

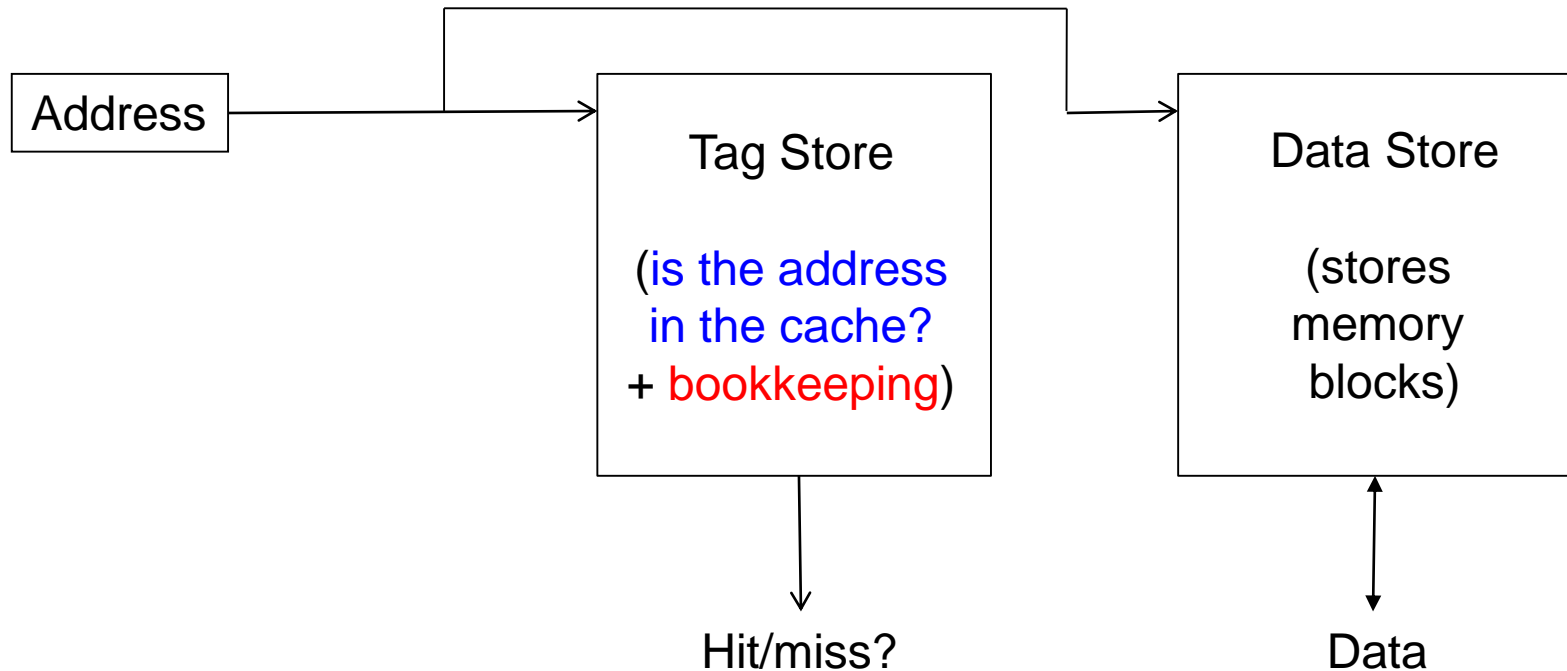
# Computer Architecture

## 10. Cache Memory

Jianhua Li

College of Computer and Information  
Hefei University of Technology

# Cache Abstraction and Metrics



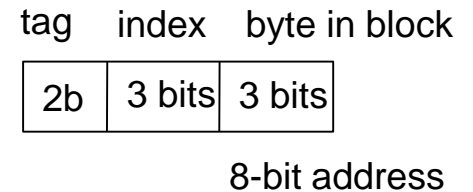
- Cache hit rate =  $(\# \text{ hits}) / (\# \text{ hits} + \# \text{ misses}) = (\# \text{ hits}) / (\# \text{ accesses})$
- Average memory access time (AMAT)  
=  $(\text{hit-rate} * \text{hit-latency}) + (\text{miss-rate} * \text{miss-latency})$
- Aside: *Can reducing AMAT reduce performance?*

# A Basic Hardware Cache Design

- We will start with a basic hardware cache design
- Then, we will examine a multitude of ideas to make it better

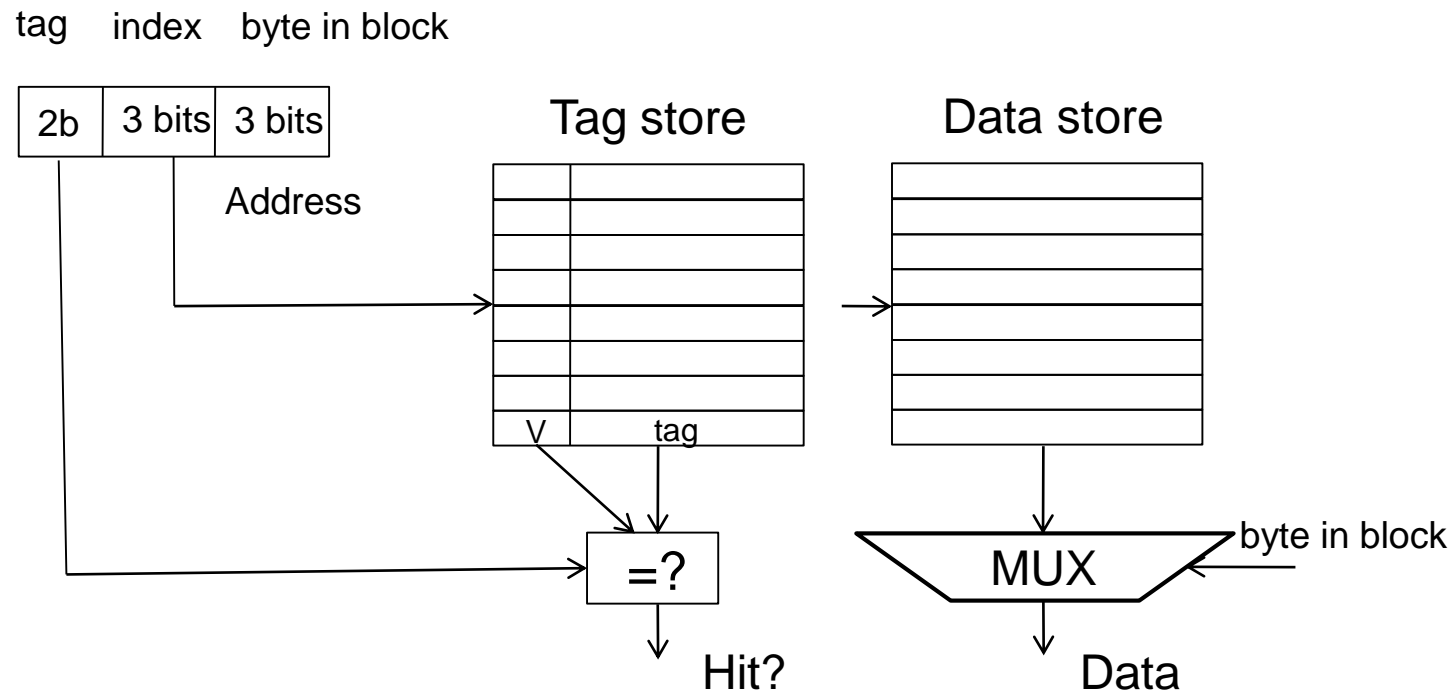
# Blocks and Addressing the Cache

- Memory is logically divided into fixed-size blocks
- Each block maps to a location in the cache, determined by **the index bits** in the address
  - used to index into the tag and data stores
- Cache access:
  - index into the tag and data stores with index bits in address
  - check valid bit in tag store
  - compare tag bits in address with the stored tag in tag store
- If a block is in the cache (cache hit), the stored tag should be valid and match the tag of the block



# Direct-Mapped Cache

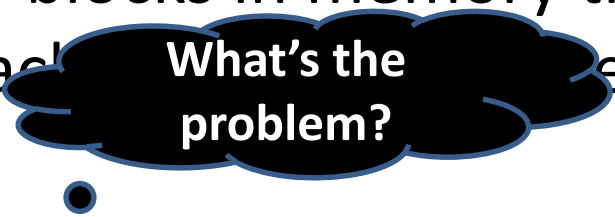
- Assume byte-addressable memory: 256 bytes, 8-byte blocks → 32 blocks
- Assume cache: 64 bytes, 8 blocks
  - Direct-mapped: A block can go to only one location



- Addresses with same index contend for the same location

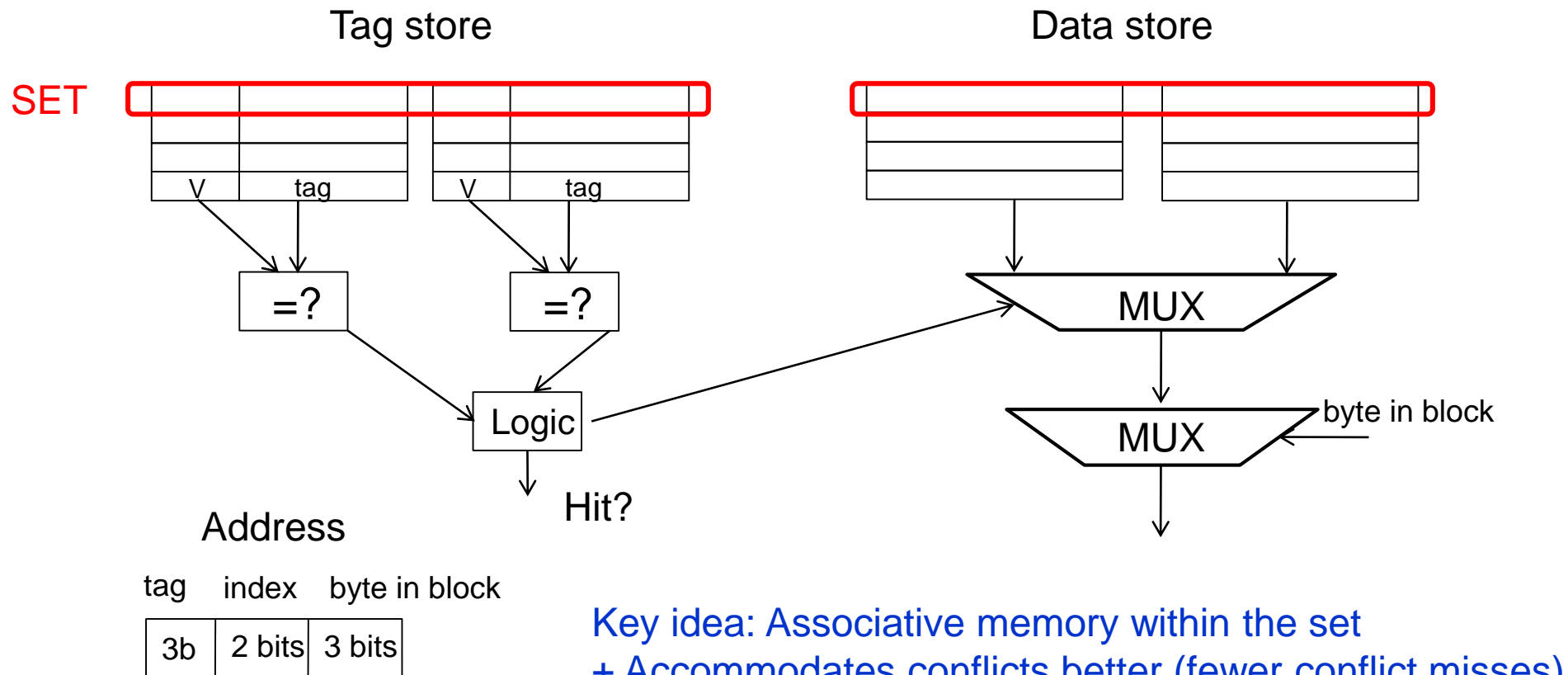
- Cause conflict misses

# Direct-Mapped Caches

- **Direct-mapped cache:** Two blocks in memory that map to the same index in the cache  
to the same index in the cache at the same time  
– One index  $\rightarrow$  one entry
- 
- What's the problem?
- Can lead to 0% hit rate if more than one block accessed in an interleaved manner map to the same index  
– Assume addresses A and B have the same index bits but different tag bits  
– A, B, A, B, A, B, A, B, ...  $\rightarrow$  conflict in the cache index  
– All accesses are **conflict misses**

# Set Associativity

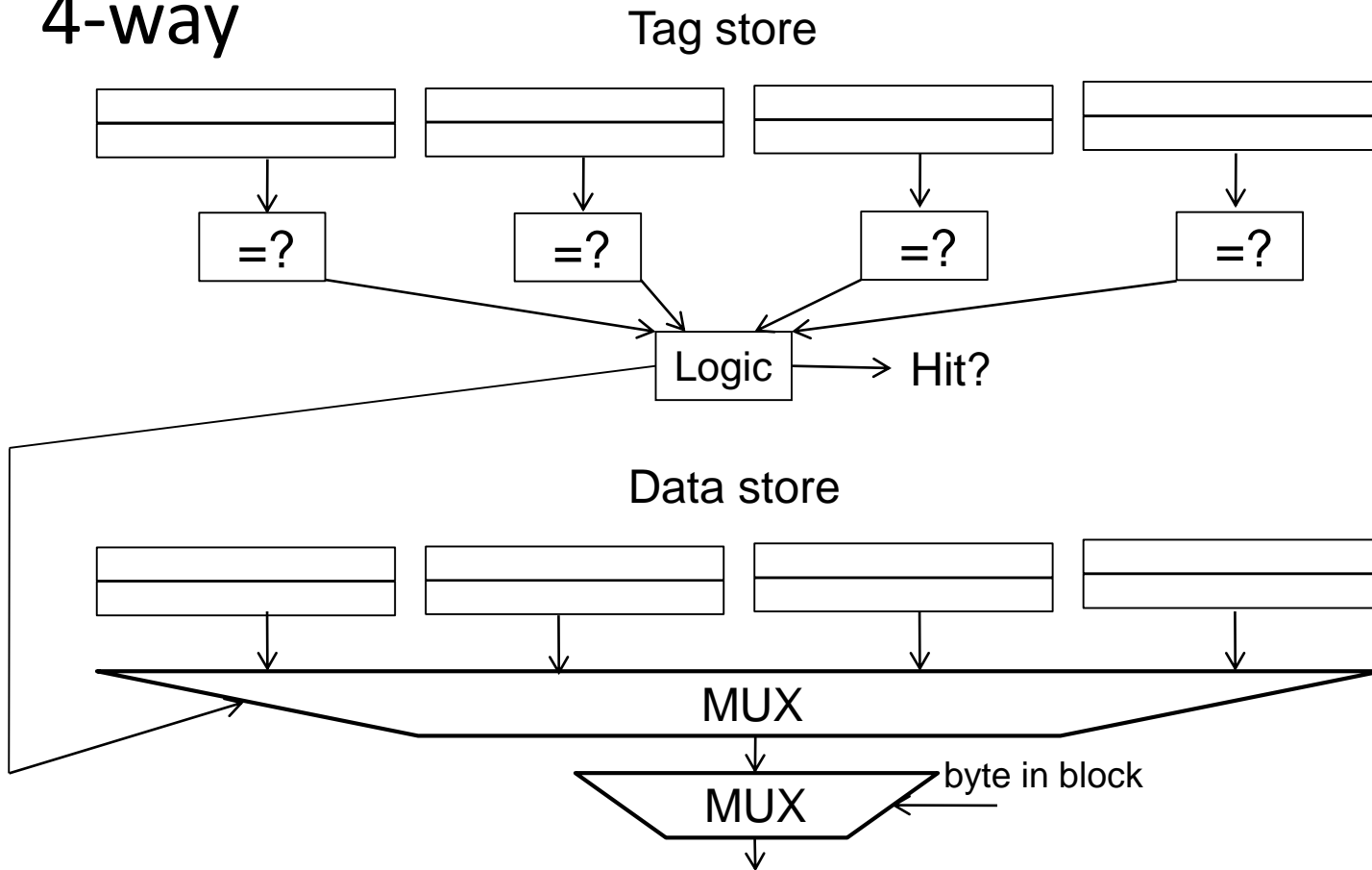
- Addresses 0 and 8 always conflict in direct mapped cache
- Instead of having one column of 8, have 2 columns of 4 blocks



Key idea: Associative memory within the set  
 + Accommodates conflicts better (fewer conflict misses)  
 -- More complex, slower access, larger tag store

# Higher Associativity

- 4-way



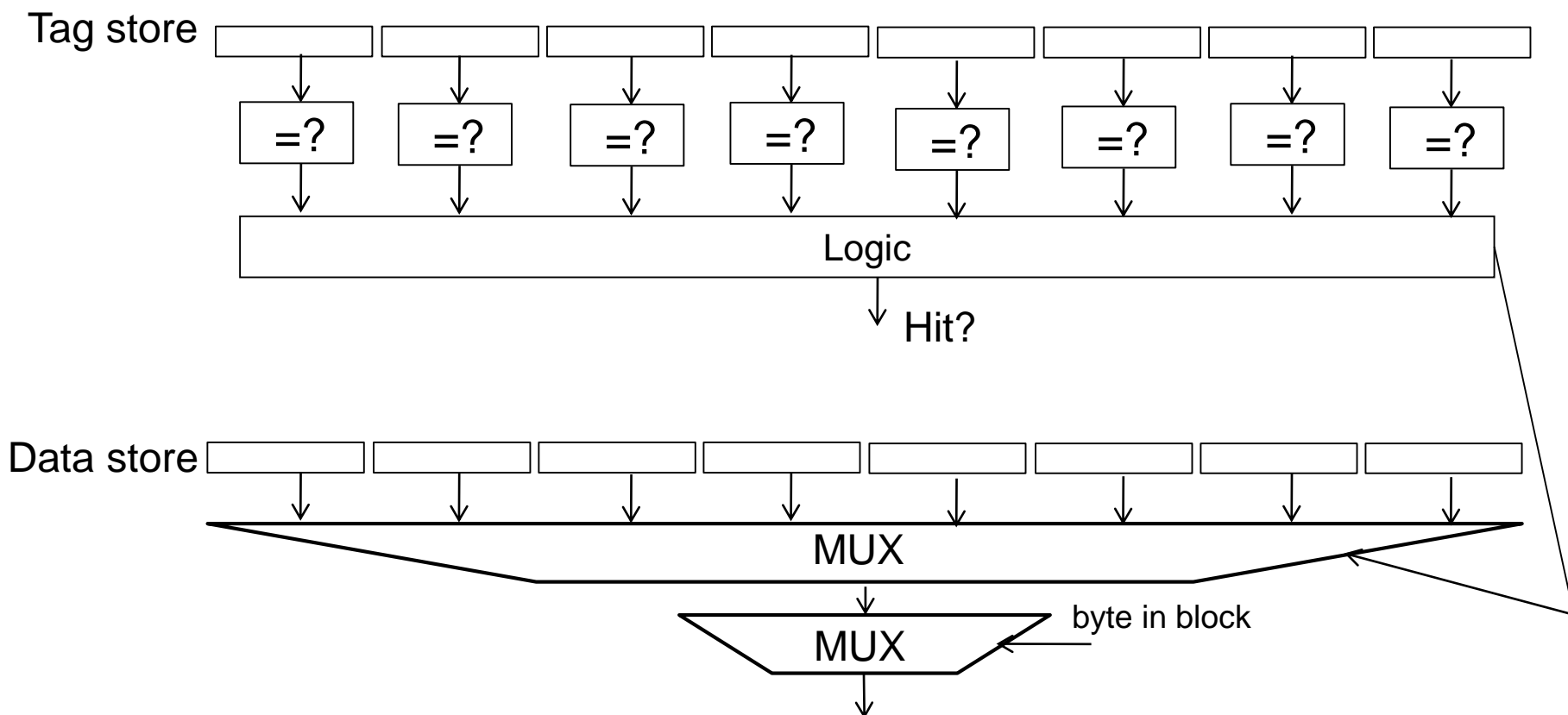
+ Likelihood of conflict misses even lower

-- More tag comparators and wider data mux; larger tags



# Full Associativity

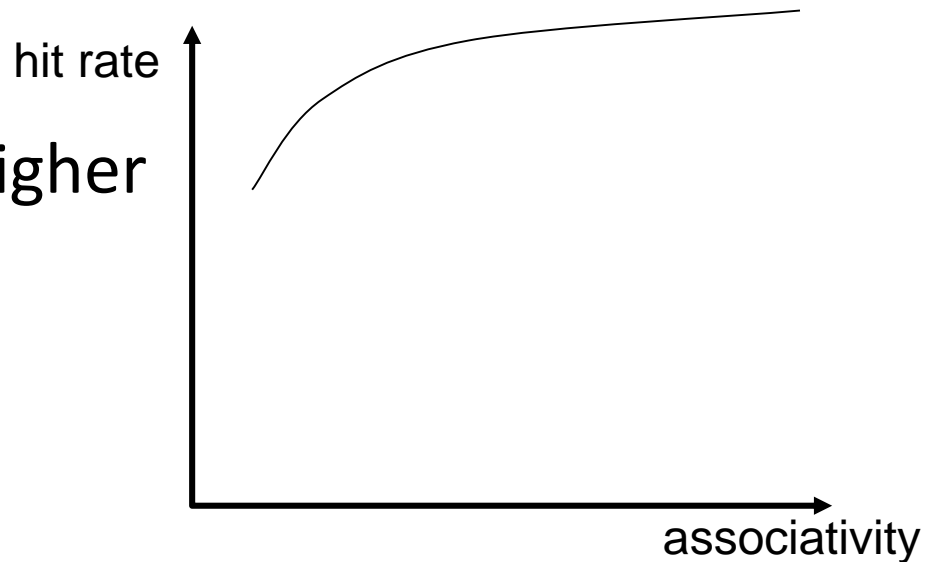
- Fully associative cache
  - A block can be placed in **any** cache location



# Associativity (and Tradeoffs)

- **Degree of associativity**: How many blocks can map to the same set?
- Higher associativity
  - ++ Higher hit rate
  - Slower cache access time (hit latency and data access latency)
  - More expensive hardware (more comparators)

- Diminishing returns from higher associativity



# Issues in Set-Associative Caches

- Think of each block in a set having a “priority”
  - Indicating how important it is to keep the block in the cache
- Key issue: **How do you determine block priorities?**
- There are three key decisions can adjust priority:
  - Insertion, promotion, eviction (replacement)
- Insertion: What happens to priorities on a cache fill?
  - Where to insert the incoming block, whether or not to insert the block
- Promotion: What happens to priorities on a cache hit?
  - Whether and how to change block priority
- Eviction/replacement: What happens to priorities on a cache miss?
  - Which block to evict and how to adjust priorities

# Here we talk about: Replacement Policy

- Which block in the set to replace on a cache miss?
  - Any invalid block first
  - If all are valid, consult the replacement policy
    - Random
    - FIFO
    - **LRU: Least recently used (how to implement?)**
    - NRU: Not most recently used
    - LFU: Least frequently used?
    - Least costly to re-fetch?
      - Why would memory accesses have different cost?
    - Hybrid replacement policies
    - Optimal replacement policy?

# Implementing LRU

- Idea: Evict the least recently accessed block
- Problem: Need to **keep track of access ordering** of blocks
- Question: 2-way set associative cache:
  - How to implement LRU perfectly?
- Question: 4-way set associative cache:
  - How to implement LRU perfectly?
  - How many different orderings possible for the 4 blocks in the set?
  - How many bits needed to encode the LRU order of a block?
  - What is the logic needed to determine the LRU victim?

# Approximations of LRU

- Most modern processors do not implement “true LRU” (also called “perfect LRU”) in highly-associative caches
- Why?
  - True LRU is complex
  - LRU is an approximation to predict locality anyway (i.e., not the best possible cache management policy)
- Alternative examples:
  - Not MRU (not most recently used)
  - Hierarchical LRU: divide the 4-way set into 2-way “groups”, track the MRU group and the MRU way in each group
  - Victim-NextVictim Replacement: Only keep track of the victim and the next victim

# LRU or Random?

- LRU vs. Random: Which one is better?
  - Example: 4-way cache, cyclic references to A, B, C, D, E
    - 0% hit rate with LRU policy
- Set thrashing: When the “program working set” in a set is larger than set associativity
  - Random replacement policy is better when thrashing occurs
- In practice:
  - Depends on workload
  - Average hit rate of LRU and Random are similar
- Best of both Worlds: Hybrid of LRU and Random
  - How to choose between the two? Set sampling
    - See Qureshi et al., “A Case for MLP-Aware Cache Replacement,” ISCA 2006.

# What Is the Optimal Replacement Policy?

- Belady's OPT
  - Replace the block that is going to be referenced furthest in the future by the program
  - Belady, “A study of replacement algorithms for a virtual-storage computer,” IBM Systems Journal, 1966.
  - How do we implement this? Simulate?
- Is this optimal for minimizing miss rate?
- Is this optimal for minimizing execution time?
  - No. Cache miss latency/cost varies from block to block!
  - Two reasons: Remote vs. local caches and miss overlapping
  - Qureshi et al. “A Case for MLP-Aware Cache Replacement,” ISCA 2006.



# Cache versus Page Replacement

- Physical memory (DRAM) is a cache for disk
  - Usually managed by system software via the virtual memory subsystem
- Page replacement is similar to cache replacement
- Page table is the “tag store” for physical memory data store
- What is the difference?
  - Access speed: cache vs. physical memory
  - Number of blocks in a cache vs. physical memory
  - “Tolerable” amount of time to find a replacement candidate
  - Role of hardware versus software

# What's in A Tag Store Entry?

- Valid bit
- Tag
- Replacement policy bits
- Dirty bit?
  - Write back vs. write through caches

# Handling Writes (I)

- When do we write the modified data in a cache to the next level?
  - **Write through**: At the time the write happens
  - **Write back**: When the block is evicted
- Write-back
  - + Can consolidate multiple writes to the same block before eviction
  - + Potentially saves bandwidth between cache levels + saves energy
  - Need a bit in the tag store indicating the block is “dirty/modified”
- Write-through
  - + Simpler
  - + All levels are up to date. **Consistency**: Simpler cache coherence because no need to check lower-level caches
  - More bandwidth intensive; no coalescing of writes

## Handling Writes (II)

- Do we allocate a cache block on a write miss?
  - Allocate on write miss: Yes
  - No-allocate on write miss: No
- Allocate on write miss
  - + Can consolidate writes instead of writing each of them individually to next level
  - + Simpler because write misses can be treated the same way as read misses
    - Requires (?) transfer of the whole cache block
- No-allocate
  - + Conserves cache space if locality of writes is low (potentially better cache hit rate)

# Sectored Caches

- Idea: Divide a block into sub-blocks (or sectors)
  - Have separate valid and dirty bits for each sector
  - When is this useful? (Think writes...)
- ++ No need to transfer entire cache block into cache
  - A write simply validates and updates a sub-block
- ++ More freedom in transferring sub-blocks into cache
  - How many sub-blocks do you transfer on a read?
- More complex design
- May not exploit spatial locality fully for reads



# Instruction vs. Data Caches

- Separate or Unified?
- Unified:
  - + Dynamic sharing of cache space: no overprovisioning that might happen with static partitioning (i.e., split I and D caches)
  - Instructions and data can thrash each other (i.e., no guaranteed space for either)
  - I and D are accessed in different places in the pipeline. Where do we place the unified cache for fast access?
- First level caches are almost always split
  - Mainly for the last reason above
- Second and higher levels are almost always unified

# Multi-level Caching in a Pipelined Design

- First-level caches (instruction and data)
  - Decisions very much affected by cycle time
  - Small, lower associativity
  - Tag store and data store accessed in parallel
- Second-level caches
  - Decisions need to **balance hit rate and access latency**
  - Usually large and highly associative; latency not as important
  - Tag store and data store accessed serially
- Serial vs. Parallel access of levels
  - Serial: Second level cache accessed only if first-level misses
  - Second level does not see the same accesses as the first
    - First level acts as a filter (filters some temporal and spatial locality)
    - Management policies are therefore different

# Cache Performance

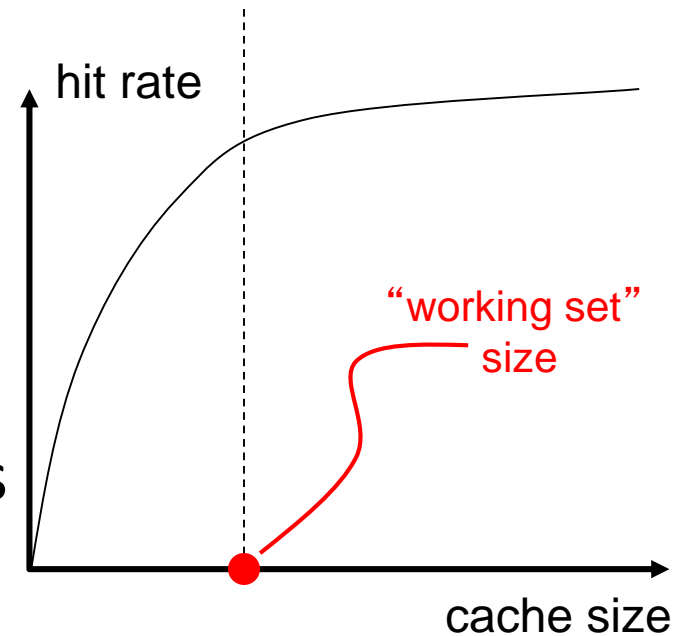


# Cache Parameters vs. Miss/Hit Rate

- Cache size
- Block size
- Associativity
- Replacement policy
- Insertion/Placement policy

# Cache Size

- Cache size: total data (not including tag) capacity
  - bigger can exploit temporal locality better
  - not ALWAYS better
- Too large a cache adversely affects hit and miss latency
  - smaller is faster => bigger is slower
  - access time may degrade critical path
- Too small a cache
  - doesn't exploit temporal locality well
  - useful data replaced often
- Working set: the whole set of data the executing application references
  - Within a time interval



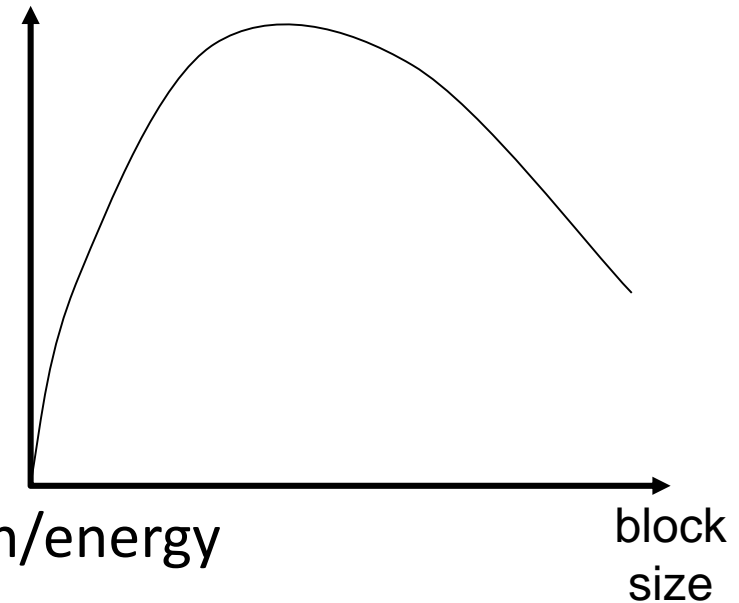
# Block Size

- Block size is the data that is associated with an address tag

Can someone explain  
this curve?

- not necessarily true in hierarchies
  - Sub-blocking: divide into smaller pieces (each with  $V$  bit)
    - Can improve “write” performance
- Too small blocks
  - don’t exploit spatial locality well
  - have larger tag overhead
- Too large blocks
  - too few total # of blocks  $\rightarrow$  less temporal locality exploitation
  - waste of cache space and bandwidth/energy if spatial locality is not high

hit rate



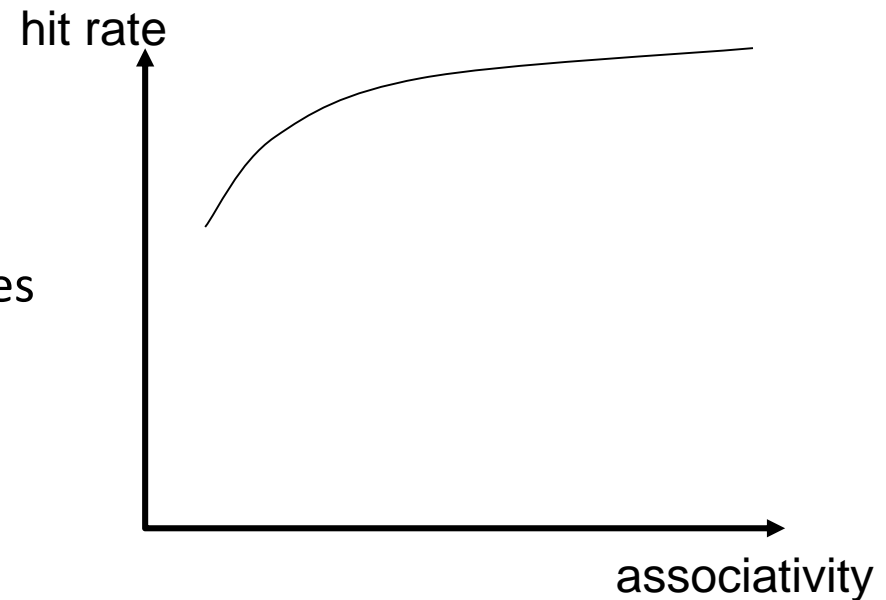
## Large Blocks: Critical-Word and Sub-blocking

- Large cache blocks can take a long time to fill into the cache
  - fill cache line **critical word first**
  - restart cache access before complete fill
- Large cache blocks can waste bus bandwidth
  - divide a block into sub-blocks
  - associate separate valid bits for each sub-block
  - **When is this useful?**



# Associativity

- How many blocks can map to the same index (or set)?
- Larger associativity
  - lower miss rate, less variation among programs
  - diminishing returns, higher hit latency
- Smaller associativity
  - lower cost
  - lower hit latency
    - Especially important for L1 caches
- Power of 2 required?



# Classification of Cache Misses

- Compulsory miss
  - first reference to an address (block) always results in a miss
  - subsequent references should hit unless the cache block is displaced for the reasons below
- Capacity miss
  - cache is too small to hold everything needed
  - defined as the misses that would occur even in a fully-associative cache (with optimal replacement) of the same capacity
- Conflict miss
  - defined as any miss that is neither a compulsory nor a capacity miss

# How to Reduce Each Miss Type

- Compulsory
  - Caching cannot help
  - Prefetching can help
- Conflict
  - More associativity
  - Other ways to get more associativity without making the cache associative
    - Victim cache
    - Hashing
    - Software hints?
- Capacity
  - Utilize cache space better: keep blocks that will be referenced
  - Software management: divide working set such that each “phase” fits in cache

# Improving Cache “Performance”

- Remember
  - Average memory access time (AMAT)  
$$= (\text{hit-rate} * \text{hit-latency}) + (\text{miss-rate} * \text{miss-latency})$$
- Reducing miss rate
  - Remind: reducing miss rate can reduce performance if more costly-to-refetch blocks are evicted
- Reducing miss latency/cost
- Reducing hit latency/cost



# Improving Basic Cache Performance

- Reducing miss rate
  - More associativity
  - Alternatives/enhancements to associativity
    - Victim caches, hashing, pseudo-associativity, skewed associativity
  - Better replacement/insertion policies
  - Software approaches
- Reducing miss latency/cost
  - Multi-level caches
  - Critical word first
  - Subblocking/sectoring
  - Better replacement/insertion policies
  - Non-blocking caches (multiple cache misses in parallel)
  - Issues in multicore caches

**Next Topic**

**Optimizing Cache Performance**