CS 251 — Chapter 7: Memory Hierarchy

Bartosz Antczak *Instructor: Stephen Mann* *March 2, 2017*

7.1 Introduction

Key Questions

- What is memory hierarchy?
- Why do we have them?

Fast Memory

When we say that memory is *fast*, we say that the time required to access it is relatively fast. For instance, from figures in 2012:

- SRAM typically only requires 0.5 - 2.5 nanoseconds to access memory (costs $500 - $1,000)
- A Magnetic Disk requires 5,000,000 - 7,000,000 nanoseconds (costs around 5 cents)

Obviously, the faster the memory, the more expensive it becomes.
The **goal** of memory hierarchies is to create an illusion of unlimited fast memory. This is done by having several types and sizes of memory, each of varying speed. We move items to smaller, faster memory automatically when they are needed, and we keep them there if they are used often. The general hierarchy is structured (from the fastest being at the top):

- Registers
- *L1, L2, L3 caches* (which are small amounts of memory on the processor
- RAM
- Disk
- Local network
- The cloud / off site storage

7.1.1 Implementation

The items not in the top level are brought up when requested. They're not copied directly to the front, however. The data is only copied between adjacent levels (this is identical to the transpose heuristic outlined in CS 240).
Starting at the top of the hierarchy, we look for the data in each section of memory. If the data is not there, move down one level. If the data is found, retrieve it and swap this memory with the memory in the upper level.
This hierarchy sees memory as a **single array**.

7.2 Implementing a Cache

Key Questions

- How is a cache organized?
- What are the basic challenges with caching?

Recall that cache is a small amount of memory in the processor. We have an L1, L2, and L3 cache:

- **L1:** 32 KB instruction, 32 KB data per core
- **L2:** 256 KB per core
- **L3:** 2-4 MB shared with all cores

7.2.1 Approach 1: Direct Mapped Cache

This approach requires that each memory location is mapped to exactly one location in the cache.

Storing and Looking for Data

Assume M blocks of cache memory, where each block has size B. When given a request for address p, translate it into a request for cache block m (if cache entry is valid). The typical mapping for a cache block is:

$$m = (p/B) \mod M$$

Example 7.2.1. *An example mapping for a cache*

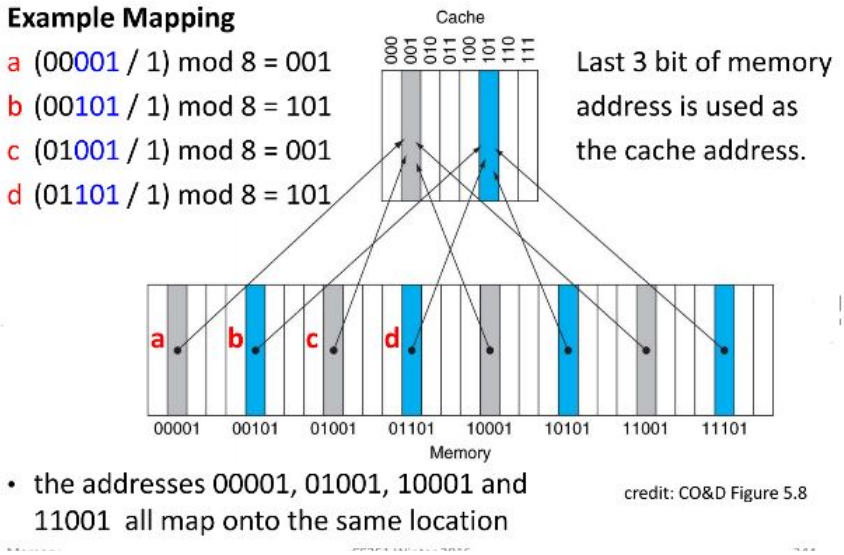

Figure 7.1: *Courtesy of Kevin Lanctot's Winter 2016 slides*

Tags

Many memory blocks map to the same cache block. Adding **tags** will identify which memory block is actually present. For example (referring to the previous example), the tag would be the upper 2 bits. If the cache location 101 (called the index) contains the value of memory location 00**101**, then its tag will be 00 (the first two bits).

Valid Bits

We must track if the cache entry is valid. We do this by adding a valid bit.

Example 7.2.2. *Storing and looking for data in a cache*

Cache

Index	Valid	Tag	Data
000	N		
001	N		
010	Y	11	Mem[11010]
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

We perform these following instructions on the cache:

Action	Word Address	Binary Address	Hit/Miss
Load Word	22	10 **110**	Miss
Load Word	26	11 **010**	Miss
Load Word	22	10 **110**	Hit (Does not alter cache)
Load Word	26	11 **010**	Hit (Does not alter cache)

Now what if we loaded the following:

Action	Word Address	Binary Address	Hit/Miss
Load Word	18	10 **010**	Miss

Here, we *overwrite* whatever was in index 010 (which was Mem[*11*010]) with Mem[*10*010].

What Happens on a Cache Miss?

We stall the entire processor until the item has been fetched.