Note 17: Special Caches

Kevin Moy

July 2020

Abstract

Last note provided a high-level introduction to caches. This note will delve into two specific types of caches: direct-mapped and set-associative.

1 Introduction

Remember that accesses to memory from the CPU are in general very expensive. Our solution was a medium called a cache, which made access time, especially for repeated accesses, much faster. It utilized the ideas of temporal and spatial locality in doing so: temporal in that we will likely access the same bit of memory multiple times in a given time period, and spatial in that we will likely access memory close to each other.

Remember that caches also hold a *subset* of overall memory. Remember also the memory hierarchy, and how size,speed,and cost change as you move up and down the hierarchy. Note the structure of a cache, and how a (32-bit) memory address is broken down to create the appropriate fields for a cache.

On a memory access, we first check our cache with these steps:

1. Look at all cache slots in parallel
2. If valid bit 0, ignore (garbage block)
3. If valid bit 1 AND tag bit matches, return block.

Additionally, on a memory write, we want to set the dirty bit to true IF we are using a write-back cache.

Remember how load instructions work with caches (`lb`) for a fully associative cache. We break up our input memory address into tag and offset. We check the cache for our tag, check the valid bit, then the offset tells which byte to actually grab from that block. In terms of hardware, the offset is the selector for a MUX whose inputs are the bytes from each block!

Remember the write-through policy and the write-back policy (with the extra dirty bit needed) for handling write hits.

Remember the policies for handling cache misses: on a read miss, we just stall execution and load the block from memory into the cache, while on a write miss, we always update memory and optionally update the cache, depending on 2 more policies. Specifically, these policies are write allocate (write to cache as well) and no write allocate (don't write cache). We know write-allocate is paired with write-back, which favors keeping our CACHE up-to-date, and no write allocate is paired with write-through, which favors keeping our MEMORY up-to-date.

2 Direct-Mapped Cache

Let's take a look at one of the specific types of caches, a direct-mapped cache.

For a fully associative cache, we can place the block anywhere inside, and implement whatever replacement policy we choose. FA caches maximize efficiency because it uses all the space available. However, there's no order, so that gives a disadvantage when doing searches.

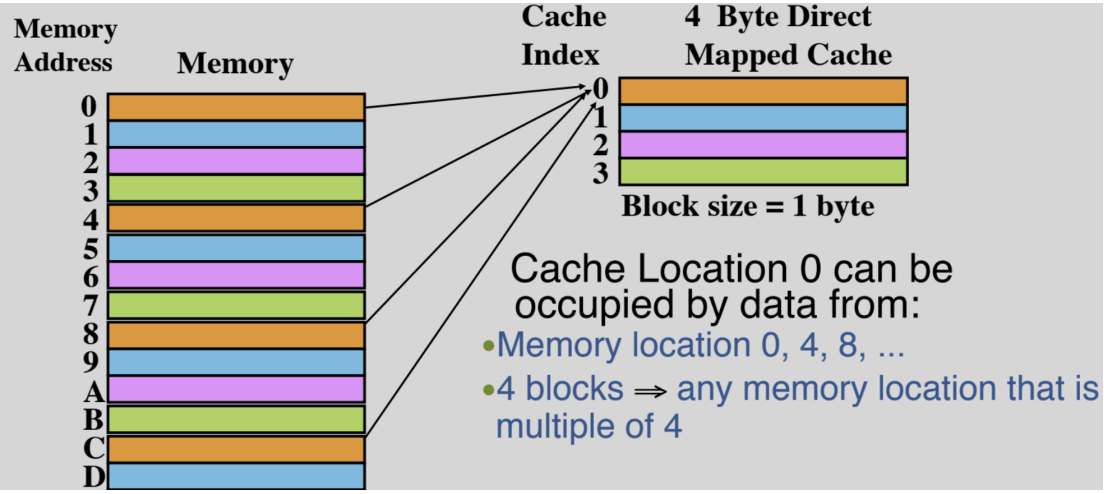
The direct mapped cache solves this by mapping every memory block to a single pre-designated slot in the cache. It finds the correct slot for each block by utilizing a hash function. With this, we are now able to *very* easily find a block in the cache, in fact in $O(1)$ time, via our hash function. It also relieves us of a need for a replacement policy, since every block has a designated spot, and we just overwrite whatever is there.

However, with this fixed slot design, there will be an issue of memory addresses mapping to the same slot in the cache. If we're always doing memory accesses which map to the same slot, we'll just be replacing blocks over and over, when we could just be utilizing the empty slots elsewhere in the cache.

As far as fields derived from a memory address, because of the order we

place, we also need an **index field** that specifies the cache slot index where the block of memory belongs. Then, the tag field will now serve an extra purpose: these distinguish between all the possible memory addresses that map to the same cache slot. But overall, the tag still serves as the unique identifier of a cache block. The offset field stays the same.

Here is a picture of a direct-mapped cache:



So in our DM cache, we have our data block, tag field, valid bit, dirty bit (no replacement policy bits). This gives a total of $2^I * (8 * 2^O + T + 1 + 1)$ bits total, where I=index bits, O=offset bits, T=tag bits. Remember we calculate tag bits AFTER number of index and offset bits.

Note we don't actually need to store the index (or offset) bits in the cache! They are stored implicitly with hardware. Address -¿ index and offset, and hardware takes care of accessing the cache byte.

3 Set-Associative Cache

The next important type of cache is the **set-associative cache**. The set associative cache attempts to compromise between the advantages and disadvantages of FA and DM caches. SA caches are more flexible than DM caches but more structured than FA caches.

For an **n-way set associative cache** we divide the cache into sets, where each set has N slots. Now for a given memory address, the index field

determines the SET, and we just place the block anywhere in that set. Note we still need the tag field to check if that particular block is already in that set or not.

For this kind of cache, we *will* be needing a replacement policy (for every set). When the set is full, we need to find the least-recently used slot in that set to evict the block there.

The TIO breakdown remains the same:

- Offset field still represents number of bytes in a block.
- Index now represents a SET, so $I = \log_2(\frac{C}{K*N})$
- Tag field is still A-I-O.

Notice that a fully associative cache is just an associative cache with one massive set, which needs 0 index bits! Direct mapped caches, on the other hand, are just 1-way associative caches!! It's an amazing generalization.

For a fixed-size cache, each increase **by a factor of two** in N (associativity) doubles the number of blocks per set, and halves the number of sets per cache. Think about it this way: if we decrease the index bits and increase the tag bits, we increase associativity by allowing more blocks per set (more tag bits) but less sets.

Thus the total size of a set-associative cache is the number of sets * associativity.