Lecture #35 – May 5, 2004

I Feel the Need for Speed

There are a number of methods for increasing the throughput of a processor. The first and most obvious is to increase the raw clock speed. While this is a laudable goal, all components of the processor chip have to be able to handle the increased speed. All the flip-flops have to keep up with the shorter time periods between clock pulse transitions, and all signals have to propagate down the wires to the inputs of the next devices before they are needed. At 11.8 inches per nanosecond, the speed of light becomes a serious factor in electronics design. As processors become faster, they must also become smaller. As devices become smaller, more heat must be dissipated in a smaller area as well. Since we can't beat the speed of light, we must turn to other ways of increasing throughput.

If you think about the speeds of the various components, you might have a 2 GHz processor coupled to 10 ns (nanosecond) memory, for example. Running the numbers tells us that one cycle of the processor takes ½ ns, much faster than the access time of the main memory. While these numbers will be different for different systems, processors are generally much faster than their primary memory. Rather than have a fast processor spend most of its time waiting for every memory access, twiddling its thumbs so to speak, we can beat the clock by inserting a small amount of ***cache memory*** between the processor core and primary memory.

Cache is much faster than primary memory, but speed comes at a price. Fast static memory takes six transistors per cell while slower dynamic memory requires only one, so any given chip area can hold only about one-sixth of the number of static cells as dynamic cells. Putting cache onto the processor chip gives the fastest response of all since the memory cells operate at the native speed of the processor logic, but the amount of memory is even more limited by the available area on the chip not otherwise occupied by logic circuits. The cache on the CPU chip is called the ***L1 cache***.

When a word is requested from primary memory by the processor, a copy is also placed into the cache. The next time that same word is requested, the processor can get it from the cache much faster than it can get it from primary memory. Since the cache is much smaller than primary memory it will quickly fill; placing a new word into the cache requires that some other word be discarded. One common replacement strategy is to discard the word used the longest time ago; this is called the ***LRU*** or ***Least Recently Used*** strategy.

In many memory systems it takes less time to read a contiguous block of words all at once than to read one individual word at a time. In such systems each cache entry contains that same number of words. Requesting one particular word from memory causes its corresponding block to be read in and placed into the cache (thus discarding the least recently used block). This makes practical sense as well, since there is a high probability that the next requested word (either instruction or data) will be from the same

block. Many caches are split into two sections, one exclusively for instructions and the other exclusively for data. Requesting a word located in the cache is called a ***cache hit***, and requesting a word not present is called a ***cache miss***.

Storing a word into memory spawns off its own set of problems with the cache. One approach, called ***write-through***, always updates primary memory on a store instruction, as well as updating the appropriate word in the cache. The cache and primary memory are always in agreement, but system performance suffers while waiting for primary memory to be updated. The second approach, called ***write-back***, changes only the cache on a store instruction. Cache and primary memory are no longer in agreement, and only when a modified block is to be discarded from the cache is it written back to primary memory. Unmodified blocks do not need to be written back before they are discarded. A special bit associated with each block, called the ***dirty bit***, is set by a store instruction to indicate that the block no longer agrees with primary memory and needs to be written back.

If a program spends a large amount of time in a section of code small enough to fit entirely into the cache, that section of code will run at the native speed of the processor. As the program progresses from one region of code to another, the cache adapts to the changes in program locality. If the cache is too small for a given section of code, however, blocks will be continuously loaded and discarded. In the worst possible case, every memory reference causes a cache miss. This is a condition known as ***thrashing***; performance suffers greatly as a result.

One approach to increasing the total amount of cache in a system (without redesigning the CPU chip) is to insert a second cache between the processor and primary memory. This cache, called the ***L2 cache***, is often significantly larger and slower than the L1 cache on the CPU chip, but still much smaller and much faster than primary memory. A miss on the L1 cache will often hit on the L2, and even when the L1 cache is thrashing the performance will not suffer as much as if every cache miss had to reference primary memory.

An L2 cache of 128K bytes usually results in a hit rate of roughly 94%; increasing to 512K brings that number up to about 96%. A full gigabyte of cache pushes that to around 97%. It is impossible to reach 100%, as values must be initially fetched and eventually stored into main memory at some point. In the Pentium II™, the processor "cartridge" contains both the main CPU chip and a second L2 cache chip.

In a future lecture we will look at how cache can be implemented using ***associative*** (content addressable) memory, and how ***set associative cache*** memories minimize thrashing. We will also examine super-scalar and pipelining techniques for increasing throughput even more.