Software Systems

Day 16 - Latency and Caching

Latency and Security

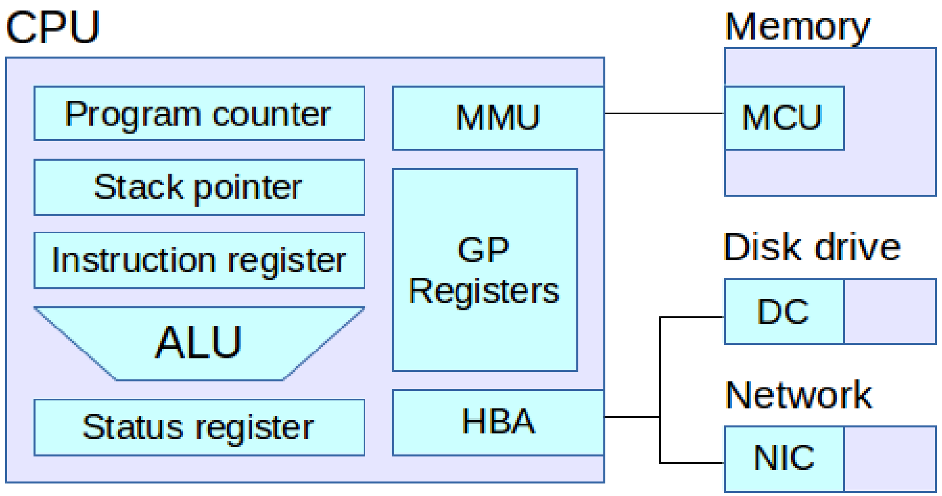
- Optimizing for latency is important, but it can affect security.
- A side-channel attack uses some metric like latency to break security.
- https://www.youtube.com/watch?v=2-zQp26nbY8

Latency

- CPUs generally follow a three-step process when running:
  - Fetch: get the next instruction from memory and store it in a register.
  - Decode: signal the appropriate parts of the CPU based on the instruction.
  - Execute: run the computation signaled by the instruction.
- Most instructions are loading/storing memory.
- It's important to consider the **latency** of different operations of your machine: how long it takes to do something.
- Sometimes, small time differences can matter!

Latency

- The CPU has its registers, which can be accessed with minimal latency.
- Memory is slower, but still relatively fast.
- Accessing the disk is considered pretty slow.
- Transmitting or receiving over the network is very slow.
- There are other tricks to speed these up, like caching (which we'll see in a bit).



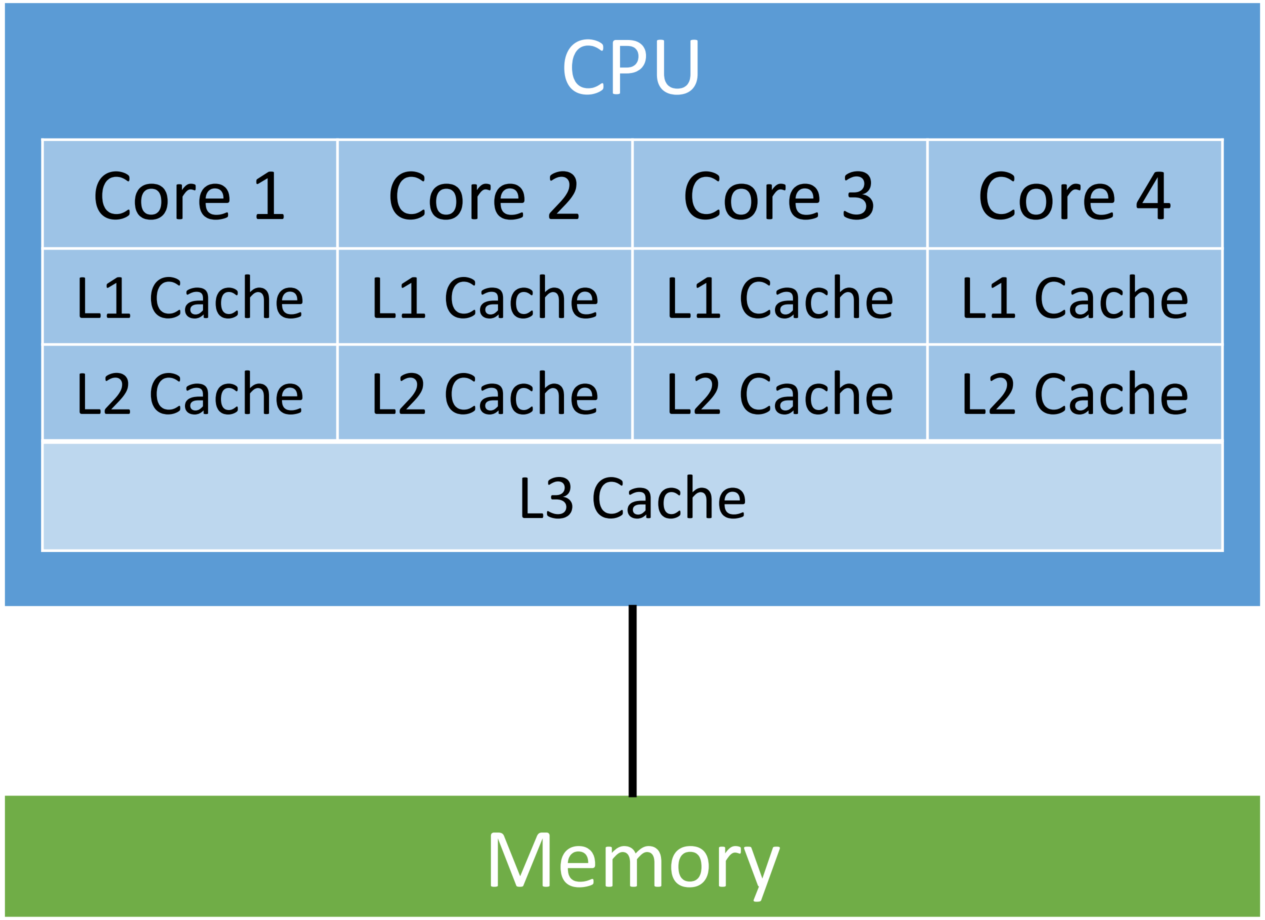
Latency

- Exercise:
  - See the Latency Numbers Every Programmer Should Know: https://bit.ly/3z4hp1q
  - When did the latency of a main memory reference last change?
  - In that time, how have the following changed:
    - L1 and L2 cache reference
    - Random SSD read
    - Disk seek
  - Note any other interesting patterns you see as you move the slider.

Some Latency Takeaways

- Memory hasn't gotten much faster.
- The latency of cache, SSD, HDD references have decreased.
- Note that these are all O(1) references - we know exactly where the memory is, but it still takes time.
- The sheer speed of the cache over memory means that good caching can make a big difference in program performance.

How Caching Works



How Caching Works

- Think of caching in terms of making chocolate chip cookies:
  - The registers are your hands - just drop the ingredients in.
  - The counter is the L1 cache - grab the ingredients, then drop them in.
  - The fridge is the L2 cache - open it, move things around, grab the ingredients, then drop them in.
  - The basement is the L3 cache - go downstairs, dig around, bring the ingredient back.
  - Main memory is the corner store - put on pants, grab your keys/wallet, walk down the street, buy the ingredients, come back, etc.
  - The disk is like driving to the next city over to get your ingredients.

How Caching Works

- Accessing memory? Check if it's cached first.
- If so, *cache hit*: get it from the cache.
- Otherwise, *cache miss*: get it from memory, and cache it.
    - If the cache is full, *evict* something from the cache.
    - Usually, cache a *line* of data at that memory address.
- If evicting data that's been changed, write it back to memory/disk.
- Caching "bubbles up": some memory is cached in L3, some of that in L2, and some of that in L1.

Caching and Latency

- Exercise:
  - See the Latency Numbers Every Programmer Should Know: https://bit.ly/3z4hp1q
  - Suppose we scaled these numbers up by a billion (1 ns -> 1 s). Find real-world events that would correspond to the time it takes to get data from:
    - The L1 cache
    - The L2 cache
    - Main memory
    - An SSD
    - A hard drive
    - The network

Average Access Time

- It's helpful to think in terms of average access time to quantify how helpful your cache is.
- Consider the cache hit rate (percentage of accesses that result in a cache hit), or the cache miss rate.
- Average access time = cache hit rate * cache reference latency + cache miss rate * memory reference latency

Average Access Time

- Exercise:
    - Suppose your cache consists entirely of a single line of 32 bytes. For purposes of this exercise, assume that an integer is 4 bytes, a cache reference takes 1 ns, and a memory reference takes 100 ns.
    - Suppose that you have an array of 8 integers, and that you iterate through every item in the array. What is the average access time for an element?
    - Suppose that you have an array of 32 integers, and that you iterate through every 8th item in the array. What is the average access time for an element?
    - Suppose that you have an array of 8 integers, and that you iterate 100 times through every 8th item in the array. What is the average access time for an element?

Improving Cache Performance

- Temporal locality
  - If you need to use a glue stick to glue 100 pages in one sitting, you don't put it back into the drawer in between uses.
  - Using the same data multiple times in a time frame benefits from caching.
- Spatial locality
  - If you know you're going to need eggs and milk, you don't make two trips to the grocery store to do it.
  - Using data close together in memory also benefits from caching.

Improving Cache Performance

- For a better cache hit rate:
  - Temporal locality: use the same cache line multiple times.
  - Spatial locality: use more of a cache line.
- Improving your cache hit rate means your average access time will get closer to the cache reference latency rather than memory latency.
- To see information on your cache, run lscpu or x86info (both on Linux - you may have to install x86info first).

Improving Cache Performance

- Exercise:
  - The class-sessions/16-latency-caching folder has two example programs.
  - `./cache_check X Y` adds together integers from an array of length X, reading every Yth element.
  - `./square_sum N V` creates an NxN matrix and adds them together, iterating row by row if V is nonzero and column by column if V is zero.
  - Build these, and then run them in the command line like this:
    `perf stat -B -e cache-references,cache-misses ./prog P1 P2`
    (where prog is the program and P1/P2 are the parameters)
  - Try different sets of parameters - what patterns do you notice in the cache miss percentage?

Closing Thoughts on Caching

- Good performance:
  - High spatial locality (e.g., arrays, strings)
  - High temporal locality and small enough to fit in cache
- Bad performance:
  - Iterating over columns vs. rows
  - Using small chunks of data once
  - Using large chunks of data that don't fit in cache

Closing Thoughts on Caching

- What about streams of instructions?
- High spatial locality: straight-line code (no branches/loops)
- Low spatial locality: branches, jumps, function calls
- High temporal locality: loops