Computer Architecture

Lecture 3b: Memory Hierarchy and Caches

Prof. Onur Mutlu
ETH Zürich

Fall 2018

26 September 2018

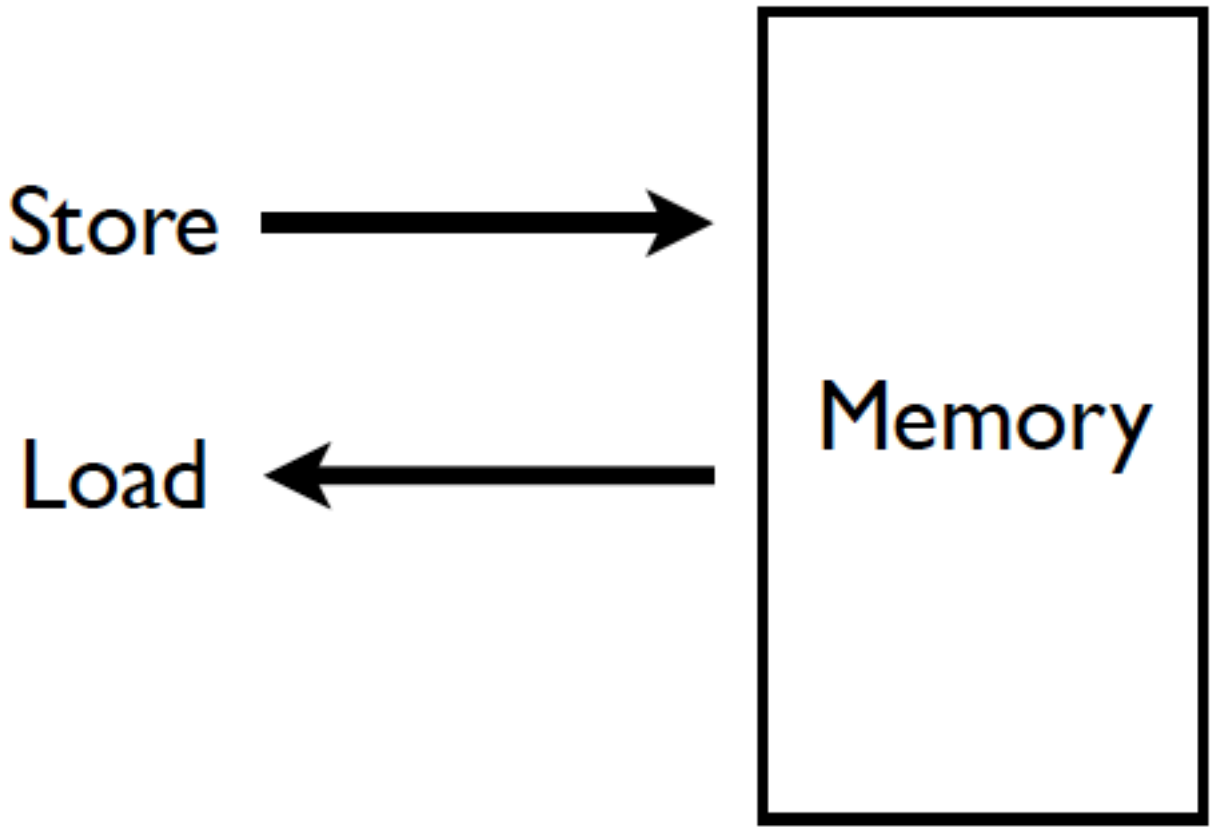
Review Cache Lectures (from Spring 2018)

- Memory Organization and Technology (Lecture 23b)
  - https://www.youtube.com/watch?v=rvBdJ1ZLo2M
- Memory Hierarchy and Caches (Lecture 24)
  - https://www.youtube.com/watch?v=sweCA3836C0
- More Caches (Lecture 25a)
  - https://www.youtube.com/watch?v=kMUZKjaPNWo
- Virtual Memory (Lecture 25b)
  - https://www.youtube.com/watch?v=na-JL1nVTSU

Optional Readings for Today

- Memory Hierarchy and Caches
- Cache chapters from P&H: 5.1-5.3
- Memory/cache chapters from Hamacher+: 8.1-8.7
- An early cache paper by Maurice Wilkes
  - Wilkes, "Slave Memories and Dynamic Storage Allocation," IEEE Trans. On Electronic Computers, 1965.
- We already covered these in Digital Circuits so we will go through them quickly and get to advanced topics

Memory (Programmer's View)



Abstraction: Virtual vs. Physical Memory

- Programmer sees virtual memory
  - Can assume the memory is "infinite"
- Reality: Physical memory size is much smaller than what the programmer assumes
- The system (system software + hardware, cooperatively) maps virtual memory addresses to physical memory
  - The system automatically manages the physical memory space transparently to the programmer

+ Programmer does not need to know the physical size of memory nor manage it → A small physical memory can appear as a huge one to the programmer → Life is easier for the programmer

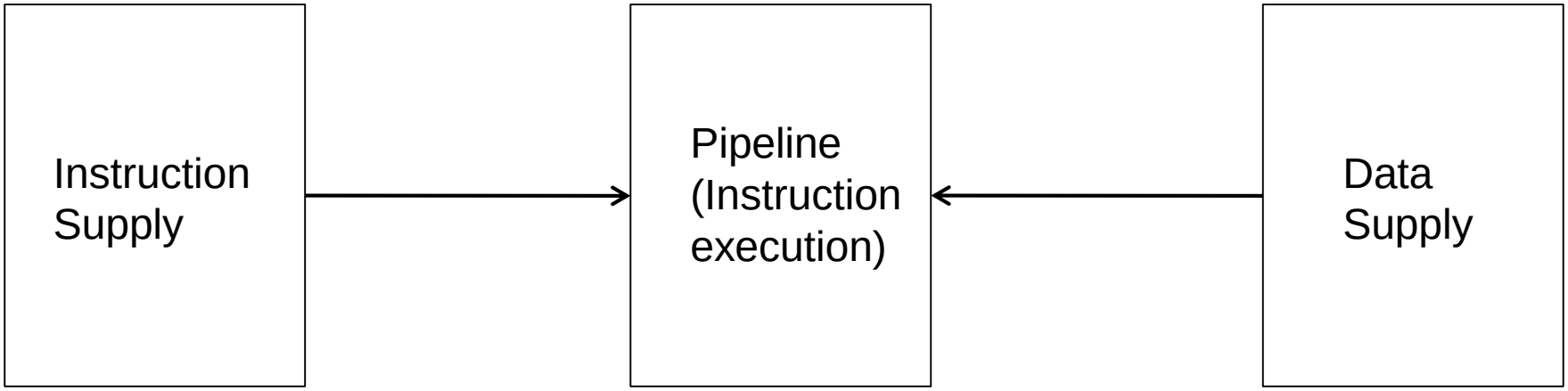
-- More complex system software and architecture

A classic example of the programmer/(micro)architect tradeoff

(Physical) Memory System

- You need a larger level of storage to manage a small amount of physical memory automatically
  → Physical memory has a backing store: disk
- We will first start with the physical memory system
- For now, ignore the virtual→physical indirection
- We will get back to it when the needs of virtual memory start complicating the design of physical memory…

Idealism



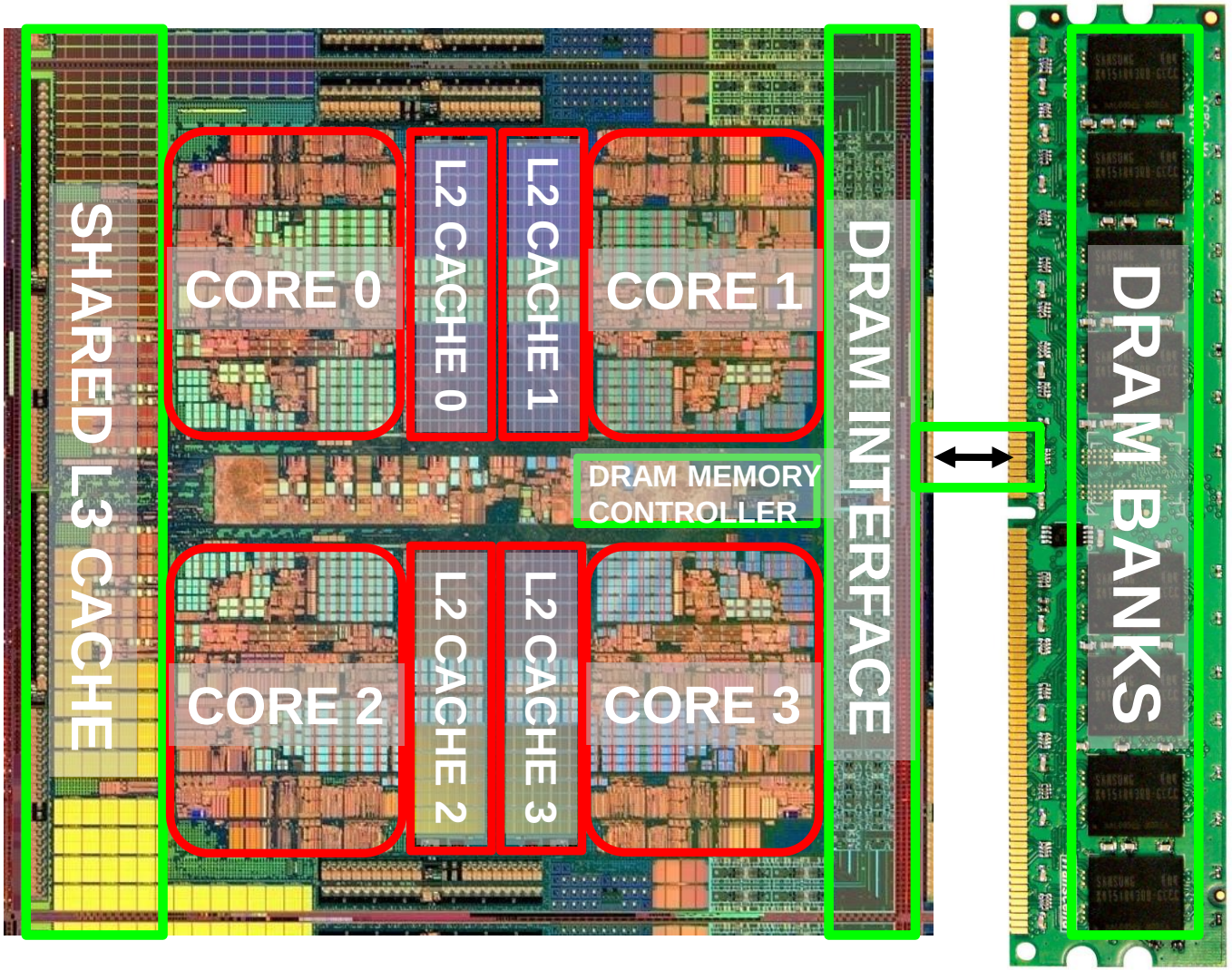
- Zero latency access
- Infinite capacity
- Zero cost
- Perfect control flow

- No pipeline stalls
- Perfect data flow (reg/memory dependencies)
- Zero-cycle interconnect (operand communication)
- Enough functional units
- Zero latency compute

- Zero latency access
- Infinite capacity
- Infinite bandwidth
- Zero cost

The Memory Hierarchy

Memory in a Modern System



Ideal Memory

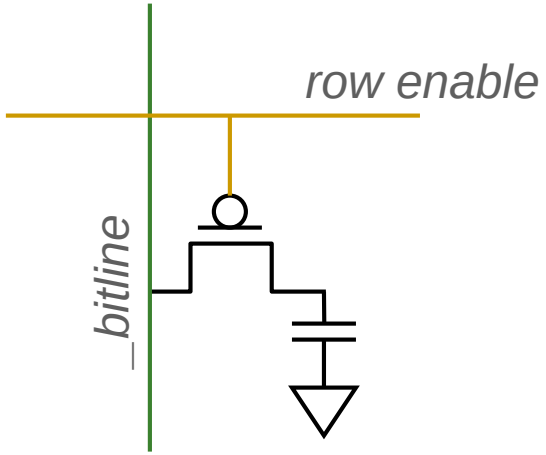
- Zero access time (latency)
- Infinite capacity
- Zero cost
- Infinite bandwidth (to support multiple accesses in parallel)

The Problem

- Ideal memory's requirements oppose each other
- Bigger is slower
  - Bigger → Takes longer to determine the location
- Faster is more expensive
  - Memory technology: SRAM vs. DRAM vs. Disk vs. Tape
- Higher bandwidth is more expensive
  - Need more banks, more ports, higher frequency, or faster technology

Memory Technology: DRAM

- Dynamic random access memory
- Capacitor charge state indicates stored value
  - Whether the capacitor is charged or discharged indicates storage of 1 or 0
  - 1 capacitor
  - 1 access transistor

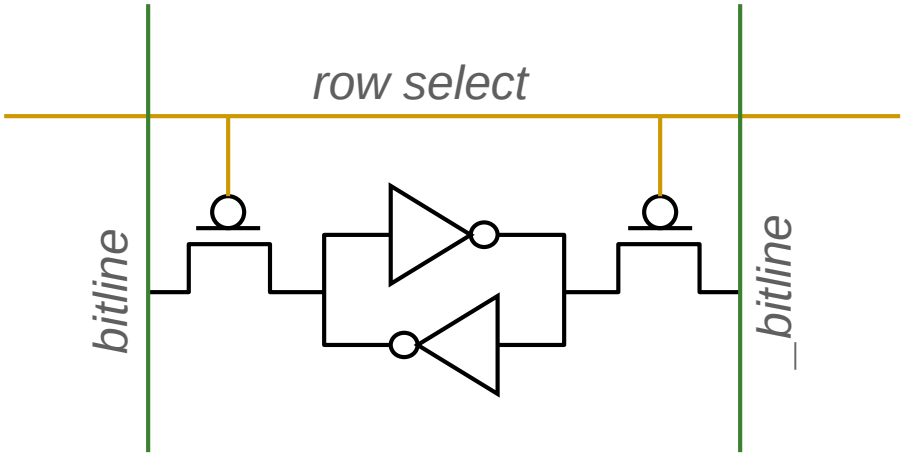


- Capacitor leaks through the RC path
  - DRAM cell loses charge over time
  - DRAM cell needs to be refreshed

  - Read Liu et al., "RAIDR: Retention-aware Intelligent DRAM Refresh," ISCA 2012.

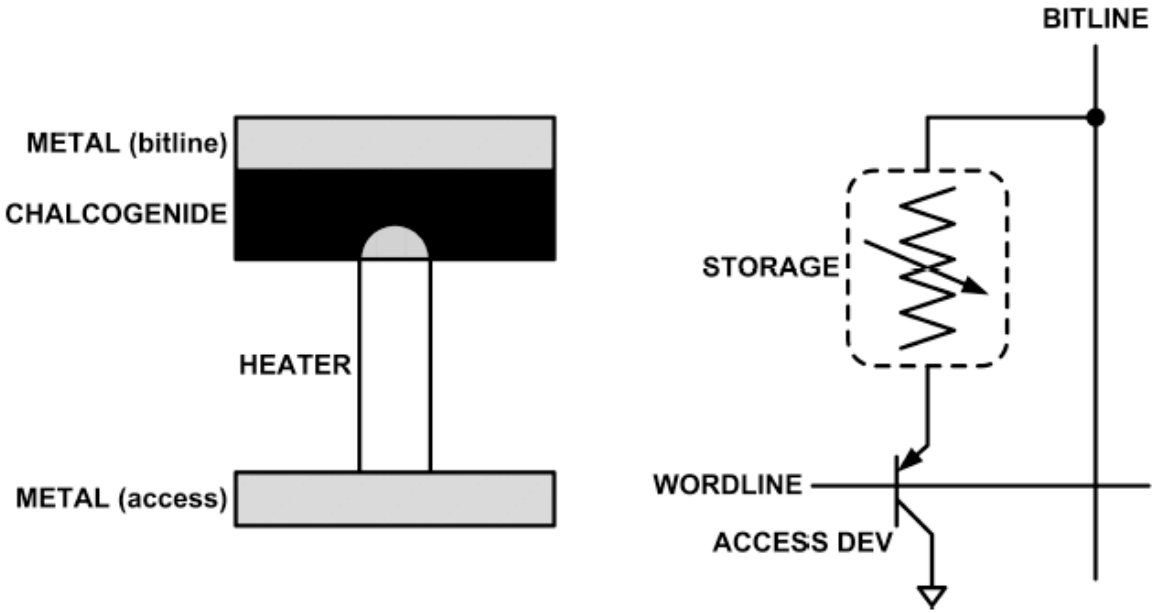
Memory Technology: SRAM

- Static random access memory
- Two cross coupled inverters store a single bit
  - Feedback path enables the stored value to persist in the "cell"
  - 4 transistors for storage
  - 2 transistors for access



An Aside: Phase Change Memory

- Phase change material (chalcogenide glass) exists in two states:
  - Amorphous: Low optical reflexivity and high electrical resistivity
  - Crystalline: High optical reflexivity and low electrical resistivity

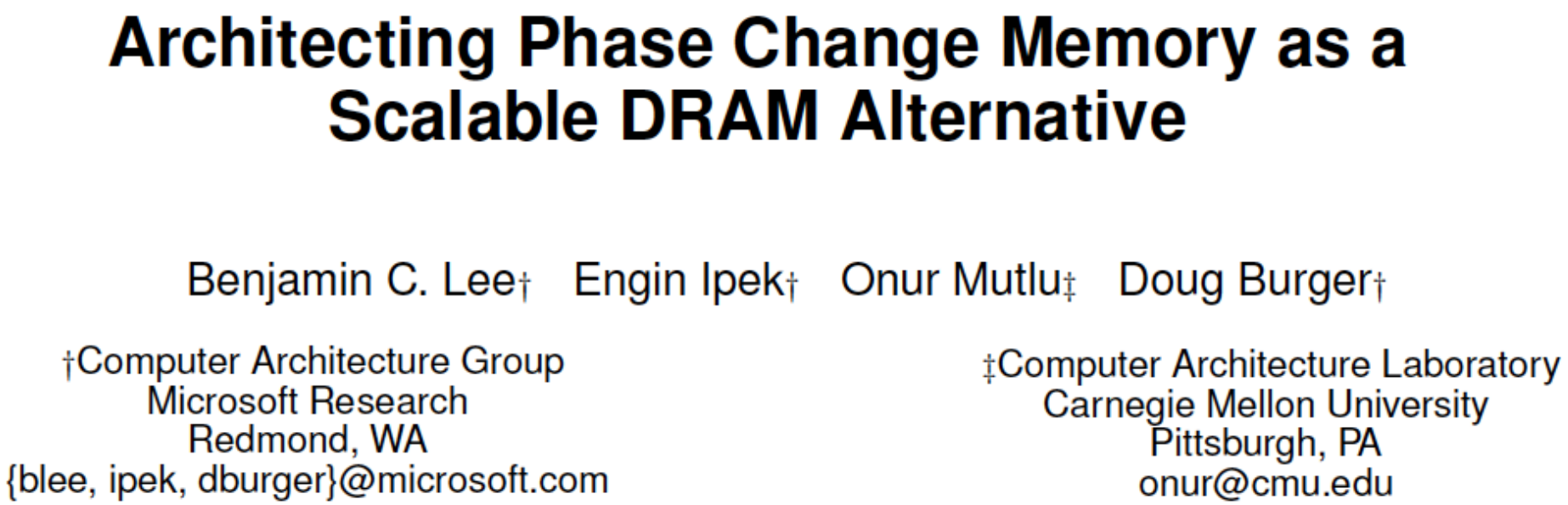


PCM is resistive memory: High resistance (0), Low resistance (1)

Lee, Ipek, Mutlu, Burger, "Architecting Phase Change Memory as a Scalable DRAM Alternative," ISCA 2009.

Reading: PCM As Main Memory

- Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger, **"Architecting Phase Change Memory as a Scalable DRAM Alternative"**
  *Proceedings of the 36th International Symposium on Computer Architecture (**ISCA**)*, pages 2-13, Austin, TX, June 2009. Slides (pdf)



Architecting Phase Change Memory as a Scalable DRAM Alternative

Benjamin C. Lee† Engin Ipek† Onur Mutlu‡ Doug Burger†

†Computer Architecture Group
Microsoft Research
Redmond, WA
{blee, ipek, dburger}@microsoft.com

‡Computer Architecture Laboratory
Carnegie Mellon University
Pittsburgh, PA
onur@cmu.edu

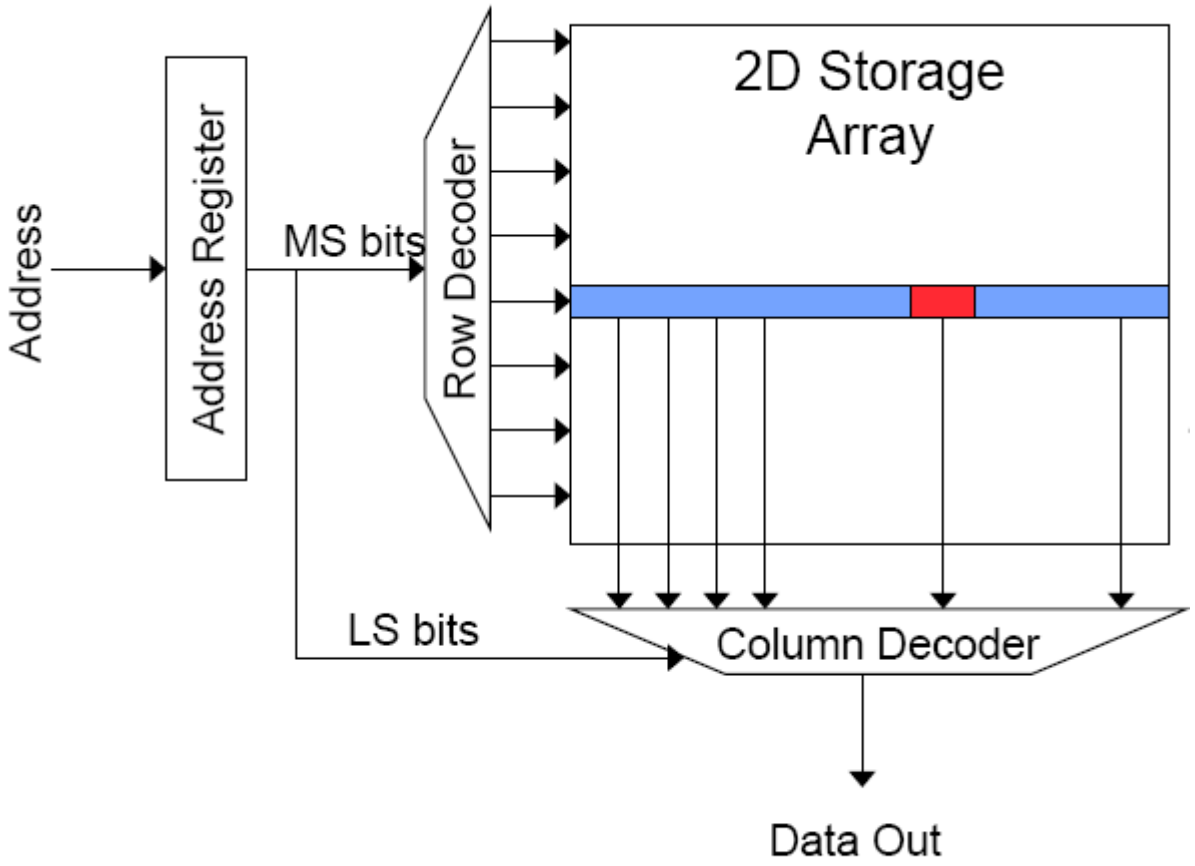
Reading: More on PCM As Main Memory

- Benjamin C. Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger, **"Phase Change Technology and the Future of Main Memory"** *IEEE Micro*, *Special Issue: Micro's Top Picks from 2009 Computer Architecture Conferences (**MICRO TOP PICKS**)*, Vol. 30, No. 1, pages 60-70, January/February 2010.

Memory Bank: A Fundamental Concept

- Interleaving (banking)
  - Problem: a single monolithic memory array takes long to access and does not enable multiple accesses in parallel
  - Goal: Reduce the latency of memory array access and enable multiple accesses in parallel
  - Idea: Divide the array into multiple banks that can be accessed independently (in the same cycle or in consecutive cycles)
    - Each bank is smaller than the entire memory storage
    - Accesses to different banks can be overlapped
  - An issue: How do you map data to different banks? (i.e., how do you interleave data across banks?)

Memory Bank Organization and Operation



- Read access sequence:

1. Decode row address & drive word-lines

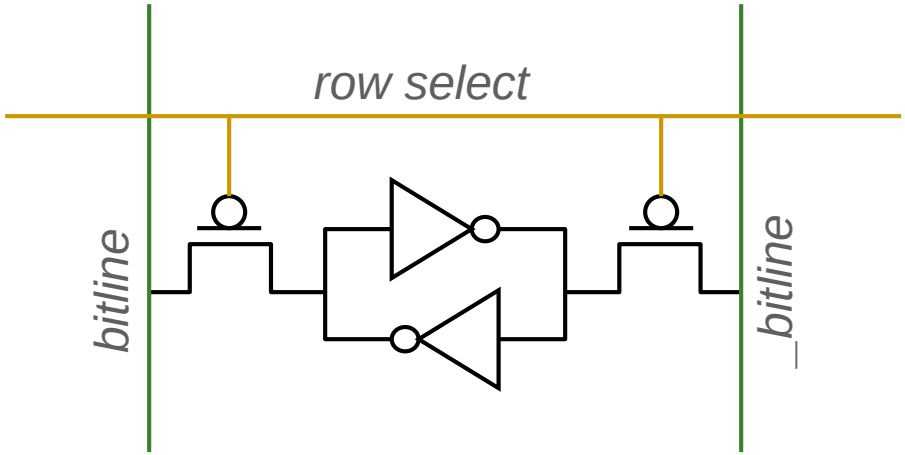
2. Selected bits drive bit-lines
   - Entire row read

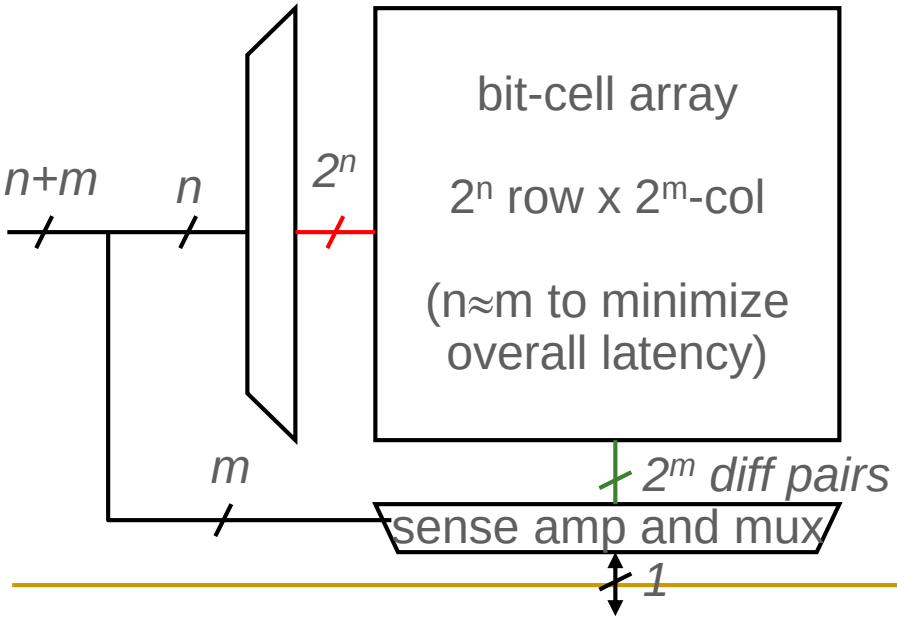
3. Amplify row data

4. Decode column address & select subset of row
   - Send to output

5. Precharge bit-lines
   - For next access

SRAM (Static Random Access Memory)





Read Sequence

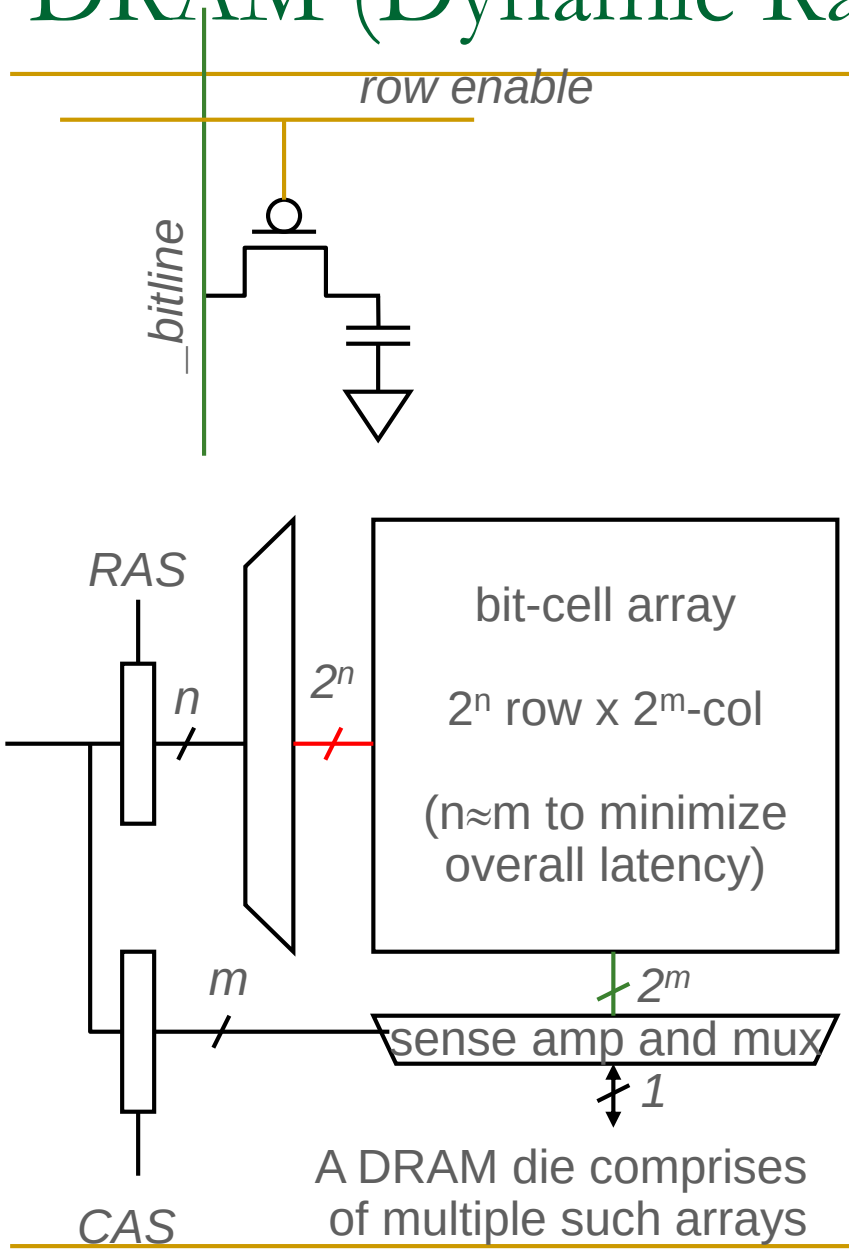
1. address decode
2. drive row select
3. selected bit-cells drive bitlines
   (entire row is read together)
4. differential sensing and column select
   (data is ready)
5. precharge all bitlines
   (for next read or write)

Access latency dominated by steps 2 and 3

Cycling time dominated by steps 2, 3 and 5

- step 2 proportional to 2^m
- step 3 and 5 proportional to 2^n

DRAM (Dynamic Random Access Memory)



Bits stored as charges on node capacitance (non-restorative)

- bit cell loses charge when read
- bit cell loses charge over time

Read Sequence

1~3 same as SRAM

4. a "flip-flopping" sense amp amplifies and regenerates the bitline, data bit is mux'ed out

5. precharge all bitlines

Destructive reads

Charge loss over time

Refresh: A DRAM controller must periodically read each row within the allowed refresh time (10s of ms) such that charge is restored

DRAM vs. SRAM

- DRAM
  - Slower access (capacitor)
  - Higher density (1T 1C cell)
  - Lower cost
  - Requires refresh (power, performance, circuitry)
  - Manufacturing requires putting capacitor and logic together

- SRAM
  - Faster access (no capacitor)
  - Lower density (6T cell)
  - Higher cost
  - No need for refresh
  - Manufacturing compatible with logic process (no capacitor)

The Problem

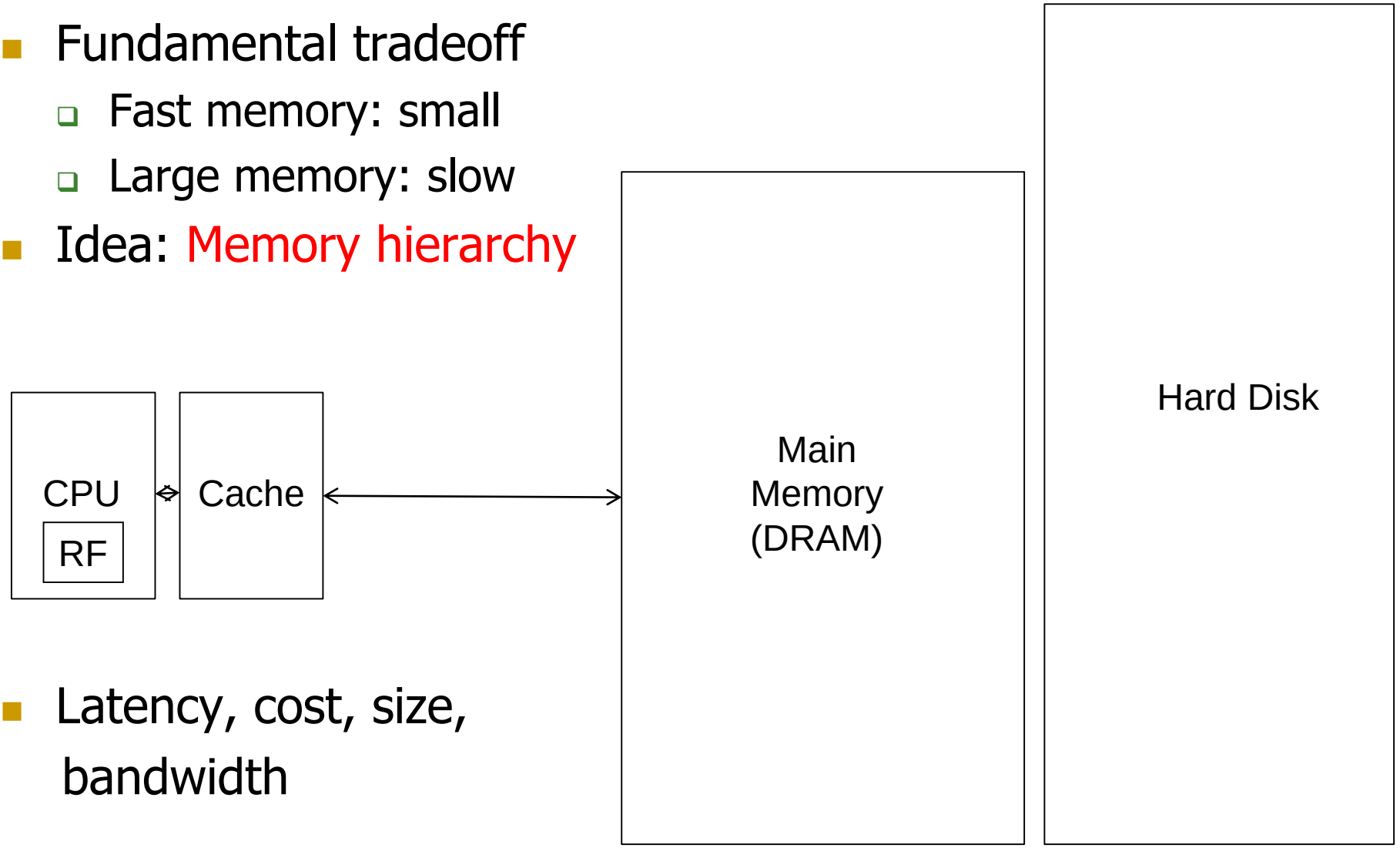
- Bigger is slower
  - SRAM, 512 Bytes, sub-nanosec
  - SRAM, KByte~MByte, ~nanosec
  - DRAM, Gigabyte, ~50 nanosec
  - Hard Disk, Terabyte, ~10 millisec
- Faster is more expensive (dollars and chip area)
  - SRAM, < 10$ per Megabyte
  - DRAM, < 1$ per Megabyte
  - Hard Disk < 1$ per Gigabyte
  - These sample values (circa ~2011) scale with time
- Other technologies have their place as well
  - Flash memory, PC-RAM, MRAM, RRAM (not mature yet)

Why Memory Hierarchy?

- We want both fast and large
- But we cannot achieve both with a single level of memory
- Idea: Have multiple levels of storage (progressively bigger and slower as the levels are farther from the processor) and ensure most of the data the processor needs is kept in the fast(er) level(s)

Memory Hierarchy

- Fundamental tradeoff
  - Fast memory: small
  - Large memory: slow
- Idea: Memory hierarchy



- Latency, cost, size, bandwidth

Locality

- One's recent past is a very good predictor of his/her near future.

- Temporal Locality: If you just did something, it is very likely that you will do the same thing again soon
  - since you are here today, there is a good chance you will be here again and again regularly

- Spatial Locality: If you did something, it is very likely you will do something similar/related (in space)
  - every time I find you in this room, you are probably sitting close to the same people

Memory Locality

- A "typical" program has a lot of locality in memory references
  - typical programs are composed of "loops"

- Temporal: A program tends to reference the same memory location many times and all within a small window of time

- Spatial: A program tends to reference a cluster of memory locations at a time
  - most notable examples:
    - 1. instruction memory references
    - 2. array/data structure references

Caching Basics: Exploit Temporal Locality

- Idea: Store recently accessed data in automatically managed fast memory (called cache)
- Anticipation: the data will be accessed again soon

- Temporal locality principle
  - Recently accessed data will be again accessed in the near future
  - This is what Maurice Wilkes had in mind:
    - Wilkes, "Slave Memories and Dynamic Storage Allocation," IEEE Trans. On Electronic Computers, 1965.
    - "The use is discussed of a fast core memory of, say 32000 words as a slave to a slower core memory of, say, one million words in such a way that in practical cases the effective access time is nearer that of the fast memory than that of the slow memory."

Caching Basics: Exploit Spatial Locality

- Idea: Store addresses adjacent to the recently accessed one in automatically managed fast memory
  - Logically divide memory into equal size blocks
  - Fetch to cache the accessed block in its entirety
- Anticipation: nearby data will be accessed soon

- Spatial locality principle
  - Nearby data in memory will be accessed in the near future
    - E.g., sequential instruction access, array traversal
  - This is what IBM 360/85 implemented
    - 16 Kbyte cache with 64 byte blocks
    - Liptay, "Structural aspects of the System/360 Model 85 II: the cache," IBM Systems Journal, 1968.

A Note on Manual vs. Automatic Management

- Manual: Programmer manages data movement across levels

  -- too painful for programmers on substantial programs

  - "core" vs "drum" memory in the 50's
  - still done in some embedded processors (on-chip scratch pad SRAM in lieu of a cache)

- Automatic: Hardware manages data movement across levels, transparently to the programmer

  ++ programmer's life is easier

  - simple heuristic: keep most recently used items in cache
  - the average programmer doesn't need to know about it
    - You don't need to know how big the cache is and how it works to write a "correct" program! (What if you want a "fast" program?)

Automatic Management in Memory Hierarchy

- Wilkes, "Slave Memories and Dynamic Storage Allocation," IEEE Trans. On Electronic Computers, 1965.

Slave Memories and Dynamic Storage Allocation

M. V. WILKES

SUMMARY

The use is discussed of a fast core memory of, say, 32 000 words as a slave to a slower core memory of, say, one million words in such a way that in practical cases the effective access time is nearer that of the fast memory than that of the slow memory.

- "By a slave memory I mean one which automatically accumulates to itself words that come from a slower main memory, and keeps them available for subsequent use without it being necessary for the penalty of main memory access to be incurred again."

Historical Aside: Other Cache Papers

- Fotheringham, "Dynamic Storage Allocation in the Atlas Computer, Including an Automatic Use of a Backing Store," CACM 1961.
  - http://dl.acm.org/citation.cfm?id=366800

- Bloom, Cohen, Porter, "Considerations in the Design of a Computer with High Logic-to-Memory Speed Ratio," AIEE Gigacycle Computing Systems Winter Meeting, Jan. 1962.

Cache in 1962 (Bloom, Cohen, Porter)

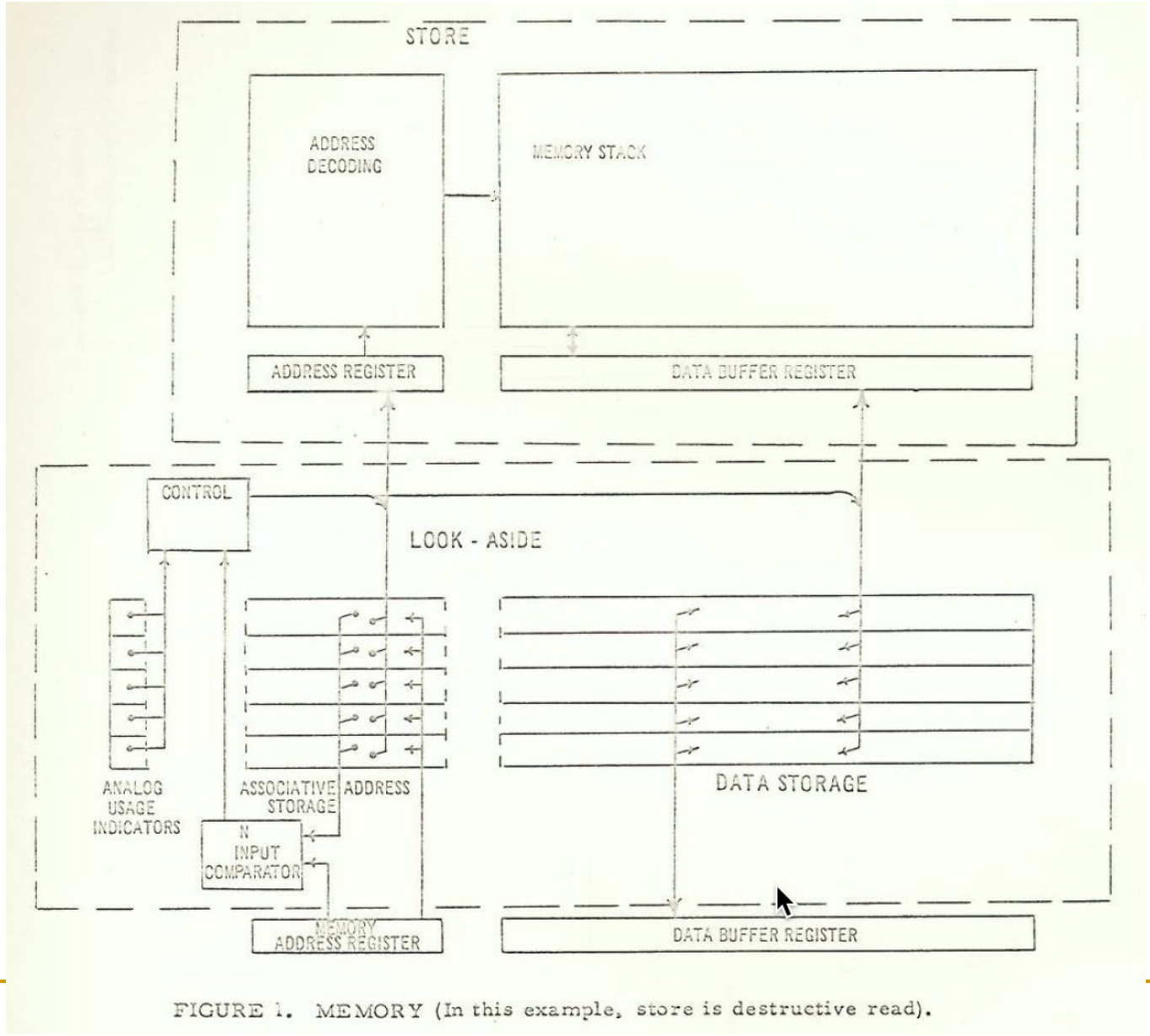
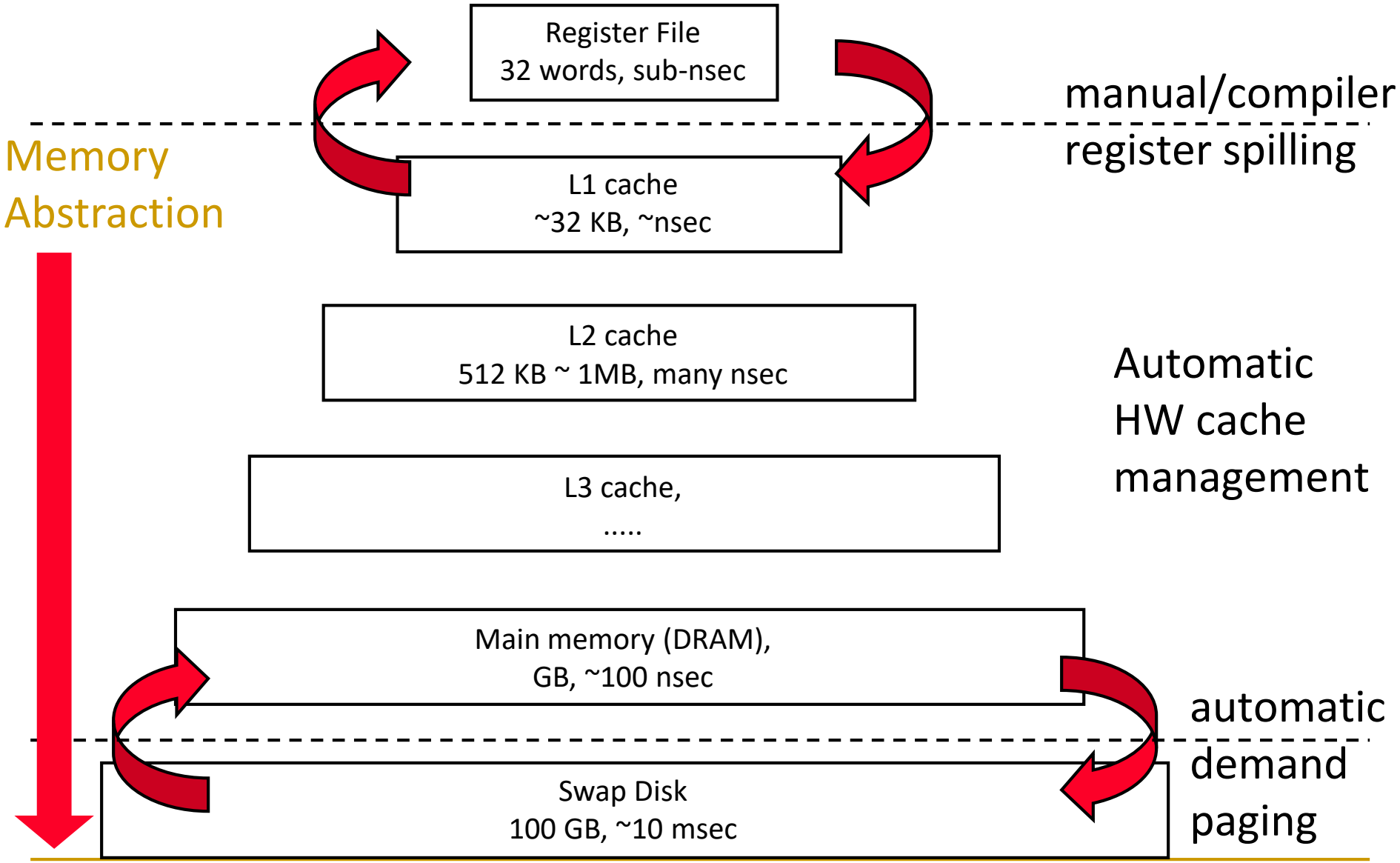


FIGURE 1. MEMORY (In this example, store is destructive read).

A Modern Memory Hierarchy



Review: Hierarchical Latency Analysis

- For a given memory hierarchy level i it has a technology-intrinsic access time of t_i, The perceived access time T_i is longer than t_i
- Except for the outer-most hierarchy, when looking for a given address there is
  - a chance (hit-rate h_i) you "hit" and access time is t_i
  - a chance (miss-rate m_i) you "miss" and access time $t_i + T_{i+1}$
  - $h_i + m_i = 1$
- Thus

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

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

h_i and m_i are defined to be the hit-rate and miss-rate of just the references that missed at L_{i-1}

Review: Hierarchy Design Considerations

- Recursive latency equation

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

- The goal: achieve desired T_1 within allowed cost
- $T_i \approx t_i$ is desirable

- Keep m_i low
  - increasing capacity C_i lowers m_i, but beware of increasing t_i
  - lower m_i by smarter management (replacement::anticipate what you don't need, prefetching::anticipate what you will need)

- Keep T_{i+1} low
  - faster lower hierarchies, but beware of increasing cost
  - introduce intermediate hierarchies as a compromise

Review: Intel Pentium 4 Example

- 90nm P4, 3.6 GHz
- L1 D-cache
  - C_1 = 16K
  - t_1 = 4 cyc int / 9 cycle fp
- L2 D-cache
  - C_2 =1024 KB
  - t_2 = 18 cyc int / 18 cyc fp
- Main memory
  - t_3 = ~ 50ns or 180 cyc
- Notice
  - best case latency is not 1
  - worst case access latencies are into 500+ cycles

if m_1=0.1, m_2=0.1
T_1=7.6, T_2=36

if m_1=0.01, m_2=0.01
T_1=4.2, T_2=19.8

if m_1=0.05, m_2=0.01
T_1=5.00, T_2=19.8

if m_1=0.01, m_2=0.50
T_1=5.08, T_2=108

Cache Basics and Operation

Cache

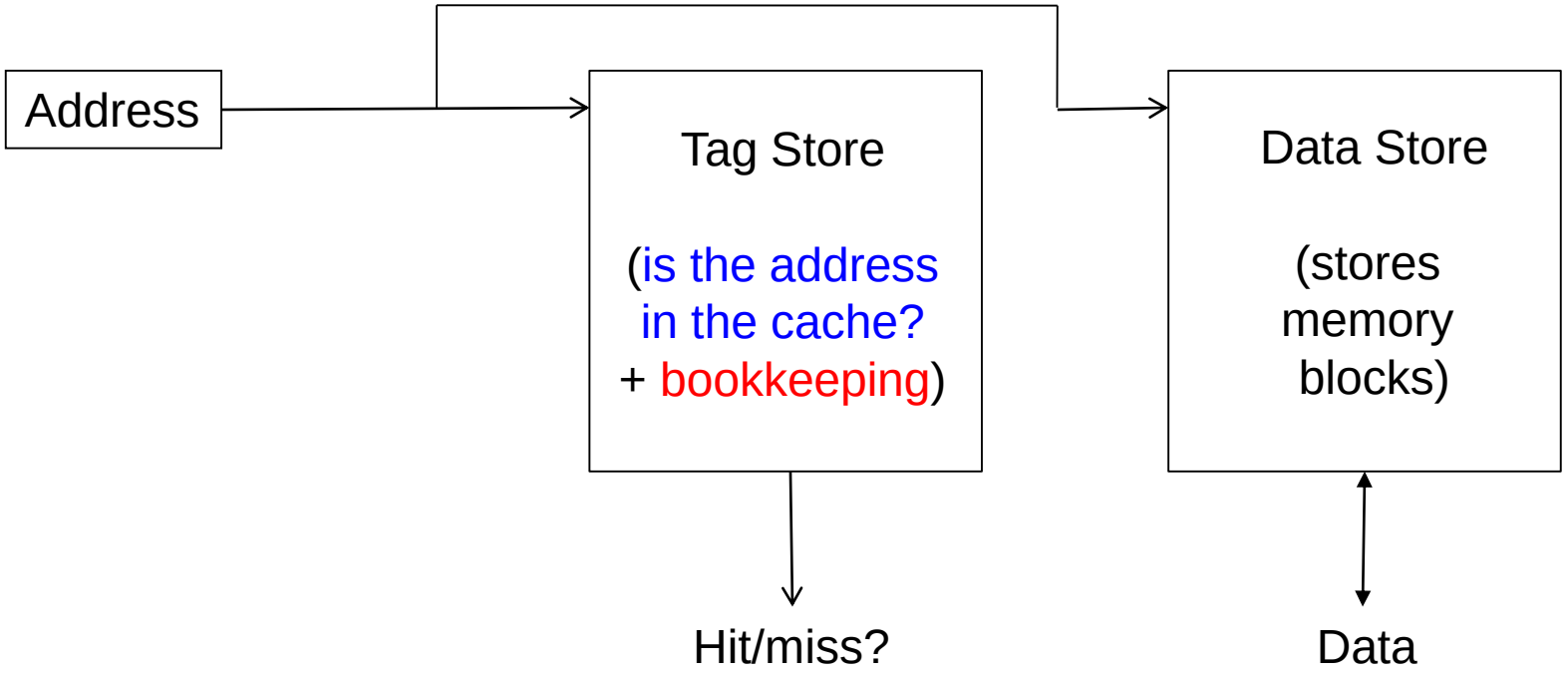
- Generically, any structure that "memoizes" frequently used results to avoid repeating the long-latency operations required to reproduce the results from scratch, e.g. a web cache

- Most commonly in the on-die context: an automatically-managed memory hierarchy based on SRAM
  - memoize in SRAM the most frequently accessed DRAM memory locations to avoid repeatedly paying for the DRAM access latency

Caching Basics

- Block (line): Unit of storage in the cache
  - Memory is logically divided into cache blocks that map to locations in the cache
- On a reference:
  - HIT: If in cache, use cached data instead of accessing memory
  - MISS: If not in cache, bring block into cache
    - Maybe have to kick something else out to do it
- Some important cache design decisions
  - Placement: where and how to place/find a block in cache?
  - Replacement: what data to remove to make room in cache?
  - Granularity of management: large or small blocks? Subblocks?
  - Write policy: what do we do about writes?
  - Instructions/data: do we treat them separately?

Cache Abstraction and Metrics



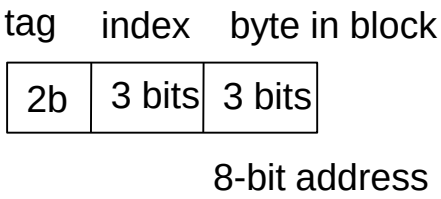
- Cache hit rate = (# hits) / (# hits + # misses) = (# hits) / (# accesses)
- Average memory access time (AMAT)
  = (hit-rate * hit-latency) + (miss-rate * miss-latency)
- Aside: *Can reducing AMAT reduce performance?*

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

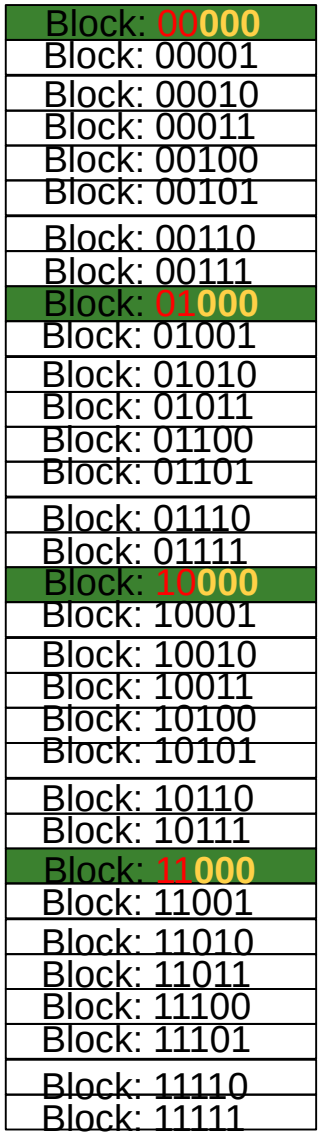


- Cache access:

  1) index into the tag and data stores with index bits in address
  2) check valid bit in tag store
  3) compare tag bits in address with the stored tag in tag store

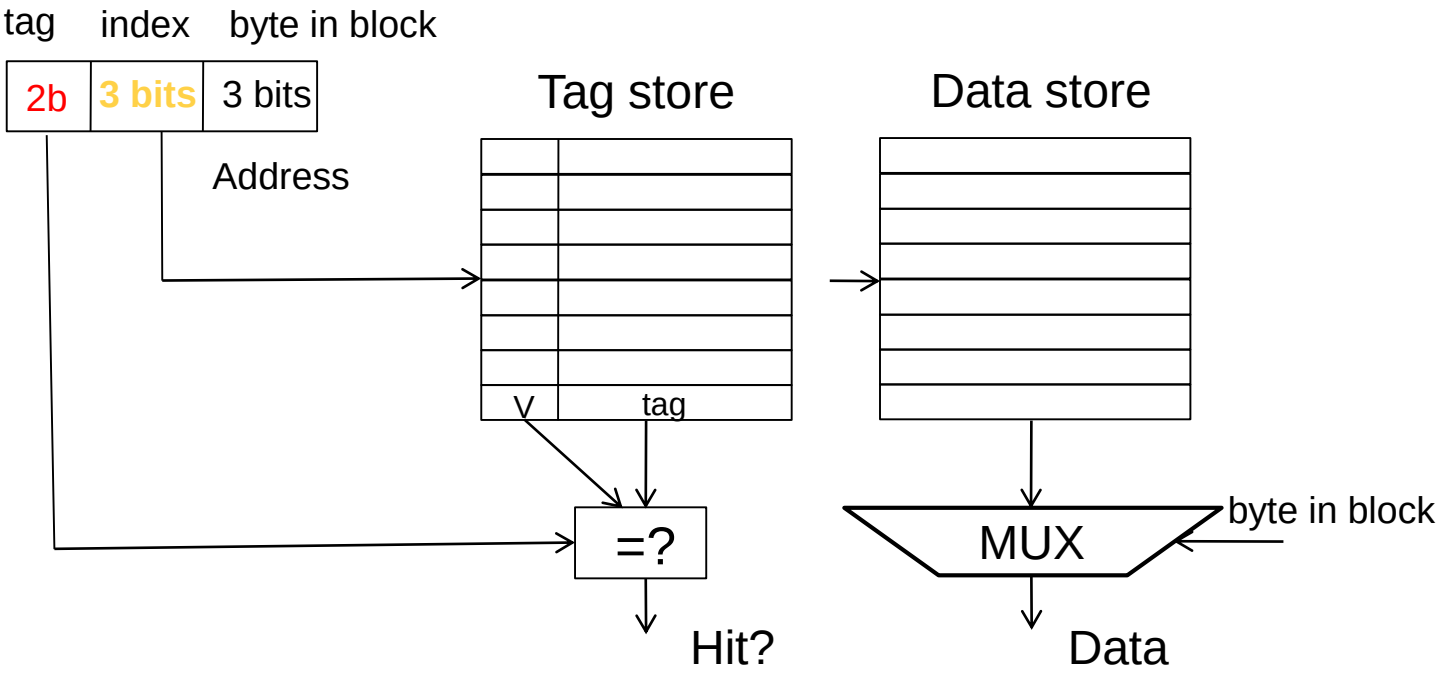
- If a block is in the cache (cache hit), the stored tag should be valid and match the tag of the block

Direct-Mapped Cache: Placement and Access



Main memory

- 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

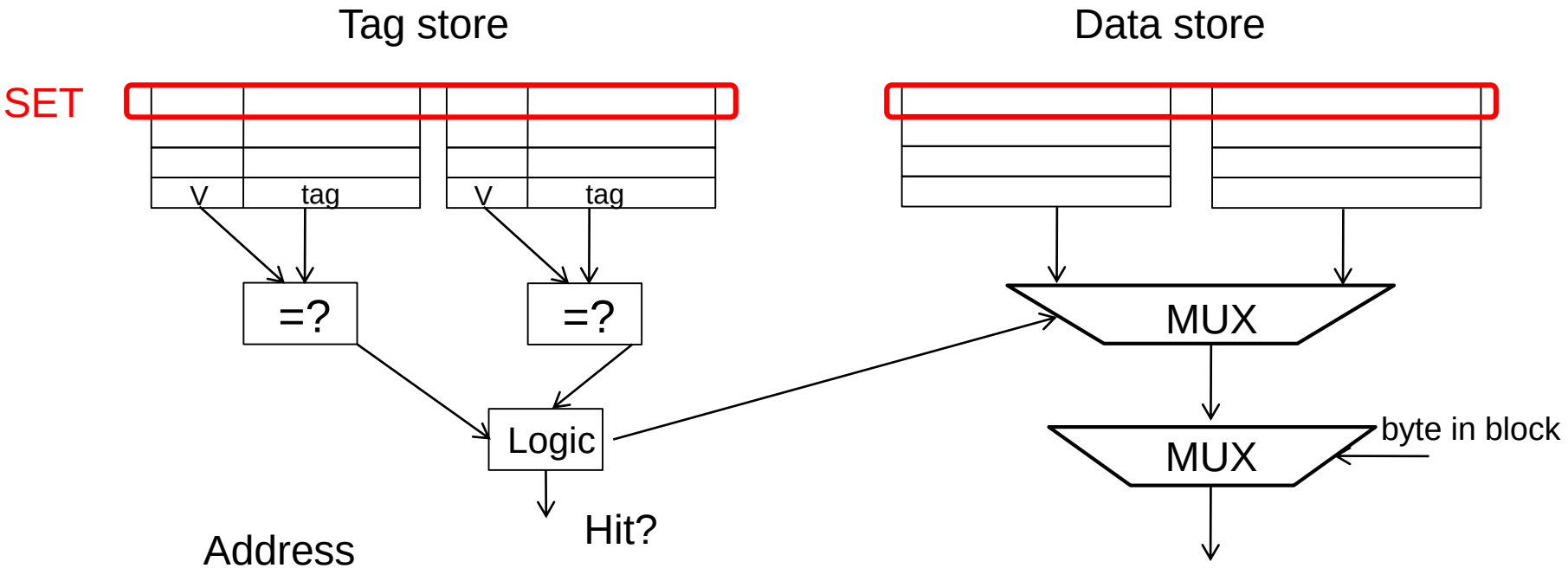
Direct-Mapped Caches

- Direct-mapped cache: Two blocks in memory that map to the same index in the cache cannot be present in the cache at the same time
  - One index → one entry

- 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, … → conflict in the cache index
  - All accesses are conflict misses

Set Associativity

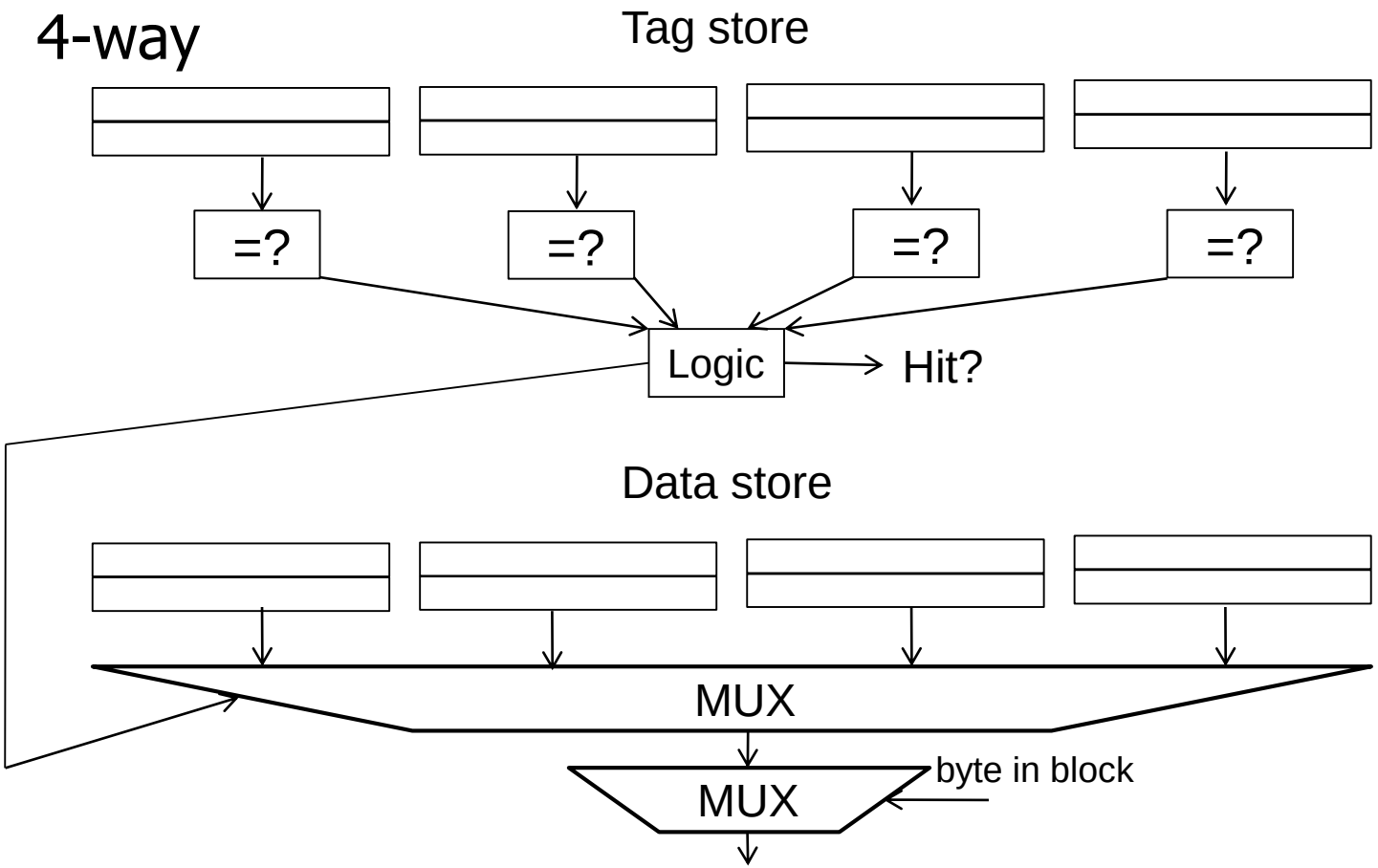
- Addresses 0 and 8 always conflict in direct mapped cache
- Instead of having one column of 8, have 2 columns of 4 blocks



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

Higher Associativity

- 4-way

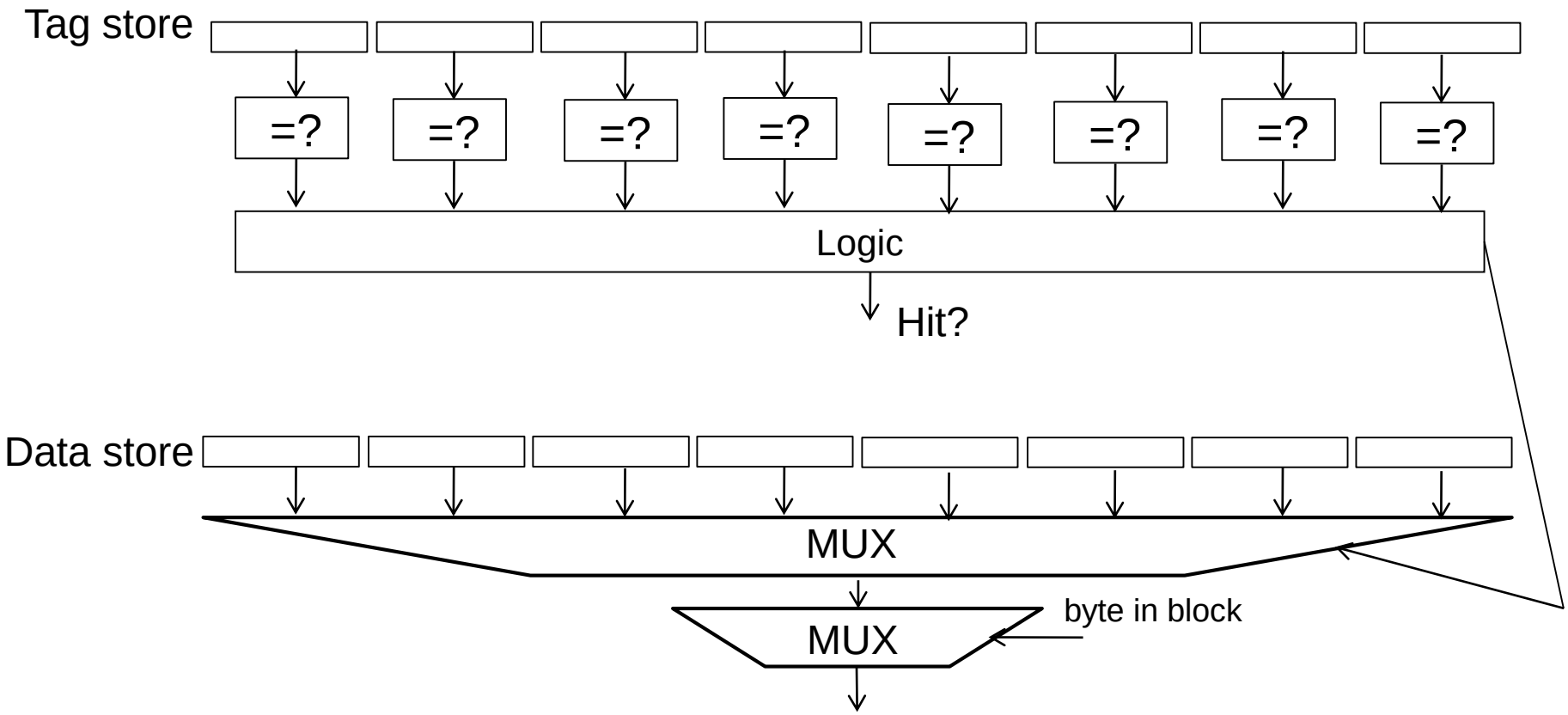


+ Likelihood of conflict misses even lower

-- 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 index (or set)?

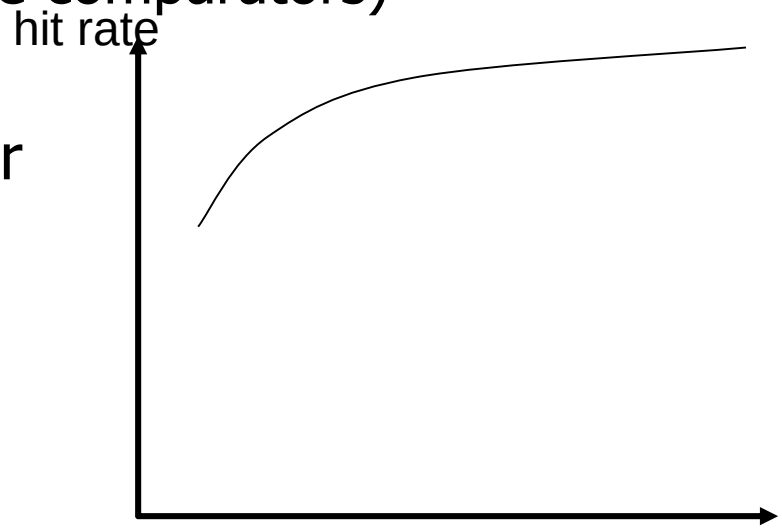
- Higher associativity

  ++ 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/adjust block priorities?
- There are three key decisions in a set:
  - 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

Eviction/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
    - Least recently used (how to implement?)
    - Not most recently used
    - 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:
  - What do you need to implement LRU perfectly?

- Question: 4-way set associative cache:
  - What do you need 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)

- Examples:
  - Not MRU (not most recently used)
  - Hierarchical LRU: divide the N-way set into M "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

Cache Replacement Policy: 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.

Reading

- Key observation: Some misses more costly than others as their latency is exposed as stall time. Reducing miss rate is not always good for performance. Cache replacement should take into account MLP of misses.

- Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu, and Yale N. Patt, **"A Case for MLP-Aware Cache Replacement"**
  *Proceedings of the 33rd International Symposium on Computer Architecture (**ISCA**)*, pages 167-177, Boston, MA, June 2006. Slides (ppt)

A Case for MLP-Aware Cache Replacement

Moinuddin K. Qureshi Daniel N. Lynch Onur Mutlu Yale N. Patt
Department of Electrical and Computer Engineering
The University of Texas at Austin
{moin, lynch, onur, patt}@hps.utexas.edu

Aside: 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?
  - Required speed of access to cache vs. physical memory
  - Number of blocks in a cache vs. physical memory
  - "Tolerable" amount of time to find a replacement candidate (disk versus memory access latency)
  - 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 combine 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 close-to-processor caches' tag stores for presence

  -- More bandwidth intensive; no combining of writes

Computer Architecture

Lecture 3b: Memory Hierarchy and Caches

Prof. Onur Mutlu
ETH Zürich

Fall 2018

26 September 2018

We did not cover the following slides in lecture. These are for your preparation for the next lecture.

Handling Writes (II)

- What if the processor writes to an entire block over a small amount of time?
- Is there any need to bring the block into the cache from memory in the first place?
- Ditto for a *portion* of the block, i.e., subblock
  - E.g., 4 bytes out of 64 bytes

Sectored Caches

- Idea: Divide a block into subblocks (or sectors)
  - Have separate valid and dirty bits for each sector
  - When is this useful? (Think writes…)

++ No need to transfer the entire cache block into the cache
(A write simply validates and updates a subblock)

++ More freedom in transferring subblocks into the cache (a cache block does not need to be in the cache fully)
(How many subblocks do you transfer on a read?)

-- More complex design

-- May not exploit spatial locality fully when used 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 (for Your Review)

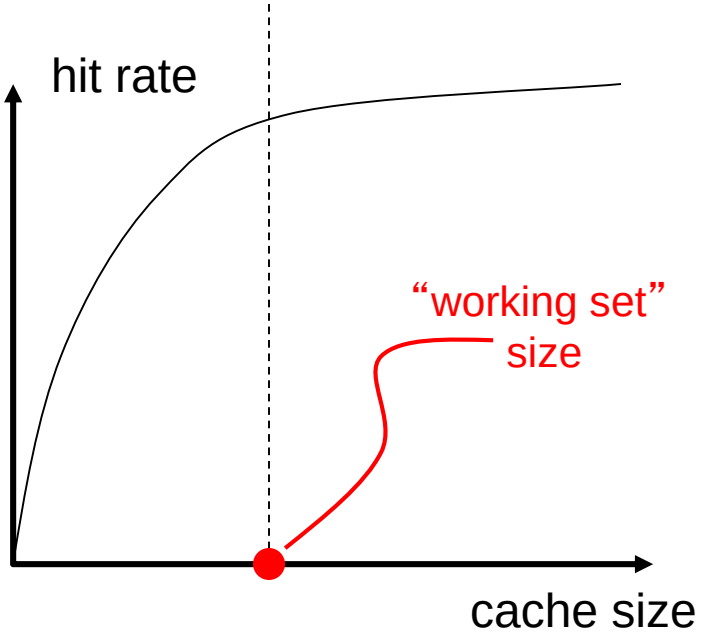
See Lecture 25a (More Caches) from Digital Circuits Spring 2018
http://www.youtube.com/watch?v=kMUZKjaPNWo

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

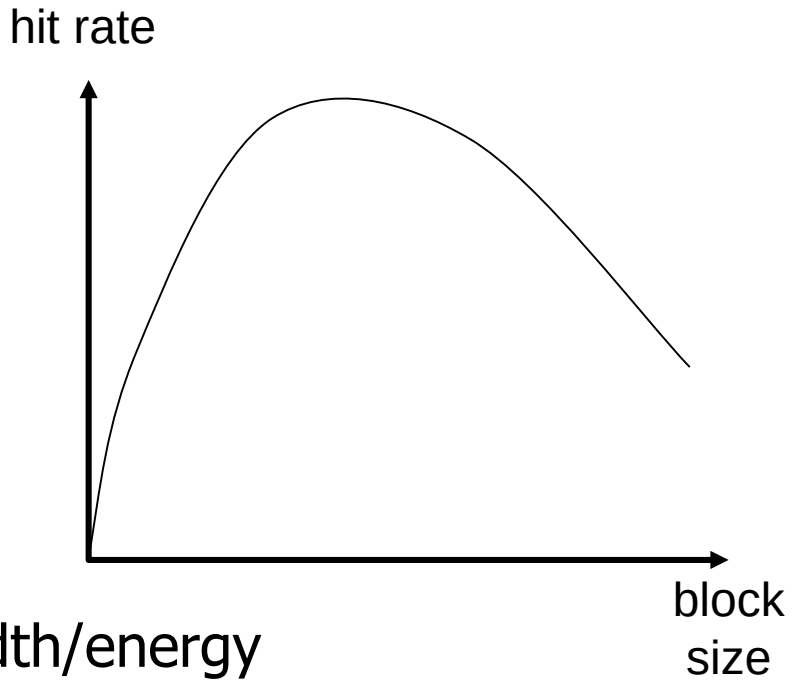


Block Size

- Block size is the data that is associated with an address tag
  - not necessarily the unit of transfer between hierarchies
    - Sub-blocking: A block divided into multiple pieces (each w/ V/D bits)

- Too small blocks
  - don't exploit spatial locality well
  - have larger tag overhead

- Too large blocks
  - too few total # of blocks → less temporal locality exploitation
  - waste of cache space and bandwidth/energy if spatial locality is not high



Large Blocks: Critical-Word and Subblocking

- 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

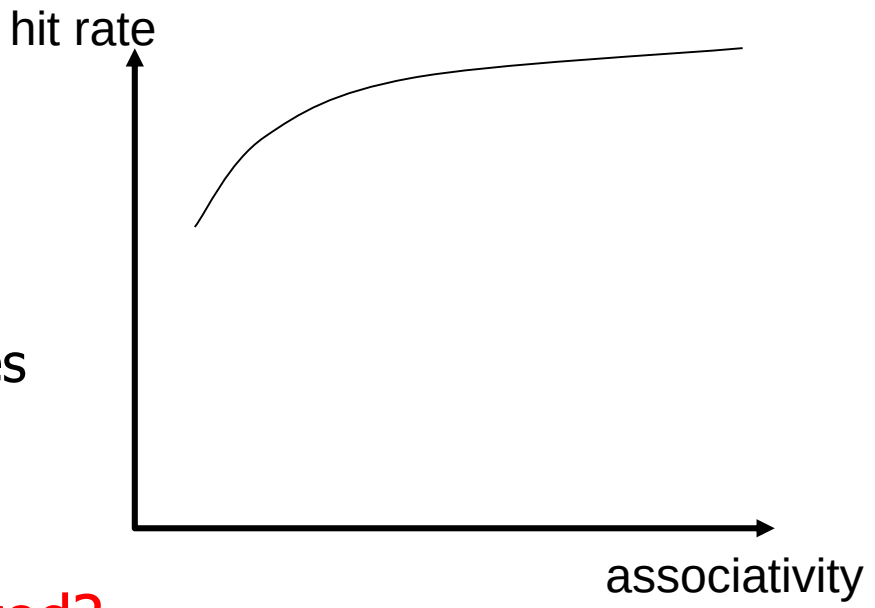
- Large cache blocks can waste bus bandwidth
  - divide a block into subblocks
  - associate separate valid and dirty bits for each subblock
  - Recall: When is this useful?

Associativity

- How many blocks can be present in the same index (i.e., set)?

- Larger associativity
  - lower miss rate (reduced conflicts)
  - higher hit latency and area cost (plus diminishing returns)

- Smaller associativity
  - lower cost
  - lower hit latency
    - Especially important for L1 caches



- Is power of 2 associativity required?

End of Cache Performance (for Your Review)

See Lecture 25a (More Caches) from Digital Circuits Spring 2018
http://www.youtube.com/watch?v=kMUZKjaPNWo

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: Anticipate which blocks will be needed soon
- Conflict
  - More associativity
  - Other ways to get more associativity without making the cache associative
    - Victim cache
    - Better, randomized indexing
    - Software hints?
- Capacity
  - Utilize cache space better: keep blocks that will be referenced
  - Software management: divide working set and computation such that each "computation phase" fits in cache

How to Improve Cache Performance

- Three fundamental goals

- Reducing miss rate
  - Caveat: reducing miss rate can reduce performance if more costly-to-refetch blocks are evicted

- Reducing miss latency or miss cost

- Reducing hit latency or hit cost

- The above three **together** affect performance

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
- Conflict
  - More associativity
  - Other ways to get more associativity without making the cache associative
    - Victim cache
    - Better, randomized indexing
    - 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

How to Improve Cache Performance

- Three fundamental goals
- Reducing miss rate
  - Caveat: reducing miss rate can reduce performance if more costly-to-refetch blocks are evicted
- Reducing miss latency or miss cost
- Reducing hit latency or hit cost
- The above three **together** affect performance

Improving Basic Cache Performance

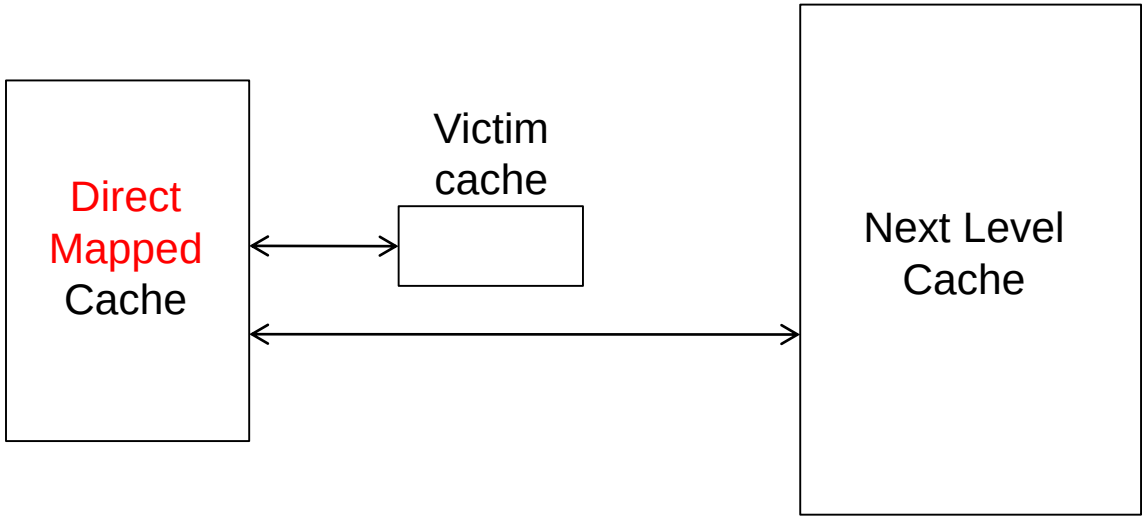
- Reducing miss rate
  - More associativity
  - Alternatives/enhancements to associativity
    - Victim caches, hashing, pseudo-associativity, skewed associativity
  - Better replacement/insertion policies
  - Software approaches
- Reducing miss latency/cost
  - Multi-level caches
  - Critical word first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

Cheap Ways of Reducing Conflict Misses

- Instead of building highly-associative caches:

- Victim Caches
- Hashed/randomized Index Functions
- Pseudo Associativity
- Skewed Associative Caches
- …

Victim Cache: Reducing Conflict Misses



- Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," ISCA 1990.
- Idea: Use a small fully-associative buffer (victim cache) to store recently evicted blocks

  + Can avoid ping ponging of cache blocks mapped to the same set (if two cache blocks continuously accessed in nearby time conflict with each other)

  -- Increases miss latency if accessed serially with L2; adds complexity

Hashing and Pseudo-Associativity

- Hashing: Use better "randomizing" index functions

  + can reduce conflict misses
    - by distributing the accessed memory blocks more evenly to sets
    - Example of conflicting accesses: strided access pattern where stride value equals number of sets in cache

  -- More complex to implement: can lengthen critical path

- Pseudo-associativity (Poor Man's associative cache)
  - Serial lookup: On a miss, use a different index function and access cache again
  - Given a direct-mapped array with K cache blocks
    - Implement K/N sets
    - Given address Addr, sequentially look up: {0,Addr[lg(K/N)-1: 0]}, {1,Addr[lg(K/N)-1: 0]}, … , {N-1,Addr[lg(K/N)-1: 0]}

  + Less complex than N-way; -- Longer cache hit/miss latency

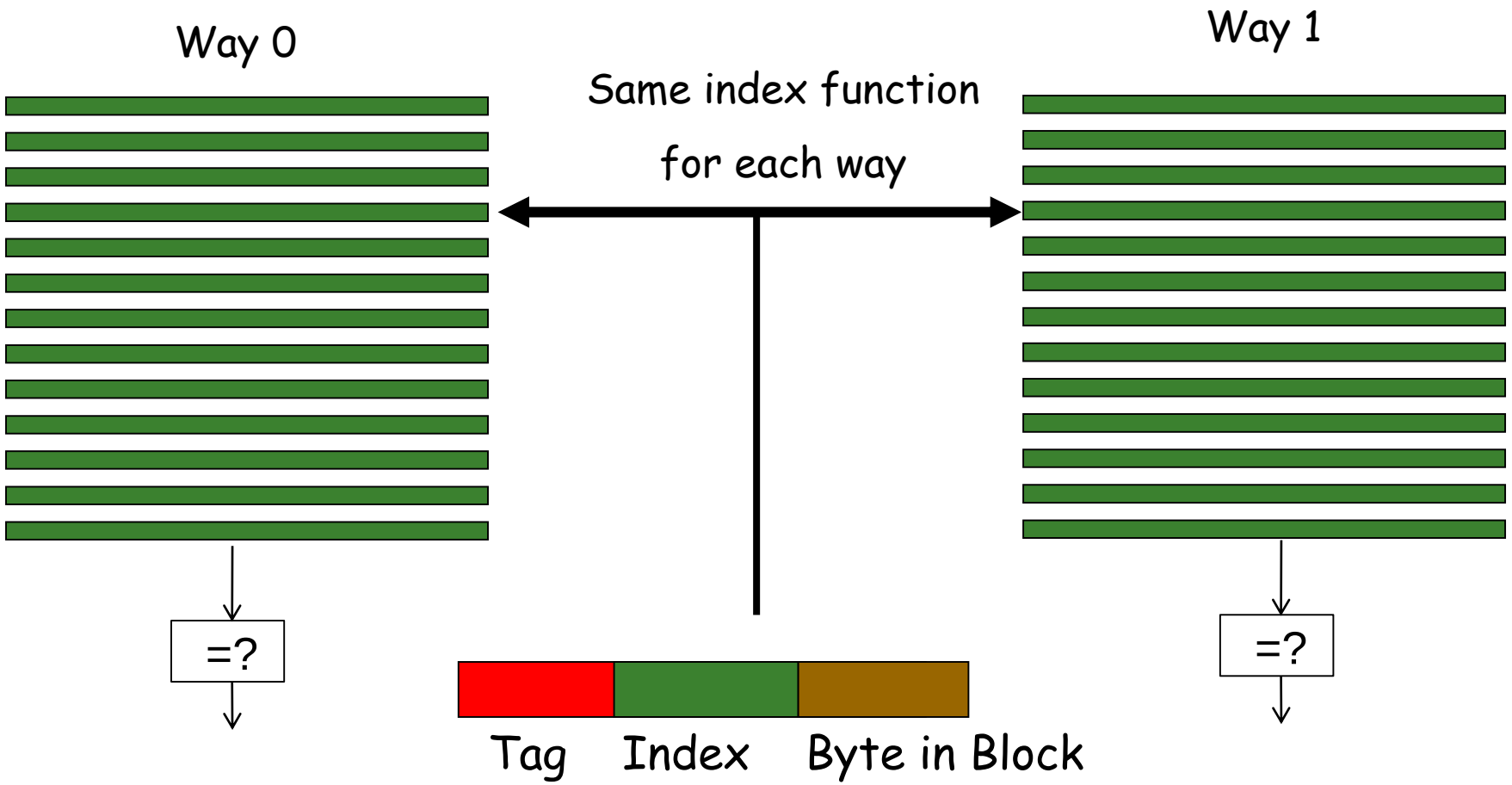
Skewed Associative Caches

- Idea: Reduce conflict misses by using different index functions for each cache way

- Seznec, "A Case for Two-Way Skewed-Associative Caches," ISCA 1993.

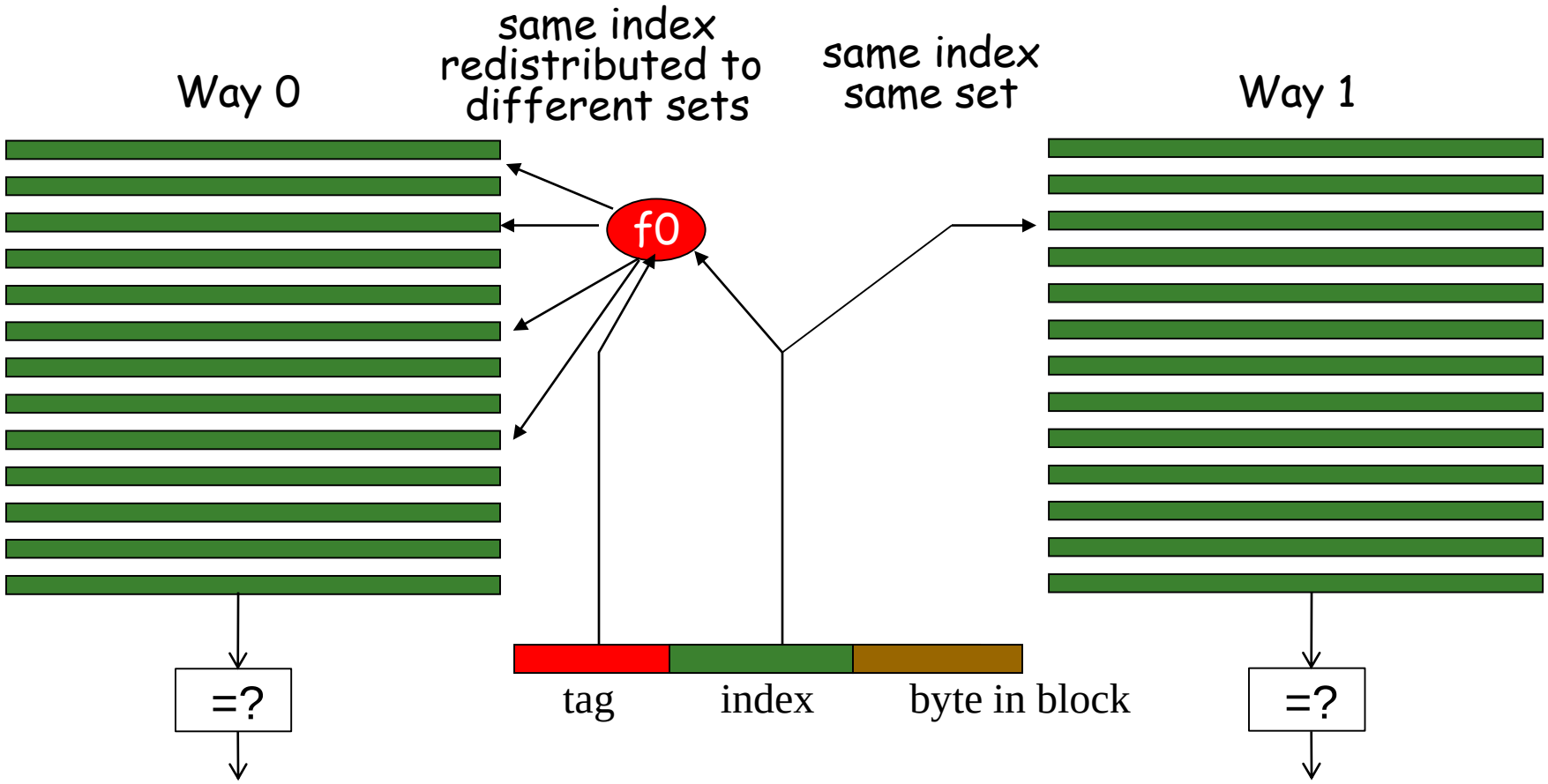
Skewed Associative Caches (I)

- Basic 2-way associative cache structure



Skewed Associative Caches (II)

- Skewed associative caches
  - Each bank has a different index function



Skewed Associative Caches (III)

- Idea: Reduce conflict misses by using different index functions for each cache way

- Benefit: indices are more randomized (memory blocks are better distributed across sets)
  - Less likely two blocks have same index (esp. with strided access)
    - Reduced conflict misses

- Cost: additional latency of hash function

- Seznec, "A Case for Two-Way Skewed-Associative Caches," ISCA 1993.

Software Approaches for Higher Hit Rate

- Restructuring data access patterns
- Restructuring data layout

- Loop interchange
- Data structure separation/merging
- Blocking
- …

Restructuring Data Access Patterns (I)

- Idea: Restructure data layout or data access patterns
- Example: If column-major
  - x[i+1,j] follows x[i,j] in memory
  - x[i,j+1] is far away from x[i,j]

Poor code

```
for i = 1, rows
    for j = 1, columns
        sum = sum + x[i,j]
```

Better code

```
for j = 1, columns
    for i = 1, rows
        sum = sum + x[i,j]
```

- This is called loop interchange
- Other optimizations can also increase hit rate
  - Loop fusion, array merging, …
- What if multiple arrays? Unknown array size at compile time?

Restructuring Data Access Patterns (II)

- Blocking
  - Divide loops operating on arrays into computation chunks so that each chunk can hold its data in the cache
  - Avoids cache conflicts between different chunks of computation
  - Essentially: Divide the working set so that each piece fits in the cache

- But, there are still self-conflicts in a block
  1. there can be conflicts among different arrays
  2. array sizes may be unknown at compile/programming time

Restructuring Data Layout (I)

```
struct Node {
    struct Node* next;
    int key;
    char [256] name;
    char [256] school;
}

while (node) {
    if (node→key == input-key) {
        // access other fields of node
    }
    node = node→next;
}
```

- Pointer based traversal (e.g., of a linked list)
- Assume a huge linked list (1B nodes) and unique keys
- Why does the code on the left have poor cache hit rate?
  - "Other fields" occupy most of the cache line even though rarely accessed!

Restructuring Data Layout (II)

```
struct Node {
    struct Node* next;
    int key;
    struct Node-data* node-data;
}

struct Node-data {
    char [256] name;
    char [256] school;
}

while (node) {
    if (node→key == input-key) {
        // access node→node-data
    }
    node = node→next;
}
```

- Idea: separate frequently-used fields of a data structure and pack them into a separate data structure
- Who should do this?
  - Programmer
  - Compiler
    - Profiling vs. dynamic
  - Hardware?
  - Who can determine what is frequently used?

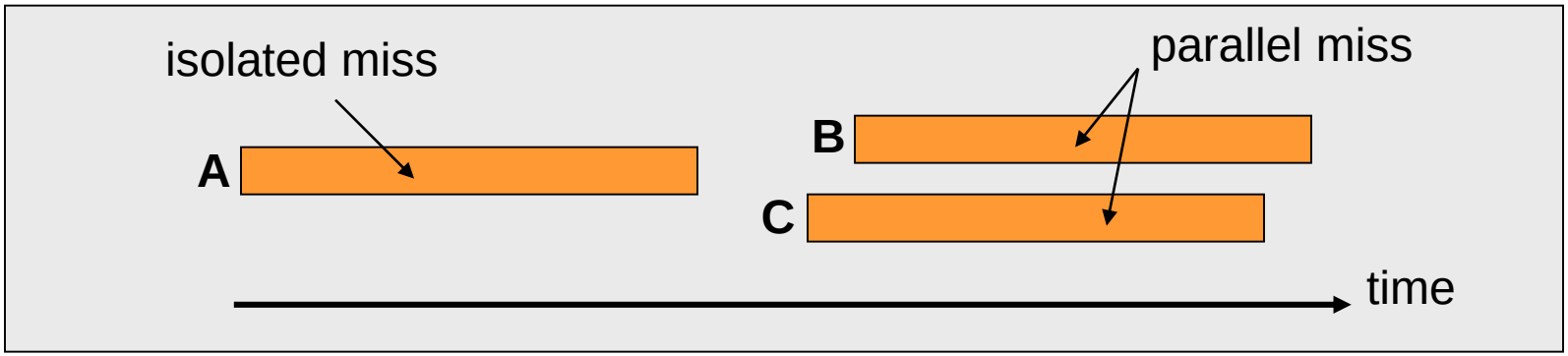
Improving Basic Cache Performance

- 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)
  - Multiple accesses per cycle
  - Software approaches

Miss Latency/Cost

- What is miss latency or miss cost affected by?
  - Where does the miss get serviced from?
    - Local vs. remote memory
    - What level of cache in the hierarchy?
    - Row hit versus row miss in DRAM
    - Queueing delays in the memory controller and the interconnect
    - …
  - How much does the miss stall the processor?
    - Is it overlapped with other latencies?
    - Is the data immediately needed?
    - …

Memory Level Parallelism (MLP)



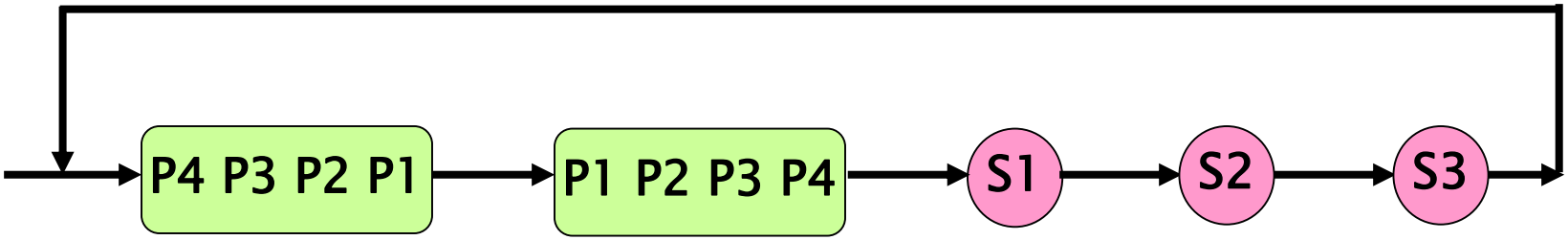
- Memory Level Parallelism (MLP) means generating and servicing multiple memory accesses in parallel [Glew' 98]
- Several techniques to improve MLP (e.g., out-of-order execution)
- MLP varies. Some misses are isolated and some parallel

How does this affect cache replacement?

Traditional Cache Replacement Policies

- Traditional cache replacement policies try to reduce miss count
- Implicit assumption: Reducing miss count reduces memory-related stall time
- Misses with varying cost/MLP breaks this assumption!
- Eliminating an isolated miss helps performance more than eliminating a parallel miss
- Eliminating a higher-latency miss could help performance more than eliminating a lower-latency miss

An Example



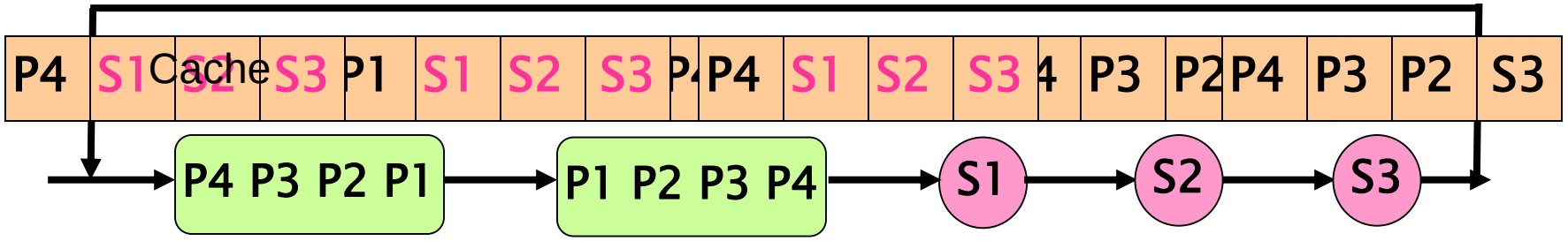
Misses to blocks P1, P2, P3, P4 can be parallel
Misses to blocks S1, S2, and S3 are isolated

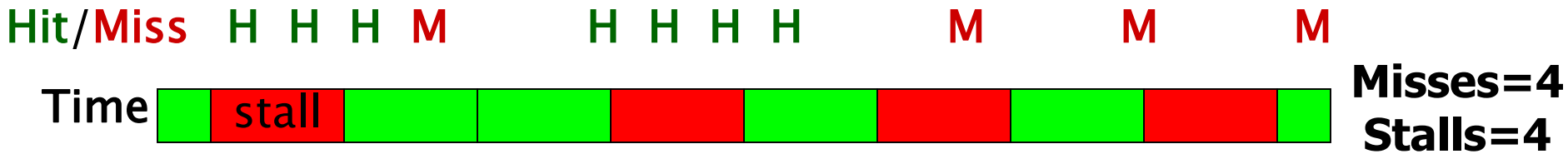
Two replacement algorithms:

1. Minimizes miss count (Belady's OPT)
2. Reduces isolated miss (MLP-Aware)

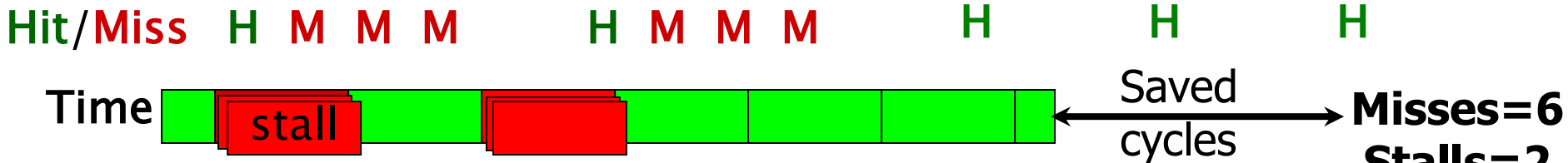
For a fully associative cache containing 4 blocks

Fewest Misses ≠ Best Performance





Belady's OPT replacement



MLP-Aware replacement

MLP-Aware Cache Replacement

- How do we incorporate MLP into replacement decisions?

- Qureshi et al., "A Case for MLP-Aware Cache Replacement," ISCA 2006.
  - Reading for review

A Case for MLP-Aware Cache Replacement

Moinuddin K. Qureshi Daniel N. Lynch Onur Mutlu Yale N. Patt
Department of Electrical and Computer Engineering
The University of Texas at Austin
{*moin, lynch, onur, patt*}*@hps.utexas.edu*

Other Recommended Cache Papers (I)

- Qureshi et al., "Adaptive Insertion Policies for High Performance Caching," ISCA 2007.

Adaptive Insertion Policies for High Performance Caching

Moinuddin K. Qureshi† Aamer Jaleel§ Yale N. Patt† Simon C. Steely Jr.§ Joel Emer§

†ECE Department
The University of Texas at Austin
{moin, patt}@hps.utexas.edu

§Intel Corporation, VSSAD
Hudson, MA
{aamer.jaleel, simon.c.steely.jr, joel.emer}@intel.com

Other Recommended Cache Papers (II)

- Seshadri et al., "The Evicted-Address Filter: A Unified Mechanism to Address Both Cache Pollution and Thrashing," PACT 2012.

The Evicted-Address Filter: A Unified Mechanism to Address Both Cache Pollution and Thrashing

Vivek Seshadri† vseshadr@cs.cmu.edu
Onur Mutlu† onur@cmu.edu
Michael A Kozuch⋆ michael.a.kozuch@intel.com
Todd C Mowry† tcm@cs.cmu.edu

†Carnegie Mellon University ⋆Intel Labs Pittsburgh

Other Recommended Cache Papers (III)

- Pekhimenko et al., "Base-Delta-Immediate Compression: Practical Data Compression for On-Chip Caches," PACT 2012.

Base-Delta-Immediate Compression: Practical Data Compression for On-Chip Caches

Gennady Pekhimenko† gpekhime@cs.cmu.edu

Vivek Seshadri† vseshadr@cs.cmu.edu

Onur Mutlu† onur@cmu.edu

Michael A. Kozuch⋆ michael.a.kozuch@intel.com

Phillip B. Gibbons⋆ phillip.b.gibbons@intel.com

Todd C. Mowry† tcm@cs.cmu.edu

†Carnegie Mellon University ⋆Intel Labs Pittsburgh

Hybrid Cache Replacement (Selecting Between Multiple Replacement Policies)

Hybrid Cache Replacement

- Problem: Not a single policy provides the highest performance
  - For any given set
  - For the entire cache overall

- Idea: Implement both policies and pick the one that is expected to perform best at runtime
  - On a per-set basis or for the entire cache

  + Higher performance

  -- Higher cost, complexity; Need selection mechanism

- How do you determine the best policy?
  - Implement multiple tag stores, each following a particular policy
  - Find the best and have the main tag store follow the best policy

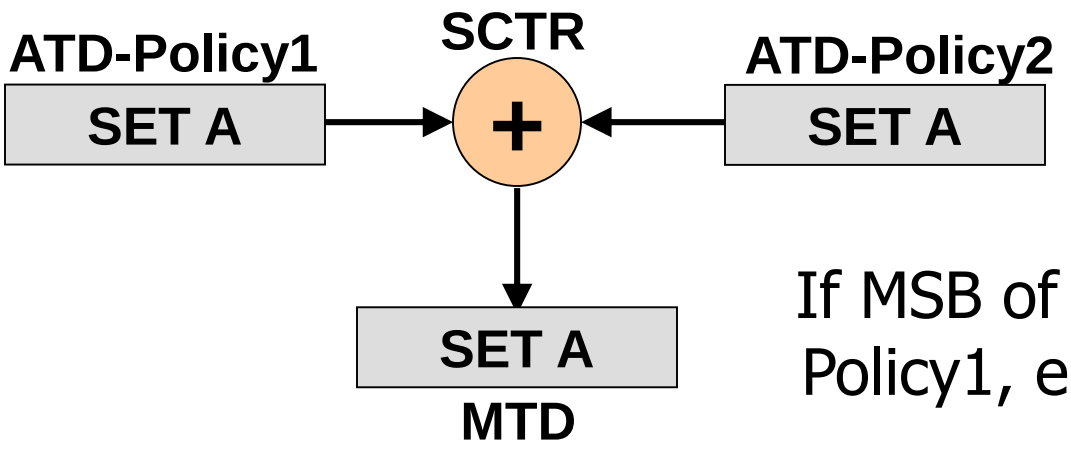
Terminology

- Tag Store is also called Tag Directory

- Main Tag Store/Directory (MTD)
  - Tag Store that is actually used to keep track of the block addresses present in the cache

- Auxiliary Tag Store/Directory (ATD-PolicyX)
  - Tag Store that is used to emulate a policy X
  - **Not** used for tracking the block addresses present in the cache
  - Used for tracking what the block addresses in the cache would have been if the cache were following Policy X

Tournament Selection (TSEL) of Replacement Policies for a Single Set



If MSB of SCTR is 1, MTD uses Policy1, else MTD uses Policy2

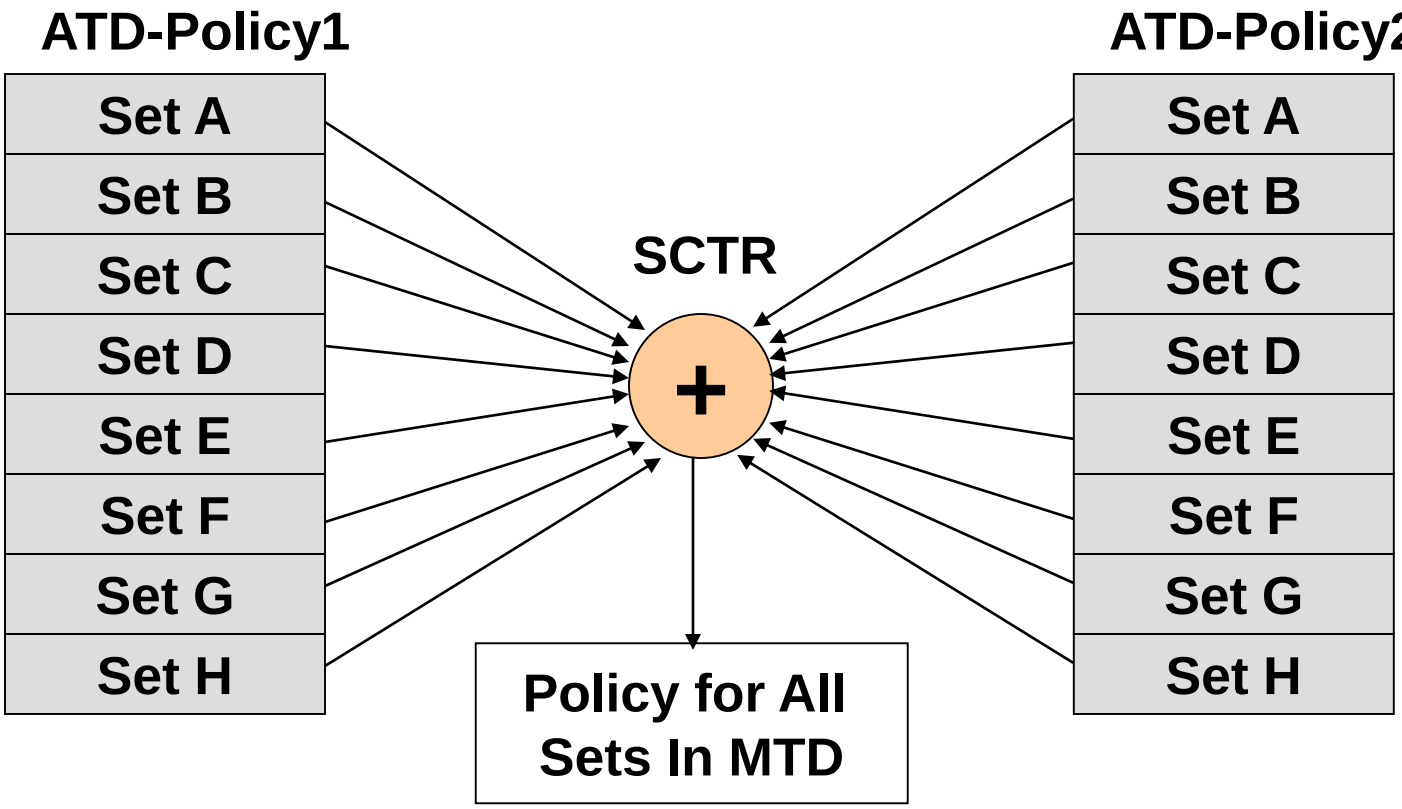
ATD-Policy1	ATD-Policy2	Saturating Counter (SCTR)
HIT	HIT	Unchanged
MISS	MISS	Unchanged
HIT	MISS	+= Cost of Miss in ATD-Policy2
MISS	HIT	-= Cost of Miss in ATD-Policy1

Qureshi et al., "A Case for MLP-Aware Cache Replacement," ISCA 2006.

Extending TSEL to All Sets

Implementing TSEL on a per-set basis is expensive

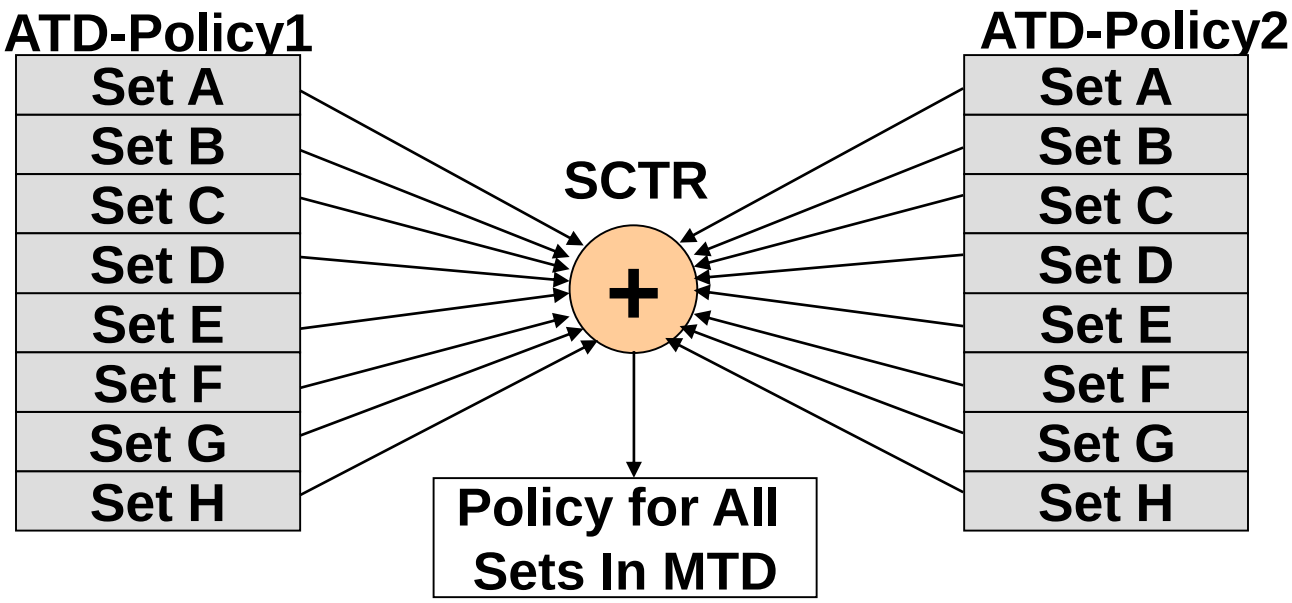
Counter overhead can be reduced by using a global counter



Qureshi et al., "A Case for MLP-Aware Cache Replacement," ISCA 2006.

Dynamic Set Sampling (DSS)

Not all sets are required to decide the best policy
Have the ATD entries only for few sets.



Sets that have ATD entries (B, E, G) are called leader sets

Qureshi et al., "A Case for MLP-Aware Cache Replacement," ISCA 2006.

Dynamic Set Sampling (DSS)

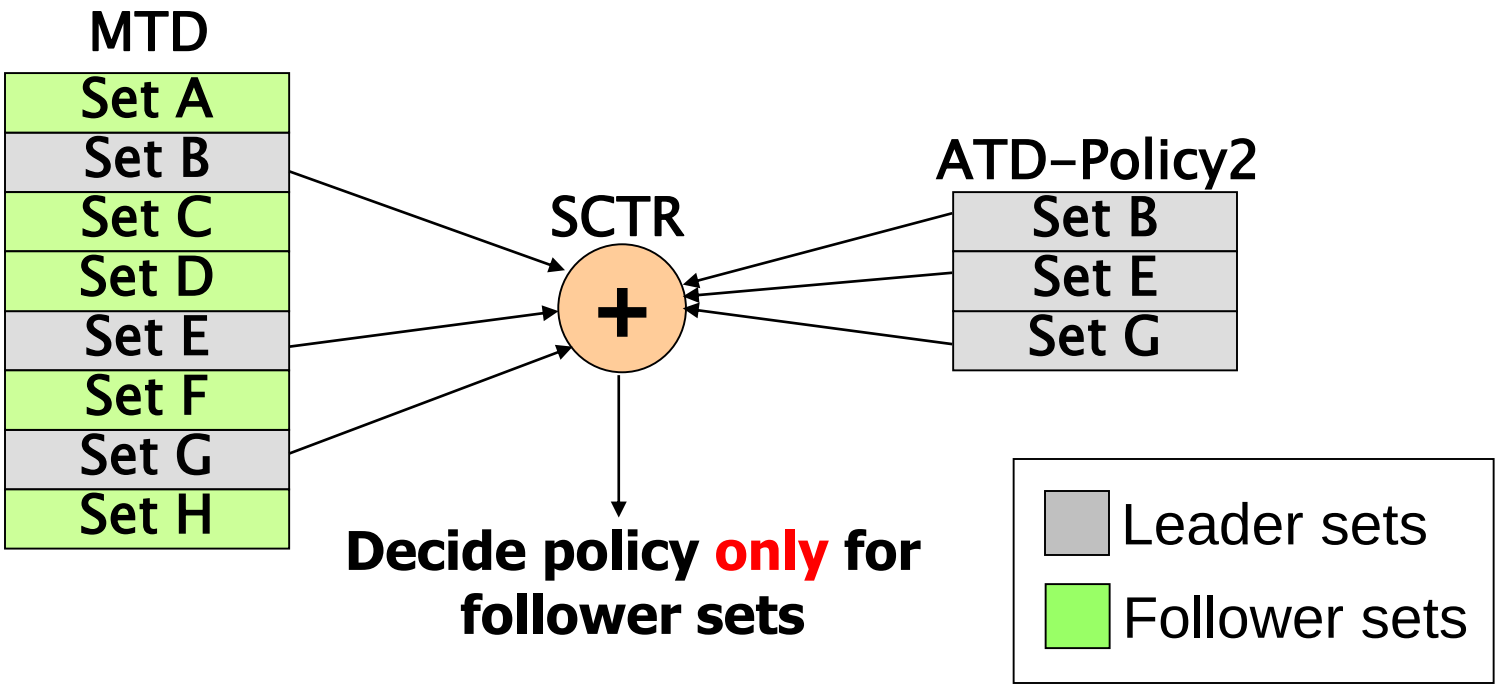
How many sets are required to choose best performing policy?

- Bounds using analytical model and simulation (in paper)
- DSS with 32 leader sets performs similar to having all sets
- Last-level cache typically contains 1000s of sets, thus ATD entries are required for only 2%-3% of the sets

ATD overhead can further be reduced by using MTD to always simulate one of the policies (say Policy1)

Qureshi et al., "A Case for MLP-Aware Cache Replacement," ISCA 2006.

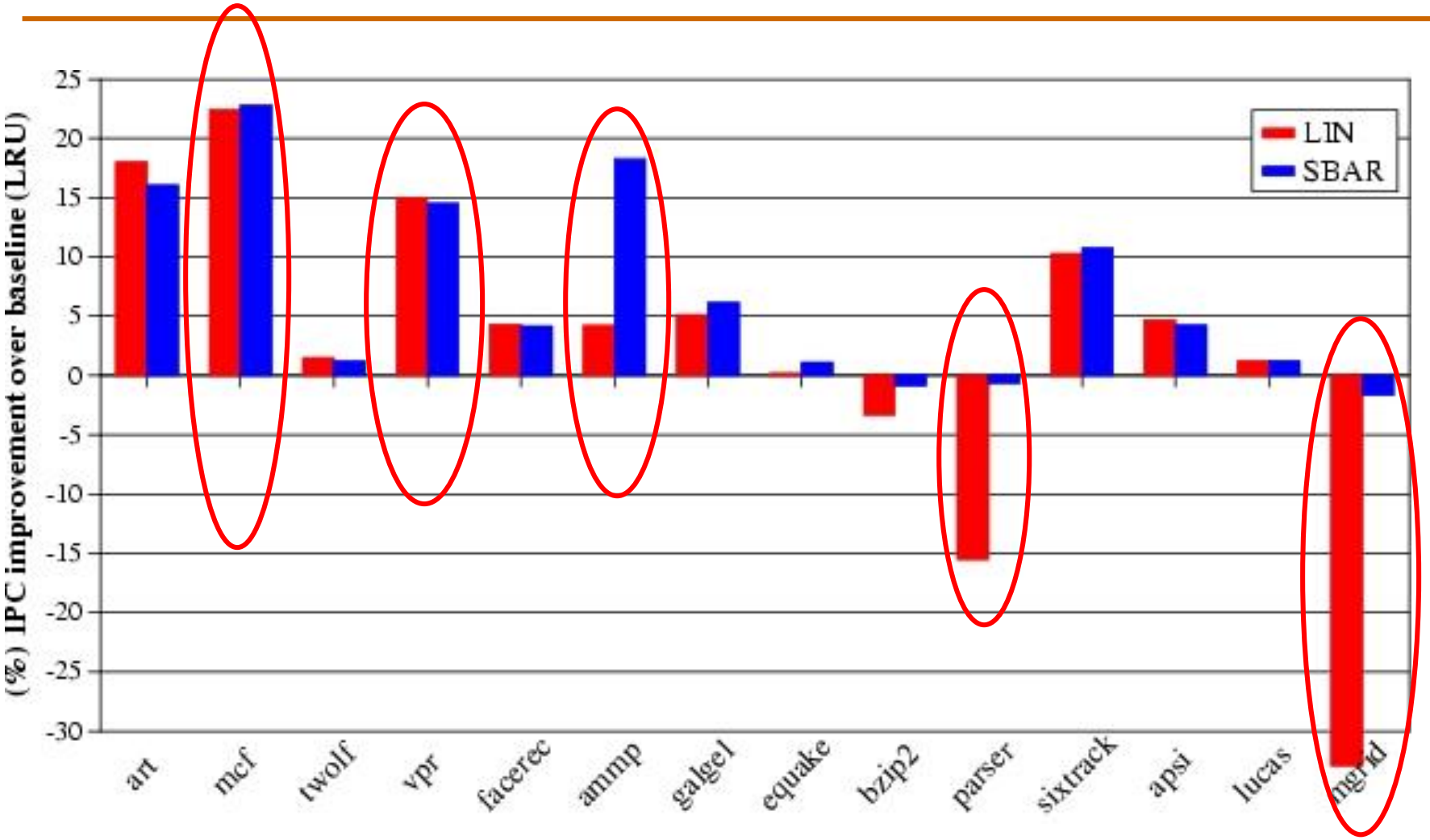
Sampling Based Adaptive Replacement (SBAR)



The storage overhead of SBAR is less than 2KB
(0.2% of the baseline 1MB cache)

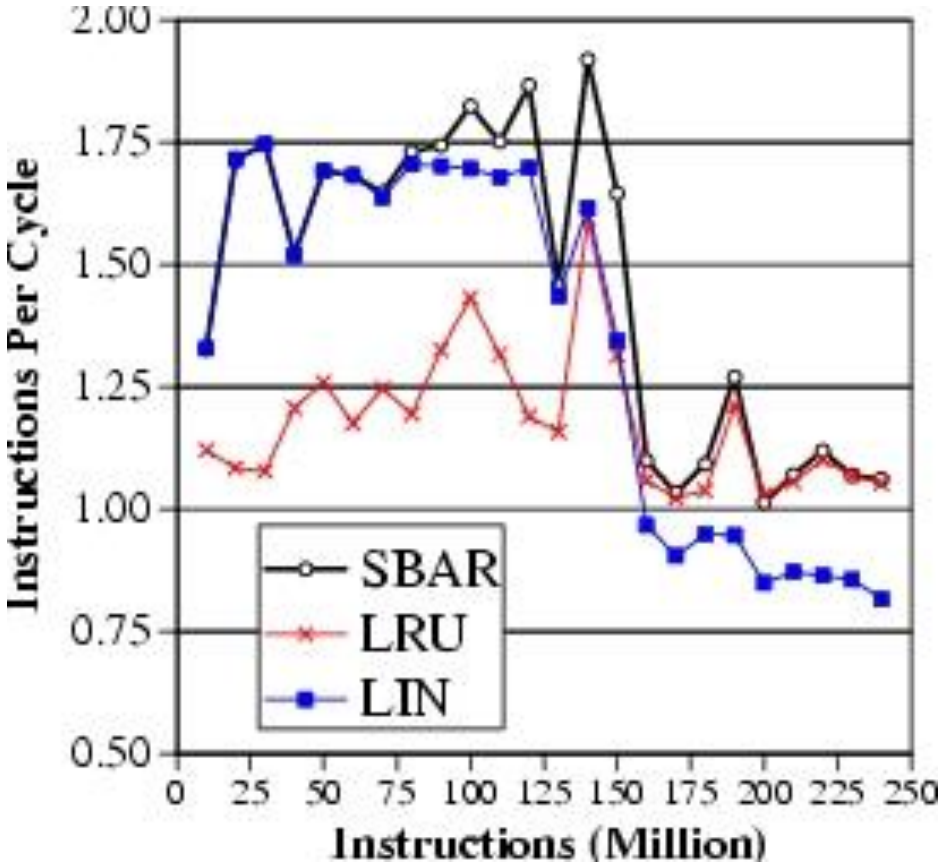
Qureshi et al., "A Case for MLP-Aware Cache Replacement," ISCA 2006.

Results for SBAR



Qureshi et al., "A Case for MLP-Aware Cache Replacement," ISCA 2006.

SBAR adaptation to phases



SBAR selects the best policy for each phase of this application

Enabling Multiple Outstanding Misses

Handling Multiple Outstanding Accesses

- Question: If the processor can generate multiple cache accesses, can the later accesses be handled while a previous miss is outstanding?
- Goal: Enable cache access when there is a pending miss
- Goal: Enable multiple misses in parallel
  - Memory-level parallelism (MLP)
- Solution: Non-blocking or lockup-free caches
  - Kroft, "Lockup-Free Instruction Fetch/Prefetch Cache Organization," ISCA 1981.

Handling Multiple Outstanding Accesses

- Idea: Keep track of the status/data of misses that are being handled in Miss Status Handling Registers (MSHRs)

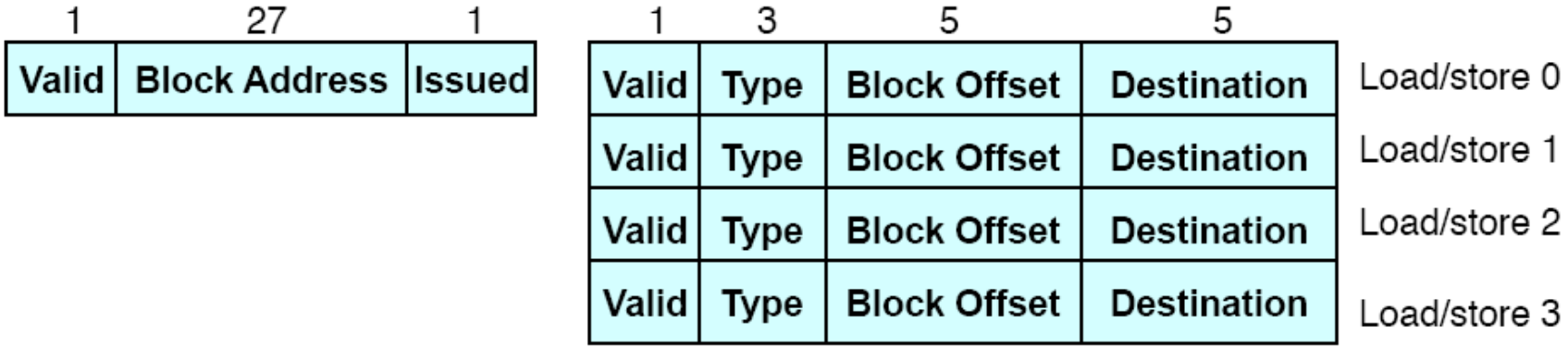
  - A cache access checks MSHRs to see if a miss to the same block is already *pending.*
    - If pending, a new request is not generated
    - If pending and the needed data available, data forwarded to later load

  - Requires buffering of outstanding miss requests

Miss Status Handling Register

- Also called “miss buffer”
- Keeps track of
  - Outstanding cache misses
  - Pending load/store accesses that refer to the missing cache block
- Fields of a single MSHR entry
  - Valid bit
  - Cache block address (to match incoming accesses)
  - Control/status bits (prefetch, issued to memory, which subblocks have arrived, etc)
  - Data for each subblock
  - For each pending load/store
    - Valid, type, data size, byte in block, destination register or store buffer entry address

Miss Status Handling Register Entry



MSHR Operation

- On a cache miss:
  - Search MSHRs for a pending access to the same block
    - Found: Allocate a load/store entry in the same MSHR entry
    - Not found: Allocate a new MSHR
    - No free entry: stall
- When a subblock returns from the next level in memory
  - Check which loads/stores waiting for it
    - Forward data to the load/store unit
    - Deallocate load/store entry in the MSHR entry
  - Write subblock in cache or MSHR
  - If last subblock, deallocate MSHR (after writing the block in cache)

Non-Blocking Cache Implementation

- When to access the MSHRs?
  - In parallel with the cache?
  - After cache access is complete?

- MSHRs need not be on the critical path of hit requests
  - Which one below is the common case?
    - Cache miss, MSHR hit
    - Cache hit

Enabling High Bandwidth Memories

Multiple Instructions per Cycle

- Processors can generate multiple cache/memory accesses per cycle
- How do we ensure the cache/memory can handle multiple accesses in the same clock cycle?

- Solutions:
  - true multi-porting
  - virtual multi-porting (time sharing a port)
  - multiple cache copies
  - banking (interleaving)

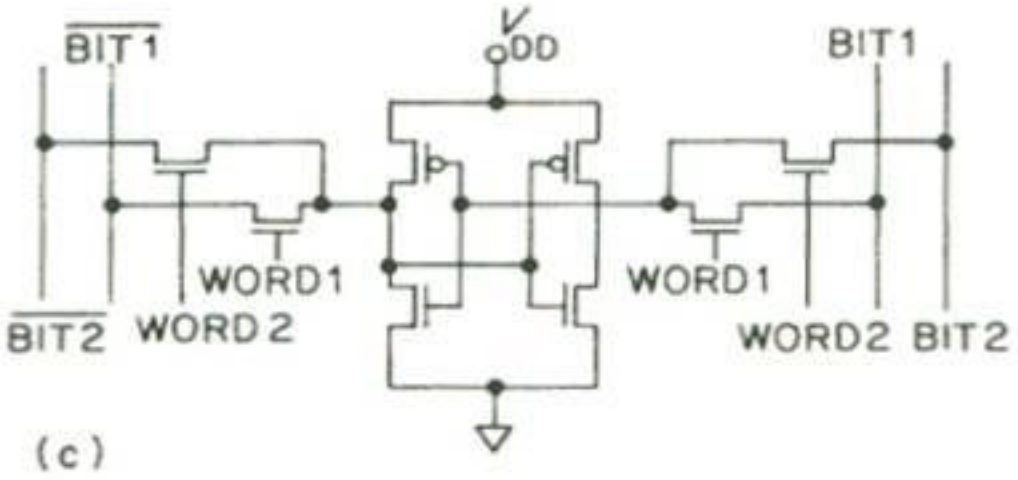
Handling Multiple Accesses per Cycle (I)

- True multiporting
  - Each memory cell has multiple read or write ports

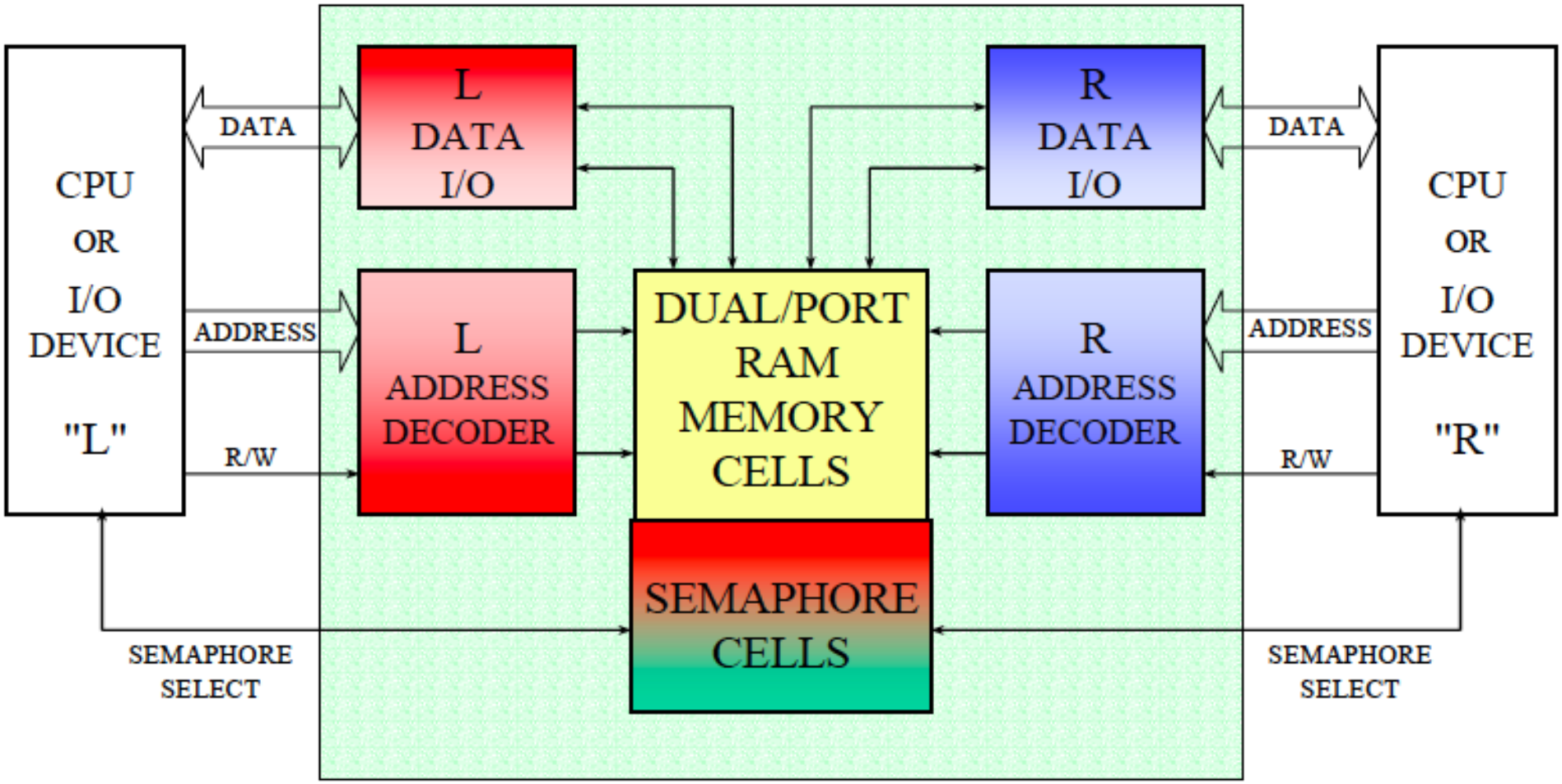
+ Truly concurrent accesses (no conflicts on read accesses)

-- Expensive in terms of latency, power, area

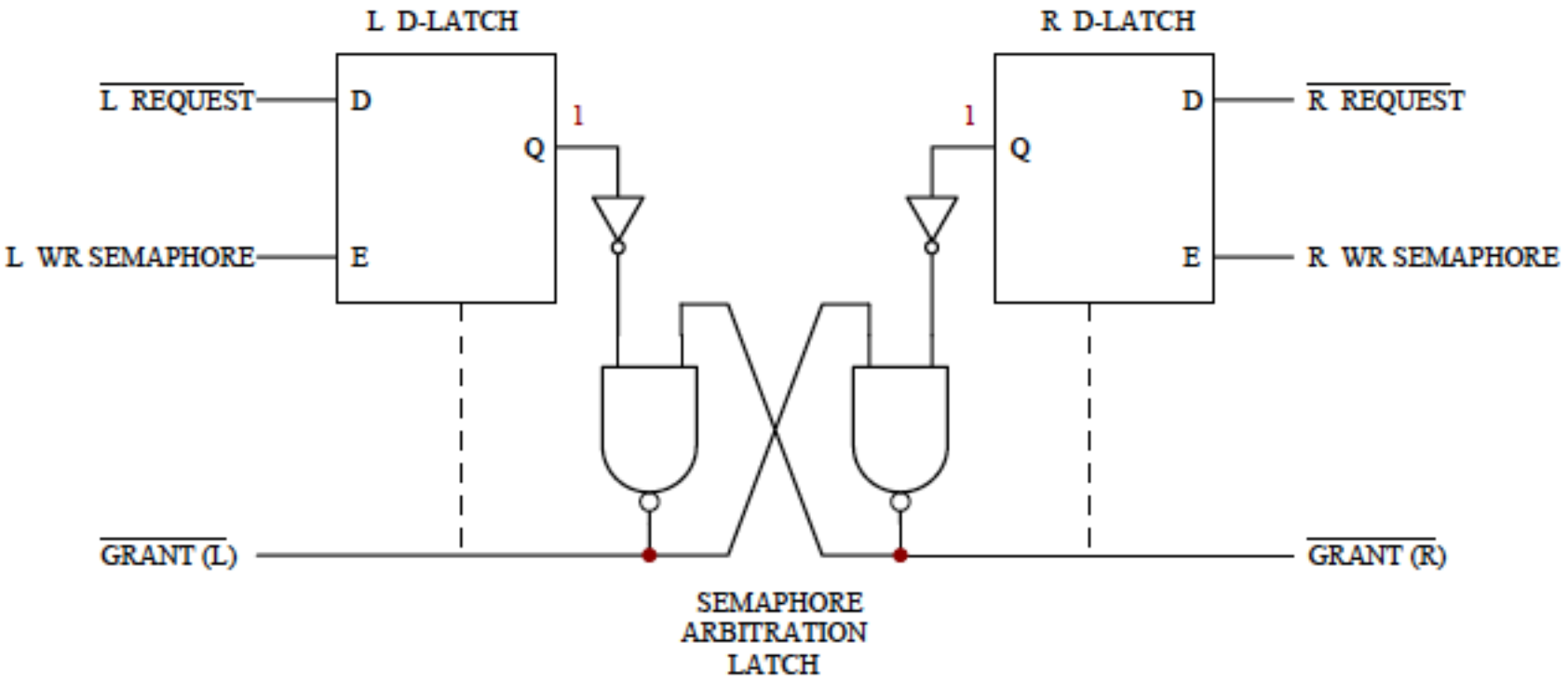
  - What about read and write to the same location at the same time?
    - Peripheral logic needs to handle this



Peripheral Logic for True Multiporting



Peripheral Logic for True Multiporting

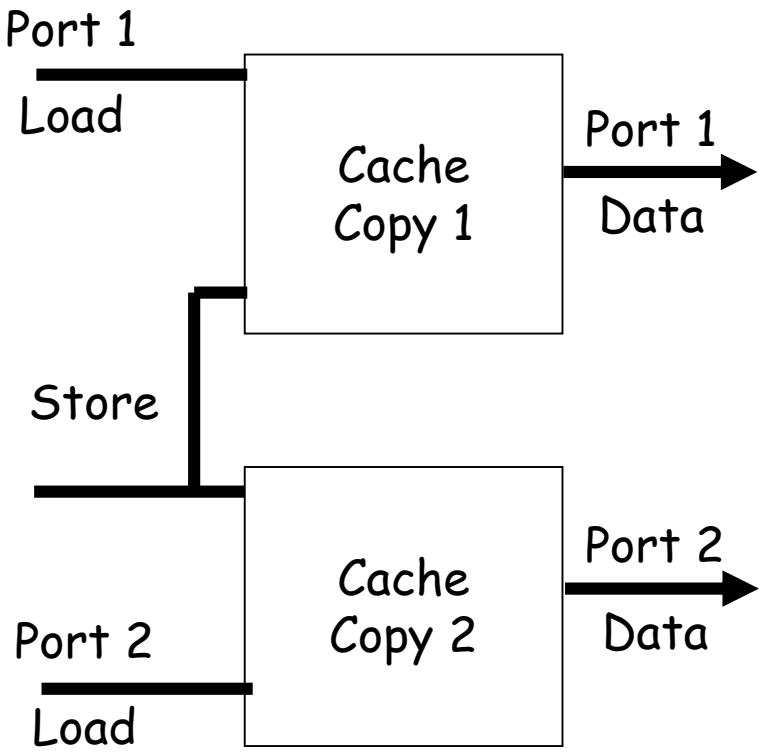


Handling Multiple Accesses per Cycle (II)

- Virtual multiporting
  - Time-share a single port
  - Each access needs to be (significantly) shorter than clock cycle
  - Used in Alpha 21264
  - Is this scalable?

Handling Multiple Accesses per Cycle (III)

- Multiple cache copies
  - Stores update both caches
  - Loads proceed in parallel
- Used in Alpha 21164
- Scalability?
  - Store operations cause a bottleneck
  - Area proportional to “ports”



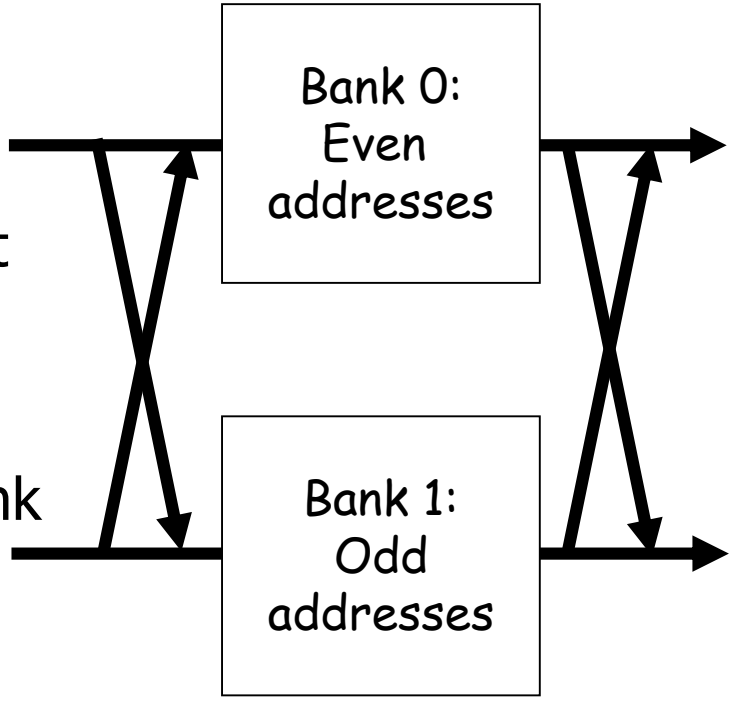
Handling Multiple Accesses per Cycle (III)

- Banking (Interleaving)
  - Address space partitioned into separate banks
    - Bits in address determines which bank an address maps to
    - Which bits to use for "bank address"?

  + No increase in data store area

  -- Cannot satisfy multiple accesses to the same bank in parallel

  -- Crossbar interconnect in input/output



- Bank conflicts
  - Concurrent requests to the same bank
  - How can these be reduced?
    - Hardware? Software?

General Principle: Interleaving

- Interleaving (banking)
  - Problem: a single monolithic memory array takes long to access and does not enable multiple accesses in parallel
  - Goal: Reduce the latency of memory array access and enable multiple accesses in parallel
  - Idea: Divide the array into multiple banks that can be accessed independently (in the same cycle or in consecutive cycles)
    - Each bank is smaller than the entire memory storage
    - Access latencies to different banks can be overlapped
  - A Key Issue: How do you map data to different banks? (i.e., how do you interleave data across banks?)

Further Readings on Caching and MLP

- Required: Qureshi et al., "A Case for MLP-Aware Cache Replacement," ISCA 2006.
- One Pager: Glew, "MLP Yes! ILP No!," ASPLOS Wild and Crazy Ideas Session, 1998.
- Mutlu et al., "Runahead Execution: An Effective Alternative to Large Instruction Windows," IEEE Micro 2003.
- Li et al., "Utility-based Hybrid Memory Management," CLUSTER 2017.
- Mutlu et al., "Parallelism-Aware Batch Scheduling," ISCA 2008

Computer Architecture

Lecture 3b: Memory Hierarchy and Caches

Prof. Onur Mutlu
ETH Zürich

Fall 2018

26 September 2018

We did not cover the following slides in lecture. These are for your preparation for the next lecture.

Efficient Cache Utilization

- Critical for performance, especially in multi-core systems
- Many works in this area
- Three sample works

- Qureshi et al., "A Case for MLP-Aware Cache Replacement," ISCA 2005.

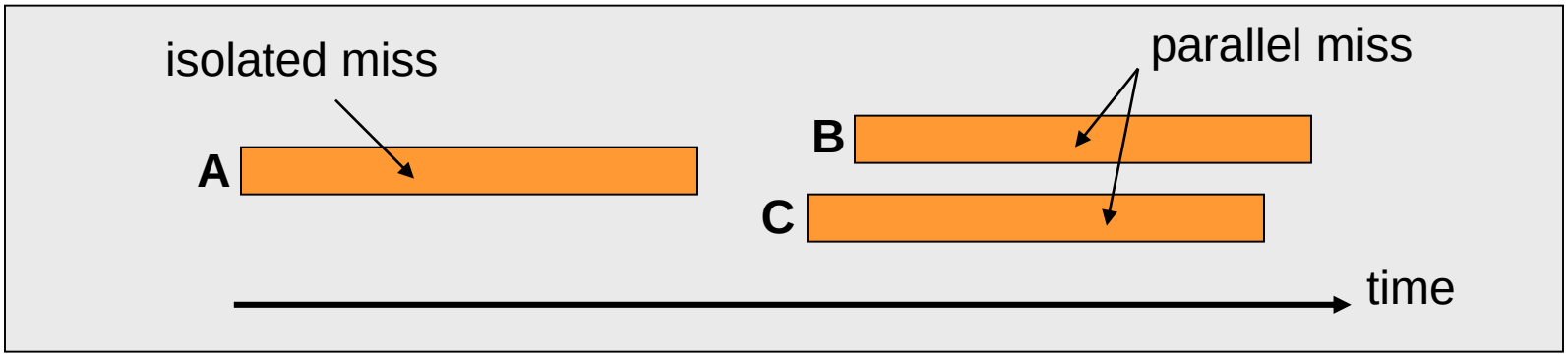
- Seshadri et al., "The Evicted-Address Filter: A Unified Mechanism to Address both Cache Pollution and Thrashing," PACT 2012.

- Pekhimenko et al., "Base-Delta-Immediate Compression: Practical Data Compression for On-Chip Caches," PACT 2012.

MLP-Aware Cache Replacement

Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu, and Yale N. Patt,
"A Case for MLP-Aware Cache Replacement"
*Proceedings of the 33rd International Symposium on Computer Architecture (**ISCA**)*, pages 167-177, Boston, MA, June 2006. Slides (ppt)

Memory Level Parallelism (MLP)



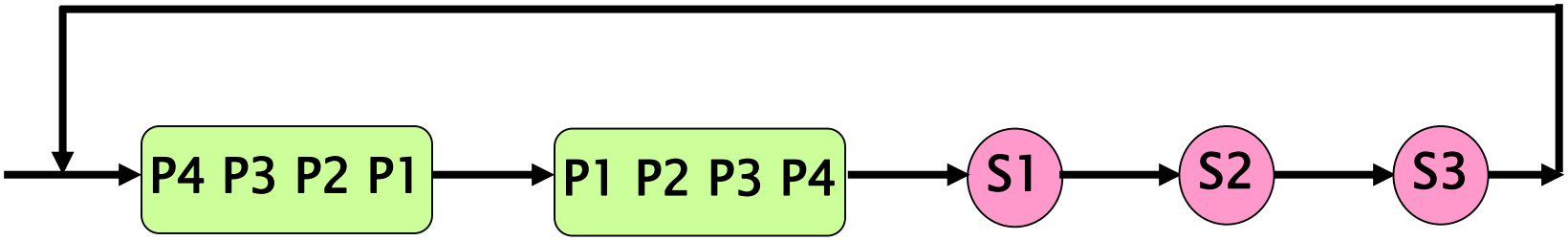
- Memory Level Parallelism (MLP) means generating and servicing multiple memory accesses in parallel [Glew' 98]
- Several techniques to improve MLP (e.g., out-of-order execution, runahead execution)
- MLP varies. Some misses are isolated and some parallel

How does this affect cache replacement?

Traditional Cache Replacement Policies

- Traditional cache replacement policies try to reduce miss count
- Implicit assumption: Reducing miss count reduces memory-related stall time
- Misses with varying cost/MLP breaks this assumption!
- Eliminating an isolated miss helps performance more than eliminating a parallel miss
- Eliminating a higher-latency miss could help performance more than eliminating a lower-latency miss

An Example



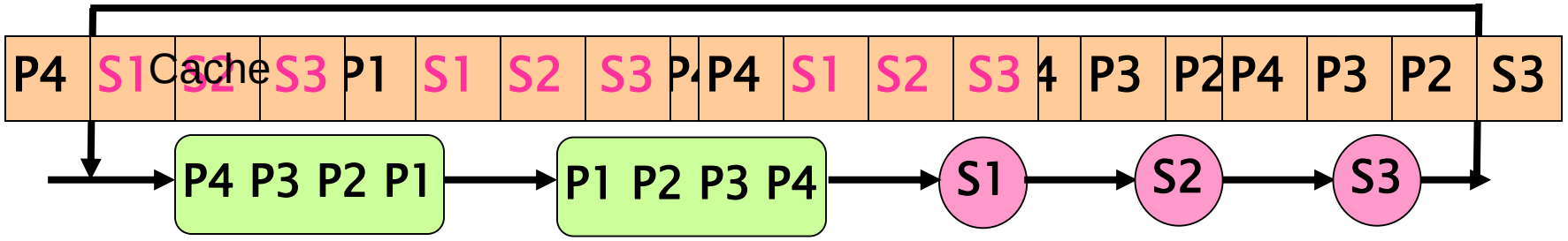
Misses to blocks P1, P2, P3, P4 can be parallel
Misses to blocks S1, S2, and S3 are isolated

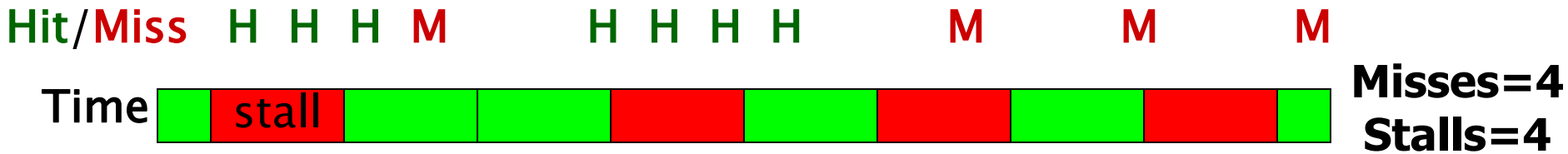
Two replacement algorithms:

1. Minimizes miss count (Belady's OPT)
2. Reduces isolated misses (MLP-Aware)

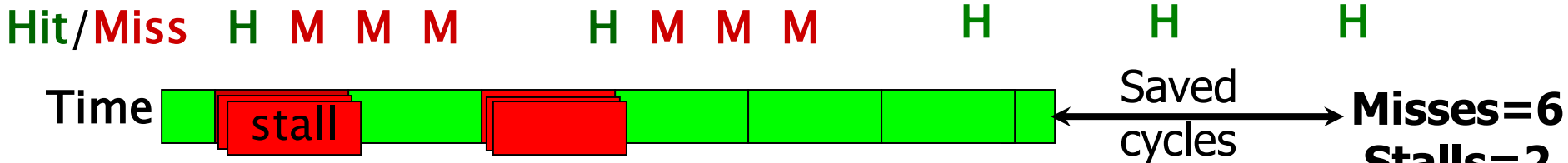
For a fully associative cache containing 4 blocks

Fewest Misses ≠ Best Performance





Belady's OPT replacement



MLP-Aware replacement

Motivation

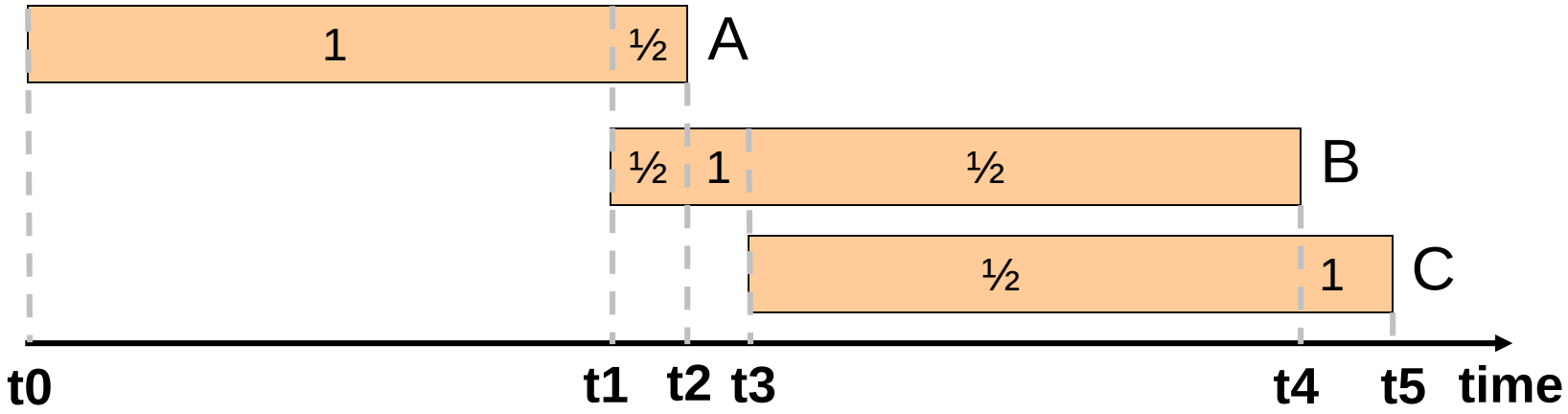
- MLP varies. Some misses more costly than others
- MLP-aware replacement can improve performance by reducing costly misses

Outline

- Introduction
- MLP-Aware Cache Replacement
  - Model for Computing Cost
  - Repeatability of Cost
  - A Cost-Sensitive Replacement Policy
- Practical Hybrid Replacement
  - Tournament Selection
  - Dynamic Set Sampling
  - Sampling Based Adaptive Replacement
- Summary

Computing MLP-Based Cost

- Cost of miss is number of cycles the miss stalls the processor
- Easy to compute for isolated miss
- Divide each stall cycle equally among all parallel misses



A First-Order Model

- Miss Status Holding Register (MSHR) tracks all in flight misses
- Add a field mlp-cost to each MSHR entry
- Every cycle for each demand entry in MSHR

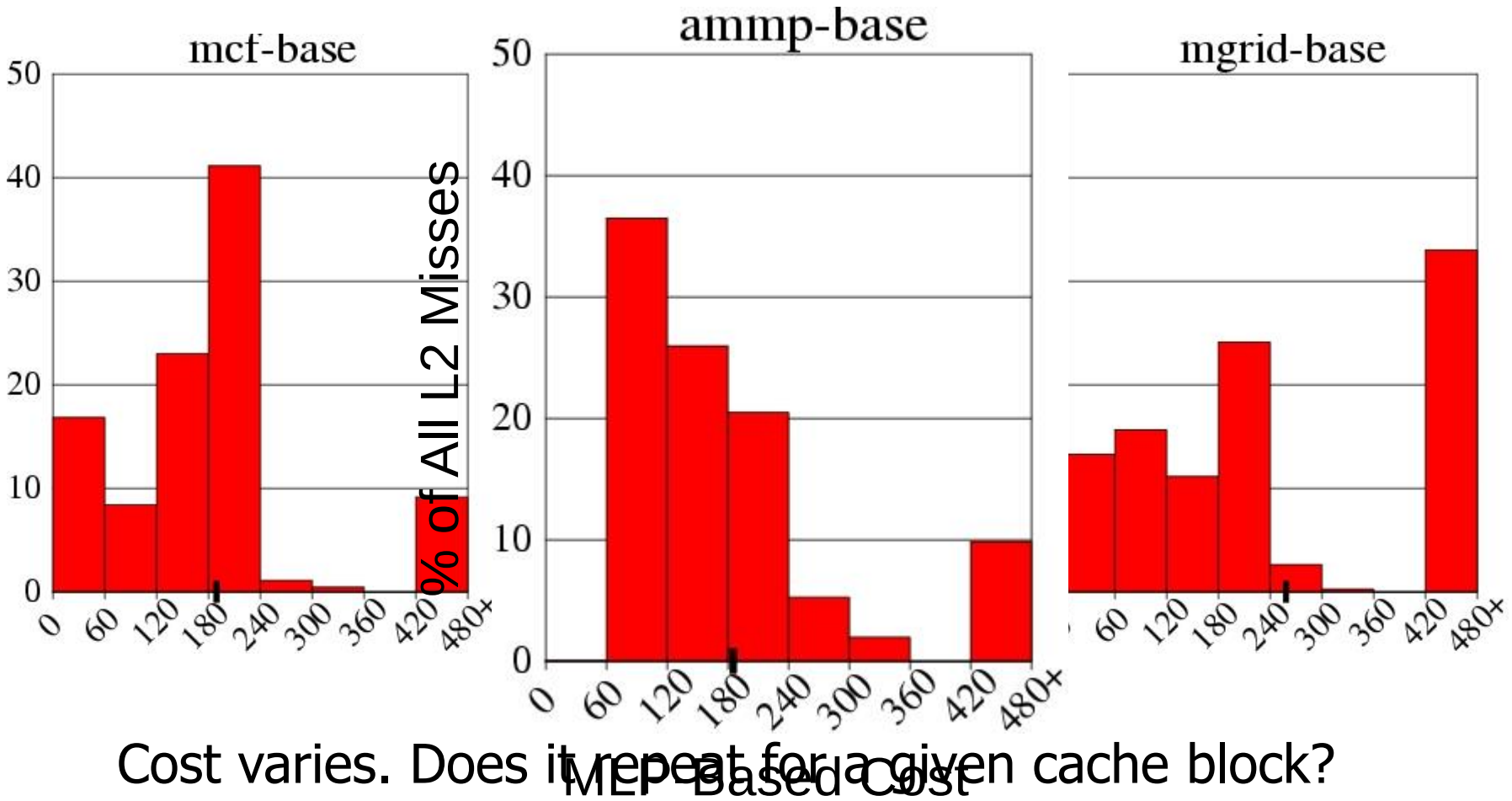
mlp-cost += (1/N)

N = Number of demand misses in MSHR

Machine Configuration

- Processor
  - aggressive, out-of-order, 128-entry instruction window
- L2 Cache
  - 1MB, 16-way, LRU replacement, 32 entry MSHR
- Memory
  - 400 cycle bank access, 32 banks
- Bus
  - Roundtrip delay of 11 bus cycles (44 processor cycles)

Distribution of MLP-Based Cost



Cost varies. Does it repeat for a given cache block?

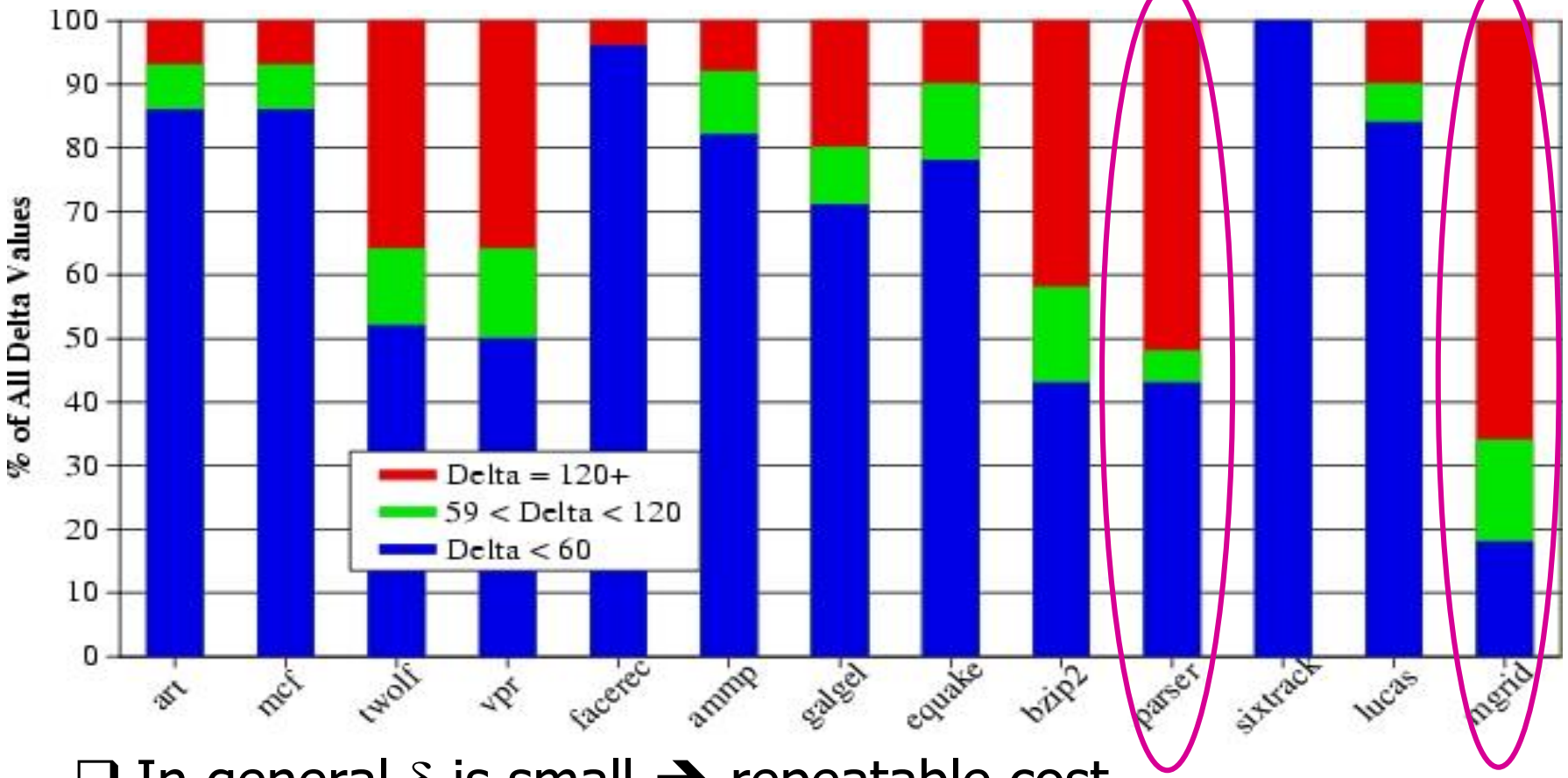
Repeatability of Cost

- An isolated miss can be parallel miss next time
- Can current cost be used to estimate future cost ?
- Let δ = difference in cost for successive miss to a block
  - Small δ ➔ cost repeats
  - Large δ ➔ cost varies significantly

Repeatability of Cost

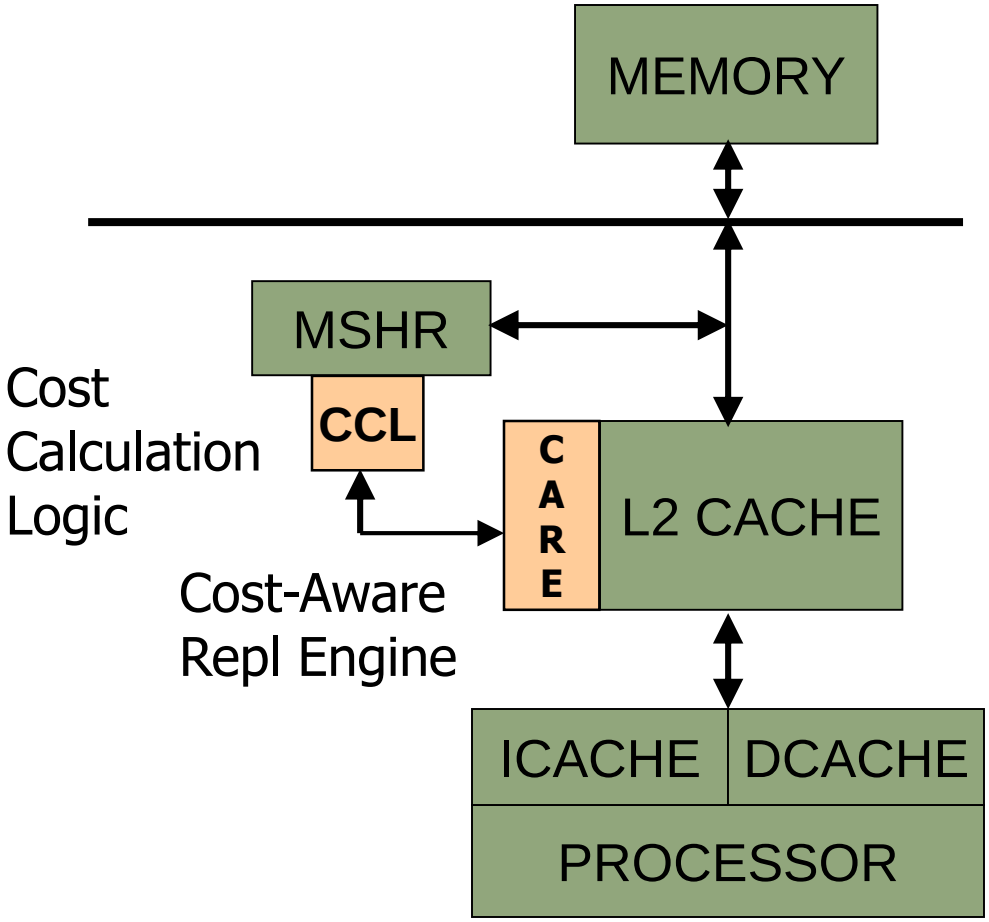
$\delta < 60$ $\delta > 120$

$59 < \delta < 120$



- In general δ is small ➔ repeatable cost
- When δ is large (e.g. parser, mgrid) ➔ performance loss

The Framework



Quantization of Cost
Computed mlp-based cost is quantized to a 3-bit value

Design of MLP-Aware Replacement policy

- LRU considers only recency and no cost

$$Victim\text{-}LRU = min \{ Recency(i) \}$$

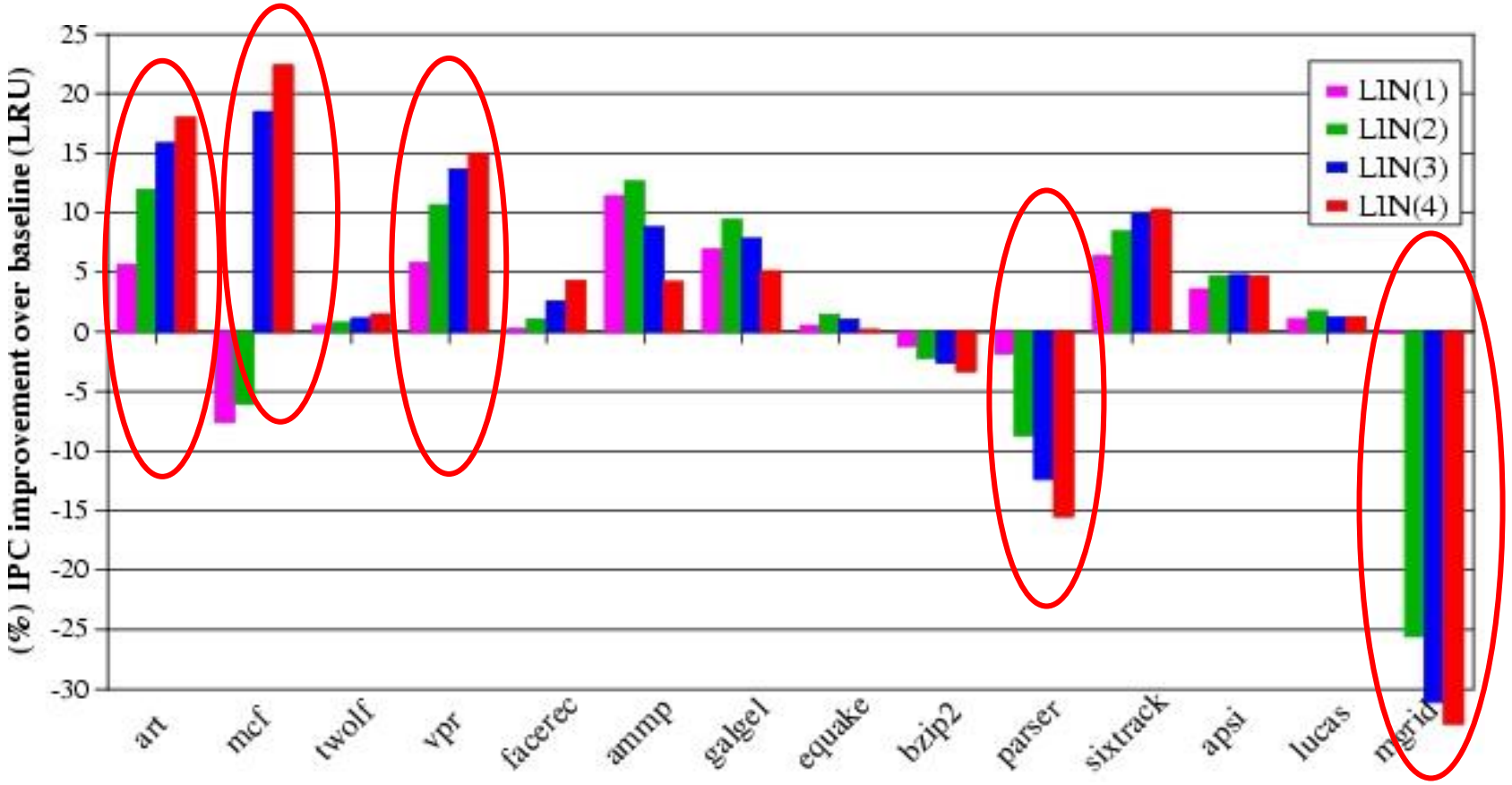
- Decisions based only on cost and no recency hurt performance. Cache stores useless high cost blocks

- A Linear (LIN) function that considers recency and cost

$$Victim\text{-}LIN = min \{ Recency(i) + S*cost(i) \}$$

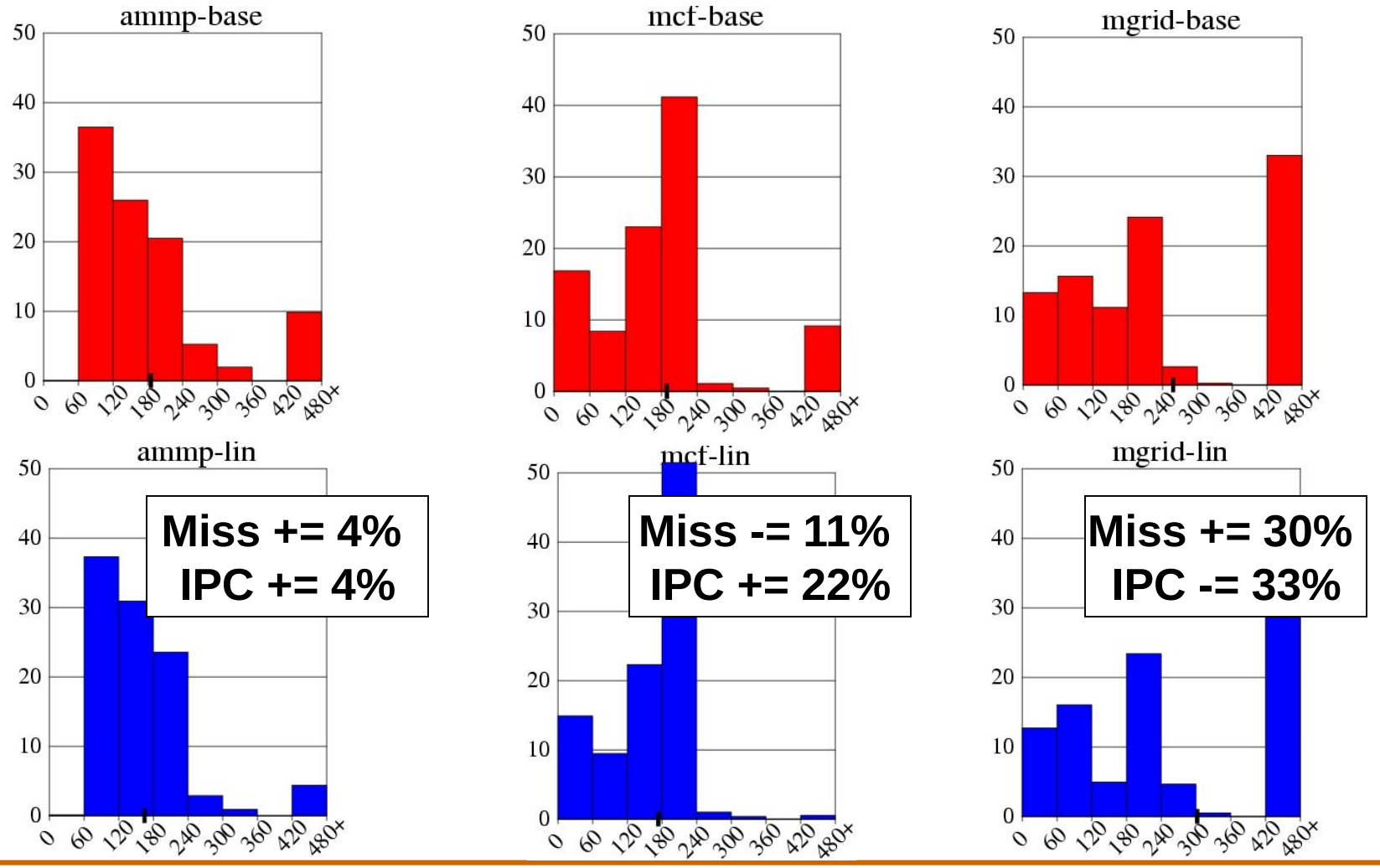
S = significance of cost. Recency(i) = position in LRU stack
cost(i) = quantized cost

Results for the LIN policy



Performance loss for parser and mgrid due to large δ

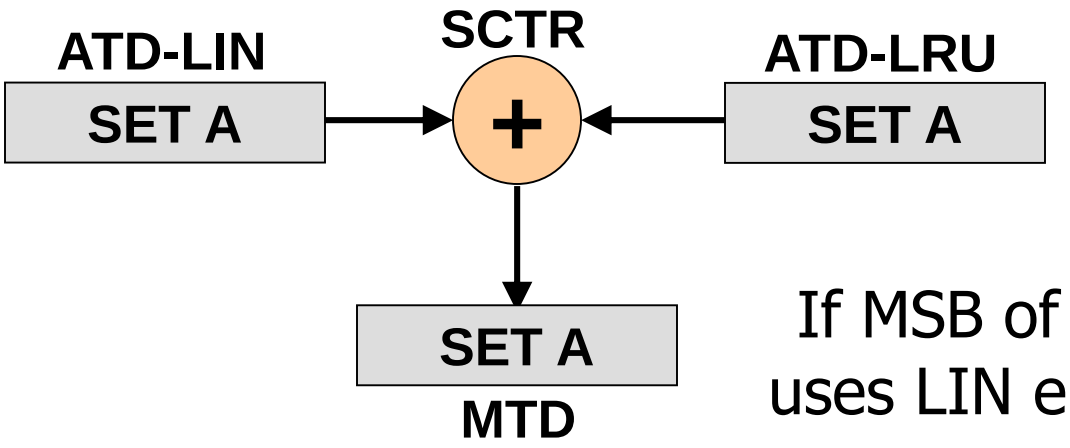
Effect of LIN policy on Cost



Outline

- Introduction
- MLP-Aware Cache Replacement
  - Model for Computing Cost
  - Repeatability of Cost
  - A Cost-Sensitive Replacement Policy
- Practical Hybrid Replacement
  - Tournament Selection
  - Dynamic Set Sampling
  - Sampling Based Adaptive Replacement
- Summary

Tournament Selection (TSEL) of Replacement Policies for a Single Set



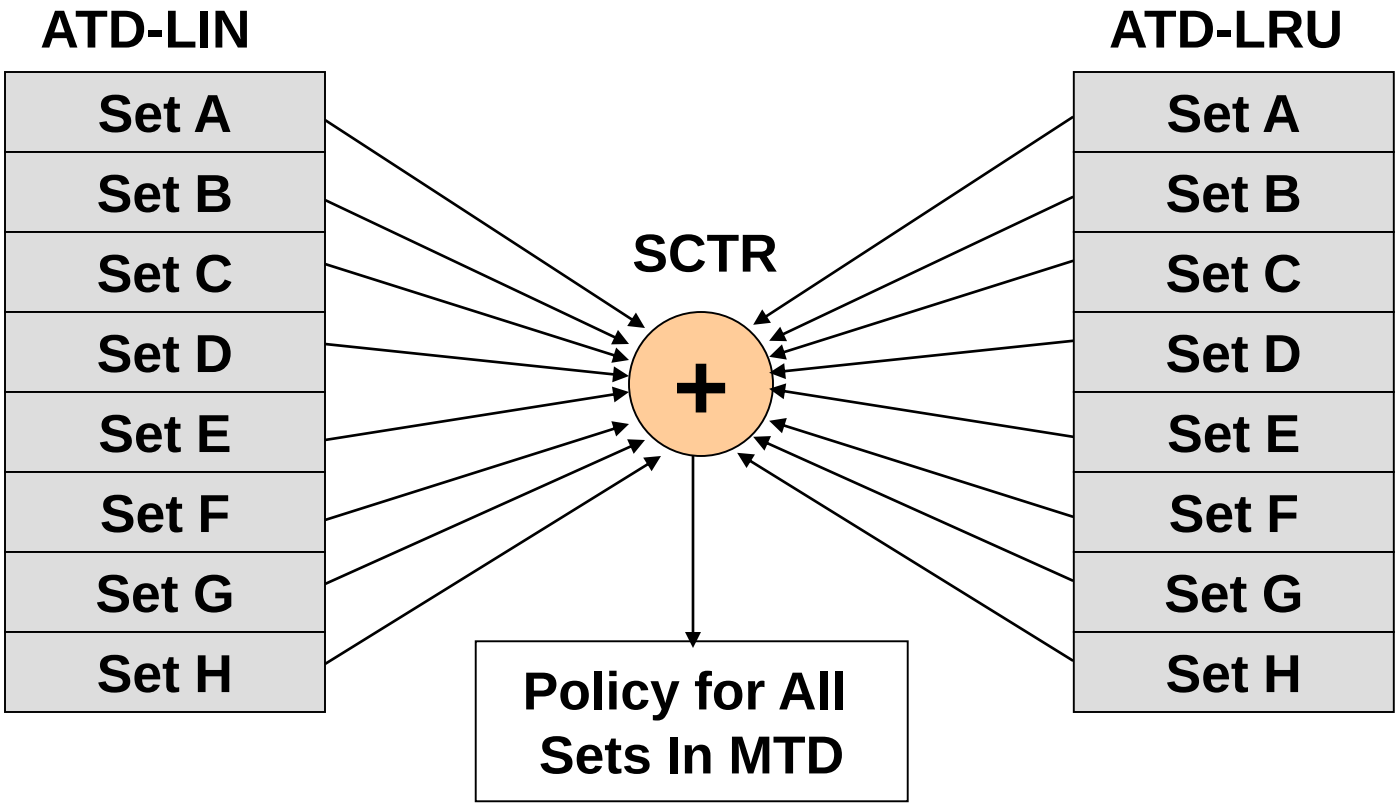
If MSB of SCTR is 1, MTD uses LIN else MTD use LRU

ATD-LIN	ATD-LRU	Saturating Counter (SCTR)
HIT	HIT	Unchanged
MISS	MISS	Unchanged
HIT	MISS	+= Cost of Miss in ATD-LRU
MISS	HIT	-= Cost of Miss in ATD-LIN

Extending TSEL to All Sets

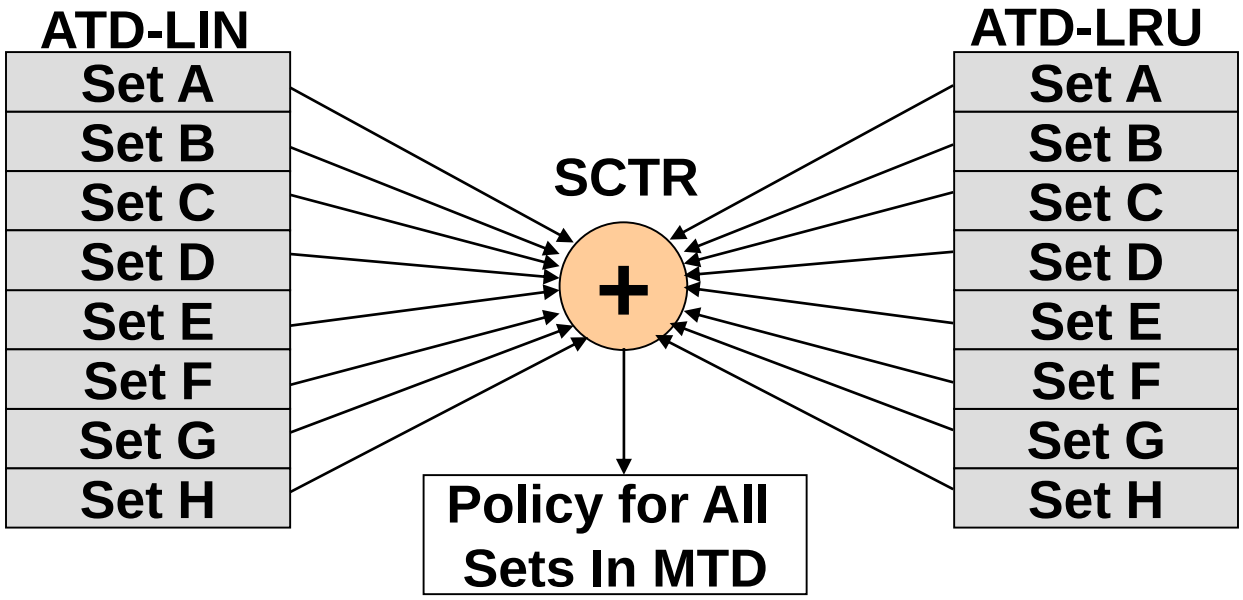
Implementing TSEL on a per-set basis is expensive

Counter overhead can be reduced by using a global counter



Dynamic Set Sampling

Not all sets are required to decide the best policy
Have the ATD entries only for few sets.



Sets that have ATD entries (B, E, G) are called leader sets

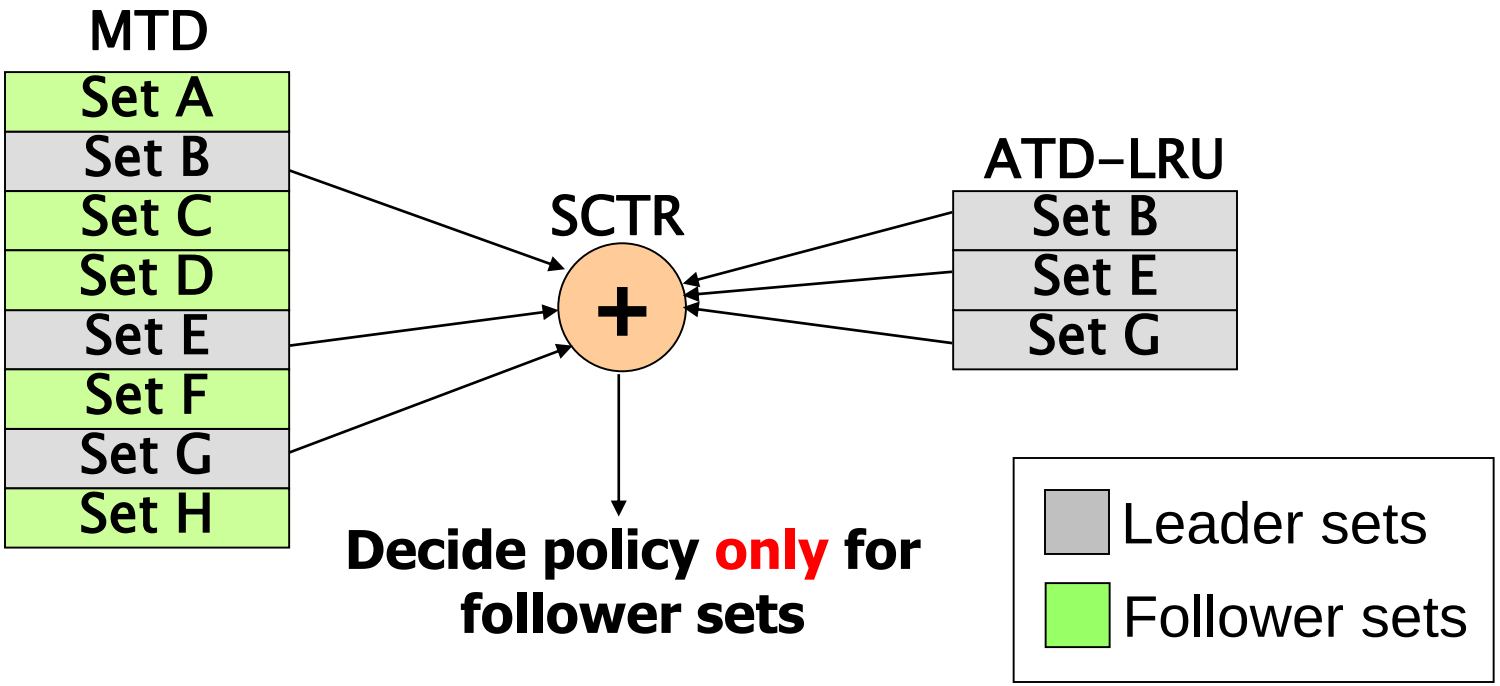
Dynamic Set Sampling

How many sets are required to choose best performing policy?

- Bounds using analytical model and simulation (in paper)
- DSS with 32 leader sets performs similar to having all sets
- Last-level cache typically contains 1000s of sets, thus ATD entries are required for only 2%-3% of the sets

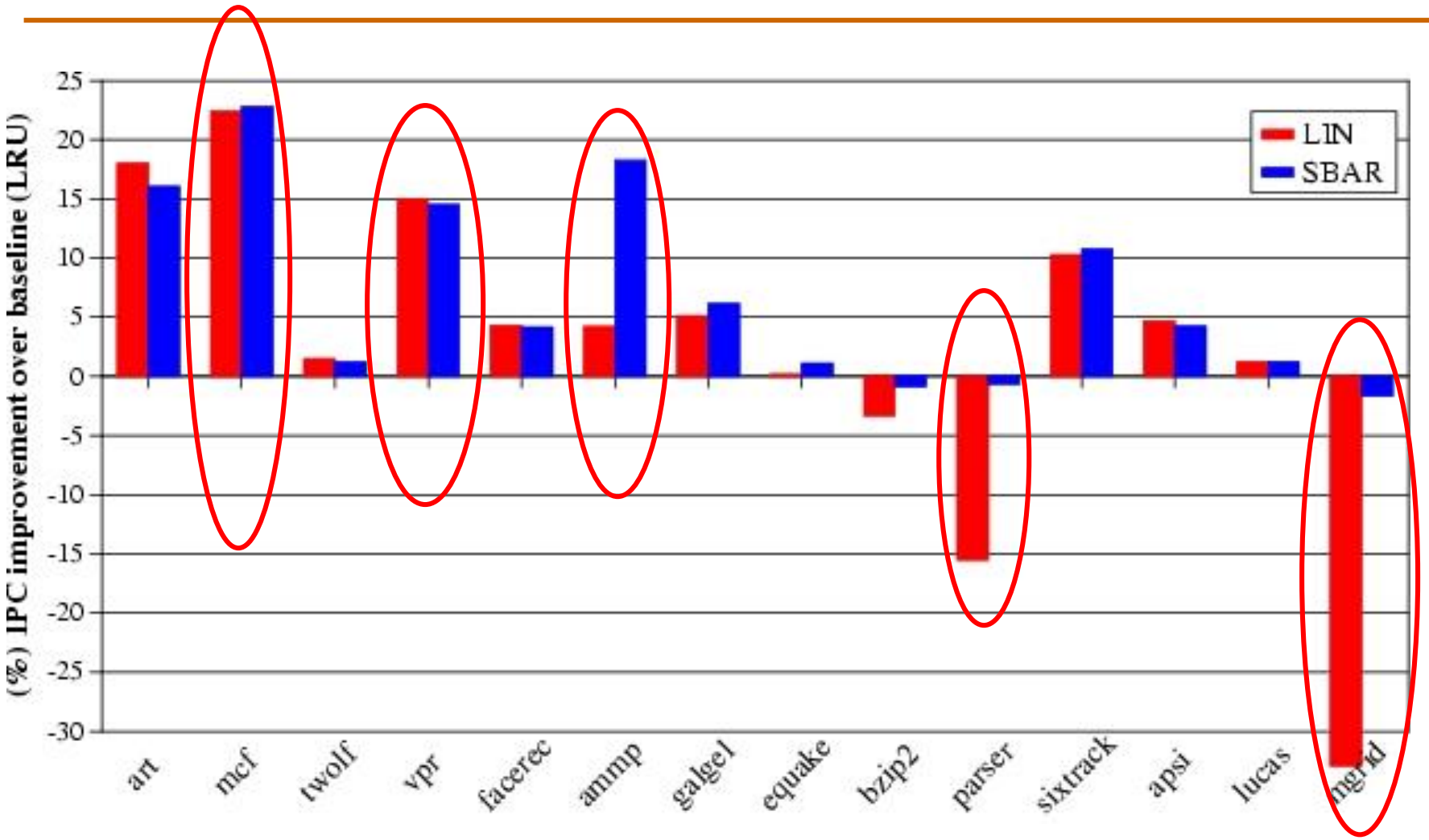
ATD overhead can further be reduced by using MTD to always simulate one of the policies (say LIN)

Sampling Based Adaptive Replacement (SBAR)

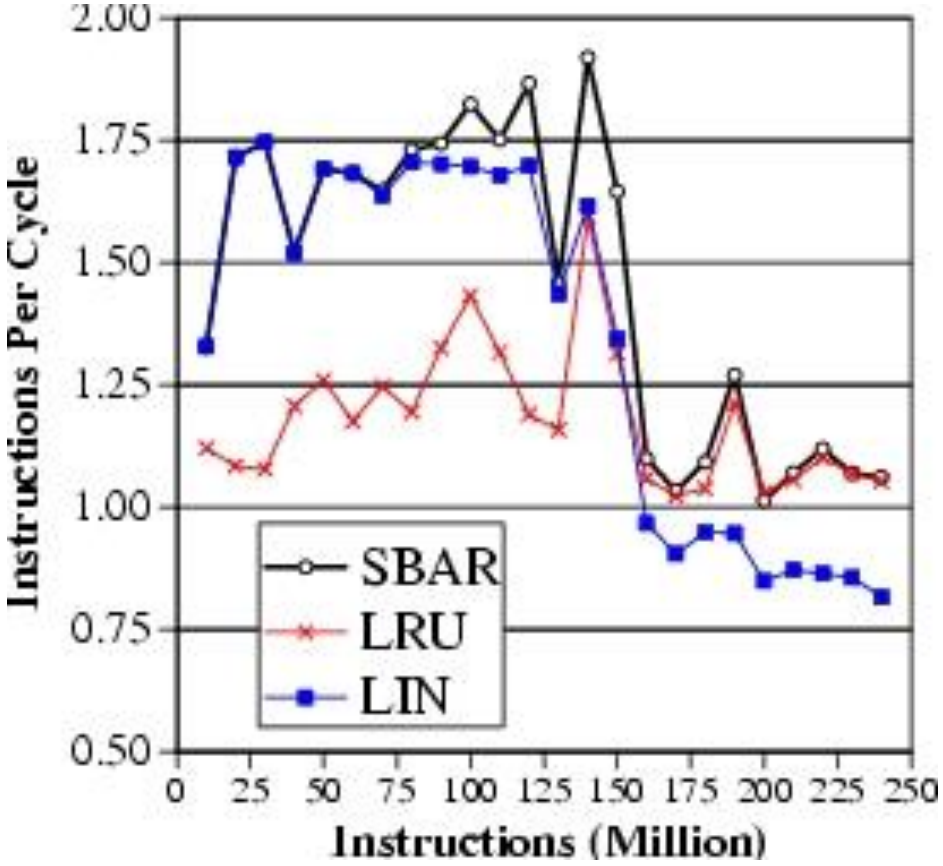


The storage overhead of SBAR is less than 2KB
(0.2% of the baseline 1MB cache)

Results for SBAR



SBAR adaptation to phases



SBAR selects the best policy for each phase of ammp

Outline

- Introduction
- MLP-Aware Cache Replacement
  - Model for Computing Cost
  - Repeatability of Cost
  - A Cost-Sensitive Replacement Policy
- Practical Hybrid Replacement
  - Tournament Selection
  - Dynamic Set Sampling
  - Sampling Based Adaptive Replacement
- Summary

Summary

- MLP varies. Some misses are more costly than others
- MLP-aware cache replacement can reduce costly misses
- Proposed a runtime mechanism to compute MLP-Based cost and the LIN policy for MLP-aware cache replacement
- SBAR allows dynamic selection between LIN and LRU with low hardware overhead
- Dynamic set sampling used in SBAR also enables other cache related optimizations

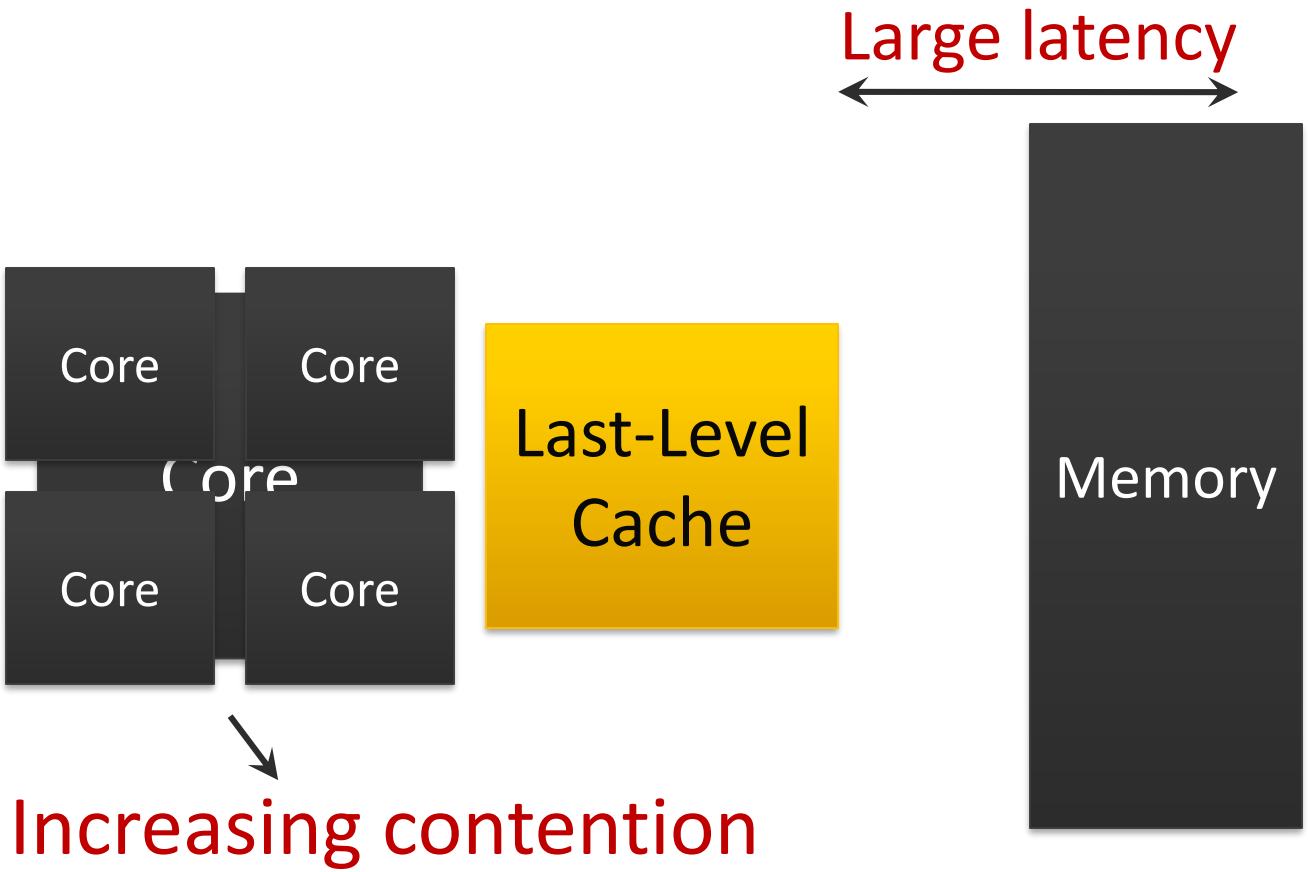
The Evicted-Address Filter

Vivek Seshadri, Onur Mutlu, Michael A. Kozuch, and Todd C. Mowry,
"The Evicted-Address Filter: A Unified Mechanism to Address Both Cache Pollution and Thrashing"
*Proceedings of the 21st ACM International Conference on Parallel Architectures and Compilation Techniques (**PACT**)*, Minneapolis, MN, September 2012. Slides (pptx)

Executive Summary

- Two problems degrade cache performance
  - Pollution and thrashing
  - Prior works don't address both problems concurrently
- Goal: A mechanism to address both problems
- EAF-Cache
  - Keep track of recently evicted block addresses in EAF
  - Insert low reuse with low priority to mitigate pollution
  - Clear EAF periodically to mitigate thrashing
  - Low complexity implementation using Bloom filter
- EAF-Cache outperforms five prior approaches that address pollution or thrashing

Cache Utilization is Important



Effective cache utilization is important

Reuse Behavior of Cache Blocks

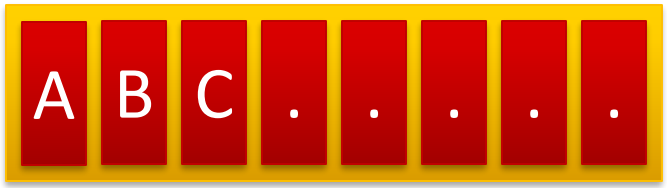
Different blocks have different reuse behavior

Access Sequence:


High-reuse block


Low-reuse block

Ideal Cache



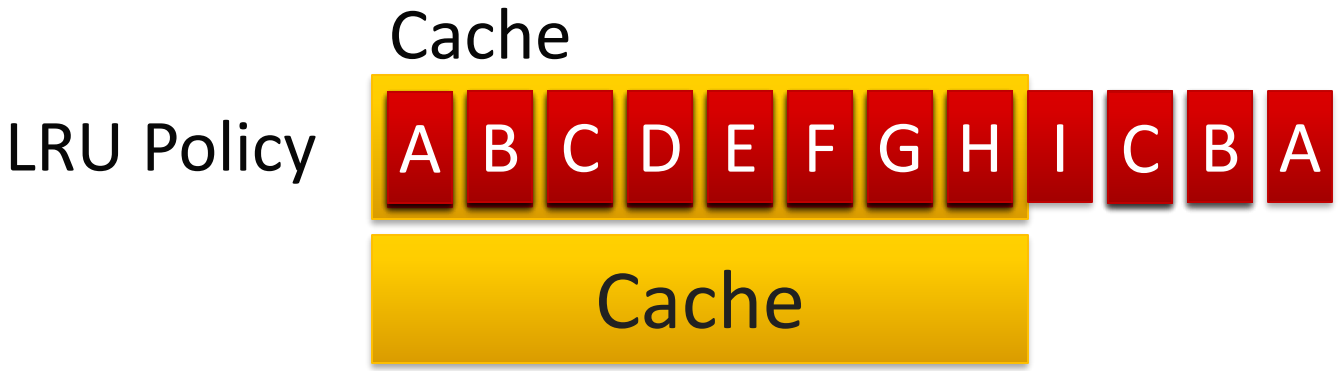
Cache Pollution

Problem: Low-reuse blocks evict high-reuse blocks

Prior work: Predict reuse behavior of missed blocks. Insert low-reuse blocks at LRU position.

Cache Thrashing

Problem: High-reuse blocks evict each other



Prior work: Insert at MRU position with a very low probability (**Bimodal insertion policy**)

Shortcomings of Prior Works

Prior works do not address both pollution and thrashing concurrently

Prior Work on Cache Pollution

No control on the number of blocks inserted with high priority into the cache

Prior Work on Cache Thrashing

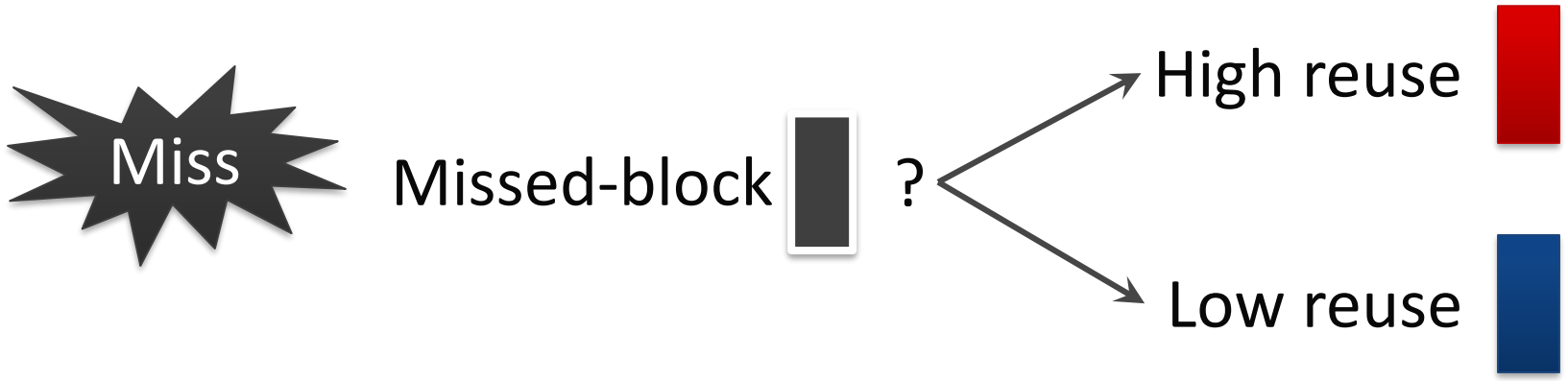
No mechanism to distinguish high-reuse blocks from low-reuse blocks

Our goal: Design a mechanism to address both pollution and thrashing concurrently

Outline

- Background and Motivation
- Evicted-Address Filter
  - Reuse Prediction
  - Thrash Resistance
- Final Design
- Advantages and Disadvantages
- Evaluation
- Conclusion

Reuse Prediction



Keep track of the reuse behavior of every cache block in the system

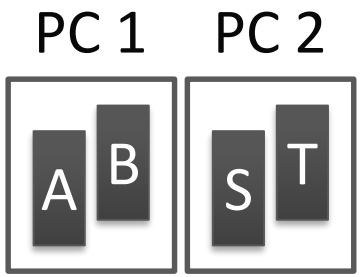
Impractical

1. High storage overhead
2. Look-up latency

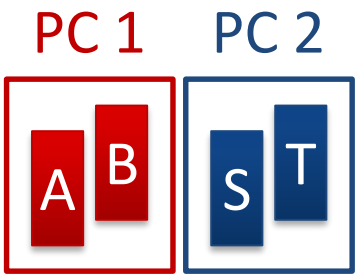
Prior Work on Reuse Prediction

Use program counter or memory region information.

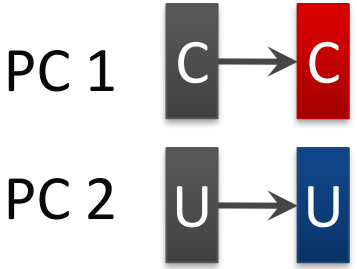
1. Group Blocks



2. Learn group behavior



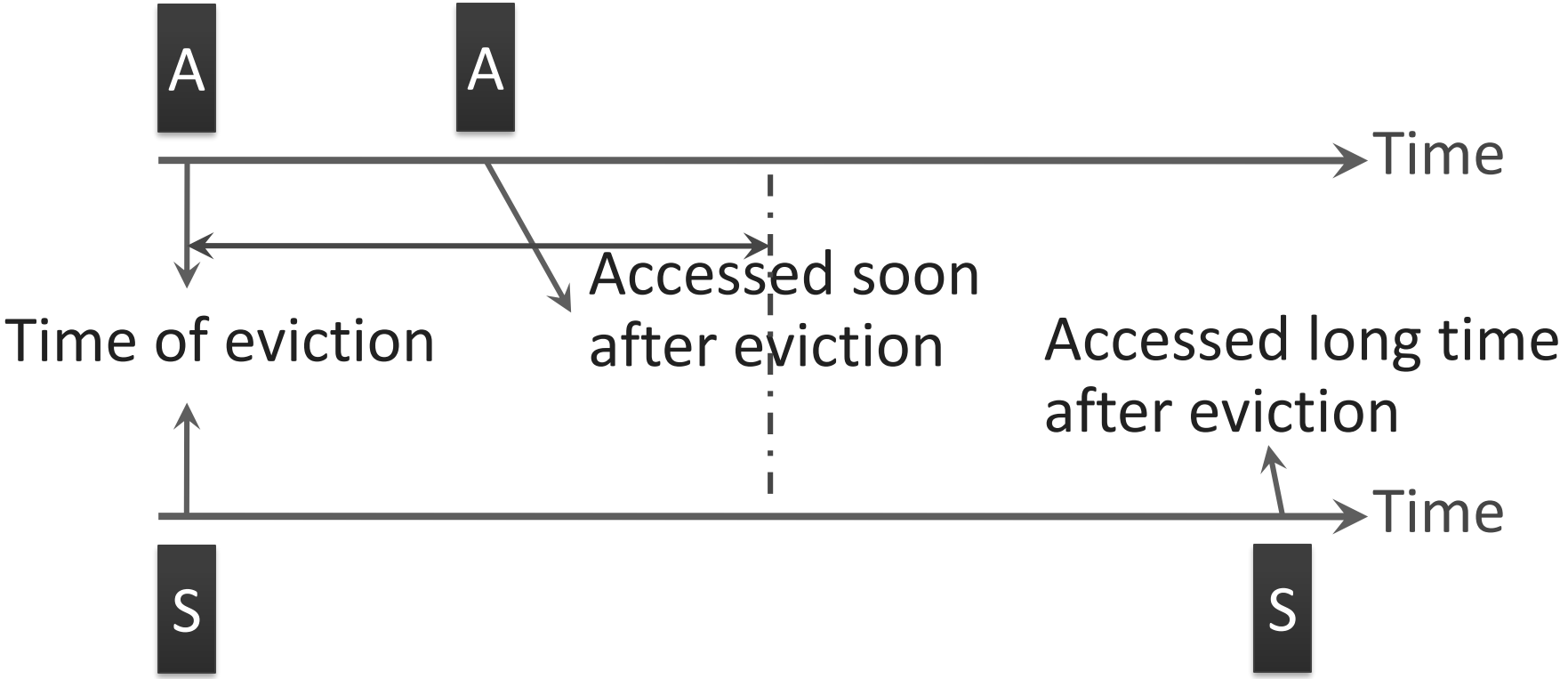
3. Predict reuse



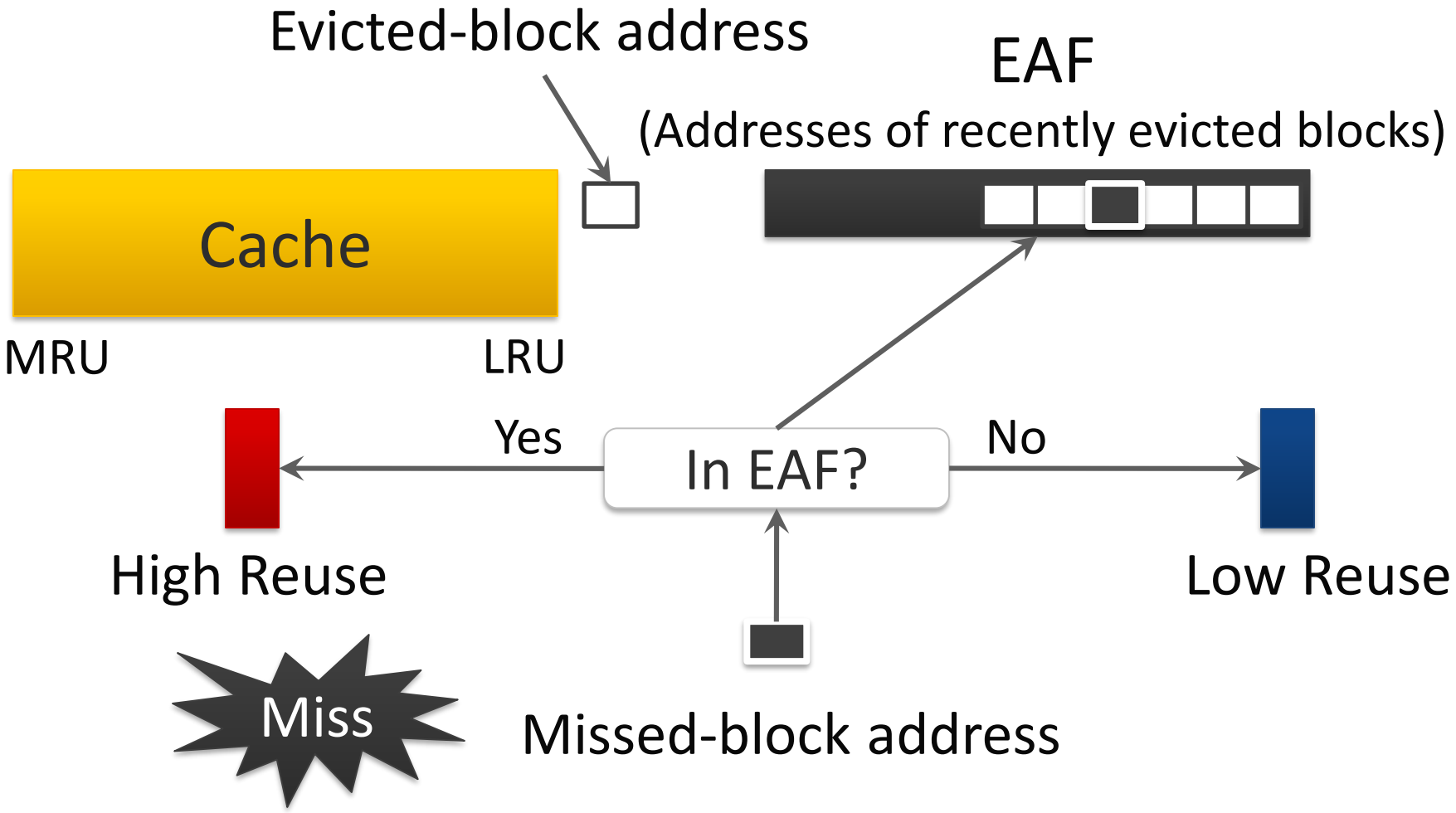
1. Same group $\not\rightarrow$ same reuse behavior
2. No control over number of high-reuse blocks

Our Approach: Per-block Prediction

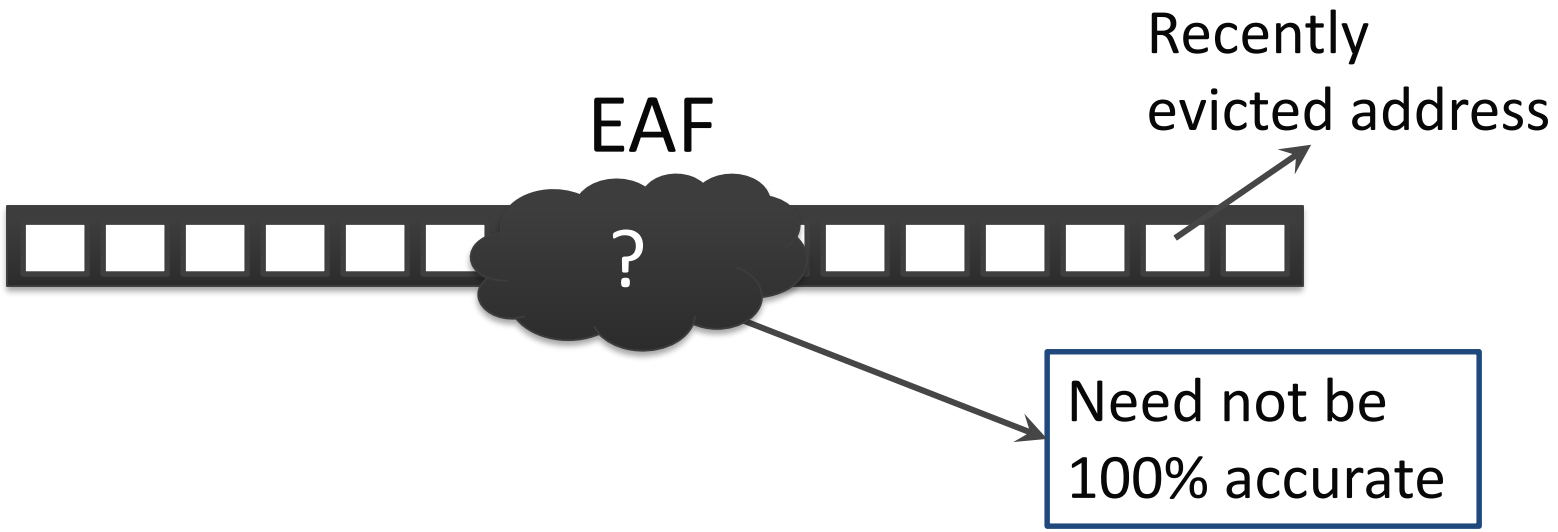
Use recency of eviction to predict reuse



Evicted-Address Filter (EAF)

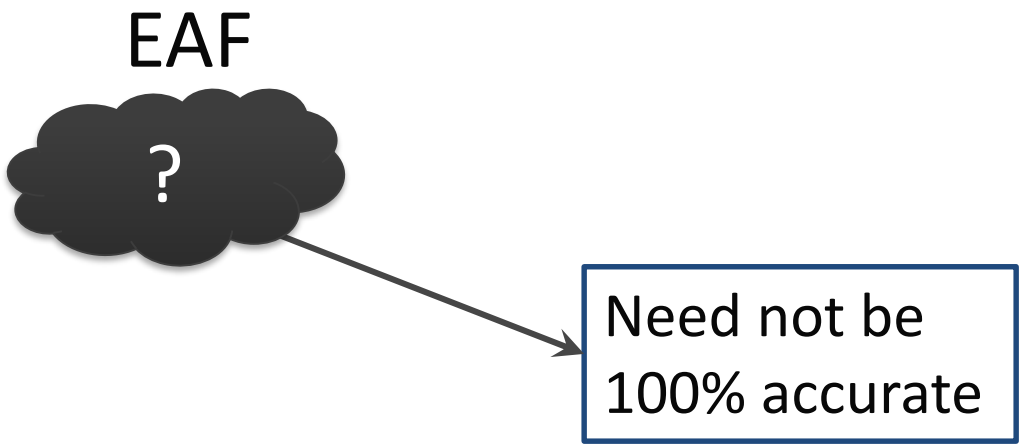


Naïve Implementation: Full Address Tags



1. Large storage overhead
2. Associative lookups – High energy

Low-Cost Implementation: Bloom Filter

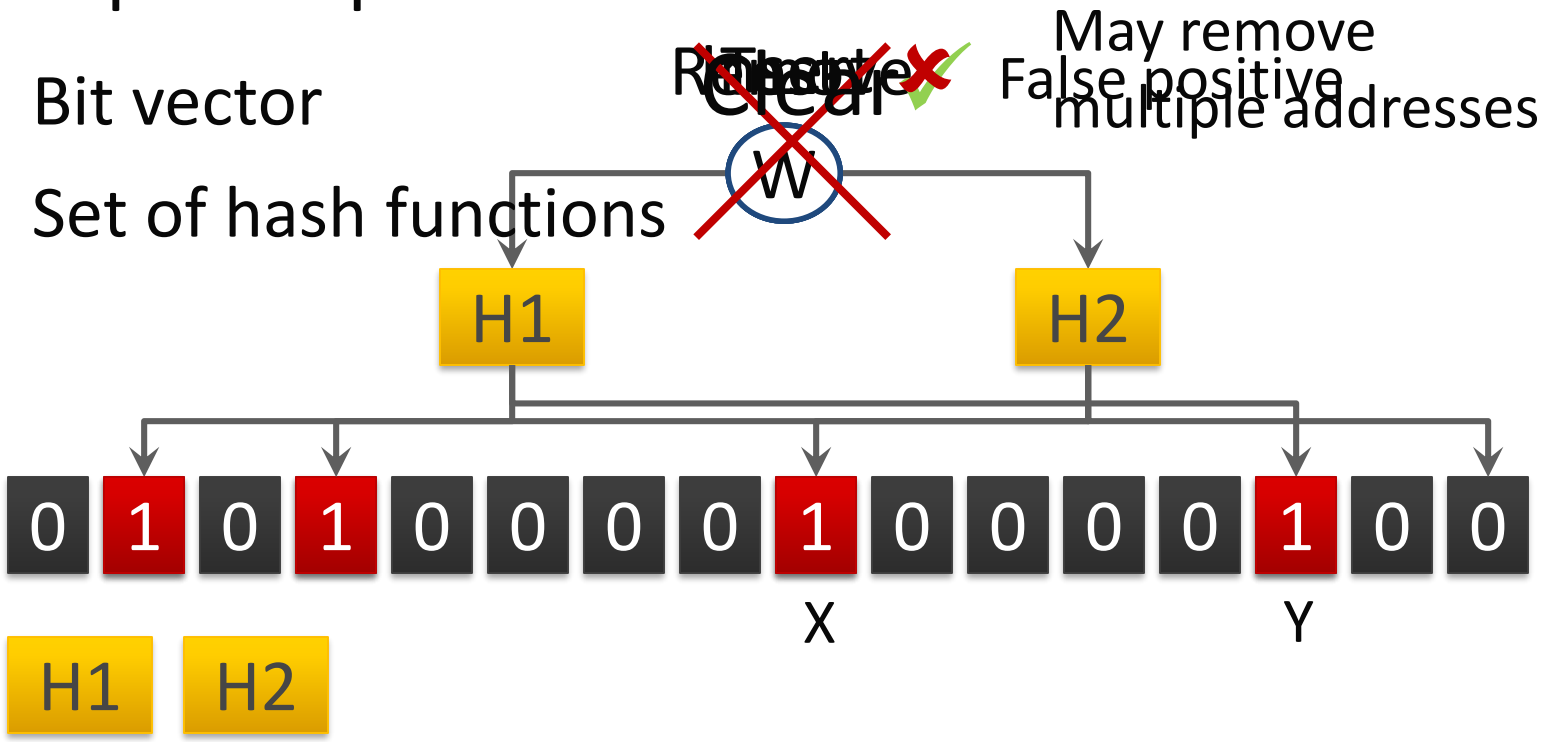


Implement EAF using a **Bloom Filter**
Low storage overhead + energy

Bloom Filter

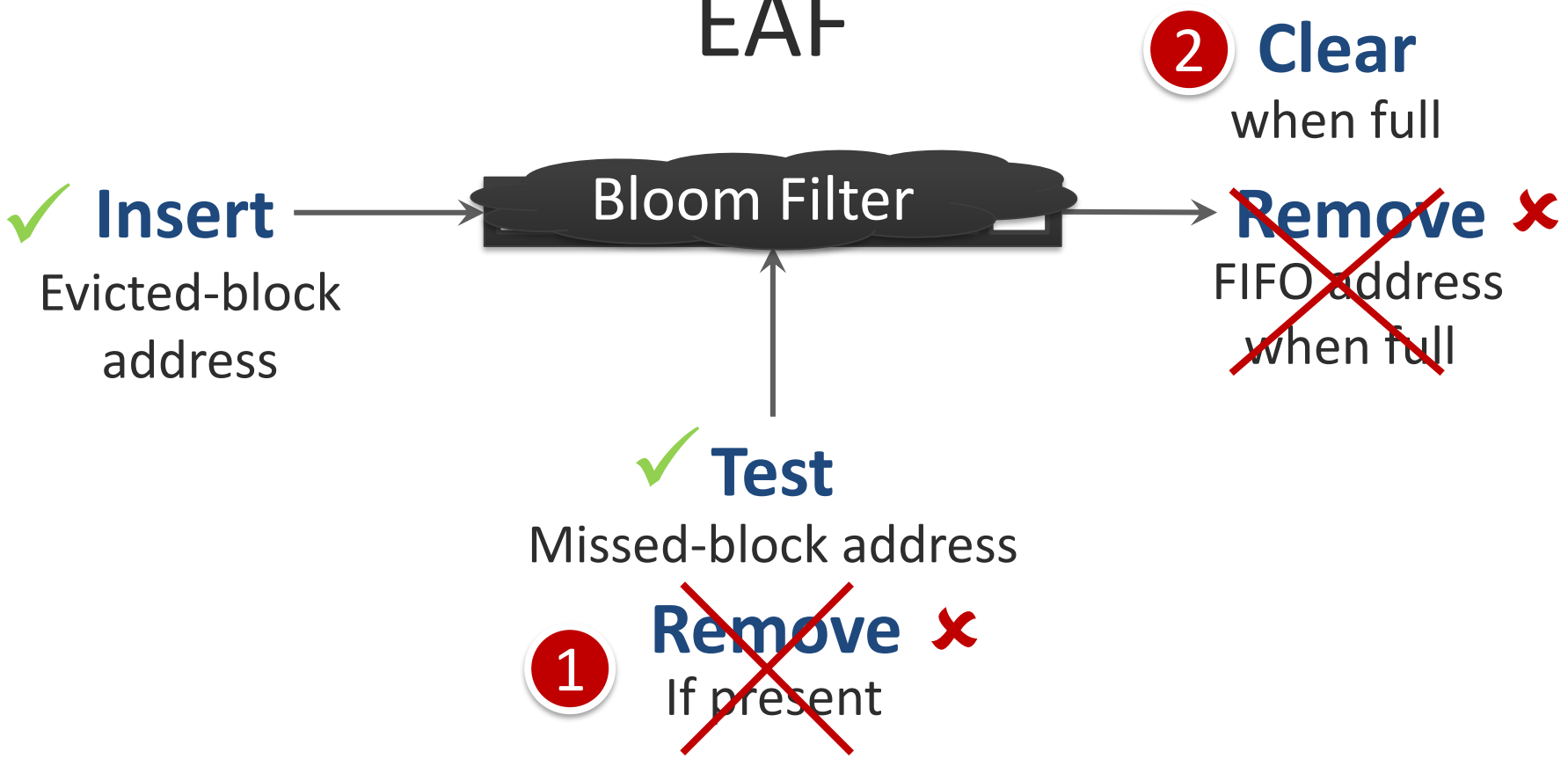
Compact representation of a set

1. Bit vector
2. Set of hash functions



Inserted Elements: X Y

EAF using a Bloom Filter



Bloom-filter EAF: 4x reduction in storage overhead, 1.47% compared to cache size

Outline

- Background and Motivation
- Evicted-Address Filter
  - Reuse Prediction
  - Thrash Resistance
- Final Design
- Advantages and Disadvantages
- Evaluation
- Conclusion

Large Working Set: 2 Cases

1. Cache < Working set < Cache + EAF

2. Cache + EAF < Working Set

Large Working Set: Case 1

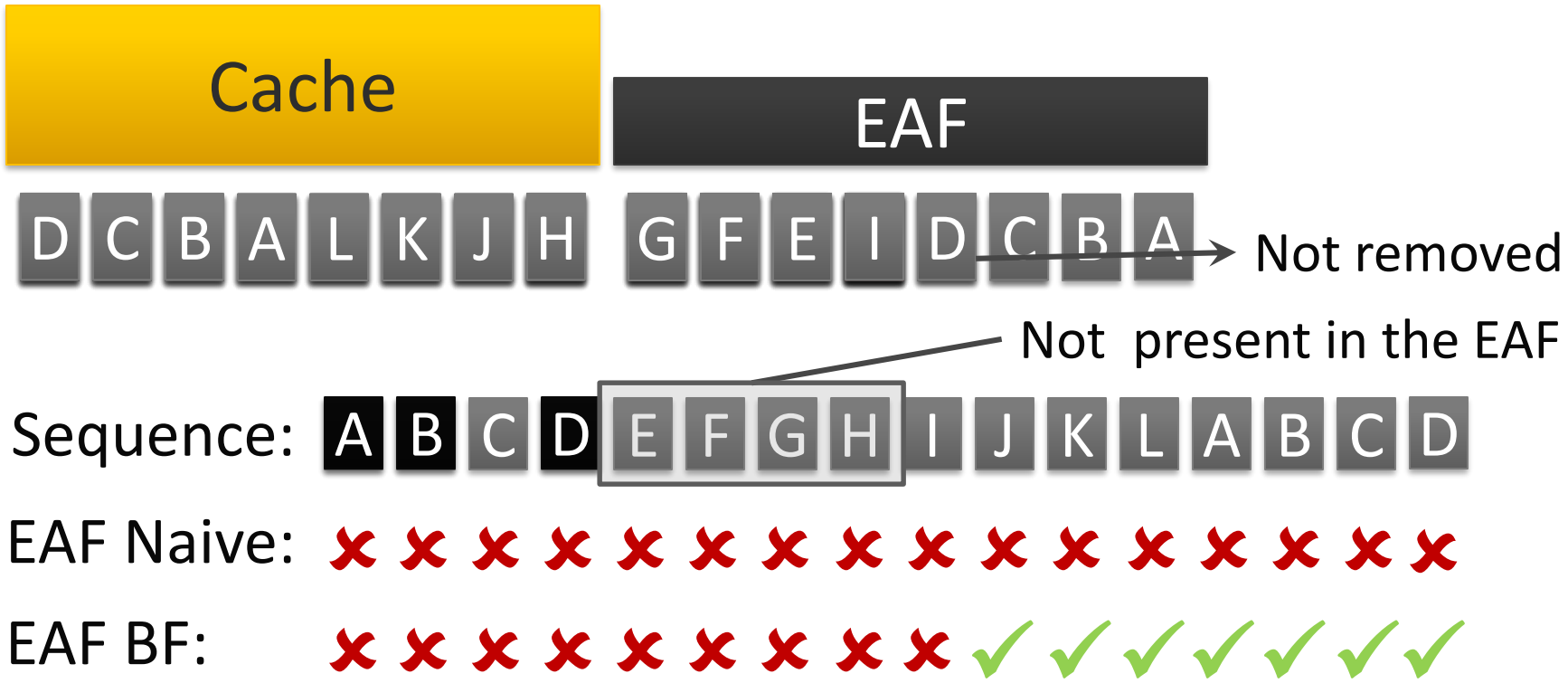
Cache < Working set < Cache + EAF

Sequence: A B C D E F G H I J K L A B C D

EAF Naive: ✗ ✗ ✗ ✗ ✗ ✗ ✗ ✗ ✗ ✗ ✗ ✗ ✗ ✗ ✗ ✗

Large Working Set: Case 1

Cache < Working set < Cache + EAF



Bloom-filter based EAF mitigates thrashing

Large Working Set: Case 2

Cache + EAF < Working Set

Problem: All blocks are predicted to have low reuse

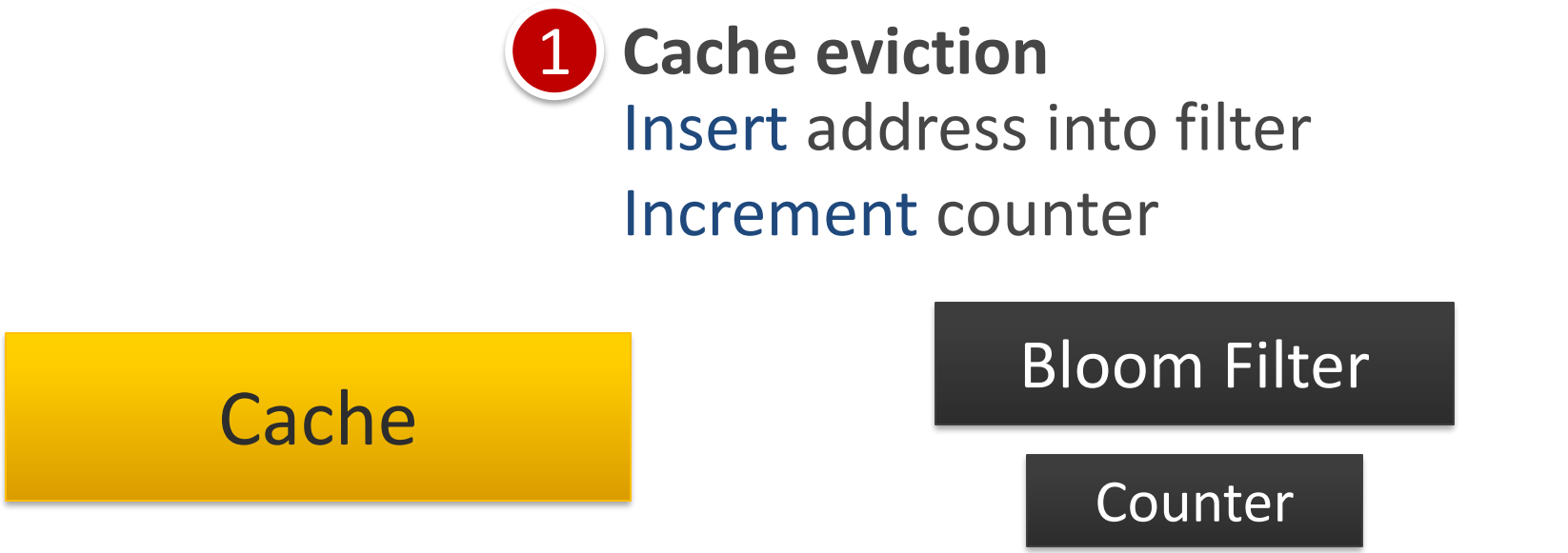
Allow a fraction of the working set to stay in the cache

Use **Bimodal Insertion Policy** for low reuse blocks. Insert few of them at the MRU position

Outline

- Background and Motivation
- Evicted-Address Filter
  - Reuse Prediction
  - Thrash Resistance
- Final Design
- Advantages and Disadvantages
- Evaluation
- Conclusion

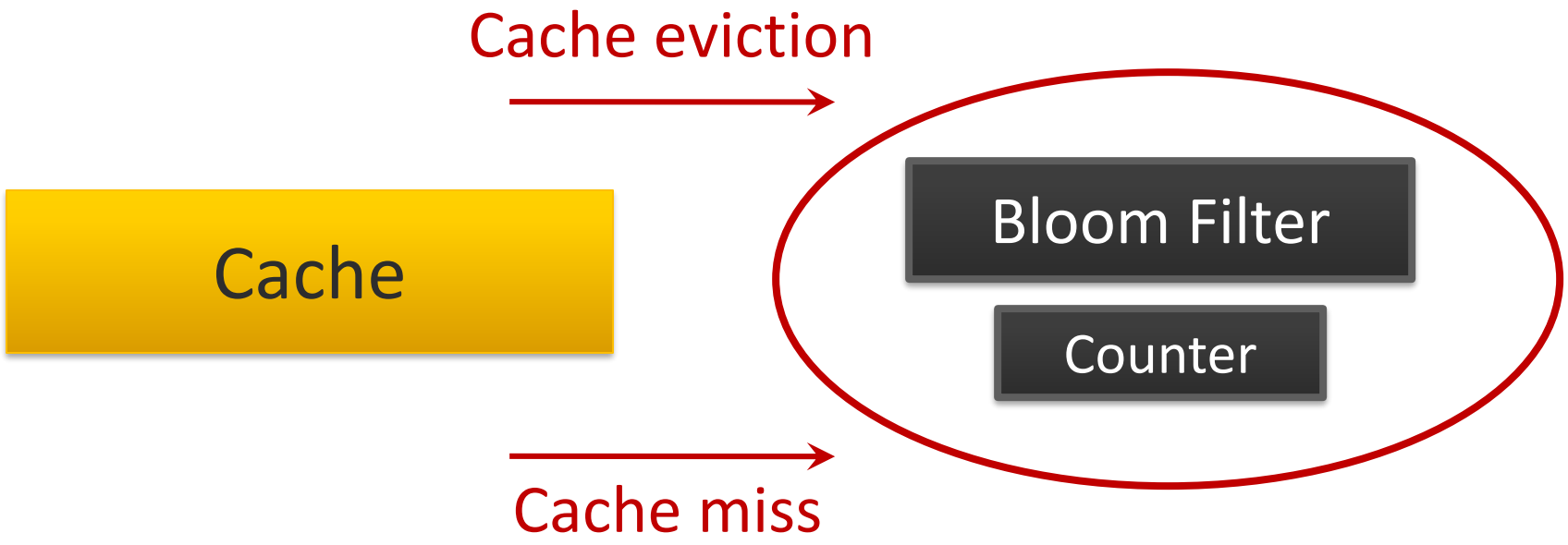
EAF-Cache: Final Design



Outline

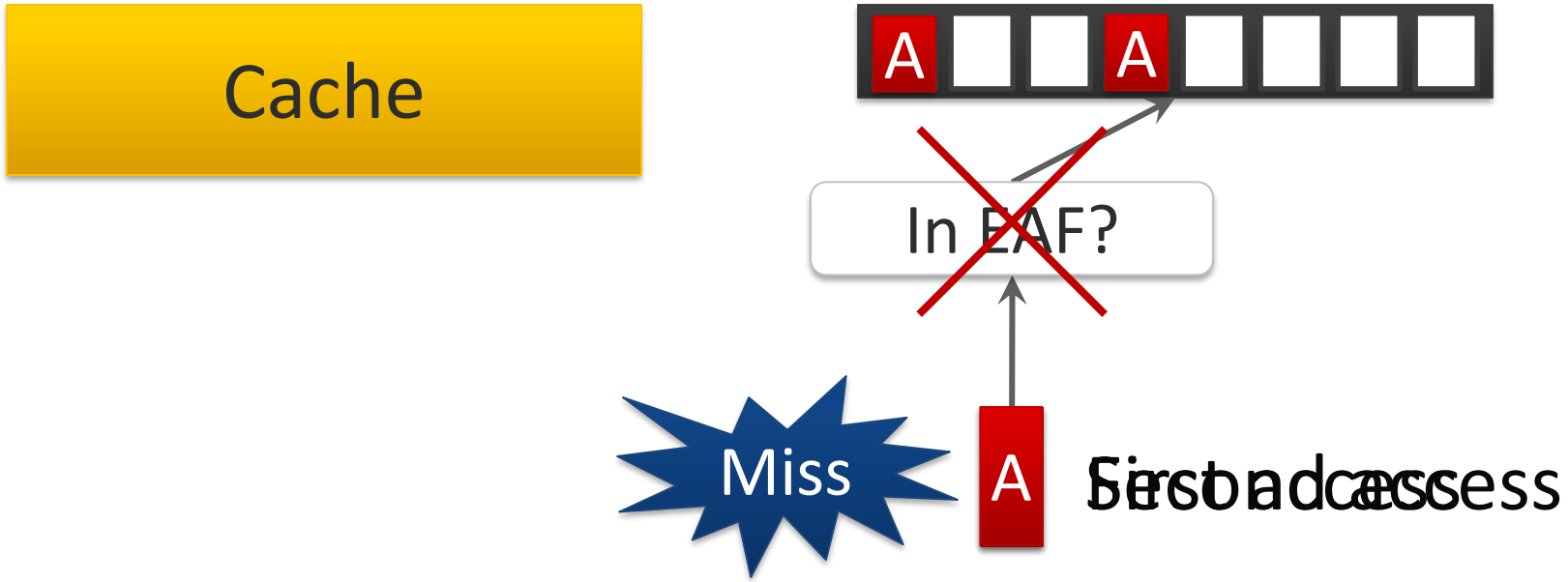
- Background and Motivation
- Evicted-Address Filter
  - Reuse Prediction
  - Thrash Resistance
- Final Design
- Advantages and Disadvantages
- Evaluation
- Conclusion

EAF: Advantages



1. Simple to implement
2. Easy to design and verify
3. Works with other techniques (replacement policy)

EAF: Disadvantage



Problem: For an **LRU-friendly application,** EAF incurs one **additional** miss for most blocks

Dueling-EAF: set dueling between EAF and LRU

Outline

- Background and Motivation
- Evicted-Address Filter
  - Reuse Prediction
  - Thrash Resistance
- Final Design
- Advantages and Disadvantages
- Evaluation
- Conclusion

Methodology

- Simulated System
  - In-order cores, single issue, 4 GHz
  - 32 KB L1 cache, 256 KB L2 cache (private)
  - Shared L3 cache (1MB to 16MB)
  - Memory: 150 cycle row hit, 400 cycle row conflict
- Benchmarks
  - SPEC 2000, SPEC 2006, TPC-C, 3 TPC-H, Apache
- Multi-programmed workloads
  - Varying memory intensity and cache sensitivity
- Metrics
  - 4 different metrics for performance and fairness
  - Present weighted speedup

Comparison with Prior Works

Addressing Cache Pollution

Run-time Bypassing (RTB) – Johnson+ ISCA'97

- Memory region based reuse prediction

Single-usage Block Prediction (SU) – Piquet+ ACSAC'07
Signature-based Hit Prediction (SHIP) – Wu+ MICRO'11

- Program counter based reuse prediction

Miss Classification Table (MCT) – Collins+ MICRO'99

- One most recently evicted block

- No control on number of blocks inserted with high priority $\Longrightarrow$ Thrashing

Comparison with Prior Works

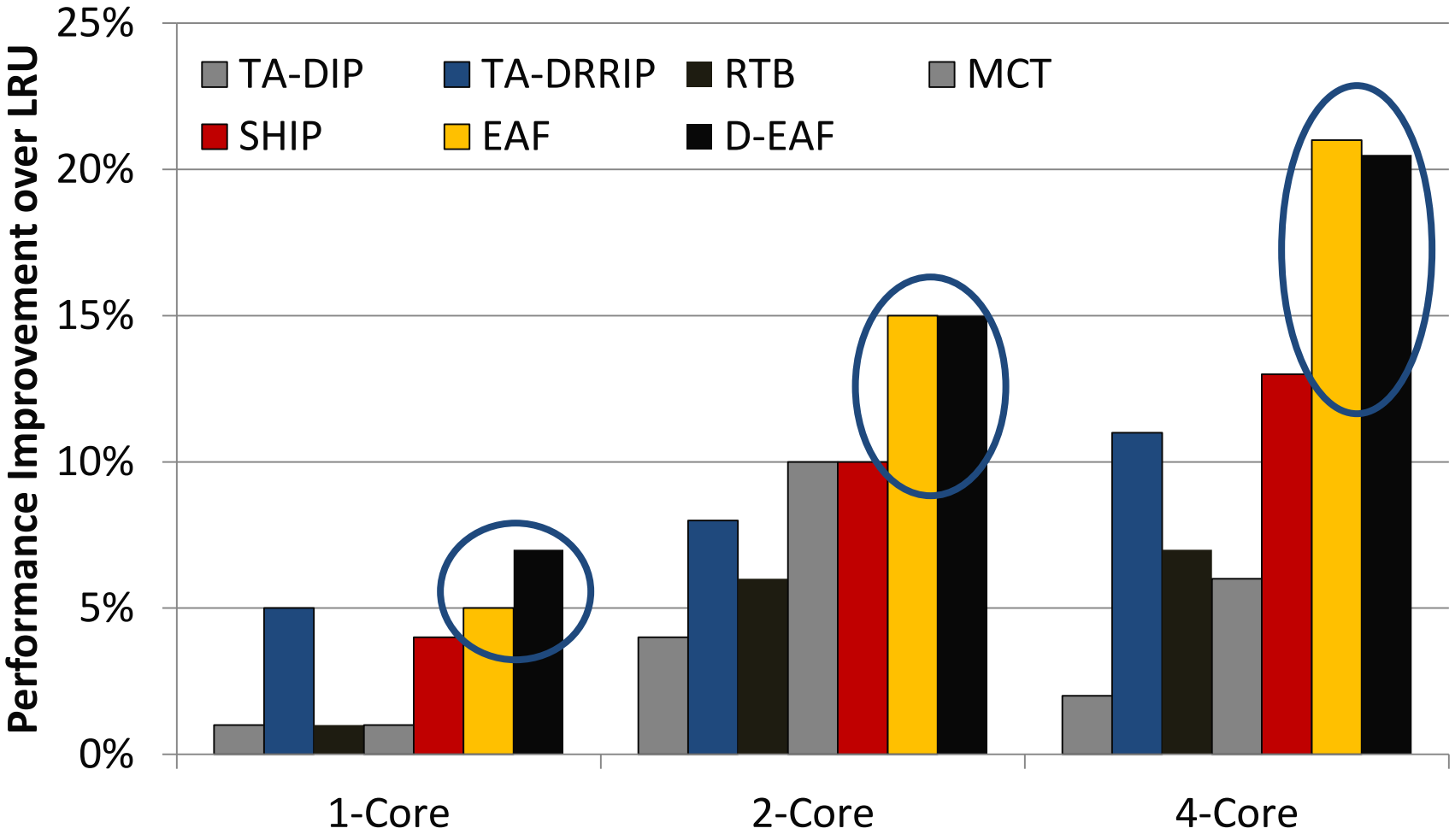
Addressing Cache Thrashing

TA-DIP – Qureshi+ ISCA'07, Jaleel+ PACT'08
TA-DRRIP – Jaleel+ ISCA'10

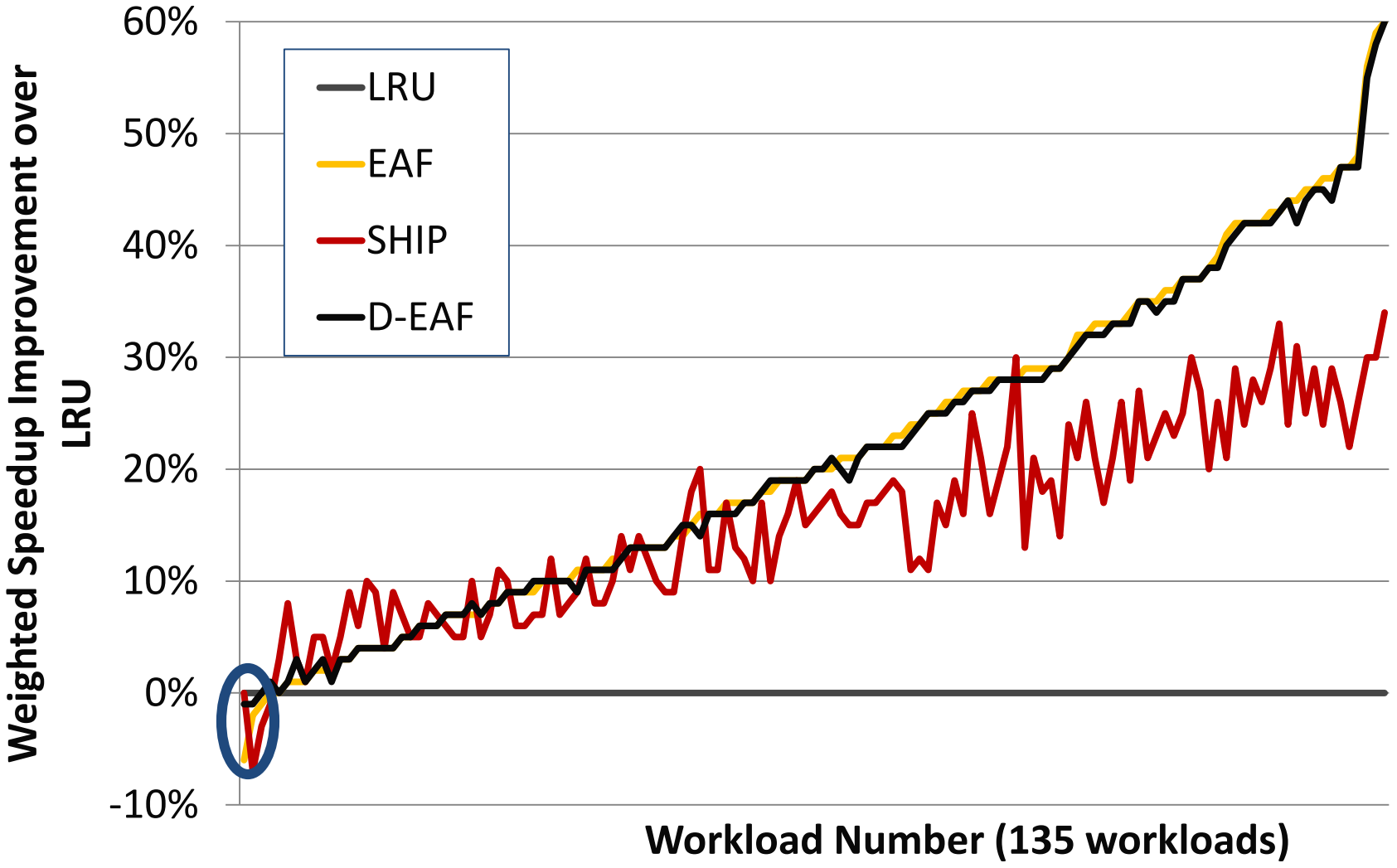
- Use set dueling to determine thrashing applications

- No mechanism to filter low-reuse blocks $\Longrightarrow$ Pollution

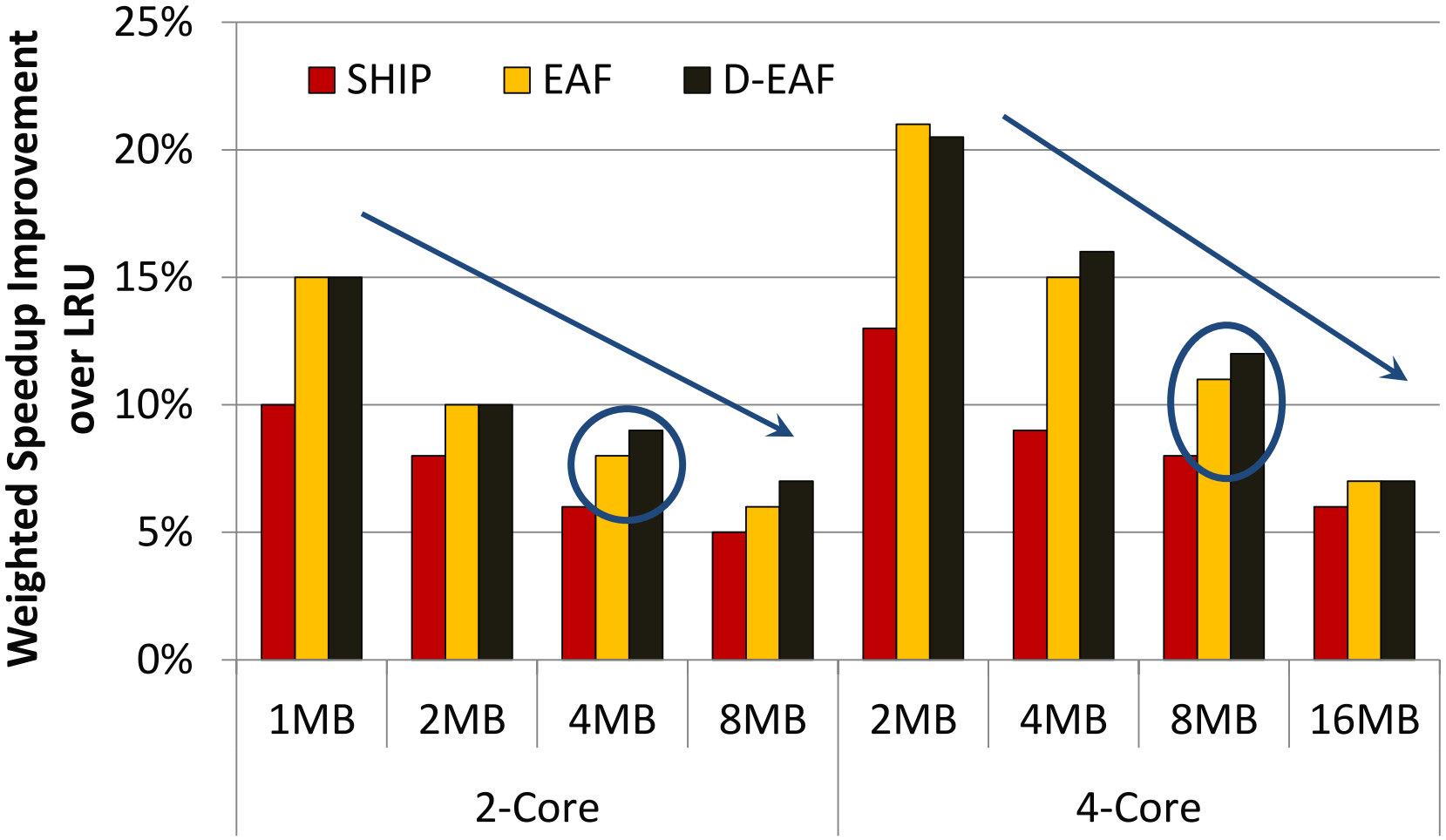
Results – Summary



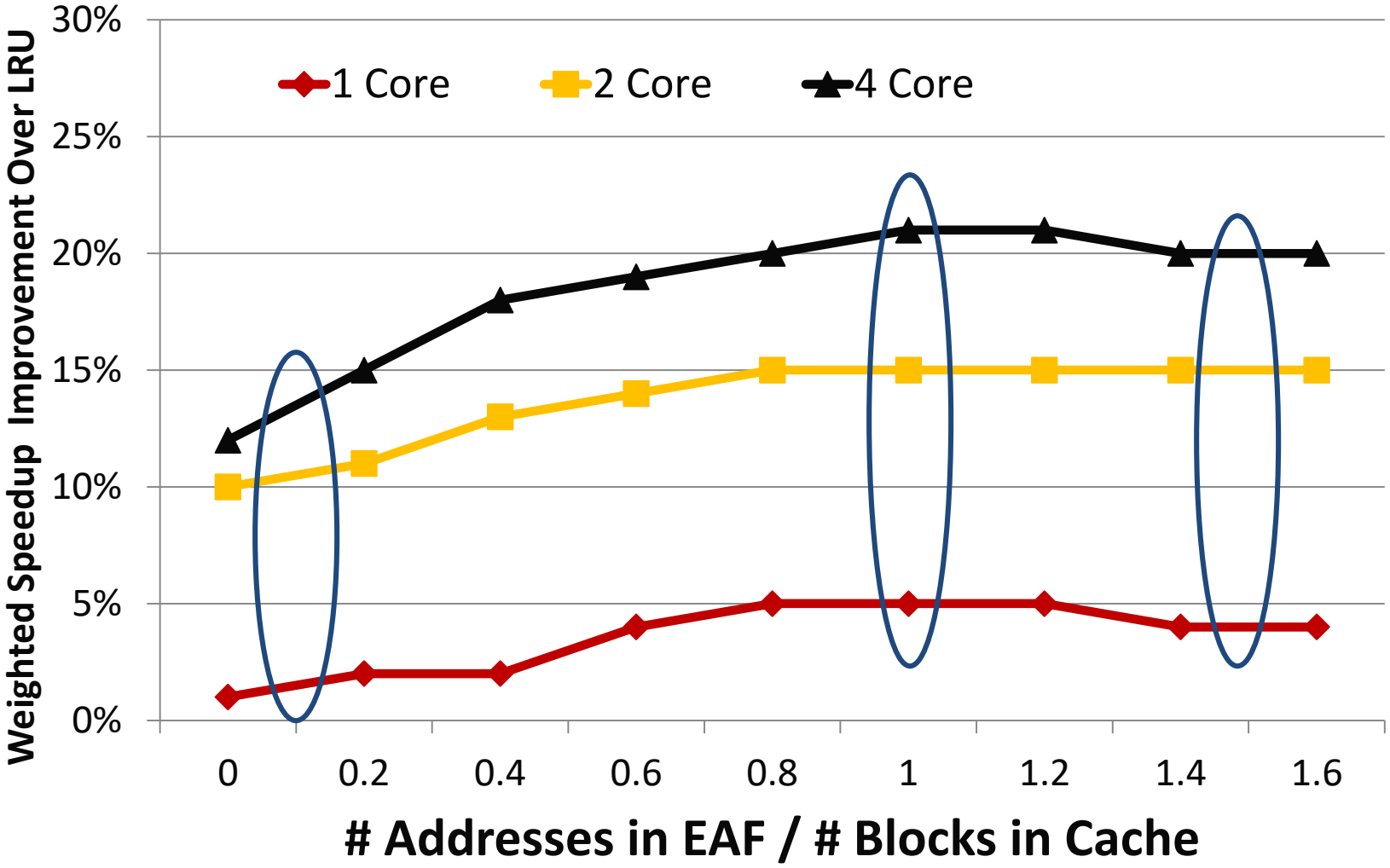
4-Core: Performance



Effect of Cache Size



Effect of EAF Size



Other Results in Paper

- EAF orthogonal to replacement policies
  - LRU, RRIP – Jaleel+ ISCA'10
- Performance improvement of EAF increases with increasing memory latency
- EAF performs well on four different metrics
  - Performance and fairness
- Alternative EAF-based designs perform comparably
  - Segmented EAF
  - Decoupled-clear EAF

Conclusion

- Cache utilization is critical for system performance
  - Pollution and thrashing degrade cache performance
  - Prior works don't address both problems concurrently

- EAF-Cache
  - Keep track of recently evicted block addresses in EAF
  - Insert low reuse with low priority to mitigate pollution
  - Clear EAF periodically and use BIP to mitigate thrashing
  - Low complexity implementation using Bloom filter

- EAF-Cache outperforms five prior approaches that address pollution or thrashing

Base-Delta-Immediate Cache Compression

Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Philip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry,
"Base-Delta-Immediate Compression: Practical Data Compression for On-Chip Caches"
*Proceedings of the 21st ACM International Conference on Parallel Architectures and Compilation Techniques (**PACT**)*, Minneapolis, MN, September 2012. Slides (pptx)

Executive Summary

- Off-chip memory latency is high
  - Large caches can help, **but** at significant cost
- Compressing data in cache enables larger cache at low cost
- **Problem**: Decompression is on the execution critical path
- **Goal**: Design a new compression scheme that has

1. low decompression latency, 2. low cost, 3. high compression ratio

- **Observation:** Many cache lines have low dynamic range data
- **Key Idea**: Encode cachelines as a base **+** multiple differences
- **Solution**: Base-Delta-Immediate compression with low decompression latency and high compression ratio
  - Outperforms three state-of-the-art compression mechanisms

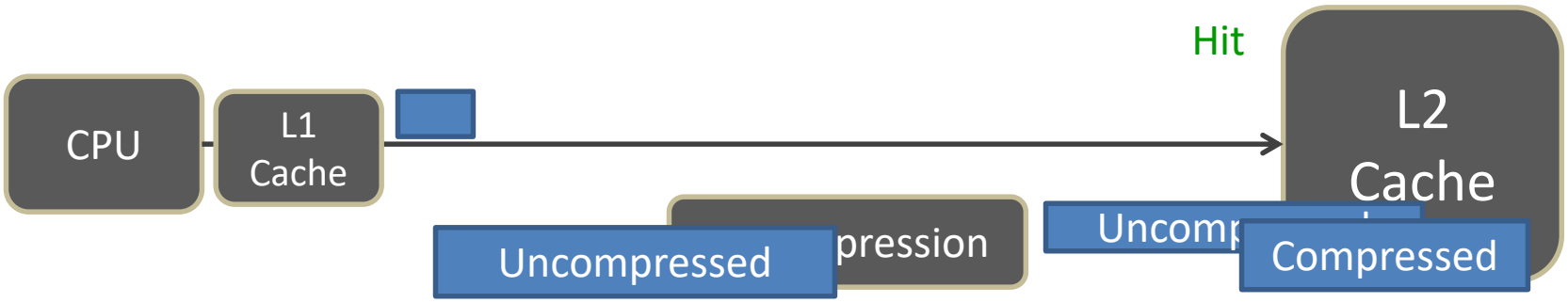
Motivation for Cache Compression

Significant redundancy in data:

How can we exploit this redundancy?

- **Cache compression** helps
- Provides effect of a larger cache without making it physically larger

Background on Cache Compression



- Key requirements:
  - **Fast** (low decompression latency)
  - **Simple** (avoid complex hardware changes)
  - **Effective** (good compression ratio)

Shortcomings of Prior Work

Compression Mechanisms	Decompression Latency	Complexity	Compression Ratio
Zero	✓	✓	✗

Shortcomings of Prior Work

Compression Mechanisms	Decompression Latency	Complexity	Compression Ratio
Zero	✓	✓	✗
Frequent Value	✗	✗	✓

Shortcomings of Prior Work

Compression Mechanisms	Decompression Latency	Complexity	Compression Ratio
Zero	✓	✓	✗
Frequent Value	✗	✗	✓
Frequent Pattern	✗	✗/✓	✓

Shortcomings of Prior Work

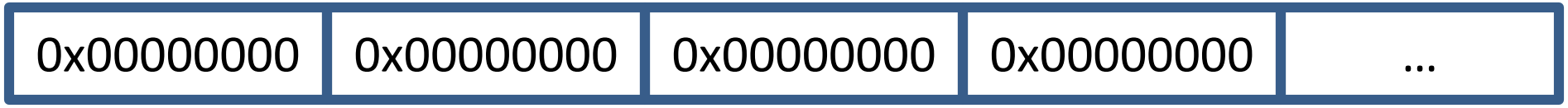
Compression Mechanisms	Decompression Latency	Complexity	Compression Ratio
Zero	✓	✓	✗
Frequent Value	✗	✗	✓
Frequent Pattern	✗	✗ / ✓	✓
Our proposal: BΔI	✓	✓	✓

Outline

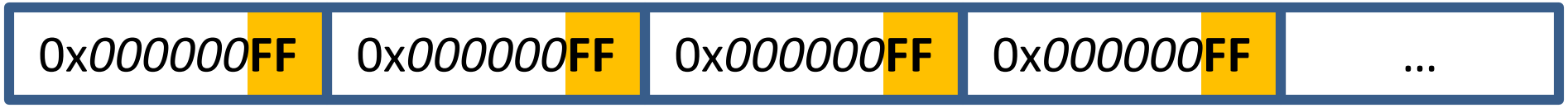
- Motivation & Background
- Key Idea & Our Mechanism
- Evaluation
- Conclusion

Key Data Patterns in Real Applications

Zero Values: initialization, sparse matrices, NULL pointers



Repeated Values: common initial values, adjacent pixels



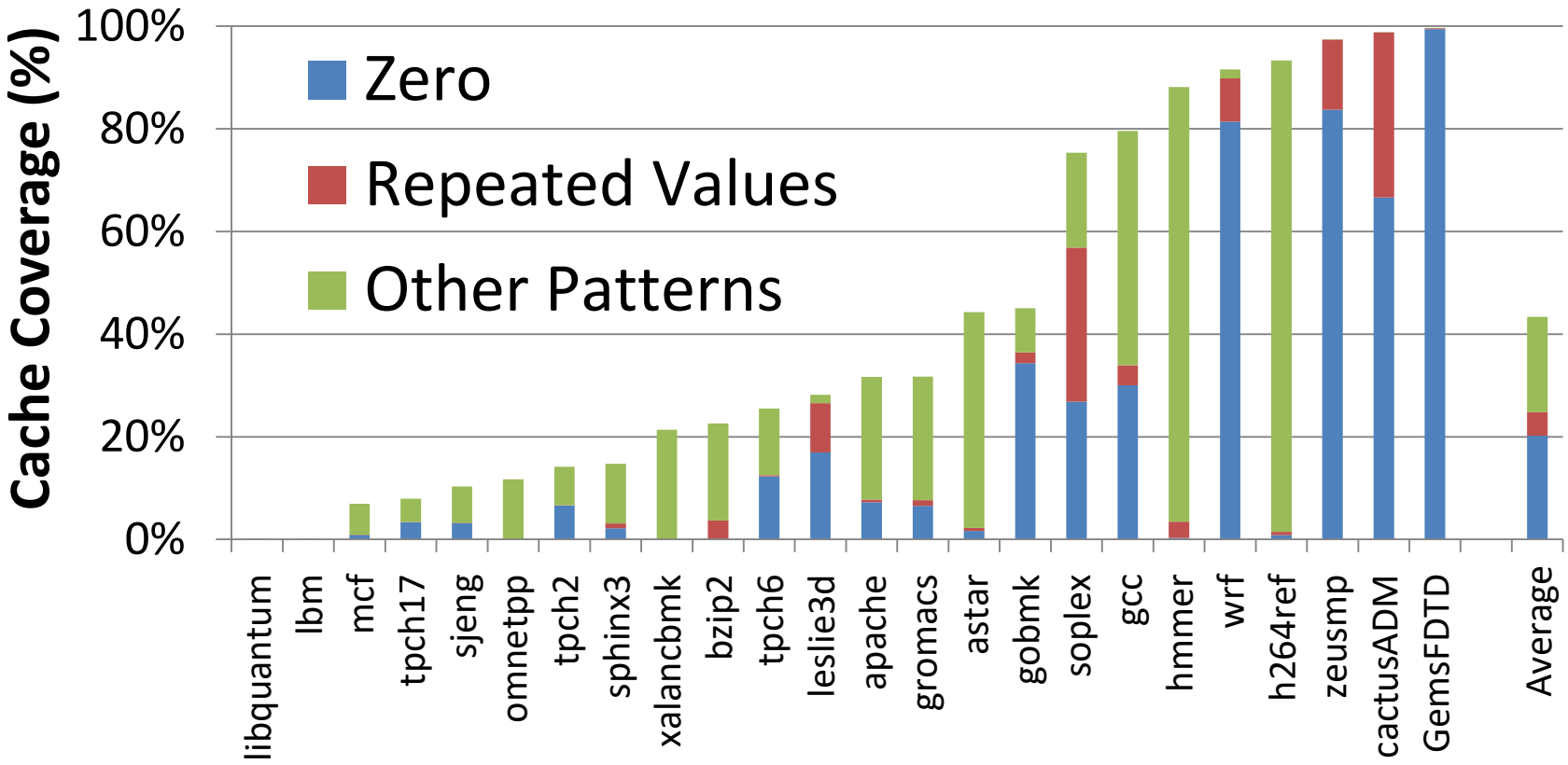
Narrow Values: small values stored in a big data type

Other Patterns: pointers to the same memory region

How Common Are These Patterns?

SPEC2006, databases, web workloads, 2MB L2 cache

"Other Patterns" include Narrow Values



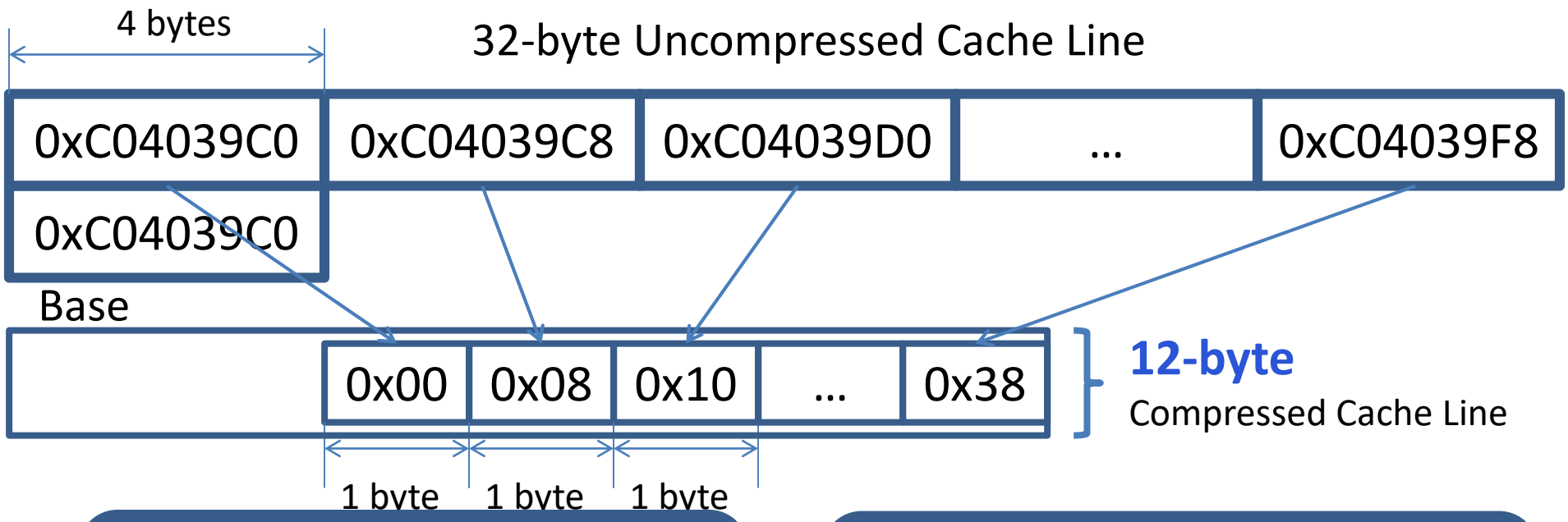
43% of the cache lines belong to key patterns

Key Data Patterns in Real Applications

Low Dynamic Range:

Differences between values are significantly smaller than the values themselves

Key Idea: Base+Delta (B+Δ) Encoding



✓ **Fast Decompression:** vector addition

✓ **Simple Hardware:** arithmetic and comparison

✓ **Effective:** good compression ratio

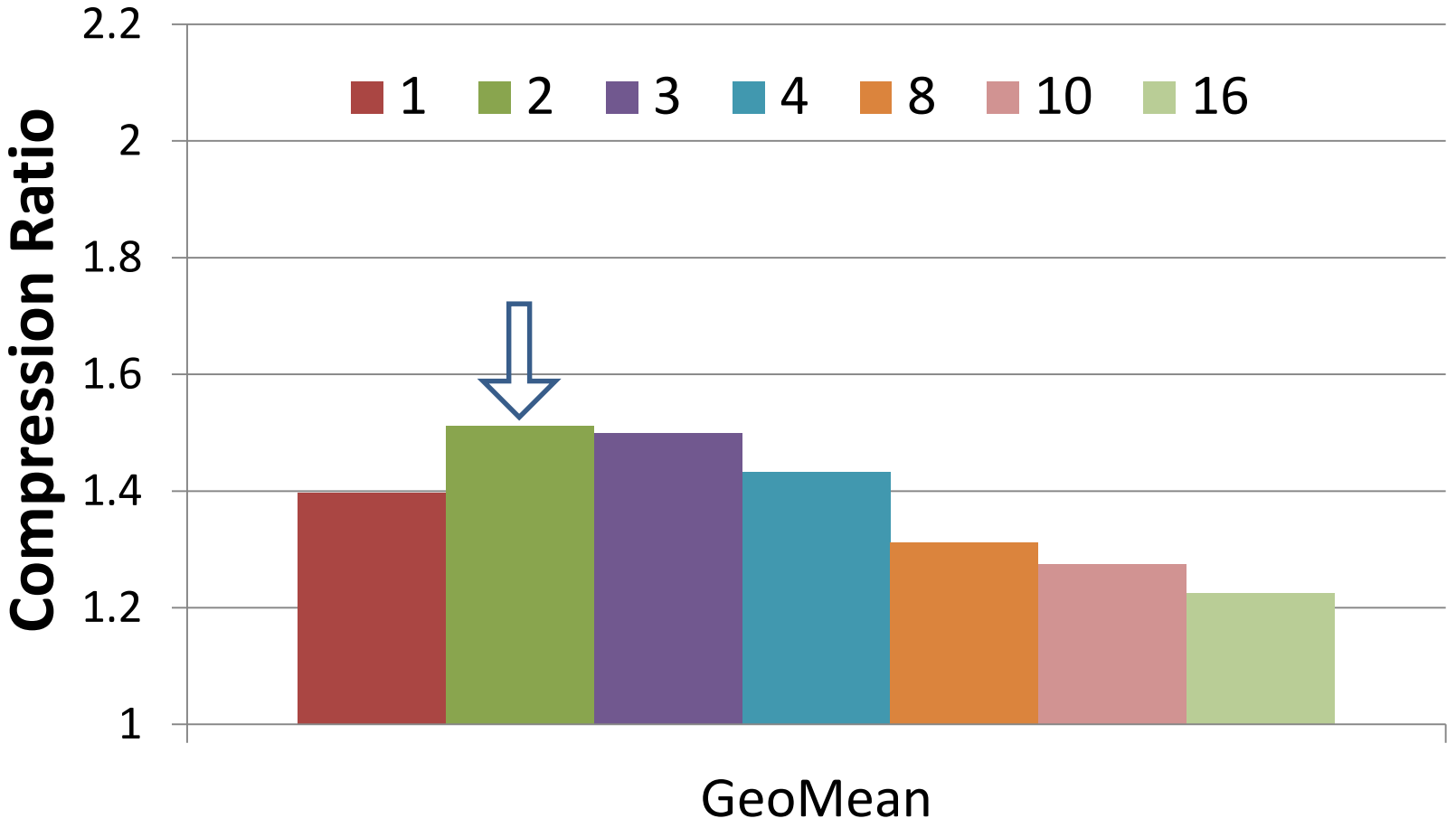
Can We Do Better?

- Uncompressible cache line (with a single base):

0x00000000	**0x09A4**0178	**0x0000000**B	**0x09A4**A838	…

- **Key idea:**
  Use more bases, e.g., two instead of one
- Pro:
  - More cache lines can be compressed
- Cons:
  - Unclear how to find these bases efficiently
  - Higher overhead (due to additional bases)

B+Δ with Multiple Arbitrary Bases



✓ **2 bases –** the best option based on evaluations

How to Find Two Bases Efficiently?

1. **First base - first element** in the cache line

   ✓ **Base+Delta part**

2. **Second base -** implicit base of **0**

   ✓ **Immediate part**

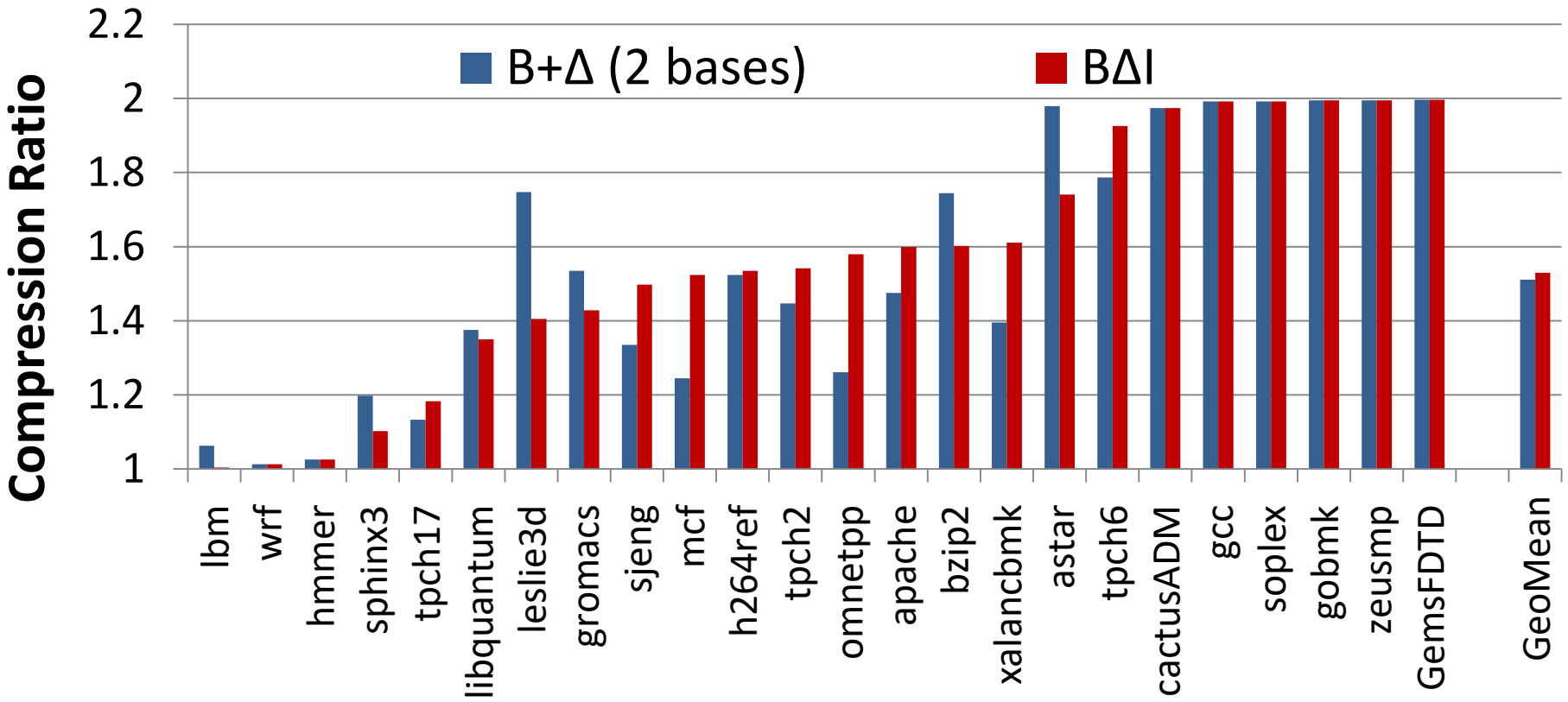
Advantages over 2 arbitrary bases:

– Better compression ratio

– Simpler compression logic

Base-Delta-Immediate (BΔI) Compression

B+Δ (with two arbitrary bases) vs. BΔI

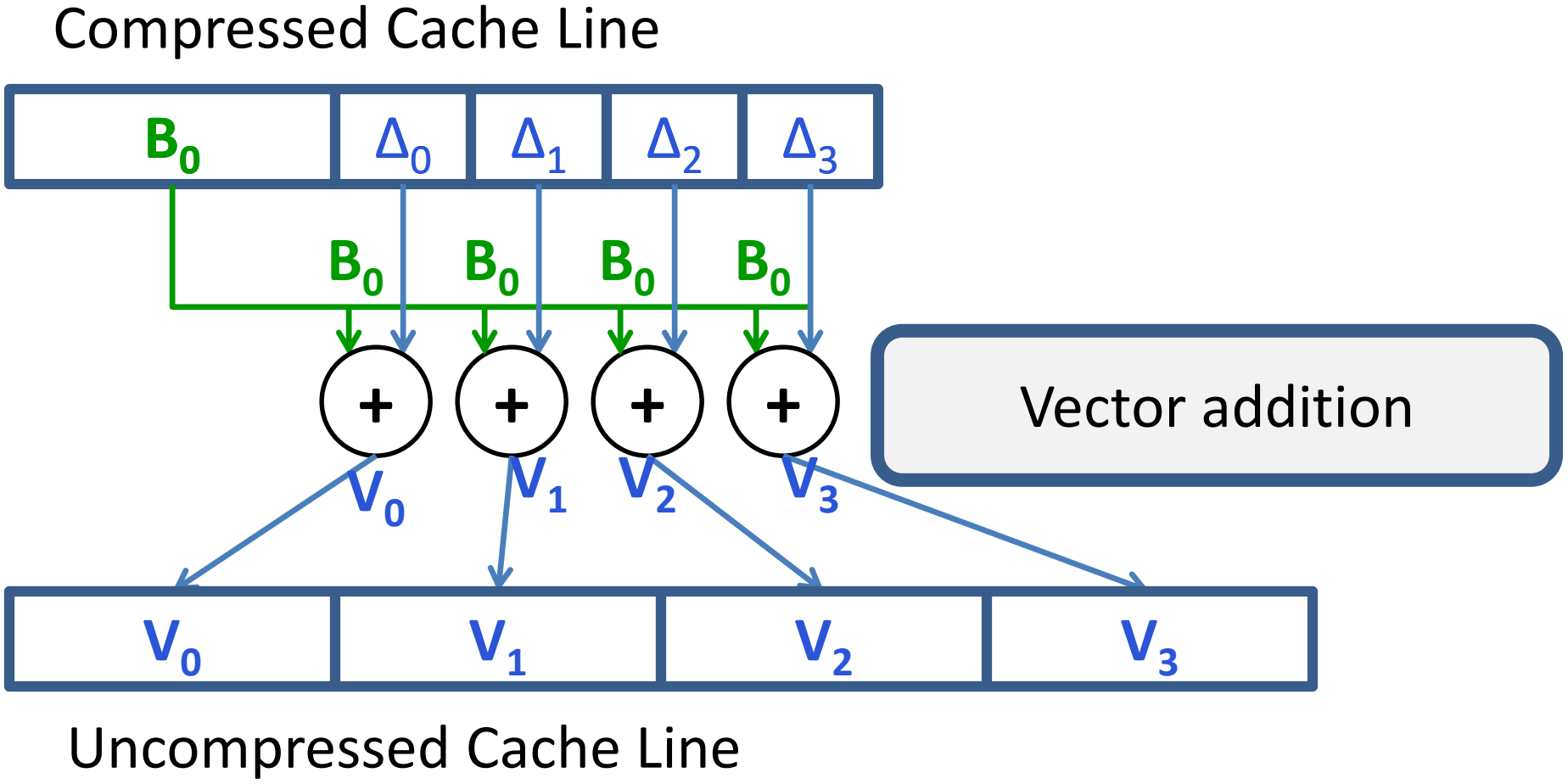


Average compression ratio is close, but **BΔI** is **simpler**

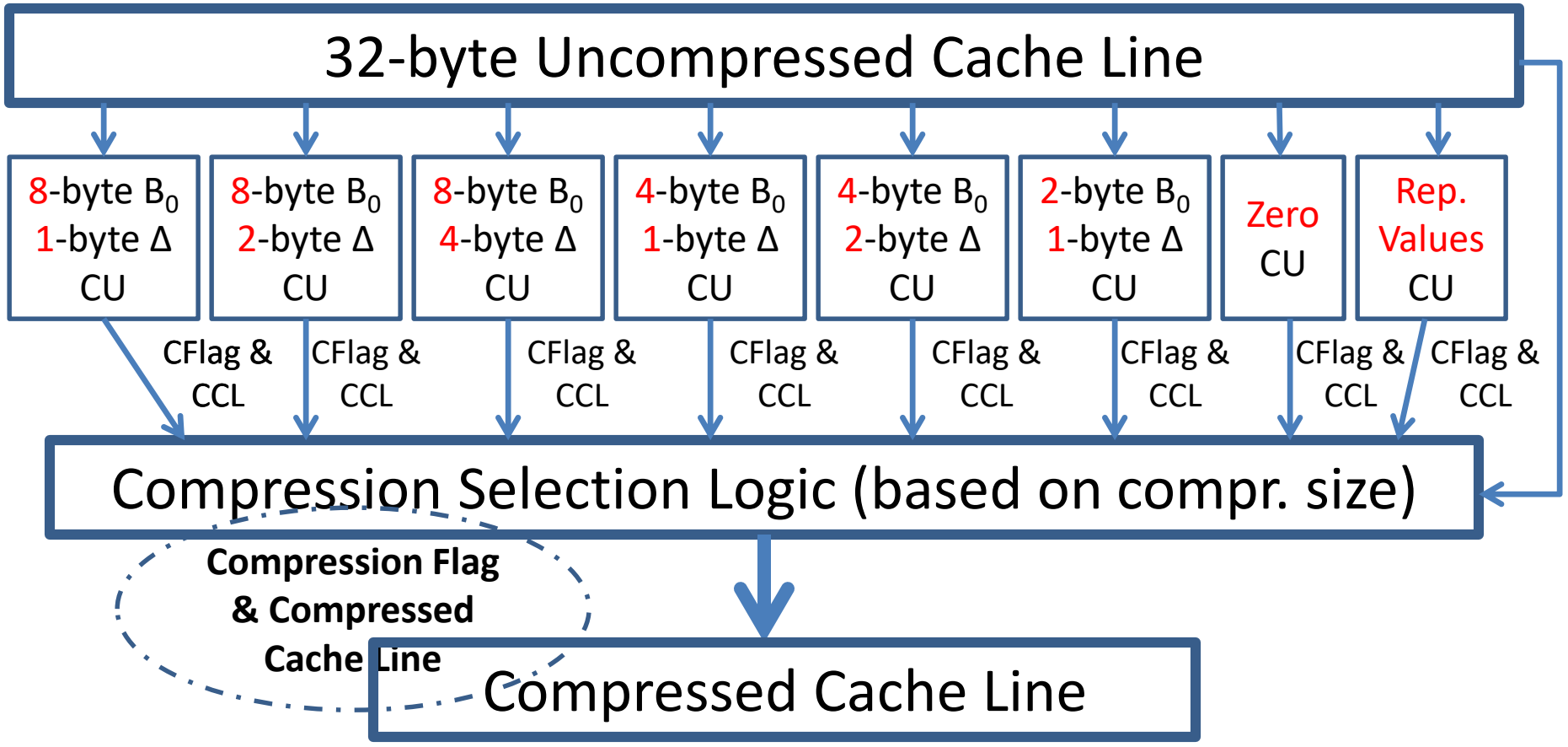
BΔI Implementation

- **Decompressor Design**
  - Low latency
- **Compressor Design**
  - Low cost and complexity
- **BΔI Cache Organization**
  - Modest complexity

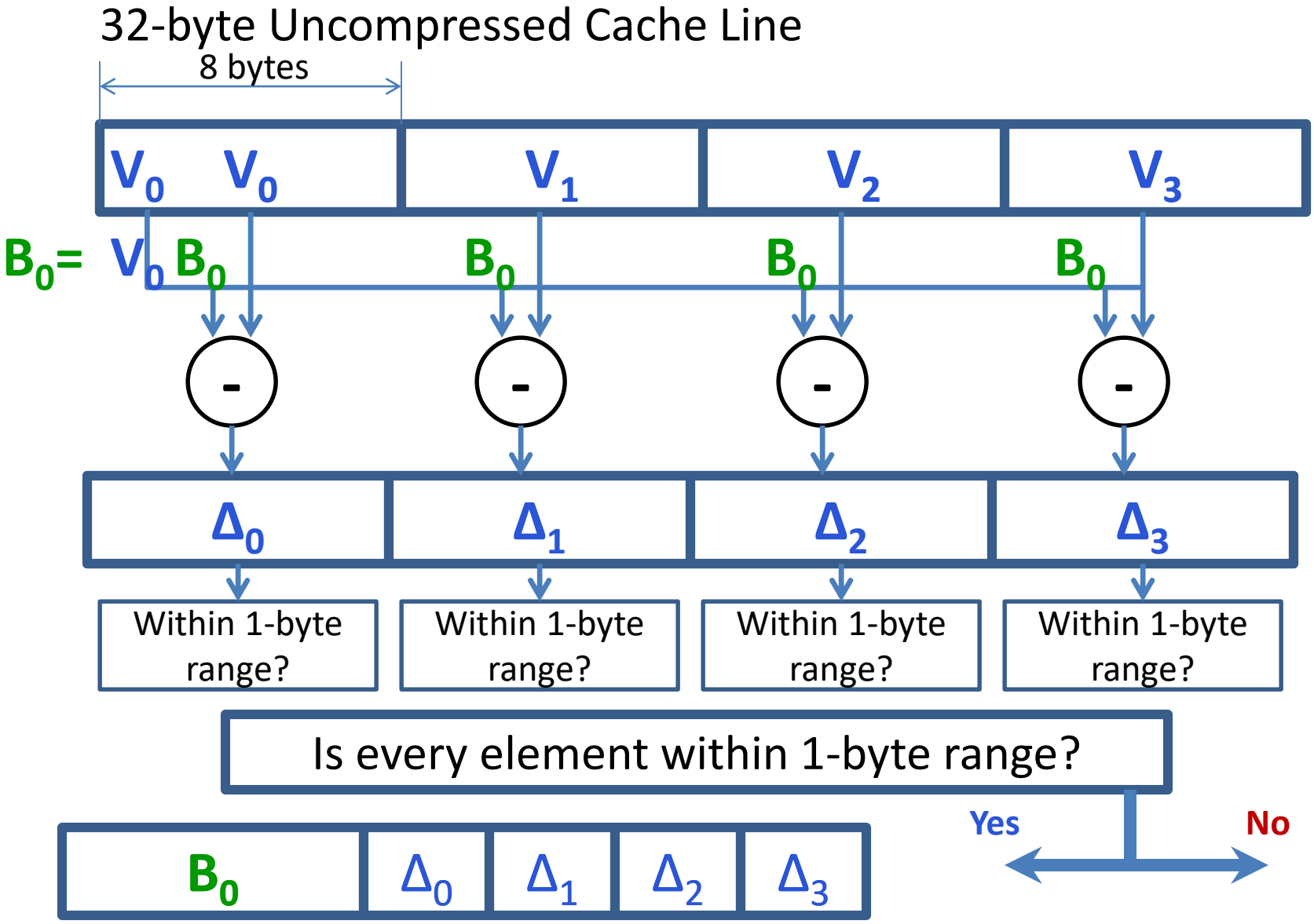
BΔI Decompressor Design



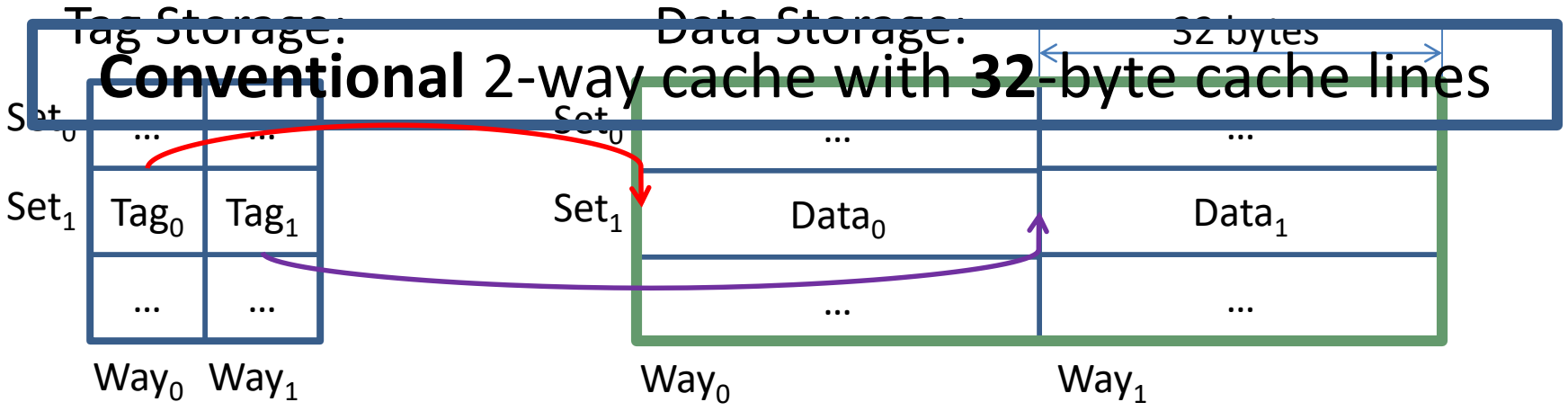
BΔI Compressor Design



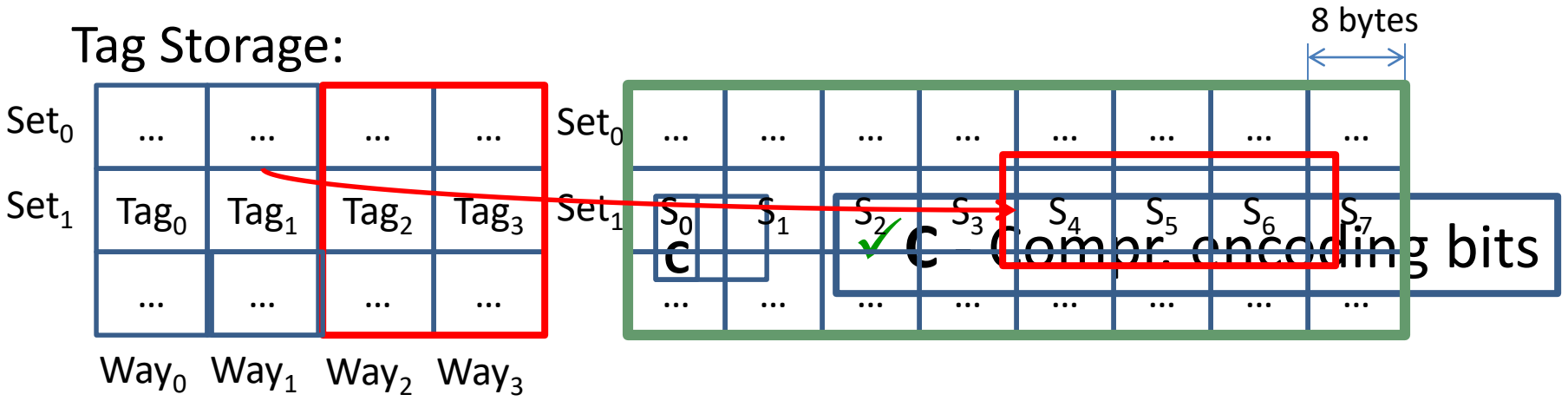
BΔI Compression Unit: 8-byte B_0 1-byte Δ



BΔI Cache Organization



BΔI: 4-way cache with **8**-byte segmented data



✓ Twice as many tags

✓ Tags map to multiple adjacent segments

2.3% overhead for 2 MB cache

Qualitative Comparison with Prior Work

- **Zero-based designs**
  - ZCA *[Dusser+, ICS'09]*: zero-content augmented cache
  - ZVC *[Islam+, PACT'09]*: zero-value cancelling
  - Limited applicability (only zero values)
- **FVC** *[Yang+, MICRO'00]*: frequent value compression
  - High decompression latency and complexity
- **Pattern-based compression designs**
  - FPC *[Alameldeen+, ISCA'04]*: frequent pattern compression
    - High decompression latency (5 cycles) and complexity
  - C-pack *[Chen+, T-VLSI Systems'10]*: practical implementation of FPC-like algorithm
    - High decompression latency (8 cycles)

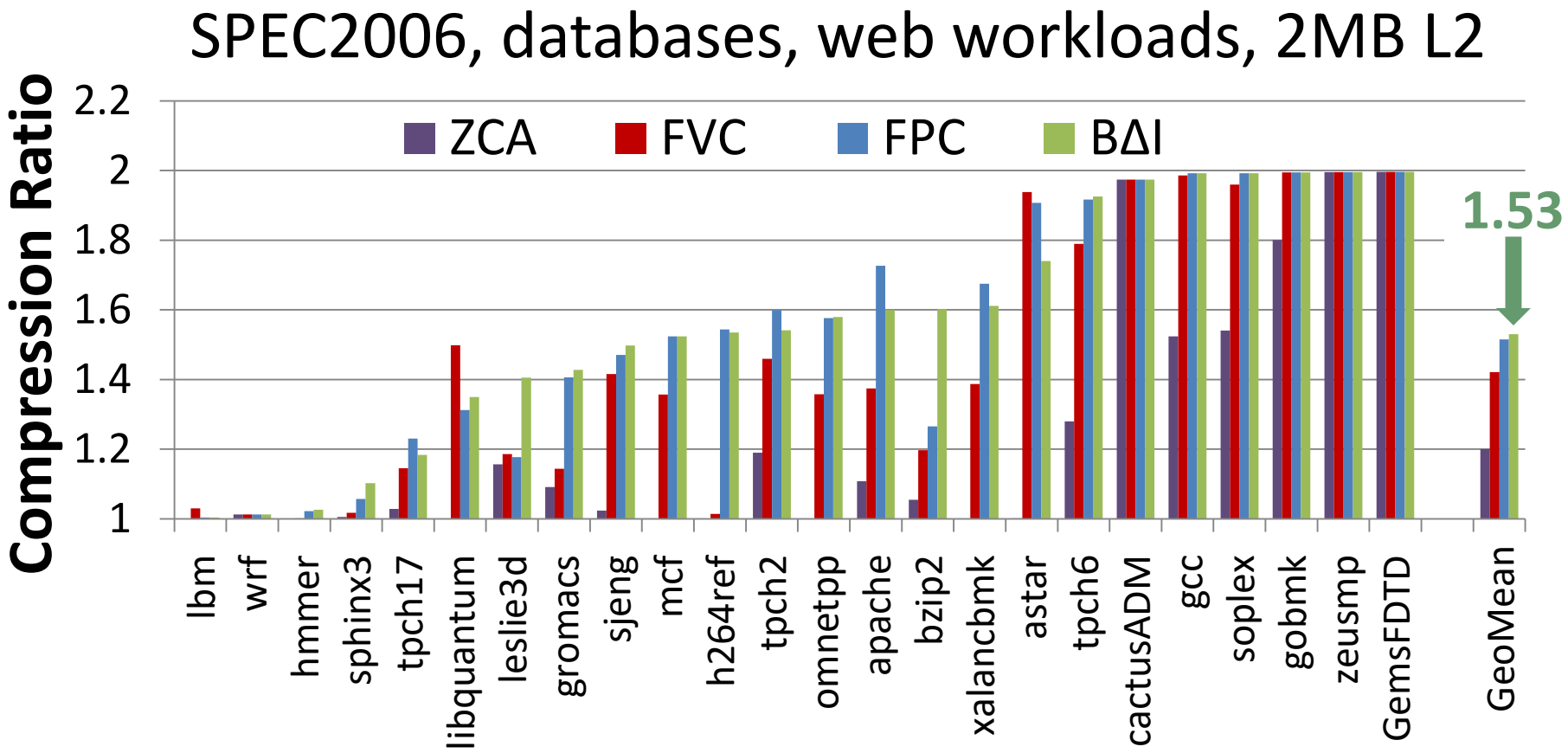
Outline

- Motivation & Background
- Key Idea & Our Mechanism
- Evaluation
- Conclusion

Methodology

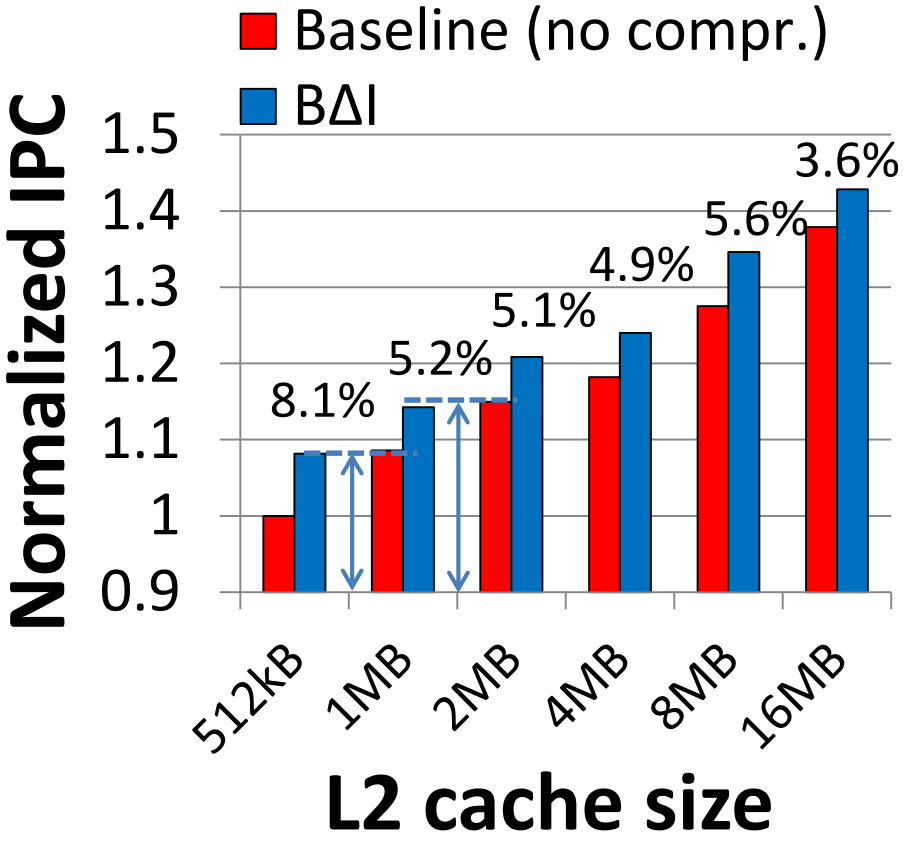
- **Simulator**
  - x86 event-driven simulator based on Simics *[Magnusson+, Computer'02]*
- **Workloads**
  - SPEC2006 benchmarks, TPC, Apache web server
  - 1 – 4 core simulations for 1 billion representative instructions
- **System Parameters**
  - L1/L2/L3 cache latencies from CACTI *[Thoziyoor+, ISCA'08]*
  - 4GHz, x86 in-order core, **512kB - 16MB** L2, simple memory model (**300**-cycle latency for row-misses)

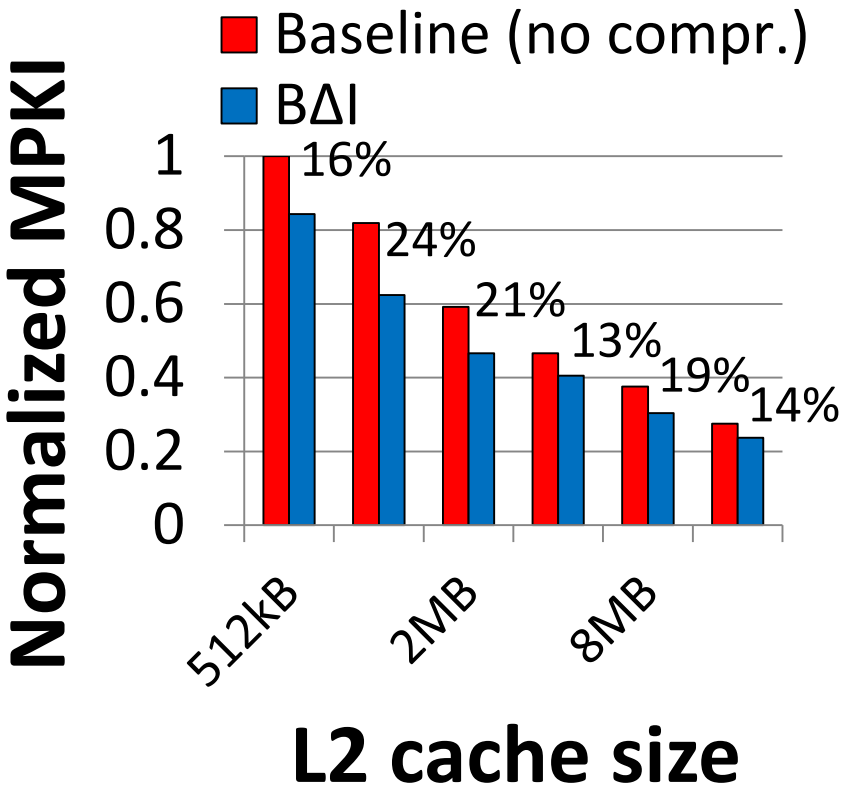
Compression Ratio: BΔI vs. Prior Work



BΔI achieves the highest compression ratio

Single-Core: IPC and MPKI





BΔI achieves the performance of a 2X-size cache

Performance improves due to the decrease in MPKI

Multi-Core Workloads

- Application classification based on

  Compressibility: effective cache size increase

  (Low Compr. (***LC***) < 1.40, High Compr. (***HC***) >= 1.40)

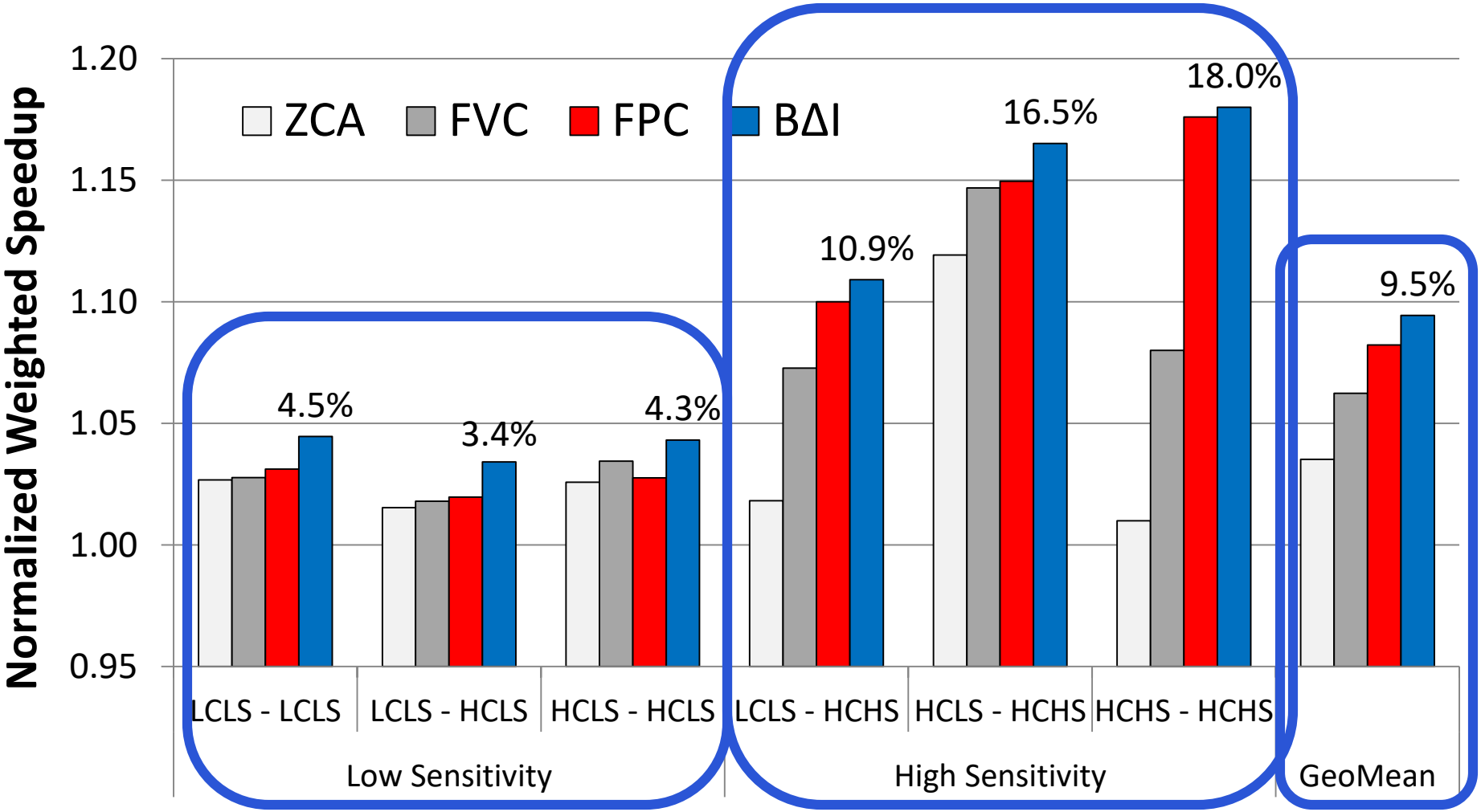
  Sensitivity: performance gain with more cache

  (Low Sens. (***LS***) < 1.10, High Sens. (***HS***) >= 1.10; 512kB -> 2MB)

- Three classes of applications:
  - LCLS, HCLS, HCHS, **no LCHS** applications

- For 2-core - **random** mixes of each possible class pairs (20 each, 120 total workloads)

Multi-Core: Weighted Speedup



If at least one application is sensitive, then the performance improves

BΔI performance improvement is the highest (9.5%)

Other Results in Paper

- IPC comparison against **upper** bounds
  - BΔI almost achieves performance of the 2X-size cache
- Sensitivity study of having **more** than 2X tags
  - Up to 1.98 average compression ratio
- Effect on **bandwidth** consumption
  - 2.31X decrease on average
- Detailed quantitative comparison with prior work
- **Cost analysis** of the proposed changes
  - 2.3% L2 cache area increase

Conclusion

- A new **Base-Delta-Immediate** compression mechanism
- Key insight: many cache lines can be efficiently represented using **base + delta encoding**
- Key properties:
  - **Low** latency decompression
  - **Simple** hardware implementation
  - **High compression ratio** with high coverage
- **Improves** *cache hit ratio* and *performance* of both single-core and multi-core workloads
  - Outperforms state-of-the-art cache compression techniques: FVC and FPC