Computer Architecture III Study Guide Cache Memories

Maximillien Courchesne-Mackie (mcour010@uottawa.ca)

I. Memory Hierarchies

Modern computer systems rely on a memory hierarchy based on speed, size and cost of memory units. The focus of this chapter is on cache memories; small memory units which attempt to bridge the speed gap within a computer system. Caches are fast memory buffers which dynamically load memory locations accessed by the processor.

There are usually multiple cache levels within a system. Typically, the *L1* cache sits nearest to the processor and holds instructions and data for the CPU. Next in line are the *L2* and *L3* caches. All caches are SRAM-type memory units.

II. Block Placement

A cache consists of two main parts: directory memory and data memory. The directory memory contains the *tag* of the memory block currently in each cache line, plus some *state bits*. The data memory contains the cached copy of the memory block. The mapping between memory and cache is based on the block address. The physical memory address is split into two fields: the memory block address and the offset. The value of these depends on how the blocks are organized within the cache.

A. Directly-Mapped Caches

In directly-mapped caches, a certain block in memory always ends up in the same cache line. This cache line is determined by hashing the block address. Directly-mapped caches provide very fast *cache hits*, however, due to the large amount of blocks mapping to the same line, these caches have a higher *miss* rate.

In a directly-mapped cache, the least significant bits are used as the offset (which byte to access), the next bits are used as an index to find which cache block to look in. The rest of the bits are used a comparative tag to ensure the data found is the data wanted.

$$|---------|---|---|$$

$$tag \qquad index \;\; offset$$

B. Set Associative Caches

Set associative caches are slower to hit than directly-mapped caches, but yield less *cache misses*. These caches are separated into sets of lines. Instead of mapping directly to a cache line, a memory block is directly mapped to a set, but can reside in any line within that set.

In set-associative caches, the least significant bits are used for an index and an offset. The index points to where in cache the block is located, while the offset is to find the specific byte wanted. The remainder of the byte address is called the tag and is used to compare with the data in the cache. This is to make sure the data at the cache location we are accessing is the data we are actually looking for. When the proper cache location is found, comparators compare the tags of each set to determine if the data is found.

$$|---------|---|---|$$

$$tag \qquad index \;\; offset$$

C. Fully-Associative Caches

Fully-associative caches have a *CAM* (instead of a RAM) directory which contains all the tags of addresses in the cache. The downside to this method is that a CAM architecture is slower than RAM due to the increased amount of comparators at each directory line. The hit rate for fully-associative caches is very good.

In fully-associative caches, the full address is stored with the data in cache. Because of this, when a cache read is requested, the control unit must compare the memory address with all addresses in the cache to see if the data is cached.

$$|---------|---|$$

$$tag \qquad offset$$

III. Block Replacement

When a memory block is accessed but is not int he cache, a *cache miss* is triggered. The memory block must be brought into the cache, meaning that a currently cached block must be replaced. In a directly-mapped cache, deciding which line to replace is simple as each memory block can only be stored in one specific line. In a set-associative cache, all blocks within the matched set are candidates for removal. In fully associative caches, all blocks in the cache are candidates. The procedure for figuring out which block to replace is called a *replacement policy*.

A. Random Replacement

The easiest method to find a replacement candidate for our new memory block is random. This is the easiest to implement, but has several shortcomings. For example, we could inadvertently replace a block that is used during the next instruction - causing another cache miss and another replacement.

B. Least Recently Used (LRU)

A better method would be to replace the block that was least recently used. The priority of each cache block must be tracked by the cache as to be able to find the candidate for the LRU method. Each line is assigned priority bits which indicate the order in which they were last executed. For example, in the case of a four line cache, each line would have two priority bits. The most recently used cache line would have priority 00, whereas the last recently used would have priority 11.

In the event of a cache hit, the line which was requested will be updated to have priority 00, and all lines that had priority smaller than it will be incremented. For example, if the second line had priority 01, it would now have priority 10.

IV. Write Strategies

When data is modified by the processor, it must be written back to the cache and the lower memory units. The process of writing and queuing this data for write is called a *write policy*.

A. Write-Through Cache

In a write-through cache, all stores are updated in all lower-level memory units. The store buffer keeps pending stores and when the bus to the next level is free they will be propagated. The processor must stall when the store buffer is full to allow the pipeline to be freed. If there is a load miss, the store buffer must first be checked to ensure that data for that address is not pending a write.

B. Write-Back Cache

In a write-back cache, writing is only done to the cache and a *dirty bit* is set to signal that the data has been modified. Writing to the lower caches won't occur until that block has been chosen as a replacement candidate.

V. Cache Performance

Cache performance is based around the average miss rate, MR. The MR is the ratio of cache misses and the number of processor references. In extension, the hit rate (HR) is equal to $1 - MR$.

The average memory access time ($AMAT$) can be calculated using the following formula:

$$AMAT = (HR)HP + (MR)MP$$

Where HP and MP are the hit penalty and miss penalty respectively.

Additionally, the CPI of the system can be calculated as follows:

$$CPI = CPI_0 + (MPI)MP$$

Where CPI_0 is the CPI when all caches hit all the time and MPI is the number of misses per instruction.

VI. Advanced Cache Topics

A. Prefetching

Cache prefetching is the act of grabbing extra blocks in memory and bringing them to the cache before they are requested. Hardware prefetching has prefetch engines which monitor the buses between memory levels and uses intelligent prediction algorithms to send cache misses to the lower levels of cache automatically. Software prefetching is done by the compiler which sets prefetch instructions in the binary at compile time.

B. Non-Blocking Caches

A blocking cache is one which can only accept one processor request at a time. A non-blocking cache is one which can handle multiple requests at a time. Remember that a cache has two controllers, one to handle processor requests and one to service misses at the next cache level. MSHRs (miss status handling registers) accumulate pending misses for the second controller to service.