Computer Architecture

Memory: Cache (Part II)

Chap 5.1-5.4, 5.8-5.9

Jaewoong Sim

Electrical and Computer Engineering

Seoul National University

Cache Associativity

Direct-Mapped Cache

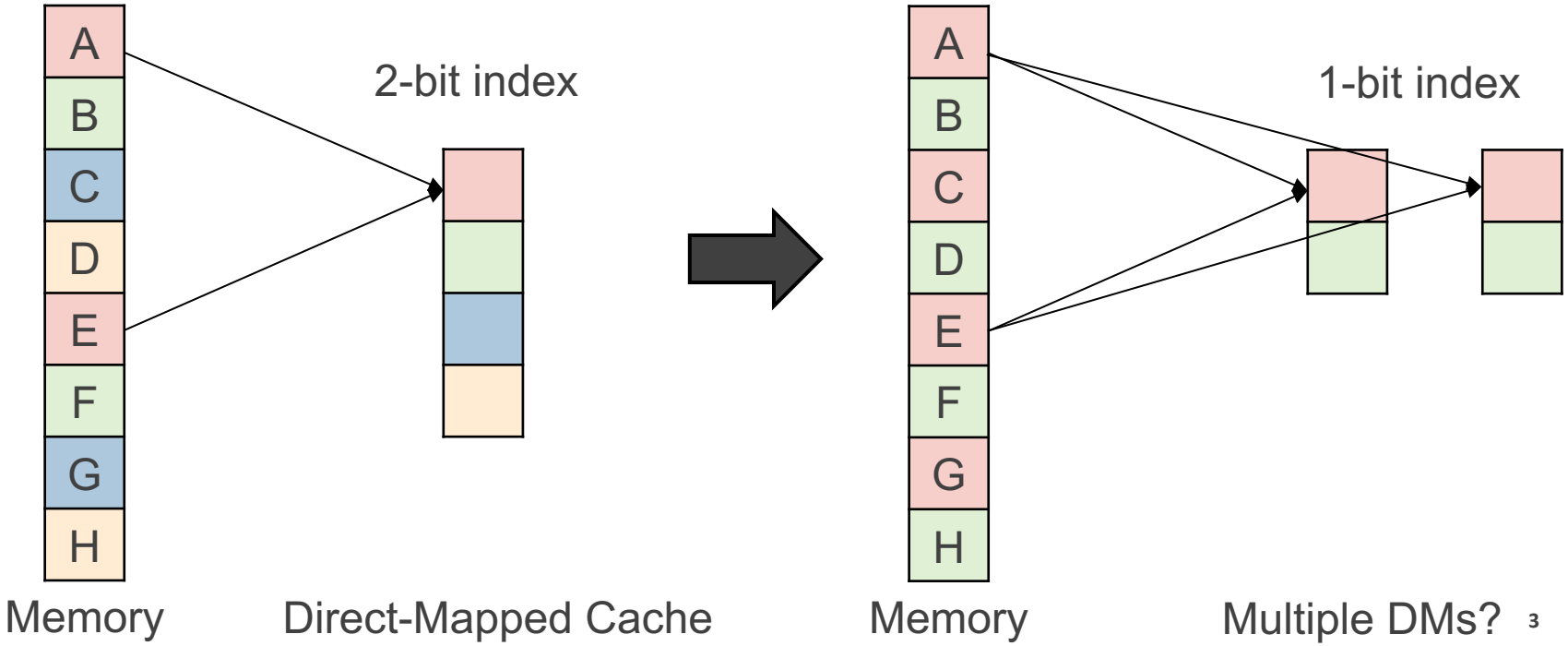
Set-Associative Cache

Fully-Associative Cache

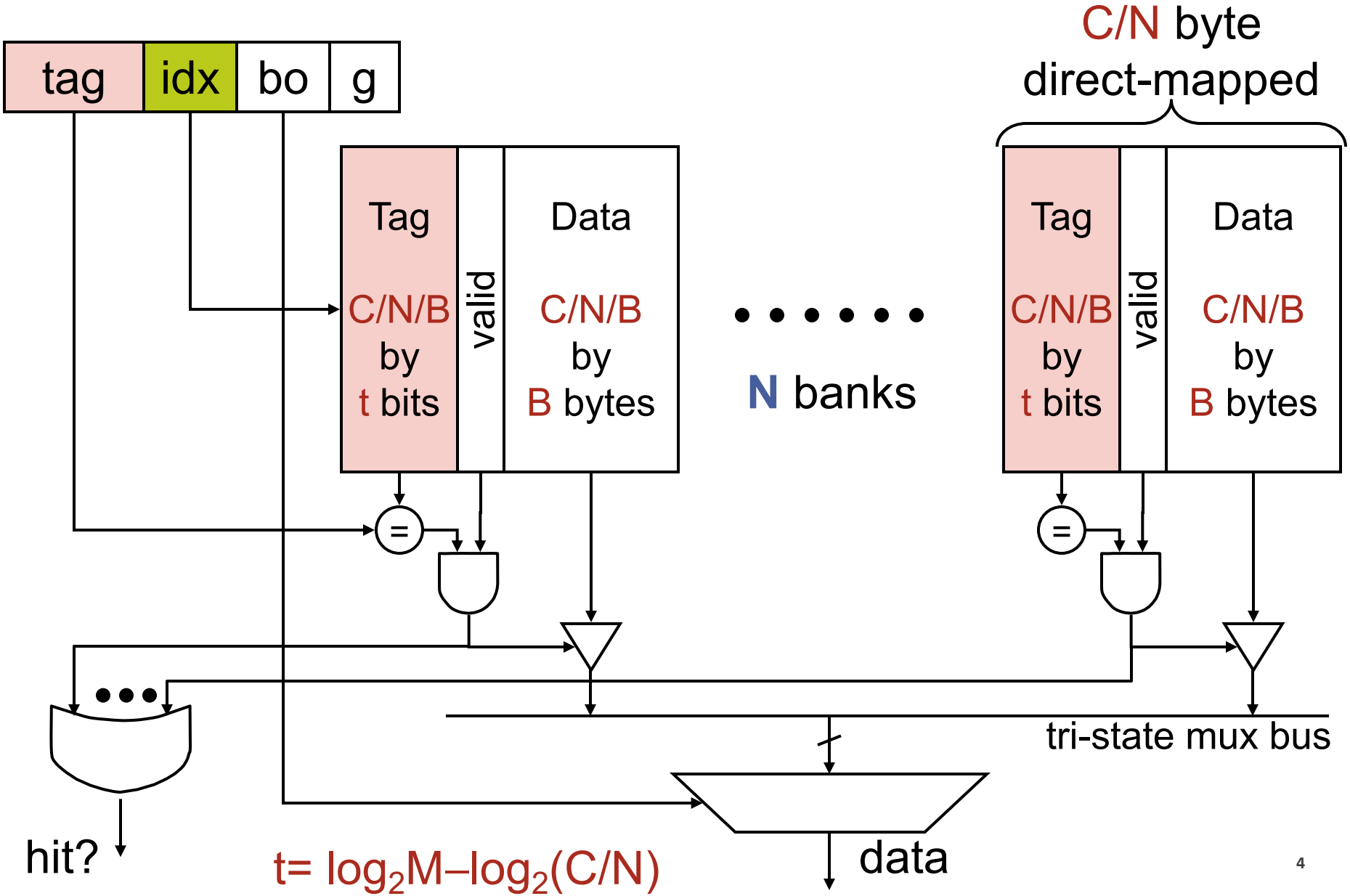
Direct-Mapped Caches

- Q: What is the downside of the direct-mapped cache?
  - Blocks with the same index cannot be present in the cache at the same time

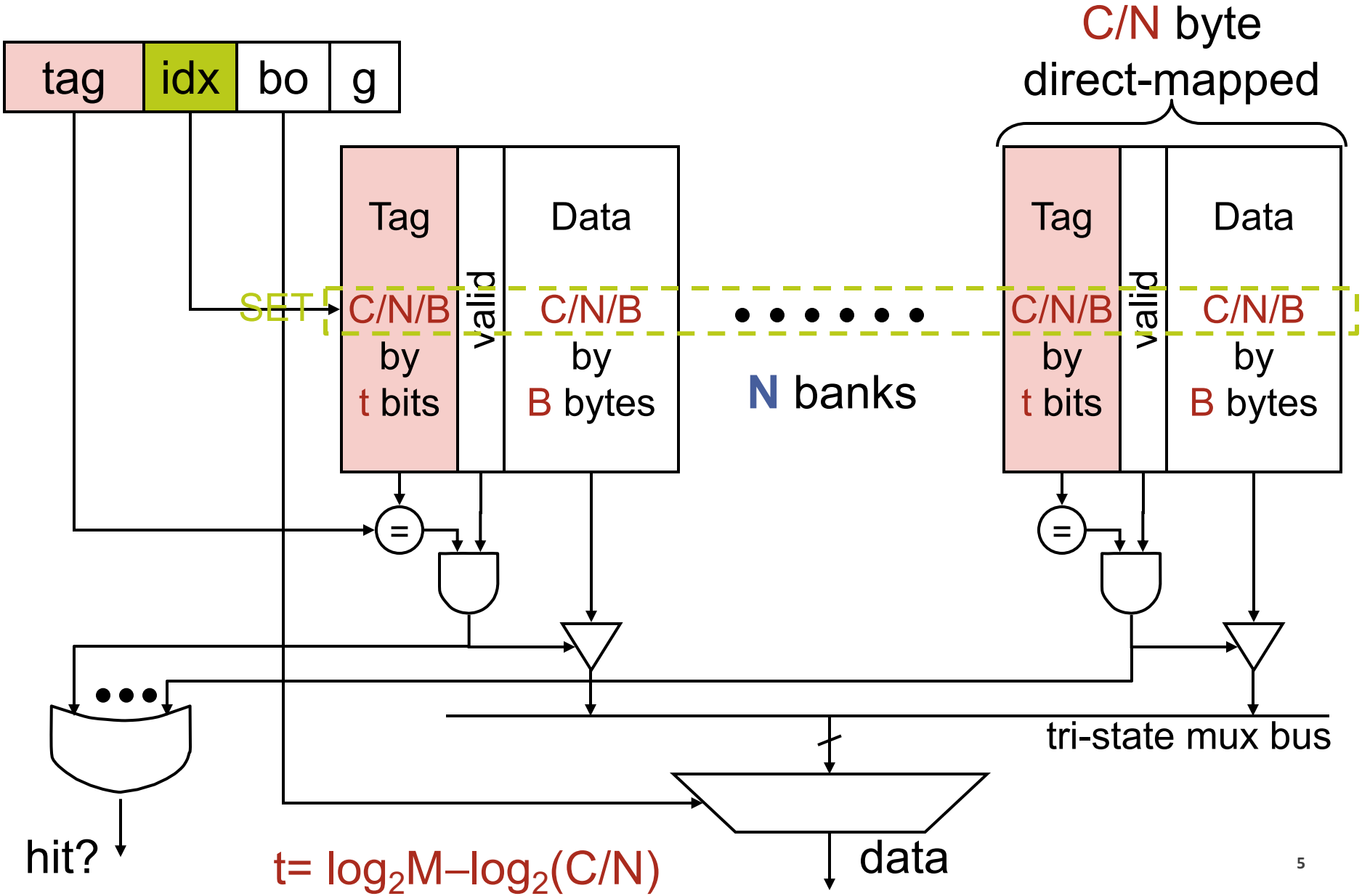
- Example) Access Pattern: A, E, A, E, A, E, A, E ➔ 0% Cache Hit Rate
  - Note that the other three cache locations are not utilized at all!



N-Way Set-Associative Cache



N-Way Set-Associative Cache



N-Way Set-Associative Cache

- C bytes of storage divided into N banks each with C/N/B blocks

- Requires N comparators and N-to-1 multiplexer

N-Way Set-Associative Cache

- C bytes of storage divided into N banks each with C/N/B blocks
- Requires N comparators and N-to-1 multiplexer
- An address is mapped to a particular block in a bank according its index field, but there are N such banks
  (together known as a “set”)
- All addresses with the same index field map to a common “set” of cache blocks
  - 2^t such addresses; SA cache can contain N such blocks at a time

Replacement Policy for Set Associative Caches

- An *N*-way set-associative cache has *N* blocks in each set
  - A new cache block for the same index can evict any of the cached block in the same set...

Replacement Policy for Set Associative Caches

- An *N*-way set-associative cache has *N* blocks in each set
  - A new cache block for the same index can evict any of the cached block in the same set...

- Which block in the set to replace on a cache miss?
  - Any invalid block first
  - If all blocks are valid, do replacement based on replacement policy
    - Random
    - FIFO
    - **LRU (Least Recently Used)**
    - NRU (Not Recently Used)
    - LFU (Least Frequently Used)
    - Optimal Replacement Policy?
    - Lots of research: e.g., RRIP (re-reference interval prediction)

LRU and Pseudo LRU

- LRU: Evict the least recently used (accessed) block
  - How? Need to keep track of access ordering of blocks

LRU and Pseudo LRU

- LRU: Evict the least recently used (accessed) block
  - How? Need to keep track of access ordering of blocks

- How many bits are needed to implement “true LRU”?
  - 2-way set-associative: 2! possible orders => 1 bit per set
  - 4-way set-associative: 4! possible orders => 5 bits per set
  - 8-way set-associative: 8! possible orders => 16 bits per set

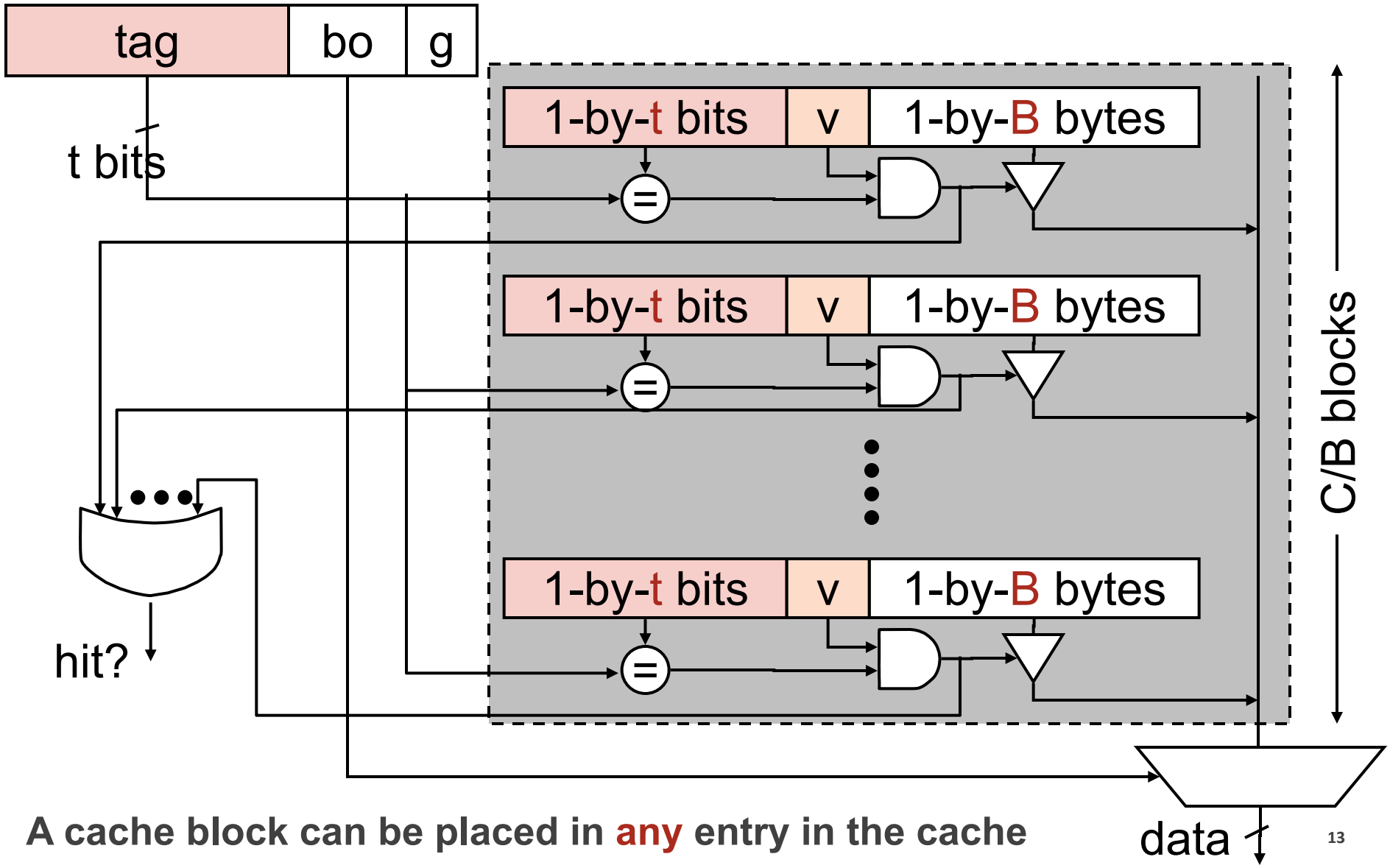
LRU and Pseudo LRU

- LRU: Evict the least recently used (accessed) block
  - How? Need to keep track of access ordering of blocks

- How many bits are needed to implement "true LRU"?
  - 2-way set-associative: 2! possible orders => 1 bit per set
  - 4-way set-associative: 4! possible orders => 5 bits per set
  - 8-way set-associative: 8! possible orders => 16 bits per set

- Most modern processors do not implement "true LRU" in highly-associative caches because it is complex and costly
  - Tree-Pseudo LRU
  - Bit-Pseudo LRU (not MRU; not most recently used)

Fully Associative Cache: N≡C/B



A cache block can be placed in **any** entry in the cache

Fully Associative Cache: N≡C/B

- So, it is a "**Content-addressable (CAM)**" memory
  - Not your regular SRAM. But, critical for some uArchitecture
  - Provide "tag" → search ALL entries → If found, return "data"
  - No index bits used in lookup (=**search every index simultaneously**)

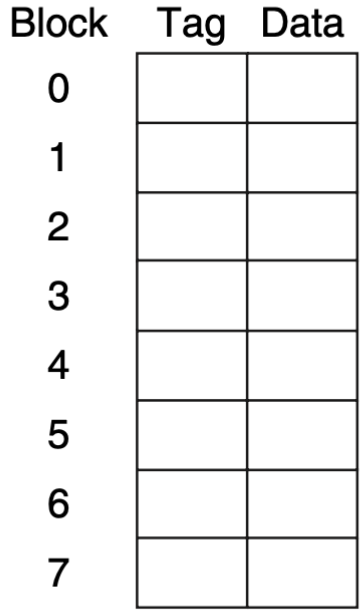
- Any address can go into any of the C/B blocks
  - if C > working set size, no collisions
- Requires one comparator per cache block, a huge multiplexer, and many long wires

  Too expensive/difficult for more than 32~64 blocks at L1 latencies

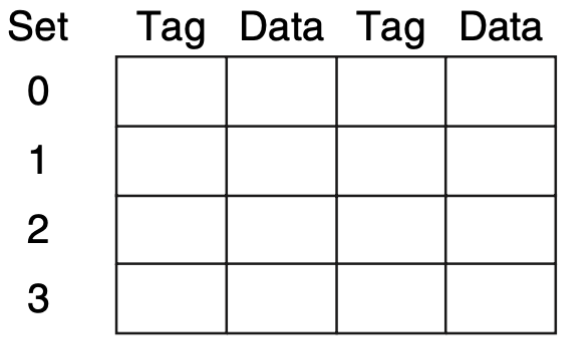
Do we really need a fully-associative cache?

One-way set associative

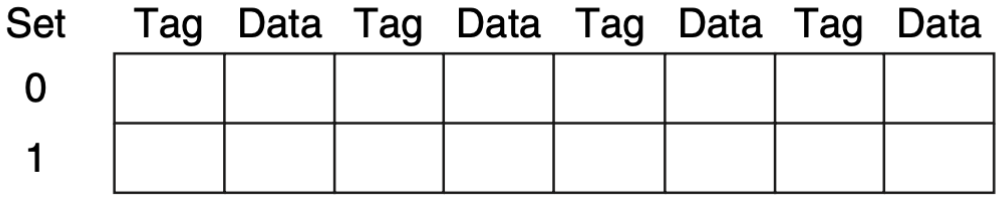
(direct mapped)



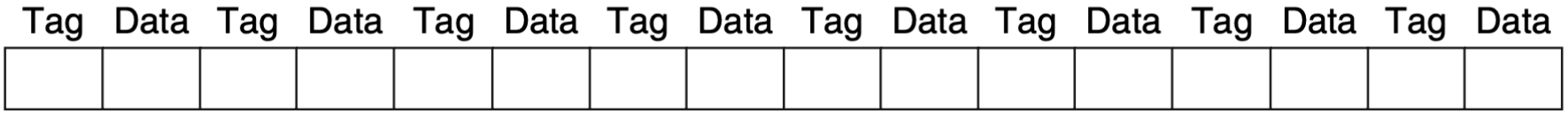
Two-way set associative



Four-way set associative

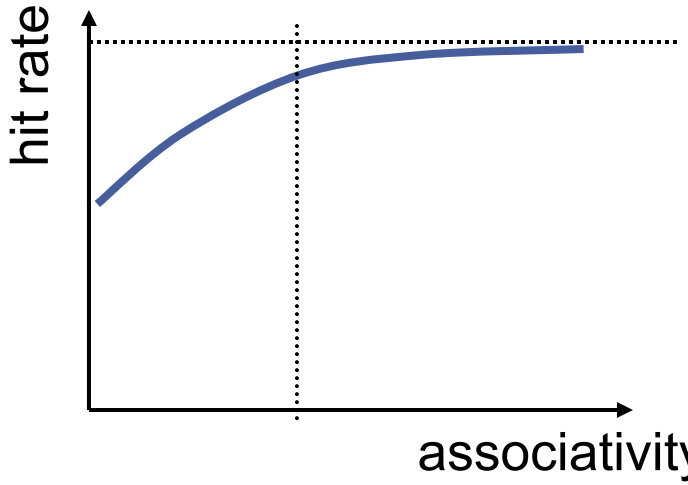


Eight-way set associative (fully associative)



Associativity and Tradeoff

- Degree of associativity
  - # of cache blocks that a cache set can contain
- Higher associativity
  - (+) Fewer conflicts; higher hit rate
  - (-) Slow cache access time; more comparators
- Diminishing returns from higher associativity
  - No need to be fully associative
  - Modern CPUs (Cooper Lake uArch, 2020)
    - `L1-I$:  32KB/core,  8-way SA`
    - `L1-D$:  32KB/core,  8-way SA`
    - `L2$:  1MB/core,  16-way SA`
    - `L3$:  1.375MB/core,  11-way SA`



Classification of Cache Misses (3C's)

- Compulsory (or Cold) miss
- Capacity miss
- Conflict miss

Classification of Cache Misses (3C's)

- **Compulsory (or Cold) 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**
- **Conflict miss**

Classification of Cache Misses (3C's)

- **Compulsory (or Cold) 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**

Classification of Cache Misses (3C's)

- **Compulsory (or Cold) 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**
  - Data displaced due to conflict under direct-mapped or set-associative allocation
  - Defined as any miss that is neither a compulsory nor a capacity miss

How to Reduce Each Cache Miss Type

- **Compulsory misses**
  - Increase block size
    - This can have a negative impact on performance due to the increase in miss penalty
  - Prefetching: predict and fetch data before the data is requested
    - Need good coverage + high accuracy + timeliness

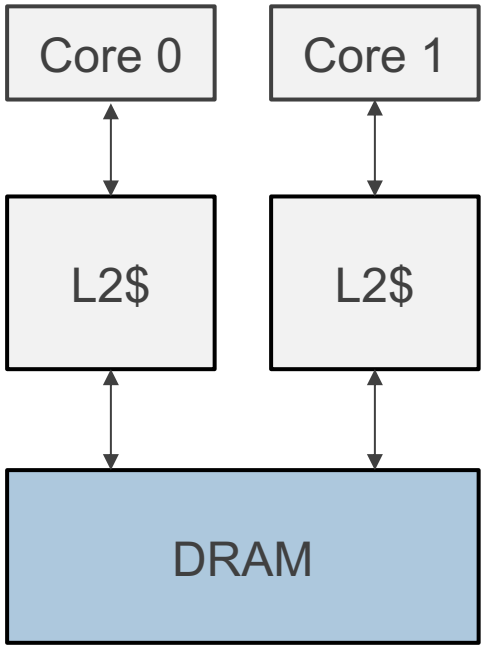
- **Capacity misses**
  - Increase cache capacity
  - Utilize cache capacity better
    - Only keep the blocks that will be likely referenced

- **Conflict misses**
  - Increase cache associativity

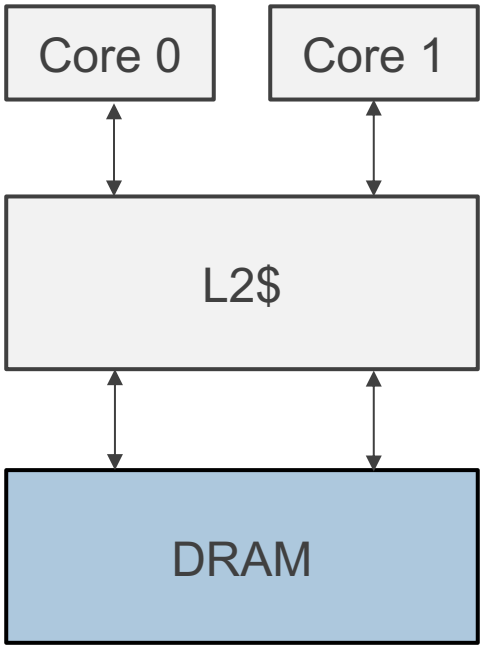
More Advanced Issues

Private vs Shared Caches

- Private cache: Cache belongs to one core (a shared block can be in multiple caches)
- Shared cache: Cache is shared by multiple cores



Private L2 Cache



Shared L2 Cache

Shared Caches Between Cores

- Advantages:
  - High effective capacity
  - Dynamic partitioning of available cache space
    - No fragmentation due to static partitioning
    - If one core does not utilize some space, another core can
  - Easier to maintain coherence (a cache block is in a single location)

- Disadvantages:
  - Slower access
  - Cores incur conflict misses due to the accesses from other cores
  - Guaranteeing fairness between cores is harder
    (in terms of capacity & bandwidth)

Blocking Cache or Non-Blocking Cache

- Does the CPU need to wait for a miss to be resolved?

- While a cache miss is being handled, should reads and writes to other addresses be allowed?
  - Essential in high-clock-rate ILP processors not to lose too many instruction opportunities during a cache miss
  - But, must take care of the ordering and dependency while memory operations are completing out-of-order

- Even in an In-Order pipeline, non-blocking write miss can be useful
  - The pipeline does not have to stall just because a SW hasn't completed all the way into the cache
  - But on a RAW-dependent LW, must either stall or forward

Non-Blocking Cache Using MSHR

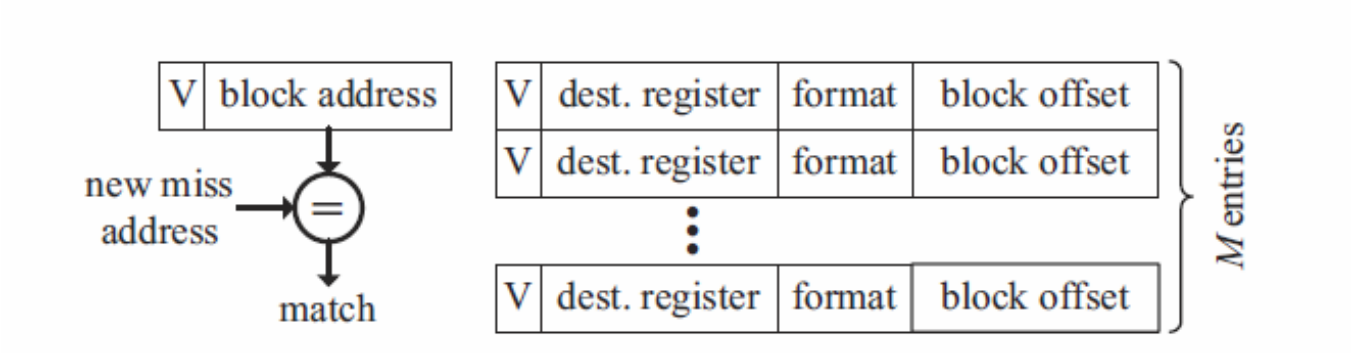
- Blocking Cache
  - Stop the pipeline on a cache miss
  - Serve only one cache miss a time

Non-Blocking Cache Using MSHR

- Blocking Cache
  - Stop the pipeline on a cache miss
  - Serve only one cache miss a time

- Non-blocking Cache (a.k.a. “lookup-free” Cache)
  - Enable multiple outstanding cache misses
  - Keep executing instructions while servicing a missing event
    - Why? To hide memory latency
  - Microarchitecture to make non-blocking cache
    - **MSHR (miss status/information holding registers)**
      - Each entry keeps track of status of outstanding memory requests to a cache miss being serviced
  - Miss definition
    - Primary miss : Miss to a cache line
    - Secondary miss : Another miss to the cache line, while the primary miss is still being serviced
    - Structural miss : All MSHRs being used

MSHR (Miss Status Handling Register)



- MSHR
  - (1) On a miss, allocate one entry per cache miss
  - (2) Prevent the pipeline from stalling due to other cache misses
  - (3) New cache miss will check against MSHR
    - On a match, we know this cache is being serviced due to miss
    - All instructions that missed to be notified when miss is completed

      Which destination register to be written?
  - (4) Pipeline is stopped only when all MSHRs are used

“Write” Can Raise Many Issues

- Let’s think about “store” instructions

- For store, we write the data only into the data cache (without changing main memory)

➔ Inconsistency between cache and memory

Write Policy

Write-Through vs Write-Back

- Q: When do we write the modified data to the next level memory?

- Write-Through Policy

- Write-Back Policy (more common)

Write Policy

Write-Through vs Write-Back

- Q: When do we write the modified data to the next level memory?

- Write-Through Policy
  - On a write-hit in L_i, update also L_{i+1}
  - + Simple policy
  - + All levels are up to date & consistent; search can be done only by looking at the lower-level hierarchy
  - -- more bandwidth intensive; no combining of writes

- Write-Back Policy (more common)

Write Policy

Write-Through vs Write-Back

- Q: When do we write the modified data to the next level memory?

- Write-Through Policy
  - On a write-hit in L_i, update also L_{i+1}
  - + Simple policy
  - + All levels are up to date & consistent; search can be done only by looking at the lower-level hierarchy
  - -- more bandwidth intensive; no combining of writes

- Write-Back Policy (more common)
  - On a write-hit in L_i, update **only** L_i with "**Dirty**" mark
  - When a cacheline is evicted, the line with "dirty=true" must be written back to lower-level L_{i+1}
  - Can combine multiple writes to the same block before eviction
    - Bandwidth & Energy Savings
    - Need a bit in the tag store indicating the block is "dirty/modified"

Write-Miss Policy

Write Allocate vs Write No Allocate

- Q: On a write miss, how will you allocate the block?

- Write Allocate (more common)
  - On a write miss in L_i, allocate the block in L_i
  - Write misses can be treated the same way as read misses
  - **Makes more sense for write-back cache**
    - On a write-miss, the block is brought in and updated there

- No Write Allocate
  - No cache allocation on a write miss in L_i
    - Writes are directly written to the memory
  - Conserves cache space if locality of writes is low
  - **Makes more sense for write-through cache**

Unified vs Split (I$ or D$)

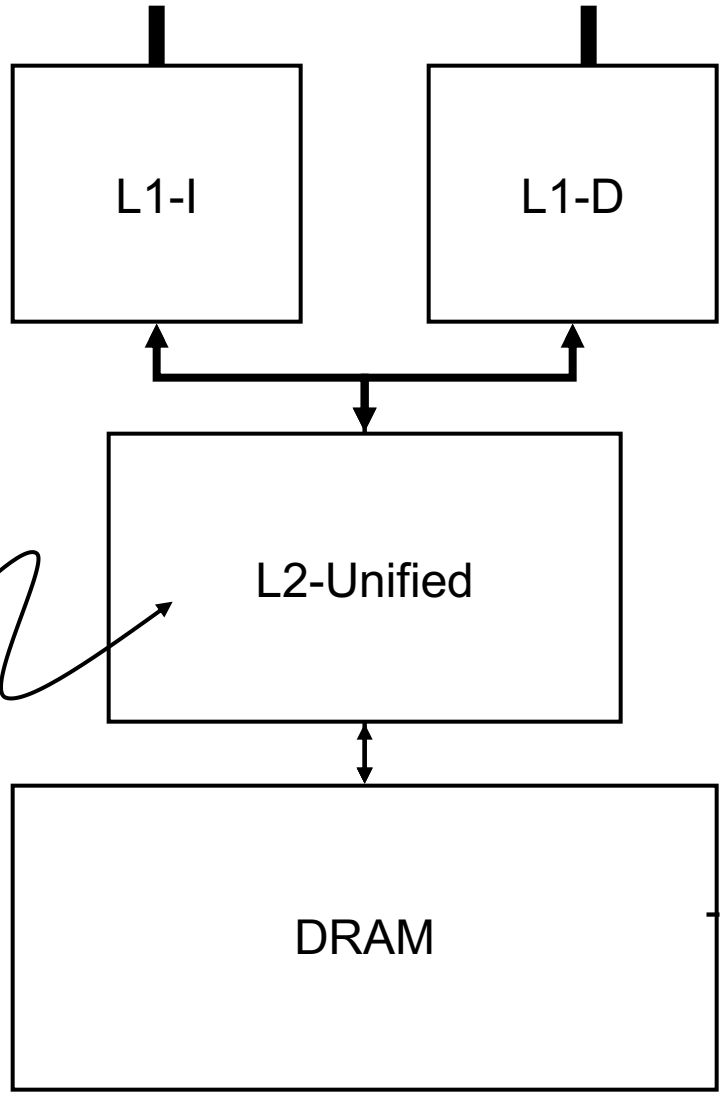
- Historical
  - "**Harvard Architecture**" refers to Aiken's Mark series at Harvard with separate storages for instruction and data
  - "**Princeton Architecture**" refers to von Neumann's unified storage for instruction and data
- Contemporary usage describes unified vs split "caches"

- Modern high-performance processors typically use split L1 caches for instruction and data
  - Instruction and data memory footprints typically disjoint
    - Instruction: smaller footprint, higher-spatial locality and read-only
  - Split L1 caches provide free doubled bandwidth, no-cross pollution, and separate design customizations

- L2 and L3 are typically unified

Multi-Level Caches

$T_i = t_i + m_i \cdot T_{i+1}$



- A few processor cycle latency
- Many GB/sec
(random access)

Intermediate hierarchy is a cheaper way to reduce T_1 than using a faster or bigger L1,

On-chip or off-chip?

- Hundreds of processor cycle latency
- ~GB/sec (sequential)

Inclusiveness

- Modern processors have multi-level cache hierarchies

- Design choice for cache inclusion
  - Inclusion: upper-level cache blocks always exist in the lower-level cache
  - Exclusion: upper-level cache blocks must not exist in the lower-level cache
  - Non-Inclusion : *may* contain the upper-level cache blocks

How To Implement Each Type of Caches?

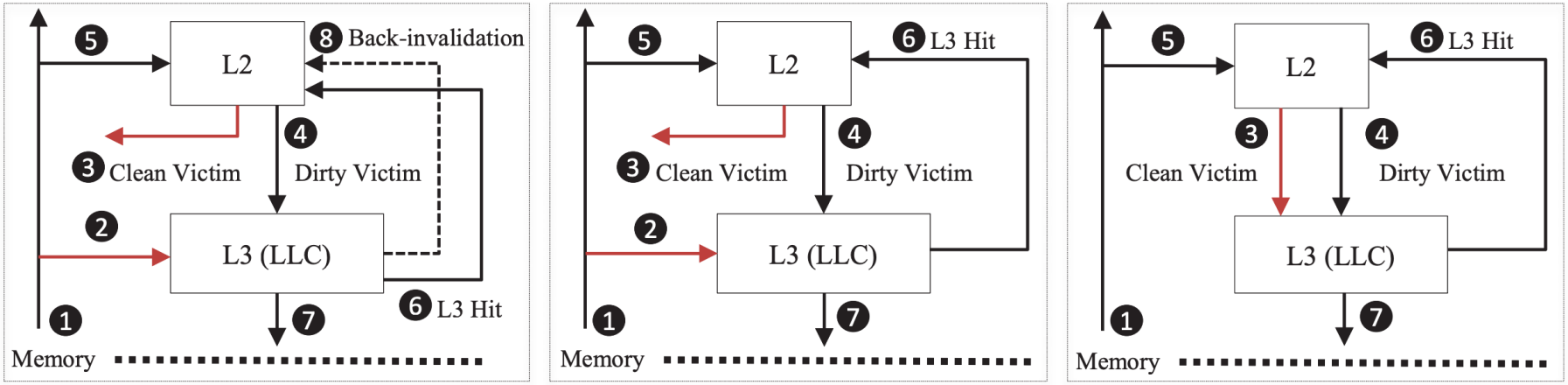


Figure 2. Block insertion/eviction flows for three different multi-level cache hierarchies.

- Exclusive Caches
  - Maximize the cache capacity in the hierarchy
  - The evicted block from level L_i must move to the lower level regardless of its dirtiness

- Inclusive Caches
  - Less effective capacity than exclusive caches due to duplicated cache blocks
  - Low on-chip traffic; better for cache coherence

Prefetching

- Goal: Hide the memory latency
- Idea: Fetch the data before it is needed (i.e., pre-fetch)

Prefetching

- Goal: Hide the memory latency
- Idea: Fetch the data before it is needed (i.e., pre-fetch)

- Why?
  - Memory latency is high. If we can prefetch accurately and timely we can reduce both the miss rate and the miss latency
    - Can eliminate compulsory cache misses!

Prefetching

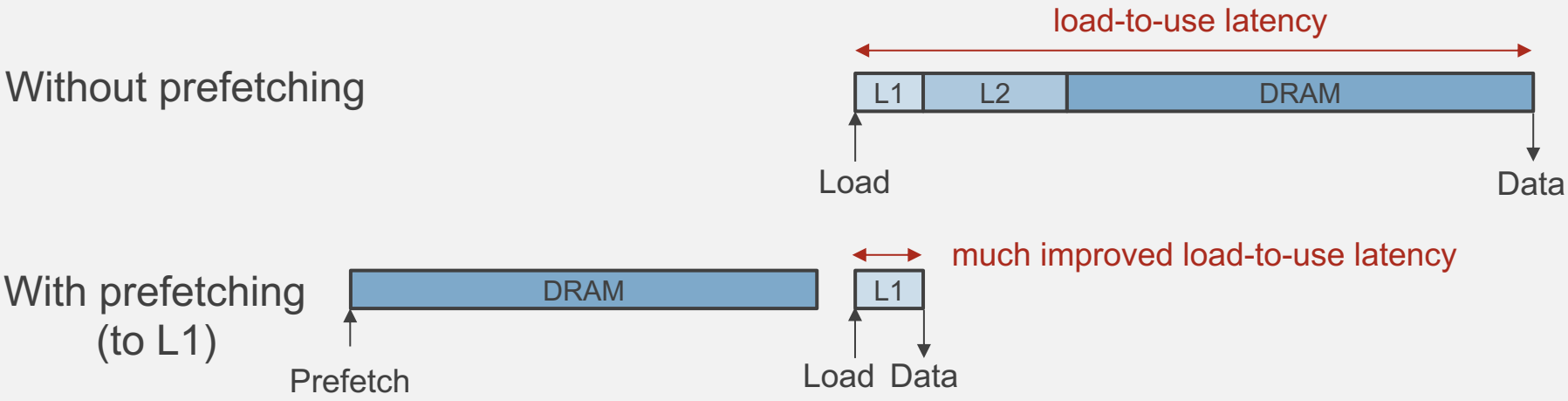
- Goal: Hide the memory latency
- Idea: Fetch the data before it is needed (i.e., pre-fetch)

- Why?
  - Memory latency is high. If we can prefetch accurately and timely we can reduce both the miss rate and the miss latency
    - Can eliminate compulsory cache misses!

- Need to predict which address will be needed in the future
  - Works if programs have predictable address access patterns
  - What if prediction is wrong? Is it okay?

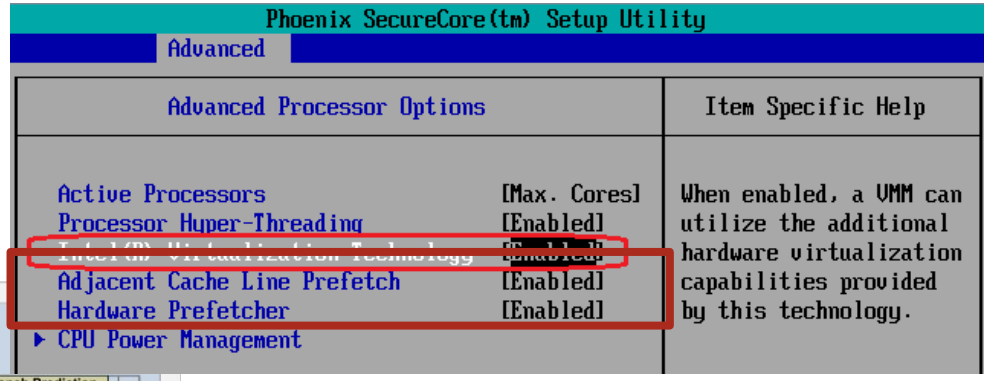
- Prefetching can be done by
  - Hardware/Compiler/Programmer

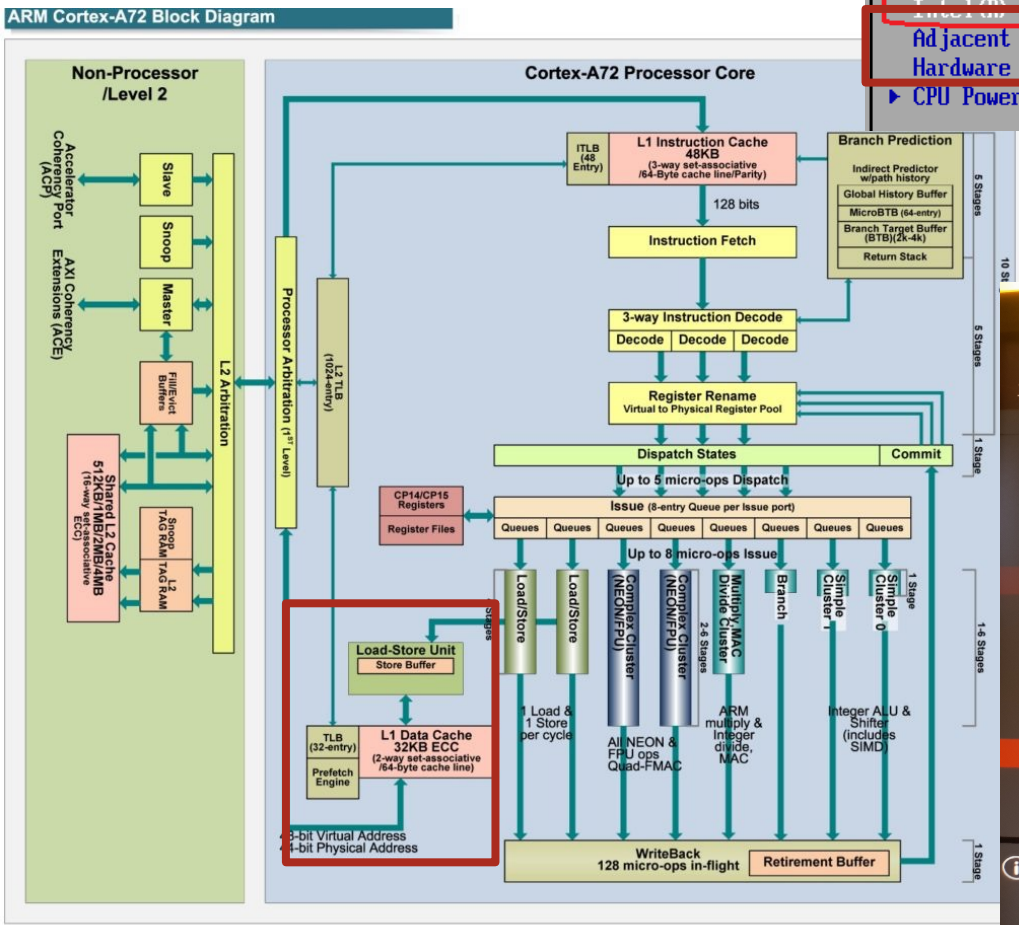
Prefetching Performance

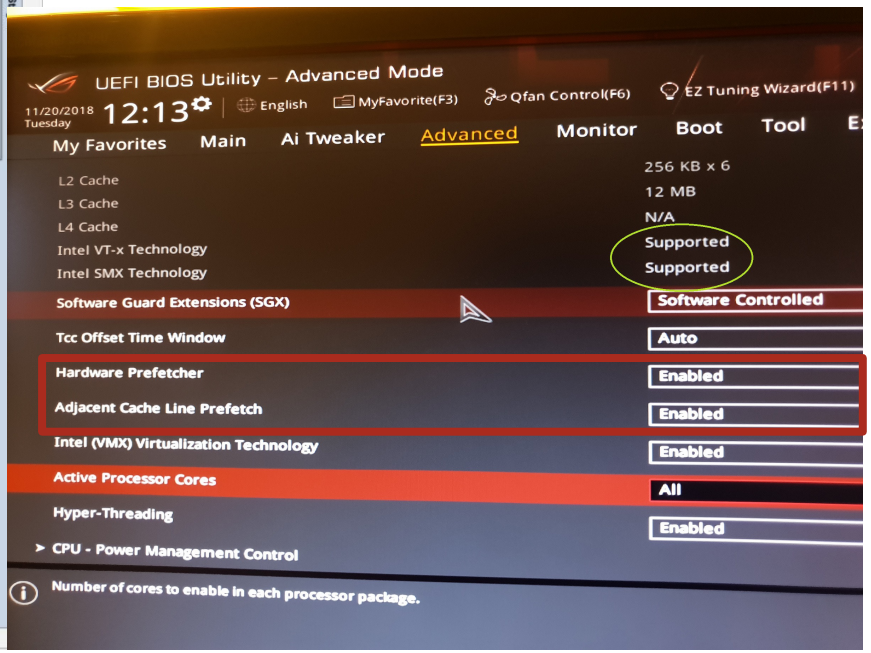


- Good prefetcher?
  - Accuracy (used prefetches / sent prefetches)
  - Coverage (useful prefetches to all misses)
  - Timeliness (on-time? late? early?)
- Things to consider...
  - Bandwidth consumption
    - Memory bandwidth consumed with prefetcher; may utilize idle bus bandwidth
  - Cache pollution
    - Extra demand misses due to prefetch placement in cache

Hardware Prefetcher







Average Memory Access Time (AMAT)

- The time to access memory on average

  AMAT = (Hit Time) + (Miss Rate) x (Miss Penalty)

- For multi-level memory hierarchy, AMAT can be extended by replacing miss penalty with AMAT for the next level
  - $AMAT_i = (Hit\ Time)_i + (Miss\ Rate)_i \times (Miss\ Penalty)_{i+1}$
  - $T_i = t_i + m_i \cdot T_{i+1}$

- Example)
  - L1$: hit latency: 1 cycle, hit rate 50%
  - L2$: hit latency: 10 cycles, hit rate 75%
  - L3$: hit latency: 100 cycles, hit rate 90%
  - Main memory: 1000 cycles
  - Average Memory Access Time?
    1 + 0.5 x (10 + 0.25 x (100 + 0.1x1000)) = 31 cycles

Summary

- Terminology
  - Temporal Locality vs Spatial Locality
  - Cache Block (cacheline)
  - Memory Address = TAG + INDEX + OFFSET
  - Associativity (Direct-Mapped, Set-Associative, Fully Associative)
  - Replacement policy
  - Write-through vs Write-back
  - Write allocate vs Write no allocate
  - Cache Inclusion
  - Prefetching
  - Unified vs Split
  - Average Memory Access Time (AMAT)
  - TAG Store ➔ valid bit + (dirty bit) + tag + bits for replacement policy

Computer Architecture

Memory: Cache (Part II)

Chap 5.1-5.4, 5.8-5.9

Jaewoong Sim

Electrical and Computer Engineering

Seoul National University