C335
Computer Structures

Memory Hierarchies (III)

Dr. Liqiang Zhang

Department of Computer and Information Sciences

Adapted from Morgan Kaufmann and others

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
  - 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)

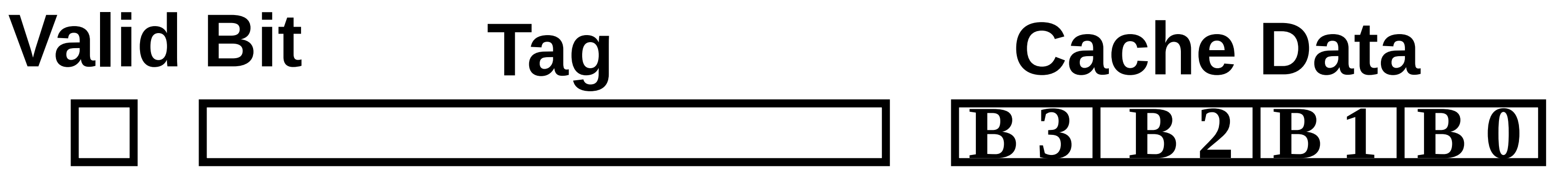
- Drawbacks of Larger Block Size
  - Larger block size means larger miss penalty
    - on a miss, takes longer time to load a new block from next level
  - If block size is too big relative to cache size, then there are too few blocks
    - Result: miss rate goes up
- In general, we want to minimize Average Memory Access Time (AMAT)

  = Hit Time x Hit Rate + Miss Penalty x Miss Rate

Block Size Tradeoff (3/3)

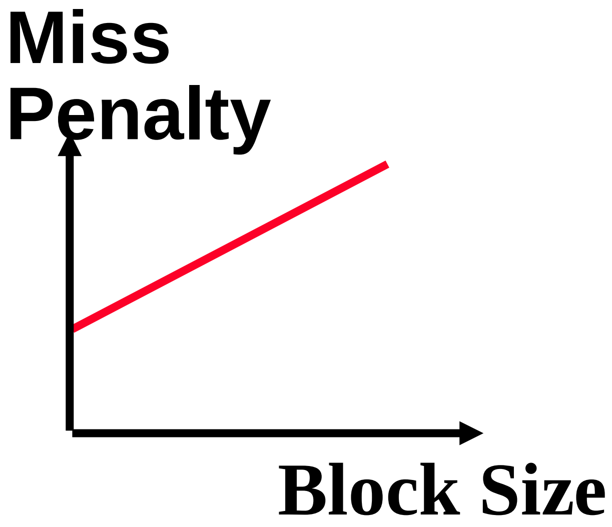
- Hit Rate = % of requests that are found in current level cache
- Miss Rate = 1 - Hit Rate
- Hit Time = time to find and retrieve data from current level cache
- Miss Penalty = average time to retrieve data on a current level miss (includes the possibility of misses on successive levels of memory hierarchy)

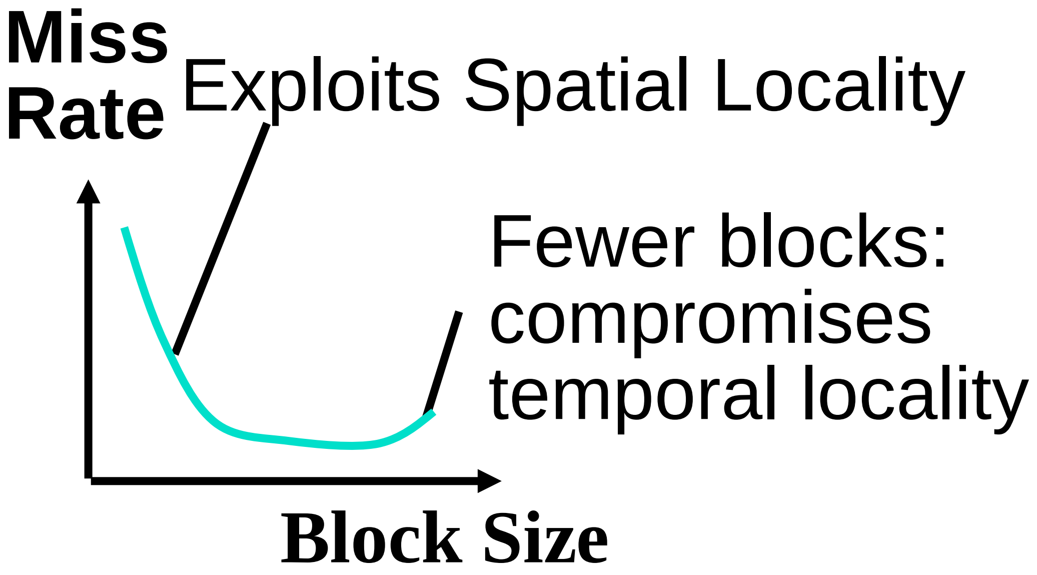
Extreme Example: One Big Block

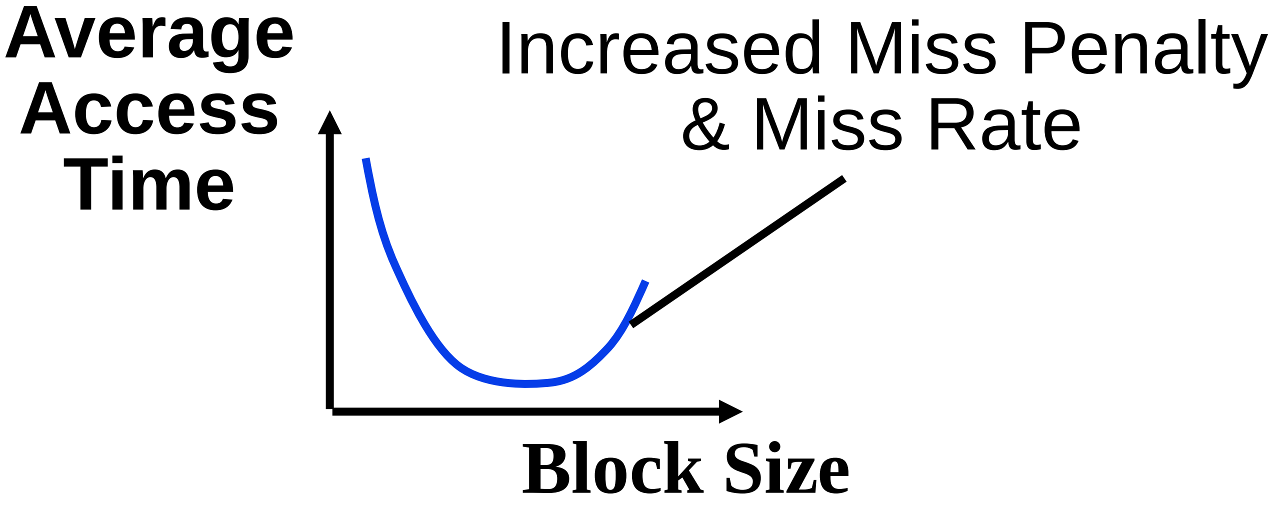


- Cache Size = 4 bytes   Block Size = 4 bytes
  - Only ONE entry (row) in the cache!
- If item accessed, likely accessed again soon
  - But unlikely will be accessed again immediately!
- The next access will likely to be a miss again
  - Continually loading data into the cache but discard data (force out) before use it again
  - Nightmare for cache designer: Ping Pong Effect

Block Size Tradeoff Conclusions







A Summary on Sources of Cache Misses

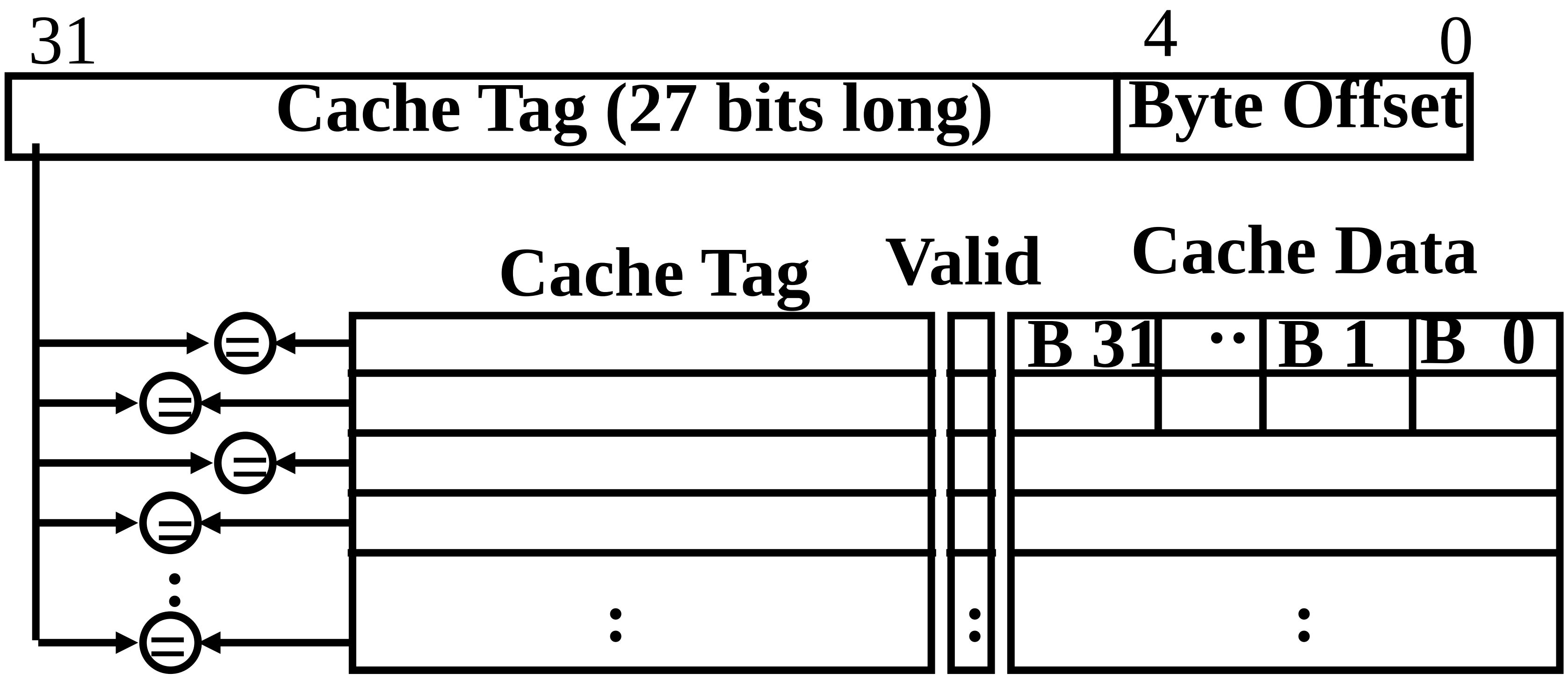
- Compulsory (cold start, first reference): first access to a block
  - “Cold” fact of life: not a whole lot you can do about it, although larger block sizes can reduce the number of misses.
- Conflict (collision):
  - Multiple memory locations mapped to the same cache location
    - Solution 1: increase cache size (index longer)
    - Solution 2: increase associativity
- Capacity:
  - Cache cannot contain all blocks accessed by the program
    - Solution: increase cache size

Fully Associative Cache (1/3)

- Memory address fields:
  - Tag: same as before
  - Offset: same as before
  - Index: nonexistent!
- What does this mean?
  - no “rows”: 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



The whole cache could be taken as one single row (the index only has one value – 0)

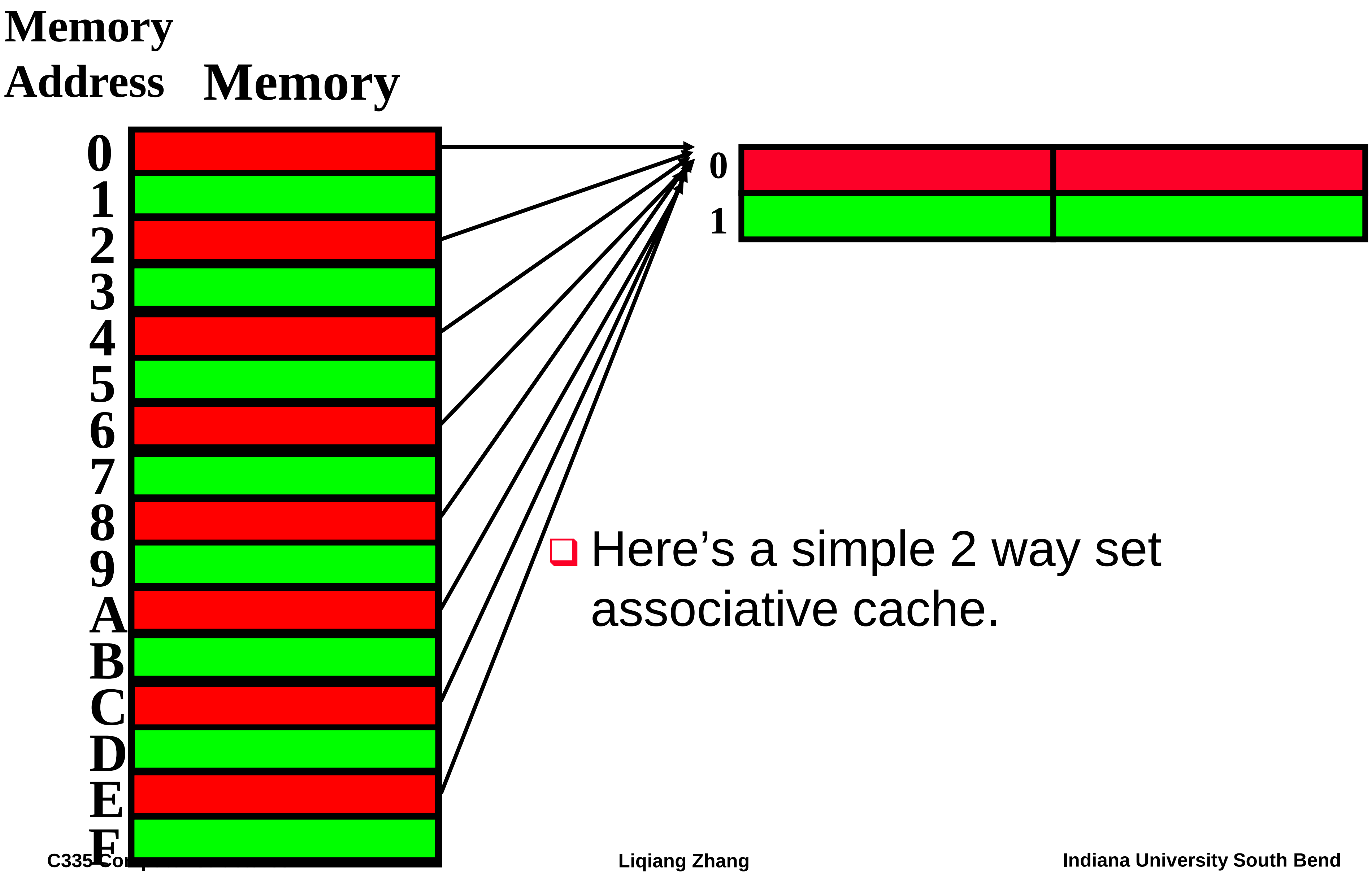
Fully Associative Cache (3/3)

- 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: infeasible

N-Way Set Associative Cache (1/3)

- Memory address fields:
  - Tag: same as before
  - Offset: same as before
  - Index: points us to the correct “set”
- 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

Associative Cache Example



- Here’s a simple 2 way set associative cache.

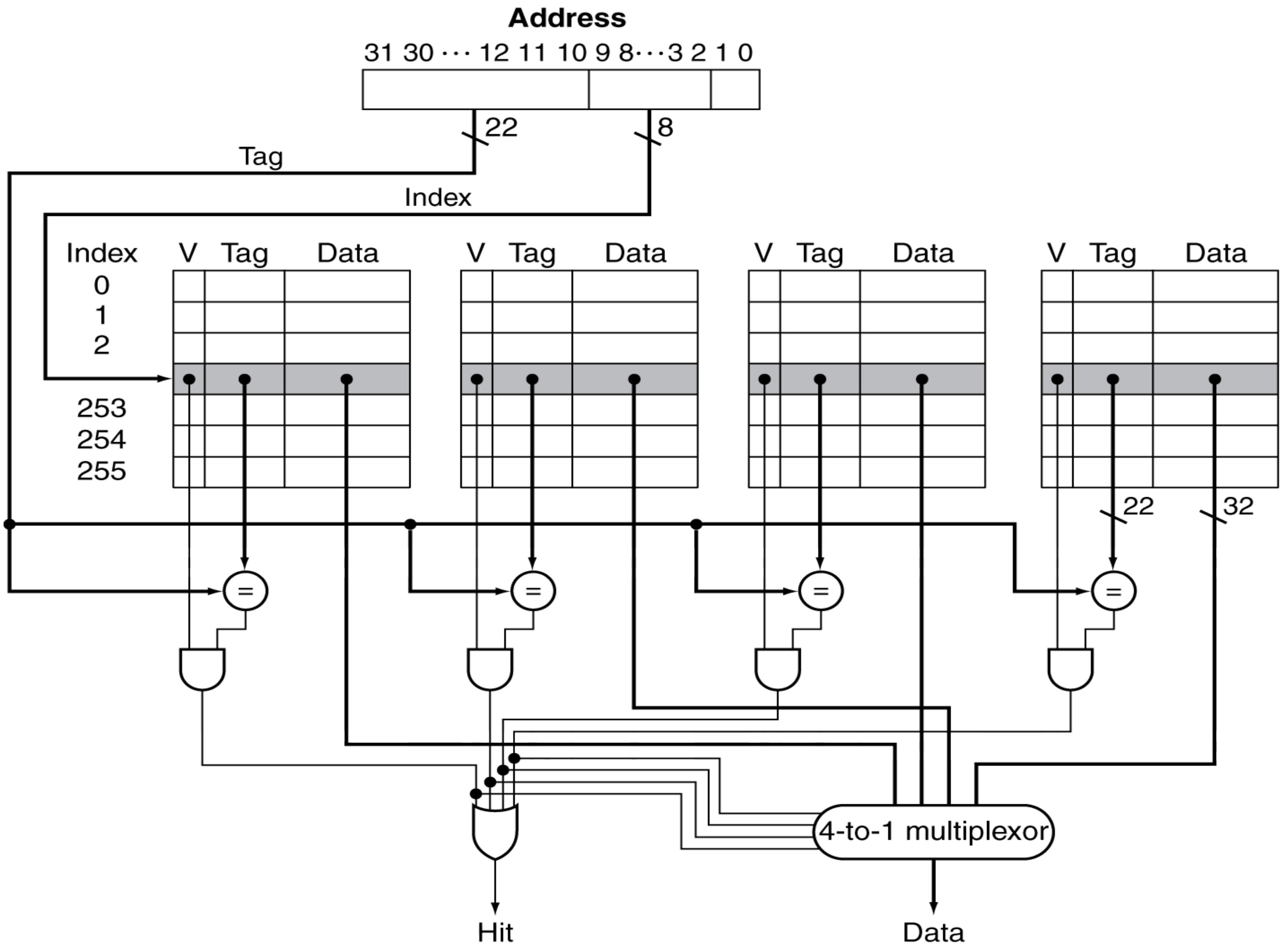
N-Way Set Associative Cache (2/3)

- Basic Idea
  - cache is direct-mapped w/respect to sets
  - each set is fully associative
  - basically N direct-mapped caches working in parallel: each has its own valid bit and data
- Given memory address:
  - Find correct set using Index value.
  - Compare Tag with all Tag values in the determined set.
  - If a match occurs, hit!, otherwise a miss.
  - Finally, use the offset field as usual to find the desired data within the block.

N-Way Set Associative Cache (3/3)

- 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
  - it's Fully Assoc if it's M-way set assoc
  - so these two are just special cases of the more general set associative design

4-Way Set Associative Cache Circuit



Example: fully associative cache (1/2)

Here is a series of address references given as word addresses: 2, 3, 6, 10, 7, 12, 2, 18, 11, and 3. Assuming a fully associative cache with 8 two-word blocks (total size 16 words) that is initially empty, label each reference in the list as a hit or a miss and show the final contents of the cache. (note: the basic unit is word!)

Tag	Data (two-word block)	Tag	Data (two-word block)	Tag	Data (two-word block)	Tag	Data (two-word block)

in one row

Tag	Data (two-word block)	Tag	Data (two-word block)	Tag	Data (two-word block)	Tag	Data (two-word block)

2 (00000010), 3(00000011), 6(00000110), 10(00001010), 7(00000111), 12(00001100), 2(00000010), 18(00010010), 11(00001011), 3(00000011)

Example: fully associative cache (2/2)

Tag	Data (two-word block)	Tag	Data (two-word block)	Tag	Data (two-word block)	Tag	Data (two-word block)
0000001	[2, 3]	0000011	[6, 7]	0000101	[10, 11]	00001110	[12, 13]

in one row

Tag	Data (two-word block)	Tag	Data (two-word block)	Tag	Data (two-word block)	Tag	Data (two-word block)
0001001	[18, 19]						

2 (00000010) → miss, 3(00000011) → hit, 6(00000110) → miss, 10(00001010) → miss, 7(00000111) → hit, 12(00001100) → miss, 2(00000010) → hit, 18(00010010) → miss, 11(00001011) → hit, 3(00000011) → ?

Example: 2-way set associative cache (1/2)

Here is a series of address references given as word addresses: 2, 3, 6, 10, 7, 12, 2, 18, 11, and 3. Assuming a 2-way set associative cache with 8 two-word blocks (total size 16 words) that is initially empty, label each reference in the list as a hit or a miss and show the final contents of the cache. (note: the basic unit is word!)

Index	Tag	Data (two-word block)	Tag	Data (two-word block)
00				
01				
10				
11				

2 (00000010), 3(00000011), 6(00000110), 10(00001010), 7(00000111), 12(00001100), 2(00000010), 18(00010010), 11(00001011), 3(00000011)

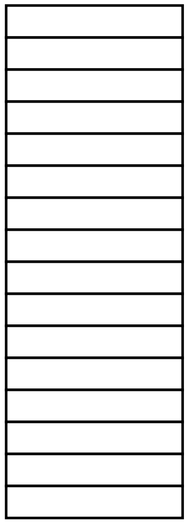
Example: 2-way set associative cache (2/2)

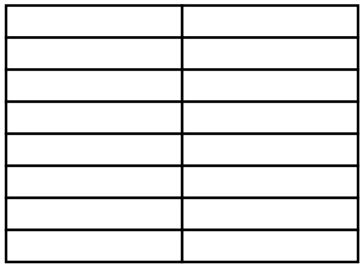
Here is a series of address references given as word addresses: 2, 3, 6, 10, 7, 12, 2, 18, 11, and 3. Assuming a 2-way set associative cache with 8 two-word blocks (total size 16 words) that is initially empty, label each reference in the list as a hit or a miss and show the final contents of the cache. (note: the basic unit is word!)

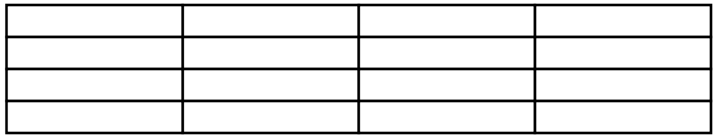
Index	Tag	Data (two-word block)	Tag	Data (two-word block)
00				
01	00000	[2, 3]	00001	[10, 11]
10	00001	[12, 13]		
11	00000	[6, 7]		

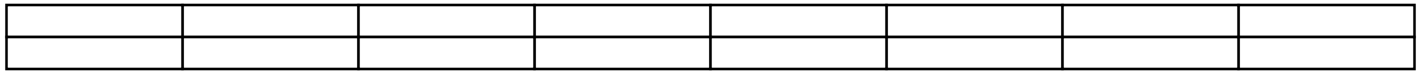
2 (00000010) → miss, 3(00000011) → hit, 6(00000110) → miss, 10(00001010) → miss, 7(00000111) → hit, 12(00001100) → miss, 2(00000010) → hit, 18(00010010) → ?
11(00001011) → ?
3(00000011) → ?

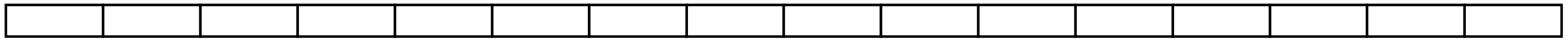
Cache Shapes (for 16 blocks)


Direct-mapped

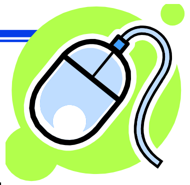

2-way set-associative


4-way set-associative


8-way set-associative


Fully associative

Cache Block Replacement: Need to Make a Decision!



- Direct Mapped Cache:
  - Each memory location can only be mapped to 1 cache location
  - No need to make any decision :-)
    - Current item replaces the previous item in that cache location
- N-way Set Associative Cache:
  - Each memory location have a choice of N cache locations
- Fully Associative Cache:
  - Each memory location can be placed in ANY cache location
- Cache miss in an N-way Set Associative or Fully Associative Cache:
  - Bring in new block from memory
  - Remove a cache block to make room for the new block
  - ====> We need to make a decision on which block to replace!

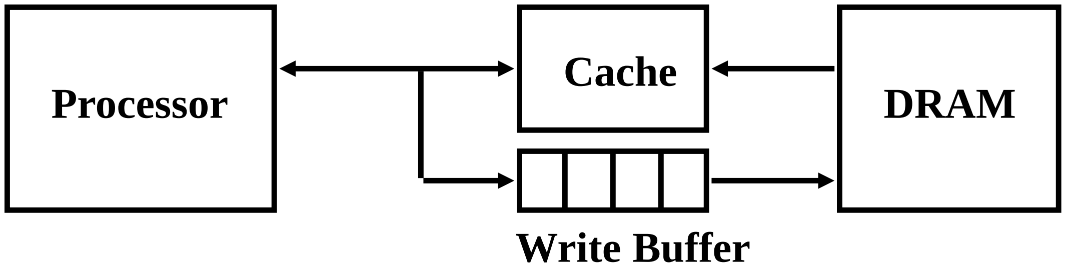
Cache Block Replacement Policy

- Random Replacement:
  - Hardware randomly selects a cache item and replaces it
- Least Recently Used (LRU):
  - Hardware keeps track of the access history
  - Replace the entry that has been unused for the longest time
- Others?

Cache Write Policy: Write Through versus Write Back

- Cache read is much easier to handle than cache write:
  - Instruction cache is much easier to design than data cache
- Cache write:
  - How do we keep data in the cache and memory consistent?
- Two options (decision time again :-)
  - Write Back: write to cache only. Write the cache block to memory when that cache block is being replaced on a cache miss.
    - Use a “dirty” bit for each cache block
    - Multiple writes within a cache block require one write to memory
    - Greatly reduces the memory bandwidth requirement
    - Control can be complex
  - Write Through: write to cache and memory at the same time.
    - Simplifies data coherency
    - What!!! How can this be? Isn’t memory too slow for this?

Write Buffer for Write Through



- A Write Buffer is placed between the Cache and Memory
  - Processor: writes data into the cache and the write buffer
  - Memory controller: write contents of the buffer to memory
- Write buffer is just a FIFO queue:
  - Typical number of entries: 4
  - Works fine if: Store frequency (w.r.t. time) << 1 / DRAM write cycle
- Memory system designer's nightmare:
  - Store frequency (w.r.t. time) > 1 / DRAM write cycle
  - Or bursty writes that exceed DRAM write speed
  - Write buffer saturation

Big Idea

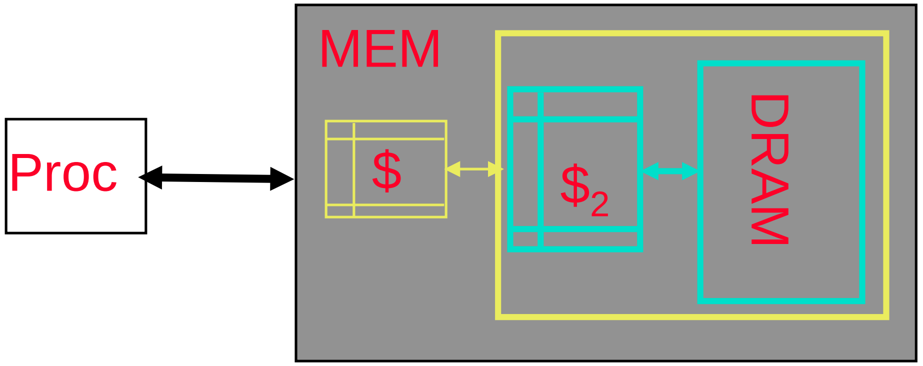
- Create the illusion of a memory that is large, cheap, and fast - on average
- How to choose between associativity, block size, replacement & write policy?
- Design against a performance model
  - Minimize: *Average Memory Access Time*

    = Hit Time X Hit Rate

    + Miss Penalty x Miss Rate
  - influenced by technology & program behavior
- How can we improve miss penalty?

Improving Miss Penalty

- When caches first became popular, Miss Penalty ~ 10 processor clock cycles
- But today, for a 2400 MHz Processor (0.4 ns per clock cycle) and 40 ns to go to DRAM
  $\Rightarrow$ 100 processor clock cycles!



Solution: another cache between memory and the processor cache: <u>Second Level (L2) Cache</u>

Multilevel Cache Example

- Given
  - CPU base CPI = 1, clock rate = 4GHz
  - Miss rate/instruction = 2%
  - Main memory access time = 100ns
- With just primary cache
  - Miss penalty = 100ns/0.25ns = 400 cycles
  - Effective CPI = 1 + 0.02 × 400 = 9

Multilevel Cache Example (cont.)

- Now add L-2 cache
  - Access time = 5ns
  - Global miss rate to main memory = 0.5%
- Primary miss with L-2 hit
  - Penalty = 5ns/0.25ns = 20 cycles
- Primary miss with L-2 miss
  - Extra penalty = 400 cycles
- CPI = $1 + 0.02 \times 20 + 0.005 \times 400 = 3.4$
- Performance ratio = 9/3.4 = 2.6

Multilevel Cache Considerations

- Primary cache
  - Focus on minimal hit time
- L-2 cache
  - Focus on low miss rate to avoid main memory access
  - Hit time has less overall impact
- Results
  - L-1 cache usually smaller than a single cache
  - L-1 block size smaller than L-2 block size

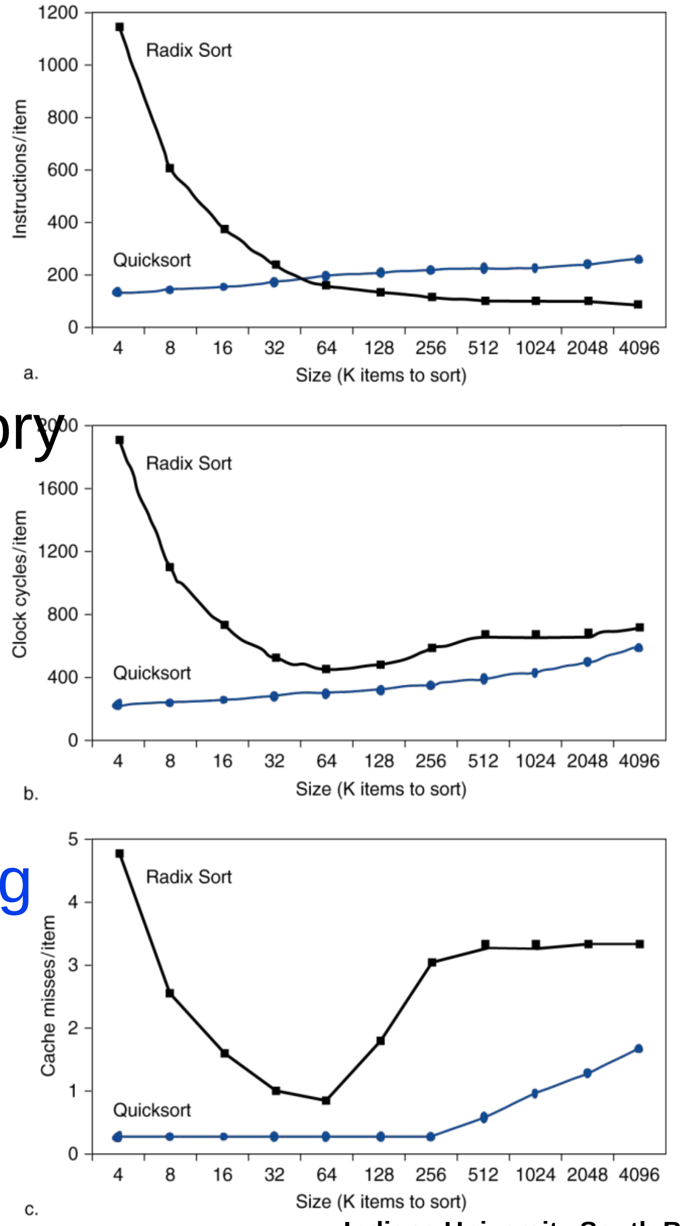
Interactions with Advanced CPUs

- Out-of-order CPUs can execute instructions during cache miss
  - Pending store stays in load/store unit
  - Dependent instructions wait
    - Independent instructions continue
- Effect of miss depends on program data flow
  - Much harder to analyze
  - Use system simulation

Interactions with Software

- ❑ Misses depend on memory access patterns
  - Algorithm behavior
  - Compiler optimization for memory access

- ❑ Understanding of the memory hierarchy is critical to understanding the performance of programs



And in Conclusion…

- We've discussed memory caching in detail. Caching in general shows up over and over in computer systems
  - File system cache
  - Web page cache
- Big idea: if something is expensive but we want to do it repeatedly, do it once and <u>cache</u> the result.

And in Conclusion…

- Cache design choices:
  - size of cache: speed v. capacity
  - direct-mapped v. associative
  - for N-way set assoc: choice of N
  - Write through v. write back
  - block replacement policy
  - 2nd level cache? 3rd level cache?
- Use performance model to pick between choices, depending on programs, technology, budget, ...