How Caching Works

by Guy Provost

If you have been shopping for a computer, then you have heard the word "cache." Modern computers have both L1 and L2 caches. You may also have gotten advice on the topic from well-meaning friends, perhaps something like "Don't buy that Celeron chip, it doesn't have any cache in it!"

It turns out that caching is an important computer-science process that appears on every computer in a variety of forms. There are memory caches, hardware and software disk caches, page caches and more. Virtual memory is even a form of caching. In this article, we will explore caching so you can understand why it is so important.

A Simple Example

Caching is a technology based on the memory subsystem of your computer. The main purpose of a cache is to accelerate your computer while keeping the price of the computer low. Caching allows you to do your computer tasks more rapidly.

To understand the basic idea behind a cache system, let's start with a super-simple example that uses a librarian to demonstrate caching concepts. Let's imagine a librarian behind his desk. He is there to give you the books you ask for. For the sake of simplicity, let's say you can't get the books yourself -- you have to ask the librarian for any book you want to read, and he fetches it for you from a set of stacks in a storeroom (the library of congress in Washington, D.C., is set up this way). First, let's start with a librarian without cache.

The first customer arrives. He asks for the book *Moby Dick*. The librarian goes into the storeroom, gets the book, returns to the counter and gives the book to the customer. Later, the client comes back to return the book. The librarian takes the book and returns it to the storeroom. He then returns to his counter waiting for another customer. Let's say the next customer asks for *Moby Dick* (you saw it coming...). The librarian then has to return to the storeroom to get the book he recently handled and give it to the client. Under this model, the librarian has to make a complete round trip to fetch every book -- even very popular ones that are requested frequently. Is there a way to improve the performance of the librarian?

Yes, there's a way -- we can put a **cache** on the librarian. Let's give the librarian a backpack into which he will be able to store 10 books (in computer terms, the librarian now has a 10-book cache). In this backpack, he will put the books the clients return to him, up to a maximum of 10. Let's use the prior example, but now with our new-and-improved caching librarian.

The day starts. The backpack of the librarian is empty. Our first client arrives and asks for *Moby Dick*. No magic here -- the librarian has to go to the storeroom to get the book. He gives it to the client. Later, the client returns and gives the book back to the librarian. Instead of returning to the storeroom to return the book, the librarian puts the book in his backpack and stands there (he checks first to see if the bag is full -- more on that later). Another client arrives and asks for *Moby Dick*. Before going to the storeroom, the librarian checks to see if this title is in his backpack. He finds it! All he has to do is take the book from the backpack and give it to the client. There's no journey into the storeroom, so the client is served more efficiently.

What if the client asked for a title not in the cache (the backpack)? In this case, the librarian is less efficient with a cache than without one, because the librarian takes the time to look for the book in his backpack first. One of the challenges of cache design is to minimize the impact of cache searches, and modern hardware has reduced this time delay to practically zero. Even in our simple librarian example, the latency time (the waiting time) of searching the cache is so small compared to the time to walk back to the storeroom that it is irrelevant. The cache is small (10 books), and the time it takes to notice a miss is only a tiny fraction of the time that a journey to the storeroom takes.

From this example you can see several important facts about caching:

- Cache technology is the use of a faster but smaller memory type to accelerate a slower but larger memory type.
- When using a cache, you must check the cache to see if an item is in there. If it is there, it's called a **cache hit**. If not, it is called a **cache miss** and the computer must wait for a round trip from the larger, slower memory area.
- A cache has some maximum size that is much smaller than the larger storage area.
- It is possible to have multiple layers of cache. With our librarian example, the smaller but faster memory type is the backpack, and the storeroom represents the larger and slower memory type. This is a one-level cache. There might be another layer of cache consisting of a shelf that can hold 100 books behind the counter. The librarian can check the backpack, then the shelf and then the storeroom. This would be a two-level cache.

Computer Caches

A computer is a machine in which we measure time in very small increments. When the microprocessor accesses the main memory (RAM), it does it in about 60 nanoseconds (60 billionths of a second). That's pretty fast, but it is much slower than the typical microprocessor. Microprocessors can have cycle times as short as 2 nanoseconds, so to a microprocessor 60 nanoseconds seems like an eternity.

What if we build a special memory bank, small but very fast (around 30 nanoseconds)? That's already two times faster than the main memory access. That's called a level 2 cache or an **L2 cache**. What if we build an even smaller but faster memory system directly into the microprocessor's chip? That way, this memory will be accessed at the speed of the microprocessor and not the speed of the memory bus. That's an **L1 cache**, which on a 233-megahertz (MHz) Pentium is 3.5 times faster than the L2 cache, which is two times faster than the access to main memory.

There are a lot of subsystems in a computer; you can put cache between many of them to improve performance. Here's an example. We have the microprocessor (the fastest thing in the computer). Then there's the L1 cache that caches the L2 cache that caches the main memory which can be used (and is often used) as a cache for even slower peripherals like hard disks and CD-ROMs. The hard disks are also used to cache an even slower medium -- your Internet connection.

Your **Internet connection** is the slowest link in your computer. So your browser (Internet Explorer, Netscape, Opera, etc.) uses the hard disk to store HTML pages, putting them into a special folder on your disk. The first time you ask for an HTML page, your browser renders it and a copy of it is also stored on your disk. The next time you request access to this page, your browser checks if the date of the file on the Internet is newer than the one cached. If the date is the same, your browser uses the one on your hard disk instead of downloading it from Internet. In this case, the smaller but faster memory system is your hard disk and the larger and slower one is the Internet.

Cache can also be built directly on **peripherals**. Modern hard disks come with fast memory, around 512 kilobytes, hardwired to the hard disk. The computer doesn't directly use this memory -- the hard-disk controller does. For the computer, these memory chips are the disk itself. When the computer asks for data from the hard disk, the hard-disk controller checks into this memory before moving the mechanical parts of the hard disk (which is very slow compared to memory). If it finds the data that the computer asked for in the cache, it will return the data stored in the cache without actually accessing data on the disk itself, saving a lot of time.

Here's an experiment you can try. Your computer caches your floppy drive with main memory, and you can actually see it happening. Access a large file from your floppy -- for example, open a 300-kilobyte text file in a text editor. The first time, you will see the light on your floppy turning on, and you will wait. The floppy disk is extremely slow, so it will take 20 seconds to load the file. Now, close the editor and open the same file again. The second time (don't wait 30 minutes or do a lot of disk access between the two tries) you won't see the light turning on, and you won't wait. The operating system checked into its memory cache for the floppy disk and found what it was looking for. So instead of waiting 20 seconds, the data was found in a memory subsystem much faster than when you first tried it (one access to the floppy disk takes 120 milliseconds, while one access to the main memory takes around 60 nanoseconds -- that's a lot faster). You could have run the same test on your hard disk, but it's more evident on the floppy drive because it's so slow.

To give you the big picture of it all, here's a list of a normal caching system:

- **L1 cache** - Memory accesses at full microprocessor speed (10 nanoseconds, 4 kilobytes to 16 kilobytes in size)
- **L2 cache** - Memory access of type SRAM (around 20 to 30 nanoseconds, 128 kilobytes to 512 kilobytes in size)
- **Main memory** - Memory access of type RAM (around 60 nanoseconds, 32 megabytes to 128 megabytes in size)
- **Hard disk** - Mechanical, slow (around 12 milliseconds, 1 gigabyte to 10 gigabytes in size)
- **Internet** - Incredibly slow (between 1 second and 3 days, unlimited size)

As you can see, the L1 cache caches the L2 cache, which caches the main memory, which can be used to cache the disk subsystems, and so on.

Cache Technology

One common question asked at this point is, "Why not make all of the computer's memory run at the same speed as the L1 cache, so no caching would be required?" That would work, but it would be incredibly expensive. The idea behind caching is to use a small amount of expensive memory to speed up a large amount of slower, less-expensive memory.

In designing a computer, the goal is to allow the microprocessor to run at its full speed as inexpensively as possible. A 500-MHz chip goes through 500 million cycles in one second (one cycle every two nanoseconds). Without L1 and L2 caches, an access to the main memory takes 60 nanoseconds, or about 30 wasted cycles accessing memory.

When you think about it, it is kind of incredible that such relatively tiny amounts of memory can maximize the use of much larger amounts of memory. Think about a 256-kilobyte L2 cache that caches 64 megabytes of RAM. In this case, 256,000 bytes efficiently caches 64,000,000 bytes. Why does that work?

In computer science, we have a theoretical concept called **locality of reference**. It means that in a fairly large program, only small portions are ever used at any one time. As strange as it may seem, locality of reference works for the huge majority of programs. Even if the executable is 10 megabytes in size, only a handful of bytes from that program are in use at any one time, and their rate of repetition is very high. Let's take a look at the following pseudo-code to see why locality of reference works (see How C Programming Works to really get into it):

```
Output to screen « Enter a number  between 1 and 100 »
Read input from user
Put value from user in variable X
Put value 100 in variable Y
Put value 1 in variable Z
Loop Y number of time
   Divide Z by X
   If the remainder of the division = 0
      then output « Z is a multiple of X »
   Add 1 to Z
Return to loop
End
```

This small program asks the user to enter a number between 1 and 100. It reads the value entered by the user. Then, the program divides every number between 1 and 100 by the number entered by the user. It checks if the remainder is 0 (modulo division). If so, the program outputs "Z is a multiple of X" (for example, 12 is a multiple of 6), for every number between 1 and 100. Then the program ends.

Even if you don't know much about computer programming, it is easy to understand that in the 11 lines of this program, the **loop** part (lines 7 to 9) are executed 100 times. All of the other lines are executed only once. Lines 7 to 9 will run significantly faster because of caching.

This program is very small and can easily fit entirely in the smallest of L1 caches, but let's say this program is huge. The result remains the same. When you program, a lot of action takes place inside loops. A word processor spends 95 percent of the time waiting for your input and displaying it on the screen. This part of the word-processor program is in the cache.

This 95%-to-5% ratio (approximately) is what we call the locality of reference, and it's why a cache works so efficiently. This is also why such a small cache can efficiently cache such a large memory system. You can see why it's not worth it to construct a computer with the fastest memory everywhere. We can deliver 95 percent of this effectiveness for a fraction of the cost.