**Understanding Datacenter Software Dynamics**

Dr. Richard L. Sites, Visiting Professor

2016.08.17

**Week 2**

**NOTE: mystery2.cc is now in /home/dclab and all four machines are backed up and available again. Grades and some brief cache-related readings are now posted at http://www.comp.nus.edu.sg/~sites/cs6280/week1.html**

**Course Outline**

Datacenter computing favors response time over throughput. This contrasts deeply with batch environments that prize throughput or total computation accomplished but ignore latency entirely.

The focus of this course is understanding and controlling sources of latency, especially the long-tail delays that ruin a user experience. To do so, we will design and build software tools to measure not only the latency of various software and hardware subsystems, but to record the unexpected ways that these layers interact. It is precisely these unexpected dynamics that produce the most difficult latency problems, which almost all turn out to be easy to fix or control once the actual dynamics are visible. Controlling slow response times results in more efficient server use, which is worth literally millions of dollars within the datacenter industry. Your tools will run in a tiny datacenter environment and reveal real delays and real program interference.

Prerequisite skills: ability to program in C or C++; ability to edit files and run simple commands on LInux; some familiarity with data structures and computer architecture at an undergraduate level; some familiarity with multithreaded programs and asynchronous I/O; a dogged sense of curiosity.

**Week 2: Memory Timing**

This is the most complex topic we will have in the first half of the semester. We will look simultaneously at several design layers, each with its own concepts and constraints. But at the end of the week, you will have a much better understanding of real computer memory systems. The layers:

* C programmer
* Compiler
* Assembly language
* CPU instructions
* Virtual memory
* DRAMs

The goal this week is to measure the size and organization of the four-level memory subsystem on our lab machines. dclab-1 and -2 are Intel G1820 chips with three levels of cache on chip, and 4GB of DRAMs off chip on a single DIMM. dclab-3 and -4 are Intel i3-4150 chips also with three levels of cache on chip (L3 is bigger than on the G1820) and 8GB of DRAM off chip, two DIMMs. Each CPU chip has two physical cores, and each core has a dedicated L1 instruction cache and dedicated L1 data cache, plus a dedicated L2 combined cache. Each chip has a single L3 cache shared across both cores. dclab-3 and -4 have cores that each have two sets of program counters and registers (hyperthreads), so appear to be four logical cores, but note that there are not four dedicated L1 cache pairs. Hyperthreading is best thought of as a cheap technique to have some other instructions to execute while one logical core is waiting for memory. In my experience, two hyperthreads give about 1.5x the processing power of a single thread CPU core.

Our measurement will be done in three steps, the first two of which are done by the supplied mystery2.cc program:

* Determine the cache line size
* Determine the total size of each level of cache
* Determine the associativity of each layer of cache

**Cache line size**

Cache memory is organized as lines (or blocks) of several bytes all of which are brought into or moved out of the cache together. The minimum useful line size holds two pointers, which means on our 64-bit chips two 8-byte pointers or 16 bytes total. The maximum useful line size is one page in a virtual memory system, which means 4KB in our chips. Thus, the only reasonable line sizes are 16, 32, 64, 128, 256, 512, 1024, 2048, and 4096 bytes. The most common choices in the industry are 64 and 128 bytes, but 32 and 256 are sometimes seen.

It is possible to look up in (often obscure) manufacturer literature such as a chip datasheet what the actual cache organization is, but we want to find out via programs that can run on a wide variety of machines.

In the discussion below and in your subsequent professional careers, the term “aligned” is prominent. An aligned reference to a four-byte item is at a byte address that is a multiple of four. An aligned reference to an eight-byte item is at a byte address that is a multiple of eight, etc. Actual cache and memory access are only done in aligned quantities. Unaligned references are done by accessing two aligned locations and doing some shifting and anding to select or modify some of those bytes.

How can we have a program determine the cache line size? There are several design choices and several available dead ends. One choice is to start with a cache containing none of the data we are about to access, load an aligned word at location X and then load the word at X + c, where c is a possible cache line size. If c is less than the line size, loading X + c will hit in the cache because it was brought in when X was brought in. If c >= the line size, loading X + c will not hit in the cache. Assuming that cache hits are O(10) times faster than cache misses, we can time a loop doing a few hundred loads to different possible cache lines and distinguish those values of c that are fast and those that are slow. The smallest slow c is the cache line size.

**What would we expect to see?** It is an important professional design discipline to ask this question at this point and to *write down or sketch* what you expect. Only then can you react to getting something quite different. Assume for a moment that the real cache line size is 128 bytes. Then for a linked list of 16-byte items, the first eight items all fit into one cache line, the second eight into another, etc. So if we fetch 200 items, we would expect to see 200 / 8 = 25 cache misses. If instead we make a list of 32-byte items, we would expect to see 200 / 4 = 50 cache misses. For 128-byte items and above, we would expect to see 200 cache misses every time. If we sketch a graph of the time per access (load), we expect something like this:

cycles per load

possible line size, log scale

where the time per load starts out as say 1/8 the real miss time, then 1/4, then 1/2, then all the rest are the real miss time, 100% misses.

This choice turns out to be a poor one. Sigh. Modern caches often prefetch line N +1 when line N is accessed, perhaps whenever the memory subsystem is otherwise idle. Also, modern CPUs do out-of-order execution, launch multiple instructions per CPU cycle, and may well have 2-24 outstanding loads waiting in parallel for memory responses. The effect can be to make ten cache misses take about the same time as just one miss, reducing the apparent miss time by a factor of 10.

The supplied program mystery2.cc implements this choice in the routine NaiveTiming(), so you can see that it gives uniformly misleading results.

Another choice is to build a linked list of items in memory, with each item the size of a possible cache line and aligned on a cache line boundary. Each item contains a pointer to the next item. Then if ptr points to the first item, a loop of ptr = ptr->next will do a series of loads, but the address for each load depends on the value fetched by the previous load, so they must be executed strictly sequentially. This design also allows us to see how long the load-to-use latency is for each level of the memory subsystem.

The supplied program mystery2.cc implements this choice in the routine LinearTiming(), so you can see that it gives better but also misleading results. The problem is that the linked lists defeat the many-loads-outstanding parallelism, but do not defeat cache prefetching.

To (attempt to) defeat the prefetching, we build the linked lists so that the items are scrambled around in the address space.

Linear:

16

Scrambled:

16

It turns out there is one more problem. We are counting cache misses by timing a few hundred memory loads and expecting twice as many misses to take about twice as long. But when you look at main memory DRAM designs, you find a surprise. Access times are not random. Actual DRAM chips first access a row of a big array of bits and then access a column of bytes from that row. The row access is actually preceded by a so-called precharge cycle to set the internal data lines to about half the power-supply voltage, so only a little bit of charge in a bit cell is needed to get them off dead center. Rows are typically 1024 bytes, and a dual inline memory module (card, DIMM) typically has eight DRAMs cycled in parallel to produce eight bytes at once, driving an 8-byte-wide memory bus to the CPU chip. In this case, the effective row size is 8 \*1024 = 8192 bytes. A CPU with two memory buses connected to two DIMMs can cycle each memory bus independently, so can move twice as many bytes per second. Usually, successive cache lines are stored in alternating DIMMs to achieve the 2x bandwidth increase on blocks of data. In this case, the effective row size is 2 \* 8KB = 16KB.

If two successive accesses to a DRAM are in two different rows, the precharge, row access, column access sequence is done for each. But if the second access is to the same row as the first one, the CPU and DRAM hardware implement a shortcut: just the column access is done. This is approximately three times faster than the full sequence. A 3x variation in memory access time is a large amount of distortion when we are trying to measure a 2x difference in total time. So we also have to defeat the DRAM shortcut timing.

We do this when building the linked lists by flipping the 16KB addess bit in every other list address, explicitly putting successive items into different DRAM rows. In the supplied mystery2.cc program, routine MakeLongList(), this is the purpose of the variable extrabit .

The supplied program mystery2.cc implements all this in the routine ScrambledTiming(), so you can see that it gives better results that finally look a bit like our sketch above.

**Cache total size of each level**

Once we have the cache line size, we next want to know for each level of the cache hierarchy how big it is. The total size of the L1 cache informs data structure designs that have working sets that likely fit into the L1 cache and hence are faster than designs that don’t fit. Similarly for L2 and L3.

Our strategy for finding the size of each cache level is to read N lines into the cache, then reread, looking at timings. If all N lines fit in the cache, then the rereads will be fast. If they don’t fit, then the rereads will miss and be filled from the next level down (assuming, as is always the case, that each level is bigger than the previous one). In contrast to the previous section where we were either hitting in the L1 cache or going to main memory that is O(100) times slower, our measurements in this section will be comparing L1 hits to L2 hits or L2 hits to L3 hits, etc. where each level is only O(10) time slower than the previous. So our signal-to-noise ratio is not very good.

What would we expect to see? For N <= L1 total size, we would expect the rereads to be fast at just a few cycles per load, the L1 access time. For L1size < N <= L2size, we would expect to see mostly the somewhat slower L2 access time, etc. So maybe like this sketch:

cycles per load

possible total cache size, log scale

...

Since there will be some noise in this measurement, it is helpful to run each experiment several times. The supplied mystery2.cc routine FindCacheSizes() does each timing measurement four times. The first time initializes by loading from main memory, so is slow and is to be ignored here. The next three times should all be similar and show the time difference we seek.

**Cache associativity of each level**

Once we have the line size and the total size of each cache level, we can find the associativity of each cache. In a set-associative cache, a given cache line cannot go any place at all in the cache (called fully associative), but only in a small number of places, one set. If a given cache line can only go one place in the cache, the set size is 1 and the cache is said to be direct mapped or 1-way associative. If a cache line can go two places in the cache, it is 2-way associative, etc.

An N-way associative cache usually accesses all N possible locations in parallel (using N times as much power as accessing one location), and also accesses N cache tags in parallel. If one of the tags matches the given address, there is a cache hit and the corresponding data is used. If none match, there is a cache miss at that level. (If many match, there is a hardware design screwup.)

Our strategy for finding the cache associativity at each cache level is to read N lines into just set[0] of the cache, then reread, looking at timings. If all N lines fit into set[0], then the rereads will be fast. If they don’t fit, then the rereads will miss and be filled from the next level down. The largest number N that fit at once is the cache associativity. You get to do this one.

**Translation buffer time**

We aren’t quite done with complications. While accessing memory in today’s processors, we are also accessing the virtual memory page tables. Reading tens of megabytes of data to trash the caches also trashes the hardware translation lookaside buffer in a CPU core. Subsequent memory accesses must first do a memory access to load the corresponding TLB entry, at least doubling the total time.

If there are 16-byte items being accessed, 256 of these fit in a single 4KB page, so accessing all 256 involves just one TLB miss. But if there are 4KB items being accessed, loading from 256 of these will also get 256 TLB misses. The TLB miss times distort all our measurements above, but in a predictable way.

**Cache underutilization**

The final complication we discuss is that of cache underutilization. Normally, the lowest bits of a memory address are used to pick which bytes to use in a cache line. If the line size is 32 bytes, this would be the lowest 5 bits. The next higher bits are used to select the associative set. If a cache size is 4KB of 32-byte lines, there are 128 cache lines total. If these are organized 2-way associative, then there are 64 sets of two cache lines each. So the next higher 6 bits of an address would be used to select the set, and the remaining higher bits compared against the two tags to determine a hit.

If we load data into a cache with 32-byte lines, but ONLY load data spaced at multiples of 128 bytes, two of the six address bits used to select a set will always be 00. So set[0], set[4], set[8], etc. will be used, but the other 3/4 of the sets will not be used. This means that the effective cache size is not 4KB but only 1KB. Something to keep in mind when accessing regular address patterns.

**Assignment 2**

Starting with the supplied mystery2.cc program *compiled optimized*, -O2, answer some questions, do some modifications, and then answer some more questions. You shouldn’t need to spend more than about 2 hours on the initial questions 2a-2g, plus maybe two more on question 2h. If you are spending much more time, set it aside and do something else, or come chat so we can get you back on track.

You will find it helpful to use Excel or Google Charts or whatever to turn the numbers into graphs so you can compare them to your sketches and think about what the patterns are telling you. *Always* label the x and y axis of your graphs, giving the units: counts, cycles, msec, KB, etc. Do this even if you are the only one looking at the graph. At some point, it will save you from making an order-of-magnitude error, or from coming to a completely false conclusion. Once you develop this habit, it will save time.

Before you change things, rerun mystery2 three to five times to see what the variation is like in the cycle counts.

**2a.** In the first part of mystery2.cc that looks at cache line size timings, what do you think the cache line size is and why?

**2b.** In the first part of mystery2.cc that looks at cache line size timings, explain a bit about the three timings for a possible line size of 256 bytes. These are the ones that should be about 30, 80, and 200 cycles per load.

**2c.** In the first part of mystery2.cc that looks at cache line size timings, make a copy of the program and in routine MakeLongList(), add a line after

int extrabit = makelinear ? 0 : (1 << 14);

that defeats the DRAM different-row address patterns:

extrabit = 0;

Explain a little about the changes this produces in the scrambled timings, especially for possible line size of 128 bytes.

**2d.** In the second part the looks at total cache size FindCacheSizes(), what do you think are the total sizes of the L1, L2, and L3 caches?

**2e.** What is your best estimate of the load-to-use time in cycles for each cache level?

**2f.** Rerun on dclab-3 or -4, which have 3MB L3 caches. How would you modify the program slightly to test for somewhat-common not-power-of-two sizes that are 3 and 5 times a power of two. (You need not do the modification; just explain what you would do.)

**2g.** In the second part that looks at total cache size FindCacheSizes(), explain a bit about the variation in cycle counts *within* each cache level. The ones that barely fill a level are somewhat faster than the ones that completely fill a level. Why could that be?

**2h.** Implement FindCacheAssociativity(). What is the associativity of each cache level?

tag

offset

The number of bit of set index is (Cache\_size/64/associative)

6bit for 64B cache block

Set index