

308-273

Caches, Part II

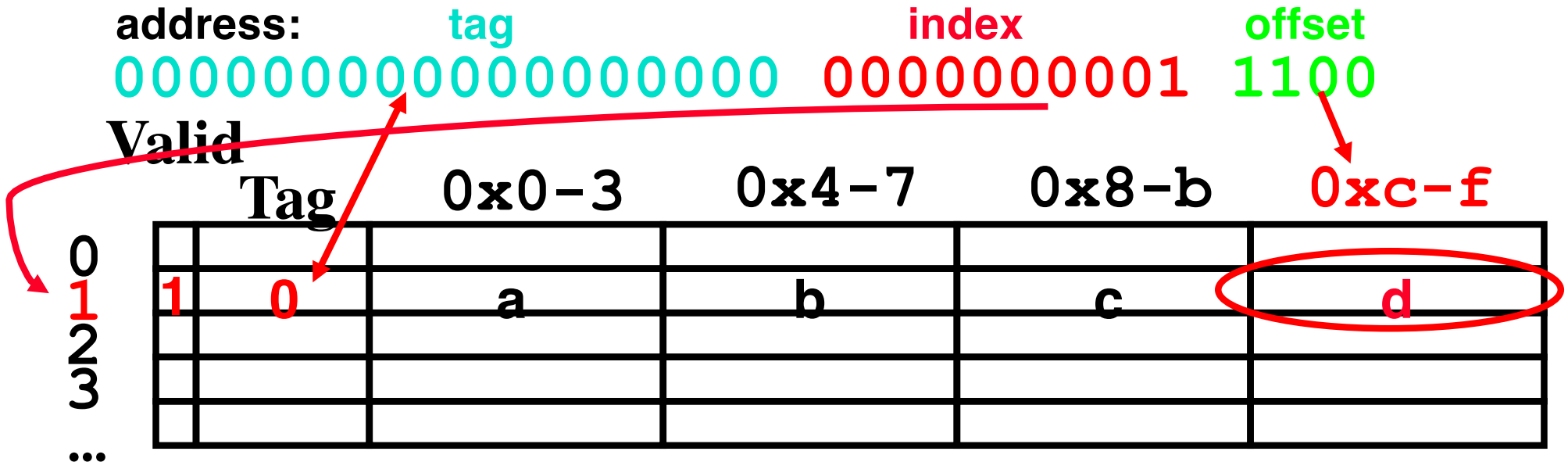
Kaleem Siddiqi

Review

- We would like to have the capacity of disk at the speed of the processor: unfortunately this is not feasible.
- So we create a memory hierarchy:
 - each successively lower level contains “most used” data from next lower level
 - exploits temporal locality
 - do the common case fast, worry less about the exceptions (design principle of MIPS)
- Locality of reference is a Big Idea

Big Idea Review (1/2)

- **Mechanism for transparent movement of data among levels of a storage hierarchy**
 - **set of address/value bindings**
 - **address => index to set of candidates**
 - **compare desired address with tag**
 - **service hit or miss**
 - **load new block and binding on miss**



Outline

- **Block Size Tradeoff**
- **Types of Cache Misses**
- **Fully Associative Cache**
- **N-Way Associative Cache**
- **Block Replacement Policy**
- **Multilevel Caches (if time)**
- **Cache write policy (if time)**

Block Size Tradeoff (1/3)

◦ Benefits of Larger Block Size

- **Spatial Locality**: if we access a given word, we're likely to access other nearby words soon (Another Big Idea)
- Very applicable with Stored-Program Concept: if we execute a given instruction, it's likely that we'll execute the next few as well
- Works nicely in sequential array accesses too

Block Size Tradeoff (2/3)

◦ Drawbacks of Larger Block Size

- Larger block size means **larger miss penalty**
 - on a miss, takes longer time to load a new block from next level
- If block size is too big relative to cache size, then there are too few blocks
 - Result: miss rate goes up

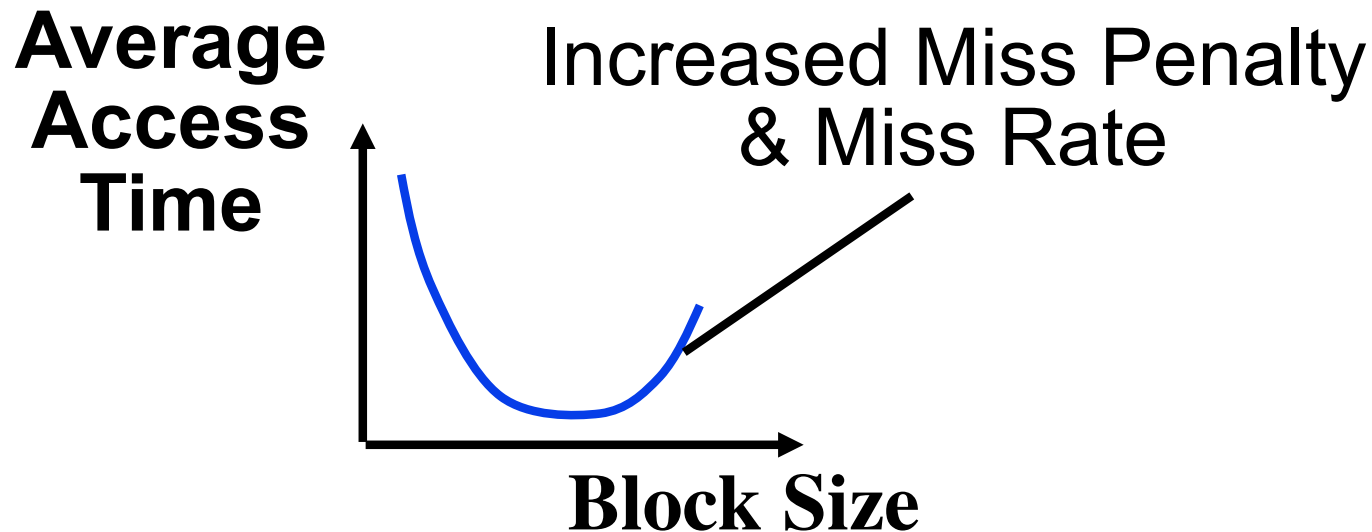
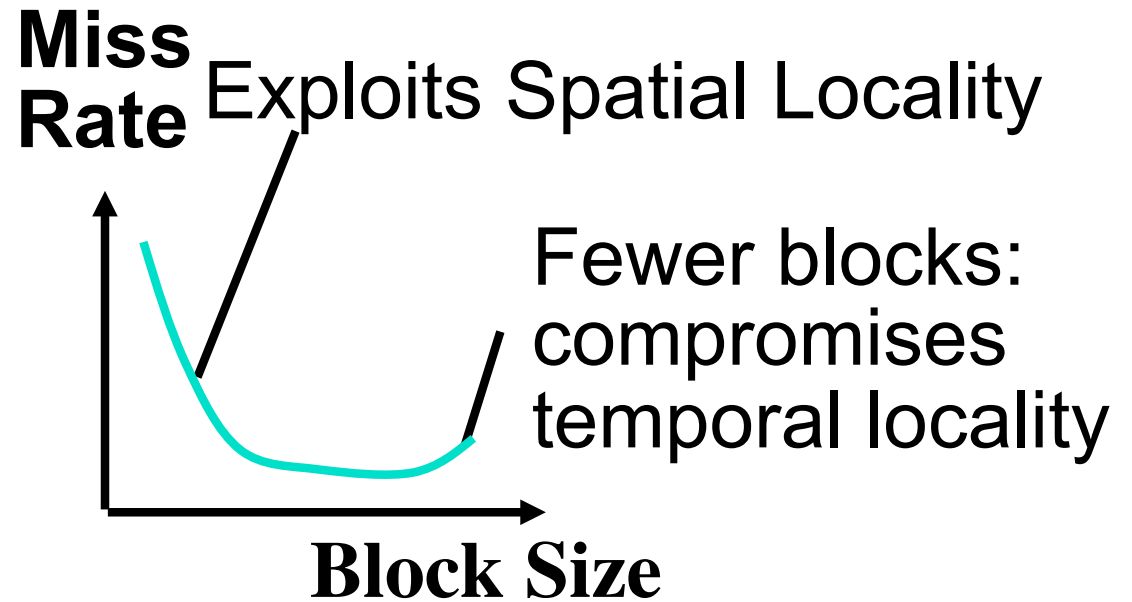
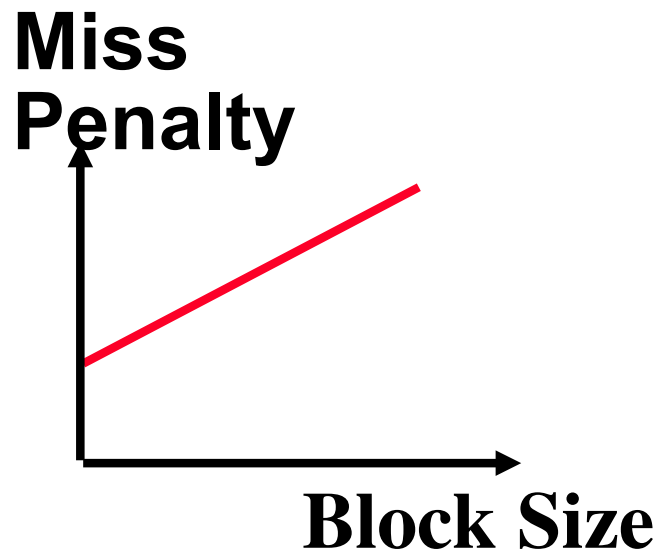
◦ In general, minimize **Average Access Time**

$$= \text{Hit Time} + \text{Miss Penalty} \times \text{Miss Rate}$$

Block Size Tradeoff (3/3)

- **Hit Time** = time to find and retrieve data from current level cache
- **Miss Penalty** = average time to retrieve data on a current level miss (includes the possibility of misses on successive levels of memory hierarchy)
- **Hit Rate** = % of requests that are found in current level cache
- **Miss Rate** = $1 - \text{Hit Rate}$

Block Size Tradeoff Conclusions



Types of Cache Misses (1/2)

◦ Compulsory Misses

- occur when a program is first started
- cache does not contain any of that program's data yet, so misses are bound to occur
- can't be avoided easily, so won't focus on these in this course

Types of Cache Misses (2/2)

◦ Conflict Misses

- **miss that occurs because two distinct memory addresses map to the same cache location**
- **two blocks (which happen to map to the same location) can keep overwriting each other**
- **big problem in direct-mapped caches**
- **how do we lessen the effect of these?**

Dealing with Conflict Misses

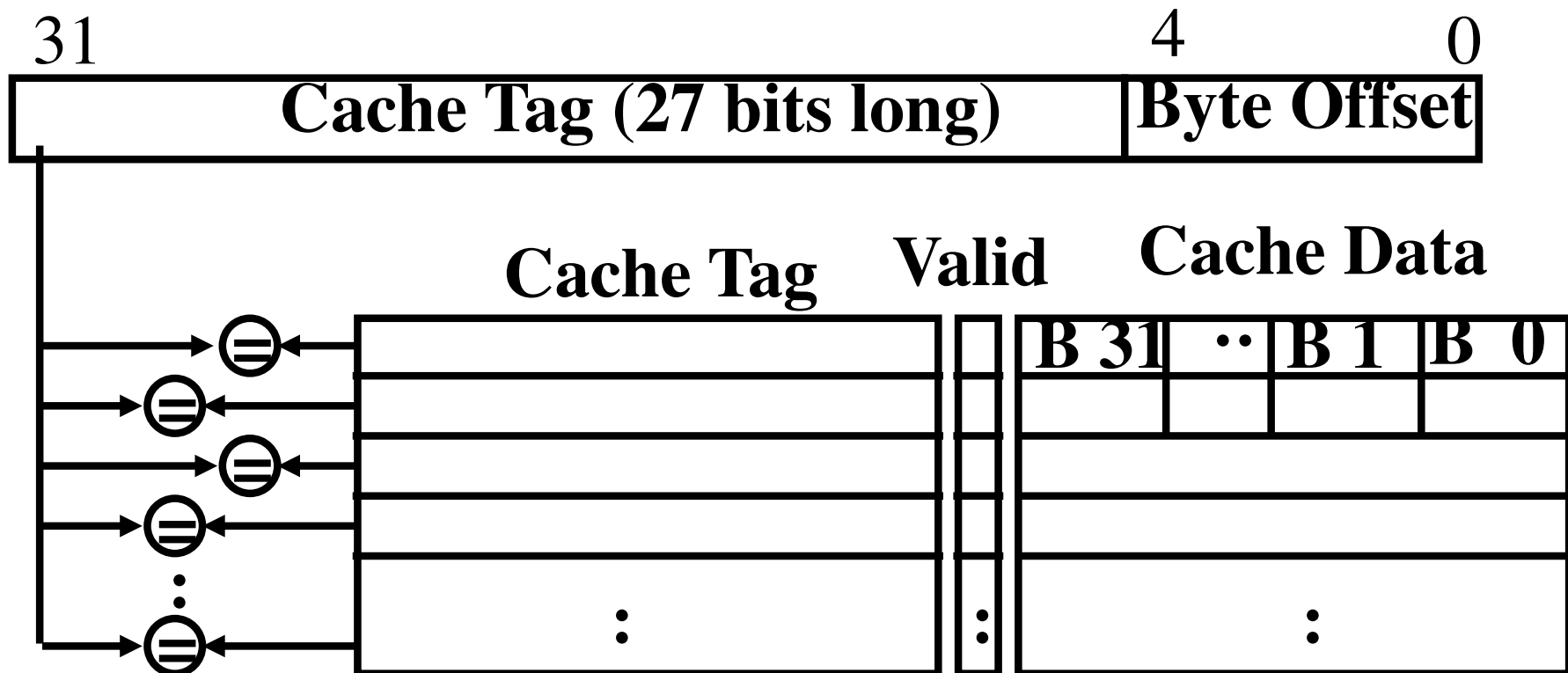
- **Solution 1: Make the cache size bigger**
 - relatively expensive
- **Solution 2: Multiple distinct blocks can fit in the same Cache Index?**

Fully Associative Cache (1/3)

- **Memory address fields:**
 - **Tag: same as before**
 - **Offset: same as before**
 - **Index: non-existent**
- **What does this mean?**
 - **any block can go anywhere in the cache**
 - **must compare with all tags in entire cache to see if data is there**

Fully Associative Cache (2/3)

- ° Fully Associative Cache (e.g., 32 B block)
 - compare tags in parallel



Fully Associative Cache (3/3)

◦ Benefit of Fully Assoc Cache

- no Conflict Misses (since data can go anywhere)

◦ Drawbacks of Fully Assoc Cache

- need hardware comparator for every single entry: if we have a 64KB of data in cache with 4B entries, we need 16K comparators: very expensive

◦ Small fully associative cache may be feasible

Third Type of Cache Miss

◦ Capacity Misses

- miss that occurs because the cache has a limited size
- miss that would not occur if we increase the size of the cache

◦ This is the primary type of miss for Fully Associate caches.

N-Way Set Associative Cache (1/4)

- **Memory address fields:**
 - **Tag: same as before**
 - **Offset: same as before**
 - **Index: points us to the correct “row” (called a set in this case)**
- **So what's the difference?**
 - **each set contains multiple blocks**
 - **once we've found correct set, must compare with all tags in that set to find our data**

N-Way Set Associative Cache (2/4)

◦ Summary:

- cache is direct-mapped with respect to sets**
- each set is fully associative**
- basically several direct-mapped caches, each of which is fully associative. Each has its own valid bit and data**

N-Way Set Associative Cache (3/4)

- **Given memory address:**
 - **Find correct set using Index value.**
 - **Compare Tag with all Tag values in the determined set.**
 - **If a match occurs, it's a hit, otherwise a miss.**
 - **Finally, use the offset field as usual to find the desired data within the desired block.**

N-Way Set Associative Cache (4/4)

- **What's so great about this?**
 - **even a 2-way set assoc cache avoids a lot of conflict misses**
 - **hardware cost isn't that bad: only need N comparators**
- **In fact, for a cache with M blocks,**
 - **it's Direct-Mapped if it's 1-way set assoc (1 block per set)**
 - **it's Fully Assoc if it's M-way set assoc (M blocks per set)**
 - **so these two are just special cases of the more general set associative design**

Block Replacement Policy (1/2)

- **Direct-Mapped Cache:** index completely specifies which position a block can go in on a miss
- **N-Way Set Assoc ($N > 1$):** index specifies a set, but block can occupy any position within the set on a miss
- **Fully Associative:** block can be written into any position (there is no index)
- **Question:** if we have the choice, where should we write an incoming block?

Block Replacement Policy (2/2)

- **Solution!**
- **If there are any locations with valid bit off (empty), then usually write the new block into the first one.**
- **If all possible locations already have a valid block, we must use a replacement policy by which we determine which block gets “cached out” on a miss.**

Block Replacement Policy: LRU

◦ LRU (Least Recently Used)

- Idea: cache out block which has been accessed (read or write) least recently
- Pro: temporal locality => recent past use implies likely future use: in fact, this is a very effective policy
- Con: with 2-way set assoc, easy to keep track (one LRU bit); with 4-way or greater, requires complicated hardware and much time to keep track of this

Block Replacement Example

- We have a 2-way set associative cache with a four word *total* capacity and one word blocks. We perform the following word accesses (ignore bytes for this problem):

0, 2, 0, 1, 4, 0, 2, 3, 5, 4

How many hits and how many misses will there be for the LRU block replacement policy?

Hint: treat addresses as TAG + INDEX

Block Replacement Example: LRU

- Addresses 0, 2, 0, 1, 4, 0, ...
 - 0: miss, bring into set 0 (loc 0)
 - 2: miss, bring into set 0 (loc 1)
 - 0: hit
 - 1: miss, bring into set 1 (loc 0)
 - 4: miss, bring into set 0 (loc 1, replace 2)
 - 0: hit

	loc 0	loc 1
set 0	0	lru
set 1		
set 0	lru 0	2
set 1		
set 0	0	lru 2
set 1		
set 0	0	lru 2
set 1	1	lru
set 0	lru 0	4
set 1	1	lru
set 0	0	lru 4
set 1	1	lru

Ways to reduce miss rate

- **Larger cache**
 - limited by cost and technology
 - hit time of first level cache $<$ cycle time
- **More places in the cache to put each block of memory - associativity**
 - **fully-associative**
 - any block any line
 - **k-way set associated**
 - k places for each block
 - direct map: $k=1$

Big Idea

- How do we choose between options of associativity, block size, replacement policy?
- Design against a performance model
 - Minimize: *Average Access Time*
$$= \text{Hit Time} + \text{Miss Penalty} \times \text{Miss Rate}$$
 - influenced by technology and program behavior

Example

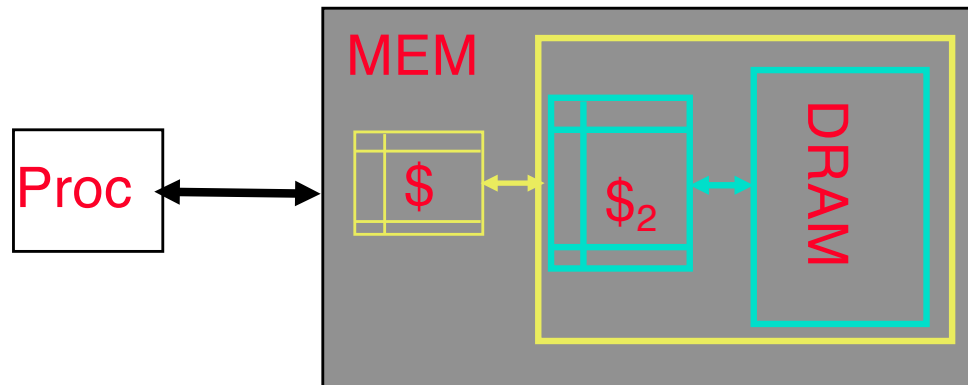
◦ Assume

- Hit Time = 1 cycle
- Miss rate = 5%
- Miss penalty = 20 cycles

◦ Avg mem access time = $1 + 0.05 \times 20$ = 2 cycle

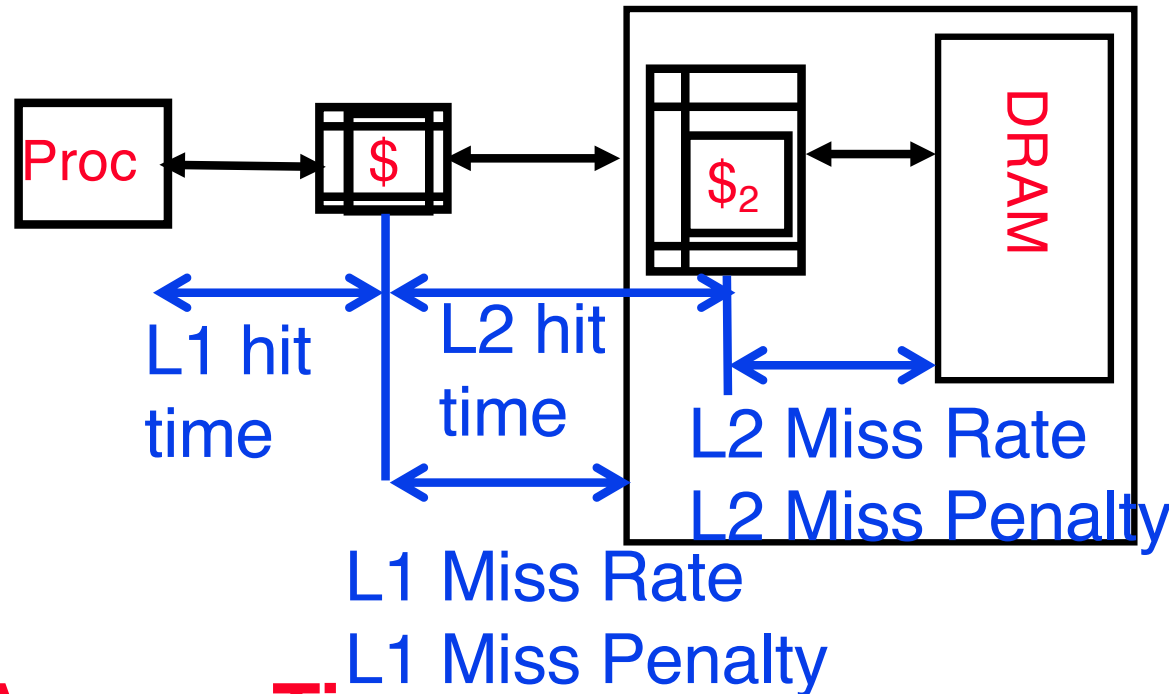
Improving Miss Penalty

- When caches first became popular, Miss Penalty ~ 10 processor clock cycles
- Today 1000 MHz Processor (1 ns per clock cycle) and 100 ns to go to DRAM $\Rightarrow 100$ processor clock cycles!



Solution: another cache between memory and the processor cache: Second Level (L2) Cache

Analyzing Multi-level cache hierarchy



Avg Mem Access Time =

L1 Hit Time + L1 Miss Rate * L1 Miss Penalty

L1 Miss Penalty = L2 Hit Time + L2 Miss Rate

*** L2 Miss Penalty**

Avg Mem Access Time =

**L1 Hit Time + L1 Miss Rate * (L2 Hit Time +
L2 Miss Rate * L2 Miss Penalty)**

Typical Scale

◦ L1

- size: tens of KB
- hit time: complete in one clock cycle
- miss rates: 1-5%

◦ L2:

- size: hundreds of KB
- hit time: few clock cycles
- miss rates: 10-20%

◦ L2 miss rate is fraction of L1 misses that also miss in L2

- why so high?

Example: without L2 cache

- **Assume**

- **L1 Hit Time = 1 cycle**
- **L1 Miss rate = 5%**
- **L1 Miss Penalty = 100 cycles**

- **Avg mem access time = $1 + 0.05 \times 100$
= 6 cycles**

Example with L2 cache

- **Assume**

- **L1 Hit Time = 1 cycle**
- **L1 Miss rate = 5%**
- **L2 Hit Time = 5 cycles**
- **L2 Miss rate = 15% (% L1 misses that miss)**
- **L2 Miss Penalty = 100 cycles**

- **L1 miss penalty = $5 + 0.15 * 100 = 20$**

- **Avg mem access time = $1 + 0.05 * 20$
= 2 cycle**

- **3x faster with L2 cache**

What to do on a write hit?

◦ Write-through

- update the word in cache block and corresponding word in memory

◦ Write-back

- update word in cache block
- allow memory word to be “stale”

=> add ‘dirty’ bit to each line indicating that memory needs to be updated when block is replaced

=> OS flushes cache before I/O !!!

◦ Performance trade-offs?

“And in conclusion...” (1/2)

- Caches are NOT mandatory:
 - Processor performs arithmetic
 - Memory stores data
 - Caches simply make data transfers go faster
- Each level of memory hierarchy is just a subset of next higher level
- Caches speed up due to **temporal locality**: store data used recently
- Block size > 1 word speeds up due to **spatial locality**: store words adjacent to the ones used recently

“And in conclusion...” (2/2)

- **Cache design choices:**
 - **size of cache: speed v. capacity**
 - **direct-mapped v. associative**
 - **for N-way set assoc: choice of N**
 - **block replacement policy**
 - **2nd level cache?**
 - **Write through v. write back?**
- **Use performance model to pick between choices, depending on programs, technology, budget, ...**