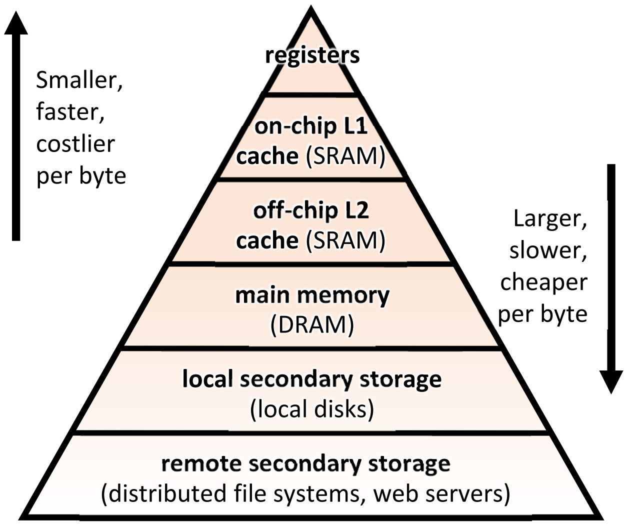
CSE 351 Lecture 17 – Memory & Caches II

The Memory Hierarchy

Caches are one tier of a larger classification of the main forms of computer storage known as the ***memory hierarchy***. This includes both local data storage (*e.g.* registers and RAM) and remote/external data storage (*e.g.* web servers). Each level of the hierarchy can be thought of conceptually as a "cache" of the level below it – a faster way to access a subset of the available data from the level below. Faster storage technologies almost always cost more per byte and therefore are used in lower capacities, which matches the design of the memory hierarchy.


Smaller, faster, costlier per byte
registers
on-chip L1 cache (SRAM)
off-chip L2 cache (SRAM)
main memory (DRAM)
local secondary storage (local disks)
remote secondary storage (distributed file systems, web servers)
Larger, slower, cheaper per byte

When programs exhibit good locality, we gain benefits from both ends of the hierarchy: we have access to the storage capacity of the lower levels at an average access time closer to those of the higher levels.

Cache Terminology (Incomplete)

Last lecture, we introduced the notion of a *block*, which is a machine-specific fixed unit of transfer between a cache and the storage level below. ***Block size*** ($\boldsymbol{K}$) refers to how many bytes there are in each block, which is always a power of 2.

Note: The textbook uses B for block size instead of K, but we want to avoid confusion with the standard shorthand for bytes (*e.g.* K = 32 B means the block size is 32 bytes).

The ***cache size*** ($\boldsymbol{C}$) measures the capacity of the cache and is defined to be the amount of *program data* that the cache can store – we'll see that additional management information is stored to ensure proper cache operation but is not counted in the cache size. Cache size is always a multiple of block size, so is sometimes given in terms of bytes (C) or in terms of blocks (C/K).

We will introduce more cache terminology in the next lecture, so this is currently an incomplete picture.

Direct-Mapped Cache Placement

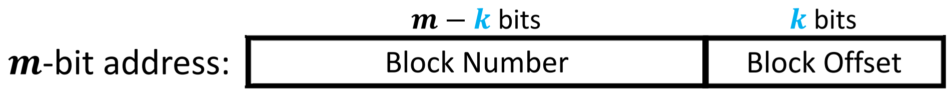
Because we can only hold a subset of the data from the storage level below, we must decide where to place blocks on a cache miss, which will also affect where we look for blocks on a cache access. Ideally, we want to make this decision fast, in order to reduce our average memory access time by reducing our hit time (looking during a cache access) and miss penalty (placing during a cache miss).

The first *placement strategy* that we will look at is known as ***direct-mapped***, where we use a simple hash function of modulus by the number of blocks the cache can hold (*i.e.* mod (C/K)) to determine where in the cache to place that specific block, giving us a very fast algorithm that also utilizes every spot in our cache. Notice that this is a deterministic placement – the cache access (*i.e.* the requested address) will always hash to the same spot. This means that we don't need a special *replacement policy* – we always kick out the existing block in that spot.

> **Note:** For **modulus** and bits, note that `x%(2^n)` returns the value of the lowest `n` bits of `x`. This is a very important fact for the understanding of the mechanics of memory accesses.

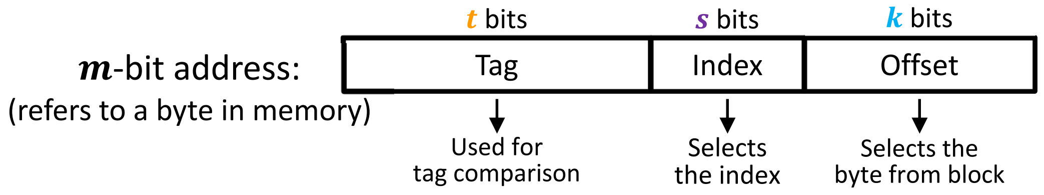
Direct-Mapped Memory Address Breakdown

So how do we perform a direct-mapped cache access? The access takes the form of a memory address (a word-sized number that we now define to be $\boldsymbol{m}$ bits wide). We know that blocks contain K bytes of contiguous memory that don't overlap (*i.e.* no two blocks contain the data from the same address). Each block contains the data for K addresses, so we need $\boldsymbol{k = \log_2 K}$ bits to represent each byte *within* a block (this is equivalent to the address modulo K). So, the upper $m - k$ bits tell us which ***block number*** our request lives in and the lower k bits tells us the ***block offset*** – where within the block our address lies:



Since we transfer data into the cache in blocks (*i.e.* all bytes with the same block number different block offset move together), our placement policy maps our block *number* to a specific place in the cache, which we will call a ***cache index***. Since our cache holds C/K blocks, there are $\boldsymbol{S = C/K}$ indices in a direct-mapped cache, meaning we need $\boldsymbol{s = \log_2 S} = \log_2(C/K)$ bits of our address to represent each possible index. More specifically, since we're mapping block number to index, our direct-mapped hashing function is (block number) mod S, meaning we use the lowest s bits of our block number to determine which index to place that block.

Finally, since multiple blocks map into the same cache index, we use the rest of the address as a ***cache tag***, an identifier to uniquely indicate which cache block is stored. Note that the tag bits are actually stored as part of the management information in the cache. So, we can determine the placement of a block from reading specific fields of the access address:



Example: If we had 8-bit addresses (m = 8), 4-byte blocks, and a cache size of 32 bytes, that means that
K = 4 bytes, k = 2 bits, C = 32 B, $S = C/K$ = 8 indices, s = 3 bits, and t = 8 - 3 - 2 = 3 bits.
A request for the address 0xAA checks index 2 for block number 42 because 0xAA = 0b101|010|10.