CS 24: Introduction to Computing Systems

The Memory Hierarchy

Data stored in memory need to somehow get to the CPU, but how does this happen? Today, we will discuss the way modern computer systems make accessing memory as fast as possible given constraints on hardware and price.

Memory

Most memory on your computer is *Random Access Memory* (or RAM) which can be written to and read from many times. There are two types of RAM: Dynamic RAM (DRAM) and Static RAM (SRAM). SRAM is significantly faster than DRAM, but it is also much more expensive. In an ideal world, every computer would have a big pool of SRAM and not need DRAM (or disk!), but this is not how it actually works. In practice, computers have a small amount of SRAM, a larger amount of DRAM, and a huge amount of disk. The big question to answer is how to use this memory; in particular, which data should be put in the fast memory and which in the slow?

Caching

A *cache* is a small lookup-table that stores results that have already been computed or looked up; they are used to speed up future requests for the same result. You've already seen several examples of caches in previous courses as well as your daily life:

- **Memoization.** In CS 2, you saw how storing a *hash table of results* (i.e., a cache) could speed up an algorithm from exponential to polynomial runtime.
- **Loading Websites.** When you reload a website, it usually appears much faster the second time. This is because your web browser is *caching* the images and scripts loaded by the webpage on your local machine.
- **Compilation.** When you `make` your projects, only the files that changed are re-compiled. The disk is used as a *cache* for the binaries you're creating.

Perhaps unsurprisingly, in addition to the *software* cache examples above, computers use *hardware* caches as well. In particular, rather than divvying up who gets fast memory and who gets slow memory, computer systems utilize the fast memory as a *cache* for the slow memory. In other words, when data is fetched by a program, it makes its way from the slow memory to the fast memory to the registers. *The same data is stored multiple times at different levels of this hierarchy.*

None of this would be necessary if the latencies of SRAM and DRAM weren't significantly different; so, it can be educational to get a sense of just how big the difference is. To do this, we need a reliable way of measuring performance.

Performance Via Clock Cycles

A *clock cycle* is a single "tick" of the processor. It's the smallest unit of time in which something can occur, and usually a small number of instructions can be run in one clock cycle. We can use the *clock rate* of a processor to convert between clock cycles and time. For example, the clock rate of `compute-cpu2` is 1995 MHz which means one cycle takes approximately 0.5 ns.

Timing Reads and Flushing the Cache

To determine tnhe difference between using the cache and not using the cache, we use two primitive which are built on assembly instructions provided by Intel. First, we can time how many clocks a single read takes, and second we can flush a particular address from the cache. This functionality is provided in `util.c` by the functions `time_read` and `flush_cache_line`, respectively.

An Experiment

Finally, we have all we need to do an experiment between using the cache and only using main memory. Consider the following code snippet:

```
1  const uint64_t N = 1000000;
2
3  int main(int argc, char *argv[]) {
4     char *array = malloc(N * sizeof(char));
5     uint64_t result = 0;
6     for (int i = 0; i < N; i++) {
7        result += time_read(&array[i]);
8        flush_cache_line(&array[i]);
9     }
10    printf("%llu\n", result);
11 }
```

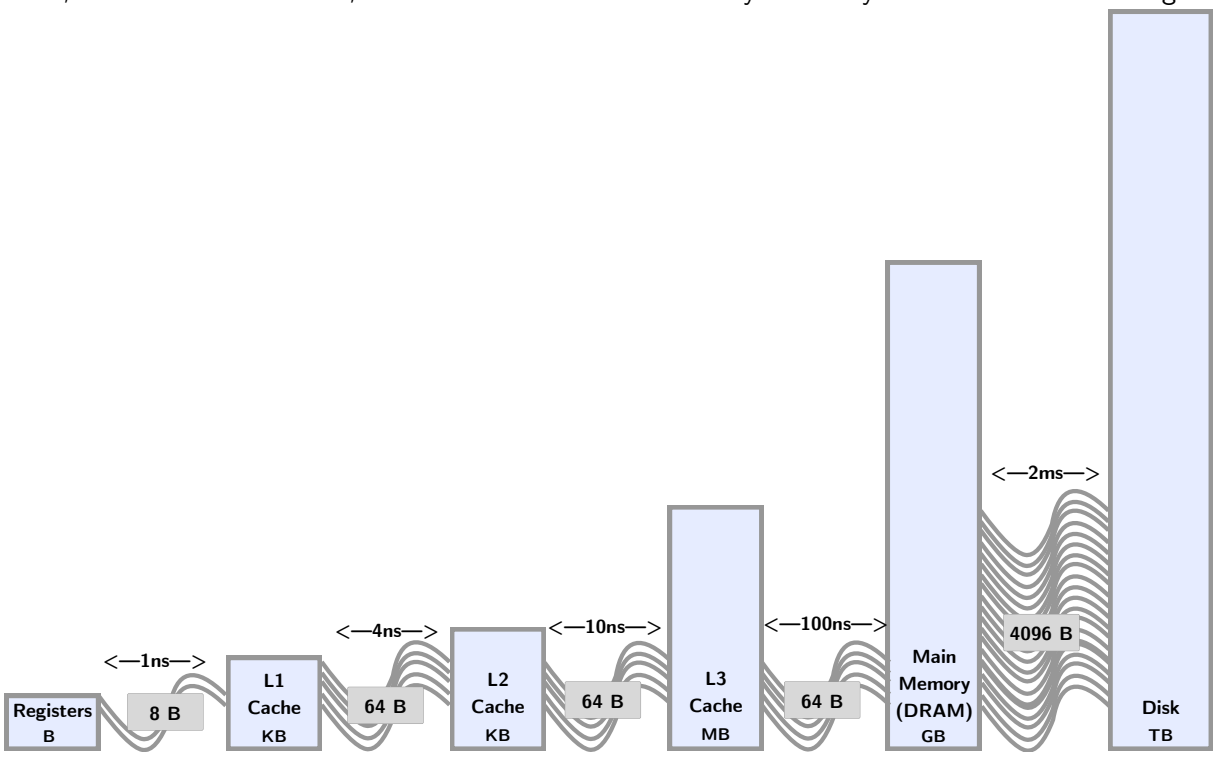
Running this on `compute-cpu` with and without the `flush_cache_line` line of code gives us tthe following (averaged over a few runs):

- With the cache: 66469872 clocks $\approx$ 0.03 seconds
- Without the cache: 343293488 clocks $\approx$ 0.17 seconds

Given that the array size is relatively small, this is a very significant difference, and it only gets worse as the array size increases.

Memory Hierarchy

In practice, there is not one cache, but *three* caches. The "memory hierarchy" looks like the following:



The goal of the memory hierarchy is to provide the *amount of memory at the right side at the speed of the left side*. This may initially seem unlikely to actually happen, but, in fact, most programs are closer to the left than the right in speed! This is possible because most memory accesses in programs exhibit a property called *locality*.

Locality

The main idea is that if a program makes a memory access, it is likely that it will (1) access memory nearby soon, and (2) access that same memory location again soon. These two ideas are called *spatial locality* and *temporal locality*. Consider the following two functions:

```
1  uint64_t pseudorandom_access_pattern(int *array) {
2     uint64_t result = 0;
3     for (size_t i = 0; i < NUM_ELEMENTS; i++) {
4        result += time_read(&array[rand() % NUM_ELEMENTS]);
5     }
6     return result;
7  }
8
9  uint64_t inorder_access_pattern(int *array) {
10    uint64_t result = 0;
11    for (size_t i = 0; i < NUM_ELEMENTS; i++) {
12       result += time_read(&array[i]);
13    }
14    return result;
15 }
```

The first function accesses elements of the array in a random order, and the second accesses them sequentially. In both functions, `result` has temporal locality, because it is accessed every iteration of the loop, but the second function has far more spatial locality than the first. Notably, the only difference between these two functions is the order of elements accessed. When these two functions are run, the in-order one consistently takes at least 20,000 fewer clocks than the random one.