EE 156: Advanced Topics in Computer Architecture

Spring 2022
Tufts University

Instructor: Prof. Mark Hempstead
mark@ece.tufts.edu

Lecture 2: Memory Hierarchy and Review of Direct Mapped and Set-Associative Caches

Unit 1: The Memory Hierarchy

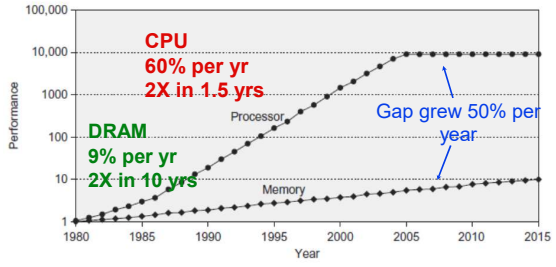
Unit 1: The Memory Hierarchy (3-4 weeks)

- **Introduction and Performance Metrics [Chapter 1]**
- **Review of Basic Caches and Set Associativity [Appendix B]**
- Advanced Cache Optimization Techniques and Replacement policies [Appendix B, Ch 2]
- Prefetching [SLCA: Falsafi and Wenisch]
- Memory consistency and Cache coherence [Chapter 5]
- Software interfaces and memory consistency
- Transactional memory
- Review of Virtual Memory and TLBs [Appendix B]
- Advanced Virtual Memory [SLCA: Bhattacharjee and Lustig]
- New Non-Volatile Memory (NVM) technologies

- Textbook Reading: Appendix B

Since 1980, CPU has outpaced DRAM

Q. How do architects address this gap?
A. Put smaller, faster "cache" memories



Memory Hierarchy: Dell with Core i7 (Nehalem)

Managed by compiler

Managed by hardware

Managed by OS, hardware, application

	Reg	L1 Inst	L1 Data	L2	L3	DRAM	Disk (SSD)	Disk (HD)
Size	1.25K	32K	32K	256K	8M	4G	120G	1 TB
Latency Cycles, Time	1, 0.4 ns	4, 1.6 ns	3, 1.6 ns	11, 4.4 ns	39, 15.6 ns	107, 42.8 ns	$5*10^5$, 200us	$8.7*10^6$, 3.2 ms

Goal: Illusion of large, fast, cheap memory

Let programs address a memory space that scales to the disk size, at a speed that is usually as fast as register access

The Principle of Locality

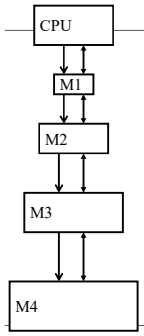
- Two Different Types of Locality:
  - Temporal Locality (Locality in Time): If an item is referenced, it will tend to be referenced again soon (e.g., loops, reuse)
  - Spatial Locality (Locality in Space): If an item is referenced, items whose addresses are close by tend to be referenced soon (e.g., straight-line code, array access)
- Locality is a property of nearly all programs that is exploited in machine design.
  - Programs that don't actually have locality won't benefit from caches. Example: copy one column from one matrix to another.

Locality Example

```
j=val1;
k=val2;
For (i=0; i<10000;i++) {
  A[i] += j;
  B[i] += k;}
```

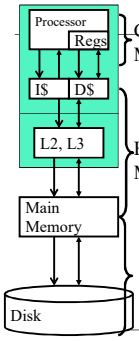
- What data locality is here?: i,A,B,j,k?
- What instruction locality? The inner loop.

Exploiting Locality: Memory Hierarchy



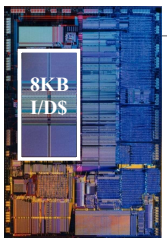
- Hierarchy of memory components
  - Upper components
    - Fast ↔ Small ↔ Expensive
  - Lower components
    - Slow ↔ Big ↔ Cheap
- Connected by "buses"
  - Which also have latency and bandwidth issues
- Most frequently accessed data in M1
  - M1 + next most frequently accessed in M2, etc.
  - Move data up-down hierarchy
- Optimize average access time
  - $latency_{avg} = latency_{hit} + \%_{miss} * latency_{miss}$
  - Attack each component

Concrete Memory Hierarchy

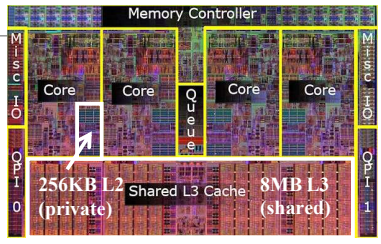


- 0th level: **Registers**
- 1st level: **Primary caches**
  - Split instruction (I$) and data (D$)
  - Typically 8KB to 64KB each
- 2nd level: **2nd and 3rd cache** (L2, L3)
  - On-chip, typically made of SRAM
  - 2nd level typically ~256KB to 512KB
  - "Last level cache" typically 4MB to 16MB
- 3rd level: **main memory**
  - Made of DRAM ("Dynamic" RAM)
  - Typically 1GB to 4GB for desktops/laptops
    - Servers can have 100s of GB
- 4th level: **disk (swap and files)**
  - Uses magnetic disks or flash drives
  - 10s of GB or over 1 TB

Evolution of Cache Hierarchies



Intel 486

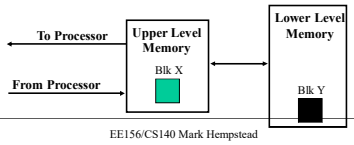


Intel Core i7 (quad core)

- Chips today are 30–70% cache by area

Memory Hierarchy: Terminology

- For any given memory level…
- Hit: data appears in some block at this level (Block X below)
  - Hit Rate: the fraction of memory access found at this level
  - Hit Time: Time to access this level; consists of
    RAM access time + Time to determine hit/miss
- Miss: data needs to be retrieve from a lower level (Block Y below)
  - Miss Rate = 1 - (Hit Rate)
  - Miss Penalty: Additional time required on a miss
- Hit Time << Miss Penalty (500 instructions on 21264!)



More terminology

- One cache line = the amount of data that the cache returns to the next level above it.
- Also called a cache block.
- Also called a cache frame.
- We will use all three terms, interchangeably, so you're ready for the various people you will meet who will use any of them ☺.

Cache tradeoffs

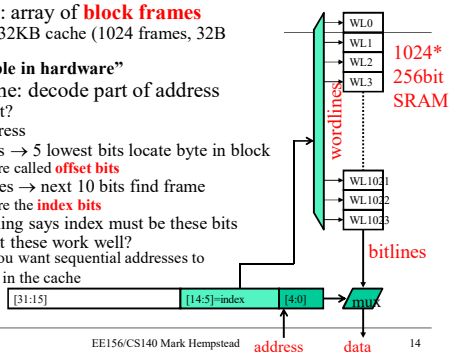
- Caches have LOTS of transistors; the potential for wasting power is correspondingly huge.
- Caches take up lots of area, which implies long wires, and thus unavoidable latencies.
- If we try to improve bandwidth by using wider buses or more pipelining, both will (as usual) cost power.
  - More details to come.

4 Questions for Memory Hierarchy at any level

- Q1: Where can a block be placed in the upper level? *(Block placement)*
- Q2: How is a block found if it is in the upper level? *(Block identification)*
- Q3: Which block should be replaced on a miss? *(Block replacement)*
- Q4: What happens on a write? *(Write strategy)*

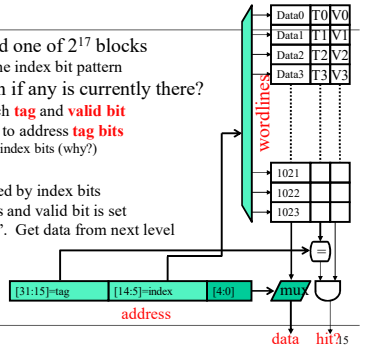
Q1: Finding Data via Indexing

- Basic cache: array of **block frames**
  - Example: 32KB cache (1024 frames, 32B blocks)
  - **"Hash table in hardware"**
- To find frame: decode part of address
  - Which part?
  - 32-bit address
  - 32B blocks → 5 lowest bits locate byte in block
    - These are called **offset bits**
  - 1024 frames → next 10 bits find frame
    - These are the **index bits**
  - Note: nothing says index must be these bits
  - Why might these work well?
    Because you want sequential addresses to spread out in the cache



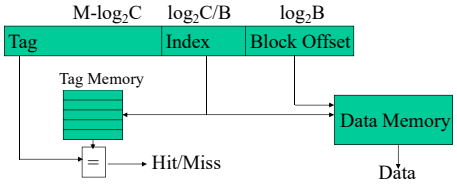
Q2: Knowing that You Found It: Tags

- Each frame can hold one of 2^{17} blocks
  - All blocks with same index bit pattern
- How to know which if any is currently there?
  - To each frame attach **tag** and **valid bit**
  - Compare frame tag to address **tag bits**
    - No need to match index bits (why?)
- Lookup algorithm
  - Read frame indicated by index bits
  - "Hit" if tag matches and valid bit is set
  - Otherwise, a "miss". Get data from next level



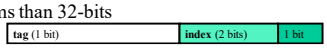
Address arithmetic

- Partition Memory Address into three regions
  - C = Cache Size
  - M = Numbers of bits in memory address
  - B = Block Size



Cache Examples

- 4-bit addresses → 16B memory
  - Simpler cache diagrams than 32-bits

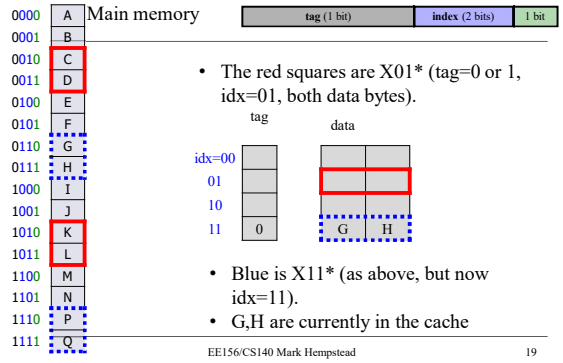


- 8B cache, 2B blocks
  - How many blocks? (8B/cache)*(1 block/2B) = 4 blocks/cache.
  - Figure out how address splits into offset/index/tag bits
    - Offset: least-significant $\log_2$(block-size) = $\log_2(2) = 1 \rightarrow 0000$
    - Index: next $\log_2$(number-of-blocks) = $\log_2(4) = 2 \rightarrow 0000$
    - Tag: rest = 4 – 1 – 2 = 1 → 0000

Larger Cache Example

- 32-bit machine
- 4KB, 16B Blocks, direct-mapped cache
  - 16B Blocks => 4 Offset Bits
  - 4KB / 16B Blocks => 256 Frames
  - 256 Frames / 1 –way (DM) => 256 Sets => 8 index bits
  - 32-bit address – 4 offset bits – 8 index bits => 20 tag bits

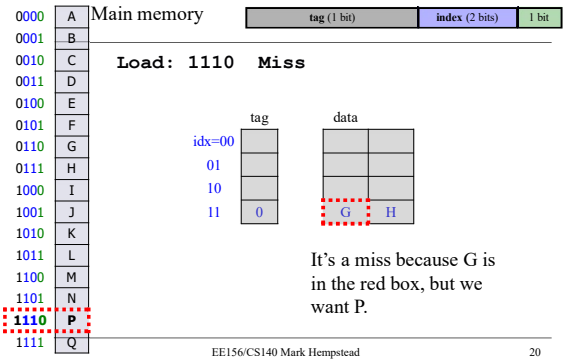
4-bit Address, 8B Cache, 2B Blocks



- The red squares are X01* (tag=0 or 1, idx=01, both data bytes).

- Blue is X11* (as above, but now idx=11).
- G,H are currently in the cache

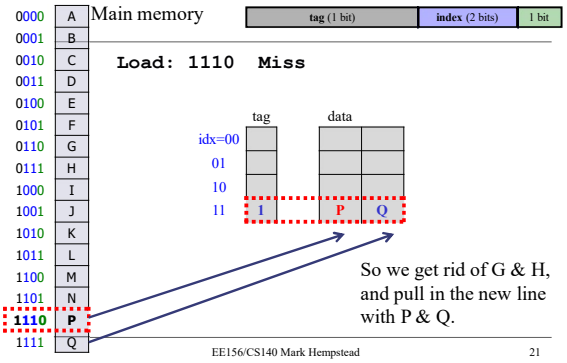
4-bit Address, 8B Cache, 2B Blocks



Load: 1110 Miss

It's a miss because G is in the red box, but we want P.

4-bit Address, 8B Cache, 2B Blocks



Load: 1110 Miss

So we get rid of G & H, and pull in the new line with P & Q.

Direct Mapped Cache

- Consider the main memory word reference string

Start with an empty cache - all blocks initially marked as not valid

0 1 2 3 4 3 4 15

- 8 requests, 6 misses

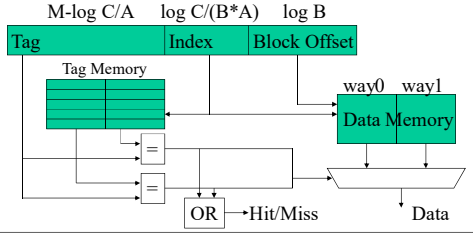
ANSWER

The problem with direct mapping

- The caches we've been looking at are "direct mapped;" each line can go in exactly one spot in the cache.
  - If the code wants to keep using (in our previous example) both 0110 and 1110, then we will keep getting misses.
  - For (int i=0; i<10; ++i) { ++mem[0110]; ++mem[1110] }
- Solution: let each line have a choice of multiple places to reside in the cache.
- Now the index points to a *set* of several cache lines. Each line of the set has its own tag; at most 1 tag can match any line.

Set Associative Caches

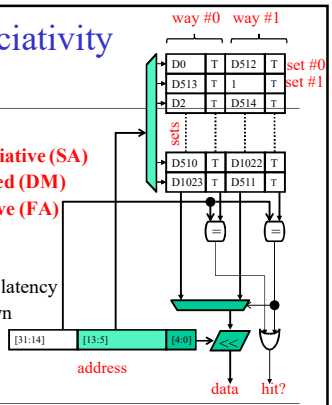
- Partition Memory Address into three regions
  - C = Cache Size, B=Block Size, A=number of members per set



Set-Associativity

- **Set-associativity**
  - This is **2-way set-associative (SA)**
  - 1-way → **direct-mapped (DM)**
  - 1-set → **fully-associative (FA)**

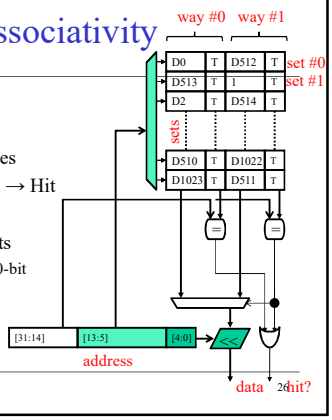
  \+ Reduces conflicts
  – New OR, mux increase latency
  – Note: valid bit not shown



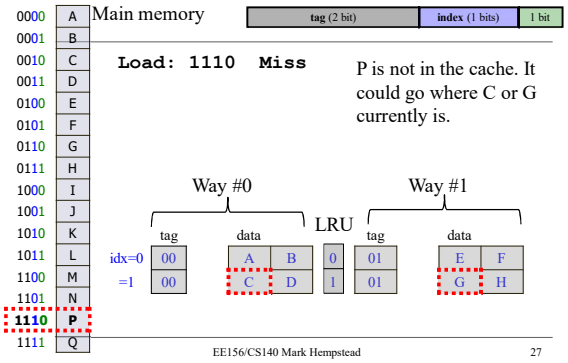
Set-Associativity

- Lookup algorithm
  - Use index bits to find set
  - Read data/tags in all frames
  - Any (match and valid bit) → Hit

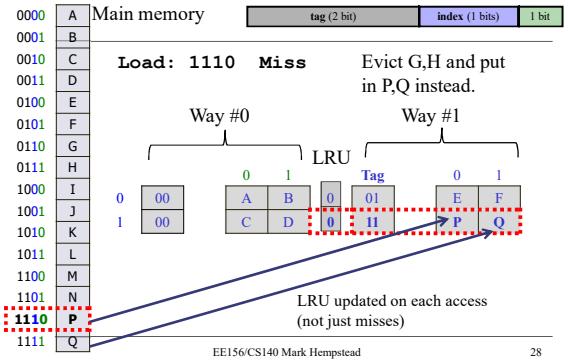
  - Notice tag/index/offset bits
    - Only 9-bit index (versus 10-bit for direct mapped)



4-bit Address, 8B Cache, 2B Blocks, 2-way



4-bit Address, 8B Cache, 2B Blocks, 2-way

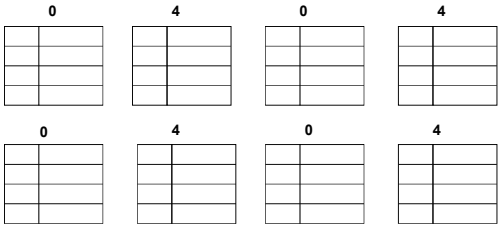


Another Reference String Mapping

- Consider the main memory word reference string

Start with an empty cache - all blocks initially marked as not valid
Assume: 4 lines; each is one word

0 4 0 4 0 4 0 4 (word addresses)



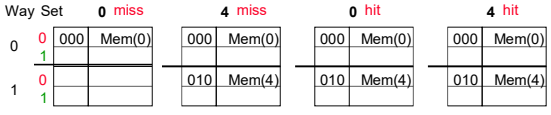
HANDOUT

Another Reference String Mapping

- Consider the main memory word reference string

Start with an empty cache - all blocks initially marked as not valid
Each line is one word

0 4 0 4 0 4 0 4 (word addresses)



- 8 requests, 2 misses

- Solves the ping pong effect in a direct mapped cache due to conflict misses since now two memory locations that map into the same cache set can co-exist!

ANSWER

Replacement Policies
"Q3: Which block should be replaced on a miss"

- Set-associative caches present a new design choice
  - On cache miss, which block in set to replace (kick out)?
- There are two hard things in computer science: cache invalidation, naming things, and off-by-one errors.
  - Unknown author.

Replacement Policies
"Q3: Which block should be replaced on a miss"

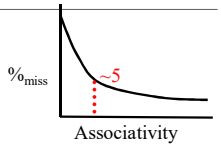
- Some options
  - **Random**
  - **FIFO (first-in first-out)**
  - **LRU (least recently used)**
    - Fits with temporal locality, LRU = least likely to be used in future
  - **NMRU (not most recently used)**
    - An easier to implement approximation of LRU
    - Is LRU for 2-way set-associative caches
  - **Belady's**: replace block that will be used furthest in future
    - Unachievable optimum
  - Sampling/Prediction Approaches
    - e.g. Re-Reference Interval Prediction (**RRIP**) (We will read this paper)

Eviction

- When a line is evicted, it may have to get updated in the next-lower-level cache
  - But only if some piece of data in the line has been modified.
- Caches keep a "dirty" bit for each line.
  - Starts at 0. Goes to 1 when the line is modified.
  - When the line is eventually evicted, it gets written back if it's dirty.
- Does an ICache need dirty bits?
  - Only for self-modifying code. Even then, usually the code goes into the DCache when it's written.

Associativity and Performance

- Higher associative caches
  - + Have better (lower) $\%_{miss}$
    - Diminishing returns
  - – However t_{hit} increases
    - The more associative, the slower
  - – What about t_{avg}?



Another Example – 3 ways

- 32-bit addresses
- 96KB, 32B block, 3-way Set Associative
- Compute Total Size of Tag Array
  - How many blocks in the cache?96KB/32B blocks => 3K
  - How many total sets? (3K Blocks)*(1 set/3 blocks)=1K sets
  - How many offset bits? 32B Blocks => 5 Offset Bits
  - How many index bits? 1K Sets => 10 index bits
  - How many tag bits? 32-bit address – 5 offset bits – 10 index bits = 17 tag bits per block.
    17 bits/block * 3K blocks = 51Kb total

Summary of Set Associativity

- Set Associative Caches
  - N-way set associativity → N tag comparators, one N:1 mux.
  - Any line can be in any of N places
- Direct Mapped
  - One place in cache, one Comparator, no muxes
- Fully Associative (for L lines in the entire cache)
  - Anywhere in cache
  - Number of comparators = L
  - L:1 mux needed

Preview: Cache Replacement Policies

- Modern caches are > 16 ways
- Commercial Processors do not use LRU
  - Managing Least Recently Used stacks is expensive and complex
  - Tree based approaches approximate LRU
  - Newer predictive approaches take the workload into account