Note 16: Caches

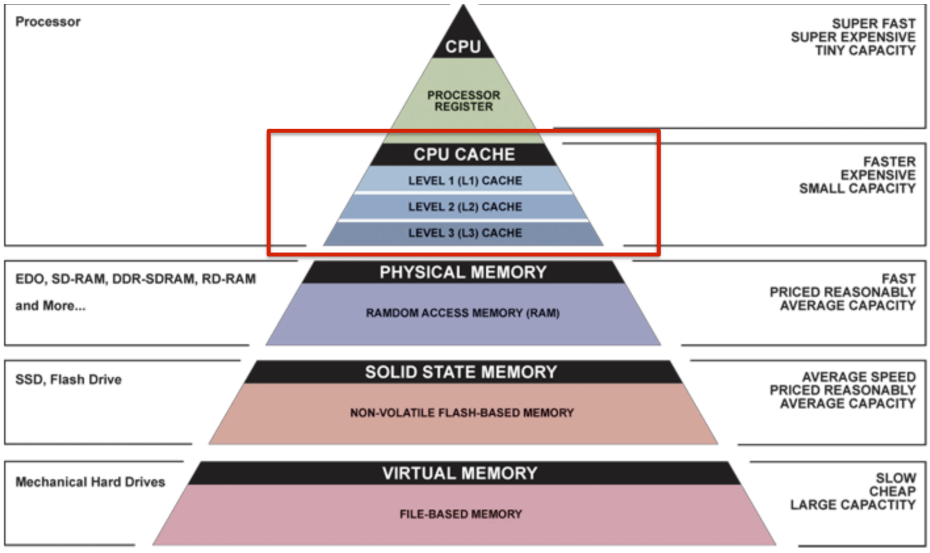
Kevin Moy

Abstract

So as far as memory, we've covered the very top, the CPU, processors, and registers, as well as the bottom- physical memory (RAM). This note, we focus on the medium of communication between the processor and memory: caches. Specifically, we'll be focusing on the principle of locality and building up our memory hierarchy.

1 Caches: An Intro

Here's another diagram of our memory hierarchy:



So far, we've covered this top layer, the processor, which contains data in register files. Such registers are incredibly fast to read from and write to,

on the scale of nanoseconds. However, they are expensive, so we must limit them. At the bottom, we have main memory. This is much larger and much less expensive than processor registers but also much slower.

The number of transistors on a computer chip is exponentially increasing over time- meaning massive gains in CPU speed. However, if the base material (memory) is really *slow*, this kind of negates it out. The gap between performance of memory and processor continues to increase through the years.

Enter the cache. It'll allow us to essentially access all memory but in a very fast way, with only some occasional mishaps. This removes the bottleneck that direct accesses to main memory incur.

1.1 The Library Analogy

A very common analogy to caches is a library. Let's say I'm writing a report in a library in 1950 (when no internet existed). We have a desk, and somewhere nearby, a shelf of books, and all around you, a massive storage of books- the library itself. Now say you need to quote some specific fact (with a citation). You first search the entire library system (main memory) for that book. I find that book, use it, then set it on my desk. Then whenever I need to use something from that book again, it's now next to me, far more convenient than searching the library again. Now, what if I want another? Well, my shelf tends to have more space. So my desk is really serving as the cache here- a space to hold resources from the library (memory) that are recently used and thus probably relevant to what I'm doing. A beautiful system that has direct analogs: the cache is your desk, the processor is your BRAIN, and main memory is the library!

However, if I'm a fucking nerd there will be a point when my desk becomes too cluttered. So to keep my desk, or cache, functional, I need to clear some books- preferably ones I won't need anymore! We'll cover in more detail how this works later on.

This is a simple, beautiful system that has direct analogs: the cache is your desk, the processor is your BRAIN, and main memory is the library! Hopefully with this analogy caches make sense at a high level.

Caches are just part of the **Principle of Locality**, which is the concept that when I make memory accesses, they will most likely be somewhat close to

each other. Think about instruction memory in our CPU datapath. Loops, as well as instructions executed in sequence, mean that code memory is VERY localized! In data memory, it's the same: stack frames try to keep local variables close to each other in that function call. Heap too. Arrays and structs are also just grouping stuff together. Locality is a natural characteristic of programs.

There are two main types of locality: **temporal locality** and **spatial locality**. Temporal locality is the concept that a memory access will probably be done more than once, and probably close together in time as well. It's like if I'm using a book on WW2 to cite one fact, I'll probably use that same book to cite another as well. Spatial locality, on the other hand, states that for a memory access, you'll probably also do accesses for memory close to it (in space). Iterations through arrays are a perfect example of this.

Caches really provide the *illusion* that we can access all of memory without needing to actually make those expensive memory accesses, at least most of the time. Note that levels in our hierarchy are just subsets of lower levels! We only have a set amount of memory for everything. So we increase the probability that we reach what we want as we go down the hierarchy.

Most of the time, the dollar sign (`$`) represents a cache ("cash"). Remember we isolate instruction memory and data memory to avoid structural hazards- accordingly, we also have separate caches (instruction cache and data cache) for them as well. Instruction cache stores recently loaded instructions from IMEM, and data cache stores recently loaded data from DMEM. Modern processors also have their own hierarchy of caches, similar to the memory hierarchy.

With registers, we dealt with one-word-sized chunks of memory. However, we want to have larger chunks as we move down the memory hierarchy. So we call a unit of transfer a **block**- and its size depends on the associated cache. The size of blocks grows as we go further down the memory hierarchy.

The great thing about caches is that it is already controlled by a separate piece hardware, called the memory manager or cache controller, that handles all cache data I/O. We shall study how this cache controller operates next.

Caches use **static RAM** (SRAM), which is very fast, very small, and very expensive. Static means the data held will only last as long as power is on. Main memory, on the other hand, uses **dynamic RAM** (DRAM), which

is much slower, much cheaper, and much larger. It also is dynamic which means it needs to be refreshed regularly to keep the current state updated.

2 Fully Associative Caches

Let's take a look at how to design a good cache. There are some fundamental questions we need to answer before doing so:

- Where/how do we load blocks from memory?
- How can we check if a block is already in cache?
- How can we quickly find a block in the cache?
- Replacement policy?

We'll start answering the first question: loading blocks from memory.

Remember that memory is byte-addressed. For this class, block size will be set to word multiples. To extract individual words or bytes from a block once we've loaded it, we specify the **offset**. Because we want to organize our cache in some way such that we can extract what we want from it, we need to implement **indexing** into our cache as well.

Finally, we must have a way of checking if the block I want is actually in the cache. Remember on a memory access, the only input is some 32-bit address. We need a way to extract information from that address such that our cache can return the correct block of memory. We use a **tag** for each memory block in our cache, specifically for this purpose. This forms a one-to-one mapping between cache memory block and memory address.

The cache, of course, also needs a way to return requested data if it already exists, or fetch and load from memory if it doesn't. It thus also must have a way of identifying its current memory blocks, so it can immediately return a requested block if it exists. Remember that a cache is initially "cold": it contains garbage blocks. We thus also need a **valid bit** set to each block in the cache that indicates if it's garbage or useful.

Ideally, our block size would be k bytes, large enough such that we exploit spatial locality, but also small enough such that multiple blocks can actually

fit in our cache (one block taking up the entire cache would make it kind of pointless).

For our offset field, we use the lowest $\log_2$(block size) bits of the memory address to get the specific offset of our block. The block size thus needs to be a power of 2 bytes.

The rest of the bits- and there are $32 - \log_2$(block size) of these bits- will be used for tag bits. Let's think about this. They will be the SAME for ALL byte addresses in the block, and thus effectively tags that block! It really determines which portion of memory the block came from.

Now let's get to the formal definition of a **fully associative cache**. This is a cache where there isn't really an organization to the blocks placed. Every memory block can map anywhere in the cache, effectively making the cache like a block bucket. This maximizes space efficiency: we just keep filling our cache with blocks until it is full. However, because of the absence of organization, it is now quite inefficient to CHECK: we now have to look at all blocks in sequence. Note that in a FA cache the tag is unique. Thus we have to account for the tag bits and valid bits for each block in our cache when allocating space.

3 Hits, Misses, and Replacement

There are ways now of checking if a cache contains our stuff or not. If the cache holds a valid copy of the block of memory we want, it returns it and we have a **cache hit**.

However, if the cache does not (and thus needs to load from memory into a correct slot) this is a **cache miss**. If the cache is full, we must replace some existing block - specifically which block is determined by a replacement policy. The two most common policies are **random replacement**, which just replaces a random block to replace, and **least recently used** (LRU), which replaces the block used least recently, the idea being that we probably won't use it again or at least will use it later than the other blocks. Ideally, we want to maximize cache hits (and minimize misses). Formally, we want to maximize our **hit rate** (HR), which is calculated by $\frac{\text{cache hits}}{\text{total accesses}}$. Miss rate, naturally is $1 -$ hit rate.

We also want to maximize cache *speed*, which we can measure by hit time (HT): the time to access the correct block in the cache, including the tag comparison after accessing. Upon a miss, we must also add a **miss penalty**: the time to fetch and load a block from memory (as well as time to possibly replace a block in the cache).

4 Write Policies

Caches can have a different version of a block from memory.

Memory addresses are not only for reads- they are writes as well! What happens when we want to write to memory? Do we write to the cache as well?

We separate reading policies and write policies. Instruction cache (and IMEM) is read-only: no reason to write to code memory (huge security vulnerability). We'll take a look, then at our instruction memory cache and data memory cache, denoted `I$` and `D$`. Read hits are the fastest possible scenario for these caches, so we want to maximize these.

Write hits are different, and have different policies.

4.1 Write-Through Policy

The write-through policy states that when you try to write to memory, you check the cache first for that block. If there's a hit, overwrite it in the cache AND write to memory as well. This policy forces cache and memory to always have the same versions of a block of data. However, this is quite slow: we have to scan both the cache AND write to memory which takes a while and can accumulate. So to solve this we have a separate **write buffer** that parallelizes the process of writing to memory. Assume we have a write buffer for this class unless otherwise specified.

4.2 Write-Back Policy

In this policy, upon a write hit we **only write to cache** (not memory). Only when the block is evicted from the cache do we write it to memory. So multiple updates to a block will be reflected in the cache and will only need a single "upload" to memory.

To implement this, we need a **dirty bit** for every cache slot, which indicates that this block was written to (and thus differs from the memory version) and needs to be written back when evicted.

5 Handling Write Misses

What about writing something that's NOT in the cache already? Obviously we need to write to memory, but do we take the extra time to update the cache too? Two more policies once again:

5.1 Write Allocate Policy

The write-allocate policy says to update the cache as well as memory upon a write miss. This policy is almost always paired with a write-back cache (a cache implementing the write-back policy). This favors multiple writes (updates) to the same memory address, since we are now caching it and not using a bunch of accesses to RAM.

5.2 No Write Allocate Policy

On the other hand, the no write allocate policy says to only update memory on a write miss. This policy is almost always paired with a write-through cache (a cache implementing the write-through policy). This favors writes to addresses that are random or infrequent, because we skip the extra store-in-cache that comes with it. Additionally, this is for those that need memory to be up-to-date at all times.