Memory Hierarchy Design - Part 1. Basics of Memory Hierarchies

John L. Hennessy, Stanford University, and David A. Patterson, University of California, Berkeley - September 25, 2012

Editor's Note: Demand for increasing functionality and performance in systems designs continues to drive the need for more memory even as hardware engineers balance the dynamics of system capability, power, and cost against the growing performance gap between processor and memory. Architectures based on memory hierarchy address these issues, and what better source for the details of this approach than an excerpt on the subject from the seminal book on Computer Architecture by John Hennessy and David Patterson. Part 1 looks at the key issues surrounding memory hierarchies and sets the stage for subsequent installments addressing cache design, memory optimization, and design approaches. Part 2, Ten advanced optimizations of cache performance, reviews advanced optimizations of cache performance.

Adapted from "Computer Architecture, Fifth Edition: A Quantitative Approach" by John Hennessy and David Patterson (Morgan Kaufmann)

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

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

Introduction
Computer pioneers correctly predicted that programmers would want unlimited amounts of fast memory. An economical solution to that desire is a memory hierarchy, which takes advantage of locality and trade-offs in the cost-performance of memory technologies. The principle of locality, presented in the first chapter, says that most programs do not access all code or data uniformly. Locality occurs in time (temporal locality) and in space (spatial locality). This principle, plus the guideline that for a given implementation technology and power budget smaller hardware can be made faster, led to hierarchies based on memories of different speeds and sizes. Figure 2.1 shows a multilevel memory hierarchy, including typical sizes and speeds of access.
Figure 2.1. The levels in a typical memory hierarchy in a server computer shown on top (a) and in a personal mobile device (PMD) on the bottom (b). As we move farther away from the processor, the memory in the level below becomes slower and larger. Note that the time units change by a factor of $10^9$ - from picoseconds to milliseconds - and that the size units change by a factor of $10^{12}$ - from bytes to terabytes. The PMD has a slower clock rate and smaller caches and main memory. A key difference is that servers and desktops use disk storage as the lowest level in the hierarchy while PMDs use Flash, which is built from EEPROM technology.

Since fast memory is expensive, a memory hierarchy is organized into several levels - each smaller, faster, and more expensive per byte than the next lower level, which is farther from the processor. The goal is to provide a memory system with cost per byte almost as low as the cheapest level of memory and speed almost as fast as the fastest level. In most cases (but not all), the data contained in a lower level are a superset of the next higher level. This property, called the inclusion property, is always required for the lowest level of the hierarchy, which consists of main memory in the case of caches and disk memory in the case of virtual memory.

Title-1

The importance of the memory hierarchy has increased with advances in performance of processors. Figure 2.2 plots single processor performance projections against the historical performance improvement in time to access main memory. The processor line shows the increase in memory requests per second on average (i.e., the inverse of the latency between memory references), while the memory line shows the increase in DRAM accesses per second (i.e., the inverse of the DRAM access latency). The situation in a uniprocessor is actually somewhat worse, since the peak memory access rate is faster than the average rate, which is what is plotted.
Figure 2.2. Starting with 1980 performance as a baseline, the gap in performance, measured as the difference in the time between processor memory requests (for a single processor or core) and the latency of a DRAM access, is plotted over time. Note that the vertical axis must be on a logarithmic scale to record the size of the processor–DRAM performance gap. The memory baseline is 64 KB DRAM in 1980, with a 1.07 per year performance improvement in latency (see Figure 2.13 on page 99). The processor line assumes a 1.25 improvement per year until 1986, a 1.52 improvement until 2000, a 1.20 improvement between 2000 and 2005, and no change in processor performance (on a per-core basis) between 2005 and 2010; see Figure 1.1 in Chapter 1.

More recently, high-end processors have moved to multiple cores, further increasing the bandwidth requirements versus single cores. In fact, the aggregate peak bandwidth essentially grows as the numbers of cores grows. A modern high-end processor such as the Intel Core i7 can generate two data memory references per core each clock cycle; with four cores and a 3.2 GHz clock rate, the i7 can generate a peak of 25.6 billion 64-bit data memory references per second, in addition to a peak instruction demand of about 12.8 billion 128-bit instruction references; this is a total peak bandwidth of 409.6 GB/sec! This incredible bandwidth is achieved by multiporting and pipelining the caches; by the use of multiple levels of caches, using separate first- and sometimes second-level caches per core; and by using a separate instruction and data cache at the first level. In contrast, the peak bandwidth to DRAM main memory is only 6% of this (25 GB/sec).

Traditionally, designers of memory hierarchies focused on optimizing average memory access time, which is determined by the cache access time, miss rate, and miss penalty. More recently, however, power has become a major consideration. In high-end microprocessors, there may be 10 MB or more of on-chip cache, and a large second- or third-level cache will consume significant power both as leakage when not operating (called static power) and as active power, as when performing a read or write (called dynamic power), as described in Section 2.3. The problem is even more acute in processors in PMDs where the CPU is less aggressive and the power budget may be 20 to 50 times smaller. In such cases, the caches can account for 25% to 50% of the total power consumption. Thus, more designs must consider both performance and power trade-offs, and we will examine both in this chapter.

Title-2

Basics of Memory Hierarchies: A Quick Review
The increasing size and thus importance of this gap led to the migration of the basics of memory hierarchy into undergraduate courses in computer architecture, and even to courses in operating systems and compilers. Thus, we’ll start with a quick review of caches and their operation. The bulk of the chapter, however, describes more advanced innovations that attack the processor–memory performance gap.

When a word is not found in the cache, the word must be fetched from a lower level in the hierarchy (which may be another cache or the main memory) and placed in the cache before continuing. Multiple words, called a block (or line), are moved for efficiency reasons, and because they are likely to be needed soon due to spatial locality. Each cache block includes a tag to indicate which memory address it corresponds to.

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

\[(\text{Block address}) \mod (\text{Number of sets in cache})\]

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

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

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

To gain insights into the causes of high miss rates, which can inspire better cache designs, the three Cs model sorts all misses into three simple categories:

- **Compulsory** - The very first access to a block cannot be in the cache, so the block must be brought into the cache. Compulsory misses are those that occur even if you had an infinite sized cache.
- **Capacity** - If the cache cannot contain all the blocks needed during execution of a program, capacity misses (in addition to compulsory misses) will occur because of blocks being discarded and later retrieved.
- **Conflict** - If the block placement strategy is not fully associative, conflict misses (in addition to compulsory and capacity misses) will occur because a block may be discarded and later retrieved if multiple blocks map to its set and accesses to the different blocks are intermingled.

Title-3

Figures B.8 and B.9 on pages B-24 and B-25 show the relative frequency of cache misses broken
down by the three Cs. As we will see in Chapters 3 and 5, multithreading and multiple cores add complications for caches, both increasing the potential for capacity misses as well as adding a fourth C, for coherency misses due to cache flushes to keep multiple caches coherent in a multiprocessor; we will consider these issues in Chapter 5.

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

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

(It is often reported as misses per 1000 instructions to use integers instead of fractions.)

The problem with both measures is that they don’t factor in the cost of a miss. A better measure is the average memory access time:

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

where hit time is the time to hit in the cache and miss penalty is the time to replace the block from memory (that is, the cost of a miss). Average memory access time is still an indirect measure of performance; although it is a better measure than miss rate, it is not a substitute for execution time. In Chapter 3 we will see that speculative processors may execute other instructions during a miss, thereby reducing the effective miss penalty. The use of multithreading (introduced in Chapter 3) also allows a processor to tolerate misses without being forced to idle. As we will examine shortly, to take advantage of such latency tolerating techniques we need caches that can service requests while handling an outstanding miss.

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

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

Title-4

1. Larger block size to reduce miss rate - The simplest way to reduce the miss rate is to take advantage of spatial locality and increase the block size. Larger blocks reduce compulsory misses, but they also increase the miss penalty. Because larger blocks lower the number of tags, they can slightly reduce static power. Larger block sizes can also increase capacity or conflict misses, especially in smaller caches. Choosing the right block size is a complex trade-off that depends on the size of cache and the miss penalty.
2. Bigger caches to reduce miss rate - The obvious way to reduce capacity misses is to increase cache capacity. Drawbacks include potentially longer hit time of the larger cache memory and higher cost and power. Larger caches increase both static and dynamic power.
3. Higher associativity to reduce miss rate - Obviously, increasing associativity reduces conflict misses. Greater associativity can come at the cost of increased hit time. As we will see shortly, associativity also increases power consumption.
4. Multilevel caches to reduce miss penalty - A difficult decision is whether to make the cache hit
time fast, to keep pace with the high clock rate of processors, or to make the cache large to reduce the gap between the processor accesses and main memory accesses. Adding another level of cache between the original cache and memory simplifies the decision (see Figure 2.3). The first-level cache can be small enough to match a fast clock cycle time, yet the second-level (or third-level) cache can be large enough to capture many accesses that would go to main memory. The focus on misses in second-level caches leads to larger blocks, bigger capacity, and higher associativity. Multi-level caches are more power efficient than a single aggregate cache. If L1 and L2 refer, respectively, to first- and second-level caches, we can redefine the average memory access time:

\[
\text{Hit time}_L + \text{Miss rate}_L \cdot (\text{Hit time}_{L2} + \text{Miss rate}_{L2} \cdot \text{Miss penalty}_{L2})
\]

Figure 2.3. Access times generally increase as cache size and associativity are increased. These data come from the CACTI model 6.5 by Tarjan, Thoziyoor, and Jouppi [2005]. The data assume a 40 nm feature size (which is between the technology used in Intel’s fastest and second fastest versions of the i7 and the same as the technology used in the fastest ARM embedded processors), a single bank, and 64-byte blocks. The assumptions about cache layout and the complex trade-offs between interconnect delays (that depend on the size of a cache block being accessed) and the cost of tag checks and multiplexing lead to results that are occasionally surprising, such as the lower access time for a 64 KB with two-way set associativity versus direct mapping. Similarly, the results with eight-way set associativity generate unusual behavior as cache size is increased. Since such observations are highly dependent on technology and detailed design assumptions, tools such as CACTI serve to reduce the search space rather than precision analysis of the trade-offs.

5. Giving priority to read misses over writes to reduce miss penalty - A write buffer is a good place to implement this optimization. Write buffers create hazards because they hold the updated value of a location needed on a read miss—that is, a read-after-write hazard through memory. One solution is to check the contents of the write buffer on a read miss. If there are no conflicts, and if the memory system is available, sending the read before the writes reduces the miss penalty. Most processors give reads priority over writes. This choice has little effect on power
6. Avoiding address translation during indexing of the cache to reduce hit time - Caches must cope
with the translation of a virtual address from the processor to a physical address to access
memory. (Virtual memory is covered in Sections 2.4 and B.4.) A common optimization is to use
the page offset—the part that is identical in both virtual and physical addresses—to index the
cache, as described in Appendix B, page B-38. This virtual index/physical tag method introduces
some system complications and/or limitations on the size and structure of the L1 cache, but the
advantages of removing the translation lookaside buffer (TLB) access from the critical path
outweigh the disadvantages.

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

Title-5

The rest of this chapter assumes familiarity with the material above and the details in Appendix B. In
the Putting It All Together section, we examine the memory hierarchy for a microprocessor designed
for a high-end server, the Intel Core i7, as well as one designed for use in a PMD, the Arm Cortex-
A8, which is the basis for the processor used in the Apple iPad and several high-end smartphones.

Within each of these classes, there is a significant diversity in approach due to the intended use of
the computer. While the high-end processor used in the server has more cores and bigger caches
than the Intel processors designed for desktop uses, the processors have similar architectures. The
differences are driven by performance and the nature of the workload; desktop computers are
primarily running one application at a time on top of an operating system for a single user, whereas
server computers may have hundreds of users running potentially dozens of applications
simultaneously.

Because of these workload differences, desktop computers are generally concerned more with
average latency from the memory hierarchy, whereas server computers are also concerned about
memory bandwidth. Even within the class of desktop computers there is wide diversity from lower
end netbooks with scaled-down processors more similar to those found in high-end PMDs, to high-
end desktops whose processors contain multiple cores and whose organization resembles that of a
low-end server.

In contrast, PMDs not only serve one user but generally also have smaller operating systems, usually
less multitasking (running of several applications simultaneously), and simpler applications. PMDs
also typically use Flash memory rather than disks, and most consider both performance and energy
consumption, which determines battery life.

Adapted from "Computer Architecture, Fifth Edition: A Quantitative Approach" by John
Hennessy and David Patterson (Morgan Kaufmann)

If you liked this and would like to see a weekly collection of related products and features delivered
directly to your inbox, click here to sign up for the EDN on Systems Design newsletter.

Related articles

- Fundamentals of solid-state memory technologies in consumer electronics - Part 1: Flash memory
- The future of computers - Part 1: Multicore and the Memory Wall
- Addressing critical-area analysis and memory redundancy
- Memory Architecture needs redefinition