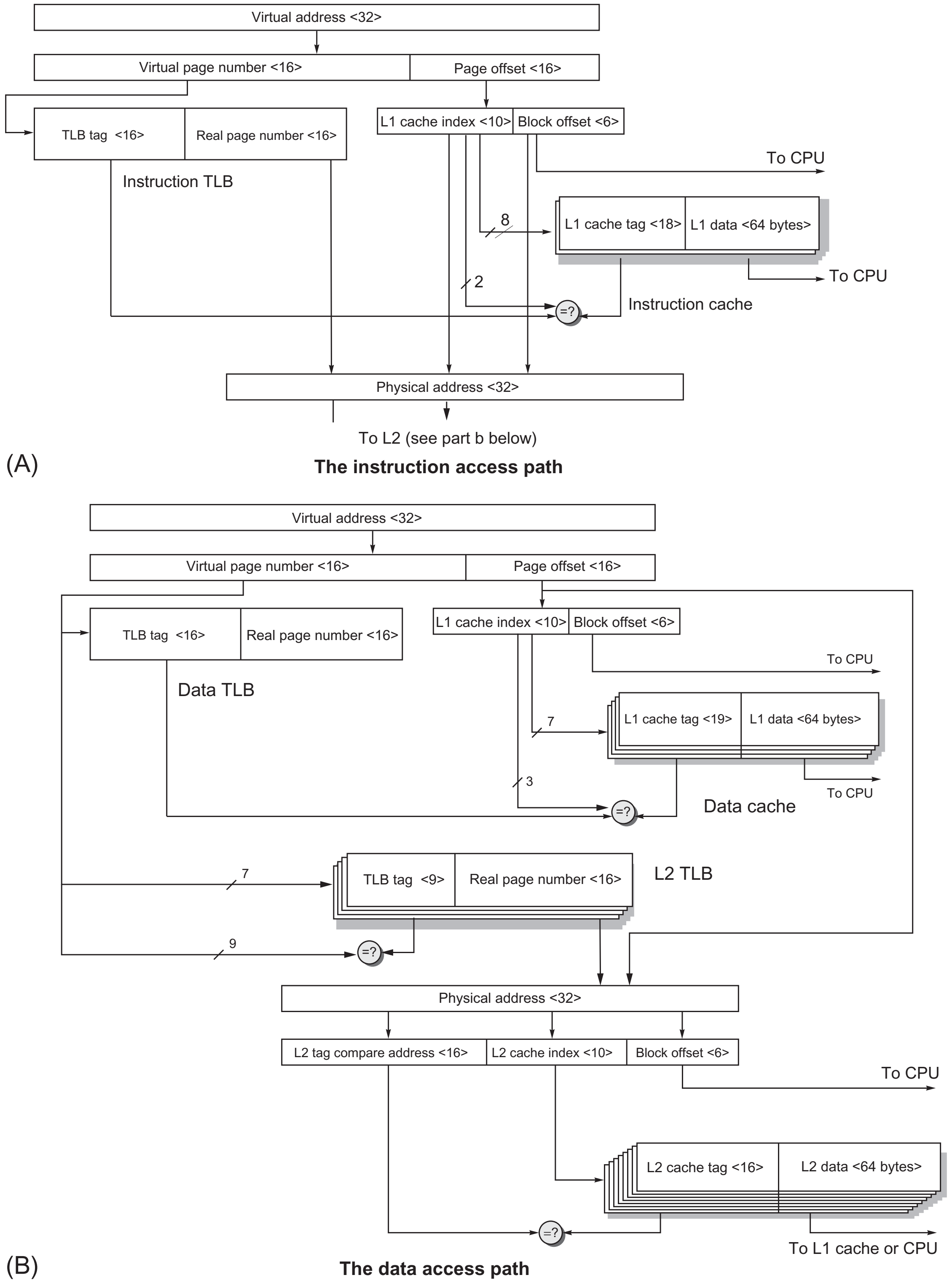
The Cortex-A53 is a configurable core that supports the ARMv8A instruction set architecture, which includes both 32-bit and 64-bit modes. The Cortex-A53 is delivered as an IP (intellectual property) core. IP cores are the dominant form of technology delivery in the embedded, PMD, and related markets; billions of ARM and MIPS 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 criteria in battery-based PMDs. The A53 core is 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 2.19 summarizes the organization of the memory hierarchy. The critical term 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; such aliases are avoided by hardware detection on a miss as in Section B.3 of Appendix B. Figure 2.20 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.

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 2.19 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 MicroTLB 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.



The instruction access path

The data access path

Figure 2.20 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 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.

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 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 2.21 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 in Figure 2.19, Figure 2.22 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 for the benchmarks that stress the memory system. In the next chapter, we will examine the impact of the cache misses on overall CPI.

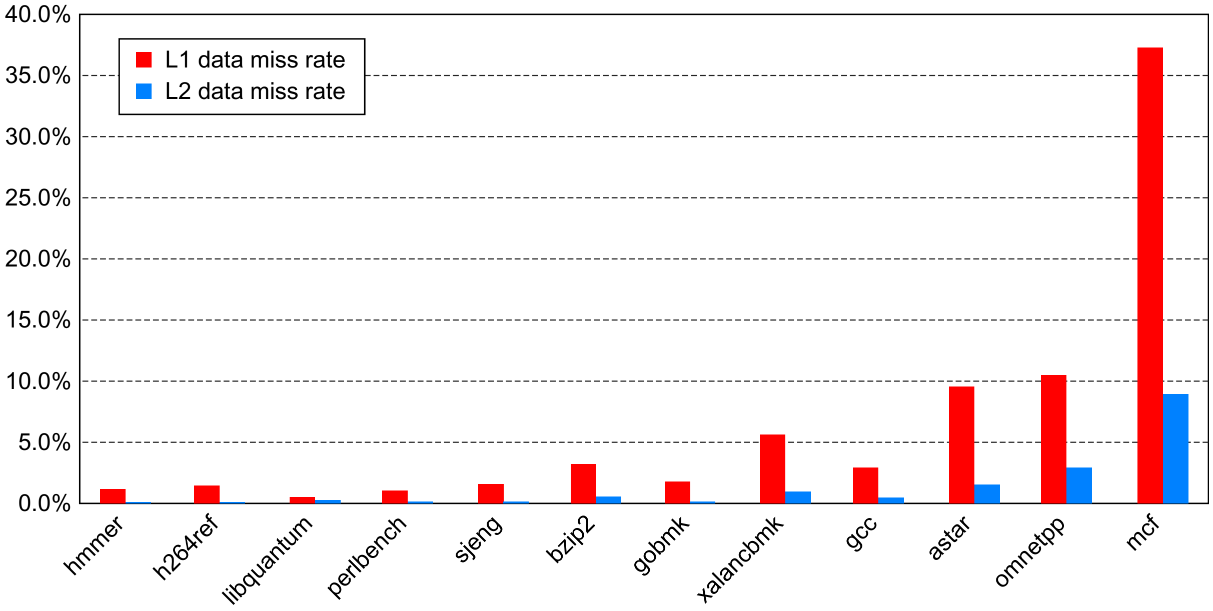


Figure 2.21 The data miss rate for ARM with a 32 KiB L1 and the global data miss rate for a 1 MiB L2 using the SPECInt2006 benchmarks are significantly affected by the applications. Applications with larger memory footprints tend to have higher miss rates in both L1 and L2. Note that the L2 rate is the global miss rate that is counting all references, including those that hit in L1. MCF is known as a cache buster.

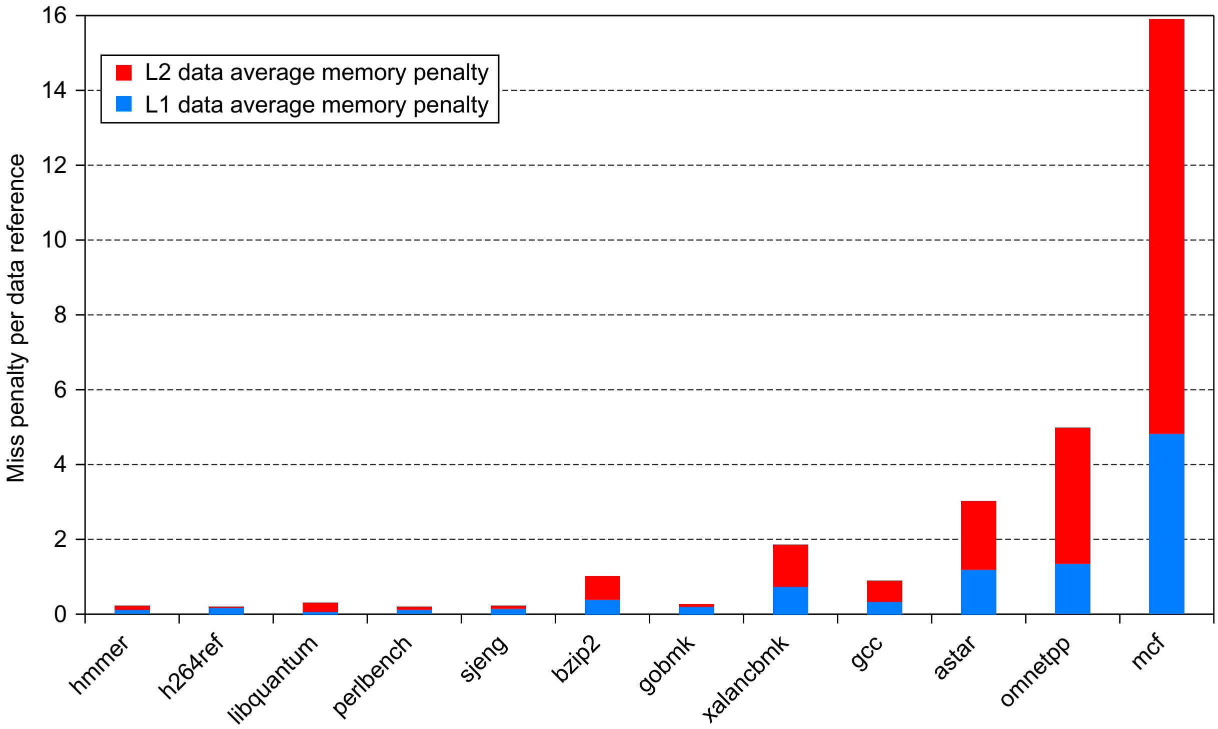


Figure 2.22 The average memory access penalty per data memory reference coming from L1 and L2 is shown for the A53 processor when running SPECInt2006. Although the miss rates for L1 are significantly higher, the L2 miss penalty, which is more than five times higher, means that the L2 misses can contribute significantly.

The Intel Core i7 6700

The i7 supports the x86-64 instruction set architecture, a 64-bit extension of the 80x86 architecture. The i7 is an out-of-order execution processor that includes four cores. In this chapter, we focus on the memory system design and performance from the viewpoint of a single core. The system performance of multiprocessor designs, including the i7 multicore, is examined in detail in Chapter 5.

Each core in an i7 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. The i7 can also support up to two simultaneous threads per processor, using a technique called simultaneous multithreading, described in Chapter 4. In 2017 the fastest i7 had a clock rate of 4.0 GHz (in Turbo Boost mode), which yielded a peak instruction execution rate of 16 billion instructions per second, or 64 billion instructions per second for the four-core design. Of course, there is a big gap between peak and sustained performance, as we will see over the next few chapters.

The i7 can support up to three memory channels, each consisting of a separate set of DIMMs, and each of which can transfer in parallel. Using DDR3-1066 (DIMM PC8500), the i7 has a peak memory bandwidth of just over 25 GB/s.

i7 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 2.23.

Figure 2.24 summarizes the i7'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. Some versions of the i7 6700 will support a fourth-level cache using HBM packaging.

Figure 2.25 is labeled with the steps of an access to the memory hierarchy. First, the PC is sent to the instruction cache. The instruction cache index is

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

Characteristic	Instruction TLB	Data DLB	Second-level TLB
Entries	128	64	1536
Associativity	8-way	4-way	12-way
Replacement	Pseudo-LRU	Pseudo-LRU	Pseudo-LRU
Access latency	1 cycle	1 cycle	8 cycles
Miss	9 cycles	9 cycles	Hundreds of cycles to access page table

Figure 2.23 Characteristics of the i7's TLB structure, which has separate 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 a limited number of entries of large 2–4 MiB pages; only 4 KiB pages are supported in the second-level TLB. The i7 has the ability to handle two L2 TLB misses in parallel. See Section L.3 of online Appendix L for more discussion of multilevel TLBs and support for multiple page sizes.

Characteristic	L1	L2	L3
Size	32 KiB I/32 KiB D	256 KiB	2 MiB per core
Associativity	both 8-way	4-way	16-way
Access latency	4 cycles, pipelined	12 cycles	44 cycles
Replacement scheme	Pseudo-LRU	Pseudo-LRU	Pseudo-LRU but with an ordered selection algorithm

Figure 2.24 Characteristics of the three-level cache hierarchy in the i7. 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 and is a total of 2 MiB per core. 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. (That is, an L1 write miss does not cause the line to be allocated.) L3 is inclusive of L1 and L2; we explore this property in further detail when we explain multiprocessor caches. Replacement is by a variant on pseudo-LRU; in the case of L3, the block replaced is always the lowest numbered way whose access bit is off. This is not quite random but is easy to compute.

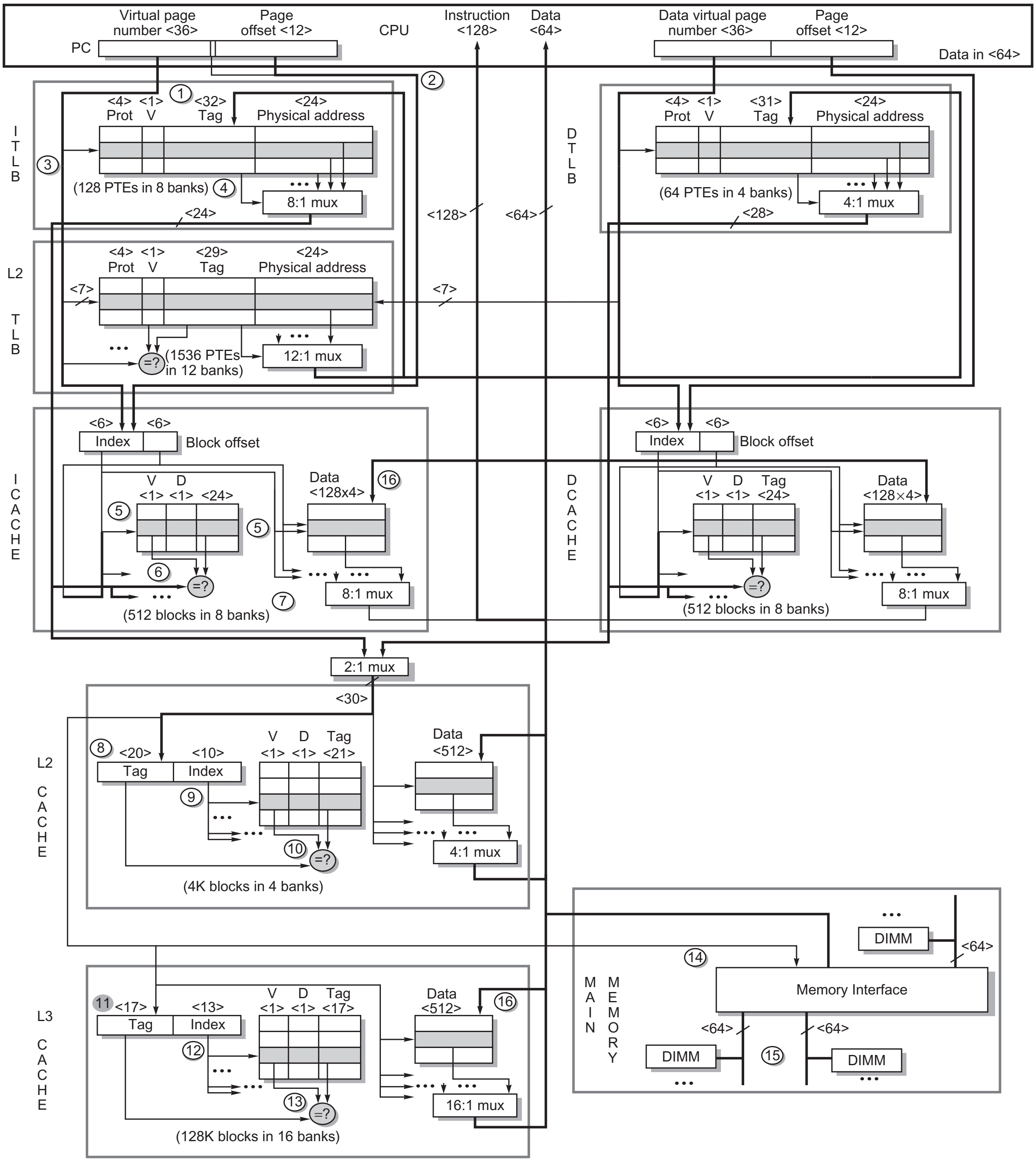


Figure 2.25 The Intel i7 memory hierarchy and the steps in both instruction and data access. We show only reads. Writes are similar, except that misses are handled by simply placing the data in a write buffer, because the L1 cache is not write-allocated.

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. The previous versions of the i7 used a four-way set associative I-cache, meaning that a block corresponding to a virtual address could actually 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 number will be identical.

The instruction TLB is accessed to find a match between the address and a valid page table entry (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, which contains 1536 PTEs of 4 KiB page sizes and is 12-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 i7 expects 16 bytes each instruction fetch, an additional 2 bits are used from the 6-bit block offset to select the appropriate 16 bytes. Therefore 6+2 or 8 bits are used to send 16 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

$$2^{\text{Index}} = \frac{\text{Cache size}}{\text{Block size} \times \text{Set associativity}} = \frac{256\text{K}}{64 \times 4} = 1024 = 2^{10}$$

so the 30-bit block address (36-bit physical address – 6-bit block offset) is divided into a 20-bit tag and a 10-bit index (step 8). Once again, the index and tag are sent to the four banks of the unified 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 12-cycle latency at a rate of 8 bytes per clock cycle.

If the L2 cache misses, the L3 cache is accessed. For a four-core i7, which has an 8 MiB L3, the index size is

$$2^{\text{Index}} = \frac{\text{Cache size}}{\text{Block size} \times \text{Set associativity}} = \frac{8\text{M}}{64 \times 16} = 8192 = 2^{13}$$

The 13-bit index (step 11) is sent to all 16 banks of the L3 (step 12). The L3 tag, which is $36 - (13 + 6) = 17$ 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 42 clock cycles, at a rate of 16 bytes per clock and placed into both L1 and L3. If L3 misses, a memory access is initiated.

If the instruction is not found in the L3 cache, the on-chip memory controller must get the block from main memory. The i7 has three 64-bit memory channels that can act as one 192-bit channel, because there is only one memory controller and the same address is sent on both channels (step 14). Wide transfers happen when both channels have identical DIMMs. Each channel supports up to four DDR DIMMs (step 15). When the data return they are placed into L3 and L1 (step 16) because L3 is inclusive.

The total latency of the instruction miss that is serviced by main memory is approximately 42 processor cycles to determine that an L3 miss has occurred, plus the DRAM latency for the critical instructions. For a single-bank DDR4-2400 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 16 bytes per I/O bus clock cycle, which takes another 5 ns or 20 clock cycles.

Because the second-level cache is a write-back cache, any miss can lead to an old block being written back to memory. The i7 has a 10-entry 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. A similar buffer is used between the L1 and L2 caches. If this initial instruction is a load, the data address is sent to the data cache and data TLBs, acting very much like an instruction cache access.

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 I7 also supports prefetching for L1 and L2 from the next level in the hierarchy. In most cases, the prefetched line is simply the next block in the cache. By prefetching only for L1 and L2, high-cost unnecessary fetches to memory are avoided.

Performance of the i7 memory system

We evaluate the performance of the i7 cache structure using the SPECint2006 benchmarks. The data in this section were collected by Professor Lu Peng and PhD student Qun Liu, both of Louisiana State University. Their analysis is based on earlier work (see Prakash and Peng, 2008).

The complexity of the i7 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 on page 110, 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. In addition to demand misses, there are prefetch misses for both instructions and data.

The i7's instruction fetch unit attempts to fetch 16 bytes every cycle, which complicates comparing instruction cache miss rates because multiple instructions are fetched every cycle (roughly 4.5 on average). In fact, the entire 64-byte cache line is read and subsequent 16-byte fetches do not require additional accesses. Thus misses are tracked only on the basis of 64-byte blocks. The 32 KiB, eight-way set associative instruction cache leads to a very low instruction miss rate for the SPECint2006 programs. If, for simplicity, we measure the miss rate of SPECint2006 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), which has a 2.9% miss rate. 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.

The frequency at which the instruction fetch unit is stalled waiting for the I-cache misses is similarly small (as a percentage of total cycles) increasing to 2% for two benchmarks and 12% for XALANCBMK, which has the highest I-cache miss rate. In the next chapter, we will see how stalls in the IFU contribute to overall reductions in pipeline throughput in the i7.

The L1 data cache is more interesting and even trickier to evaluate because in addition to the effects of prefetching and speculation, the L1 data cache is not write-allocated, and writes to cache blocks that are not present are not treated as misses. For this reason, we focus only on memory reads. The performance monitor measurements in the i7 separate out prefetch accesses from demand accesses, but only keep demand accesses for those instructions that graduate. The effect of speculative instructions that do not graduate is not negligible, although pipeline effects probably dominate secondary cache effects caused by speculation; we will return to the issue in the next chapter.

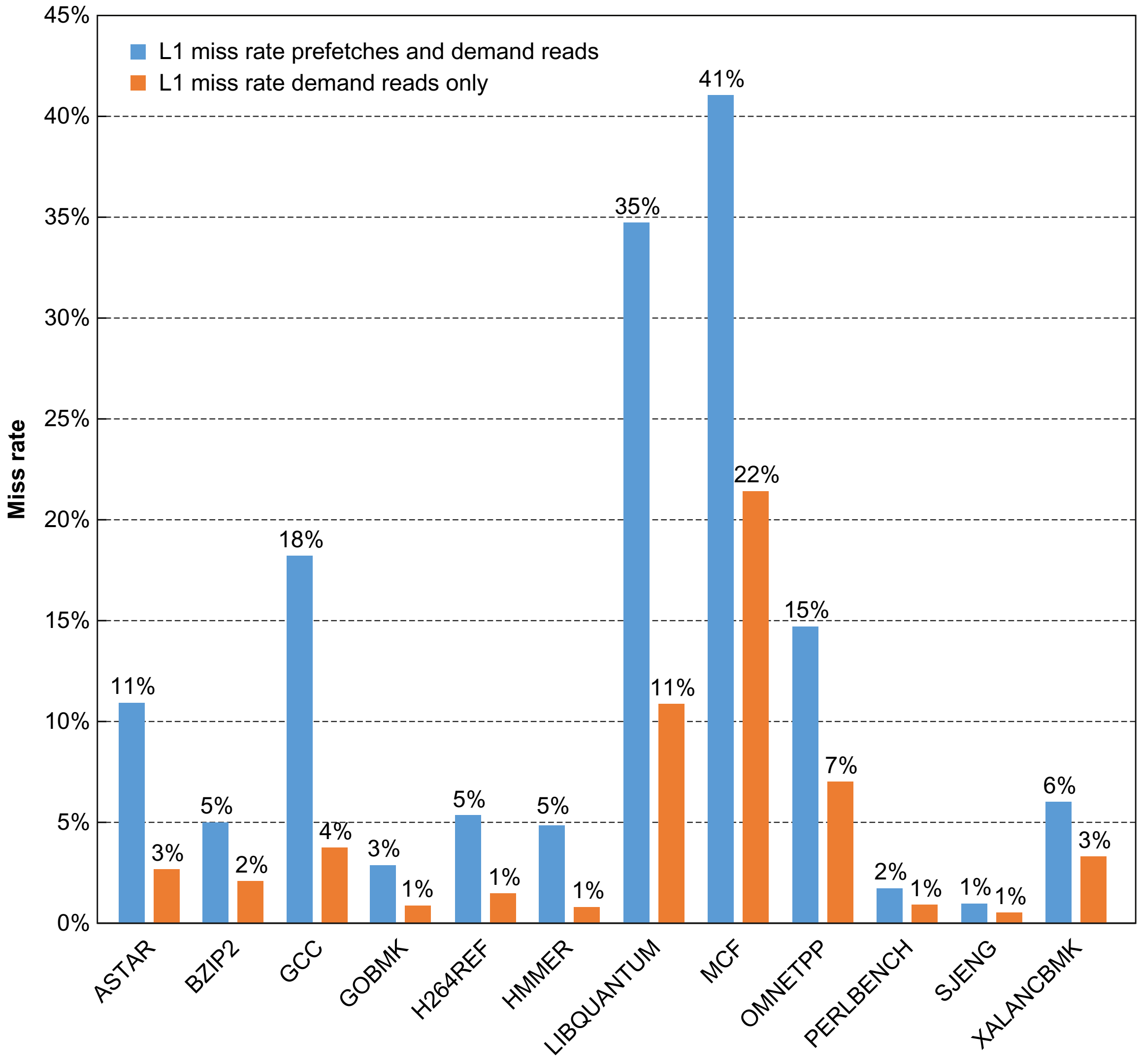


Figure 2.26 The L1 data cache miss rate for the SPECint2006 benchmarks is shown in two ways relative to the demand L1 reads: one including both demand and prefetch accesses and one including only demand accesses. The i7 separates out L1 misses for a block not present in the cache and L1 misses for a block already outstanding that is being prefetched from L2; we treat the latter group as hits because they would hit in a blocking cache. These data, like the rest in this section, were collected by Professor Lu Peng and PhD student Qun Liu, both of Louisiana State University, based on earlier studies of the Intel Core Duo and other processors (see Peng et al., 2008).

To address these issues, while keeping the amount of data reasonable, Figure 2.26 shows the L1 data cache misses in two ways:

1. The L1 miss rate relative to demand references given by the L1 miss rate including prefetches and speculative loads/L1 demand read references for those instructions that graduate.

2. The demand miss rate given by L1 demand misses/L1 demand read references, both measurements only for instructions that graduate.

On average, the miss rate including prefetches is 2.8 times as high as the demand-only miss rate. Comparing this data to that from the earlier i7 920, which had the same size L1, we see that the miss rate including prefetches is higher on the newer i7, but the number of demand misses, which are more likely to cause a stall, are usually fewer.

To understand the effectiveness of the aggressive prefetch mechanisms in the i7, let's look at some measurements of prefetching. Figure 2.27 shows both the fraction of L2 requests that are prefetches versus demand requests and the prefetch miss rate. The data are probably astonishing at first glance: there are roughly 1.5 times as many prefetches as there are L2 demand requests, which come directly from L1 misses. Furthermore, the prefetch miss rate is amazingly high, with an average miss rate of 58%. Although the prefetch ratio varies considerably, the prefetch miss rate is always significant. At first glance, you might conclude that the designers made a mistake: they are prefetching too much, and the miss rate is too high. Notice, however, that the benchmarks with the higher prefetch ratios (ASTAR, BZIP2, HMMER, LIBQUANTUM, and OMNETPP) also show the greatest gap between the prefetch miss rate and the demand miss rate, more than a factor of 2 in each case. The aggressive prefetching is trading prefetch misses, which occur earlier, for demand misses, which occur later; and as a result, a pipeline stall is less likely to occur due to the prefetching.

Similarly, consider the high prefetch miss rate. Suppose that the majority of the prefetches are actually useful (this is hard to measure because it involves tracking individual cache blocks), then a prefetch miss indicates a likely L2 cache miss in the future. Uncovering and handling the miss earlier via the prefetch is likely to reduce the stall cycles. Performance analysis of speculative superscalars, like the i7, has shown that cache misses tend to be the primary cause of pipeline stalls, because it is hard to keep the processor going, especially for longer running L2 and L3 misses. The Intel designers could not easily increase the size of the caches without incurring both energy and cycle time impacts; thus the use of aggressive prefetching to try to lower effective cache miss penalties is an interesting alternative approach.

With the combination of the L1 demand misses and prefetches going to L2, roughly 17% of the loads generate an L2 request. Analyzing L2 performance requires including the effects of writes (because L2 is write-allocated), as well as the prefetch hit rate and the demand hit rate. Figure 2.28 shows the miss rates of the L2 caches for demand and prefetch accesses, both versus the number of L1 references (reads and writes). As with L1, prefetches are a significant contributor, generating 75% of the L2 misses. Comparing the L2 demand miss rate with that of earlier i7 implementations (again with the same L2 size) shows that the i7 6700 has a lower L2 demand miss rate by an approximate factor of 2, which may well justify the higher prefetch miss rate.

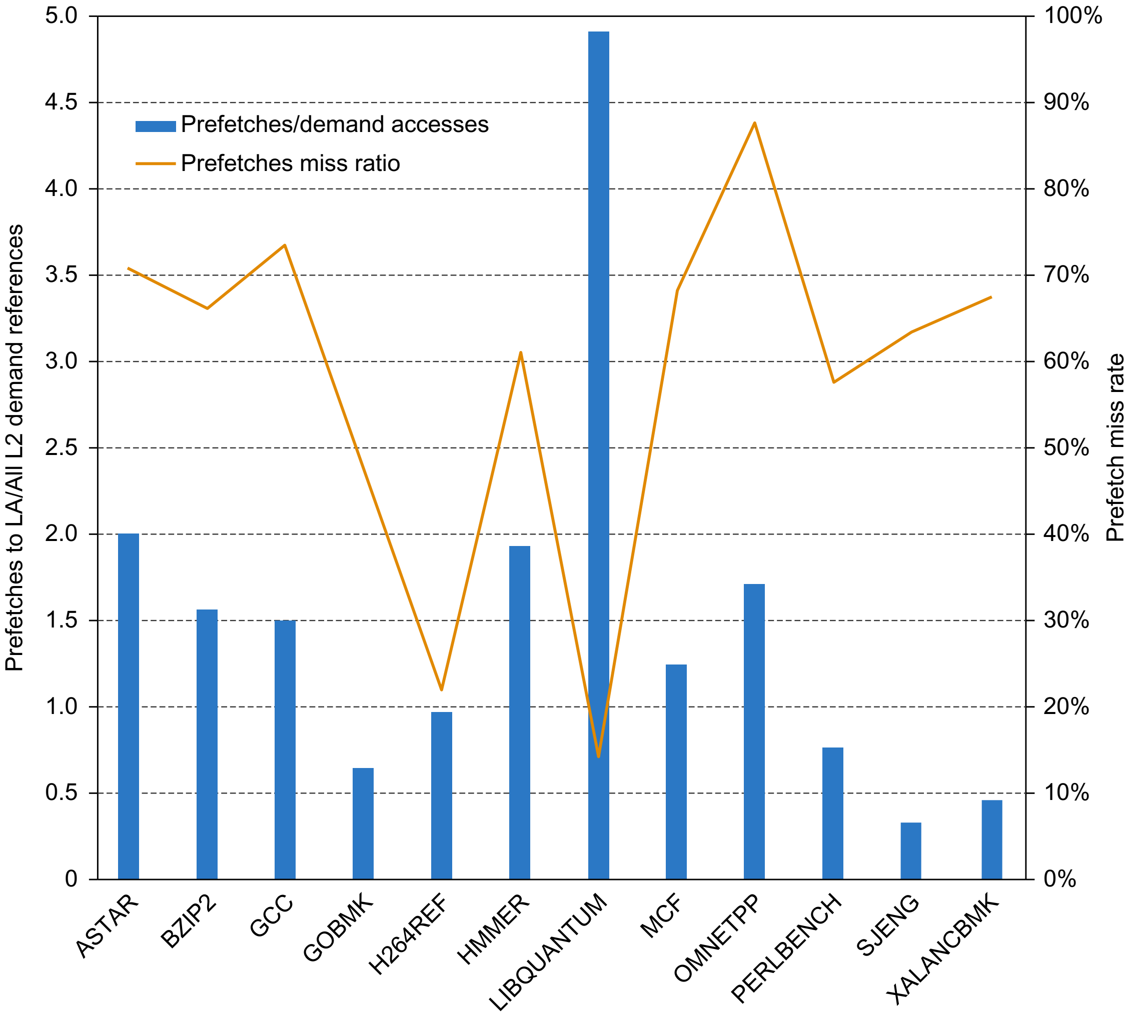


Figure 2.27 The fraction of L2 requests that are prefetches is shown via the columns and the left axis. The right axis and the line shows the prefetch hit rate. These data, like the rest in this section, were collected by Professor Lu Peng and PhD student Qun Liu, both of Louisiana State University, based on earlier studies of the Intel Core Duo and other processors (see Peng et al., 2008).

Because the cost for a miss to memory is over 100 cycles and the average data miss rate in L2 combining both prefetch and demand misses is over 7%, L3 is obviously critical. Without L3 and assuming that about one-third of the instructions are loads or stores, L2 cache misses could add over two cycles per instruction to the CPI! Obviously, prefetching past L2 would make no sense without an L3.

In comparison, the average L3 data miss rate of 0.5% is still significant but less than one-third of the L2 demand miss rate and 10 times less than the L1 demand miss rate. Only in two benchmarks (OMNETPP and MCF) is the L3 miss rate

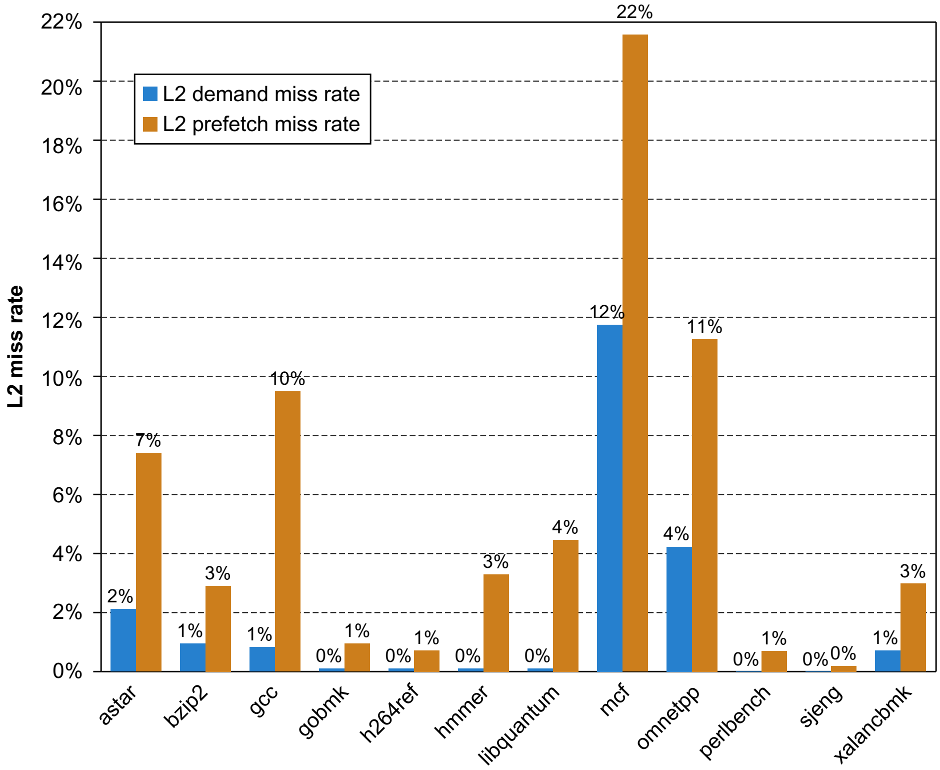


Figure 2.28 The L2 demand miss rate and prefetch miss rate, both shown relative to all the references to L1, which also includes prefetches, speculative loads that do not complete, and program-generated loads and stores (demand references). These data, like the rest in this section, were collected by Professor Lu Peng and PhD student Qun Liu, both of Louisiana State University.

above 0.5%; in those two cases, the miss rate of about 2.3% likely dominates all other performance losses. In the next chapter, we will examine the relationship between the i7 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 2.29 shows the instruction miss rates and data miss rates for three programs from the SPEC2000 benchmark suite as cache size varies. Depending on the

B

Review of Memory Hierarchy

Cache: a safe place for hiding or storing things.

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

B.1 Introduction

This appendix is a quick refresher of the memory hierarchy, including the basics of cache and virtual memory, performance equations, and simple optimizations. This first section reviews the following 36 terms:

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

If this review goes too quickly, you might want to look at Chapter 7 in *Computer Organization and Design*, which we wrote for readers with less experience.

Cache is the name given to the highest or first level of the memory hierarchy encountered once the address leaves the processor. Because the principle of locality applies at many levels, and taking advantage of locality to improve performance is popular, the term *cache* is now applied whenever buffering is employed to reuse commonly occurring items. Examples include *file caches*, *name caches*, and so on.

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

The time required for the cache miss depends on both the latency and bandwidth of the memory. Latency determines the time to retrieve the first word of the block, and bandwidth determines the time to retrieve the rest of this block. A cache miss is handled by hardware and causes processors using in-order execution to pause, or stall, until the data are available. With out-of-order execution, an instruction using the result must still wait, but other instructions may proceed during the miss.

Similarly, not all objects referenced by a program need to reside in main memory. *Virtual memory* means some objects may reside on disk. The address space is

Level	1	2	3	4
Name	Registers	Cache	Main memory	Disk storage
Typical size	<4 KiB	32 KiB to 8 MiB	<1 TB	>1 TB
Implementation technology	Custom memory with multiple ports, CMOS	On-chip CMOS SRAM	CMOS DRAM	Magnetic disk or FLASH
Access time (ns)	0.1–0.2	0.5–10	30–150	5,000,000
Bandwidth (MiB/sec)	1,000,000–10,000,000	20,000–50,000	10,000–30,000	100–1000
Managed by	Compiler	Hardware	Operating system	Operating system
Backed by	Cache	Main memory	Disk or FLASH	Other disks and DVD

Figure B.1 The typical levels in the hierarchy slow down and get larger as we move away from the processor for a large workstation or small server. Embedded computers might have no disk storage and much smaller memories and caches. Increasingly, FLASH is replacing magnetic disks, at least for first level file storage. The access times increase as we move to lower levels of the hierarchy, which makes it feasible to manage the transfer less responsively. The implementation technology shows the typical technology used for these functions. The access time is given in nanoseconds for typical values in 2017; these times will decrease over time. Bandwidth is given in megabytes per second between levels in the memory hierarchy. Bandwidth for disk/FLASH storage includes both the media and the buffered interfaces.

usually broken into fixed-size blocks, called *pages*. At any time, each page resides either in main memory or on disk. When the processor references an item within a page that is not present in the cache or main memory, a *palt* occurs, and the entire page is moved from the disk to main memory. Because page faults take so long, they are handled in software and the processor is not stalled. The processor usually switches to some other task while the disk access occurs. From a high-level perspective, the reliance on locality of references and the relative relationships in size and relative cost per bit of cache versus main memory are similar to those of main memory versus disk.

Figure B.1 shows the range of sizes and access times of each level in the memory hierarchy for computers ranging from high-end desktops to low-end servers.

Cache Performance Review

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

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

This equation assumes that the CPU clock cycles include the time to handle a cache hit and that the processor is stalled during a cache miss. Section B.2 reexamines this simplifying assumption.

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

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

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

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

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

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

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

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

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

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

Example Assume we have a computer where the cycles per instruction (CPI) is 1.0 when all memory accesses hit in the cache. The only data accesses are loads and stores, and these total 50% of the instructions. If the miss penalty is 50 clock cycles and the miss rate is 1%, how much faster would the computer be if all instructions were cache hits?

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

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

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

$$\begin{aligned}\text{Memory stall cycles} &= \text{IC} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss rate} \times \text{Miss penalty}\\ &= \text{IC} \times (1 + 0.5) \times 0.01 \times 50\\ &= \text{IC} \times 0.75\end{aligned}$$

where the middle term $(1+0.5)$ represents one instruction access and 0.5 data accesses per instruction. The total performance is thus

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

The performance ratio is the inverse of the execution times:

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

The computer with no cache misses is 1.75 times faster.

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

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

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

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

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

The advantage of misses per instruction is that it is independent of the hardware implementation. For example, speculative processors fetch about twice as many instructions as are actually committed, which can artificially reduce the miss rate if measured as misses per memory reference rather than per instruction. The drawback is that misses per instruction is architecture dependent; for example, the average number of memory accesses per instruction may be very different for an 80x86 versus RISC V. Thus, misses per instruction are most popular with architects working with a single computer family, although the similarity of RISC architectures allows one to give insights into others.

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

Answer Recomputing the memory stall cycles:

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

We get the same answer as on page B-5, showing equivalence of the two equations.

Four Memory Hierarchy Questions

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

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

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

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

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

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

Q1: Where Can a Block be Placed in a Cache?

Figure B.2 shows that the restrictions on where a block is placed create three categories of cache organization:

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

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

- If a block can be placed anywhere in the cache, the cache is said to be *fully associative*.

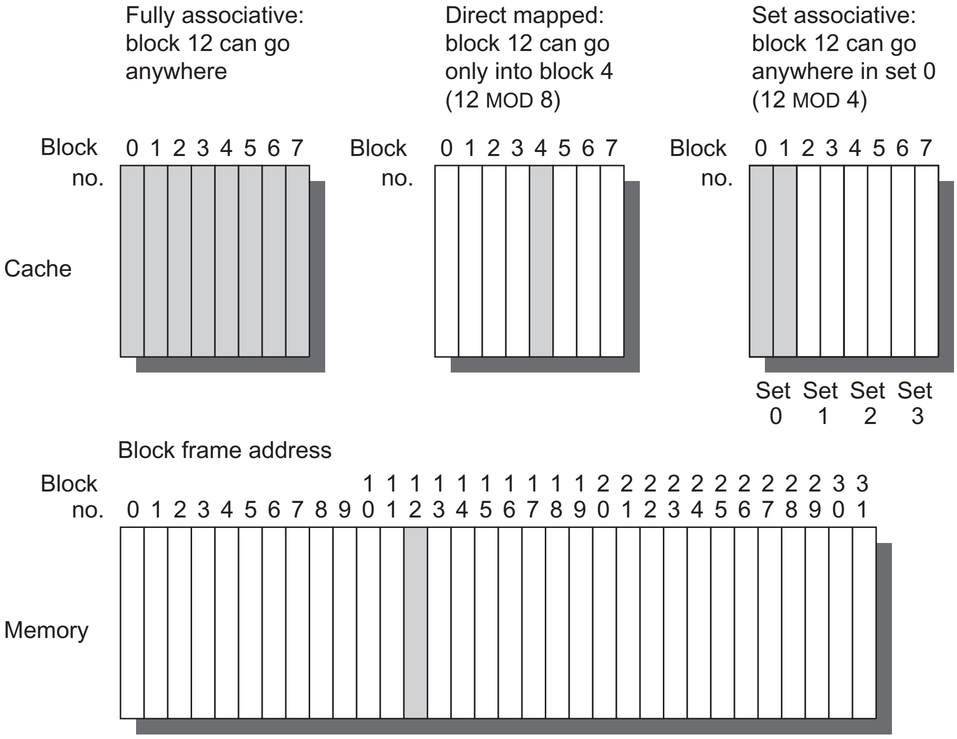


Figure B.2 This example cache has eight block frames and memory has 32 blocks. The three options for caches are shown left to right. In fully associative, block 12 from the lower level can go into any of the eight block frames of the cache. With direct mapped, block 12 can only be placed into block frame 4 (12 modulo 8). Set associative, which has some of both features, allows the block to be placed anywhere in set 0 (12 modulo 4). With two blocks per set, this means block 12 can be placed either in block 0 or in block 1 of the cache. Real caches contain thousands of block frames, and real memories contain millions of blocks. The set associative organization has four sets with two blocks per set, called *two-way set associative*. Assume that there is nothing in the cache and that the block address in question identifies lower-level block 12.

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

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

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

The range of caches from direct mapped to fully associative is really a continuum of levels of set associativity. Direct mapped is simply one-way set associative, and a fully associative cache with *m* blocks could be called "*m*-way set associative." Equivalently, direct mapped can be thought of as having *m* sets, and fully associative as having one set.

The vast majority of processor caches today are direct mapped, two-way set associative, or four-way set associative, for reasons we will see shortly.

Q2: How Is a Block Found If It Is in the Cache?

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

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

Before proceeding to the next question, let's explore the relationship of a processor address to the cache. Figure B.3 shows how an address is divided. The first division is between the *block address* and the *block offset.* The block frame address can be further divided into the *tag field* and the *index field.* The block offset field selects the desired data from the block, the index field selects the set, and the tag field is compared against it for a hit. Although the comparison could be made on more of the address than the tag, there is no need because of the following:

- The offset should not be used in the comparison, because the entire block is present or not, and hence all block offsets result in a match by definition.

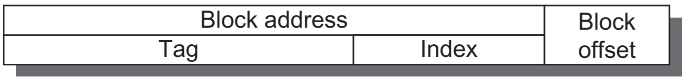


Figure B.3 The three portions of an address in a set associative or direct-mapped cache. The tag is used to check all the blocks in the set, and the index is used to select the set. The block offset is the address of the desired data within the block. Fully associative caches have no index field.

- Checking the index is redundant, because it was used to select the set to be checked. An address stored in set 0, for example, must have 0 in the index field or it couldn't be stored in set 0; set 1 must have an index value of 1; and so on. This optimization saves hardware and power by reducing the width of memory size for the cache tag.

If the total cache size is kept the same, increasing associativity increases the number of blocks per set, thereby decreasing the size of the index and increasing the size of the tag. That is, the tag-index boundary in Figure B.3 moves to the right with increasing associativity, with the end point of fully associative caches having no index field.

Q3: Which Block Should be Replaced on a Cache Miss?

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

- *Random*—To spread allocation uniformly, candidate blocks are randomly selected. Some systems generate pseudorandom block numbers to get reproducible behavior, which is particularly useful when debugging hardware.
- *Least recently used* (LRU)—To reduce the chance of throwing out information that will be needed soon, accesses to blocks are recorded. Relying on the past to predict the future, the block replaced is the one that has been unused for the longest time. LRU relies on a corollary of locality: if recently used blocks are likely to be used again, then a good candidate for disposal is the least recently used block.
- *First in, first out* (FIFO)—Because LRU can be complicated to calculate, this approximates LRU by determining the *oldest* block rather than the LRU.

A virtue of random replacement is that it is simple to build in hardware. As the number of blocks to keep track of increases, LRU becomes increasingly expensive and is usually only approximated. A common approximation (often called pseudo-LRU) has a set of bits for each set in the cache with each bit corresponding to a single way (a *way* is bank in a set associative cache; there are four ways in four-way set associative cache) in the cache. When a set is accessed, the bit corresponding to the way containing the desired block is turned on; if all the bits associated with a set are turned on, they are reset with the exception of the most recently turned on bit. When a block must be replaced, the processor chooses a block from the way whose bit is turned off, often randomly if more than one choice is available. This approximates LRU, because the block that is replaced will not have

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

Figure B.4 Data cache misses per 1000 instructions comparing least recently used, random, and first in, first out replacement for several sizes and associativities. There is little difference between LRU and random for the largest size cache, with LRU outperforming the others for smaller caches. FIFO generally outperforms random in the smaller cache sizes. These data were collected for a block size of 64 bytes for the Alpha architecture using 10 SPEC2000 benchmarks. Five are from SPECint2000 (gap, gcc, gzip, mcf, and perl) and five are from SPECfp2000 (applu, art, equake, lucas, and swim). We will use this computer and these benchmarks in most figures in this appendix.

been accessed since the last time that all the blocks in the set were accessed. Figure B.4 shows the difference in miss rates between LRU, random, and FIFO replacement.

Q4: What Happens on a Write?

Reads dominate processor cache accesses. All instruction accesses are reads, and most instructions don't write to memory. Figures A.32 and A.33 in Appendix A suggest a mix of 10% stores and 26% loads for RISC V programs, making writes 10%/(100% + 26% + 10%) or about 7% of the overall memory traffic. Of the *data cache* traffic, writes are 10%/(26% + 10%) or about 28%. Making the common case fast means optimizing caches for reads, especially because processors traditionally wait for reads to complete but need not wait for writes. Amdahl's Law (Section 1.9) reminds us, however, that high-performance designs cannot neglect the speed of writes.

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

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

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

- *Write through*—The information is written to both the block in the cache *and* to the block in the lower-level memory.
- *Write back*—The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.

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

Both write back and write through have their advantages. With write back, writes occur at the speed of the cache memory, and multiple writes within a block require only one write to the lower-level memory. Because some writes don't go to memory, write back uses less memory bandwidth, making write back attractive in multiprocessors. Since write back uses the rest of the memory hierarchy and memory interconnect less than write through, it also saves power, making it attractive for embedded applications.

Write through is easier to implement than write back. The cache is always clean, so unlike write back read misses never result in writes to the lower level. Write through also has the advantage that the next lower level has the most current copy of the data, which simplifies data coherency. Data coherency is important for multiprocessors and for I/O, which we examine in Chapter 4 and Appendix D. Multilevel caches make write through more viable for the upper-level caches, as the writes need only propagate to the next lower level rather than all the way to main memory.

As we will see, I/O and multiprocessors are fickle: they want write back for processor caches to reduce the memory traffic and write through to keep the cache consistent with lower levels of the memory hierarchy.

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

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

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

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

Example Assume a fully associative write-back cache with many cache entries that starts empty. Following is a sequence of five memory operations (the address is in square brackets):

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

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

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

For write allocate, the first accesses to 100 and 200 are misses, and the rest are hits because 100 and 200 are both found in the cache. Thus, the result for write allocate is two misses and three hits.

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

An Example: The Opteron Data Cache

To give substance to these ideas, Figure B.5 shows the organization of the data cache in the AMD Opteron microprocessor. The cache contains 65,536 (64 K) bytes of data in 64-byte blocks with two-way set associative placement, least-recently used replacement, write back, and write allocate on a write miss.

Let's trace a cache hit through the steps of a hit as labeled in Figure B.5. (The four steps are shown as circled numbers.) As described in Section B.5, the Opteron presents a 48-bit virtual address to the cache for tag comparison, which is simultaneously translated into a 40-bit physical address.

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

The physical address coming into the cache is divided into two fields: the 34-bit block address and the 6-bit block offset ($64=2^6$ and $34+6=40$). The block

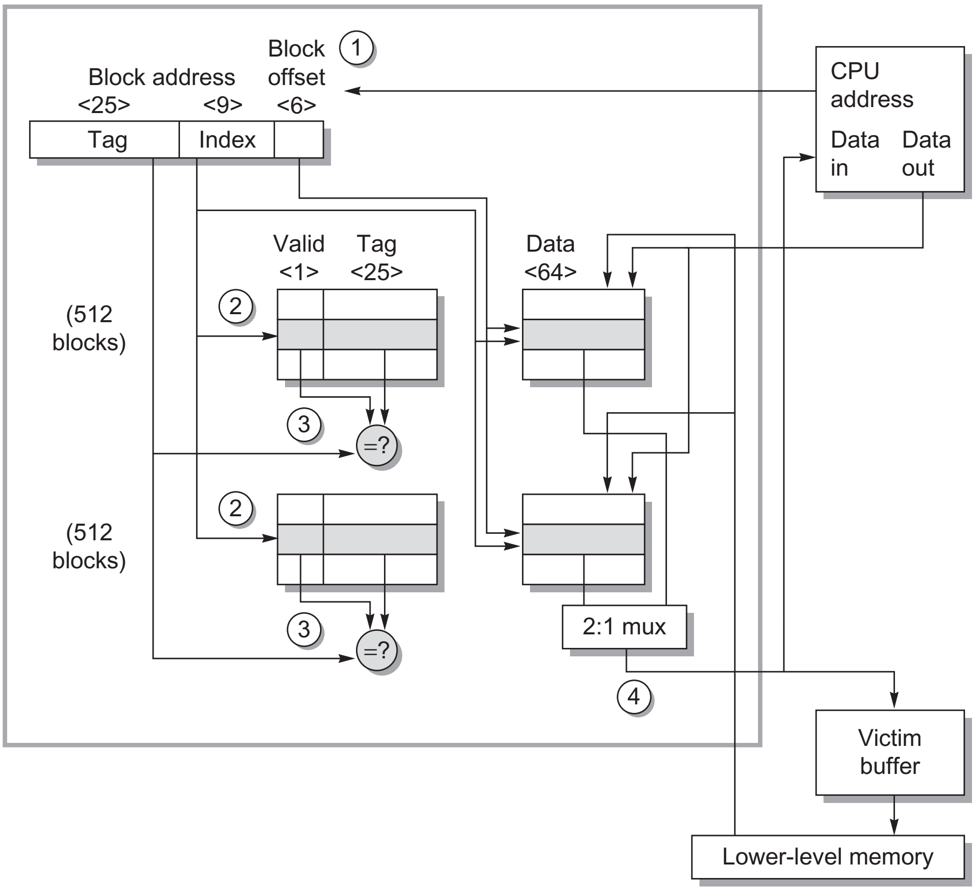


Figure B.5 The organization of the data cache in the Opteron microprocessor. The 64 KiB cache is two-way set associative with 64-byte blocks. The 9-bit index selects among 512 sets. The four steps of a read hit, shown as circled numbers in order of occurrence, label this organization. Three bits of the block offset join the index to supply the RAM address to select the proper 8 bytes. Thus, the cache holds two groups of 4096 64-bit words, with each group containing half of the 512 sets. Although not exercised in this example, the line from lower-level memory to the cache is used on a miss to load the cache. The size of address leaving the processor is 40 bits because it is a physical address and not a virtual address. Figure B.24 on page B-47 explains how the Opteron maps from virtual to physical for a cache access.

address is further divided into an address tag and cache index. Step 1 shows this division.

The cache index selects the tag to be tested to see if the desired block is in the cache. The size of the index depends on cache size, block size, and set associativity. For the Opteron cache the set associativity is set to two, and we calculate the index as follows:

$$2^{\text{Index}} = \frac{\text{Cache size}}{\text{Block size} \times \text{Set associativity}} = \frac{65{,}536}{64 \times 2} = 512 = 2^9$$

Hence, the index is 9 bits wide, and the tag is 34 − 9 or 25 bits wide. Although that is the index needed to select the proper block, 64 bytes is much more than the processor wants to consume at once. Hence, it makes more sense to organize the data portion of the cache memory 8 bytes wide, which is the natural data word of the 64-bit Opteron processor. Thus, in addition to 9 bits to index the proper cache block, 3 more bits from the block offset are used to index the proper 8 bytes. Index selection is step 2 in Figure B.5.

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

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

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

So far we have assumed the common case of a cache hit. What happens on a miss? On a read miss, the cache sends a signal to the processor telling it the data are not yet available, and 64 bytes are read from the next level of the hierarchy. The latency is 7 clock cycles to the first 8 bytes of the block, and then 2 clock cycles per 8 bytes for the rest of the block. Because the data cache is set associative, there is a choice on which block to replace. Opteron uses LRU, which selects the block that was referenced longest ago, so every access must update the LRU bit. Replacing a block means updating the data, the address tag, the valid bit, and the LRU bit.

Because the Opteron uses write back, the old data block could have been modified, and hence it cannot simply be discarded. The Opteron keeps 1 dirty bit per block to record if the block was written. If the "victim" was modified, its data and address are sent to the victim buffer. (This structure is similar to a *write buffer* in other computers.) The Opteron has space for eight victim blocks. In parallel with other cache actions, it writes victim blocks to the next level of the hierarchy. If the victim buffer is full, the cache must wait.

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

We have seen how it works, but the *data* cache cannot supply all the memory needs of the processor: the processor also needs instructions. Although a single cache could try to supply both, it can be a bottleneck. For example, when a load or store instruction is executed, the pipelined processor will simultaneously request both a data word *and* an instruction word. Hence, a single cache would present a structural hazard for loads and stores, leading to stalls. One simple way to conquer

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

Figure B.6 Miss per 1000 instructions for instruction, data, and unified caches of different sizes. The percentage of instruction references is about 74%. The data are for two-way associative caches with 64-byte blocks for the same computer and benchmarks as Figure B.4.

this problem is to divide it: one cache is dedicated to instructions and another to data. Separate caches are found in most recent processors, including the Opteron. Hence, it has a 64 KiB instruction cache as well as the 64 KiB data cache.

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

Figure B.6 shows that instruction caches have lower miss rates than data caches. Separating instructions and data removes misses due to conflicts between instruction blocks and data blocks, but the split also fixes the cache space devoted to each type. Which is more important to miss rates? A fair comparison of separate instruction and data caches to unified caches requires the total cache size to be the same. For example, a separate 16 KiB instruction cache and 16 KiB data cache should be compared with a 32 KiB unified cache. Calculating the average miss rate with separate instruction and data caches necessitates knowing the percentage of memory references to each cache. From the data in Appendix A we find the split is 100%/(100% + 26% + 10%) or about 74% instruction references to (26% + 10%)/(100% + 26% + 10%) or about 26% data references. Splitting affects performance beyond what is indicated by the change in miss rates, as we will see shortly.

B.2 Cache Performance

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

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

where *hit time* is the time to hit in the cache; we have seen the other two terms before. The components of average access time can be measured either in absolute time—say, 0.25–1.0 ns on a hit—or in the number of clock cycles that the processor waits for the memory—such as a miss penalty of 150–200 clock cycles. Remember that average memory access time is still an indirect measure of performance; although it is a better measure than miss rate, it is not a substitute for execution time.

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

Example Which has the lower miss rate: a 16 KiB instruction cache with a 16 KiB data cache or a 32 KiB unified cache? Use the miss rates in Figure B.6 to help calculate the correct answer, assuming 36% of the instructions are data transfer instructions. Assume a hit takes 1 clock cycle and the miss penalty is 100 clock cycles. A load or store hit takes 1 extra clock cycle on a unified cache if there is only one cache port to satisfy two simultaneous requests. Using the pipelining terminology of Chapter 3, the unified cache leads to a structural hazard. What is the average memory access time in each case? Assume write-through caches with a write buffer and ignore stalls due to the write buffer.

Answer First let's convert misses per 1000 instructions into miss rates. Solving the preceding general formula, the miss rate is

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

Because every instruction access has exactly one memory access to fetch the instruction, the instruction miss rate is

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

Because 36% of the instructions are data transfers, the data miss rate is

$$\text{Miss rate}_{\text{16 KB data}} = \frac{40.9/1000}{0.36} = 0.114$$

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

$$\text{Miss rate}_{\text{32 KB unified}} = \frac{43.3/1000}{1.00 + 0.36} = 0.0318$$

As stated herein, about 74% of the memory accesses are instruction references. Thus, the overall miss rate for the split caches is

$$(74\% \times 0.004) + (26\% \times 0.114) = 0.0326$$

Thus, a 32 KiB unified cache has a slightly lower effective miss rate than two 16 KiB caches.

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

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

Therefore, the time for each organization is

$$\begin{aligned}
&\text{Average memory access time}_{\text{split}}\\
&\quad = 74\% \times (1 + 0.004 \times 200) + 26\% \times (1 + 0.114 \times 200)\\
&\quad = (74\% \times 1.80) + (26\% \times 23.80) = 1.332 + 6.188 = 7.52\\
&\text{Average memory access time}_{\text{unified}}\\
&\quad = 74\% \times (1 + 0.0318 \times 200) + 26\% \times (1 + 1 + 0.0318 \times 200)\\
&\quad = (74\% \times 7.36) + (26\% \times 8.36) = 5.446 + 2.174 = 7.62
\end{aligned}$$

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

Average Memory Access Time and Processor Performance

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

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

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

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

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

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

We can now explore the impact of caches on performance.

Example Let's use an in-order execution computer for the first example. Assume that the cache miss penalty is 200 clock cycles, and all instructions usually take 1.0 clock cycles (ignoring memory stalls). Assume that the average miss rate is 2%, there is an average of 1.5 memory references per instruction, and the average number of cache misses per 1000 instructions is 30. What is the impact on performance when behavior of the cache is included? Calculate the impact using both misses per instruction and miss rate.

Answer

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

The performance, including cache misses, is

$$\begin{aligned}\text{CPU time}_{\text{with cache}} &= \text{IC} \times [1.0 + (30/1000 \times 200)] \times \text{Clock cycle time}\\ &= \text{IC} \times 7.00 \times \text{Clock cycle time}\end{aligned}$$

Now calculating performance using miss rate:

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

$$\begin{aligned}\text{CPU time}_{\text{with cache}} &= \text{IC} \times [1.0 + (1.5 \times 2\% \times 200)] \times \text{Clock cycle time}\\ &= \text{IC} \times 7.00 \times \text{Clock cycle time}\end{aligned}$$

The clock cycle time and instruction count are the same, with or without a cache. Thus, CPU time increases sevenfold, with CPI from 1.00 for a "perfect cache" to 7.00 with a cache that can miss. Without any memory hierarchy at all the CPI would increase again to $1.0 + 200 \times 1.5$ or 301—a factor of more than 40 times longer than a system with a cache!

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

1. The lower the $\text{CPI}_{\text{execution}}$, the higher the *relative* impact of a fixed number of cache miss clock cycles.
2. When calculating CPI, the cache miss penalty is measured in processor clock cycles for a miss. Therefore, even if memory hierarchies for two computers are

identical, the processor with the higher clock rate has a larger number of clock cycles per miss and hence a higher memory portion of CPI.

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

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

Example What is the impact of two different cache organizations on the performance of a processor? Assume that the CPI with a perfect cache is 1.0, the clock cycle time is 0.35 ns, there are 1.4 memory references per instruction, the size of both caches is 128 KiB, and both have a block size of 64 bytes. One cache is direct mapped and the other is two-way set associative. Figure B.5 shows that for set associative caches we must add a multiplexor to select between the blocks in the set depending on the tag match. Because the speed of the processor can be tied directly to the speed of a cache hit, assume the processor clock cycle time must be stretched 1.35 times to accommodate the selection multiplexor of the set associative cache. To the first approximation, the cache miss penalty is 65 ns for either cache organization. (In practice, it is normally rounded up or down to an integer number of clock cycles.) First, calculate the average memory access time and then processor performance. Assume the hit time is 1 clock cycle, the miss rate of a direct-mapped 128 KiB cache is 2.1%, and the miss rate for a two-way set associative cache of the same size is 1.9%.

Answer Average memory access time is

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

Thus, the time for each organization is

$$\text{Average memory access time}_{\text{1-way}} = 0.35 + (.021 \times 65) = 1.72\,\text{ns}$$

$$\text{Average memory access time}_{\text{2-way}} = 0.35 \times 1.35 + (.019 \times 65) = 1.71\,\text{ns}$$

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

The processor performance is

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

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

Substituting 65 ns for (Miss penalty × Clock cycle time), the performance of each cache organization is

$$\text{CPU time}_{\text{1-way}} = \text{IC} \times [1.0 \times 0.35 + (0.021 \times 1.4 \times 65)] = 2.26 \times \text{IC}$$

$$\text{CPU time}_{\text{2-way}} = \text{IC} \times [1.0 \times 0.35 \times 1.35 + (0.019 \times 1.4 \times 65)] = 2.20 \times \text{IC}$$

and relative performance is

$$\frac{\text{CPU time}_{\text{2-way}}}{\text{CPU time}_{\text{1-way}}} = \frac{2.26 \times \text{Instruction count}}{2.20 \times \text{Instruction count}} = 1.03$$

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

Miss Penalty and Out-of-Order Execution Processors

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

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

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

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

We now have to decide the following:

- *Length of memory latency*—What to consider as the start and the end of a memory operation in an out-of-order processor.
- *Length of latency overlap*—What is the start of overlap with the processor (or, equivalently, when do we say a memory operation is stalling the processor)?

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

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

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

Example Let's redo the preceding example, but this time we assume the processor with the longer clock cycle time supports out-of-order execution yet still has a direct-mapped cache. Assume 30% of the 65 ns miss penalty can be overlapped; that is, the average CPU memory stall time is now 45.5 ns.

Answer Average memory access time for the out-of-order (OOO) computer is

$$\text{Average memory access time}_{\text{1-way,OOO}} = 0.35 \times 1.35 + (0.021 \times 45.5) = 1.43\,\text{ns}$$

The performance of the OOO cache is

$$\text{CUP time}_{\text{1-way,OOO}} = \text{IC} \times [1.6 \times 0.35 \times 1.35 + (0.021 \times 1.4 \times 45.5)] = 2.09 \times \text{IC}$$

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

In summary, although the state of the art in defining and measuring memory stalls for out-of-order processors is complex, be aware of the issues because they significantly affect performance. The complexity arises because out-of-order processors tolerate some latency due to cache misses without hurting performance. Consequently, designers usually use simulators of the out-of-order processor and memory when evaluating trade-offs in the memory hierarchy to be sure that an improvement that helps the average memory latency actually helps program performance.

To help summarize this section and to act as a handy reference, Figure B.7 lists the cache equations in this appendix.

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

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

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

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

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

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

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

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

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

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

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

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

Figure B.7 Summary of performance equations in this appendix. The first equation calculates the cache index size, and the rest help evaluate performance. The final two equations deal with multilevel caches, which are explained early in the next section. They are included here to help make the figure a useful reference.

B.3 Six Basic Cache Optimizations

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

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

Hence, we organize six cache optimizations into three categories:

- *Reducing the miss rate*—larger block size, larger cache size, and higher associativity
- *Reducing the miss penalty*—multilevel caches and giving reads priority over writes
- *Reducing the time to hit in the cache*—avoiding address translation when indexing the cache

Figure B.18 on page B-40 concludes this section with a summary of the implementation complexity and the performance benefits of these six techniques.

The classical approach to improving cache behavior is to reduce miss rates, and we present three techniques to do so. To gain better insights into the causes of misses, we first start with a model that sorts all misses into three simple categories:

- *Compulsory*—The very first access to a block *cannot* be in the cache, so the block must be brought into the cache. These are also called *cold-start misses* or *first-reference misses.*
- *Capacity*—If the cache cannot contain all the blocks needed during execution of a program, capacity misses (in addition to compulsory misses) will occur because of blocks being discarded and later retrieved.
- *Conflict*—If the block placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory and capacity misses) will occur because a block may be discarded and later retrieved if too many blocks map to its set. These misses are also called *collision misses*. The idea is that hits in a fully associative cache that become misses in an n-way set-associative cache are due to more than n requests on some popular sets.

(Chapter 5 adds a fourth C, for *coherency* misses due to cache flushes to keep multiple caches coherent in a multiprocessor; we won't consider those here.)

Figure B.8 shows the relative frequency of cache misses, broken down by the three C's. Compulsory misses are those that occur in an infinite cache. Capacity misses are those that occur in a fully associative cache. Conflict misses are those that occur going from fully associative to eight-way associative, four-way associative, and so on. Figure B.9 presents the same data graphically. The top graph shows absolute miss rates; the bottom graph plots the percentage of all the misses by type of miss as a function of cache size.

To show the benefit of associativity, conflict misses are divided into misses caused by each decrease in associativity. Here are the four divisions of conflict misses and how they are calculated:

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

As we can see from the figures, the compulsory miss rate of the SPEC2000 programs is very small, as it is for many long-running programs.

Having identified the three C's, what can a computer designer do about them? Conceptually, conflicts are the easiest: Fully associative placement avoids all

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

Figure B.8 Total miss rate for each size cache and percentage of each according to the three C's. Compulsory misses are independent of cache size, while capacity misses decrease as capacity increases, and conflict misses decrease as associativity increases. Figure B.9 shows the same information graphically. Note that a direct-mapped cache of size *N* has about the same miss rate as a two-way set-associative cache of size *N*/2 up through 128 K. Caches larger than 128 KiB do not prove that rule. Note that the Capacity column is also the fully associative miss rate. Data were collected as in Figure B.4 using LRU replacement.

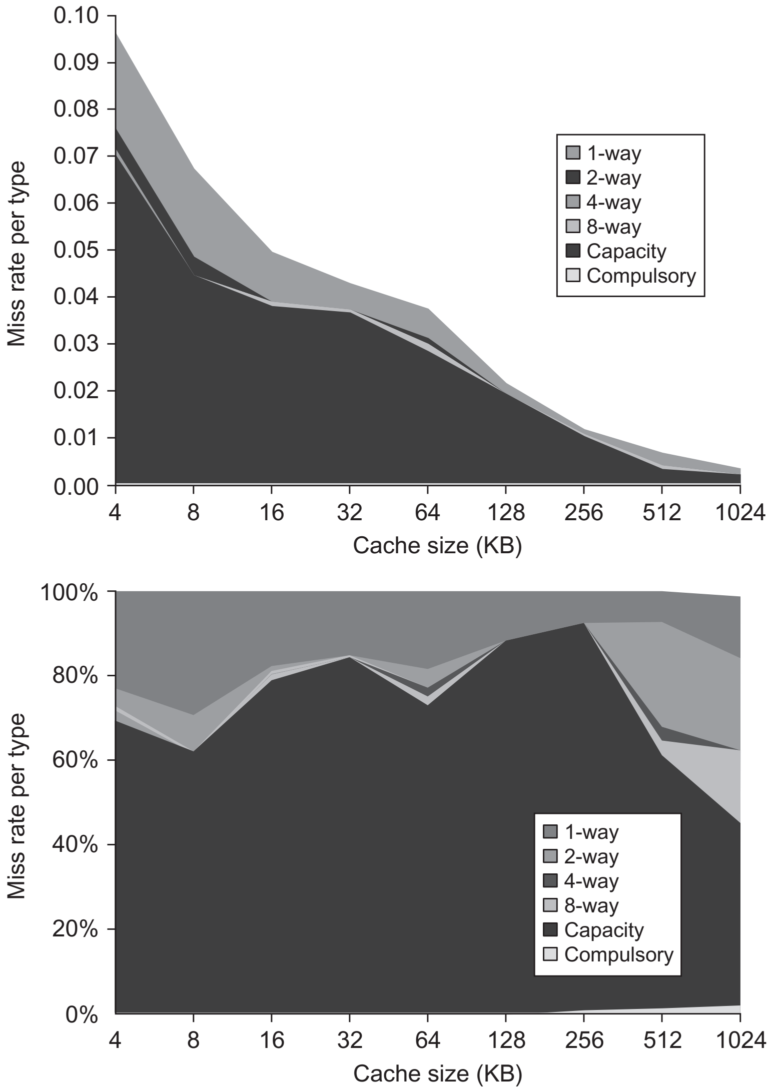


Figure B.9 Total miss rate (top) and distribution of miss rate (bottom) for each size cache according to the three C's for the data in Figure B.8. The top diagram shows the actual data cache miss rates, while the bottom diagram shows the percentage in each category. (*Space allows* the graphs to show one extra cache size than can fit in Figure B.8.)

conflict misses. Full associativity is expensive in hardware, however, and may slow the processor clock rate (see the example on page B-29), leading to lower overall performance.

There is little to be done about capacity except to enlarge the cache. If the upper-level memory is much smaller than what is needed for a program, and a significant percentage of the time is spent moving data between two levels in the hierarchy, the memory hierarchy is said to *thrash*. Because so many replacements are required, thrashing means the computer runs close to the speed of the lower-level memory, or maybe even slower because of the miss overhead.

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

The three C's give insight into the cause of misses, but this simple model has its limits; it gives you insight into average behavior but may not explain an individual miss. For example, changing cache size changes conflict misses as well as capacity misses, because a larger cache spreads out references to more blocks. Thus, a miss might move from a capacity miss to a conflict miss as cache size changes. Similarly, changing the block size can sometimes reduce capacity misses (in addition to the expected reduction in compusolory misses), as Gupta et al. (2013) show.

Note also that the three C's also ignore replacement policy, because it is difficult to model and because, in general, it is less significant. In specific circumstances the replacement policy can actually lead to anomalous behavior, such as poorer miss rates for larger associativity, which contradicts the three C's model. (Some have proposed using an address trace to determine optimal placement in memory to avoid placement misses from the three C's model; we've not followed that advice here.)

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

First Optimization: Larger Block Size to Reduce Miss Rate

The simplest way to reduce miss rate is to increase the block size. Figure B.10 shows the trade-off of block size versus miss rate for a set of programs and cache sizes. Larger block sizes will reduce also compulsory misses. This reduction occurs because the principle of locality has two components: temporal locality and spatial locality. Larger blocks take advantage of spatial locality.

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

Example Figure B.11 shows the actual miss rates plotted in Figure B.10. Assume the memory system takes 80 clock cycles of overhead and then delivers 16 bytes every 2 clock cycles. Thus, it can supply 16 bytes in 82 clock cycles, 32 bytes in 84 clock cycles, and so on. Which block size has the smallest average memory access time for each cache size in Figure B.11?

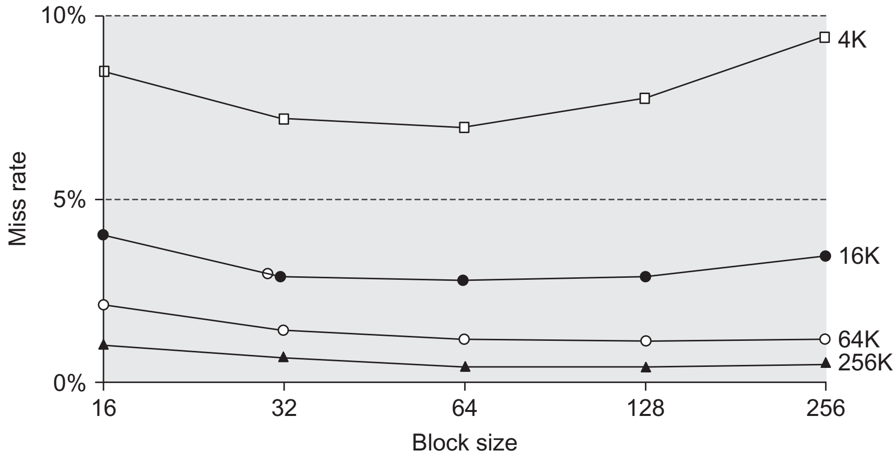


Figure B.10 Miss rate versus block size for five different-sized caches. Note that miss rate actually goes up if the block size is too large relative to the cache size. Each line represents a cache of different size. Figure B.11 shows the data used to plot these lines. Unfortunately, SPEC2000 traces would take too long if block size were included, so these data are based on SPEC92 on a DECstation 5000 (Gee et al. 1993).

Answer Average memory access time is

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

If we assume the hit time is 1 clock cycle independent of block size, then the access time for a 16-byte block in a 4 KiB cache is

$$\text{Average memory access time} = 1 + (8.57\% \times 82) = 8.027 \text{ clock cycles}$$

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

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

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

Figure B.11 Actual miss rate versus block size for the five different-sized caches in Figure B.10. Note that for a 4 KiB cache, 256-byte blocks have a higher miss rate than 32-byte blocks. In this example, the cache would have to be 256 KiB in order for a 256-byte block to decrease misses.

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

Figure B.12 Average memory access time versus block size for five different-sized caches in Figure B.10. Block sizes of 32 and 64 bytes dominate. The smallest average time per cache size is boldfaced.

Figure B.12 shows the average memory access time for all block and cache sizes between those two extremes. The boldfaced entries show the fastest block size for a given cache size: 32 bytes for 4 KiB and 64 bytes for the larger caches. These sizes are, in fact, popular block sizes for processor caches today.

As in all of these techniques, the cache designer is trying to minimize both the miss rate and the miss penalty. The selection of block size depends on both the latency and bandwidth of the lower-level memory. High latency and high bandwidth encourage large block size because the cache gets many more bytes per miss for a small increase in miss penalty. Conversely, low latency and low bandwidth encourage smaller block sizes because there is little time saved from a larger block. For example, twice the miss penalty of a small block may be close to the penalty of a block twice the size. The larger number of small blocks may also reduce conflict misses. Note that Figures B.10 and B.12 show the difference between selecting a block size based on minimizing miss rate versus minimizing average memory access time.

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

Second Optimization: Larger Caches to Reduce Miss Rate

The obvious way to reduce capacity misses in Figures B.8 and B.9 is to increase capacity of the cache. The obvious drawback is potentially longer hit time and higher cost and power. This technique has been especially popular in off-chip caches.

Third Optimization: Higher Associativity to Reduce Miss Rate

Figures B.8 and B.9 show how miss rates improve with higher associativity. There are two general rules of thumb that can be gleaned from these figures. The first is that eight-way set associative is for practical purposes as effective in reducing misses for these sized caches as fully associative. You can see the difference by comparing the eight-way entries to the capacity miss column in Figure B.8, because capacity misses are calculated using fully associative caches.

The second observation, called the *2:1 cache rule of thumb*, is that a direct-mapped cache of size *N* has about the same miss rate as a two-way set associative cache of size *N*/2. This held in three C's figures for cache sizes less than 128 KiB.

Like many of these examples, improving one aspect of the average memory access time comes at the expense of another. Increasing block size reduces miss rate while increasing miss penalty, and greater associativity can come at the cost of increased hit time. Hence, the pressure of a fast processor clock cycle encourages simple cache designs, but the increasing miss penalty rewards associativity, as the following example suggests.

Example Assume that higher associativity would increase the clock cycle time as listed as follows:

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

Assume that the hit time is 1 clock cycle, that the miss penalty for the direct-mapped case is 25 clock cycles to a level 2 cache (see next subsection) that never misses, and that the miss penalty need not be rounded to an integral number of clock cycles. Using Figure B.8 for miss rates, for which cache sizes are each of these three statements true?

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

Answer Average memory access time for each associativity is

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

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

$$\text{Average memory access time}_{\text{1-way}} = 1.00 + (0.098 \times 25) = 3.44$$

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

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

Using these formulas and the miss rates from Figure B.8, Figure B.13 shows the average memory access time for each cache and associativity. The figure shows

	Associativity			
Cache size (KiB)	**1-way**	**2-way**	**4-way**	**8-way**
4	3.44	3.25	3.22	**3.28**
8	2.69	2.58	2.55	**2.62**
16	2.23	**2.40**	**2.46**	**2.53**
32	2.06	**2.30**	**2.37**	**2.45**
64	1.92	**2.14**	**2.18**	**2.25**
128	1.52	**1.84**	**1.92**	**2.00**
256	1.32	**1.66**	**1.74**	**1.82**
512	1.20	**1.55**	**1.59**	**1.66**

Figure B.13 Average memory access time using miss rates in Figure B.8 for parameters in the example. *Boldface* type means that this time is higher than the number to the left, that is, higher associativity *increases* average memory access time.

that the formulas in this example hold for caches less than or equal to 8 KiB for up to four-way associativity. Starting with 16 KiB, the greater hit time of larger associativity outweighs the time saved due to the reduction in misses.

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

Fourth Optimization: Multilevel Caches to Reduce Miss Penalty

Reducing cache misses had been the traditional focus of cache research, but the cache performance formula assures us that improvements in miss penalty can be just as beneficial as improvements in miss rate. Moreover, Figure 2.2 on page 80 shows that technology trends have improved the speed of processors faster than DRAMs, making the relative cost of miss penalties increase over time.

This performance gap between processors and memory leads the architect to this question: Should I make the cache faster to keep pace with the speed of processors, or make the cache larger to overcome the widening gap between the processor and main memory?

One answer is, do both. Adding another level of cache between the original cache and memory simplifies the decision. The first-level cache can be small enough to match the clock cycle time of the fast processor. Yet, the second-level cache can be large enough to capture many accesses that would go to main memory, thereby lessening the effective miss penalty.

Although the concept of adding another level in the hierarchy is straightforward, it complicates performance analysis. Definitions for a second level of cache are not always straightforward. Let's start with the definition of *average memory access time* for a two-level cache. Using the subscripts L1 and L2 to refer, respectively, to a first-level and a second-level cache, the original formula is

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

and

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

so

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

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

- *Local miss rate*—This rate is simply the number of misses in a cache divided by the total number of memory accesses to this cache. As you would expect, for the first-level cache it is equal to Miss rate$_{L1}$, and for the second-level cache it is Miss rate$_{L2}$.
- *Global miss rate*—The number of misses in the cache divided by the total number of memory accesses generated by the processor. Using the terms above, the global miss rate for the first-level cache is still just Miss rate$_{L1}$, but for the second-level cache it is Miss rate$_{L1}$ × Miss rate$_{L2}$.

This local miss rate is large for second-level caches because the first-level cache skims the cream of the memory accesses. This is why the global miss rate is the more useful measure: It indicates what fraction of the memory accesses that leave the processor go all the way to memory.

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

$$\begin{aligned}\text{Average memory stalls per instruction} = {} & \text{Misses per instruction}_{L1} \times \text{Hit time}_{L2} \\ & + \text{Misses per instruction}_{L2} \times \text{Miss penalty}_{L2}\end{aligned}$$

Example Suppose that in 1000 memory references there are 40 misses in the first-level cache and 20 misses in the second-level cache. What are the various miss rates? Assume the miss penalty from the L2 cache to memory is 200 clock cycles, the hit time of the L2 cache is 10 clock cycles, the hit time of L1 is 1 clock cycle, and there are 1.5 memory references per instruction. What is the average memory access time and average stall cycles per instruction? Ignore the impact of writes.

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

$$\begin{aligned}\text{Average memory access time} &= \text{Hit time}_{L1} + \text{Miss rate}_{L1} \times (\text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2}) \\ &= 1 + 4\% \times (10 + 50\% \times 200) = 1 + 4\% \times 110 = 5.4 \text{ clock cycles}\end{aligned}$$

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

$$\begin{aligned}\text{Average memory stalls per instruction} &= \text{Misses per instruction}_{\text{L1}} \times \text{Hit time}_{\text{L2}} + \text{Misses per instruction}_{\text{L2}} \\ &\quad \times \text{Miss penalty}_{\text{L2}} \\ &= (60/1000) \times 10 + (30/1000) \times 200 \\ &= 0.060 \times 10 + 0.030 \times 200 = 6.6 \text{ clock cycles}\end{aligned}$$

If we subtract the L1 hit time from the average memory access time (AMAT) and then multiply by the average number of memory references per instruction, we get the same average memory stalls per instruction:

$$(5.4 - 1.0) \times 1.5 = 4.4 \times 1.5 = 6.6 \text{ clock cycles}$$

As this example shows, there may be less confusion with multilevel caches when calculating using misses per instruction versus miss rates.

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

Figures B.14 and B.15 show how miss rates and relative execution time change with the size of a second-level cache for one design. From these figures we can gain two insights. The first is that the global cache miss rate is very similar to the single cache miss rate of the second-level cache, provided that the second-level cache is much larger than the first-level cache. Hence, our intuition and knowledge about the first-level caches apply. The second insight is that the local cache miss rate is *not* a good measure of secondary caches; it is a function of the miss rate of the first-level cache, and hence can vary by changing the first-level cache. Thus, the global cache miss rate should be used when evaluating second-level caches.

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

The initial decision is the size of a second-level cache. Since everything in the first-level cache is likely to be in the second-level cache, the second-level cache should be much bigger than the first. If second-level caches are just a little bigger, the local miss rate will be high. This observation inspires the design of huge second-level caches—the size of main memory in older computers!

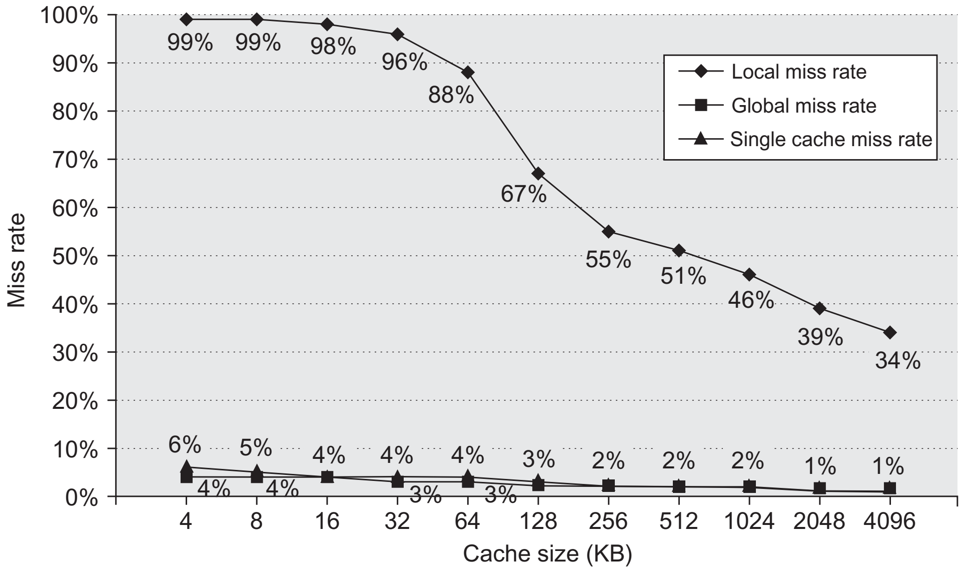


Figure B.14 Miss rates versus cache size for multilevel caches. Second-level caches *smaller* than the sum of the two 64 KiB first-level caches make little sense, as reflected in the high miss rates. After 256 KiB the single cache is within 10% of the global miss rates. The miss rate of a single-level cache versus size is plotted against the local miss rate and global miss rate of a second-level cache using a 32 KiB first-level cache. The L2 caches (unified) were two-way set associative with replacement. Each had split L1 instruction and data caches that were 64 KiB two-way set associative with LRU replacement. The block size for both L1 and L2 caches was 64 bytes. Data were collected as in Figure B.4.

One question is whether set associativity makes more sense for second-level caches.

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

- Hit time$_{L2}$ for direct mapped = 10 clock cycles.
- Two-way set associativity increases hit time by 0.1 clock cycle to 10.1 clock cycles.
- Local miss rate$_{L2}$ for direct mapped = 25%.
- Local miss rate$_{L2}$ for two-way set associative = 20%.
- Miss penalty$_{L2}$ = 200 clock cycles.

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

$$\text{Miss penalty}_{\text{1-way L2}} = 10 + 25\% \times 200 = 60.0\,\text{clock cycles}$$

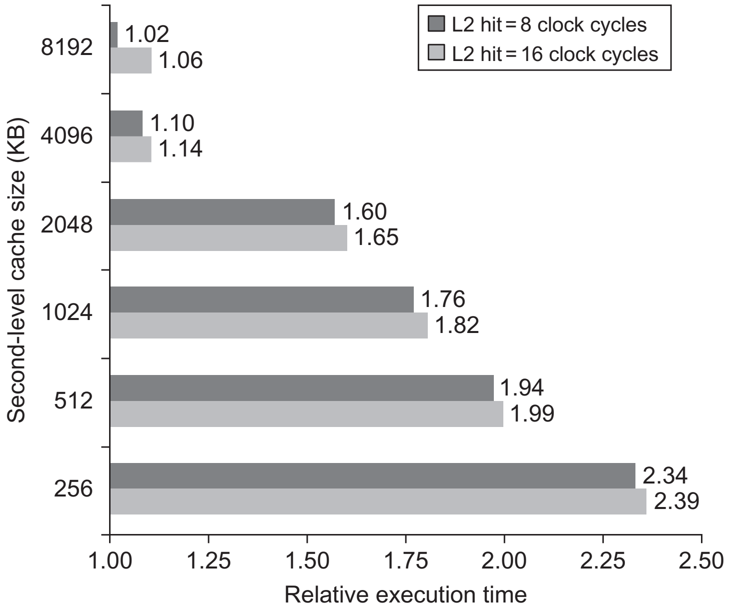


Figure B.15 Relative execution time by second-level cache size. The two bars are for different clock cycles for an L2 cache hit. The reference execution time of 1.00 is for an 8192 KiB second-level cache with a 1-clock-cycle latency on a second-level hit. These data were collected the same way as in Figure B.14, using a simulator to imitate the Alpha 21264.

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

$$\text{Miss penalty}_{\text{2-way L2}} = 10.1 + 20\% \times 200 = 50.1 \text{ clock cycles}$$

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

$$\text{Miss penalty}_{\text{2-way L2}} = 10 + 20\% \times 200 = 50.0 \text{ clock cycles}$$
$$\text{Miss penalty}_{\text{2-way L2}} = 11 + 20\% \times 200 = 51.0 \text{ clock cycles}$$

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

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

One drawback to inclusion is that measurements can suggest smaller blocks for the smaller first-level cache and larger blocks for the larger second-level cache. For example, the Pentium 4 has 64-byte blocks in its L1 caches and 128-byte blocks in its L2 cache. Inclusion can still be maintained with more work on a second-level miss. The second-level cache must invalidate all first-level blocks that map onto the second-level block to be replaced, causing a slightly higher first-level miss rate. To avoid such problems, many cache designers keep the block size the same in all levels of caches.

However, what if the designer can only afford an L2 cache that is slightly bigger than the L1 cache? Should a significant portion of its space be used as a redundant copy of the L1 cache? In such cases a sensible opposite policy is *multilevel exclusion:* L1 data are *never* found in an L2 cache. Typically, with exclusion a cache miss in L1 results in a swap of blocks between L1 and L2 instead of a replacement of an L1 block with an L2 block. This policy prevents wasting space in the L2 cache. For example, the AMD Opteron chip obeys the exclusion property using two 64 KiB L1 caches and 1 MiB L2 cache.

As these issues illustrate, although a novice might design the first- and second-level caches independently, the designer of the first-level cache has a simpler job given a compatible second-level cache. It is less of a gamble to use a write through, for example, if there is a write-back cache at the next level to act as a backstop for repeated writes and it uses multilevel inclusion.

The essence of all cache designs is balancing fast hits and few misses. For second-level caches, there are far fewer hits than in the first-level cache, so the emphasis shifts to fewer misses. This insight leads to much larger caches and techniques to lower the miss rate, such as higher associativity and larger blocks.

Fifth Optimization: Giving Priority to Read Misses over Writes to Reduce Miss Penalty

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

With a write-through cache the most important improvement is a write buffer of the proper size. Write buffers, however, do complicate memory accesses because they might hold the updated value of a location needed on a read miss.

Example Look at this code sequence:

```
sd x3, 512(x0);M[512] ¬ R3 (cache index 0)
ld x1, 1024(x0);x1 ¬ M[1024](cache index 0)
ld x2, 512(x0);x2 ¬ M[512] (cache index 0)
```

Assume a direct-mapped, write-through cache that maps 512 and 1024 to the same block, and a four-word write buffer that is not checked on a read miss. Will the value in x2 always be equal to the value in x3?

Answer Using the terminology from Chapter 2, this is a read-after-write data hazard in memory. Let's follow a cache access to see the danger. The data in x3 are placed into the write buffer after the store. The following load uses the same cache index and is therefore a miss. The second load instruction tries to put the value in location 512 into register x2; this also results in a miss. If the write buffer hasn't completed writing to location 512 in memory, the read of location 512 will put the old, wrong value into the cache block, and then into x2. Without proper precautions, x3x1 would not be equal to x2!

The simplest way out of this dilemma is for the read miss to wait until the write buffer is empty. The alternative is to check the contents of the write buffer on a read miss, and if there are no conflicts and the memory system is available, let the read miss continue. Virtually all desktop and server processors use the latter approach, giving reads priority over writes.

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

Now that we have five optimizations that reduce cache miss penalties or miss rates, it is time to look at reducing the final component of average memory access time. Hit time is critical because it can affect the clock rate of the processor; in many processors today the cache access time limits the clock cycle rate, even for processors that take multiple clock cycles to access the cache. Hence, a fast hit time is multiplied in importance beyond the average memory access time formula because it helps everything.

Sixth Optimization: Avoiding Address Translation During Indexing of the Cache to Reduce Hit Time

Even a small and simple cache must cope with the translation of a virtual address from the processor to a physical address to access memory. As described in Section B.4, processors treat main memory as just another level of the memory hierarchy, and thus the address of the virtual memory that exists on disk must be mapped onto the main memory.

The guideline of making the common case fast suggests that we use virtual addresses for the cache, because hits are much more common than misses. Such caches are termed *virtual caches*, with *physical cache* used to identify the traditional cache that uses physical addresses. As we will shortly see, it is important to distinguish two tasks: indexing the cache and comparing addresses. Thus, the issues are whether a virtual or physical address is used to index the cache and whether a virtual or physical address is used in the tag comparison. Full virtual addressing for both indices and tags eliminates address translation time from a cache hit. Then why doesn't everyone build virtually addressed caches?

One reason is protection. Page-level protection is checked as part of the virtual to physical address translation, and it must be enforced no matter what. One solution is to copy the protection information from the TLB on a miss, add a field to hold it, and check it on every access to the virtually addressed cache.

Another reason is that every time a process is switched, the virtual addresses refer to different physical addresses, requiring the cache to be flushed. Figure B.16 shows the impact on miss rates of this flushing. One solution is to increase the width of the cache address tag with a *process-identifier tag* (PID). If the operating system assigns these tags to processes, it only need flush the cache when a PID is

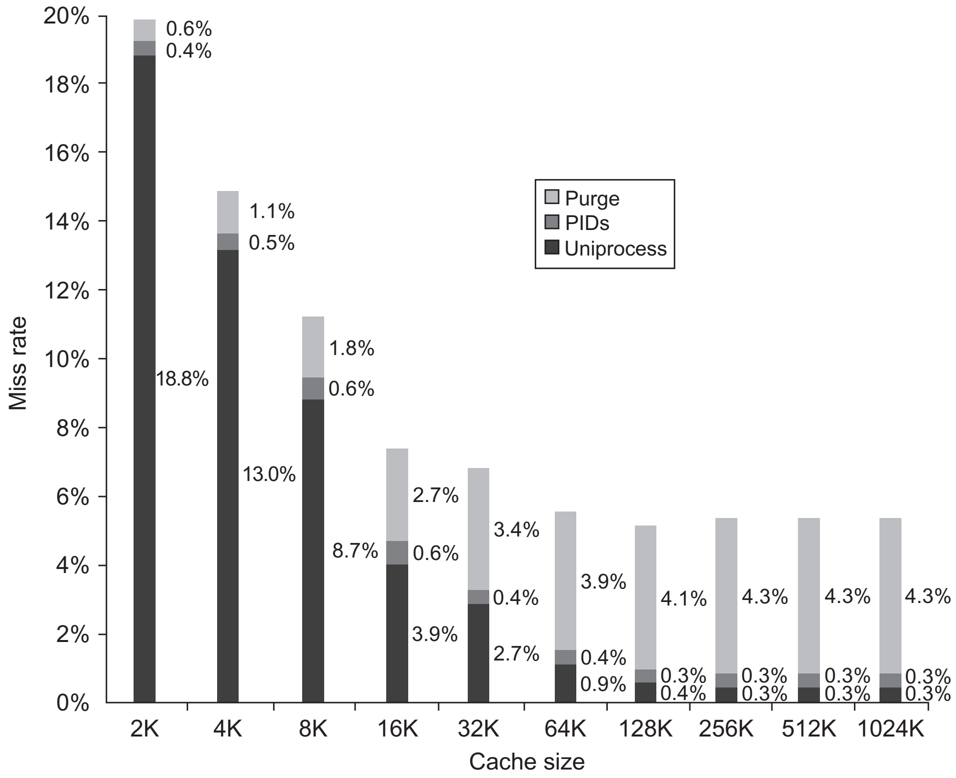


Figure B.16 Miss rate versus virtually addressed cache size of a program measured three ways: without process switches (uniprocess), with process switches using a process-identifier tag (PID), and with process switches but without PIDs (purge). PIDs increase the uniprocess absolute miss rate by 0.3%–0.6%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

recycled; that is, the PID distinguishes whether or not the data in the cache are for this program. Figure B.16 shows the improvement in miss rates by using PIDs to avoid cache flushes.

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

Hardware solutions to the synonym problem, called *antialiasing*, guarantee every cache block a unique physical address. For example, the AMD Opteron uses a 64 KiB instruction cache with a 4 KiB page and two-way set associativity; hence, the hardware must handle aliases involved with the three virtual address bits in the set index. It avoids aliases by simply checking all eight possible locations on a miss—two blocks in each of four sets—to be sure that none matches the physical address of the data being fetched. If one is found, it is invalidated, so when the new data are loaded into the cache their physical address is guaranteed to be unique.

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

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

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

This alternative allows the cache read to begin immediately, and yet the tag comparison is still with physical addresses. The limitation of this *virtually indexed, physically tagged* alternative is that a direct-mapped cache can be no bigger than the page size. For example, in the data cache in Figure B.5 on page B-13, the index is 9 bits and the cache block offset is 6 bits. To use this trick, the virtual page size would have to be at least $2^{(9+6)}$ bytes or 32 KiB. If not, a portion of the index must be translated from virtual to physical address. Figure B.17 shows the organization of the caches, translation lookaside buffers (TLBs), and virtual memory when this technique is used.

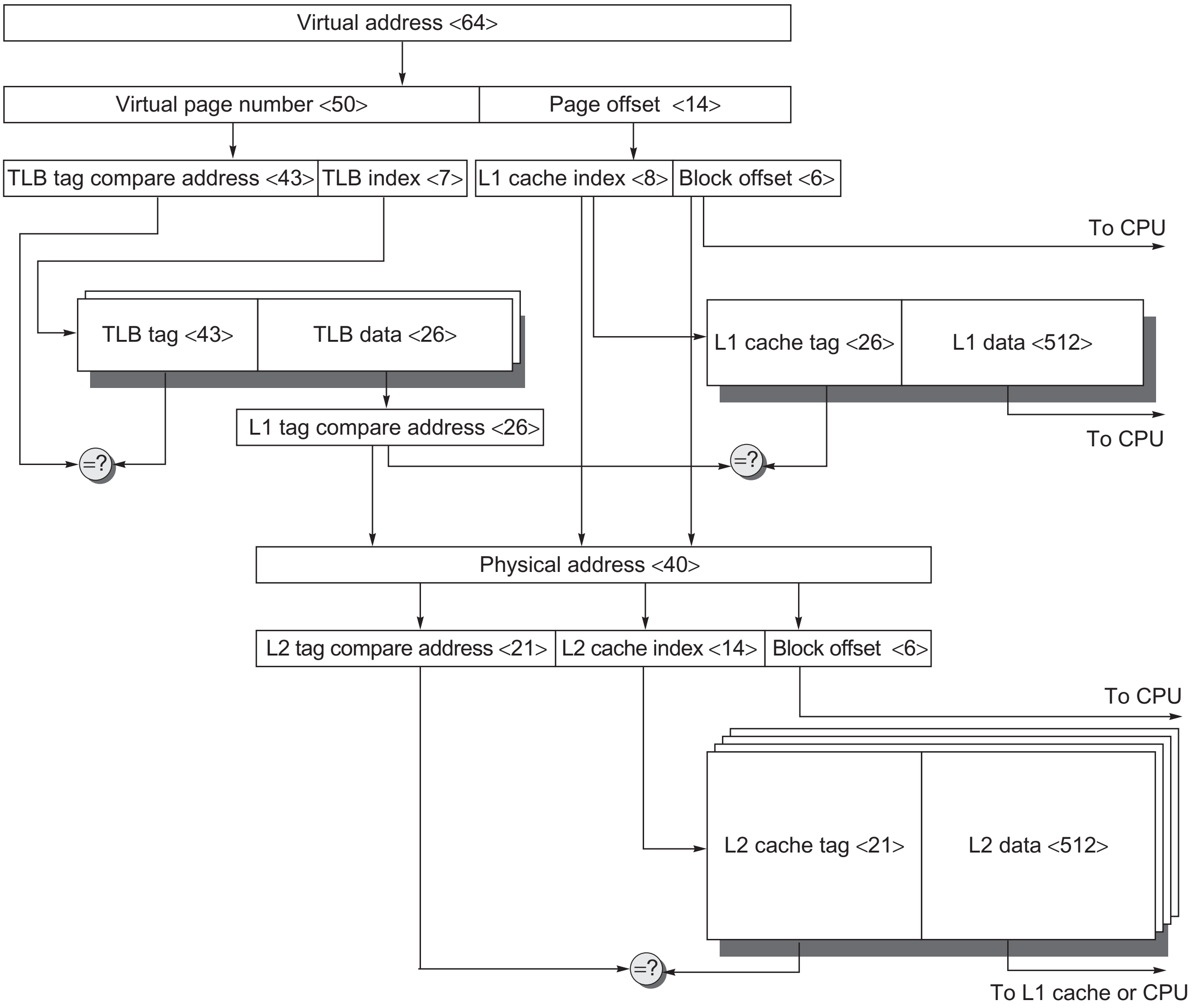


Figure B.17 The overall picture of a hypothetical memory hierarchy going from virtual address to L2 cache access. The page size is 16 KiB. The TLB is two-way set associative with 256 entries. The L1 cache is a direct-mapped 16 KiB, and the L2 cache is a four-way set associative with a total of 4 MiB. Both use 64-byte blocks. The virtual address is 64 bits and the physical address is 40 bits.

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

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

For example, doubling associativity and doubling the cache size does not change the size of the index. The IBM 3033 cache, as an extreme example, is 16-way set associative, even though studies show there is little benefit to miss rates above

Technique	Hit time	Miss penalty	Miss rate	Hardware complexity	Comment
Larger block size		–	+	0	Trivial; Pentium 4L2 uses 128 bytes
Larger cache size	–		+	1	Widely used, especially for L2 caches
Higher associativity	–		+	1	Widely used
Multilevel caches		+		2	Costly hardware; harder if L1 block size ≠ L2 block size; widely used
Read priority over writes		+		1	Widely used
Avoiding address translation during cache indexing	+			1	Widely used

Figure B.18 Summary of basic cache optimizations showing impact on cache performance and complexity for the techniques in this appendix. Generally a technique helps only one factor. + 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.

8-way set associativity. This high associativity allows a 64 KiB cache to be addressed with a physical index, despite the handicap of 4 KiB pages in the IBM architecture.

Summary of Basic Cache Optimization

The techniques in this section to improve miss rate, miss penalty, and hit time generally impact the other components of the average memory access equation as well as the complexity of the memory hierarchy. Figure B.18 summarizes these techniques and estimates the impact on complexity, with + meaning that the technique improves the factor, – meaning it hurts that factor, and blank meaning it has no impact. No optimization in this figure helps more than one category.

B.4 Virtual Memory

> … a system has been devised to make the core drum combination appear to the programmer as a single level store, the requisite transfers taking place automatically.
>
> Kilburn et al. (1962)

At any instant in time computers are running multiple processes, each with its own address space. (Processes are described in the next section.) It would be too expensive to dedicate a full address space worth of memory for each process, especially because many processes use only a small part of their address space. Hence, there must be a means of sharing a smaller amount of physical memory among many processes.

One way to do this, *virtual memory*, divides physical memory into blocks and allocates them to different processes. Inherent in such an approach must be a *protection* scheme that restricts a process to the blocks belonging only to that process. Most forms of virtual memory also reduce the time to start a program, because not all code and data need be in physical memory before a program can begin.

Although protection provided by virtual memory is essential for current computers, sharing is not the reason that virtual memory was invented. If a program became too large for physical memory, it was the programmer's job to make it fit. Programmers divided programs into pieces, then identified the pieces that were mutually exclusive, and loaded or unloaded these *overlays* under user program control during execution. The programmer ensured that the program never tried to access more physical main memory than was in the computer, and that the proper overlay was loaded at the proper time. As you can well imagine, this responsibility eroded programmer productivity.

Virtual memory was invented to relieve programmers of this burden; it automatically manages the two levels of the memory hierarchy represented by main memory and secondary storage. Figure B.19 shows the mapping of virtual memory to physical memory for a program with four pages.

In addition to sharing protected memory space and automatically managing the memory hierarchy, virtual memory also simplifies loading the program for execution. Called *relocation*, this mechanism allows the same program to run in any location in physical memory. The program in Figure B.19 can be placed anywhere

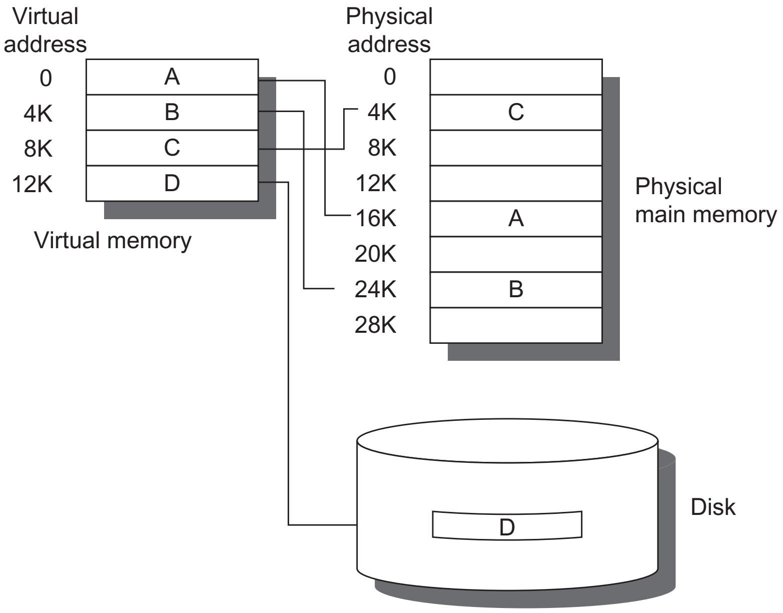


Figure B.19 The logical program in its contiguous virtual address space is shown on the left. It consists of four pages, A, B, C, and D. The actual location of three of the blocks is in physical main memory and the other is located on the disk.

in physical memory or disk just by changing the mapping between them. (Prior to the popularity of virtual memory, processors would include a relocation register just for that purpose.) An alternative to a hardware solution would be software that changed all addresses in a program each time it was run.

Several general memory hierarchy ideas from Chapter 1 about caches are analogous to virtual memory, although many of the terms are different. *Page* or *segment* is used for block, and *page fault* or *address fault* is used for miss. With virtual memory, the processor produces *virtual addresses* that are translated by a combination of hardware and software to *physical addresses*, which access main memory. This process is called *memory mapping* or *address translation*. Today, the two memory hierarchy levels controlled by virtual memory are DRAMs and magnetic disks. Figure B.20 shows a typical range of memory hierarchy parameters for virtual memory.

There are further differences between caches and virtual memory beyond those quantitative ones mentioned in Figure B.20:

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

Parameter	**First-level cache**	**Virtual memory**
Block (page) size	16–128 bytes	4096–65,536 bytes
Hit time	1–3 clock cycles	100–200 clock cycles
Miss penalty	8–200 clock cycles	1,000,000–10,000,000 clock cycles
(access time)	(6–160 clock cycles)	(800,000–8,000,000 clock cycles)
(transfer time)	(2–40 clock cycles)	(200,000–2,000,000 clock cycles)
Miss rate	0.1%–10%	0.00001%–0.001%
Address mapping	25–45-bit physical address to 14–20-bit cache address	32–64-bit virtual address to 25–45-bit physical address

Figure B.20 Typical ranges of parameters for caches and virtual memory. Virtual memory parameters represent increases of 10–1,000,000 times over cache parameters. Usually, first-level caches contain at most 1 MiB of data, whereas physical memory contains 256 MiB to 1 TB.

Virtual memory also encompasses several related techniques. Virtual memory systems can be categorized into two classes: those with fixed-size blocks, called *pages*, and those with variable-size blocks, called *segments*. Pages are typically fixed at 4096–8192 bytes, while segment size varies. The largest segment supported on any processor ranges from 2^{16} bytes up to 2^{32} bytes; the smallest segment is 1 byte. Figure B.21 shows how the two approaches might divide code and data.

The decision to use paged virtual memory versus segmented virtual memory affects the processor. Paged addressing has a single fixed-size address divided into page number and offset within a page, analogous to cache addressing. A single address does not work for segmented addresses; the variable size of segments requires 1 word for a segment number and 1 word for an offset within a segment, for a total of 2 words. An unsegmented address space is simpler for the compiler.

The pros and cons of these two approaches have been well documented in operating systems textbooks; Figure B.22 summarizes the arguments. Because of the

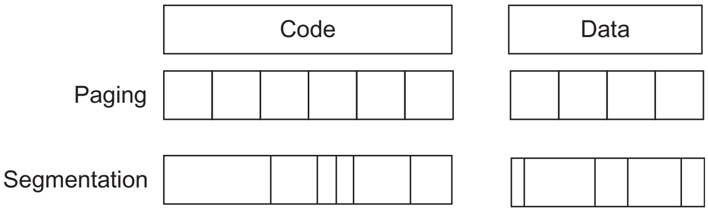


Figure B.21 Example of how paging and segmentation divide a program.

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

Figure B.22 Paging versus segmentation. Both can waste memory, depending on the block size and how well the segments fit together in main memory. Programming languages with unrestricted pointers require both the segment and the address to be passed. A hybrid approach, called *paged segments*, shoots for the best of both worlds: segments are composed of pages, so replacing a block is easy, yet a segment may be treated as a logical unit.

replacement problem (the third line of the figure), few computers today use pure segmentation. Some computers use a hybrid approach, called *paged segments*, in which a segment is an integral number of pages. This simplifies replacement because memory need not be contiguous, and the full segments need not be in main memory. A more recent hybrid is for a computer to offer multiple page sizes, with the larger sizes being powers of 2 times the smallest page size. The IBM 405CR embedded processor, for example, allows 1 KiB, 4 KiB ($2^2 \times 1$ KiB), 16 KiB ($2^4 \times 1$ KiB), 64 KiB ($2^6 \times 1$ KiB), 256 KiB ($2^8 \times 1$ KiB), 1024 KiB ($2^{10} \times 1$ KiB), and 4096 KiB ($2^{12} \times 1$ KiB) to act as a single page.

Four Memory Hierarchy Questions Revisited

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

Q1: Where Can a Block be Placed in Main Memory?

The miss penalty for virtual memory involves access to a rotating magnetic storage device and is therefore quite high. Given the choice of lower miss rates or a simpler placement algorithm, operating systems designers usually pick lower miss rates because of the exorbitant miss penalty. Thus, operating systems allow blocks to be placed anywhere in main memory. According to the terminology in Figure B.2 on page B-8, this strategy would be labeled fully associative.

Q2: How Is a Block Found If It Is in Main Memory?

Both paging and segmentation rely on a data structure that is indexed by the page or segment number. This data structure contains the physical address of the block. For segmentation, the offset is added to the segment's physical address to obtain the final physical address. For paging, the offset is simply concatenated to this physical page address (see Figure B.23).

This data structure, containing the physical page addresses, usually takes the form of a *page table.* Indexed by the virtual page number, the size of the table is the number of pages in the virtual address space. Given a 32-bit virtual address, 4 KiB pages, and 4 bytes per page table entry (PTE), the size of the page table would be $(2^{32}/2^{12}) \times 2^2 = 2^{22}$ or 4 MiB.

To reduce the size of this data structure, some computers apply a hashing function to the virtual address. The hash allows the data structure to be the length of the number of *physical* pages in main memory. This number could be much smaller than the number of virtual pages. Such a structure is called an *inverted page table*. Using the previous example, a 512 MiB physical memory would only need 1 MiB (8×512 MiB/4 KiB) for an inverted page table; the extra 4 bytes per page table entry are for the virtual address. The HP/Intel IA-64 covers both bases by offering

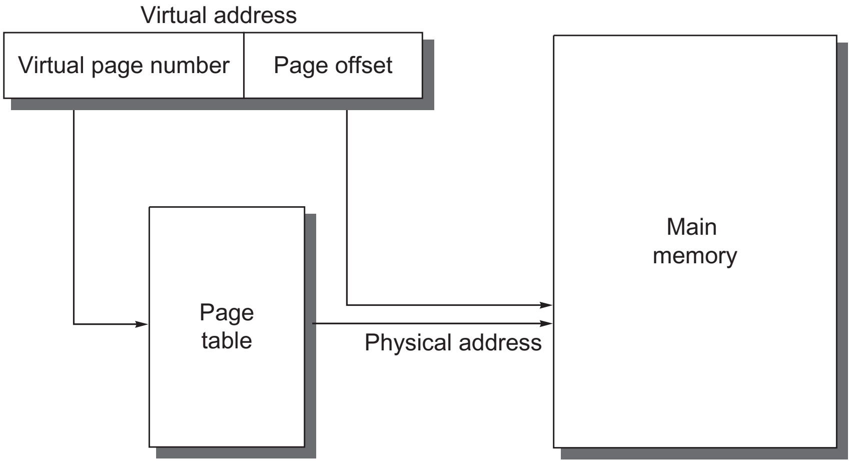


Figure B.23 The mapping of a virtual address to a physical address via a page table.

both traditional pages tables *and* inverted page tables, leaving the choice of mechanism to the operating system programmer.

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

Q3: Which Block Should be Replaced on a Virtual Memory Miss?

As mentioned earlier, the overriding operating system guideline is minimizing page faults. Consistent with this guideline, almost all operating systems try to replace the least recently used (LRU) block because if the past predicts the future, that is the one less likely to be needed.

To help the operating system estimate LRU, many processors provide a *use bit* or *reference bit*, which is logically set whenever a page is accessed. (To reduce work, it is actually set only on a translation buffer miss, which is described shortly.) The operating system periodically clears the use bits and later records them so it can determine which pages were touched during a particular time period. By keeping track in this way, the operating system can select a page that is among the least recently referenced.

Q4: What Happens on a Write?

The level below main memory contains rotating magnetic disks that take millions of clock cycles to access. Because of the great discrepancy in access time, no one has yet built a virtual memory operating system that writes through main memory to disk on every store by the processor. (This remark should not be interpreted as an opportunity to become famous by being the first to build one!) Thus, the write strategy is always write-back.

Because the cost of an unnecessary access to the next-lower level is so high, virtual memory systems usually include a dirty bit. It allows blocks to be written to disk only if they have been altered since being read from the disk.

Techniques for Fast Address Translation

Page tables are usually so large that they are stored in main memory and are sometimes paged themselves. Paging means that every memory access logically takes at least twice as long, with one memory access to obtain the physical address and a second access to get the data. As mentioned in Chapter 2, we use locality to avoid the extra memory access. By keeping address translations in a special cache, a memory access rarely requires a second access to translate the data. This special address translation cache is referred to as a *translation look aside buffer* (TLB), also called a *translation buffer* (TB).

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

Figure B.24 shows the Opteron data TLB organization, with each step of the translation labeled. This TLB uses fully associative placement; thus, the translation begins (steps 1 and 2) by sending the virtual address to all tags. Of course, the tag must be marked valid to allow a match. At the same time, the type of memory access is checked for a violation (also in step 2) against protection information in the TLB.

For reasons similar to those in the cache case, there is no need to include the 12 bits of the page offset in the TLB. The matching tag sends the corresponding physical address through effectively a 40:1 multiplexor (step 3). The page offset is then combined with the physical page frame to form a full physical address (step 4). The address size is 40 bits.

Address translation can easily be on the critical path determining the clock cycle of the processor, so the Opteron uses virtually addressed, physically tagged L1 caches.

Selecting a Page Size

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

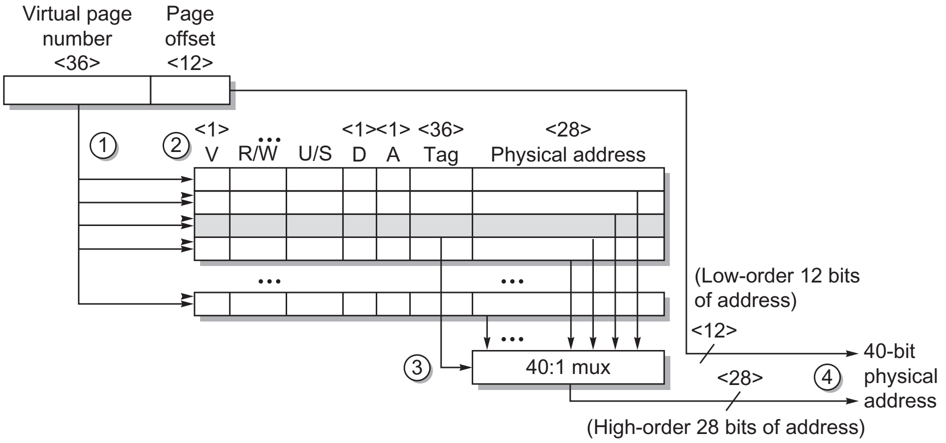


Figure B.24 Operation of the Opteron data TLB during address translation. The four steps of a TLB hit are shown as *circled numbers*. This TLB has 40 entries. Section B.5 describes the various protection and access fields of an Opteron page table entry.

- The size of the page table is inversely proportional to the page size; memory (or other resources used for the memory map) can therefore be saved by making the pages bigger.
- As mentioned in Section B.3, a larger page size can allow larger caches with fast cache hit times.
- Transferring larger pages to or from secondary storage, possibly over a network, is more efficient than transferring smaller pages.
- The number of TLB entries is restricted, so a larger page size means that more memory can be mapped efficiently, thereby reducing the number of TLB misses.

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

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

Summary of Virtual Memory and Caches

With virtual memory, TLBs, first-level caches, and second-level caches all mapping portions of the virtual and physical address space, it can get confusing what bits go where. Figure B.25 gives a hypothetical example going from a 64-bit virtual address to a 41-bit physical address with two levels of cache. This L1 cache is virtually indexed, and physically tagged because both the cache size and the page size are 8 KiB. The L2 cache is 4 MiB. The block size for both is 64 bytes.

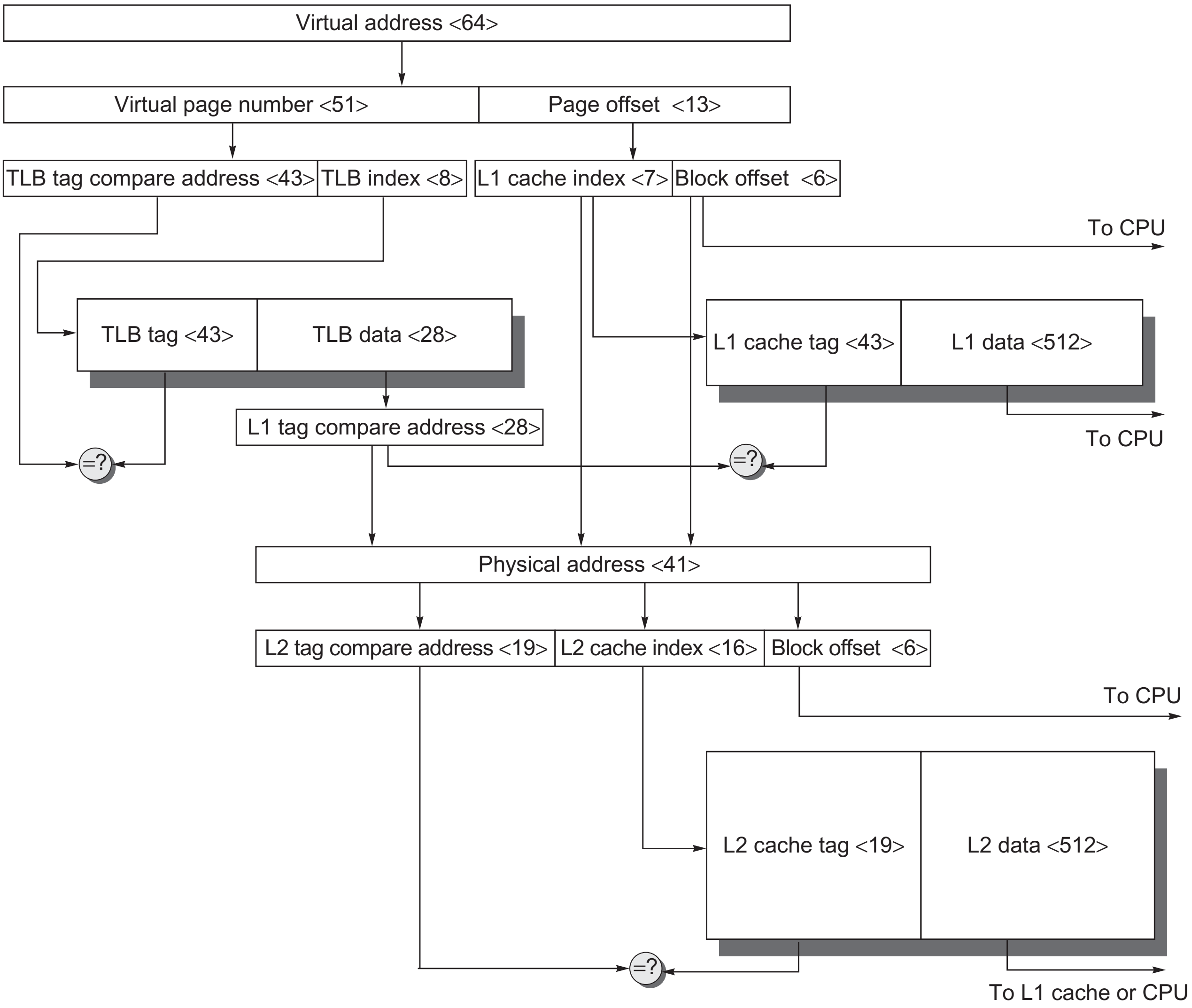


Figure B.25 The overall picture of a hypothetical memory hierarchy going from virtual address to L2 cache access. The page size is 8 KiB. The TLB is direct mapped with 256 entries. The L1 cache is a direct-mapped 8 KiB, and the L2 cache is a direct-mapped 4 MiB. Both use 64-byte blocks. The virtual address is 64 bits and the physical address is 41 bits. The primary difference between this simple figure and a real cache is replication of pieces of this figure.

COMPUTER ARCHITECTURE

Sixth Edition

A Quantitative Approach

John L. Hennessy | David A. Patterson

Foreword by Norman P. Jouppi

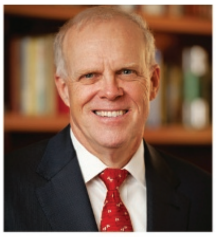
"This sixth edition comes at a critical time: Moore's Law is fading just as deep learning demands unprecedented compute cycles. The new chapter on domain-specific architectures documents a number of promising approaches and prophesies a rebirth in computer architecture. Like the scholars of the European Renaissance, computer architects must understand our own history, and then combine the lessons of that history with new techniques to remake the world."
—Cliff Young, Google

Computer Architecture: A Quantitative Approach, Sixth Edition has been considered essential reading by instructors, students, and practitioners of computer design for nearly 30 years. The sixth edition of this classic textbook is fully revised with the latest developments in processor and system architecture. It now features examples from the RISC-V ("RISC Five") instruction set architecture, a modern RISC instruction set developed and designed to be a free and openly adoptable standard. It also includes a new chapter on domain-specific architectures and an updated chapter on warehouse-scale computing that features the first public information on Google's newest WSC. True to its original mission of demystifying computer architecture, this edition continues the longstanding tradition of focusing on areas where the most exciting computing innovation is happening, while always keeping an emphasis on good engineering design.

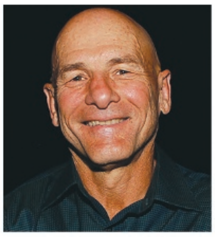
Features

- Includes a new chapter on domain-specific architectures, explaining how they are the only path forward for improved performance and energy efficiency given the end of Moore's Law and Dennard scaling
- Features the introduction of four DSAs from industry: Google Tensor Processing Unit, Google Pixel Visual Core, Intel Nervana Neural Network Processor, and Microsoft Catapult
- Features extensive updates to the chapter on warehouse-scale computing, with the first public information on the newest Google WSC
- Offers updates to other chapters including new material dealing with the use of stacked DRAM; data on the performance of new NVIDIA Pascal GPU vs. new AVX-512 Intel Skylake CPU; and extensive additions to content covering multicore architecture and organization

About the Authors



John L. Hennessy
Professor of Electrical Engineering
and Computer Science
President Emeritus
Stanford University



David A. Patterson
Distinguished Engineer, Google
Pardee Chair of Computer Science,
Emeritus
University of California at Berkeley

Computer Systems and Design
Computer Engineering

ISBN 978-0-12-811905-1

MORGAN KAUFMANN PUBLISHERS
AN IMPRINT OF ELSEVIER
elsevier.com/books-and-journals