Memory - Caches*

Charles Severance
Kevin Dowd

This work is produced by OpenStax-CNX and licensed under the
Creative Commons Attribution License 3.0†

Once we go beyond the registers in the memory hierarchy, we encounter caches. Caches are small amounts of SRAM that store a subset of the contents of the memory. The hope is that the cache will have the right subset of main memory at the right time.

The actual cache architecture has had to change as the cycle time of the processors has improved. The processors are so fast that off-chip SRAM chips are not even fast enough. This has lead to a multilevel cache approach with one, or even two, levels of cache implemented as part of the processor. Table 1 shows the approximate speed of accessing the memory hierarchy on a 500-MHz DEC 21164 Alpha.

Registers	2 ns
L1 On-Chip	4 ns
L2 On-Chip	5 ns
L3 Off-Chip	30 ns
Memory	220 ns

Table 1: Memory Access Speed on a DEC 21164 Alpha

When every reference can be found in a cache, you say that you have a 100% hit rate. Generally, a hit rate of 90% or better is considered good for a level-one (L1) cache. In level-two (L2) cache, a hit rate of above 50% is considered acceptable. Below that, application performance can drop off steeply.

One can characterize the average read performance of the memory hierarchy by examining the probability that a particular load will be satisfied at a particular level of the hierarchy. For example, assume a memory architecture with an L1 cache speed of 10 ns, L2 speed of 30 ns, and memory speed of 300 ns. If a memory reference were satisfied from L1 cache 75% of the time, L2 cache 20% of the time, and main memory 5% of the time, the average memory performance would be:

```
(0.75 * 10 ) + ( 0.20 * 30 ) + ( 0.05 * 300 ) = 28.5 ns
```

You can easily see why it's important to have an L1 cache hit rate of 90% or higher.

Given that a cache holds only a subset of the main memory at any time, it's important to keep an index of which areas of the main memory are currently stored in the cache. To reduce the amount of space that must be dedicated to tracking which memory areas are in cache, the cache is divided into a number of equal sized

*Version 1.3: Aug 25, 2010 11:04 am -0500
†http://creativecommons.org/licenses/by/3.0/

slots known as *lines*. Each line contains some number of sequential main memory locations, generally four to sixteen integers or real numbers. Whereas the data within a line comes from the same part of memory, other lines can contain data that is far separated within your program, or perhaps data from somebody else's program, as in Figure 1 (Cache lines can come from different parts of memory). When you ask for something from memory, the computer checks to see if the data is available within one of these cache lines. If it is, the data is returned with a minimal delay. If it's not, your program may be delayed while a new line is fetched from main memory. Of course, if a new line is brought in, another has to be thrown out. If you're lucky, it won't be the one containing the data you are just about to need.

Cache lines can come from different parts of memory

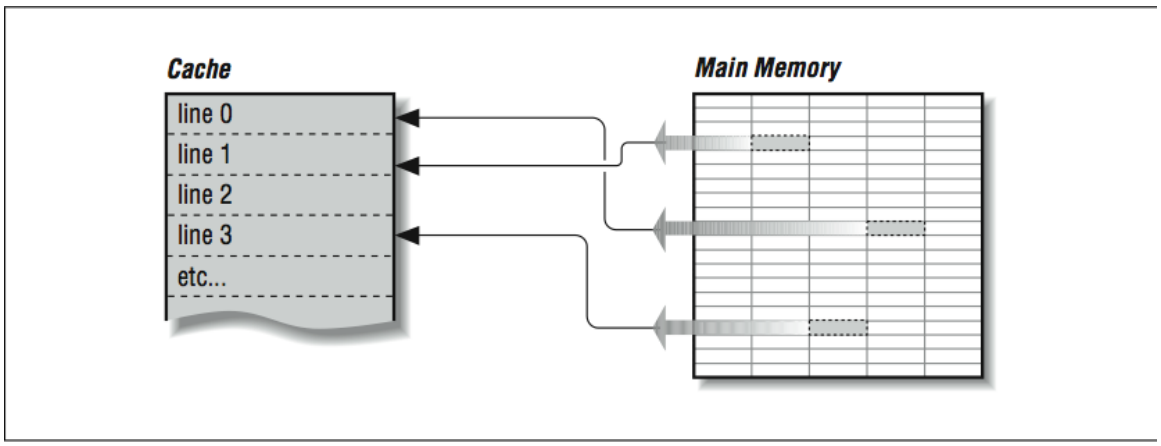

Figure 1

On multiprocessors (computers with several CPUs), written data must be returned to main memory so the rest of the processors can see it, or all other processors must be made aware of local cache activity. Perhaps they need to be told to invalidate old lines containing the previous value of the written variable so that they don't accidentally use stale data. This is known as maintaining *coherency* between the different caches. The problem can become very complex in a multiprocessor system.[1]

Caches are effective because programs often exhibit characteristics that help kep the hit rate high. These characteristics are called *spatial* and *temporal locality of reference*; programs often make use of instructions and data that are near to other instructions and data, both in space and time. When a cache line is retrieved from main memory, it contains not only the information that caused the cache miss, but also some neighboring information. Chances are good that the next time your program needs data, it will be in the cache line just fetched or another one recently fetched.

Caches work best when a program is reading sequentially through the memory. Assume a program is reading 32-bit integers with a cache line size of 256 bits. When the program references the first word in the cache line, it waits while the cache line is loaded from main memory. Then the next seven references to memory are satisfied quickly from the cache. This is called *unit stride* because the address of each successive data element is incremented by one and all the data retrieved into the cache is used. The following loop is a unit-stride loop:

[1] (<http://cnx.org/content/m32797/latest/>) describes cache coherency in more detail.

```
DO I=1,1000000
  SUM = SUM + A(I)
END DO
```

When a program accesses a large data structure using "non-unit stride," performance suffers because data is loaded into cache that is not used. For example:

```
DO I=1,1000000, 8
  SUM = SUM + A(I)
END DO
```

This code would experience the same number of cache misses as the previous loop, and the same amount of data would be loaded into the cache. However, the program needs only one of the eight 32-bit words loaded into cache. Even though this program performs one-eighth the additions of the previous loop, its elapsed time is roughly the same as the previous loop because the memory operations dominate performance.

While this example may seem a bit contrived, there are several situations in which non-unit strides occur quite often. First, when a FORTRAN two-dimensional array is stored in memory, successive elements in the first column are stored sequentially followed by the elements of the second column. If the array is processed with the row iteration as the inner loop, it produces a unit-stride reference pattern as follows:

```
REAL*4 A(200,200)
DO J = 1,200
  DO I = 1,200
    SUM = SUM + A(I,J)
  END DO
END DO
```

Interestingly, a FORTRAN programmer would most likely write the loop (in alphabetical order) as follows, producing a non-unit stride of 800 bytes between successive load operations:

```
REAL*4 A(200,200)
DO I = 1,200
  DO J = 1,200
    SUM = SUM + A(I,J)
  END DO
END DO
```

Because of this, some compilers can detect this suboptimal loop order and reverse the order of the loops to make best use of the memory system. As we will see in here[2], however, this code transformation may produce different results, and so you may have to give the compiler "permission" to interchange these loops in this particular example (or, after reading this book, you could just code it properly in the first place).

```
while ( ptr != NULL ) ptr = ptr->next;
```

The next element that is retrieved is based on the contents of the current element. This type of loop bounces all around memory in no particular pattern. This is called *pointer chasing* and there are no good ways to improve the performance of this code.

A third pattern often found in certain types of codes is called gather (or scatter) and occurs in loops such as:

```
SUM = SUM + ARR ( IND(I) )
```

where the IND array contains offsets into the ARR array. Again, like the linked list, the exact pattern of memory references is known only at runtime when the values stored in the IND array are known. Some special-purpose systems have special hardware support to accelerate this particular operation.

[2] "Floating-Point Numbers - Introduction" <http://cnx.org/content/m32739/latest/>