CS 61C Lecture Notes Week 10

Topic: Cache

Reading: Patterson & Hennessy, Sections 7.1–7.3

I've mentioned several times that memory accesses used to be much slower than arithmetic operations, but that memory speed has been catching up. That's partly the result of faster electronics, but more importantly, it's the result of widespread use of the *cache* idea.

There is an inherent tradeoff between speed and size in developing memory technology. There are two reasons for this. First, for a small memory (such as the 32 general registers in the MIPS processor), you can afford to use the very fastest hardware possible, but you couldn't afford to scale that technology to a several-megabyte main memory or a gigabyte file storage. Second, cost aside, the bigger the memory, the longer it takes to *address* a particular word in the memory because more logic gate delays are involved.

Any computer has three levels of speed/size tradeoff:

1. A small number of registers inside the processor (32 words, 1 nanosecond).

2. A main memory for active programs and data (8 Mb, 10–100 nanoseconds).

3. A disk or other long-term file storage medium (1 Gb, 10 milliseconds).

The numbers in parentheses are very rough estimates, and change all the time. But the ratios don't change nearly as much.

The cache idea is to add a new level of speed/size tradeoff between number 1 (registers) and number 2 (main memory). A cache might hold 256 kilobytes and have an access time of 2–3 nanoseconds.

Transferring information between levels is generally done by explicit program control. For example, the program carries out a *lw* instruction to load a word from level 2 (memory) to level 1 (register). But the goal of the cache is to be invisible to the programmer; the program should think it's the main memory. In effect, we are trying to achieve the capacity of main memory (8 Mb) with the speed of cache memory (2 ns).

Locality of reference

Of course we can't really do it. If we had 8 Mb of cache, it would be just as slow as the main memory. But what we can do is this: Keep a copy of some of the stuff from memory in the cache. If the program tries to read something that happens to be in the cache, it'll work at cache speed. If the program tries to read something that *isn't* in the cache, then the transfer slows to main memory speed (and the pipeline stalls).

But it will turn out that the memory location the program wants is *almost* always already in the cache (probability 80–90%). That's because most computer programs have two useful qualities:

Temporal locality of reference: If the program uses memory location N, it will probably use location N again soon.

Spatial locality of reference: If the program uses memory location N, it will probably use N+4, N+8, etc., soon.

Temporal locality is what makes the cache idea possible at all; once we've copied a word of memory to the cache, that copying will probably pay off, several times. We use spatial locality by designing the cache so that it copies chunks of several words (e.g., eight words) at a time from memory.

Where to put the cache

There are two possible places to put a cache: (1) Between the main memory and the bus. (2) Between the processor and the bus.

The advantage of the first idea is that memory is used by i/o devices as well as by the processor; putting the cache near the memory means that every device benefits. But i/o devices are generally slower than memory anyway, so they won't benefit that much from caching. So this is outweighed by the benefit of the second idea: If the cache is near the processor, we can use high-speed technology (including simple things like more wires) to connect the cache to the processor. It's the cache-to-processor speed that we want to maximize. If the cache were near the memory, then cache-to-processor transfers would be limited by the speed and width of the bus.

Another implication of allowing a wide channel between cache and processor is that we can have *two* caches, one for instructions and one for data, and can transfer information from both caches into the processor at once. This will be important when we talk about pipelining.

The cache viewed abstractly

A cache is a *content-addressible* memory, also called an *associative* memory. In the usual memory arrangement, each word in the memory has a fixed address, and you fetch a particular word by sending its address to the memory. In an associative memory, each word doesn't have a fixed address. Instead, each entry (or *slot*) includes space to store an address as well as some data:

```
+-+--------------------------------+--------------------------------+
|V|              addr              |              data              |
+-+--------------------------------+--------------------------------+
```

The V (Valid) bit is on if this particular cache entry is in use. When your program asks for some memory address, the first step is that the processor sends the address to the cache. Every word of the cache checks to see if its address field matches the one you want, and its valid bit is on. If so, the cache sends the data field to the processor without bothering the main memory at all. If not, the cache forwards the request to the main memory. When the memory returns the desired data word, that word is both sent to the processor and also stored, along with the address, in a previously-vacant cache entry. (If no cache entries are vacant, the cache has to forget some old entry; we'll talk about the details later.)

When the program wants to store a word into memory, the word and its address go into a vacant cache slot. Some caches also send the word to the memory right away; others wait until this slot must be reused for a different word. (Details later.)

The crucial point is that all of this happens without the processor knowing anything about it. In principle we could unplug the cache, and just connect the processor directly to the bus, and the program would run the same (but more slowly).

The gory details

There are three factors that affect the performance of the cache:

1. The hit rate: What percentage of memory requests from the processor are already in the cache? We need a very high hit rate to maintain the illusion of a fast memory. For the best hit rate, the cache should have as many entries as possible, so that it can remember as many past requests as possible. Also, because of spatial locality, the cache should be as *wide* as possible. That is, when the processor requests a word of memory, we should actually read 4 or 8 or 16 words of memory into the cache, to be ready for anticipated future requests.

2. The cost of a hit. When the word we're looking for is already in the cache, how long does it take us to find it? This depends on the size of the cache; it takes longer to find a word in a bigger cache. So from this perspective we should keep the cache small!

3. The cost of a miss. Even though we hope to find everything already in the cache, sometimes we're going to have to read from main memory. We'd like that not to be any more expensive than it would have been without the cache. That argues for a *narrow* cache, so that we only have to wait for one word to be read, instead of a whole block of words. (But we can read the requested word first, let the processor continue, and

then read the rest of the block asynchronously. Then the processor is delayed only if *another* miss happens before we finish reading the entire block.)

Since these goals contradict each other, we have to do statistical measurements to choose good compromises.

One recent example of a compromise is the *two-level* cache. Since there are reasons for a small cache and reasons for a large one, we have two caches. Typically a fairly small cache (128kB, say) is built into the processor chip itself, and a larger cache (perhaps as much as 1Mb) can be connected external to the processor. The result is that, let's say, 70% of all requests are found in the small, fast cache, another 20% are found in the large, slower cache, and 10% are not found in either cache.

Associative vs. direct-mapped caches

When the cache fills up and another request comes in, the cache must delete one of its entries. For the best hit rate, we'd like to delete the least recently used entry, because of temporal locality of reference. In order to make this work, the cache must be able to copy any memory address into any cache entry, as in the abstract model shown earlier. A cache with this property is a *fully associative* cache.

Fully associative caches are very expensive to build. It's hard to look at every cache entry for the desired address without spending a long time in comparison circuitry. Designers therefore looked for simpler, almost-as-good solutions.

One possibility is to have a function that maps every possible memory address into a particular cache slot. When a request comes in for a certain address, the cache will only have to look at that one slot to determine whether or not it already has the desired value. This is called a *direct mapped* cache.

Suppose for the sake of argument that we have an 8Mb main memory and a cache with 8 entries. It would be very bad to decide that the first megabyte of memory is mapped to the first cache slot, the second megabyte to the second slot, and so on. Because of spatial locality, almost every memory reference would have to use the same slot that the previous reference used! We would have a hit rate near zero. Instead we use the low-order bits of the address to choose the slot.

Here's an example. Let's say we have a 256Kb cache, eight words (32 bytes) wide. 256K is 2^{18}; 32 is 2^5. So this cache has 2^{13} (8K) entries. When the processor wants to load a word of memory, the cache interprets the address this way:

```
+--------------+-------------+-----+
|  lookup key  | slot number |offst|
|      14      |      13     |  5  |
+--------------+-------------+-----+
```

The rightmost five bits of the address are the offset of the desired information (word or byte) within a cache entry. The next 13 bits tell the cache which entry to use. The remaining 14 bits must match the address bits stored in the cache slot along with the eight data words. (The cache entry need not store the slot number because that will be constant for any given slot, and need not remember the offset because the slot stores the entire 8 words.)

A direct mapped cache is much simpler to build than an associative cache, so its hit cost (time to find an entry) can be less, and so can its money cost. On the other hand, its hit rate could be dramatically worse than that of a fully associative cache. In practice, for most kinds of programs, the hit rate is only 2–5% worse.

If even that's too much of a penalty to pay, there is a compromise called a *set associative* cache. Let's take our same 8K cache entries and group the slots in pairs. Any desired memory address will be mapped into one of these pairs, but can occupy either slot of the pair. (In this case we'll use 12 bits of the address to determine the slot-pair number, instead of 13 bits to determine the exact slot.) It turns out that this two-way set associative cache has a hit rate only about 1% worse than that of a fully associative cache, but it's easier to build. (It's also possible to have a four-way set associative cache, in which the slots are organized in groups of four, and a given address can be in any of those four slots, but this turns out not to be much better than two-way set associativity.)

Replacement policy

If a cache is direct mapped, then there's only one choice for which old data to forget when a new slot is needed.

But if there is any associativity (full or set), the cache must choose which slot to vacate. The theoretical best thing is to remember how recently each slot was used, and some caches have been built that way. But the extra processing to keep track of that information is expensive and also hurts the cost of a hit.

Perhaps surprisingly, it turns out that a *random* choice is almost as good as an LRU (Least Recently Used) choice, and of course it's much less costly. This is analogous to the idea you probably learned in 61B about choosing the partition point for a quicksort: Choosing the median value is theoretically optimal, but choosing a random value is much faster and almost as good in practice.

When we study virtual memory next week, we'll deal with almost the same question in that context, but the answer will be different; it's instructive to compare these two topics later.

Memory updating policy

When a program tries to store a word into memory, there are two choices for the cache designer. A *write-through* cache remembers the data in the cache, but also sends it immediately over the bus to the main memory, so that the memory is always up to date. A *write-back* cache doesn't update the main memory until the cache slot is needed for another memory location.

The advantage of a write-back cache is that it can be faster, for two reasons. First, it avoids most pipeline stalls from a store instruction that has to wait for the bus to be free. Second, if (as is likely) several consecutive words of memory are written, the entire cache slot can be transferred to memory in one operation, instead of a separate write for each word.

On the other hand, the write-back cache slows down *loads* from memory, because a cache miss now requires writing the old data back to memory before the new data can be read. Since most programs do more loading than storing, this may be a bad bargain.

Also, if the contents of main memory aren't always up to date, attempted memory references by i/o devices are problematic. The operating system will have to know about the cache, and will have to *flush* the cache (explicitly write everything back to memory) before starting an i/o operation. These days, an even more difficult related problem is the use of multiple processors on the same memory bus, each with its own cache. If each cache is making different delayed modifications to memory, it's very hard for one processor to find out what another one has changed.

Nevertheless, most caches are write-back these days, with special circuitry to allow different processors' caches to communicate.