CSE 351 Lecture 18 – Memory & Caches III

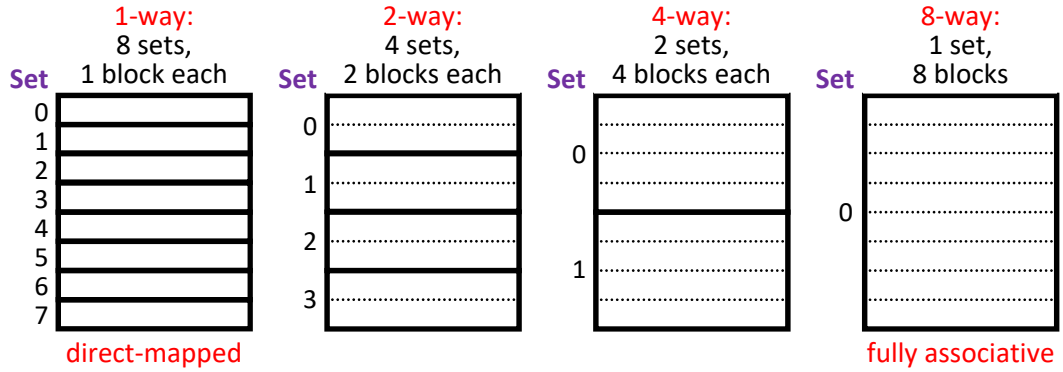
Cache State

Caches are hardware, meaning that there is *always* data present, be it actual program data or garbage. In order to tell the difference, we include a ***valid bit***. This is part of the ***management bits*** (along with the tag bits) stored in the cache for *each* block. We call the combination of the cache block and management bits a ***cache line***. A cache with no valid program data is called a ***cold cache*** (*i.e.* "empty").

Associativity and Replacement

The fact that a direct-mapped cache always maps blocks to a single index introduces a problem – what happens if we alternate between different blocks that map to the same set? The blocks keep evicting each other and we lose the benefits of temporal locality! The problem here is the rigidity of the hash function: each block can only go in one specific index in the cache.

To help with this issue, we introduce the notion of ***associativity***. In a *set associative cache*, we allow each block to fit into a specified ***set*** of locations, which should reduce the number of replacements. An ***n-way set associative cache*** uses sets of size n, meaning a block can be placed in the set n different "ways." A direct-mapped cache is a 1-way set associative cache and, on the other end of the spectrum, a ***fully-associative cache*** is a C/K-way set associative cache (*i.e.* the whole cache is a single set).

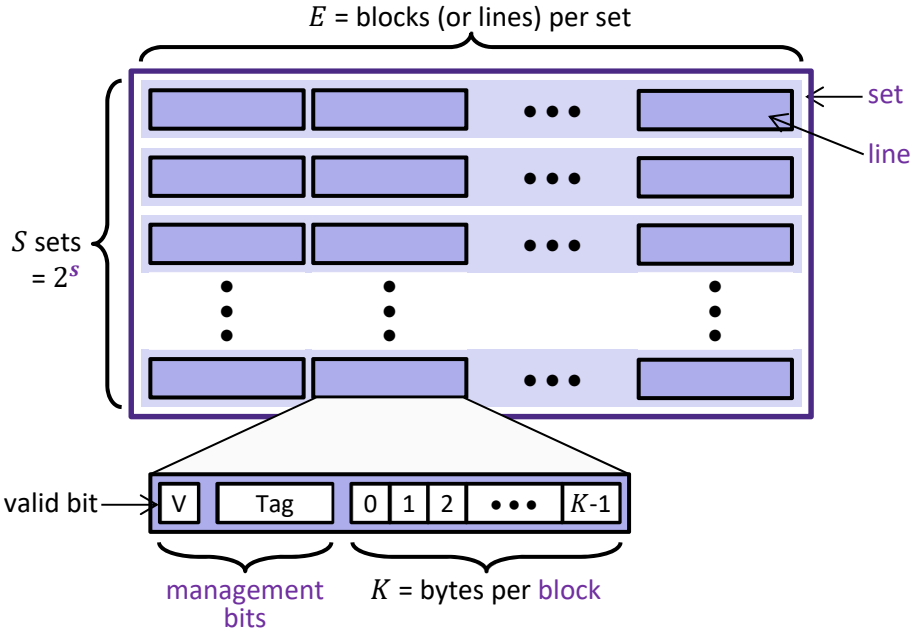


On an access, the TIO address breakdown will still tell us which *set* the block should be found. The number of sets is $\boldsymbol{S = (C/K)/E}$, which is the number of blocks the cache can hold divided by the associativity. Notice that the associativity changes where the TIO address breakdown splits the tag and index fields since $s = \log_2 S$ and $t = m - s - k$ still. But now there might be multiple lines within the set to look, so we will need to check all of their tag fields simultaneously, increasing the data access time with increasing associativity.

On a cache miss, there now might be multiple locations within the designated set that we can place our block of interest. If any of the lines in the set are invalid/empty, then we can place our block there. If the set if full, then a variety of replacement policies could be used, but we will mostly assume that we will replace the ***least recently used (LRU)*** block in order to maximize our temporal locality.

General Cache Organization & Notation Review

The geometry of a cache can be completely described with just the number of sets (S), the associativity (E), and the block size (K). In fact, the cache size is the product of these terms ($C = S \times E \times K$).



Parameter	Var
Block size	K
Cache size	C
Associativity	E
Number of Sets	S
Address width	m
Tag field width	t
Index field width	s
Offset field width	k

Cache Misses

We've seen that reducing our miss rate can greatly affect our code performance (*e.g.* average memory access time). Understanding the cause of misses can sometimes help us identify inefficiencies in our code. We generally categorize cache misses into the following:

- ***Compulsory misses***: the 1st time accessing a block is guaranteed to miss (also called a *cold miss*).
- ***Conflict misses***: if more references map to the same set than our associativity allow to coexist, this causes misses when we revisit the earlier references.
- ***Capacity misses***: the total amount of data we are using exceeds the size of the cache, so we simply don't have the capacity to keep all of the blocks in the cache at once and the evicted blocks cause these misses to occur.

The different cache parameters have different effects on each type of miss:

- Larger block size reduces compulsory misses because more data is brought into the cache with each miss.
- Higher associativity reduces conflict misses by allowing more blocks to coexist in the same set.
- Larger cache size (or reducing our working set of data) helps reduce capacity misses.

A combination of carefully chosen caching system parameters and code writing can help us optimize the memory performance for our specific application.