Computer Architecture

Lecture 3: Cache Management and Memory Parallelism

Prof. Onur Mutlu
ETH Zurich

Fall 2017

27 September 2017

Summary of Last Lecture

- ISA vs. Microarchitecture
- Dataflow
- Memory Hierarchy
- Cache Design

Agenda for Today

- Issues in Caching
- More Effective Cache Design
- Enabling Multiple Concurrent Memory Accesses
  - Memory Level Parallelism
- Multi-Core Issues in Caching

Takeaway From Lectures 1 & 2

Breaking the abstraction layers (between components and transformation hierarchy levels)

and knowing what is underneath

enables you to **understand** and **solve** problems

Computer Architecture

Guidelines on Paper Reviews

Prof. Onur Mutlu

ETH Zürich

Fall 2017

How to Do the Paper/Talk Reviews

- 1: Summary
  - What is the problem the paper is trying to solve?
  - What are the key ideas of the paper? Key insights?
  - What are the key mechanisms? What is the implementation?
  - What are the key results? Key conclusions?
- 2: Strengths (most important ones)
  - Does the paper solve the problem well? Is it well written? …
- 3: Weaknesses (most important ones)
  - This is where you should think critically. Every paper/idea has a weakness. This does not mean the paper is necessarily bad. It means there is room for improvement and future research can accomplish this.
- 4: Can you do (much) better? Present your thoughts/ideas.
- 5: Takeaways: What you learned/enjoyed/disliked? Why?
- 6: Any other comments you would like to make.

- Review should be short and concise (~one page)

Advice on Paper/Talk Reviews

- When doing the reviews, be very critical

- Always think about better ways of solving the problem or related problems
  - Question the problem as well

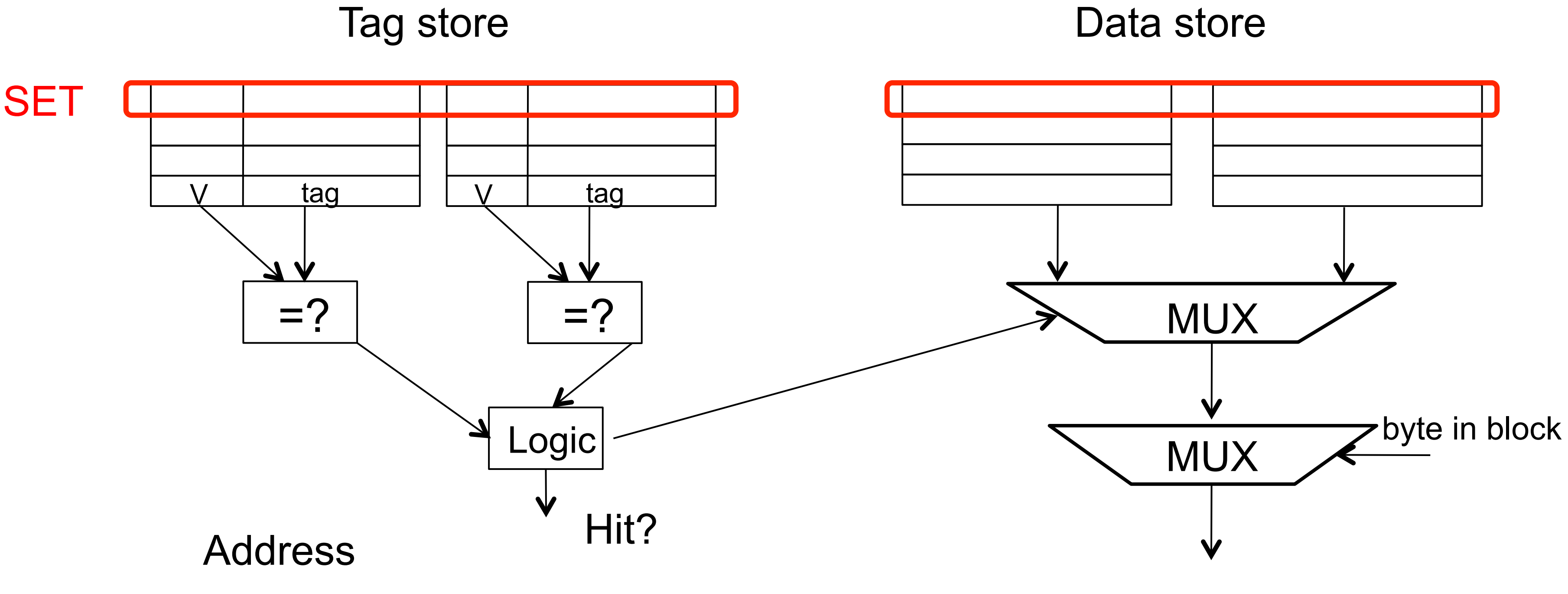
- This is how things progress in science and engineering (or anywhere), and how you can make big leaps
  - By critical analysis

- Sample reviews provided online

Back to Caching

Review: Set Associativity

- Multiple blocks can be stored in the same cache set (i.e., index)
- Example: 2-way cache:

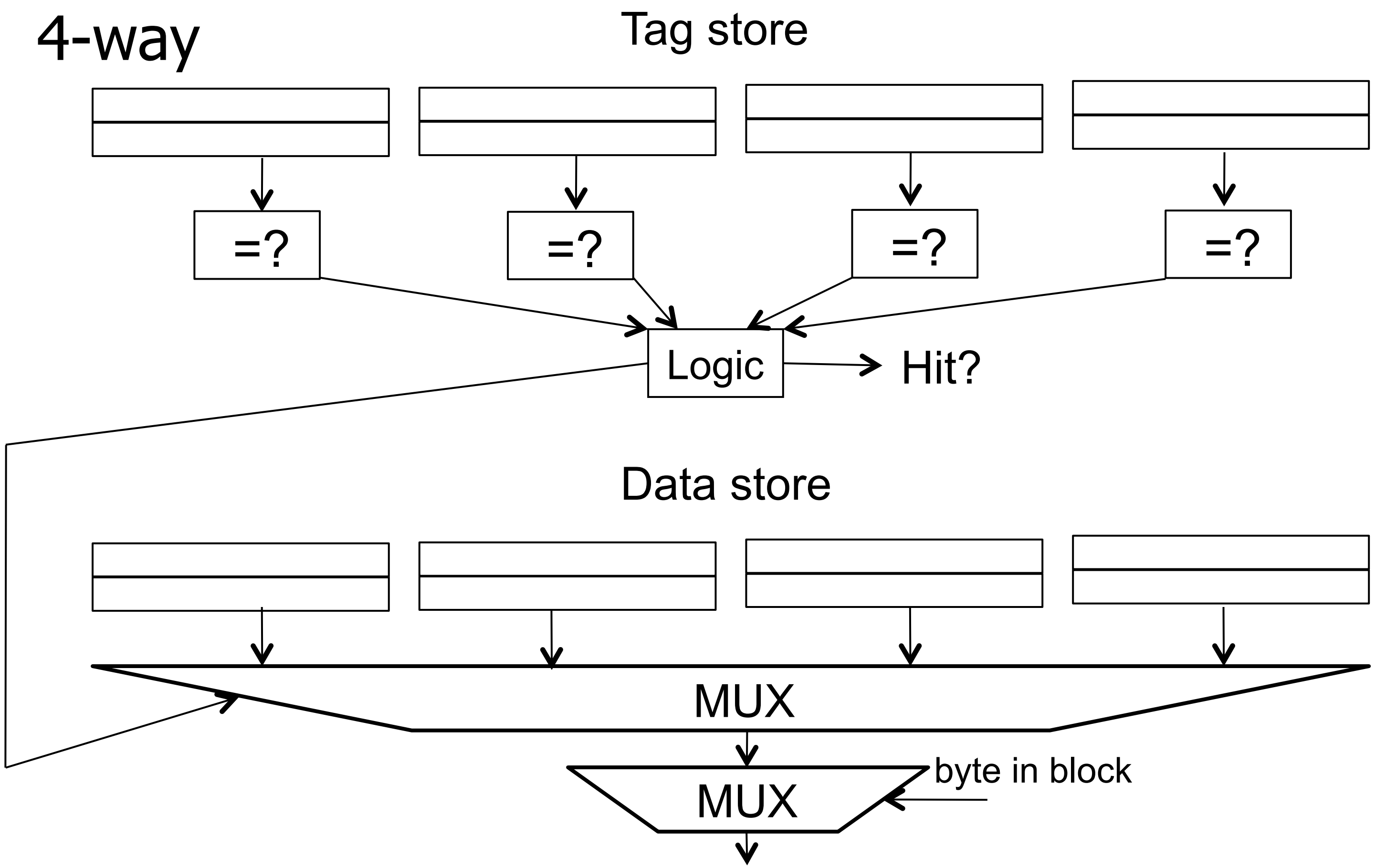


tag	index	byte in block
3b	2 bits	3 bits

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

Review: Higher Associativity

- 4-way

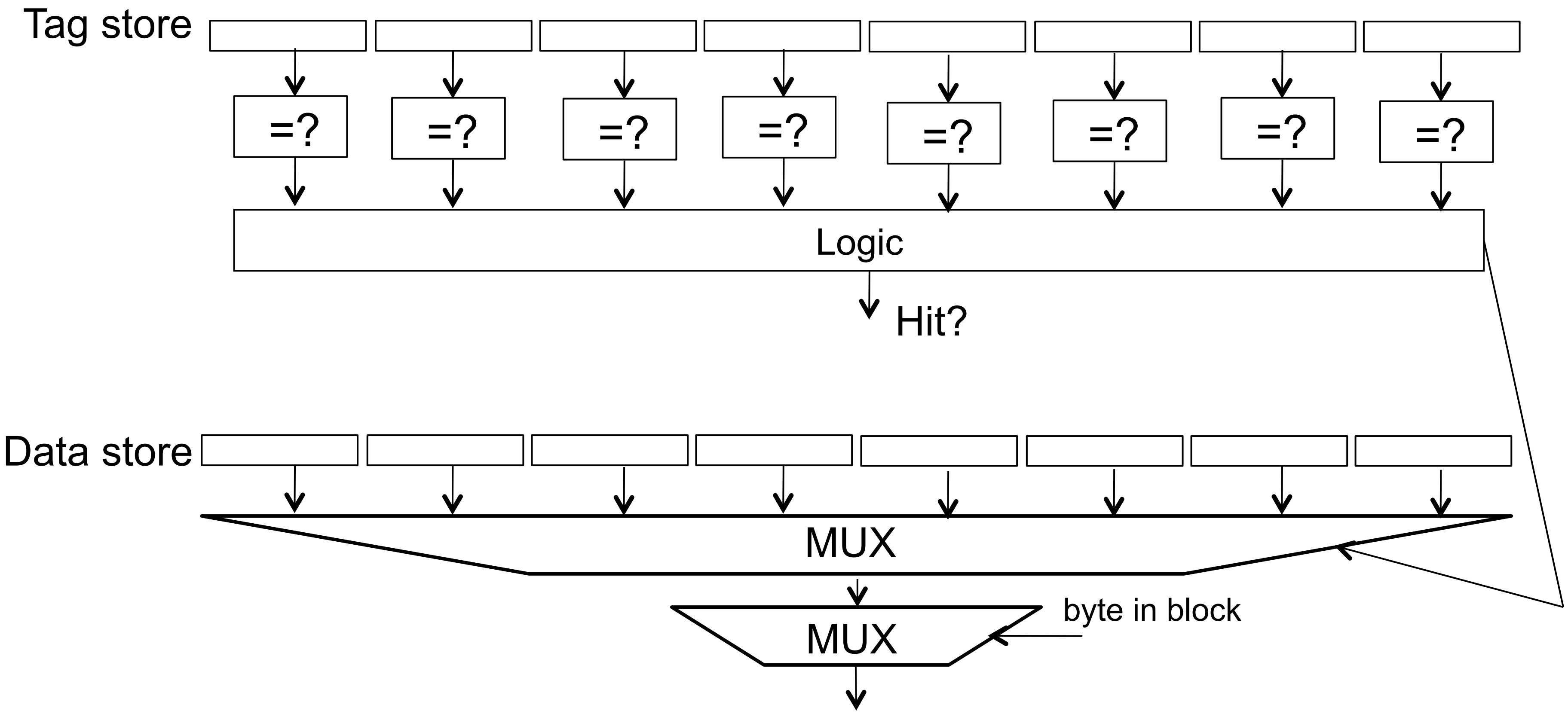


+ Likelihood of conflict misses even lower

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

Review: Full Associativity

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



Review: Associativity (and Tradeoffs)

- Degree of associativity: How many blocks can be present at the same index (or set)?

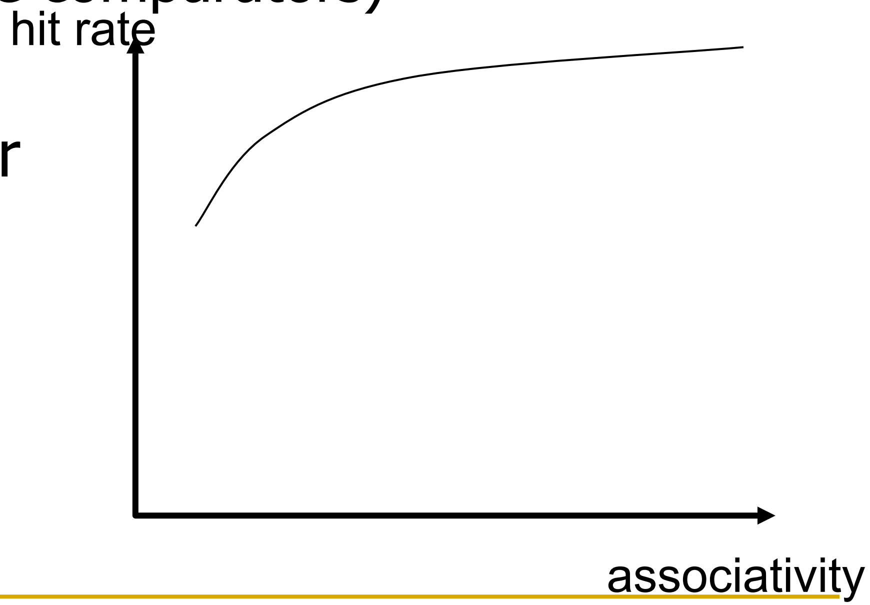
- Higher associativity

  ++ 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

Hierarchical LRU (not MRU)

- Divide a set into multiple groups
- Keep track of *only* the MRU group
- Keep track of *only* the MRU block in each group

- On replacement, select victim as:
  - A not-MRU block in one of the not-MRU groups (randomly pick one of such blocks/groups)

Hierarchical LRU (not MRU): Questions

- 16-way cache
- 2 8-way groups

- What is an access pattern that performs worse than true LRU?

- What is an access pattern that performs better than true LRU?

Victim/Next-Victim Policy

- Only 2 blocks' status tracked in each set:
  - victim (V), next victim (NV)
  - all other blocks denoted as (O) – Ordinary block

- On a cache miss
  - Replace V
  - Demote NV to V
  - Randomly pick an O block as NV

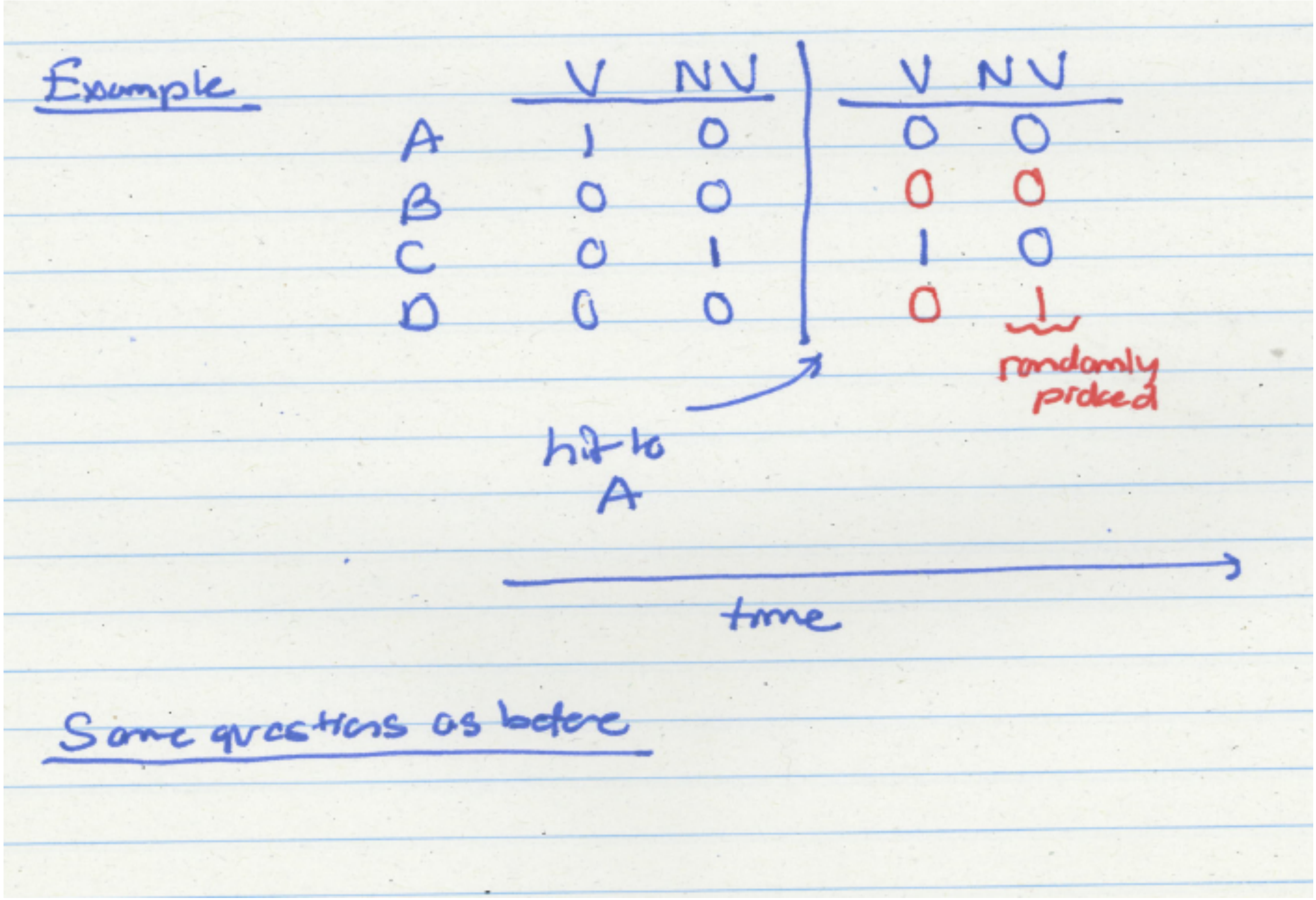
- On a cache hit to V
  - Demote NV to V
  - Randomly pick an O block as NV
  - Turn V to O

Victim/Next-Victim Policy (II)

- On a cache hit to NV
  - Randomly pick an O block as NV
  - Turn NV to O

- On a cache hit to O
  - Do nothing

Victim/Next-Victim Example



Example

	V	NV	V	NV
A	1	0	0	0
B	0	0	0	0
C	0	1	1	0
D	0	0	0	1

hit to A
randomly picked
time

Same questions as before

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?

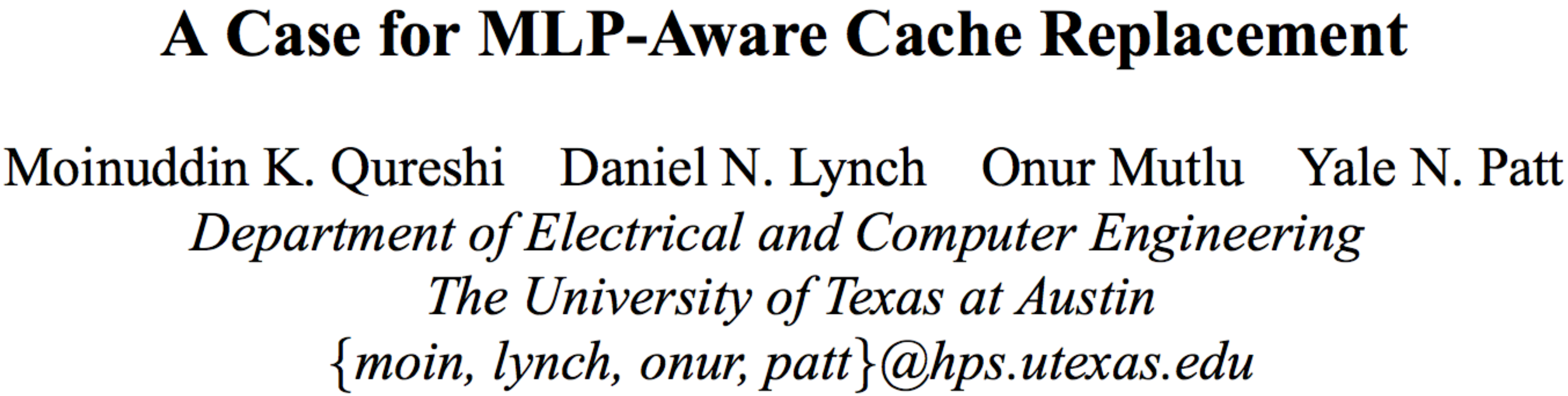
- Belady’s OPT
  - Replace the block that is going to be referenced furthest in the future by the program
  - Belady, “A study of replacement algorithms for a virtual-storage computer,” IBM Systems Journal, 1966.
  - How do we implement this? Simulate?

- 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

Handling Writes (II)

- Do we allocate a cache block on a write miss?
  - Allocate on write miss: Yes
  - No-allocate on write miss: No

- Allocate on write miss

  + Can combine writes instead of writing each of them individually to next level

  + Simpler because write misses can be treated the same way as read misses

  -- Requires (?) transfer of the whole cache block

- No-allocate

  + Conserves cache space if locality of writes is low (potentially better cache hit rate)

Handling Writes (III)

- 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

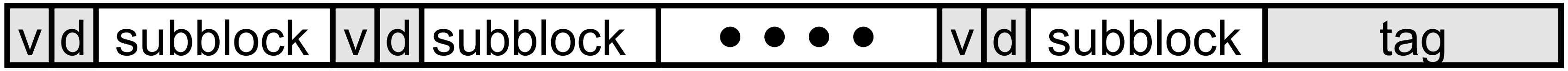
(A write simply validates and updates a subblock)

++ 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

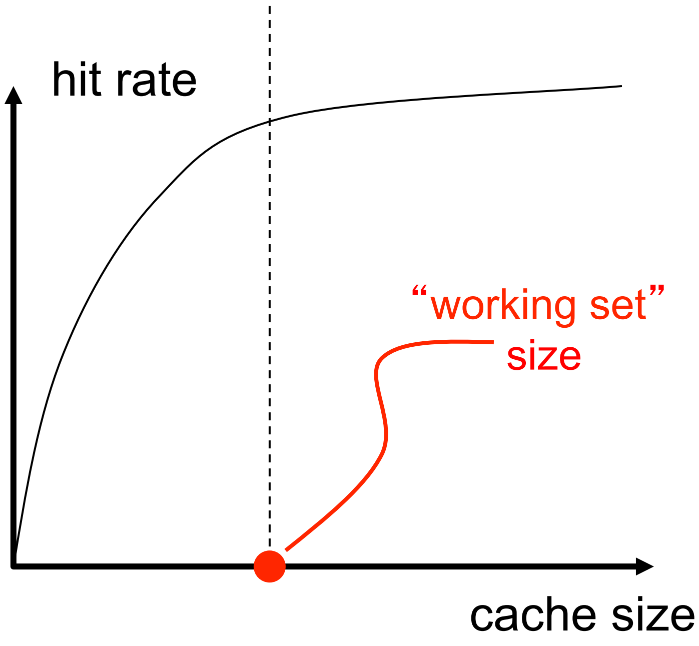
Cache Parameters vs. Miss/Hit Rate

- 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

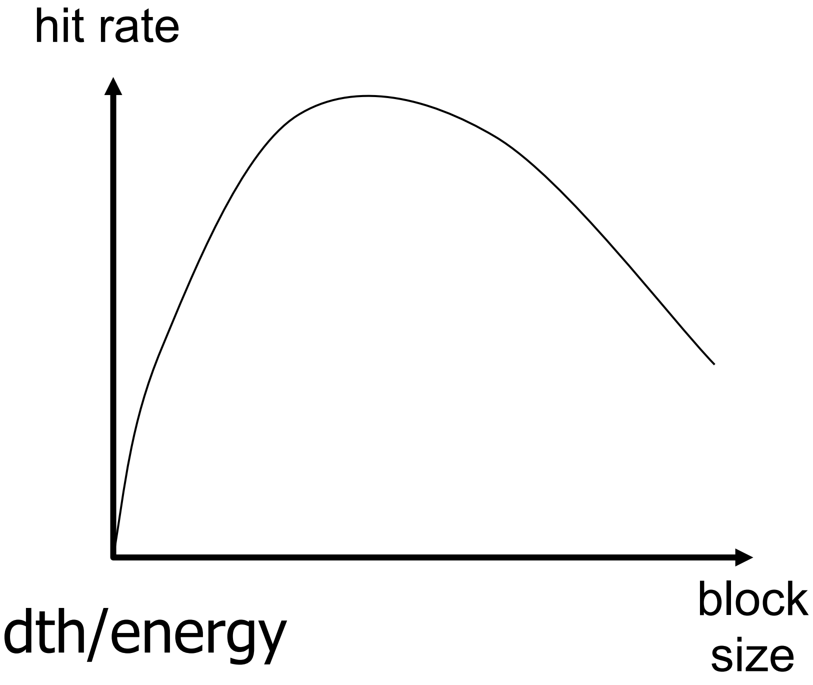


Block Size

- 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 with V bit)
      - Can improve "write" performance

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

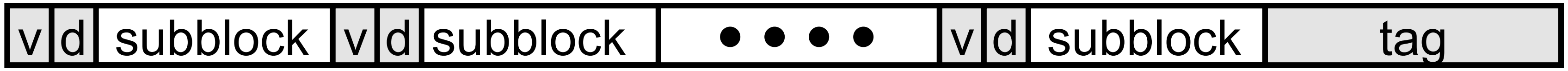
- Too large blocks
  - too few total # of blocks → less temporal locality exploitation
  - waste of cache space and bandwidth/energy if spatial locality is not high



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 bits for each subblock
  - When is this useful?



v	d	subblock	v	d	subblock	• • • •	v	d	subblock	tag

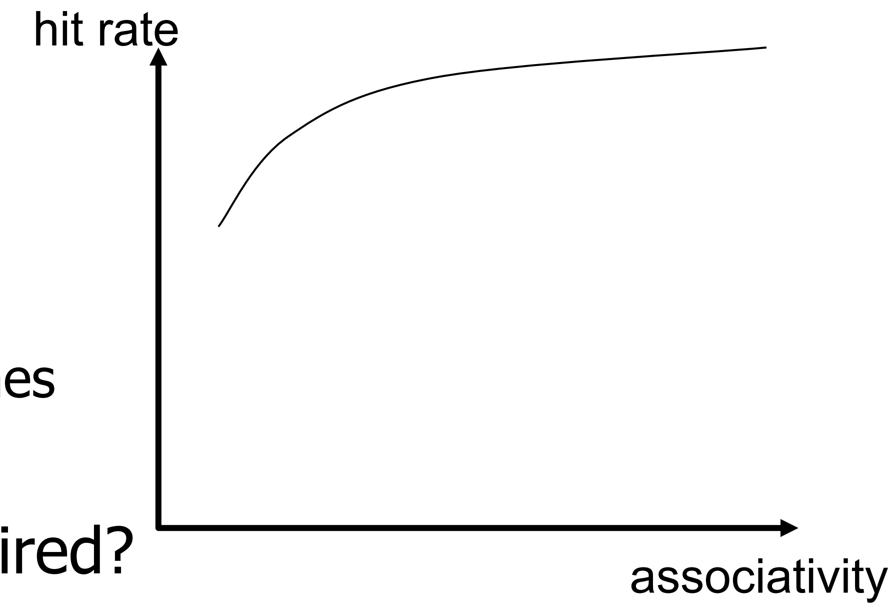
Associativity

- 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?



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

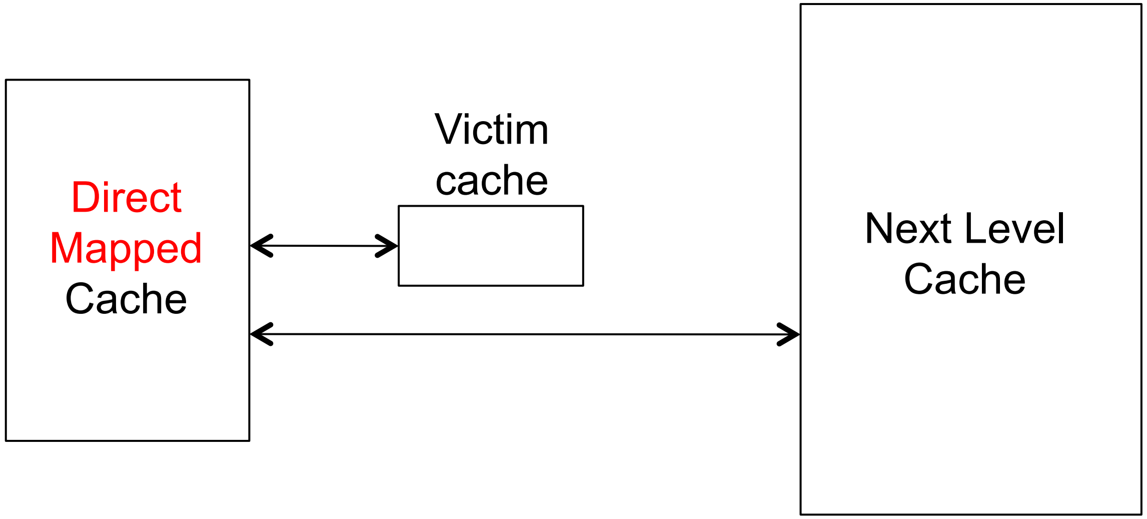
- Reducing miss rate
  - More associativity
  - Alternatives/enhancements to associativity
    - Victim caches, hashing, pseudo-associativity, skewed associativity
  - Better replacement/insertion policies
  - Software approaches
- Reducing miss latency/cost
  - Multi-level caches
  - Critical word first
  - 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, <u>sequentially</u> 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

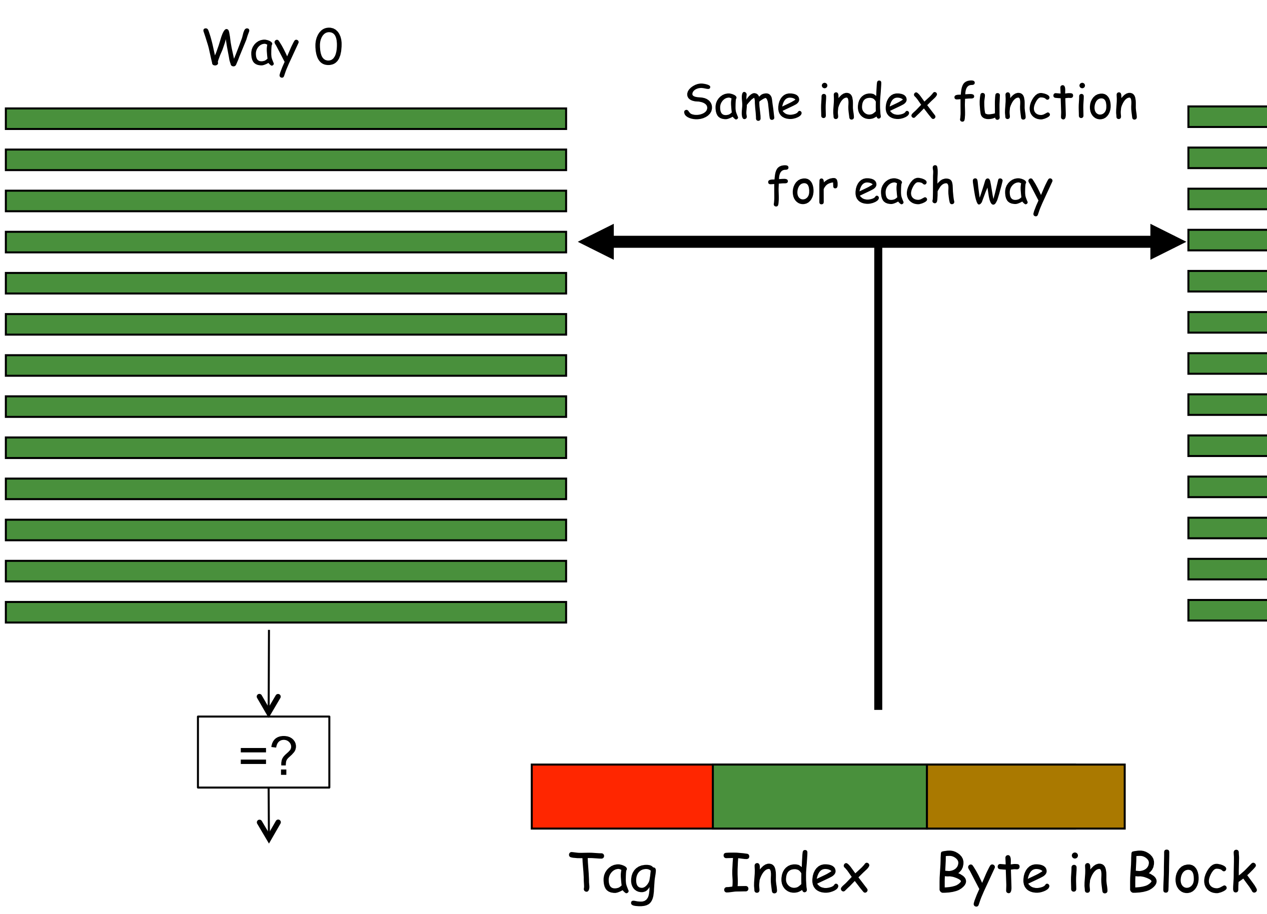
Skewed Associative Caches

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

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

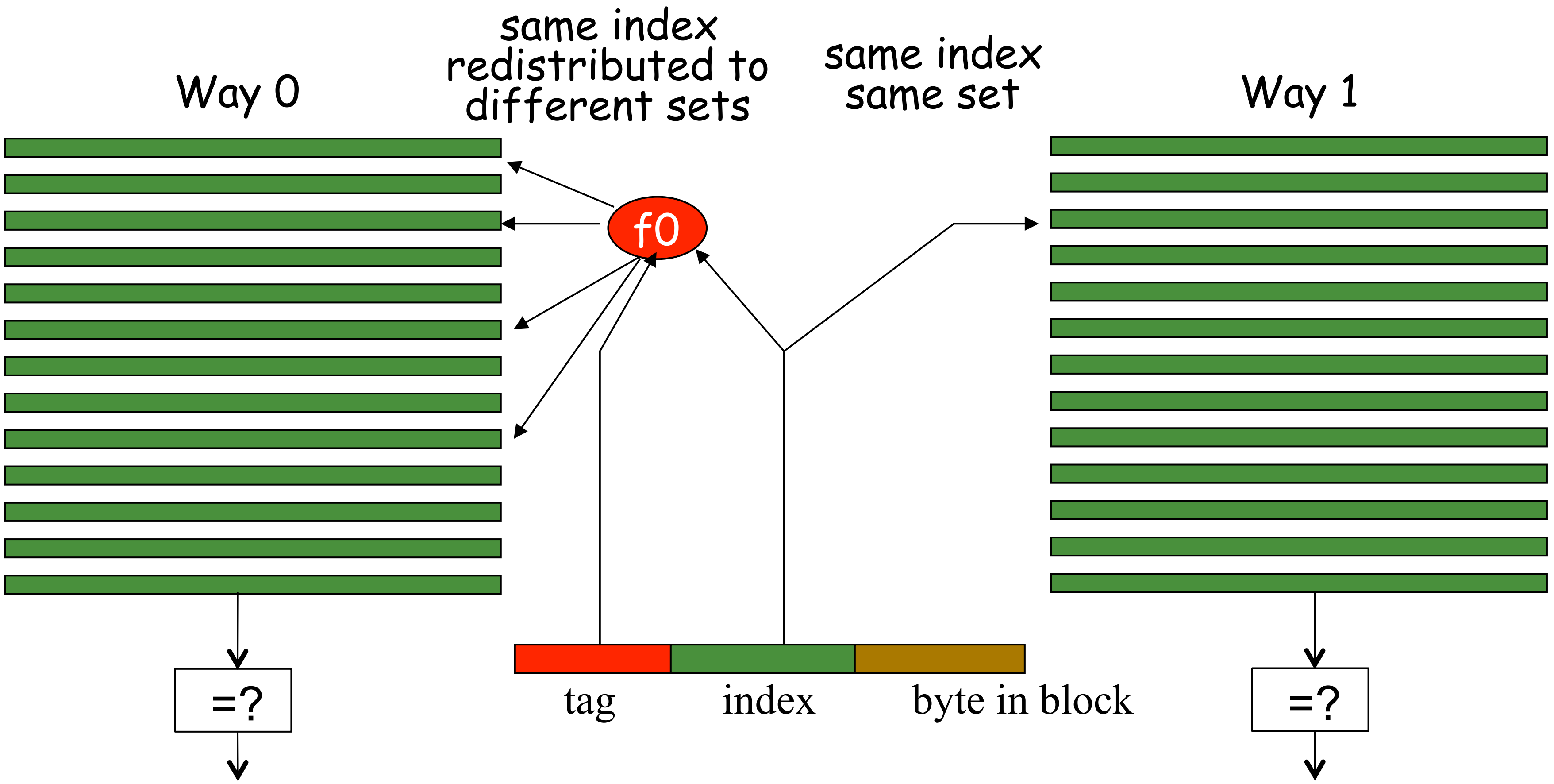
Skewed Associative Caches (I)

- 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?

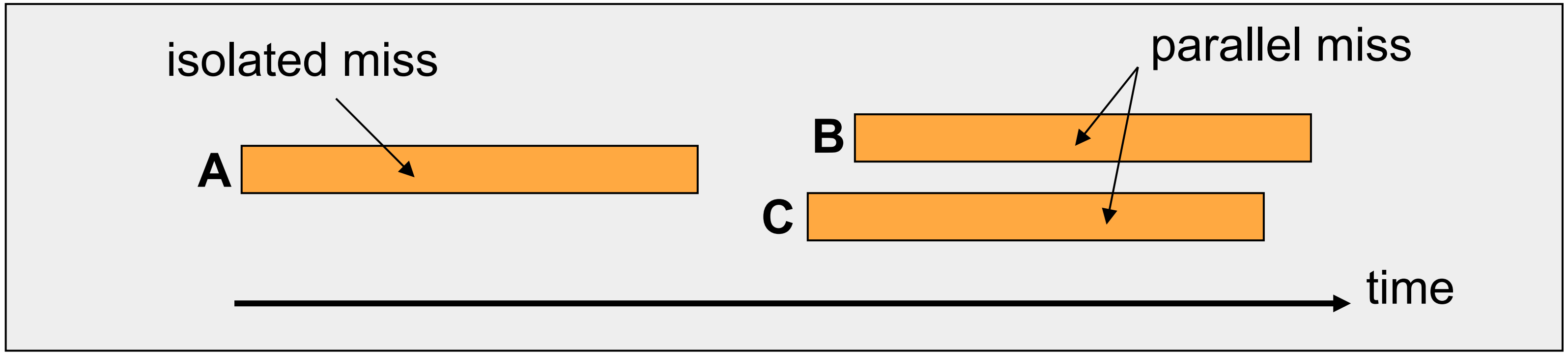
Improving Basic Cache Performance

- 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

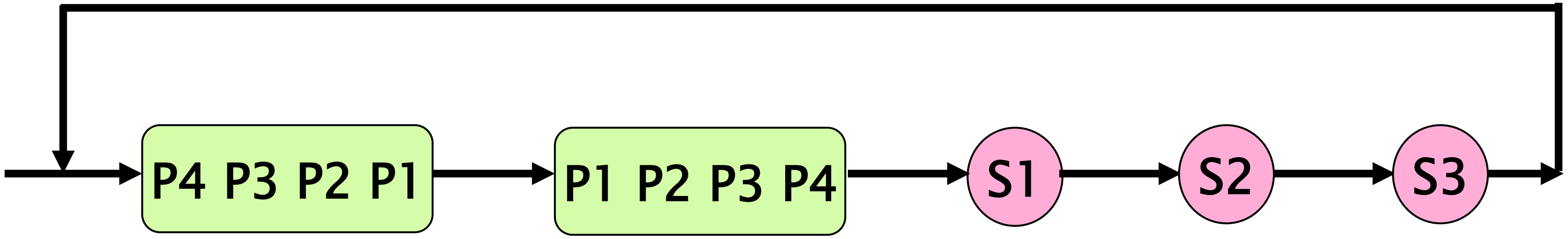
- Traditional cache replacement policies try to reduce miss count

- 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



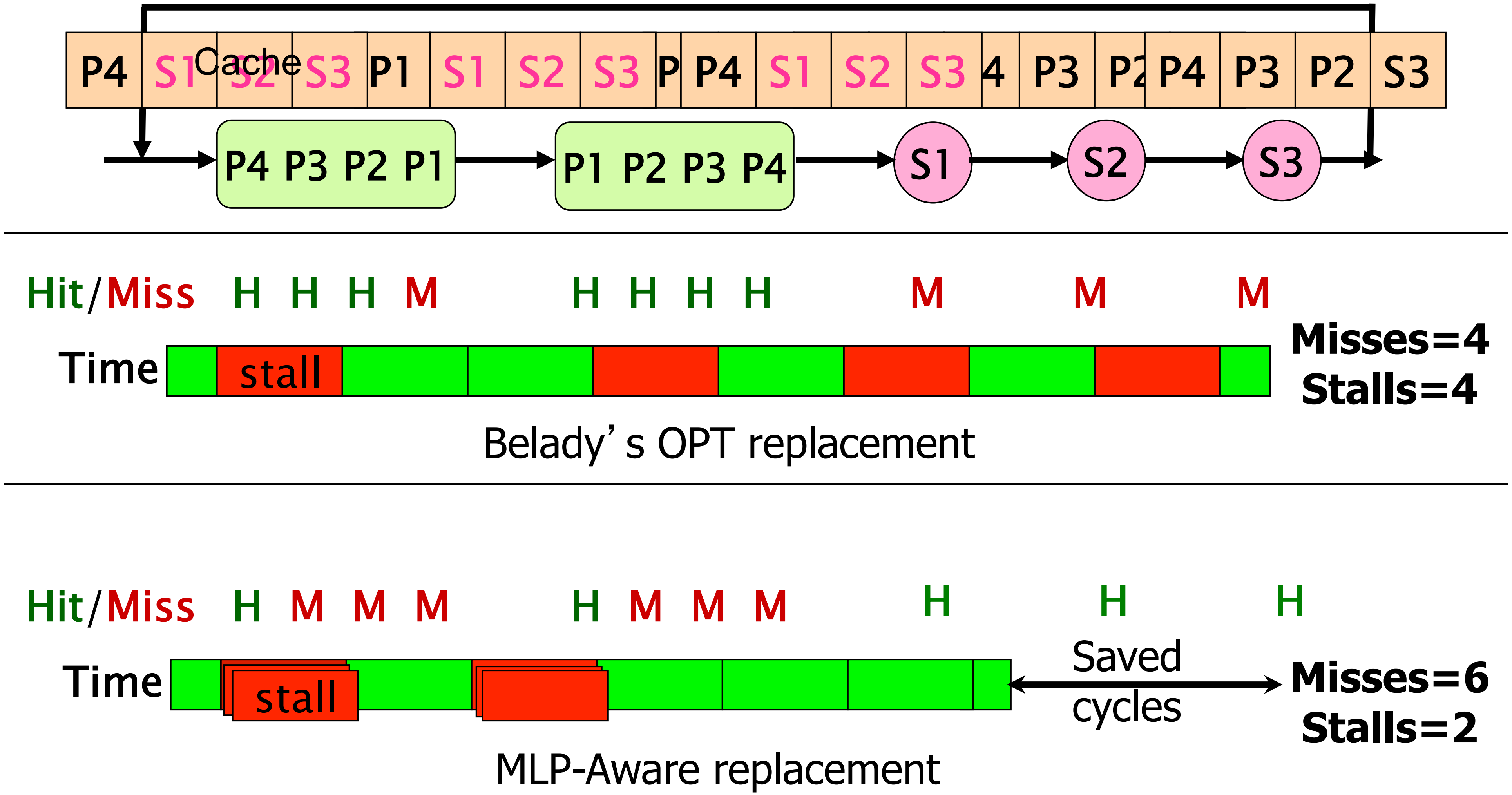
Misses to blocks P1, P2, P3, P4 can be parallel
Misses to blocks S1, S2, and S3 are isolated

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



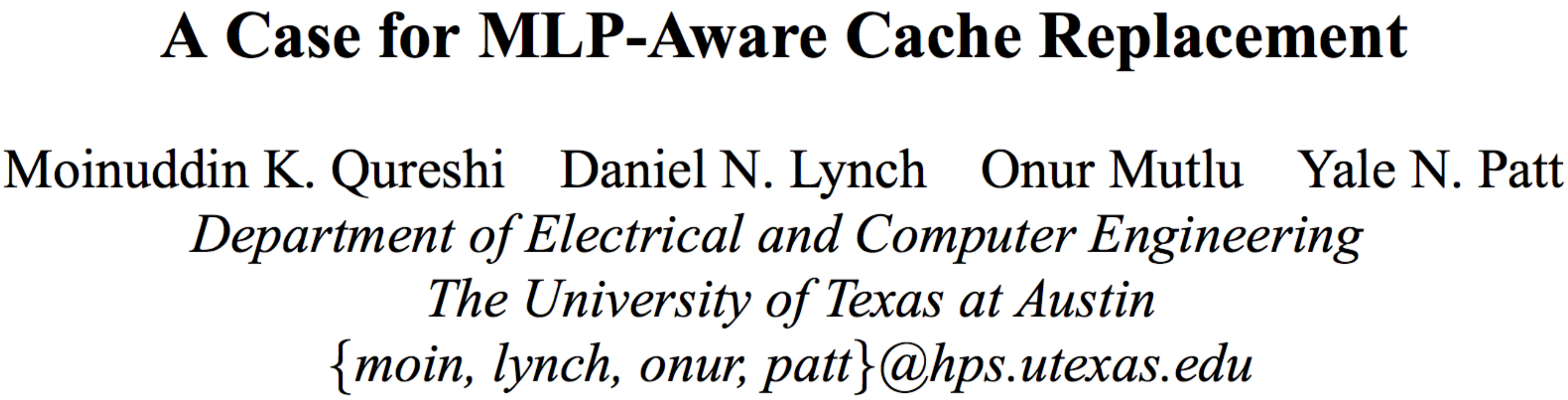
Belady's OPT replacement

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



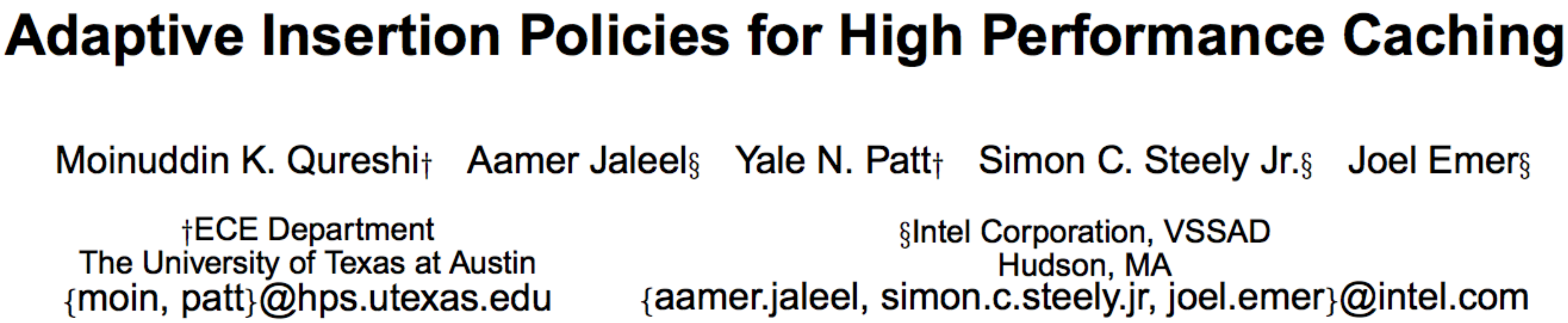
A Case for MLP-Aware Cache Replacement

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

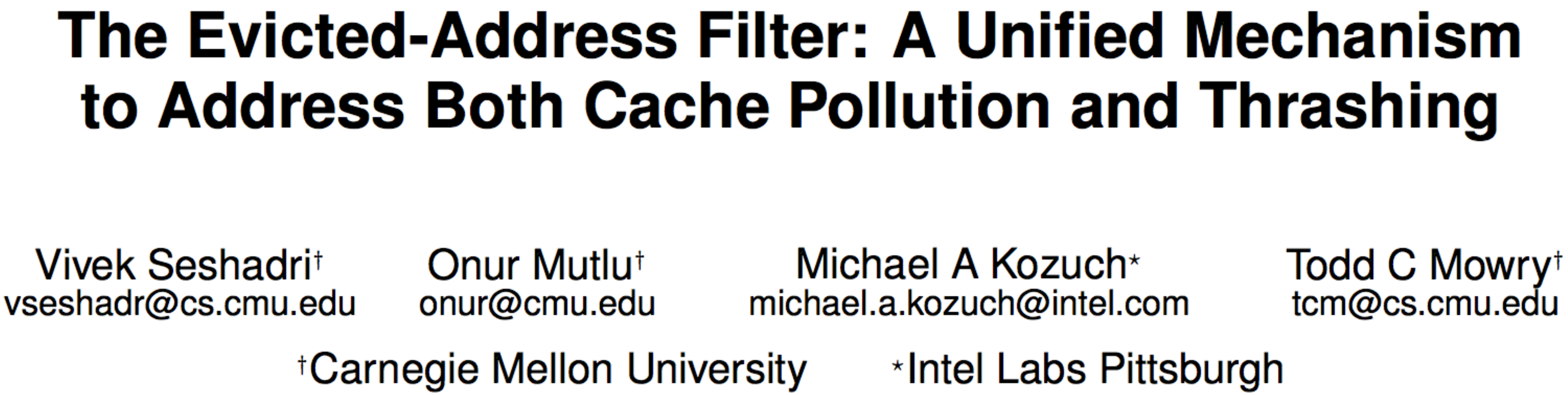
Moinuddin K. Qureshi† Aamer Jaleel§ Yale N. Patt† Simon C. Steely Jr.§ Joel Emer§

†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

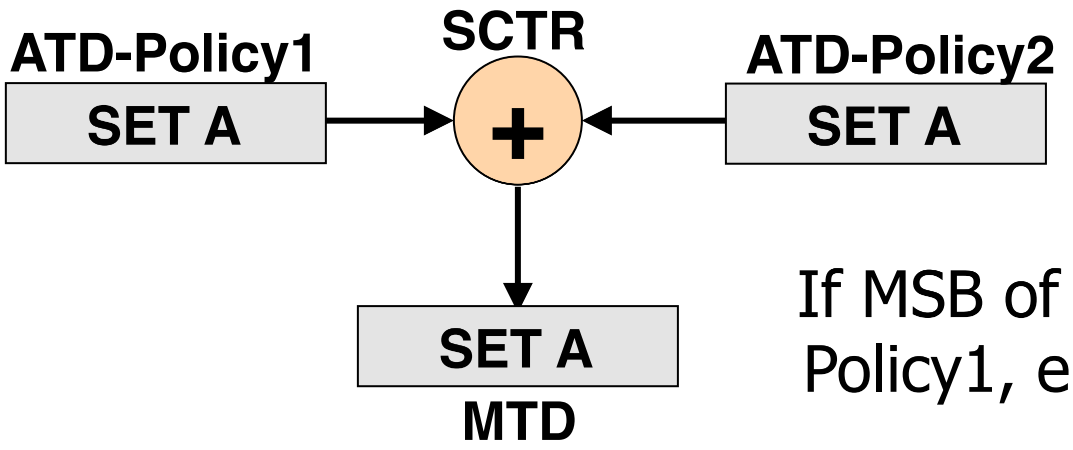
Terminology

- 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



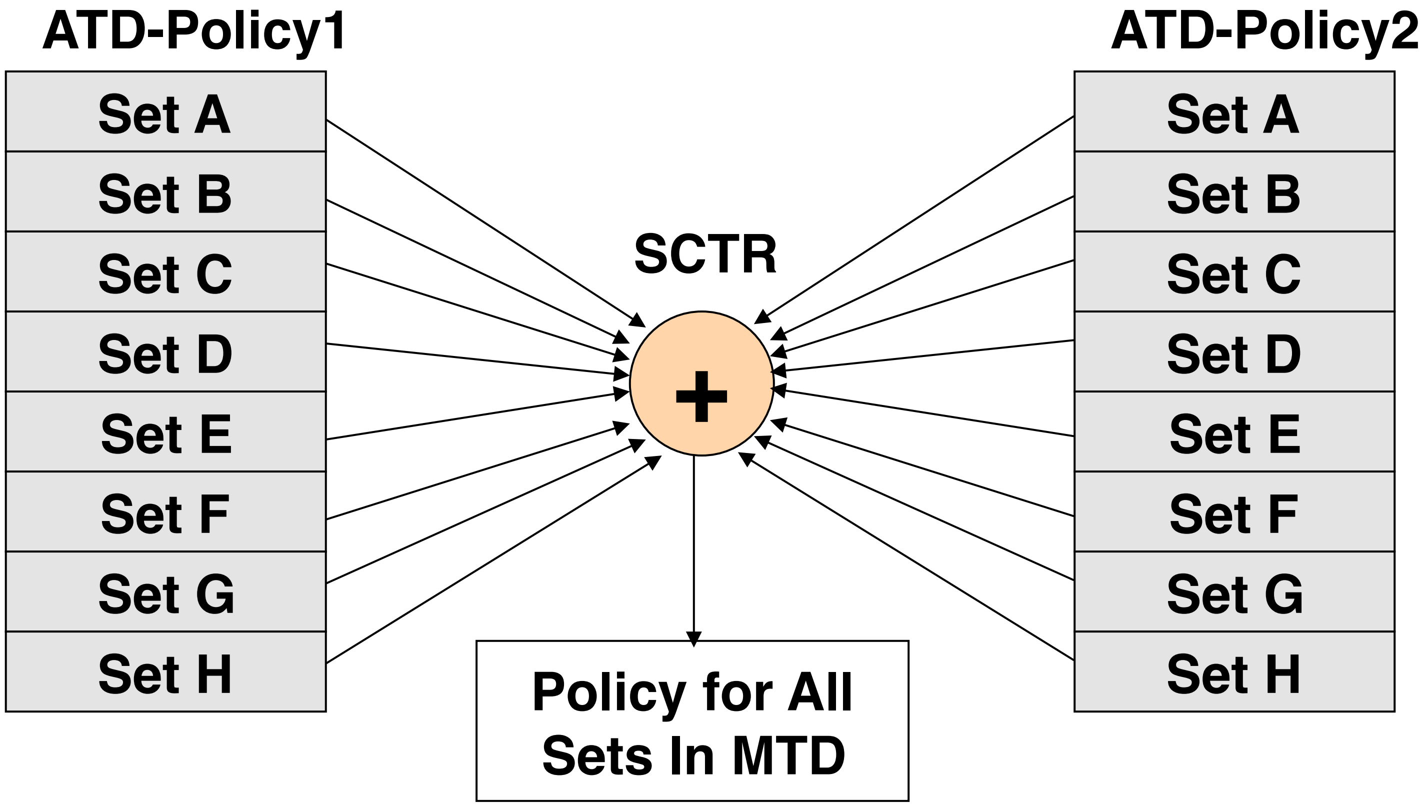
If MSB of SCTR is 1, MTD uses Policy1, else MTD uses Policy2

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

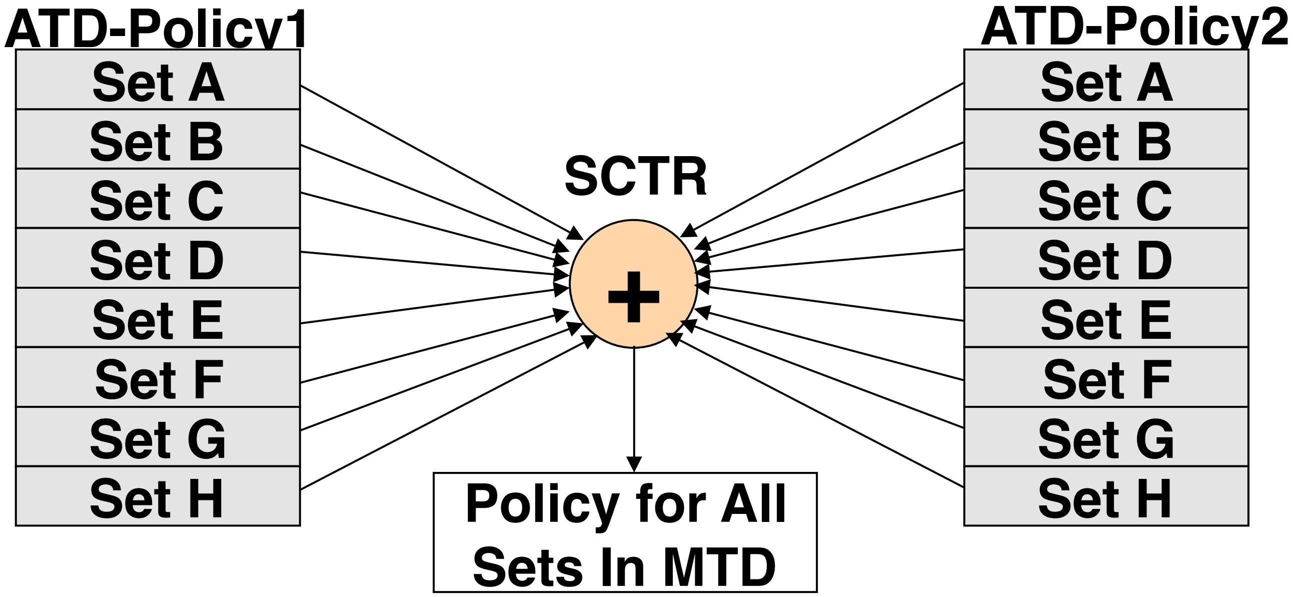
Implementing TSEL on a per-set basis is expensive
Counter overhead can be reduced by using a global counter



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)

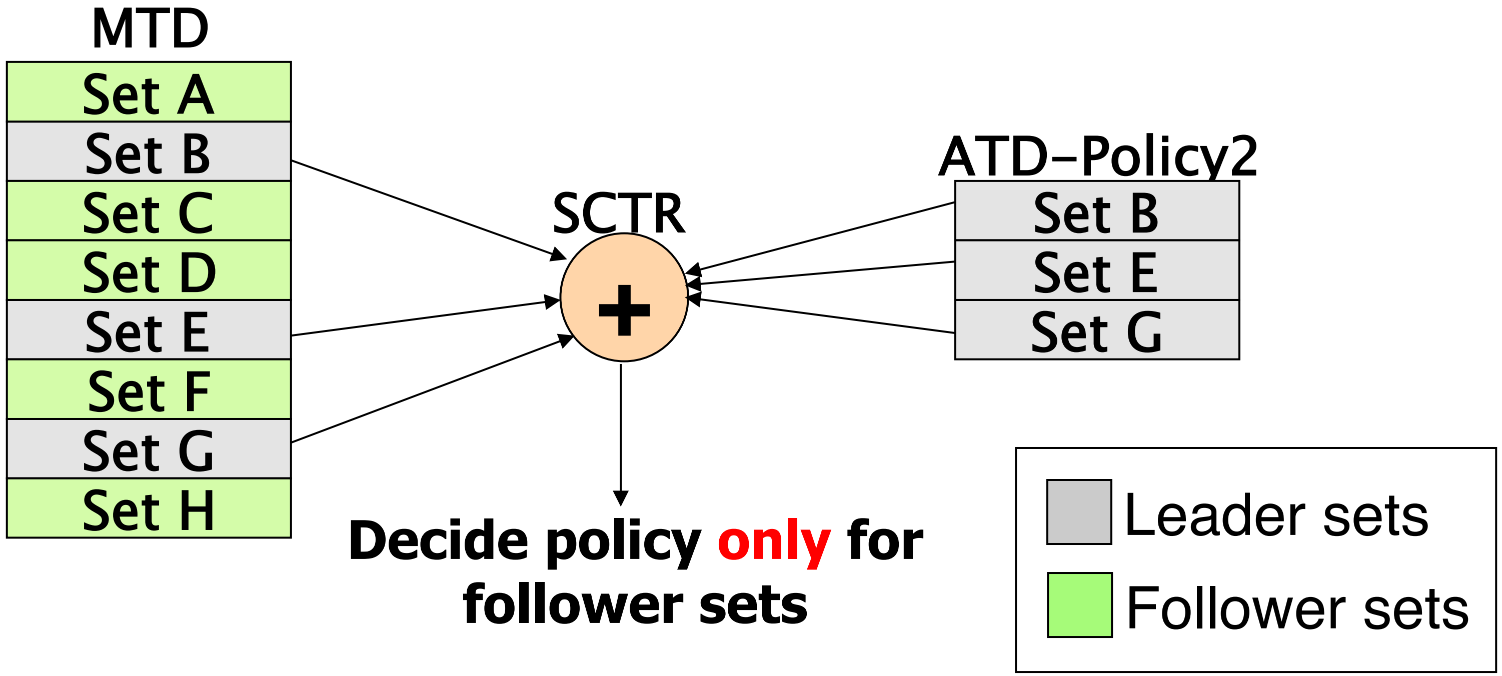
How many sets are required to choose best performing policy?

- ❑ 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.

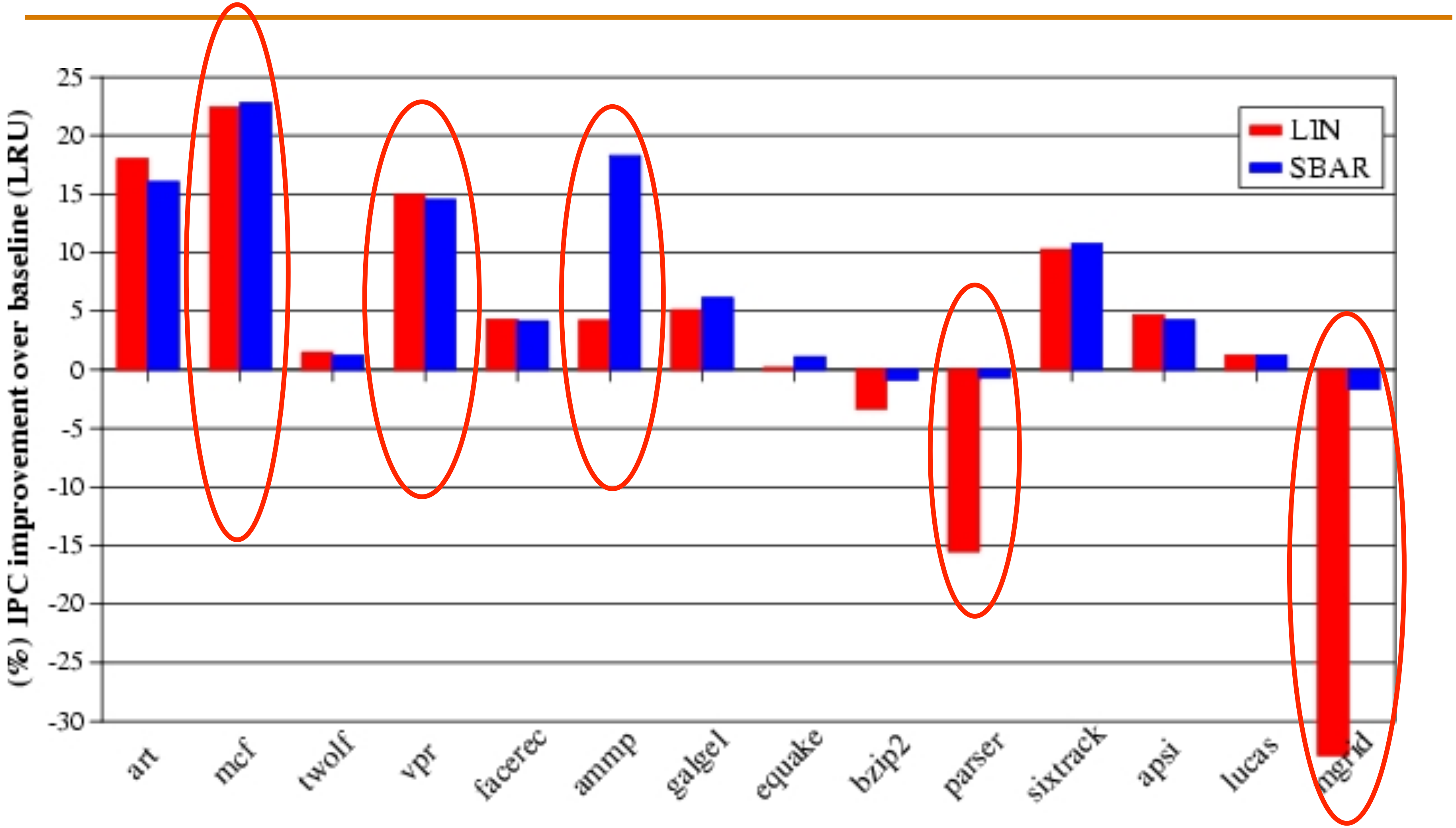
Sampling Based Adaptive Replacement (SBAR)



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

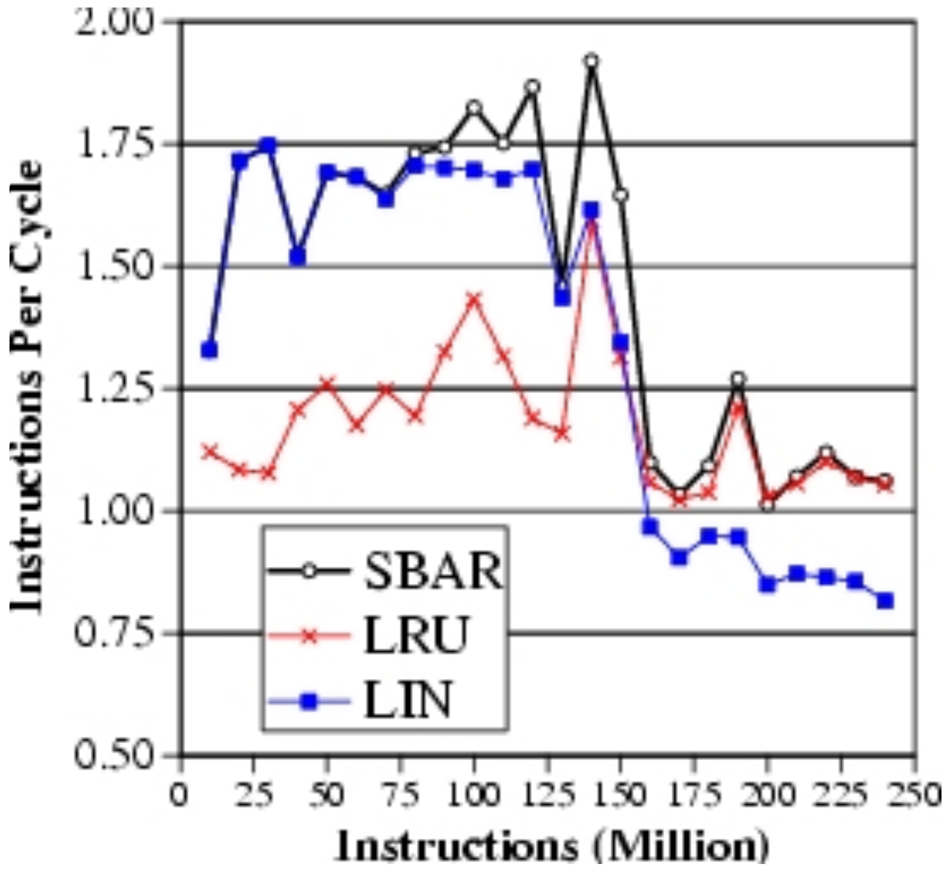
Qureshi et al., "A Case for MLP-Aware Cache Replacement," ISCA 2006.

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

1	27	1
Valid	Block Address	Issued

1	3	5	5	
Valid	Type	Block Offset	Destination	Load/store 0
Valid	Type	Block Offset	Destination	Load/store 1
Valid	Type	Block Offset	Destination	Load/store 2
Valid	Type	Block Offset	Destination	Load/store 3

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

Computer Architecture

Lecture 3: Cache Management and Memory Parallelism

Prof. Onur Mutlu
ETH Zürich

Fall 2017

27 September 2017

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

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)

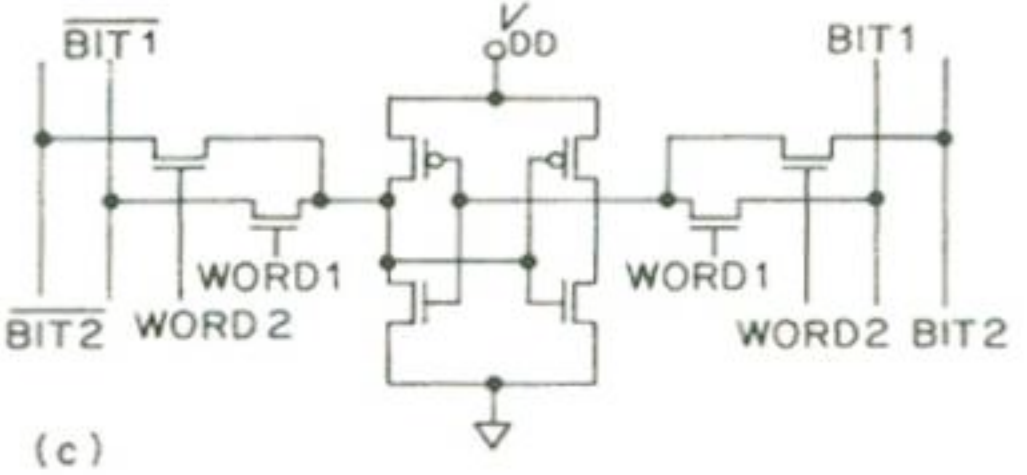
Handling Multiple Accesses per Cycle (I)

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

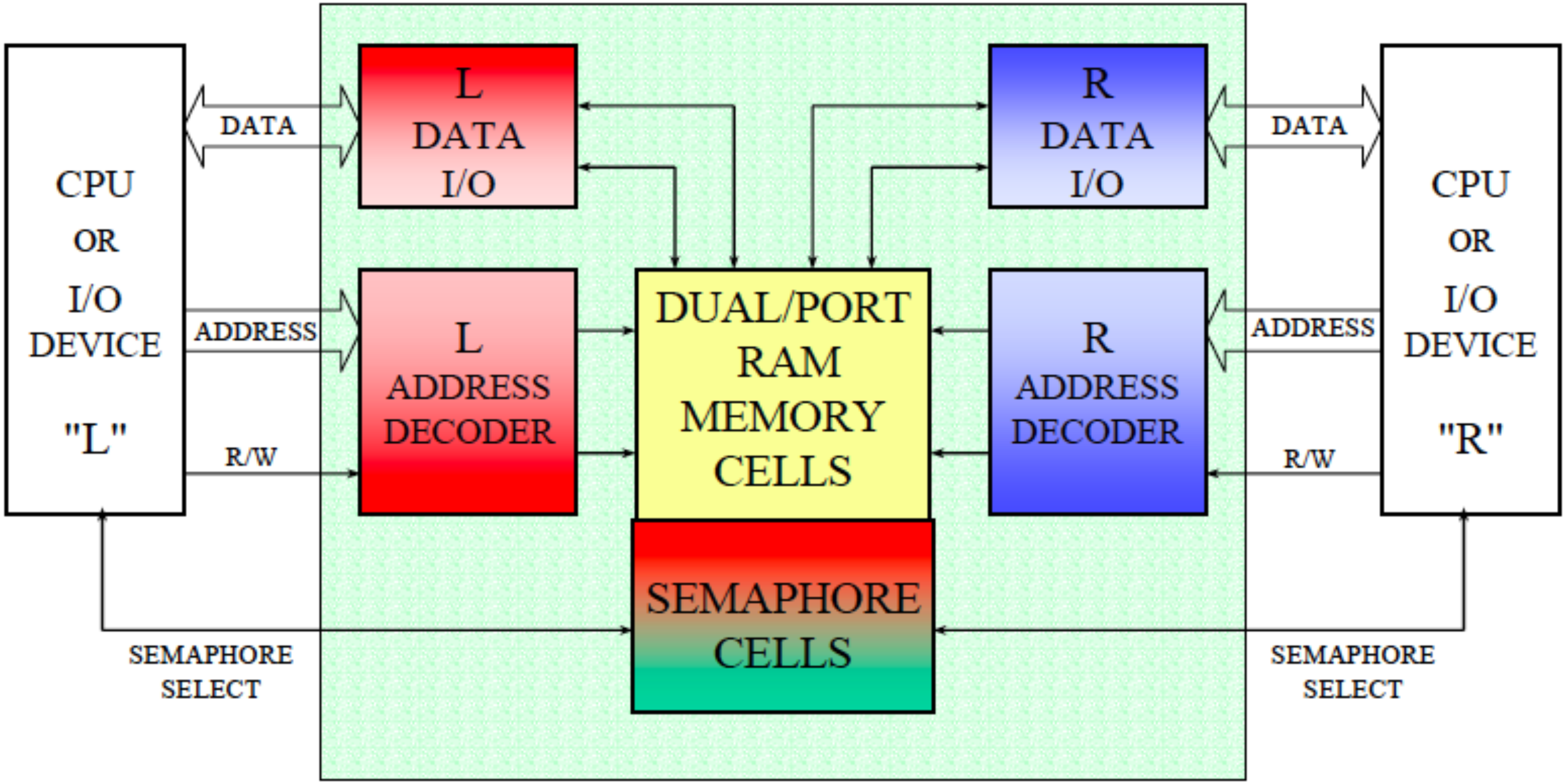
  + Truly concurrent accesses (no conflicts on read accesses)

  -- Expensive in terms of latency, power, area

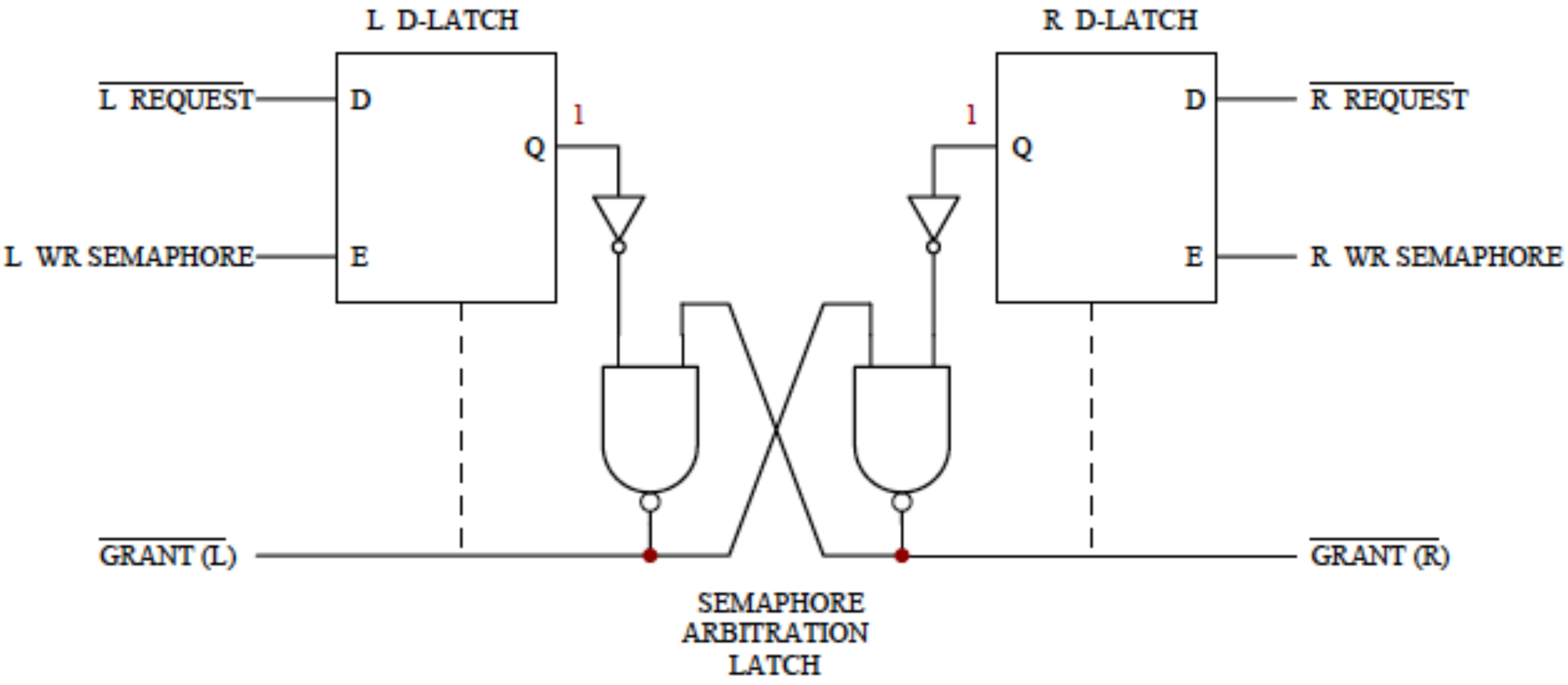
  - What about read and write to the same location at the same time?
    - Peripheral logic needs to handle this



Peripheral Logic for True Multiporting



Peripheral Logic for True Multiporting



Handling Multiple Accesses per Cycle (II)

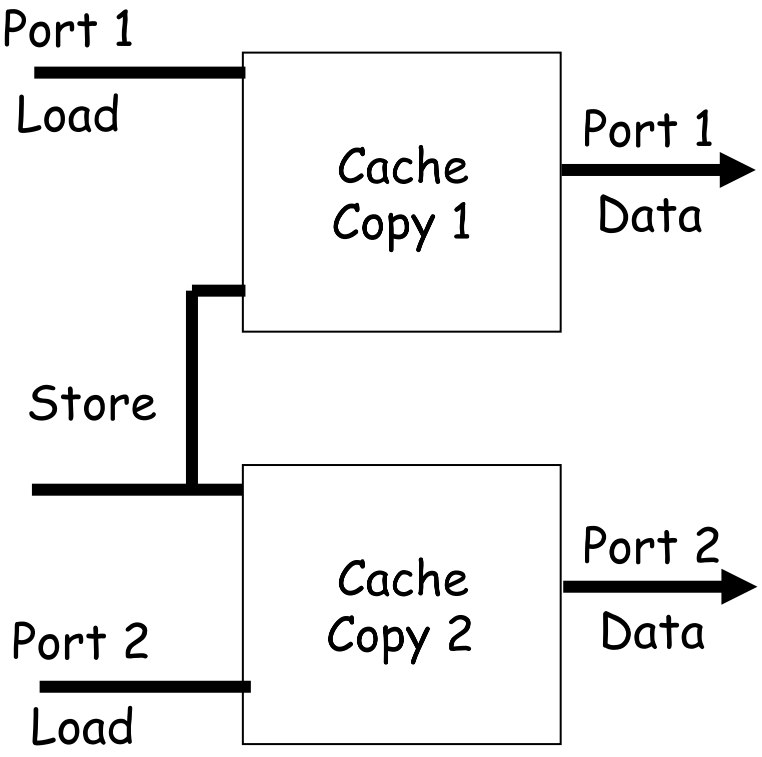
- Virtual multiporting
  - Time-share a single port
  - Each access needs to be (significantly) shorter than clock cycle
  - Used in Alpha 21264
  - Is this scalable?

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”



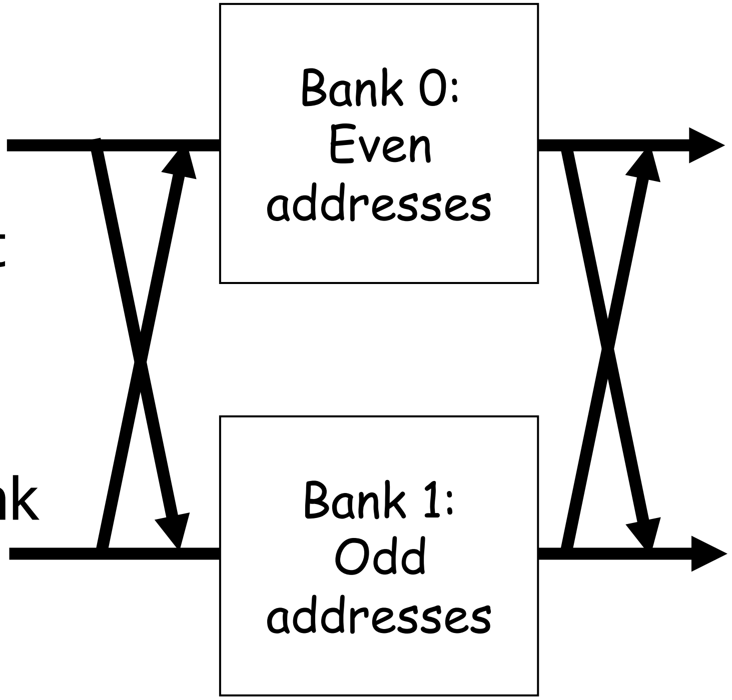
Handling Multiple Accesses per Cycle (III)

- 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
    - Accesses 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.

Multi-Core Issues in Caching

Caches in Multi-Core Systems

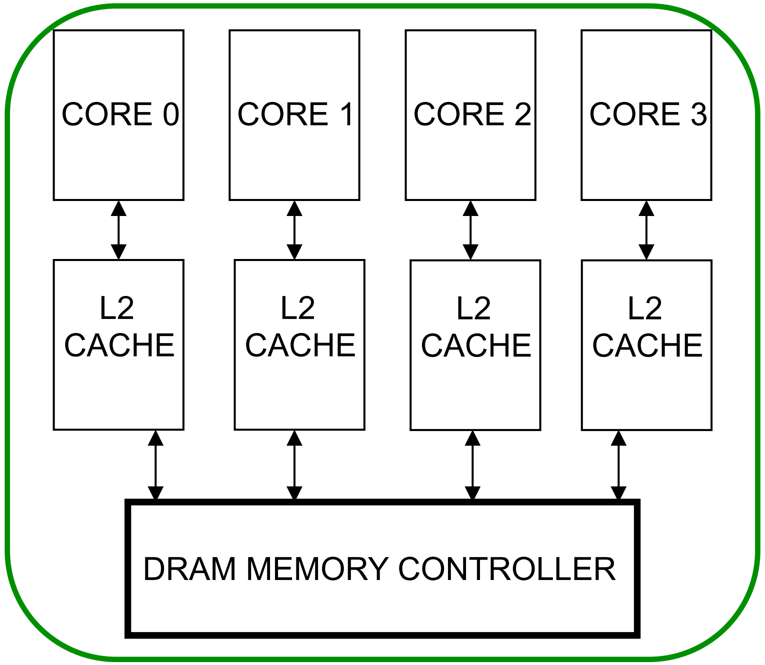
- Cache efficiency becomes even more important in a multi-core/multi-threaded system
  - Memory bandwidth is at premium
  - Cache space is a limited resource across cores/threads

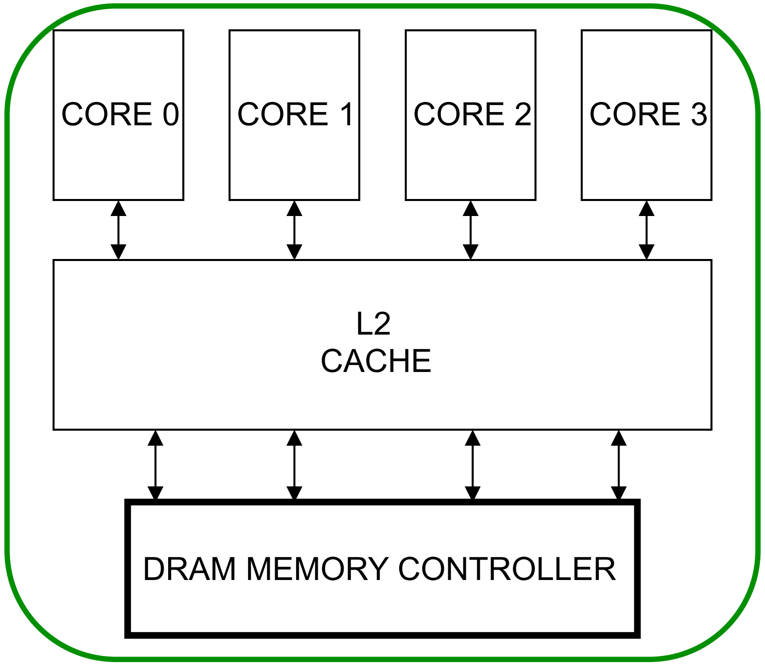
- How do we design the caches in a multi-core system?

- Many decisions
  - Shared vs. private caches
  - How to maximize performance of the entire system?
  - How to provide QoS to different threads in a shared cache?
  - Should cache management algorithms be aware of threads?
  - How should space be allocated to threads in a shared cache?

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





Resource Sharing Concept and Advantages

- Idea: Instead of dedicating a hardware resource to a hardware context, allow multiple contexts to use it
  - Example resources: functional units, pipeline, caches, buses, memory
- Why?

+ Resource sharing improves utilization/efficiency → throughput
  - When a resource is left idle by one thread, another thread can use it; no need to replicate shared data

+ Reduces communication latency
  - For example, data shared between multiple threads can be kept in the same cache in multithreaded processors

+ Compatible with the shared memory programming model

Resource Sharing Disadvantages

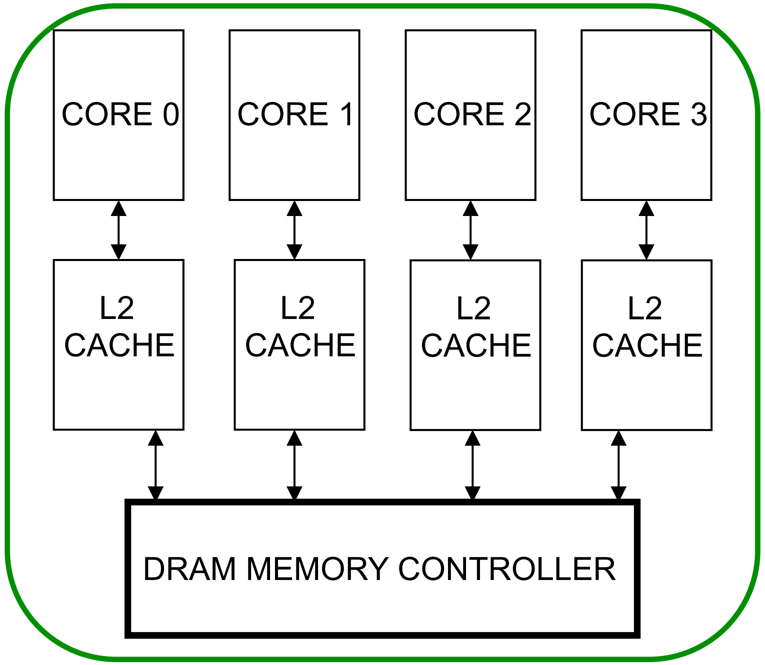
- Resource sharing results in contention for resources
  - When the resource is not idle, another thread cannot use it
  - If space is occupied by one thread, another thread needs to re-occupy it

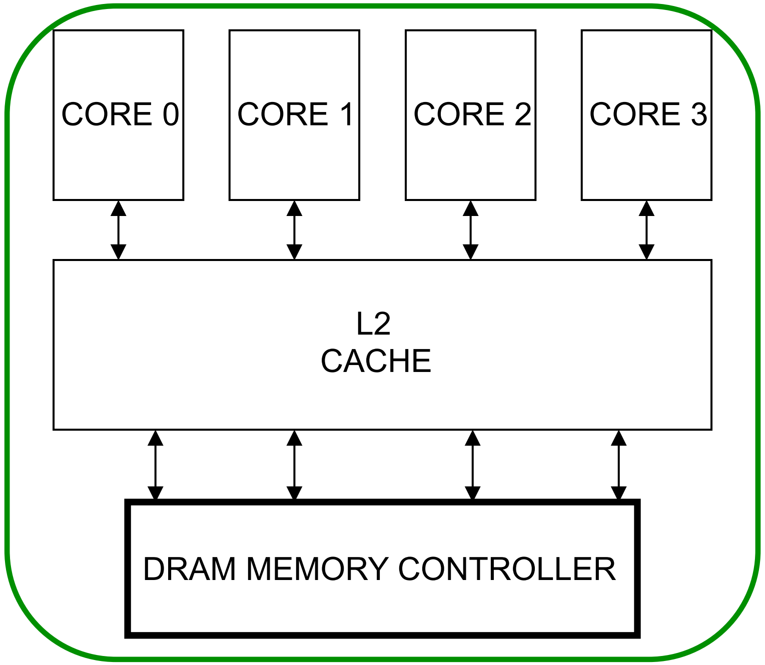
- Sometimes reduces each or some thread's performance
  - Thread performance can be worse than when it is run alone
- Eliminates performance isolation → inconsistent performance across runs
  - Thread performance depends on co-executing threads
- Uncontrolled (free-for-all) sharing degrades QoS
  - Causes unfairness, starvation

Need to efficiently and fairly utilize shared resources

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





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)
  - Shared data and locks do not ping pong between caches – stay in one place

- Disadvantages
  - Slower access (cache not tightly coupled with the core)
  - Cores incur conflict misses due to other cores' accesses
    - Misses due to inter-core interference
    - Some cores can destroy the hit rate of other cores
  - Guaranteeing a minimum level of service (or fairness) to each core is harder (how much space, how much bandwidth?)

Shared Caches: How to Share?

- Free-for-all sharing
  - Placement/replacement policies are the same as a single core system (usually LRU or pseudo-LRU)
  - Not thread/application aware
  - An incoming block evicts a block regardless of which threads the blocks belong to

- Problems
  - Inefficient utilization of cache: LRU is not the best policy
  - A cache-unfriendly application can destroy the performance of a cache friendly application
  - Not all applications benefit equally from the same amount of cache: free-for-all might prioritize those that do not benefit
  - Reduced performance, reduced fairness

Example: Utility Based Shared Cache Partitioning

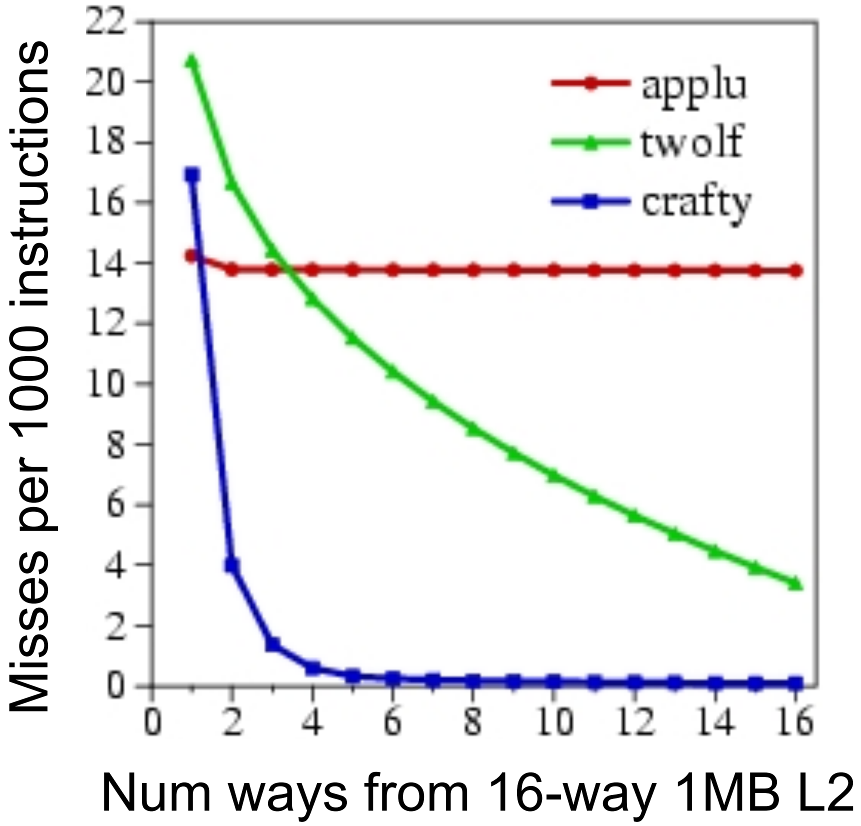
- Goal: Maximize system throughput
- Observation: Not all threads/applications benefit equally from caching → simple LRU replacement not good for system throughput
- Idea: Allocate more cache space to applications that obtain the most benefit (i.e., marginal utility) from more space

- The high-level idea can be applied to other shared resources as well.

- Qureshi and Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," MICRO 2006.
- Suh et al., "A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning," HPCA 2002.

Marginal Utility of a Cache Way

Utility U_a^b = Misses with a ways – Misses with b ways

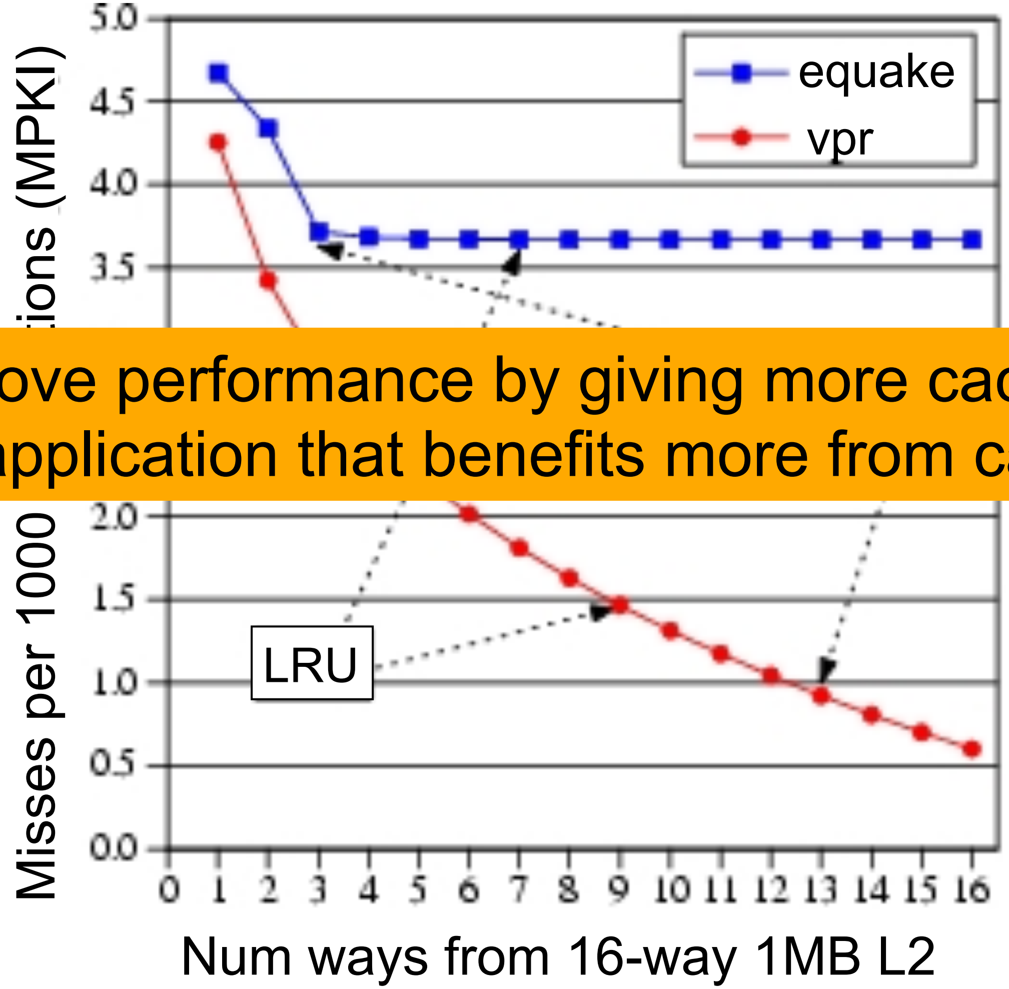


Low Utility

High Utility

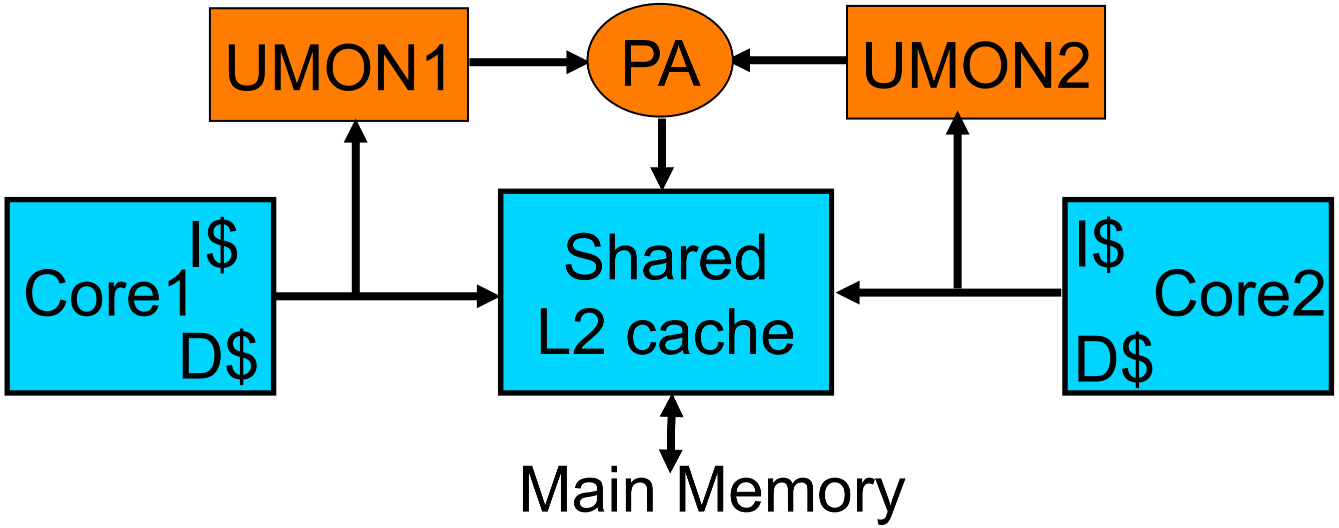
Saturating Utility

Utility Based Shared Cache Partitioning Motivation



Improve performance by giving more cache to the application that benefits more from cache

Utility Based Cache Partitioning (III)



Three components:

- ❑ Utility Monitors (UMON) per core
- ❑ Partitioning Algorithm (PA)
- ❑ Replacement support to enforce partitions

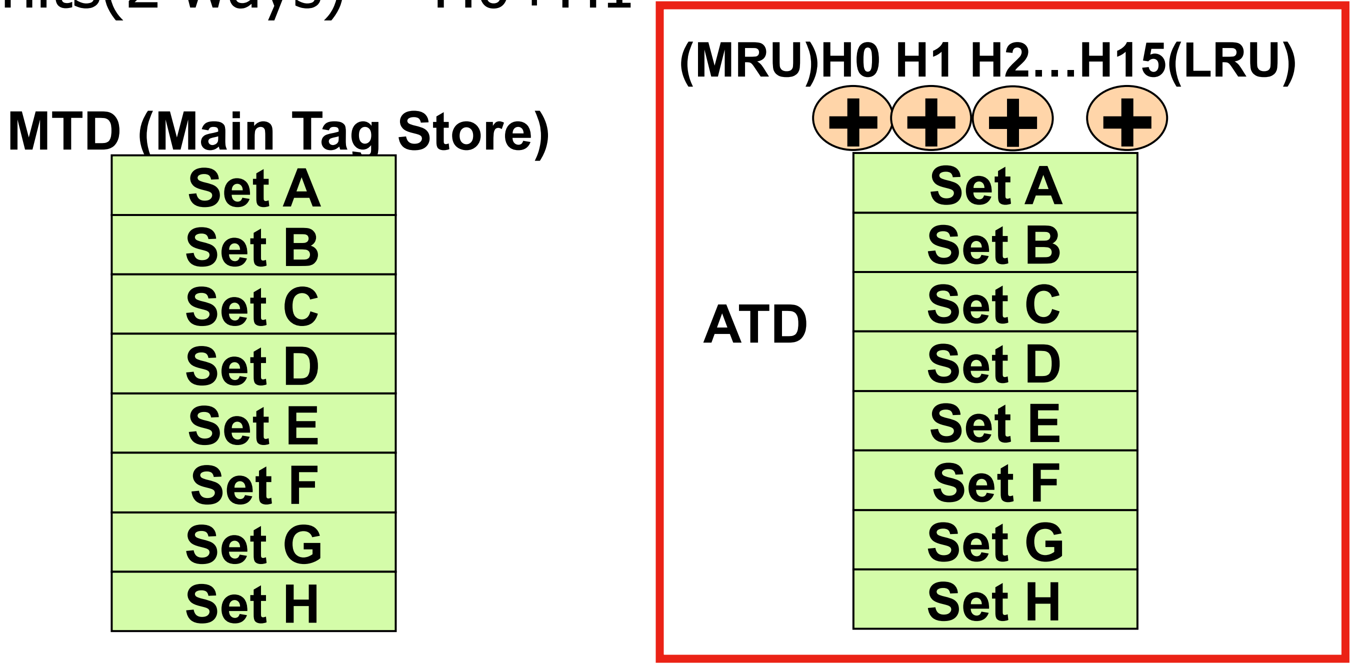
1. Utility Monitors

- For each core, simulate LRU policy using a separate tag store called ATD (auxiliary tag directory/store)

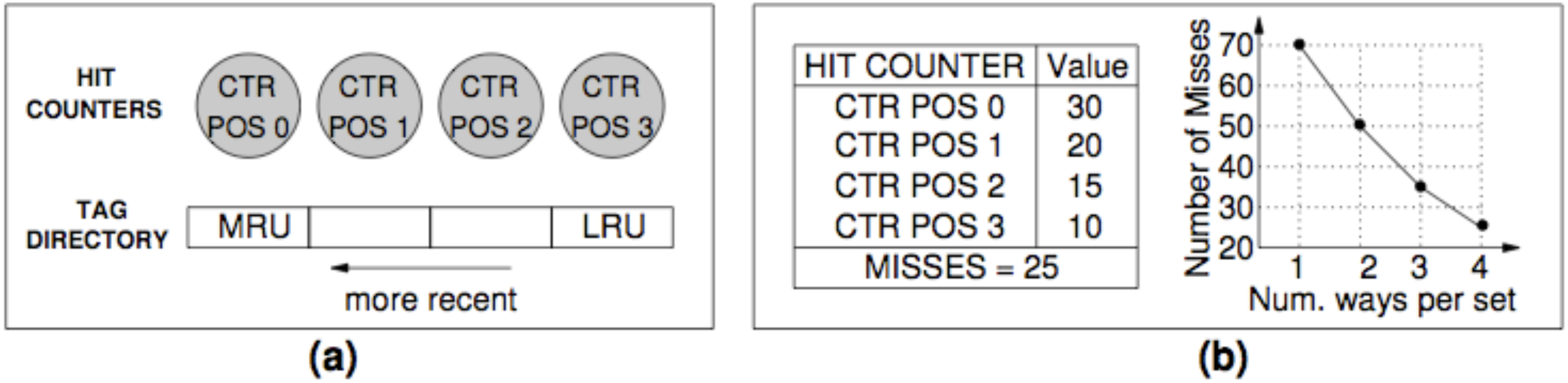
- Hit counters in ATD to count hits per recency position

- LRU is a stack algorithm: hit counts ➔ utility

  E.g. hits(2 ways) = H0+H1



Utility Monitors

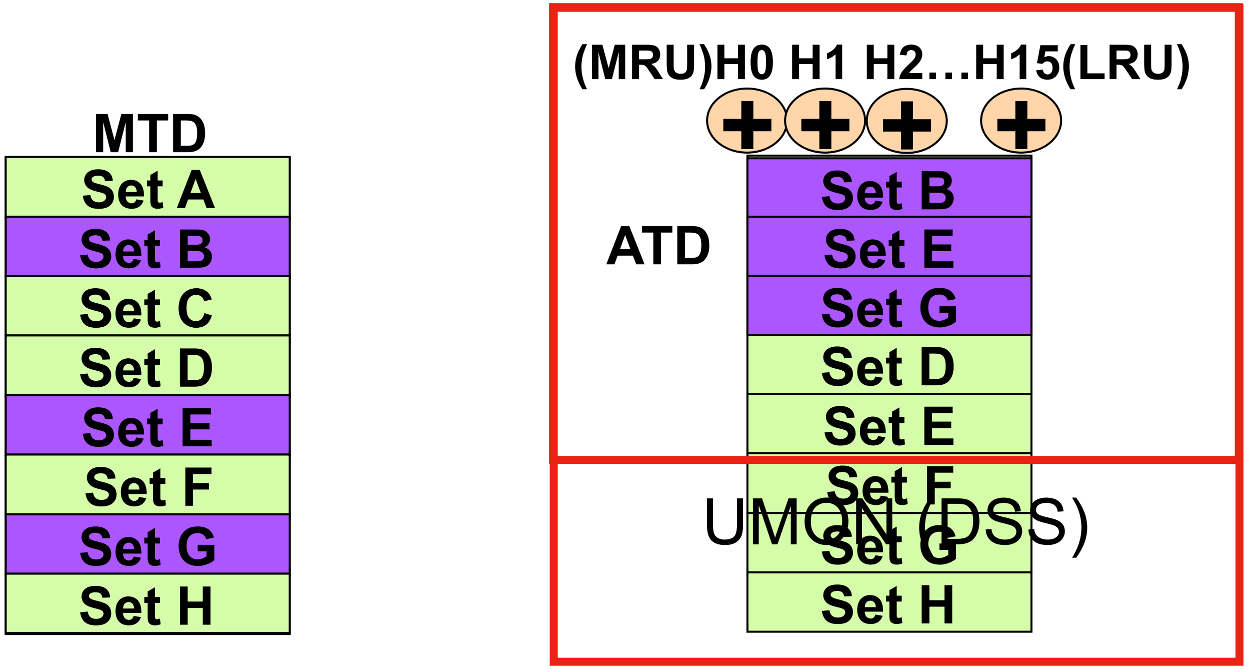


HIT COUNTER	Value
CTR POS 0	30
CTR POS 1	20
CTR POS 2	15
CTR POS 3	10
MISSES = 25	

Figure 4. (a) Hit counters for each recency position. (b) Example of how utility information can be tracked with stack property.

Dynamic Set Sampling

- Extra tags incur hardware and power overhead
- Sampling sets reduces overhead [Qureshi+ ISCA' 06]
- Sampling 32 sets sufficient (analytical bounds)
- Storage < 2kB/UMON



2. Partitioning Algorithm

- Evaluate all possible partitions and select the best

- With a ways to core1 and (16-a) ways to core2:

  $\text{Hits}_{\text{core1}} = (H_0 + H_1 + \dots + H_{a-1})$ ---- from UMON1

  $\text{Hits}_{\text{core2}} = (H_0 + H_1 + \dots + H_{16-a-1})$ ---- from UMON2

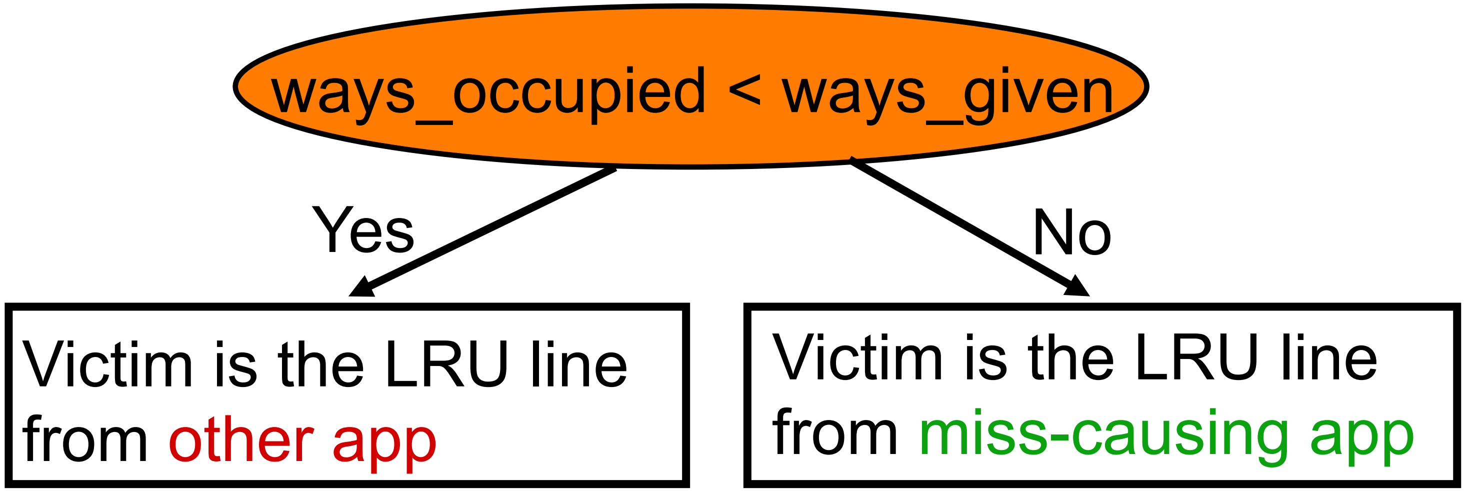
- Select a that maximizes $(\text{Hits}_{\text{core1}} + \text{Hits}_{\text{core2}})$

- Partitioning done once every 5 million cycles

3. Enforcing Partitions: Way Partitioning

Way partitioning support: [Suh+ HPCA'02, Iyer ICS'04]

1. Each line has core-id bits
2. On a miss, count ways_occupied in set by miss-causing app


ways_occupied < ways_given
Yes
No
Victim is the LRU line from other app
Victim is the LRU line from miss-causing app

Performance Metrics

- Three metrics for performance:

1. Weighted Speedup (default metric)

   ➔ perf = $IPC_1/AloneIPC_1 + IPC_2/AloneIPC_2$

   ➔ correlates with reduction in execution time

2. Throughput

   ➔ perf = $IPC_1 + IPC_2$

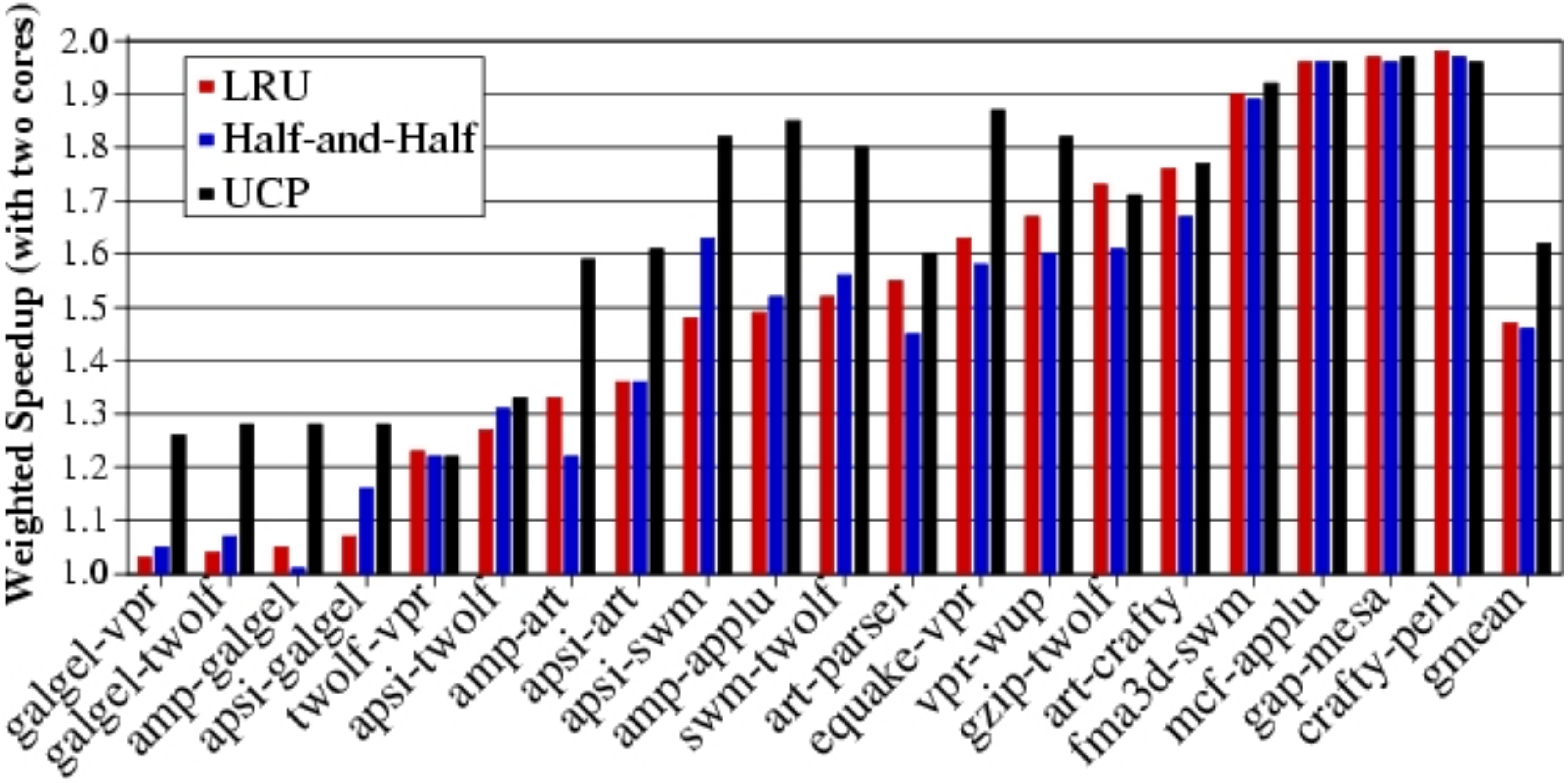
   ➔ can be unfair to low-IPC application

3. Harmonic mean of Speedups

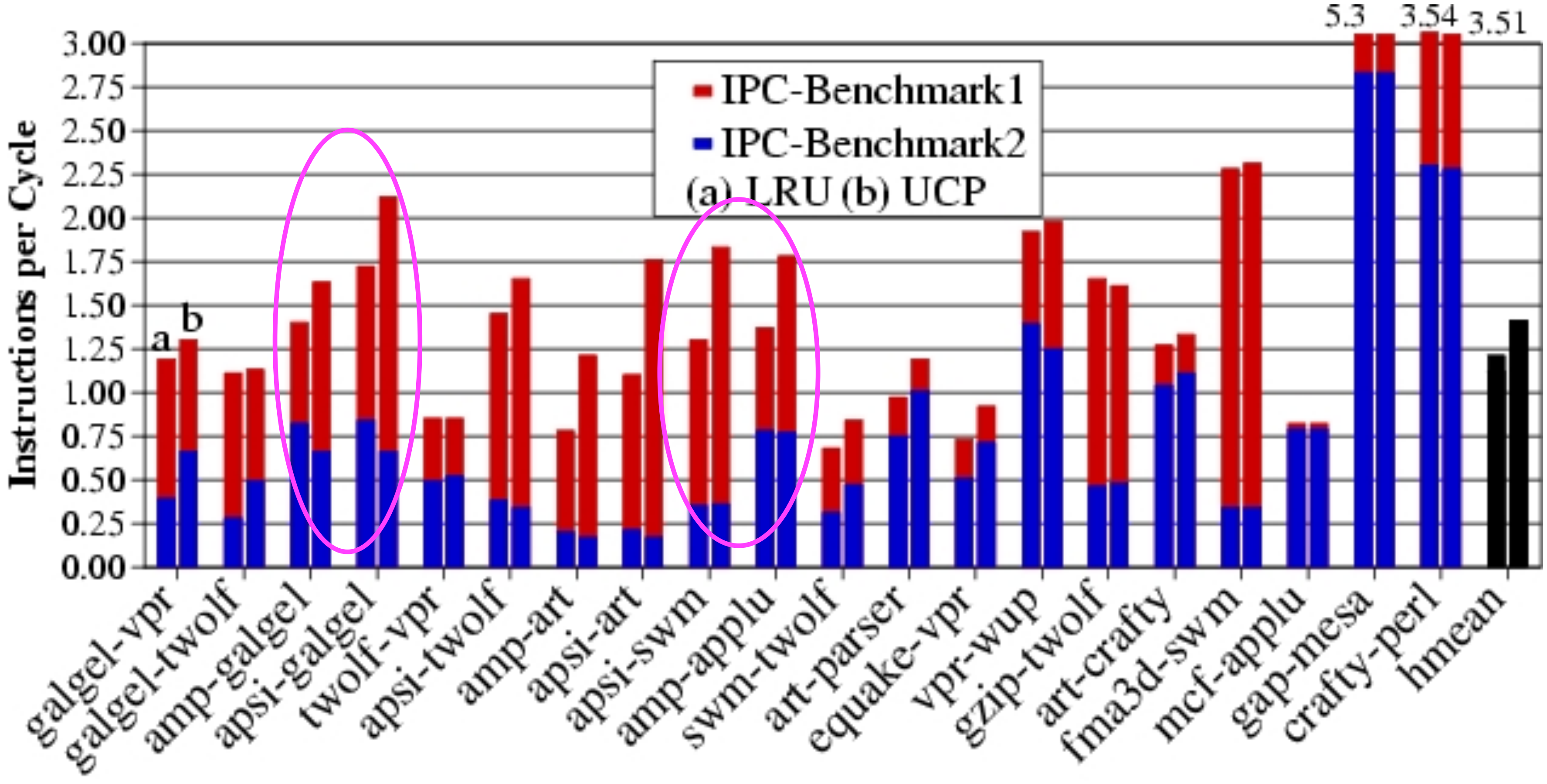
   ➔ perf = hmean($IPC_1/AloneIPC_1$, $IPC_2/AloneIPC_2$)

   ➔ balances fairness and performance

Weighted Speedup Results for UCP



IPC Results for UCP



UCP improves average throughput by 17%

Any Problems with UCP So Far?

- Scalability

- Non-convex curves?

- Time complexity of partitioning low for two cores (number of possible partitions ≈ number of ways)

- Possible partitions increase exponentially with cores

- For a 32-way cache, possible partitions:
  - 4 cores → 6545
  - 8 cores → 15.4 million

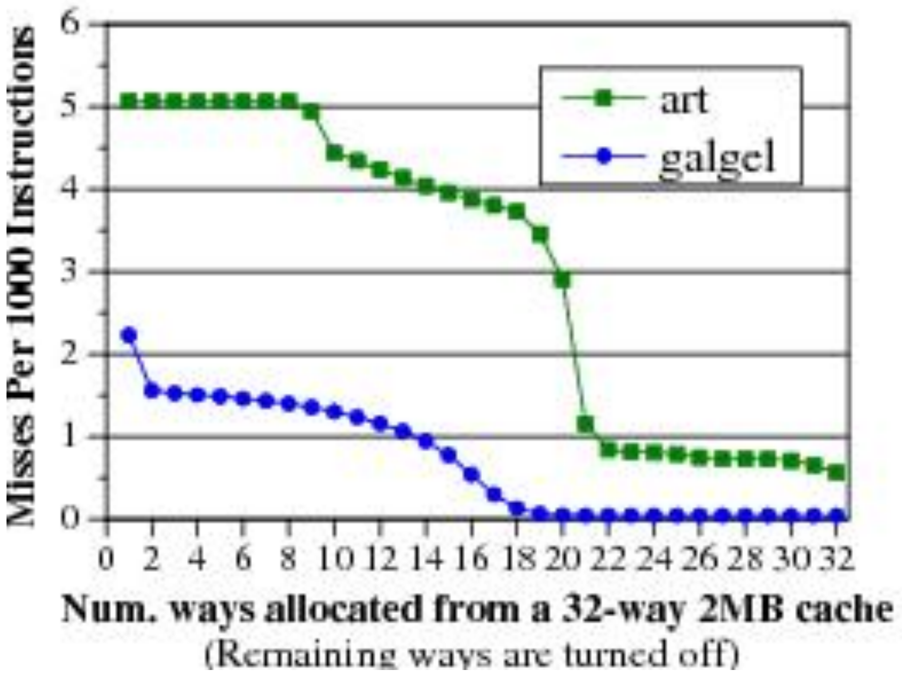
- Problem NP hard → need scalable partitioning algorithm

Greedy Algorithm [Stone+ ToC ' 92]

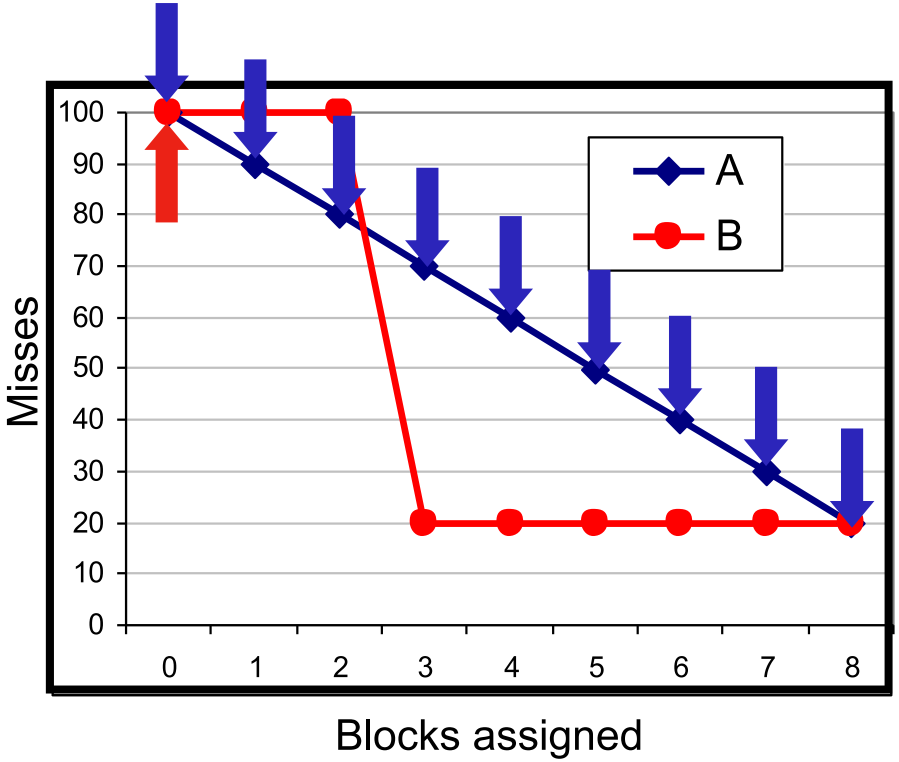
- GA allocates 1 block to the app that has the max utility for one block. Repeat till all blocks allocated

- Optimal partitioning when utility curves are convex

- Pathological behavior for non-convex curves



Problem with Greedy Algorithm



In each iteration, the utility for 1 block:

U(A) = 10 misses
U(B) = 0 misses

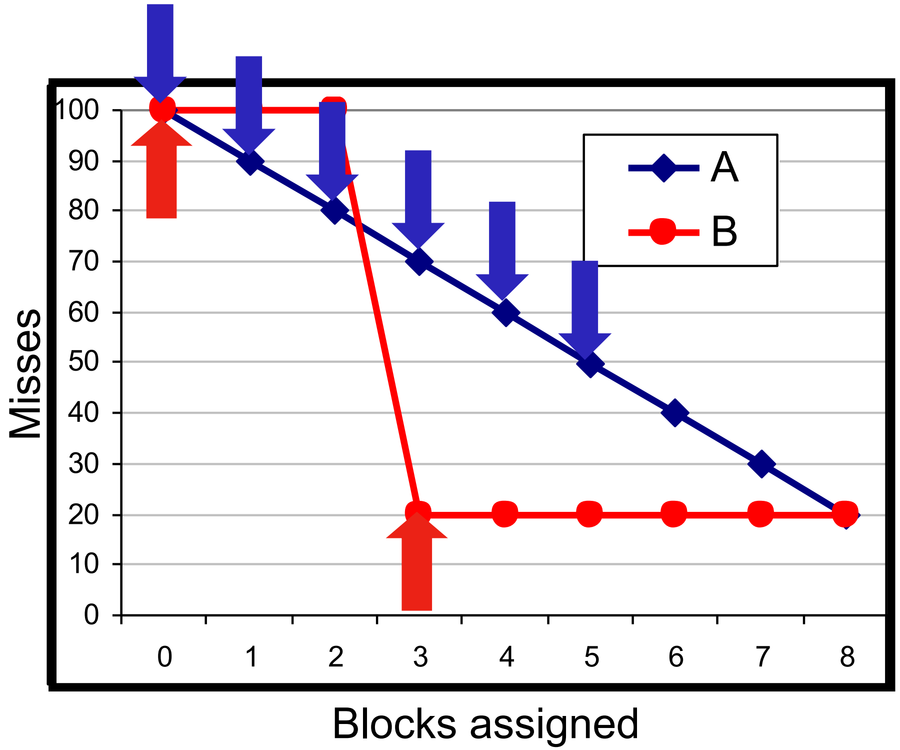
All blocks assigned to A, even if B has same miss reduction with fewer blocks

- Problem: GA considers benefit only from the immediate block. Hence, it fails to exploit large gains from looking ahead

Lookahead Algorithm

- Marginal Utility (MU) = Utility per cache resource
  - $MU_a^b = U_a^b/(b-a)$
- GA considers MU for 1 block. LA considers MU for all possible allocations
- Select the app that has the max value for MU. Allocate it as many blocks required to get max MU
- Repeat till all blocks assigned

Lookahead Algorithm Example



Iteration 1:

MU(A) = 10/1 block

MU(B) = 80/3 blocks

B gets 3 blocks

Next five iterations:

MU(A) = 10/1 block

MU(B) = 0

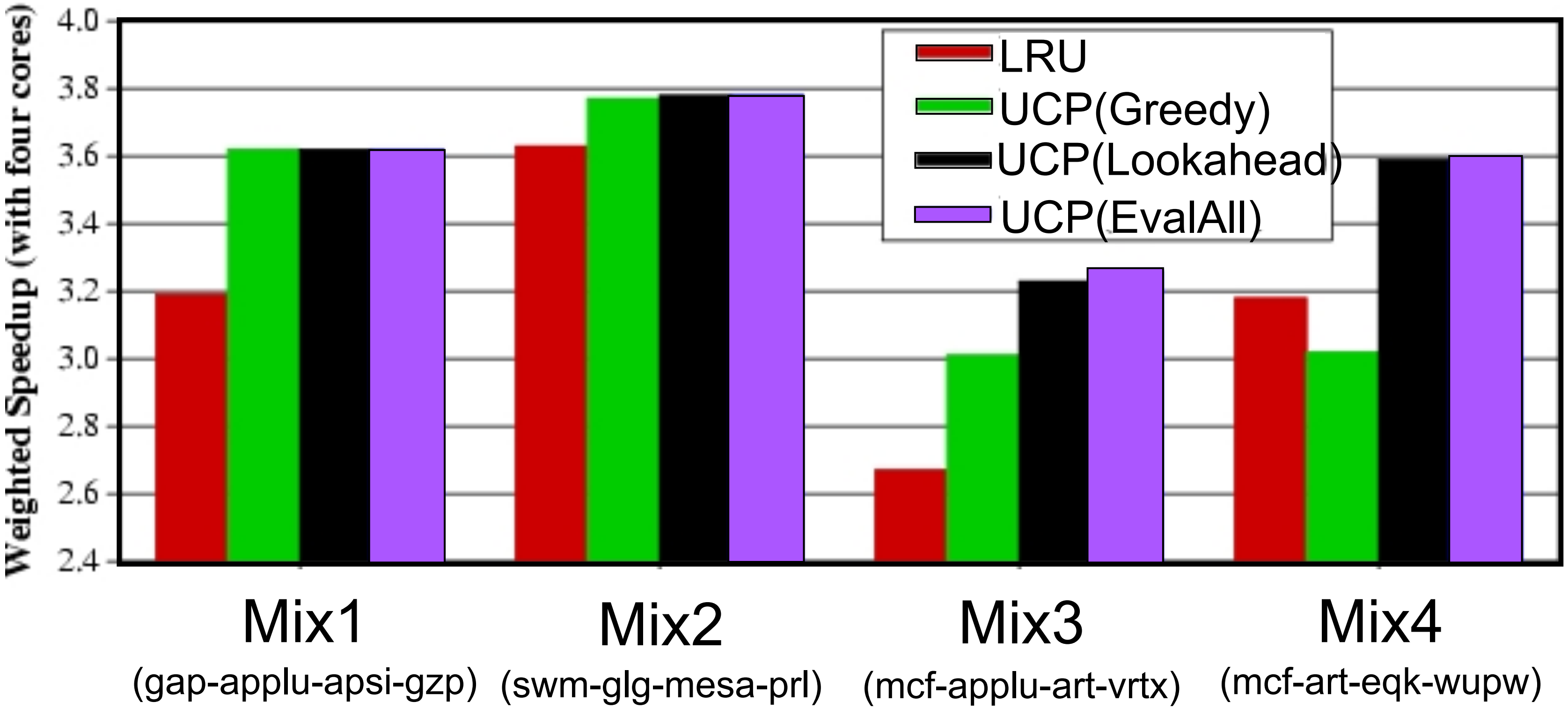
A gets 1 block

Result: A gets 5 blocks and B gets 3 blocks (Optimal)

Time complexity ≈ $\text{ways}^2/2$ (512 ops for 32-ways)

UCP Results

Four cores sharing a 2MB 32-way L2



LA performs similar to EvalAll, with low time-complexity

Utility Based Cache Partitioning

- Advantages over LRU

  + Improves system throughput

  + Better utilizes the shared cache

- Disadvantages

  - Fairness, QoS?

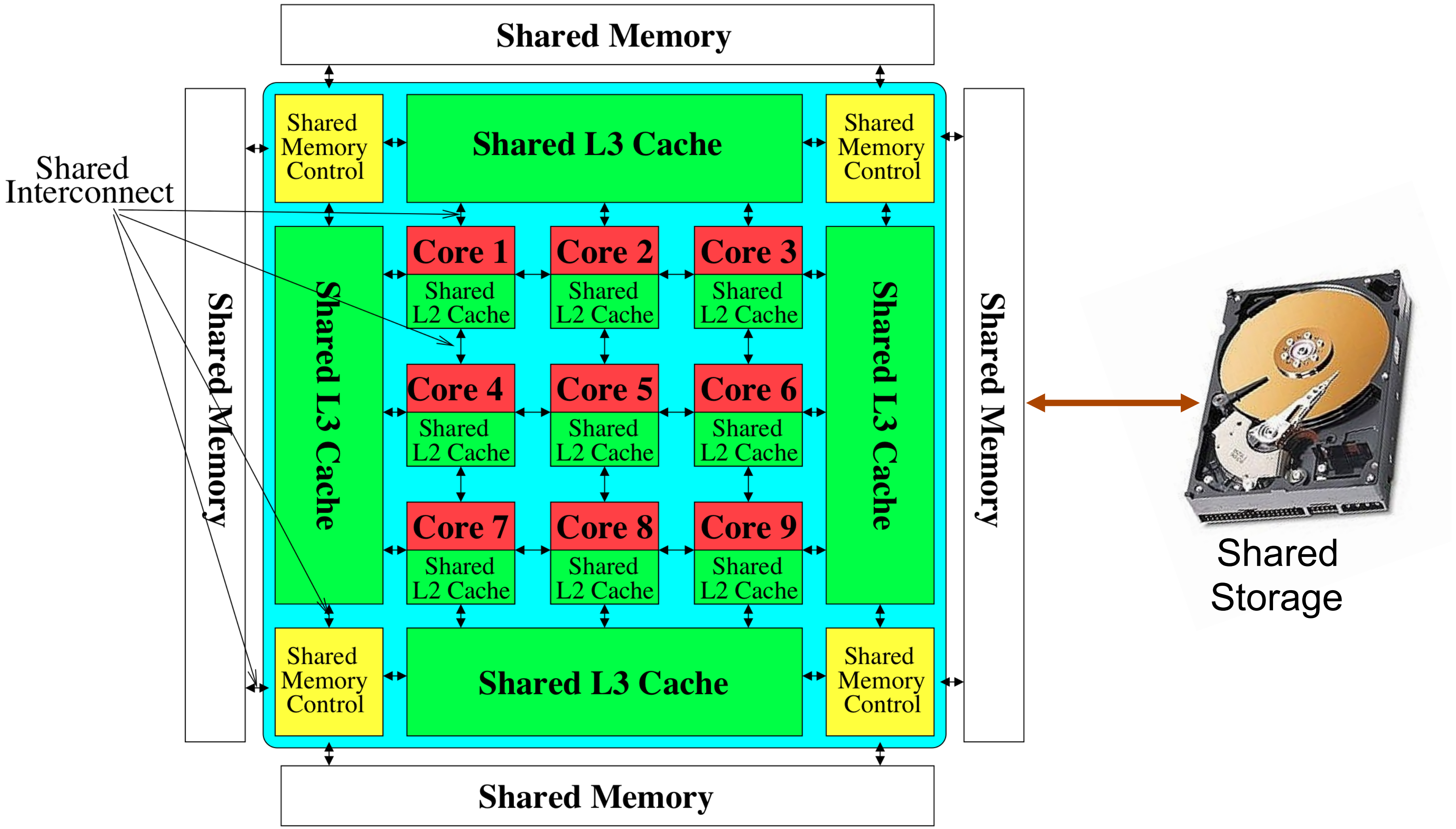
- Limitations

  - Scalability: Partitioning limited to ways. What if you have numWays < numApps?

  - Scalability: How is utility computed in a distributed cache?

  - What if past behavior is not a good predictor of utility?

The Multi-Core System: A *Shared Resource* View



Need for QoS and Shared Resource Mgmt.

- Why is unpredictable performance (or lack of QoS) bad?

- Makes programmer's life difficult
  - An optimized program can get low performance (and performance varies widely depending on co-runners)

- Causes discomfort to user
  - An important program can starve
  - Examples from shared software resources

- Makes system management difficult
  - How do we enforce a Service Level Agreement when hardware resources are sharing is uncontrollable?

Resource Sharing vs. Partitioning

- Sharing improves throughput
  - Better utilization of space

- Partitioning provides performance isolation (predictable performance)
  - Dedicated space

- Can we get the benefits of both?

- Idea: Design shared resources such that they are efficiently utilized, controllable and partitionable
  - No wasted resource + QoS mechanisms for threads

Shared Hardware Resources

- Memory subsystem (in both multithreaded and multi-core systems)
  - Non-private caches
  - Interconnects
  - Memory controllers, buses, banks

- I/O subsystem (in both multithreaded and multi-core systems)
  - I/O, DMA controllers
  - Ethernet controllers

- Processor (in multithreaded systems)
  - Pipeline resources
  - L1 caches

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.

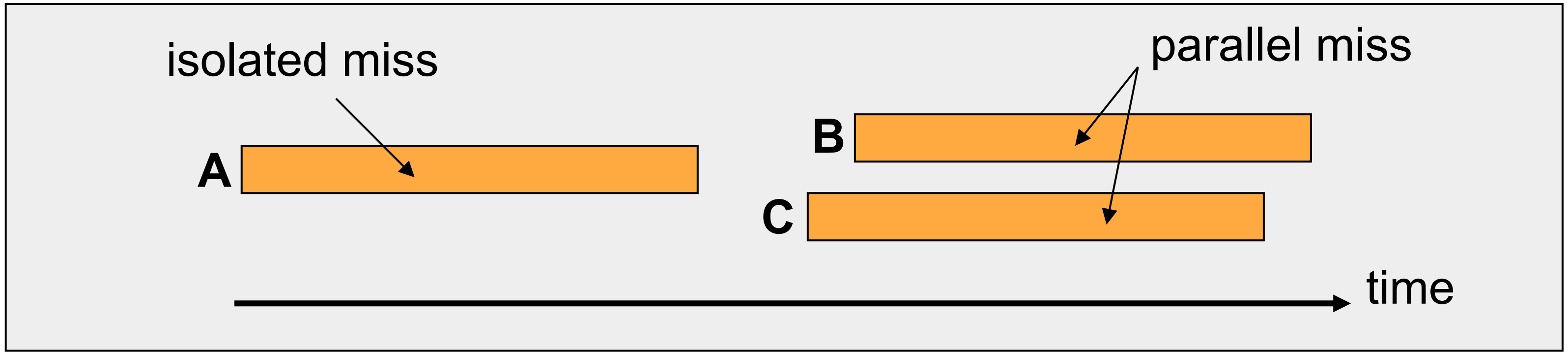
- Seshadri et al., "The Evicted-Address Filter: A Unified Mechanism to Address both Cache Pollution and Thrashing," PACT 2012.

- 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

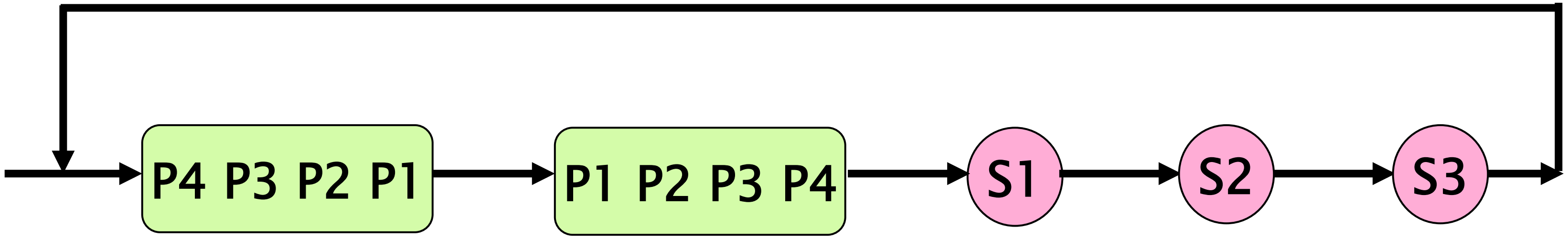
- Traditional cache replacement policies try to reduce miss count

- 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



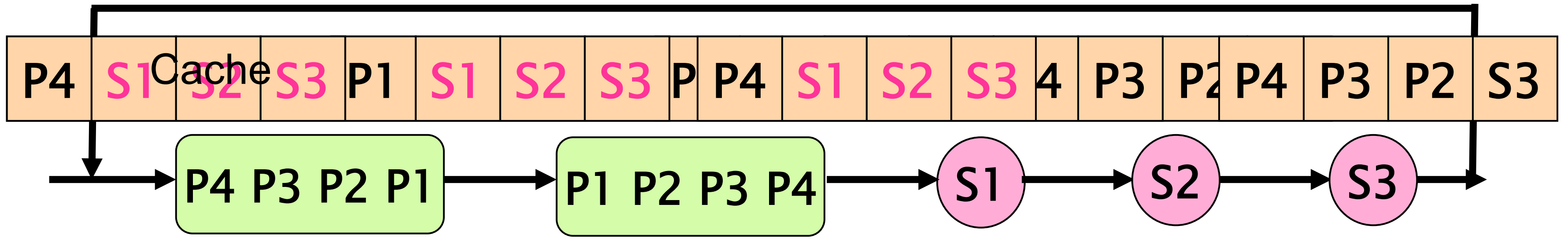
Misses to blocks P1, P2, P3, P4 can be parallel
Misses to blocks S1, S2, and S3 are isolated

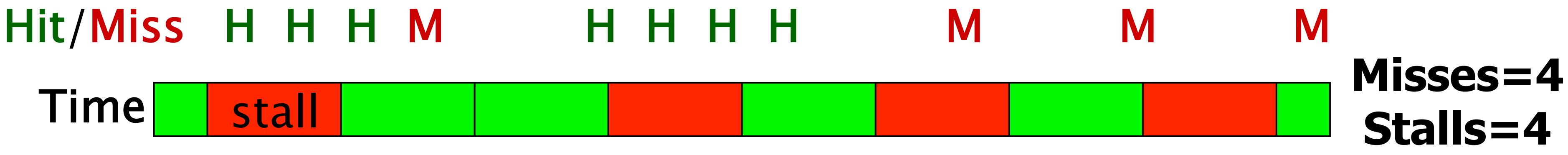
Two replacement algorithms:

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

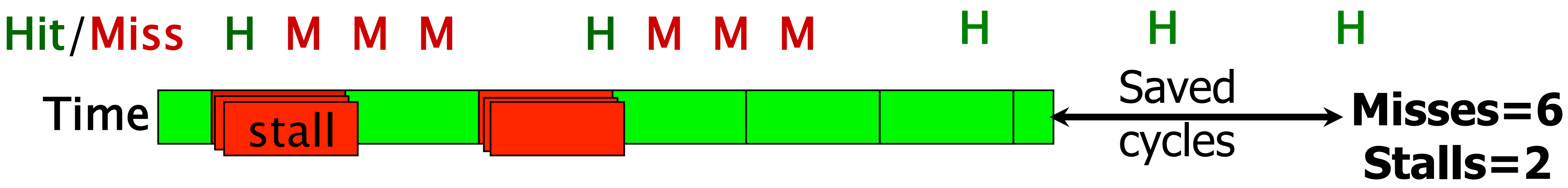
For a fully associative cache containing 4 blocks

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

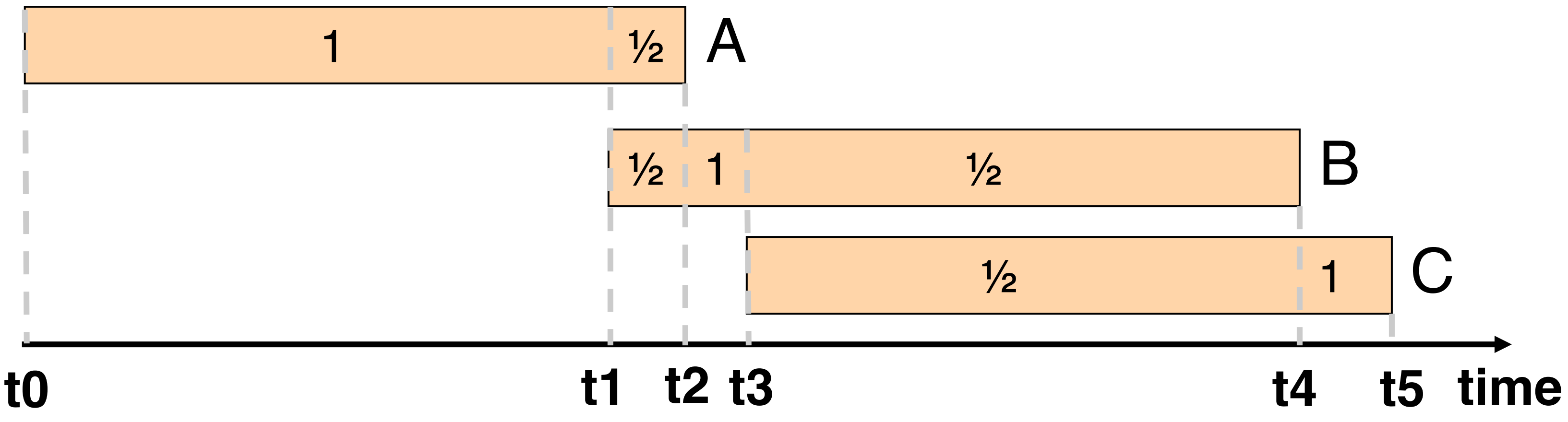
- ❑ Summary

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

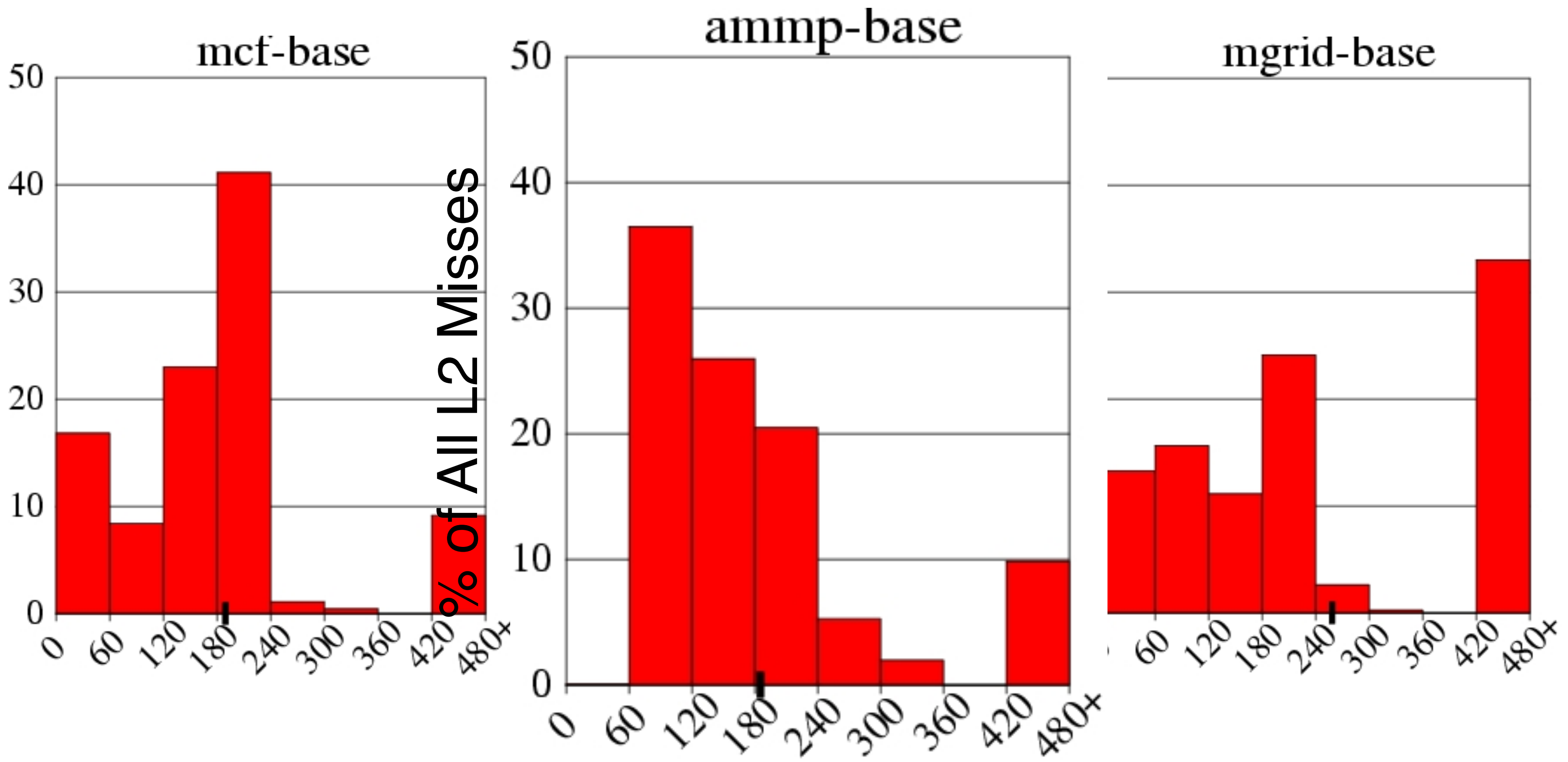
$$\boxed{\textbf{mlp-cost += (1/N)}}$$

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 ?

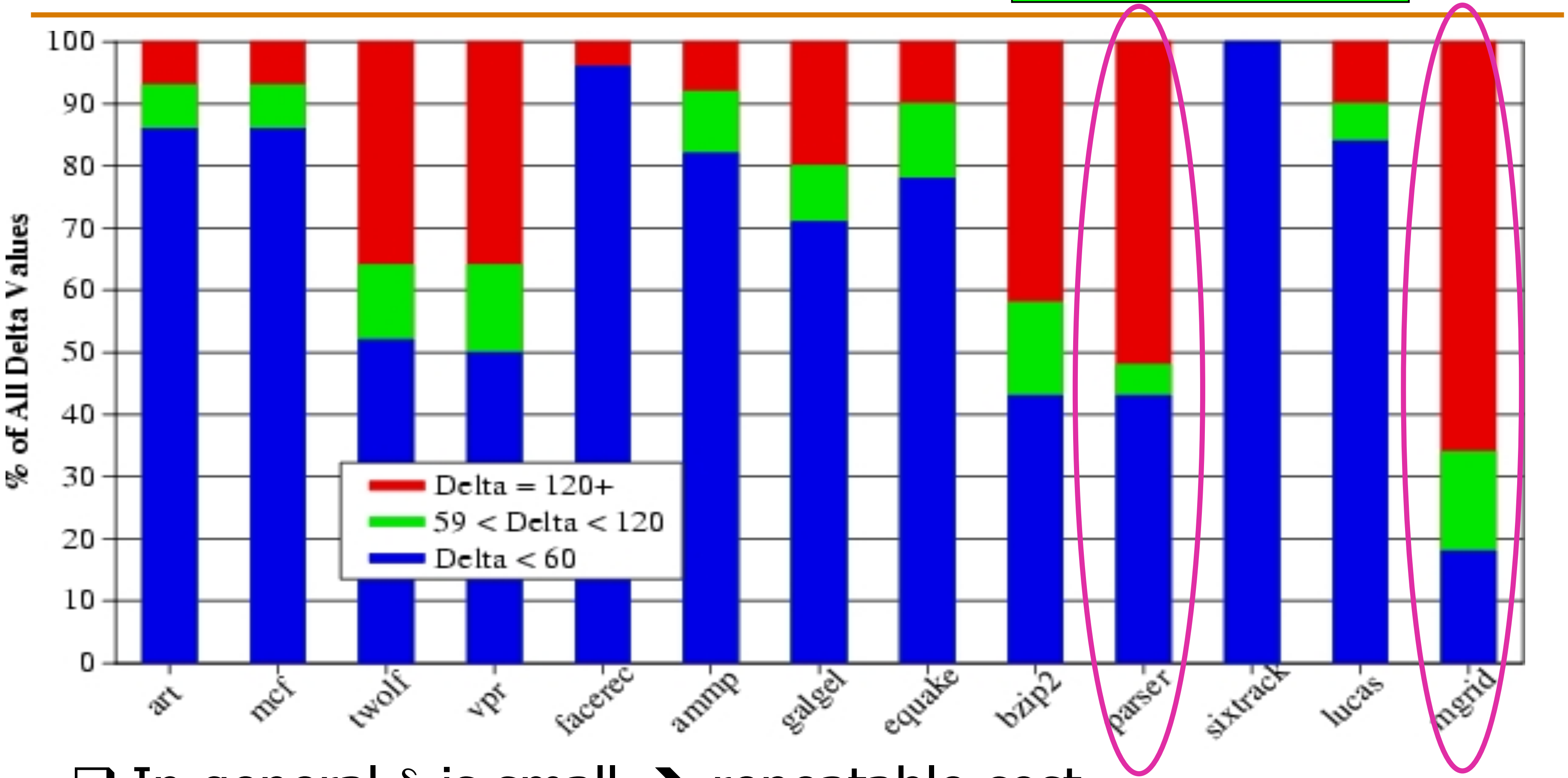
- ❑ Let δ = difference in cost for successive miss to a block
  - ▪ Small δ ➔ cost repeats
  - ▪ Large δ ➔ cost varies significantly

Repeatability of Cost

δ < 60

δ > 120

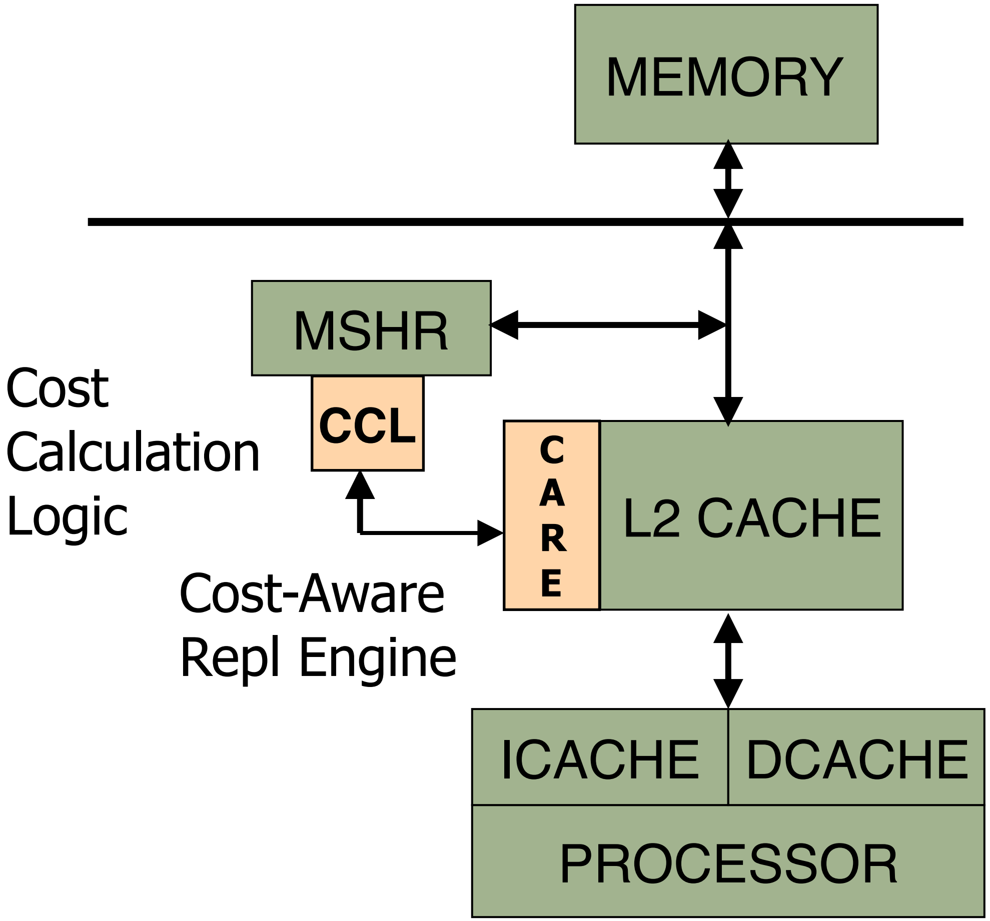
59 < δ < 120



❑ 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

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

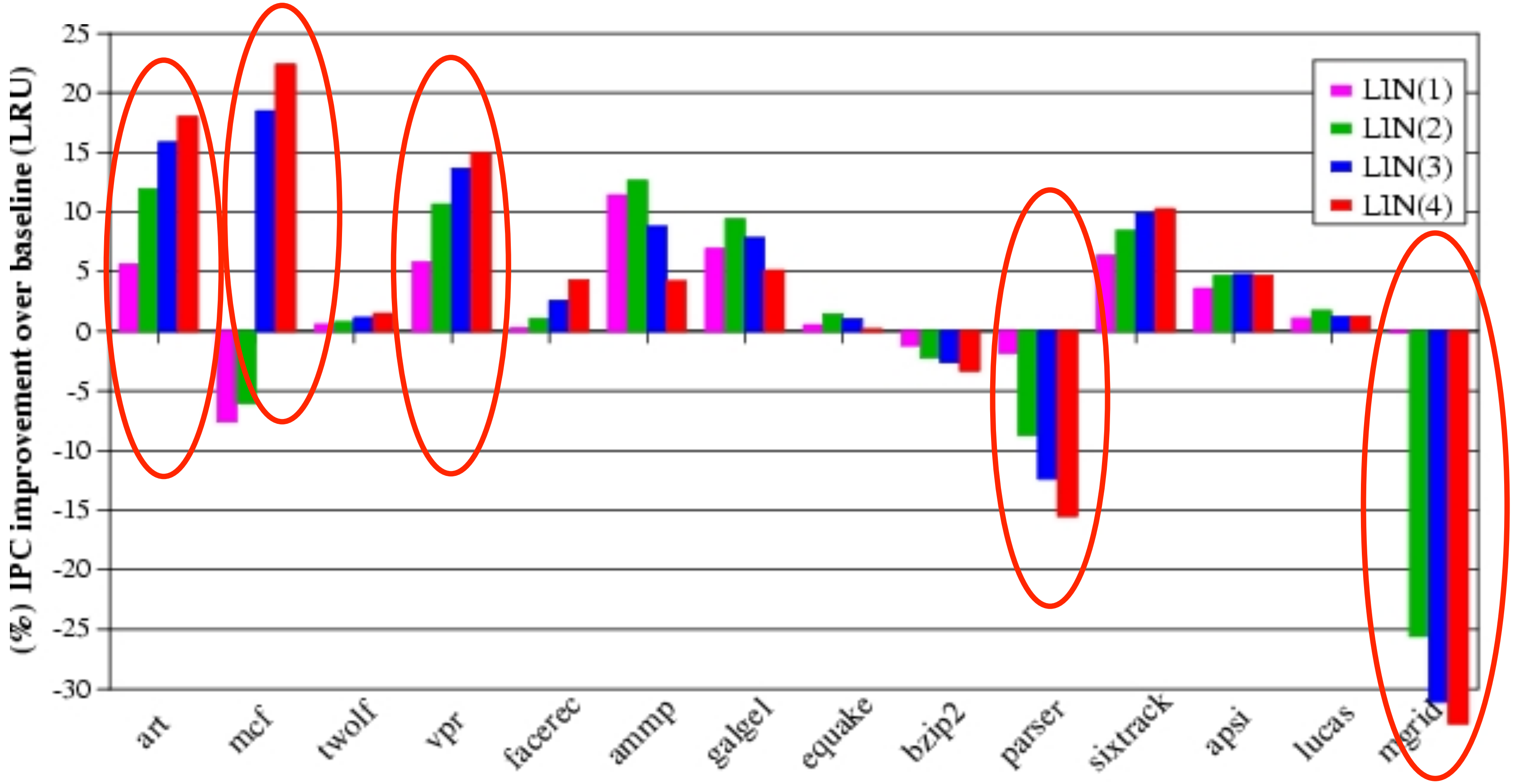
- Decisions based only on cost and no recency hurt performance. Cache stores useless high cost blocks

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

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

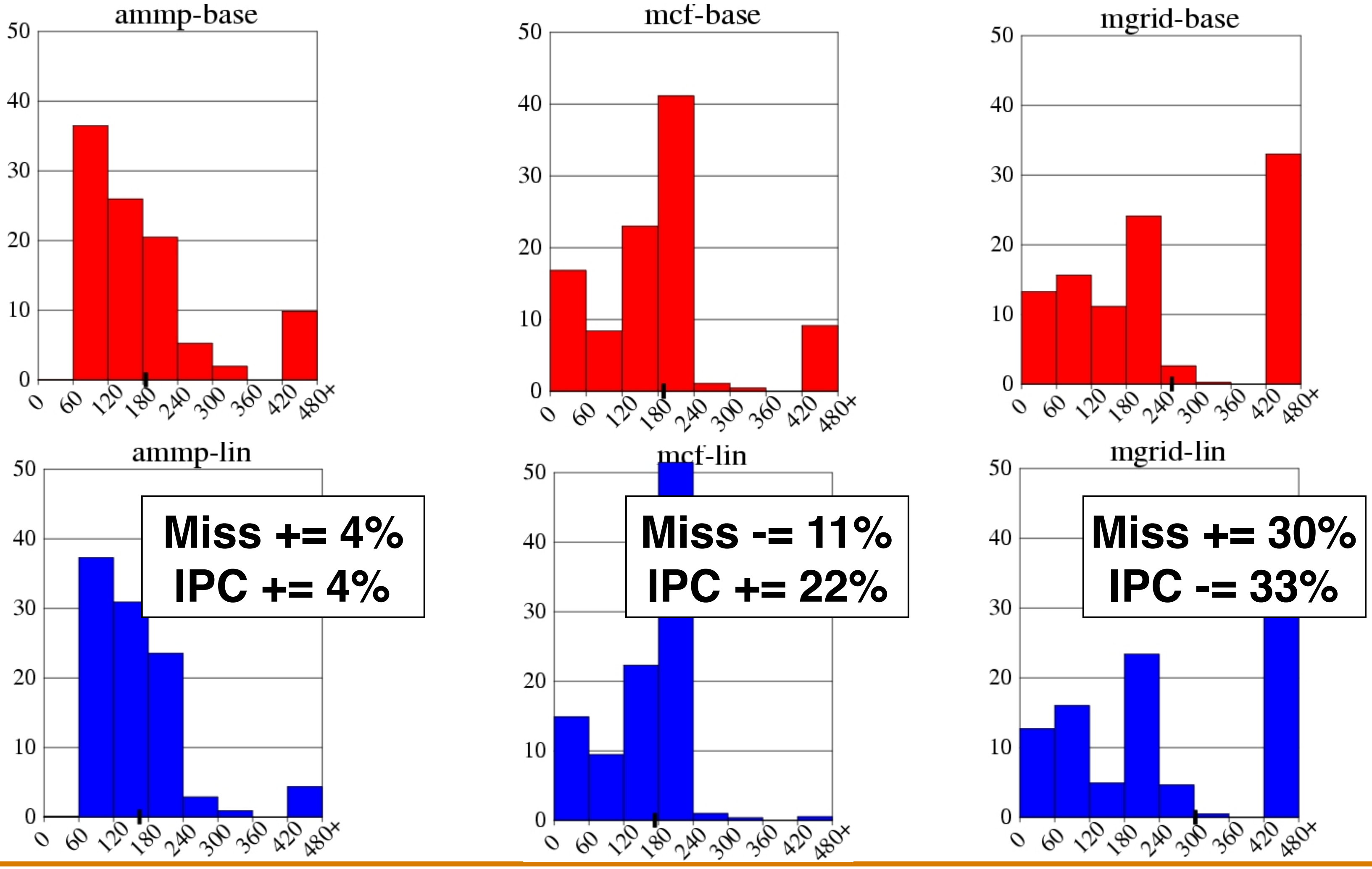
S = significance of cost. Recency(i) = position in LRU stack
cost(i) = quantized cost

Results for the LIN policy



Performance loss for parser and mgrid due to large δ

Effect of LIN policy on Cost



Miss += 4%
IPC += 4%

Miss -= 11%
IPC += 22%

Miss += 30%
IPC -= 33%

Outline

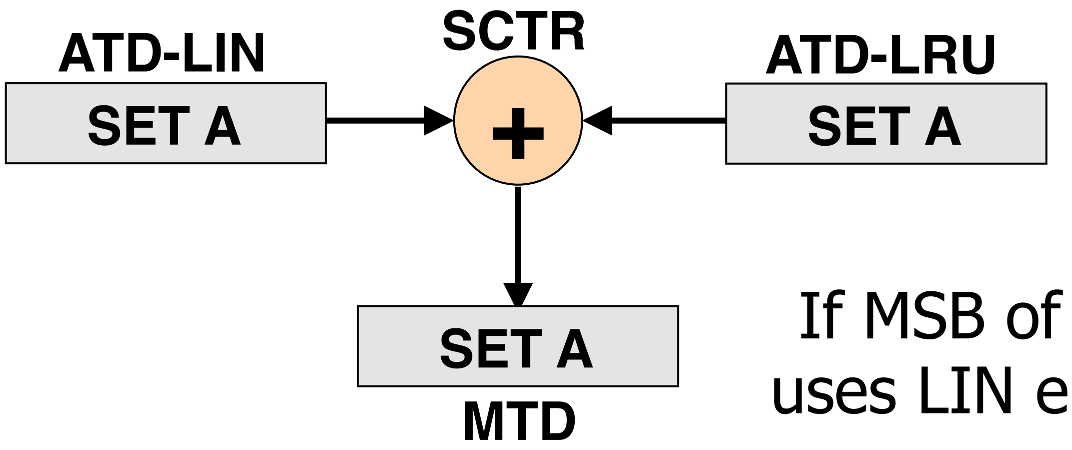
- ❑ Introduction

- ❑ 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



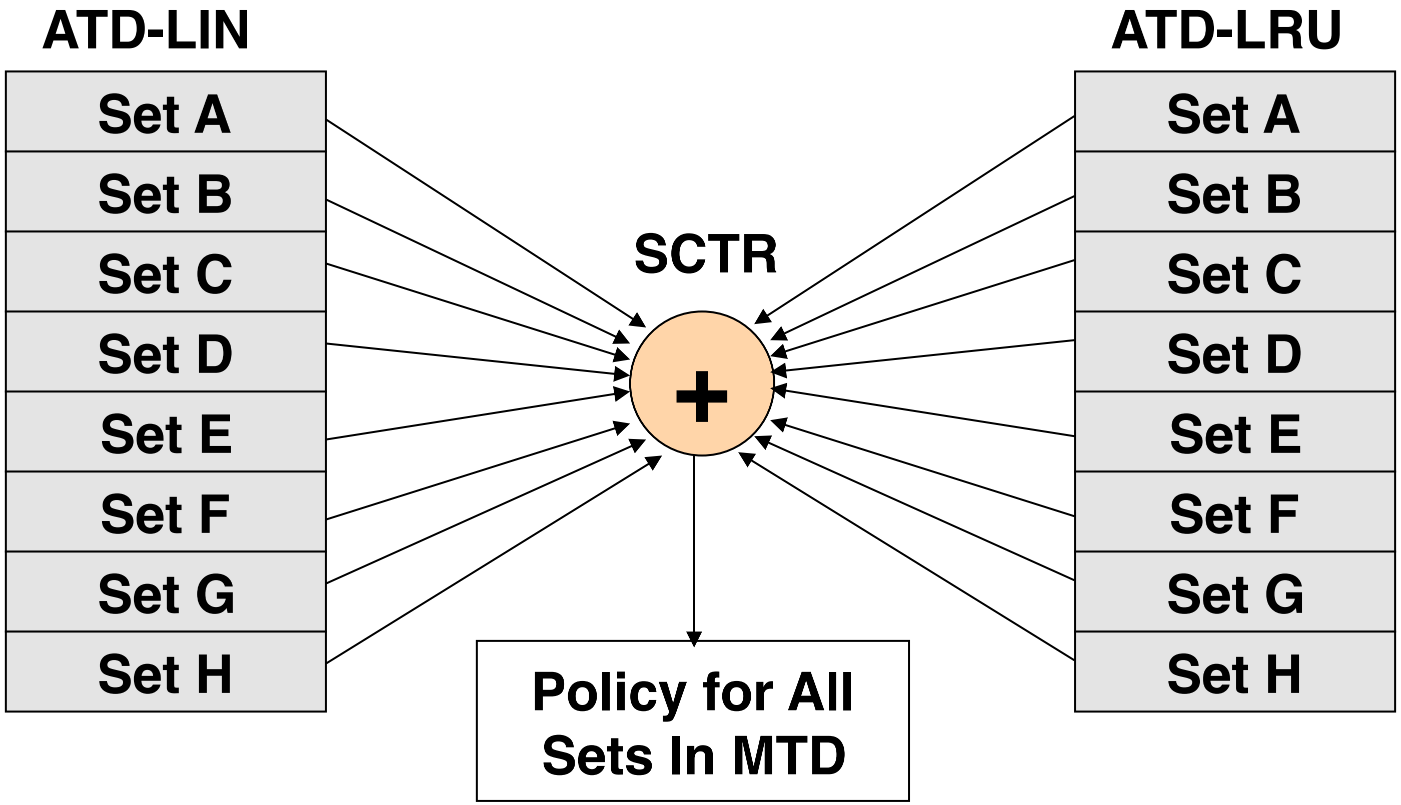
If MSB of SCTR is 1, MTD uses LIN else MTD use LRU

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



ATD-LIN

Set A
Set B
Set C
Set D
Set E
Set F
Set G
Set H

SCTR

+

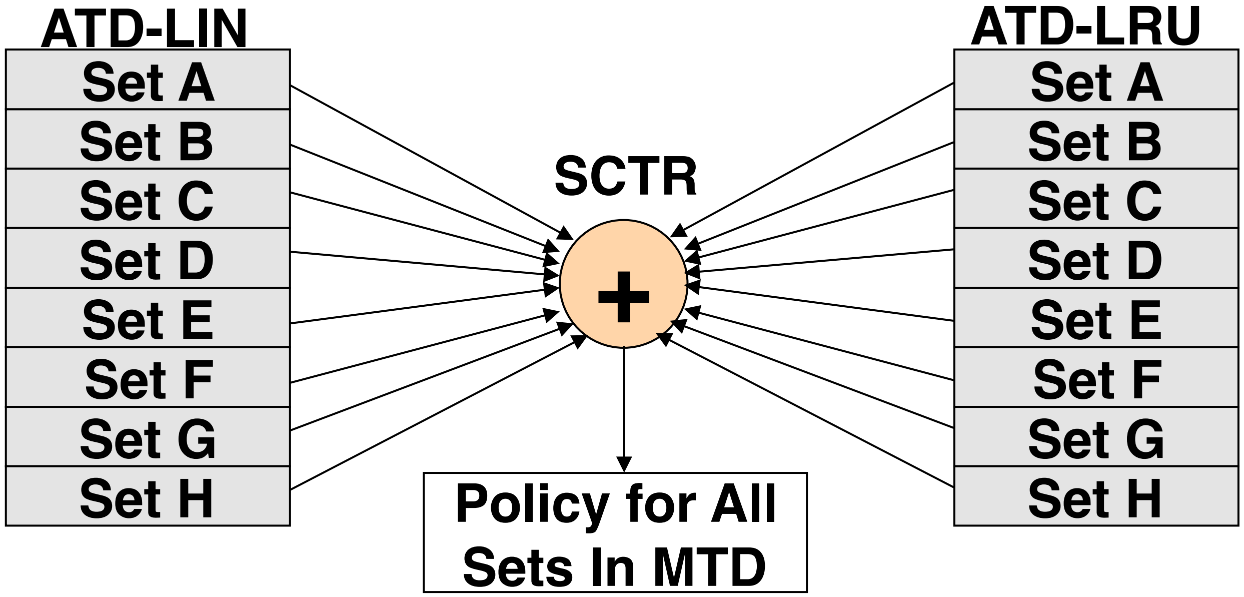
Policy for All Sets In MTD

ATD-LRU

Set A
Set B
Set C
Set D
Set E
Set F
Set G
Set H

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?

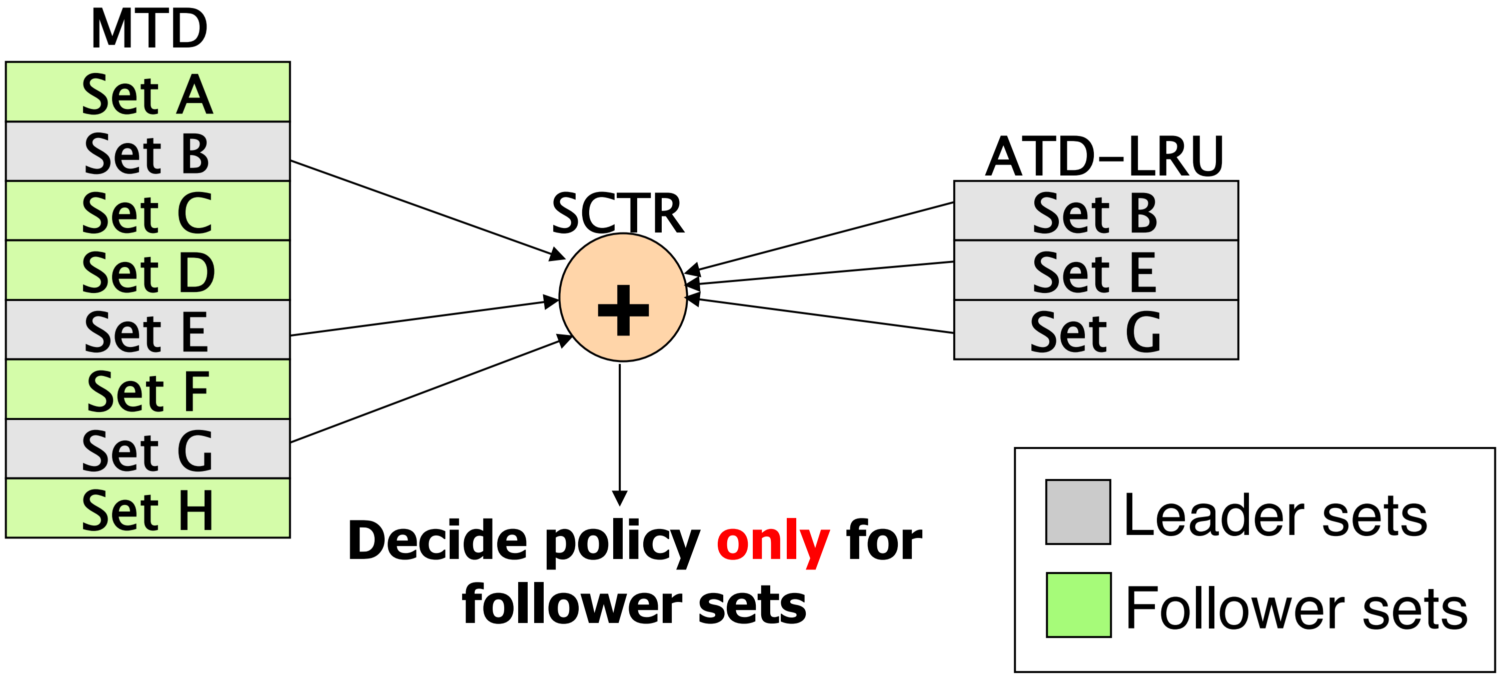
❑ Bounds using analytical model and simulation (in paper)

❑ 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

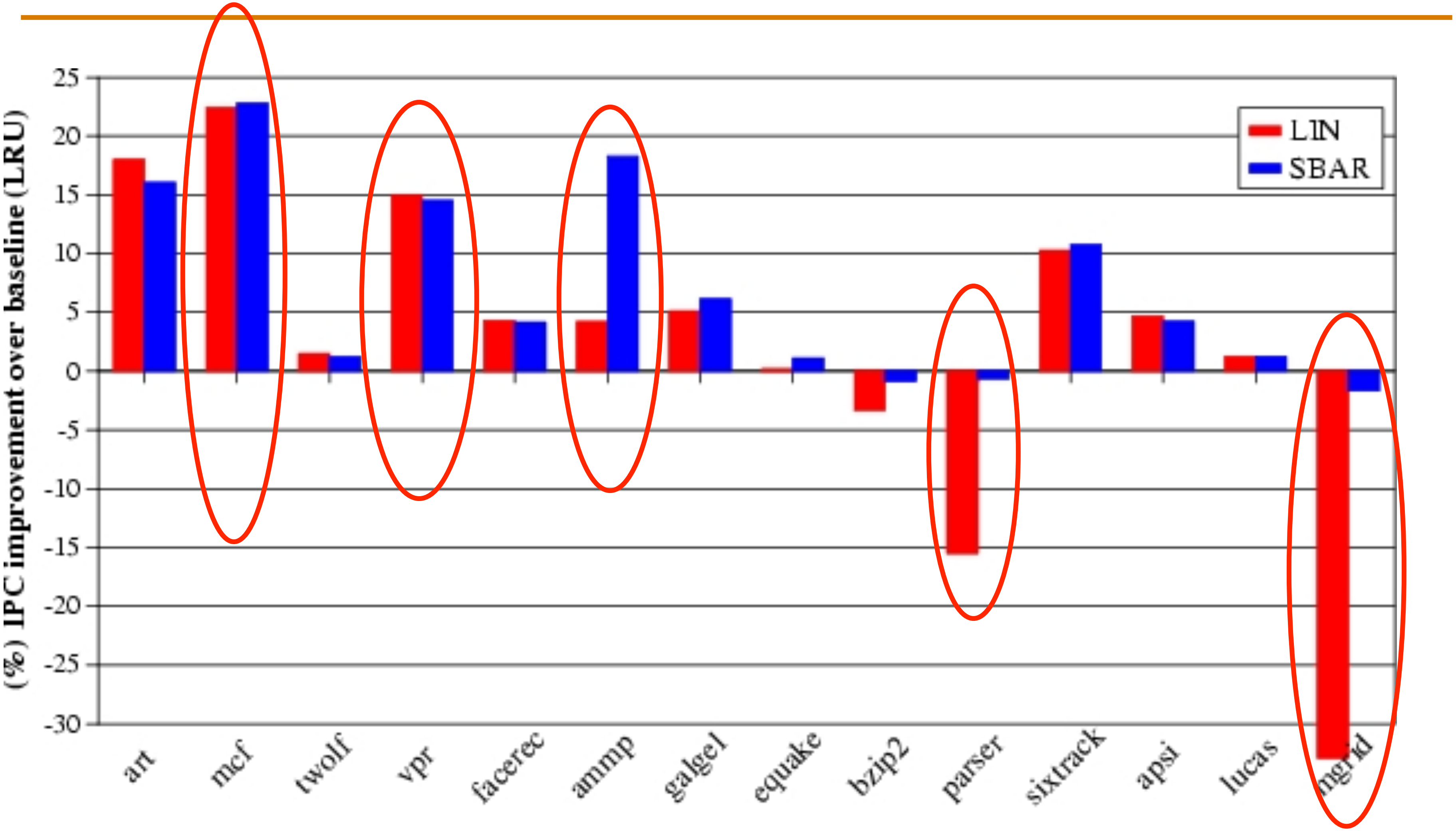
ATD overhead can further be reduced by using MTD to always simulate one of the policies (say LIN)

Sampling Based Adaptive Replacement (SBAR)

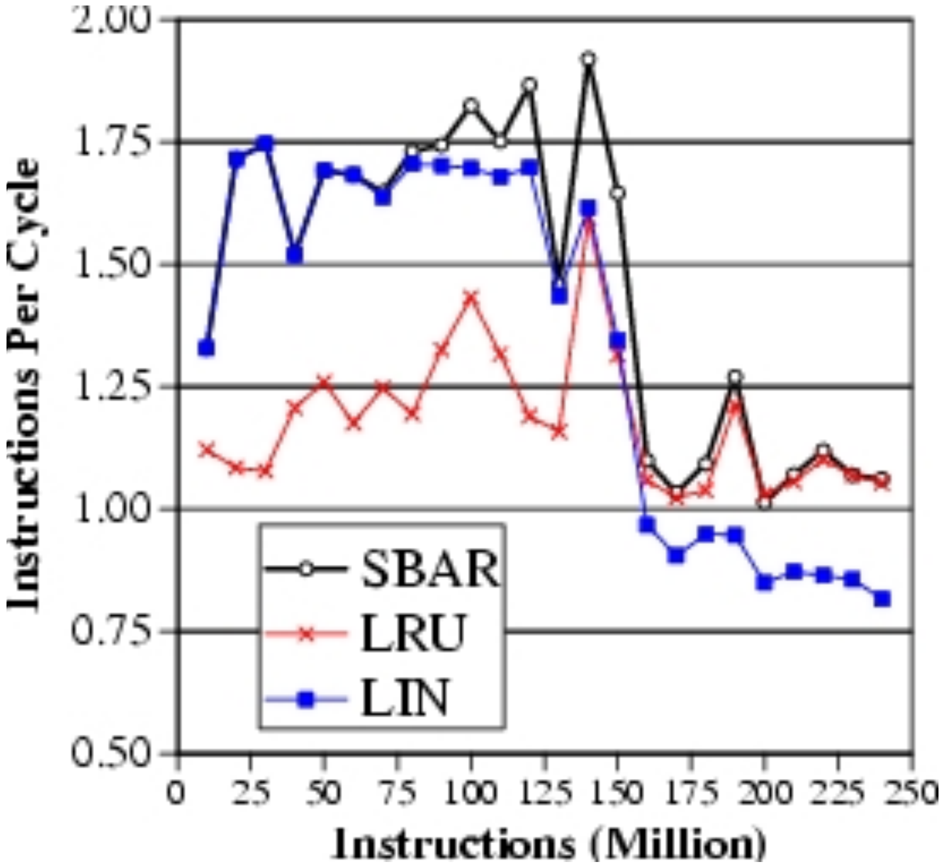


The storage overhead of SBAR is less than 2KB
(0.2% of the baseline 1MB cache)

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

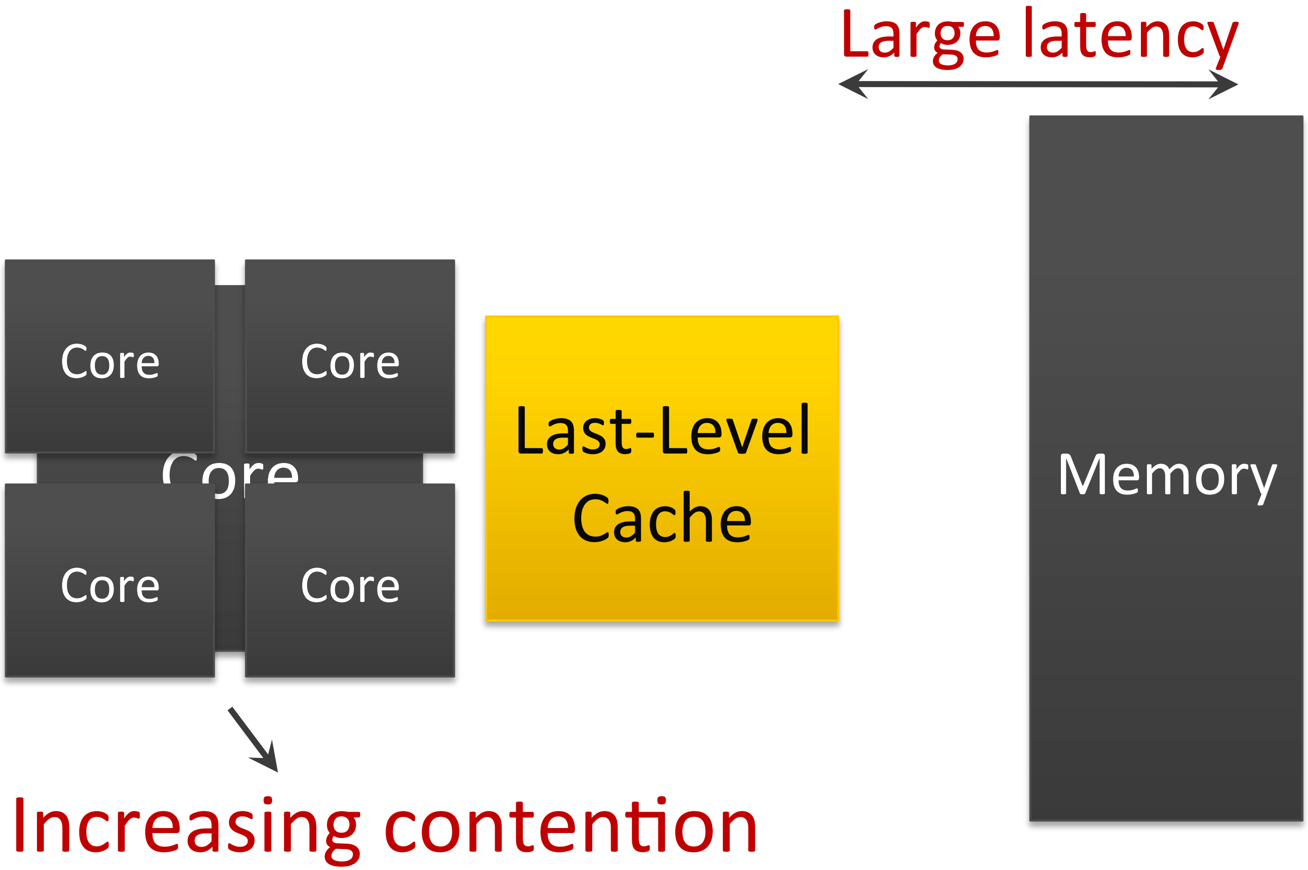
The Evicted-Address Filter

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

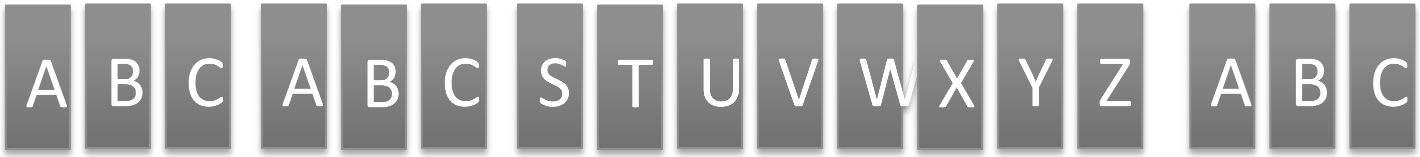


Effective cache utilization is important

Reuse Behavior of Cache Blocks

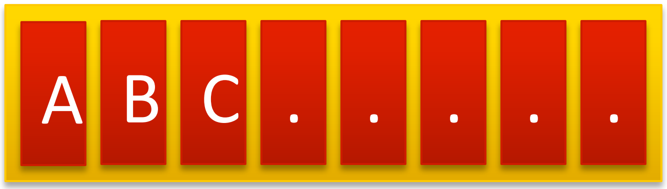
Different blocks have different reuse behavior

Access Sequence:



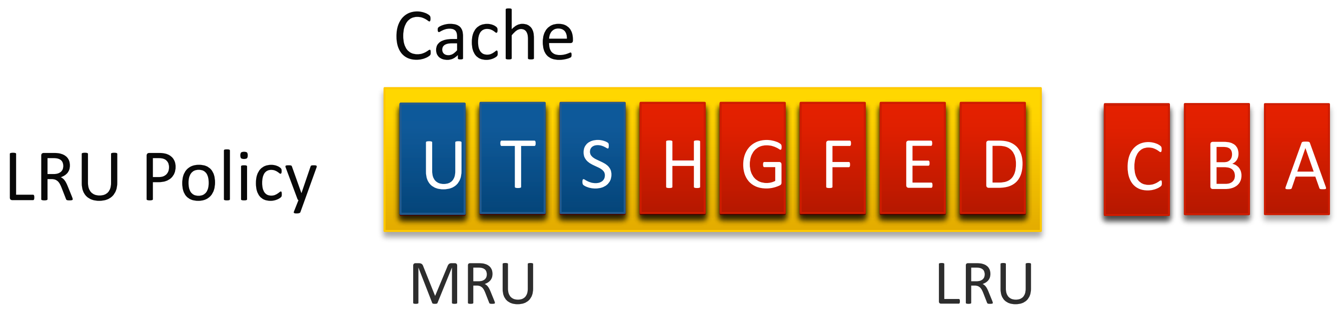
🟥 High-reuse block 🟦 Low-reuse block

Ideal Cache

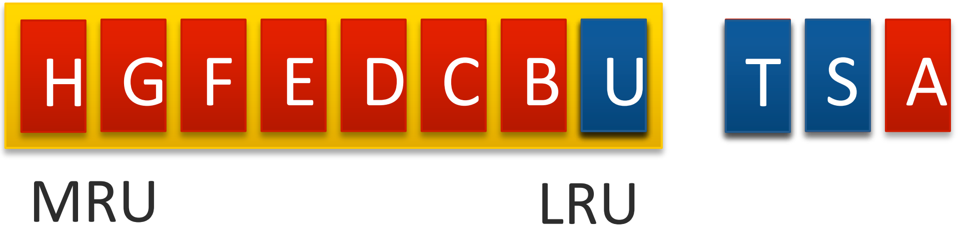


Cache Pollution

Problem: Low-reuse blocks evict high-reuse blocks

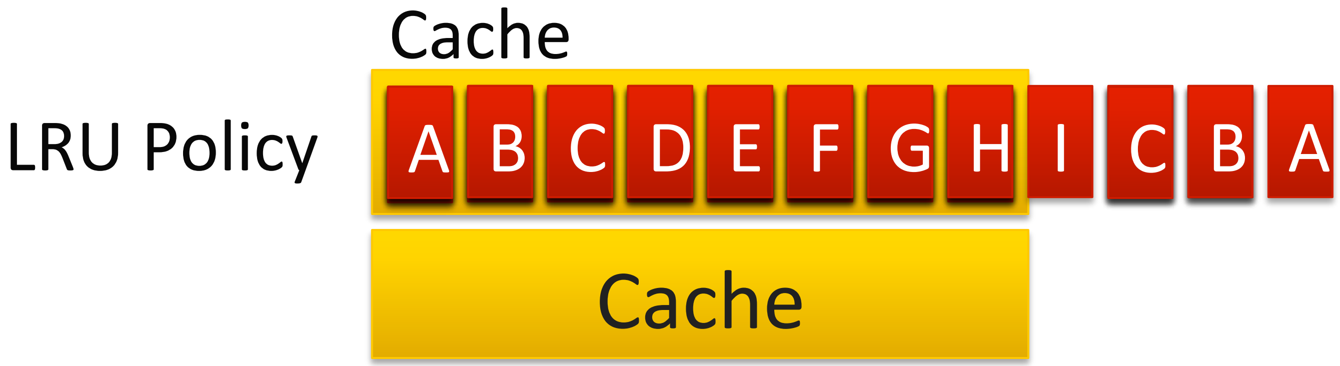


Prior work: Predict reuse behavior of missed blocks. Insert low-reuse blocks at LRU position.



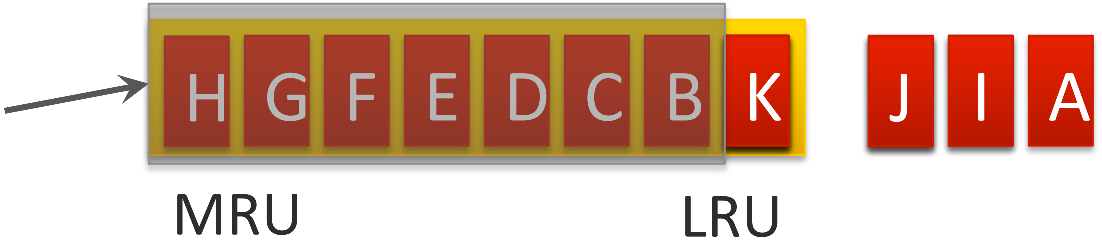
Cache Thrashing

Problem: High-reuse blocks evict each other



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

A fraction of working set stays in cache



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

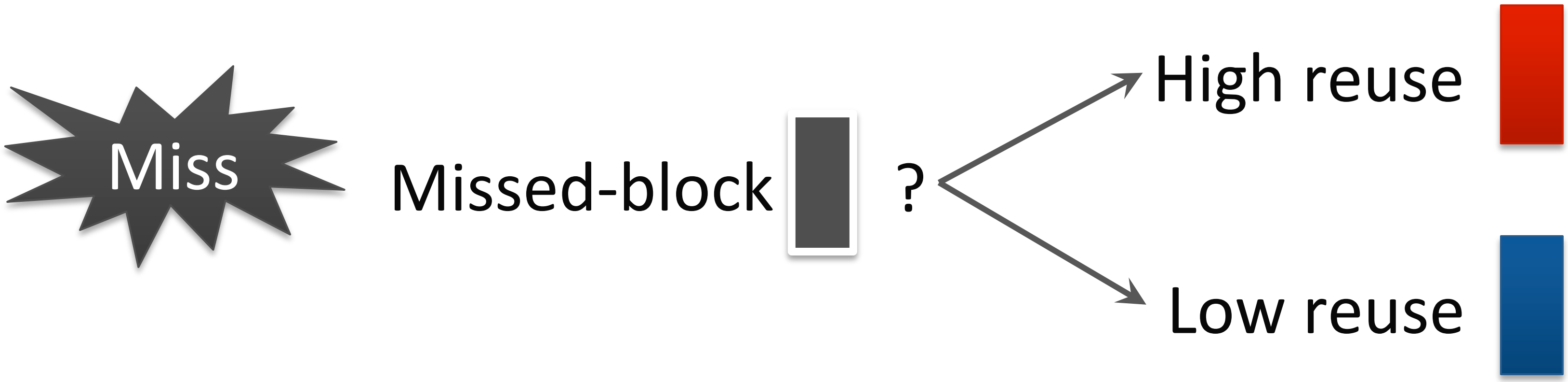
No mechanism to distinguish high-reuse blocks from low-reuse blocks

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



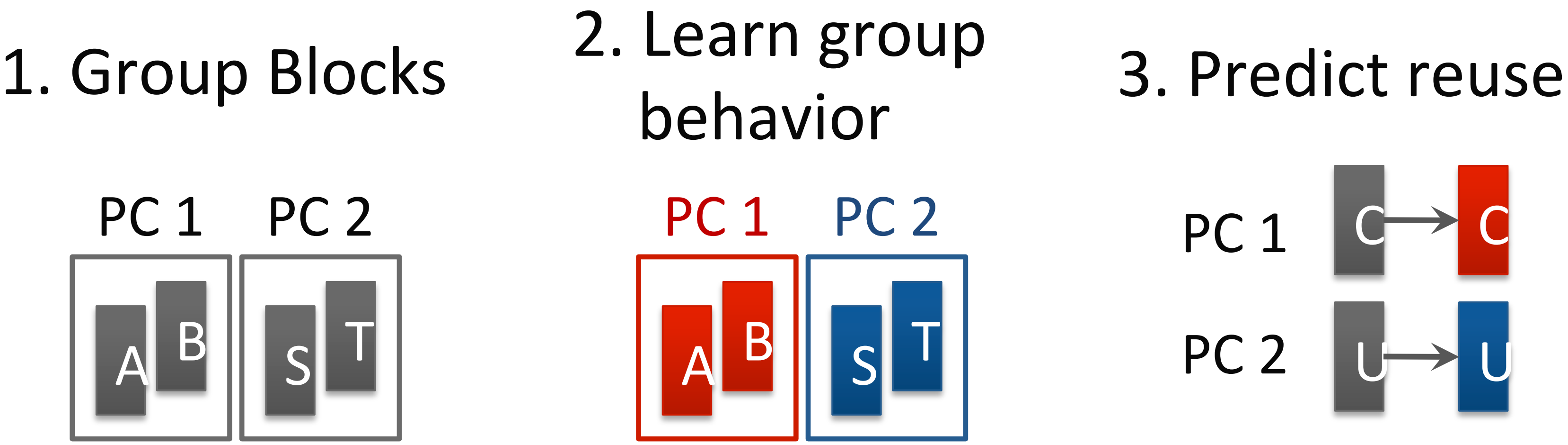
Keep track of the reuse behavior of every cache block in the system

Impractical

1. High storage overhead
2. Look-up latency

Prior Work on Reuse Prediction

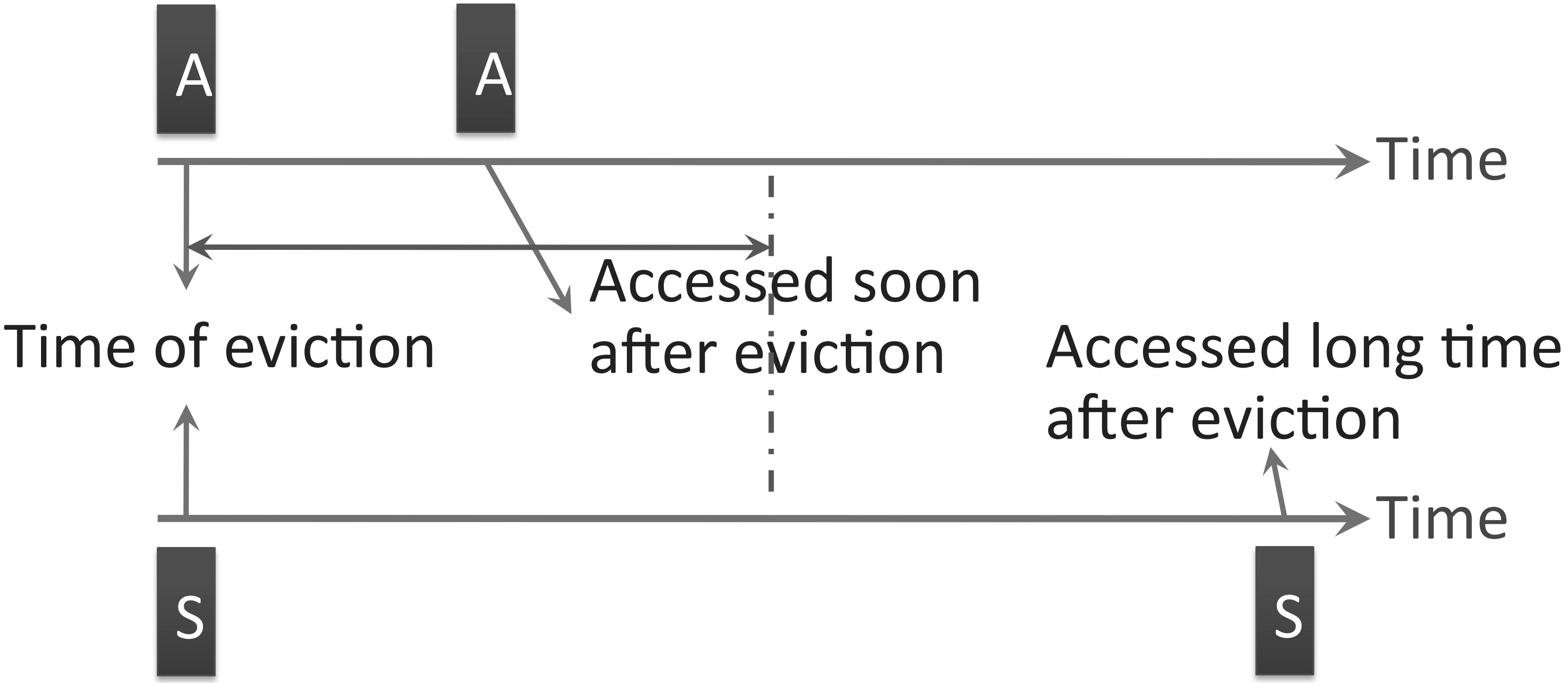
Use program counter or memory region information.



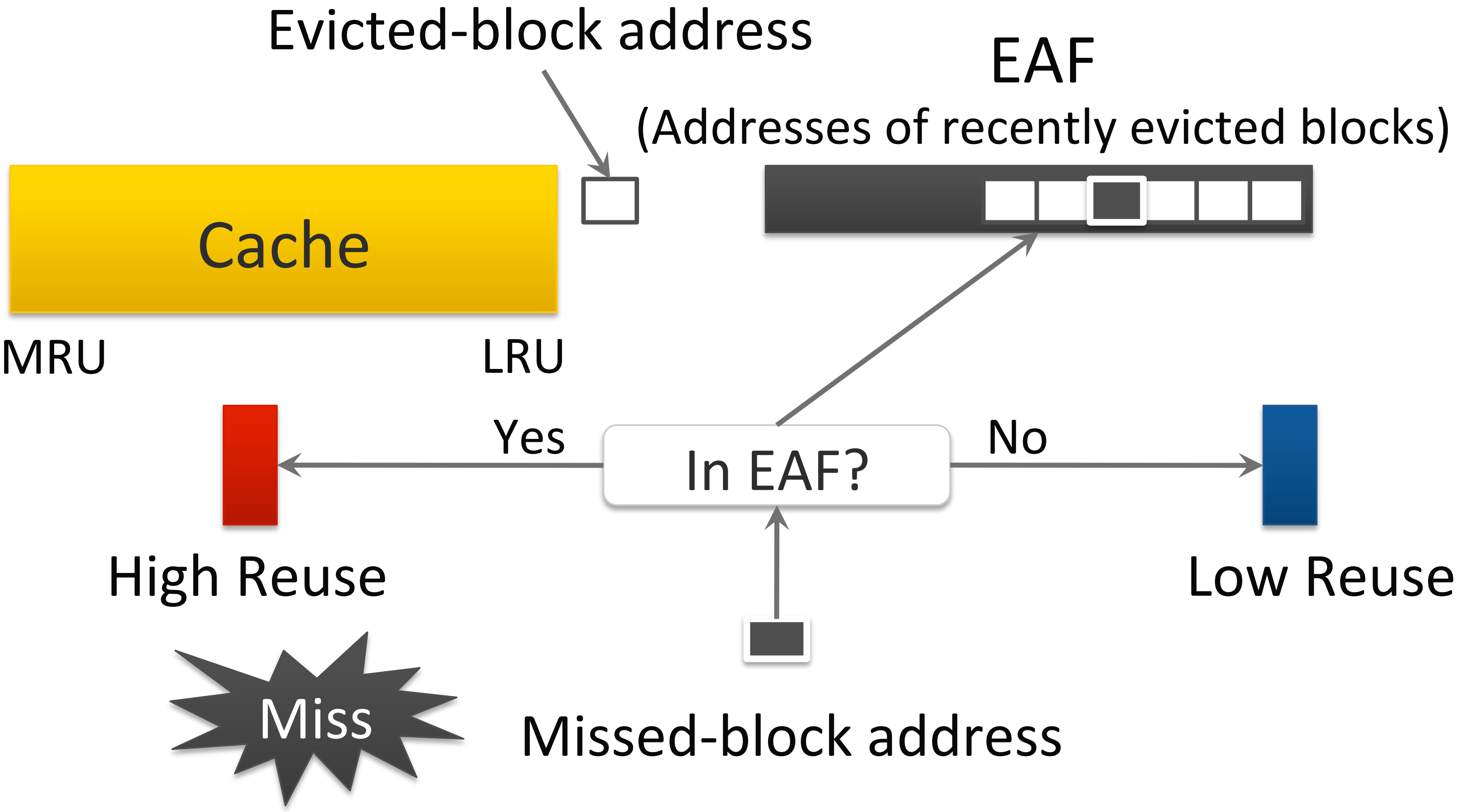
1. Same group $\not\rightarrow$ same reuse behavior
2. No control over number of high-reuse blocks

Our Approach: Per-block Prediction

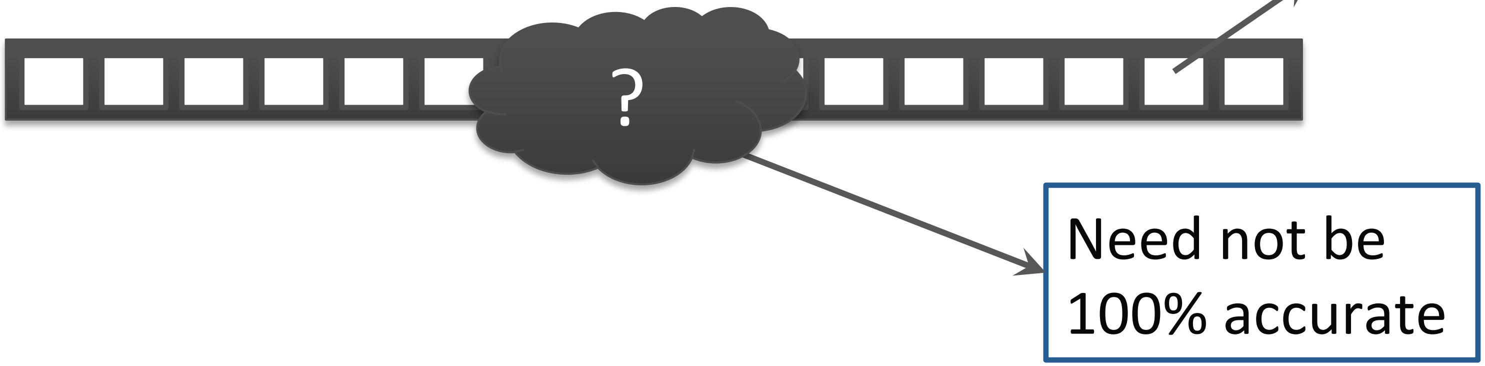
💡 Use recency of eviction to predict reuse


A
A
Time
Time of eviction
Accessed soon after eviction
Accessed long time after eviction
Time
S
S

Evicted-Address Filter (EAF)



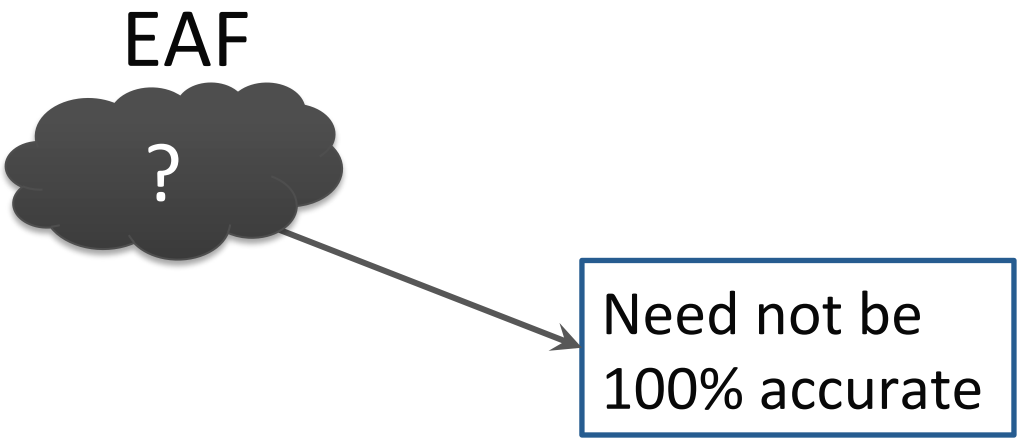
Naïve Implementation: Full Address Tags



Need not be 100% accurate

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

Low-Cost Implementation: Bloom Filter



EAF

?

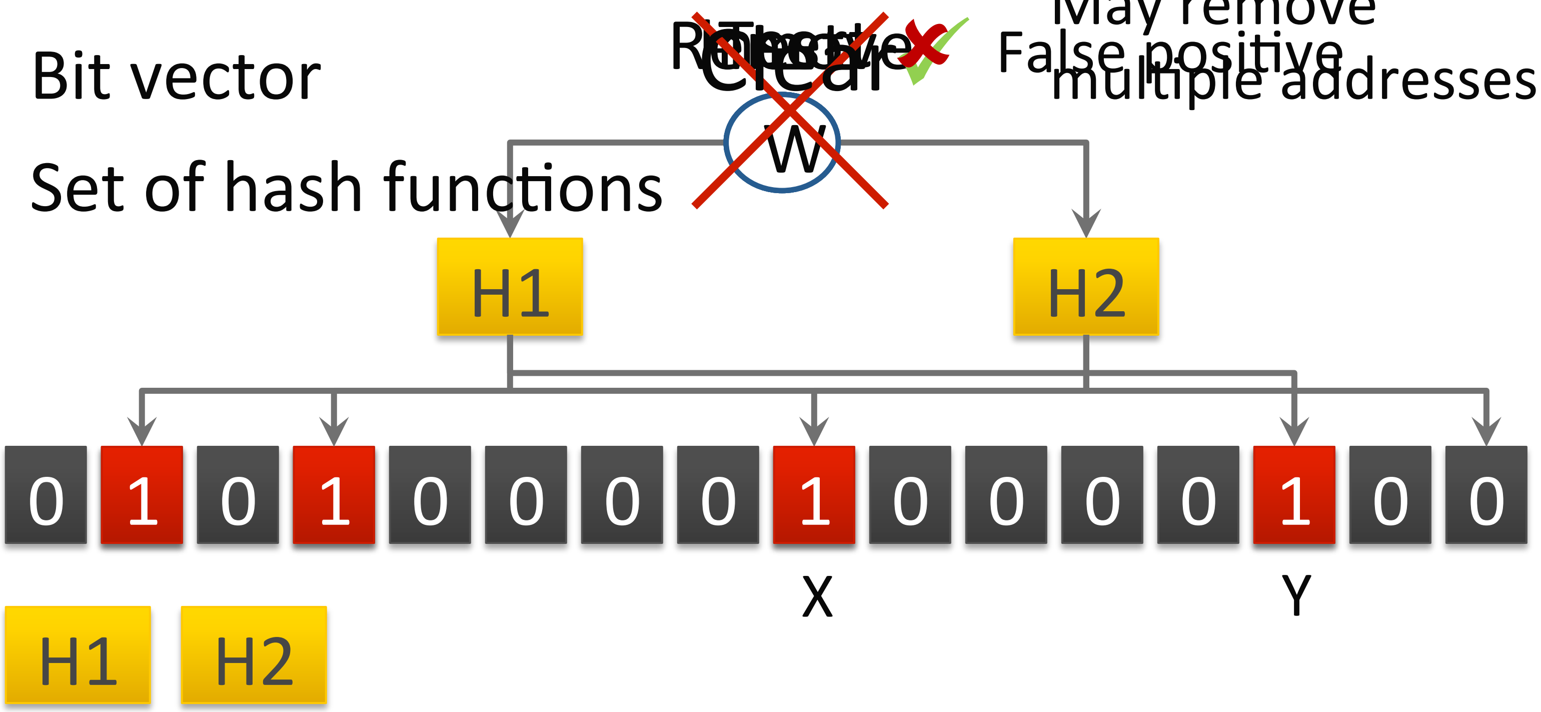
Need not be 100% accurate

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

Bloom Filter

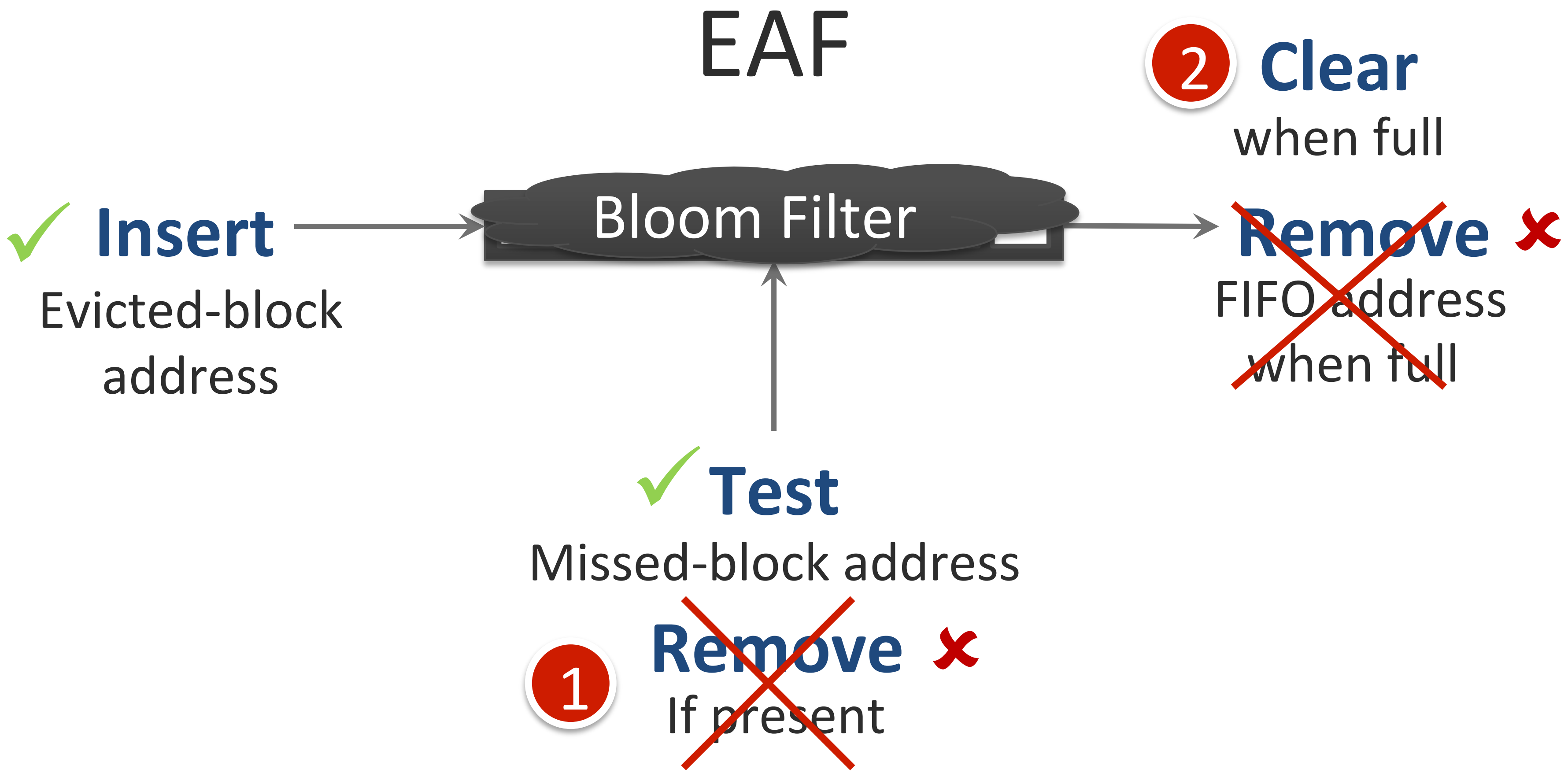
Compact representation of a set

1. Bit vector
2. Set of hash functions



Inserted Elements: X Y

EAF using a Bloom Filter



EAF

Bloom Filter

✓ **Insert**

Evicted-block address

✓ **Test**

Missed-block address

1 ~~**Remove**~~ ✗

~~If present~~

~~**Remove**~~ ✗

~~FIFO address when full~~

2 **Clear**

when full

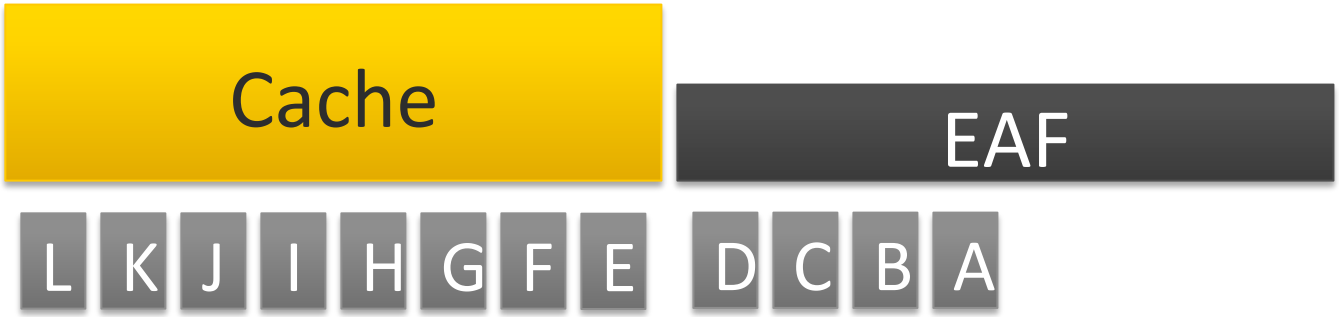
Bloom-filter EAF: 4x reduction in storage overhead, 1.47% compared to cache size

Outline

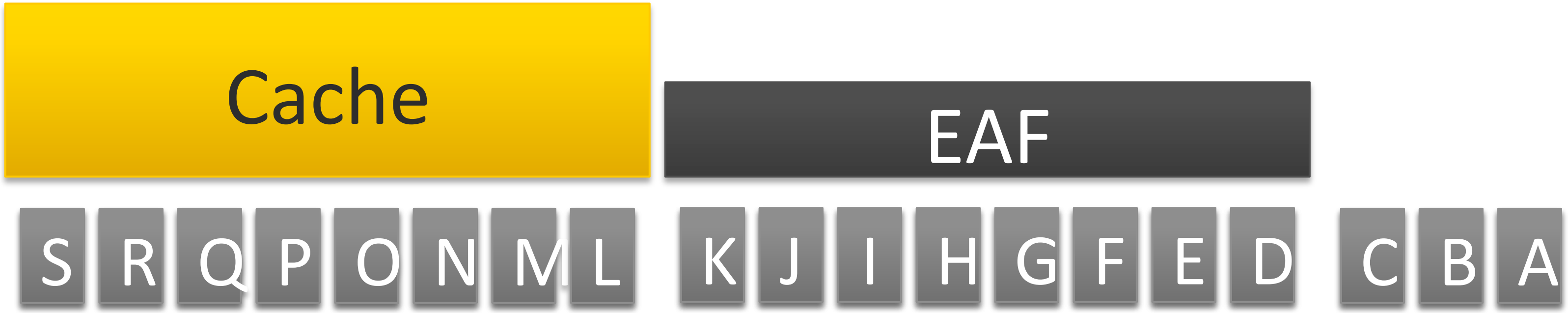
- Background and Motivation
- Evicted-Address Filter
  - Reuse Prediction
  - Thrash Resistance
- Final Design
- Advantages and Disadvantages
- Evaluation
- Conclusion

Large Working Set: 2 Cases

1 Cache < Working set < Cache + EAF

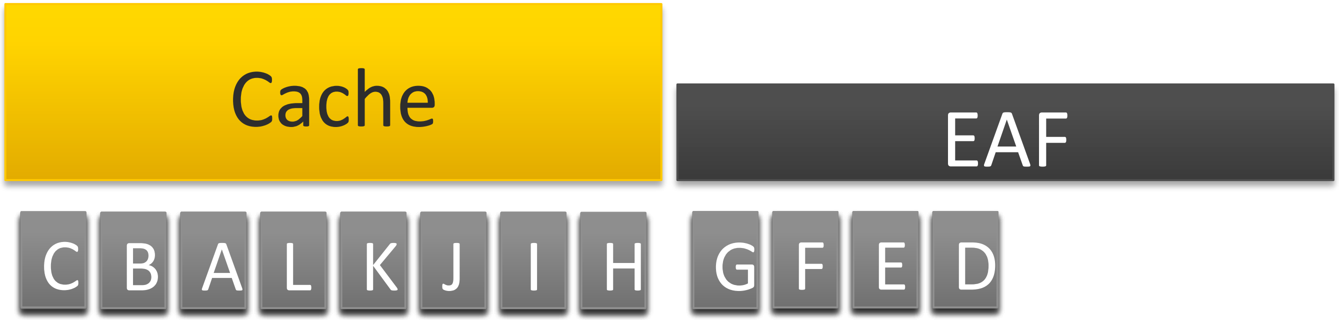


2 Cache + EAF < Working Set



Large Working Set: Case 1

Cache < Working set < Cache + EAF



Cache

EAF

C B A L K J I H

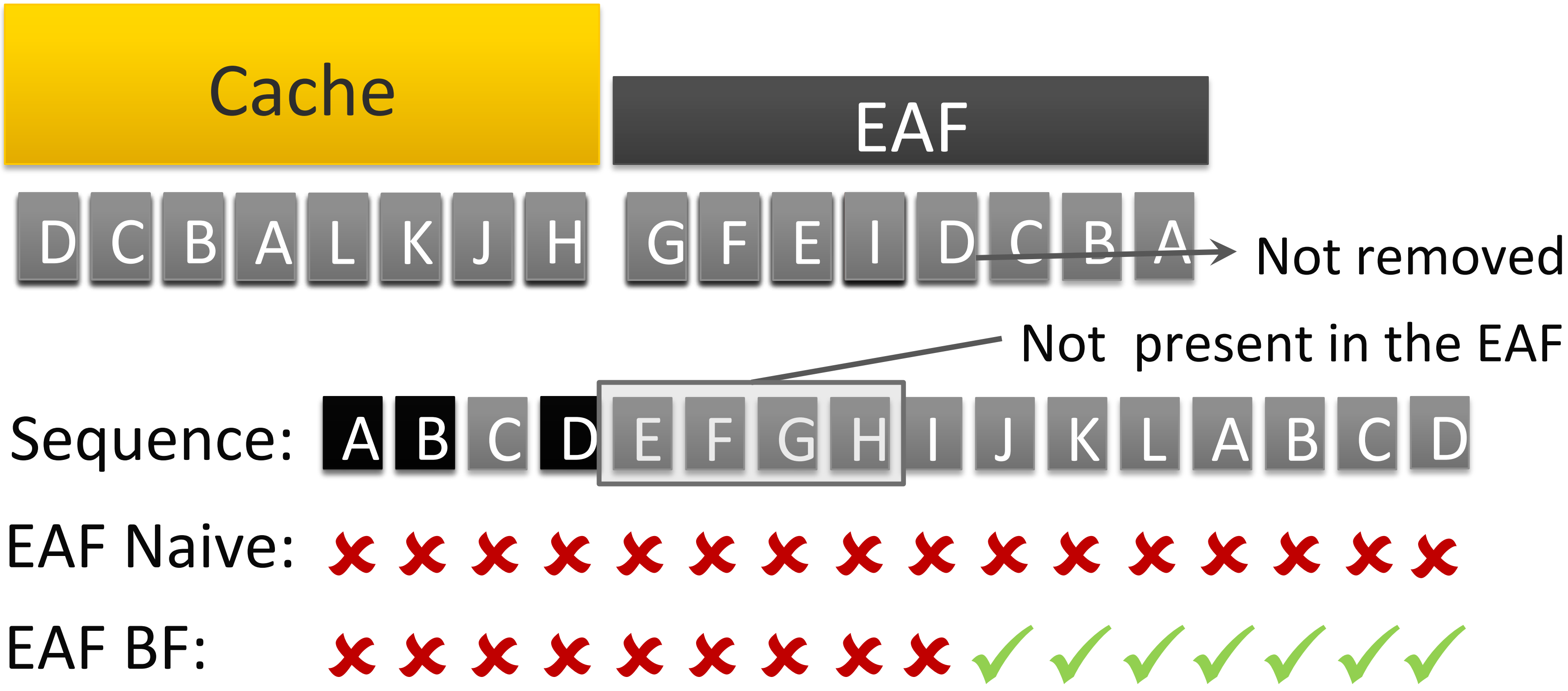
G F E D

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



Cache

EAF

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

Not removed

Not present in the EAF

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

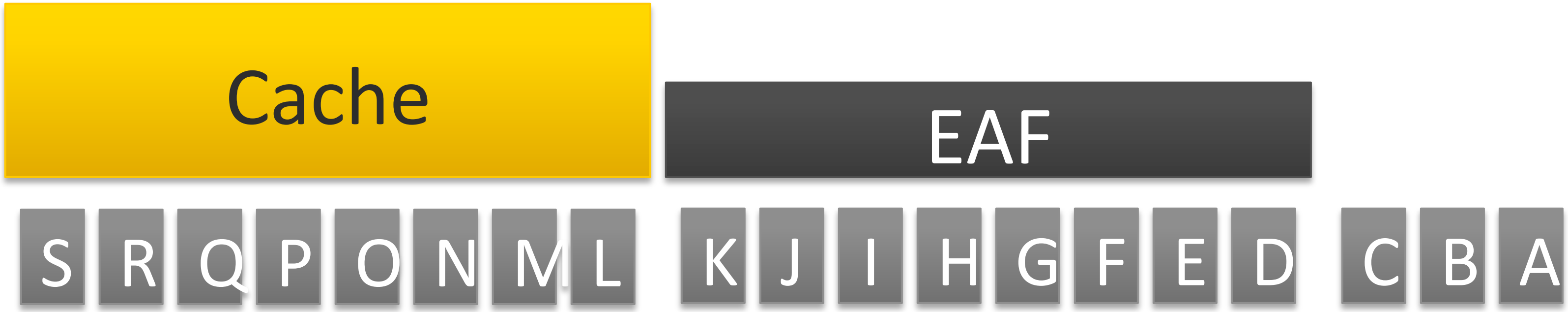
EAF Naive: ✗ ✗ ✗ ✗ ✗ ✗ ✗ ✗ ✗ ✗ ✗ ✗ ✗ ✗ ✗ ✗

EAF BF: ✗ ✗ ✗ ✗ ✗ ✗ ✗ ✗ ✗ ✓ ✓ ✓ ✓ ✓ ✓ ✓

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

1 **Cache eviction**

Insert address into filter

Increment counter

Cache
Bloom Filter
Counter

3 **Counter reaches max**

Clear filter and counter

2 **Cache miss**

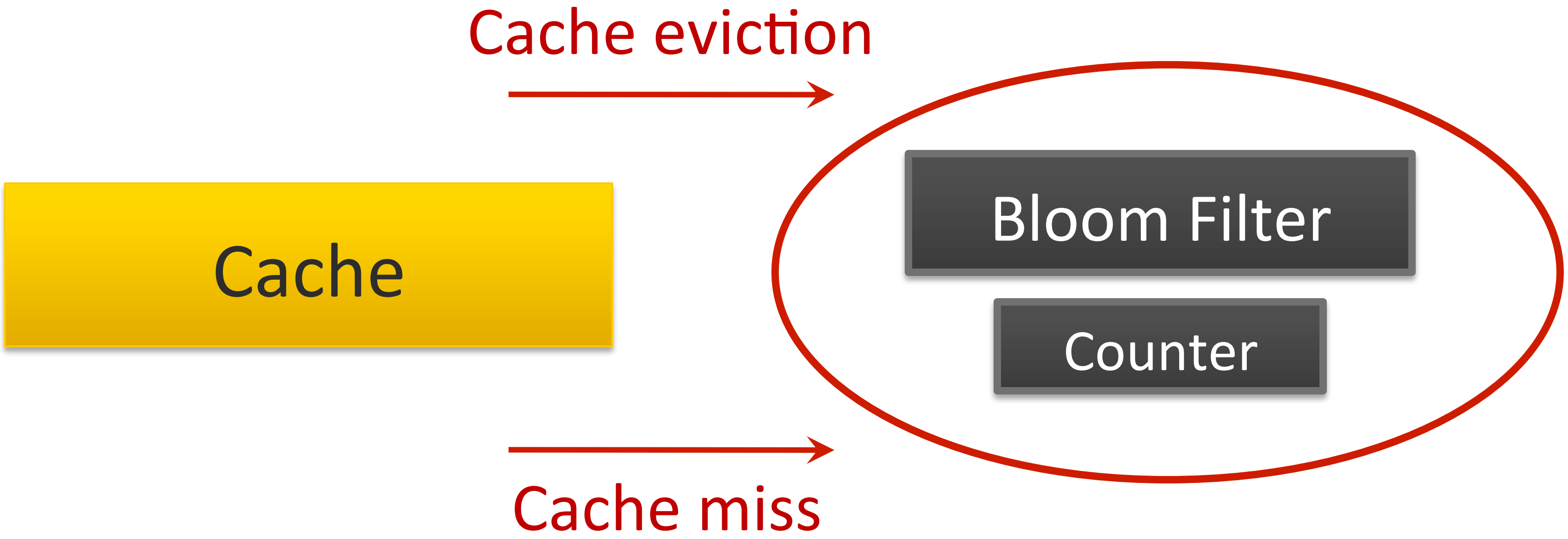
Test if address is present in filter

Yes, insert at MRU. No, insert with BIP

Outline

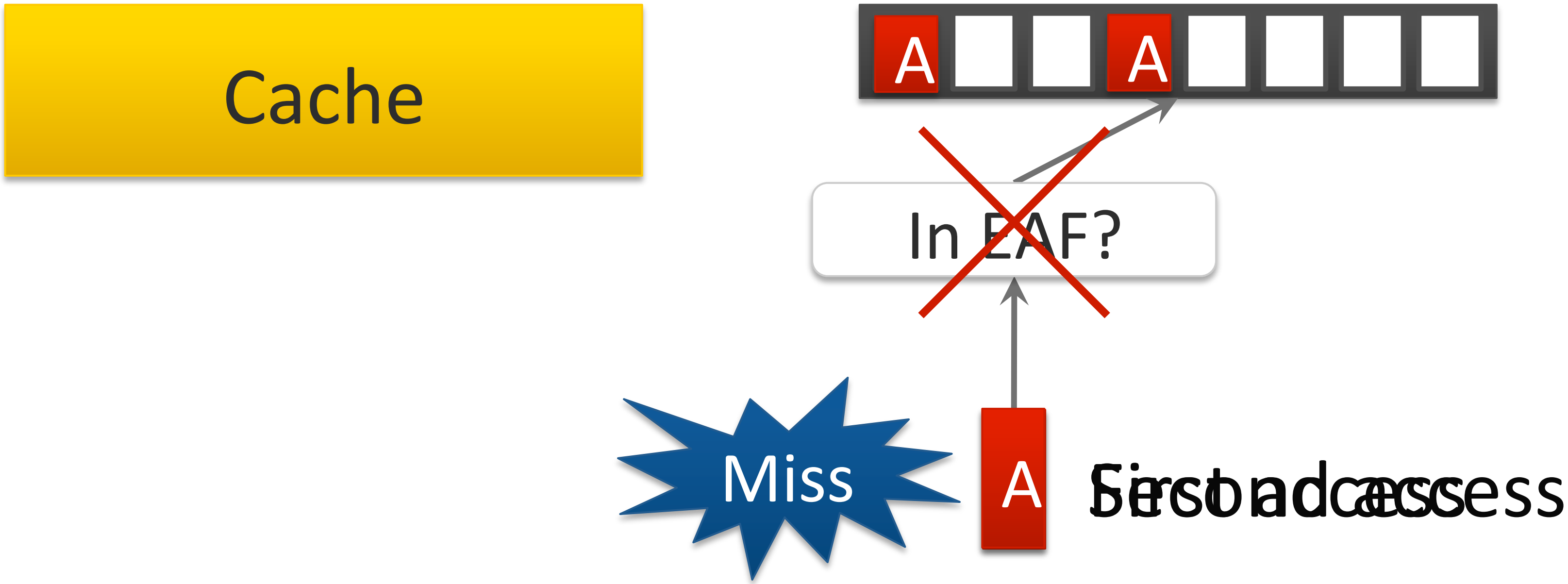
- Background and Motivation
- Evicted-Address Filter
  - Reuse Prediction
  - Thrash Resistance
- Final Design
- Advantages and Disadvantages
- Evaluation
- Conclusion

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 ⟹ Thrashing

Comparison with Prior Works

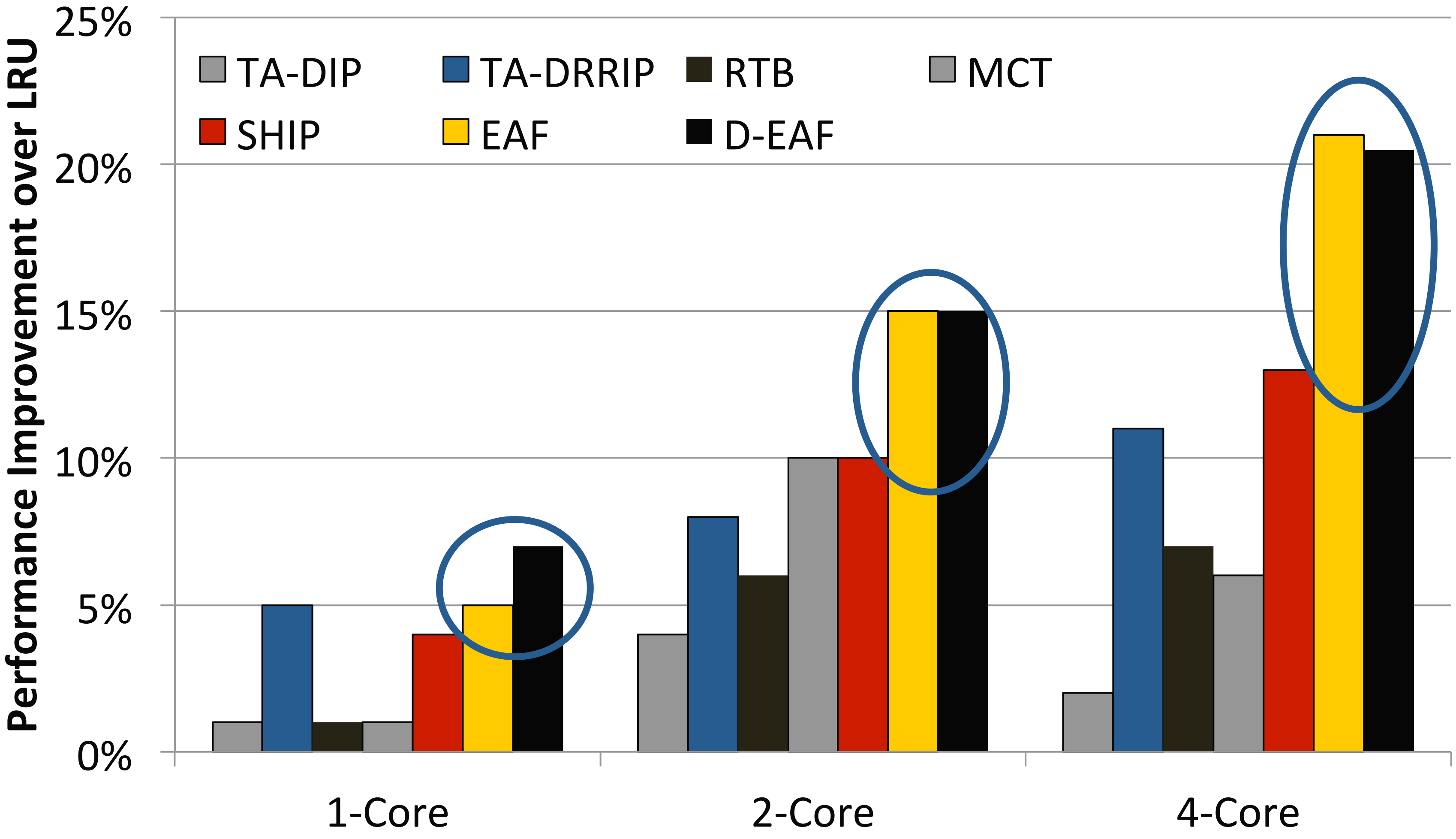
Addressing Cache Thrashing

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

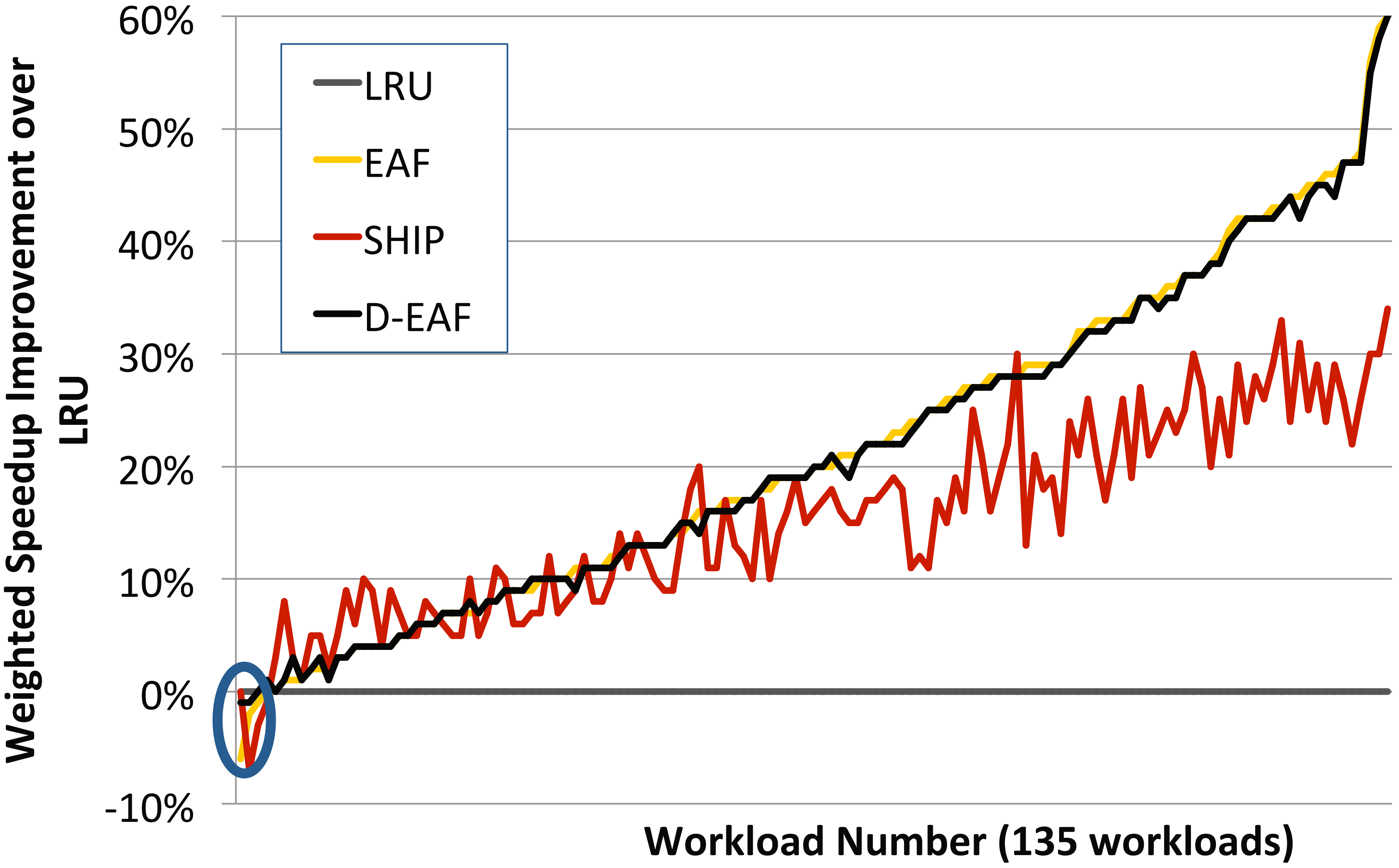
- Use set dueling to determine thrashing applications

- No mechanism to filter low-reuse blocks ⟹ Pollution

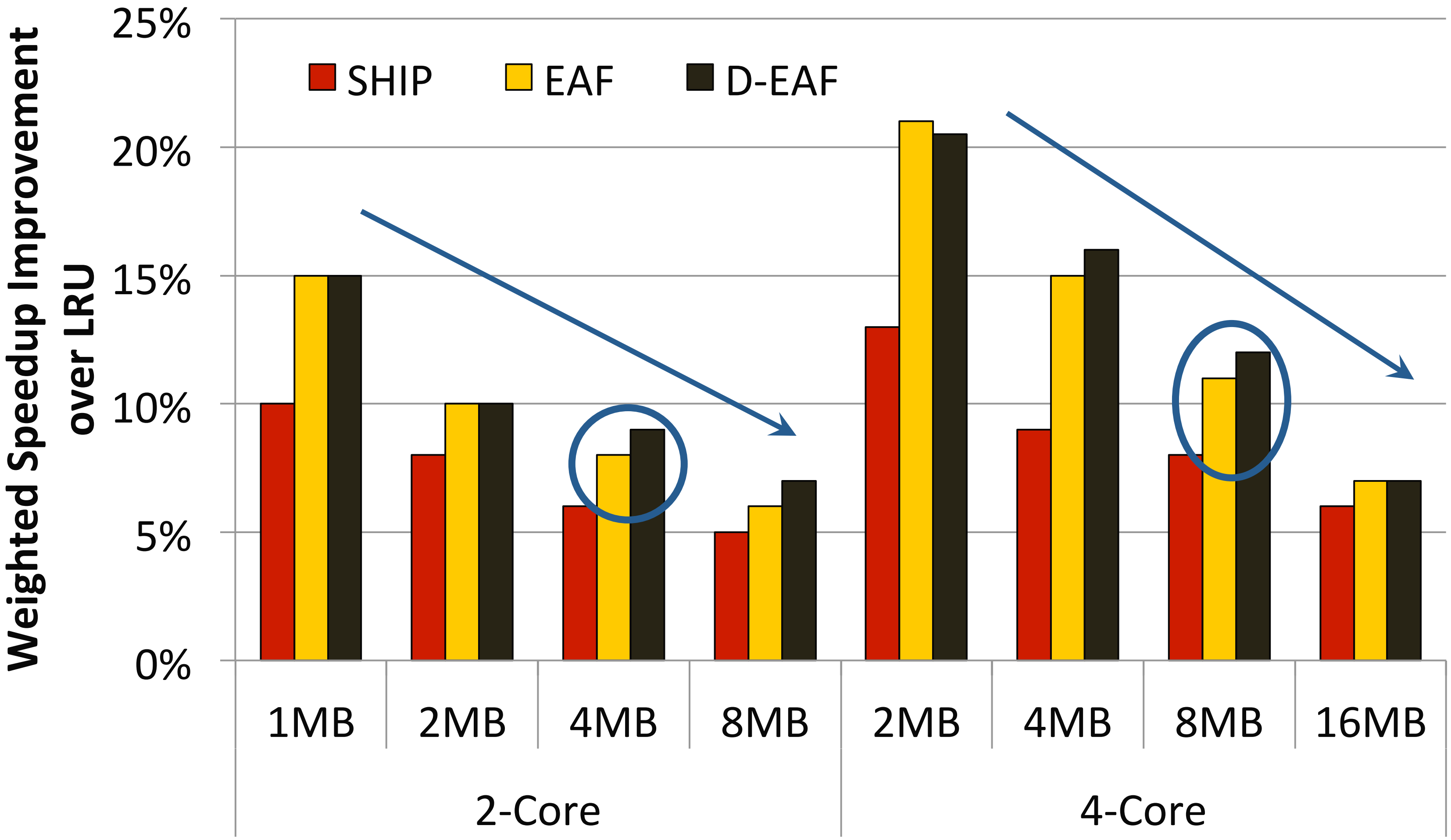
Results – Summary



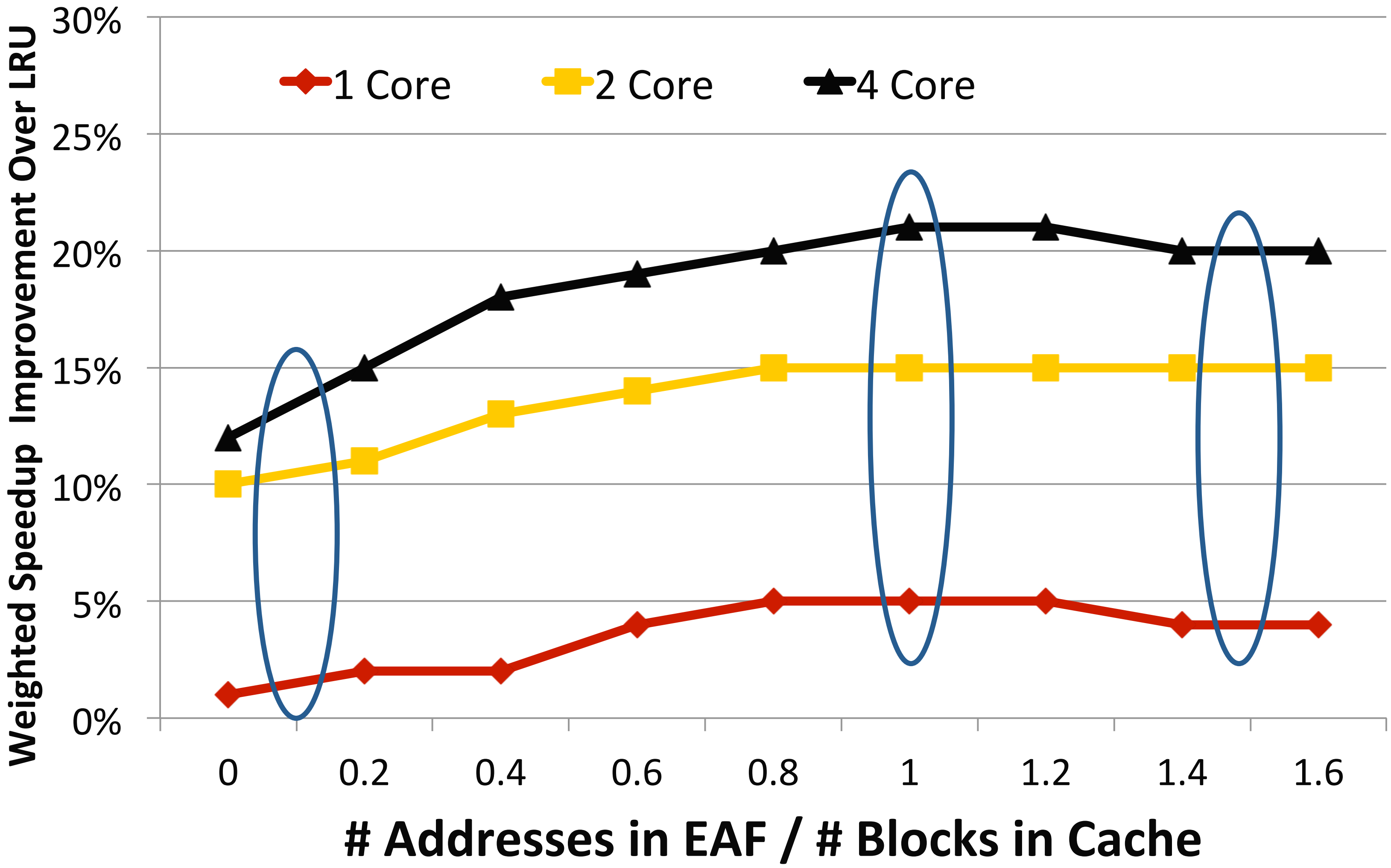
4-Core: Performance



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

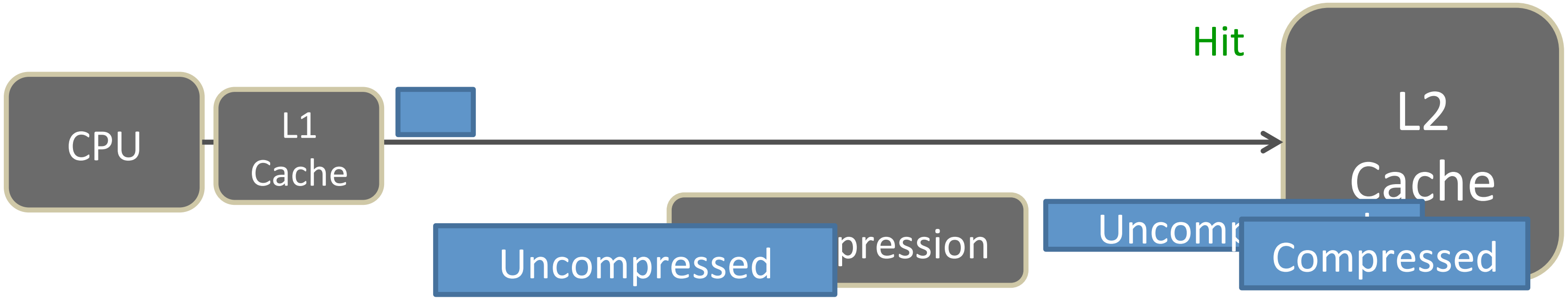
Significant redundancy in data:

0x**000000**00	0x**000000**0B	0x**000000**03	0x**000000**04	…

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

0x00000000	0x00000000	0x00000000	0x00000000	…

Repeated Values: common initial values, adjacent pixels

0x*000000***FF**	0x*000000***FF**	0x*000000***FF**	0x*000000***FF**	…

Narrow Values: small values stored in a big data type

0x*000000***00**	0x*000000***0B**	0x*000000***03**	0x*000000***04**	…

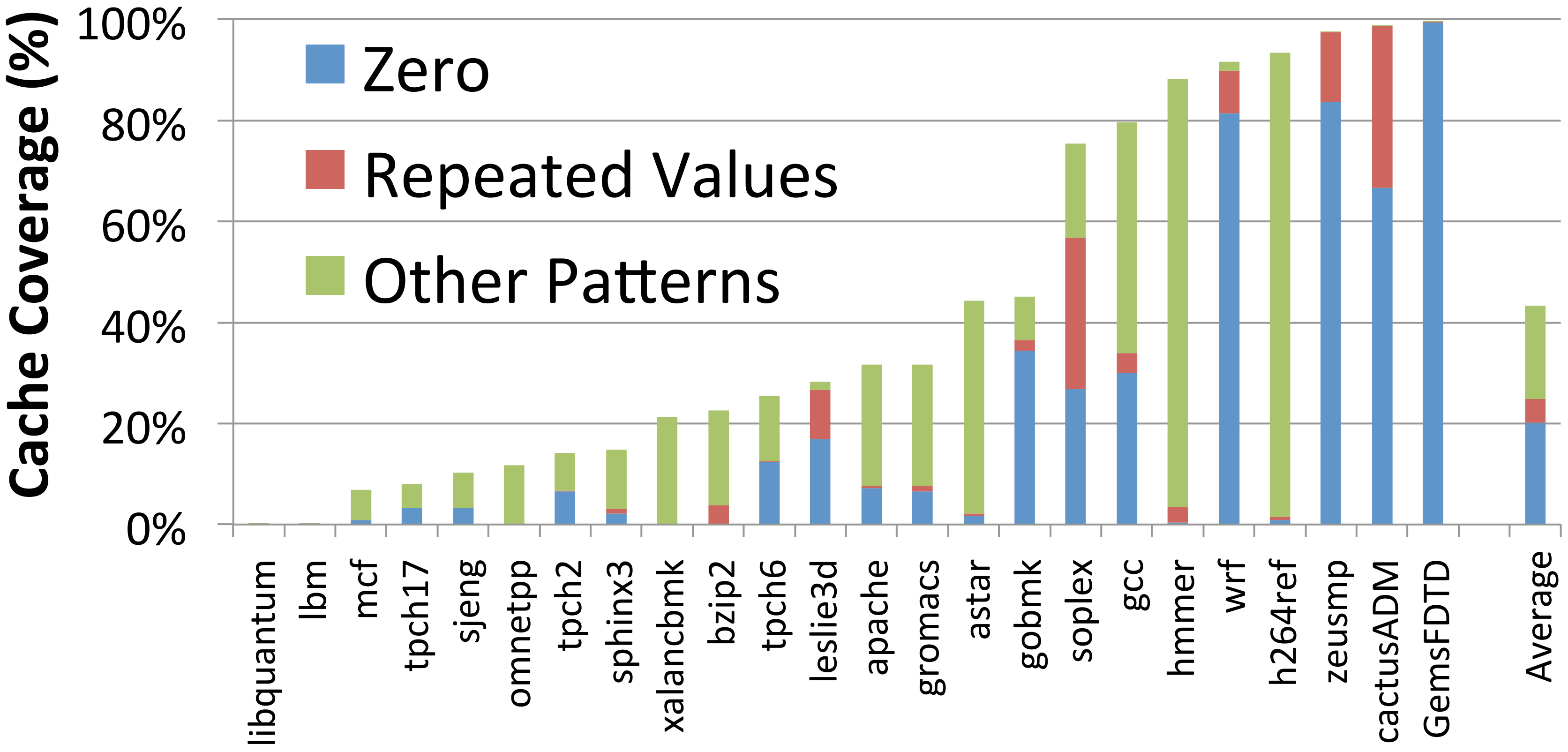
Other Patterns: pointers to the same memory region

0x*C04039***C0**	0x*C04039***C8**	0x*C04039***D0**	0x*C04039***D8**	…

How Common Are These Patterns?

SPEC2006, databases, web workloads, 2MB L2 cache

“Other Patterns” include Narrow Values



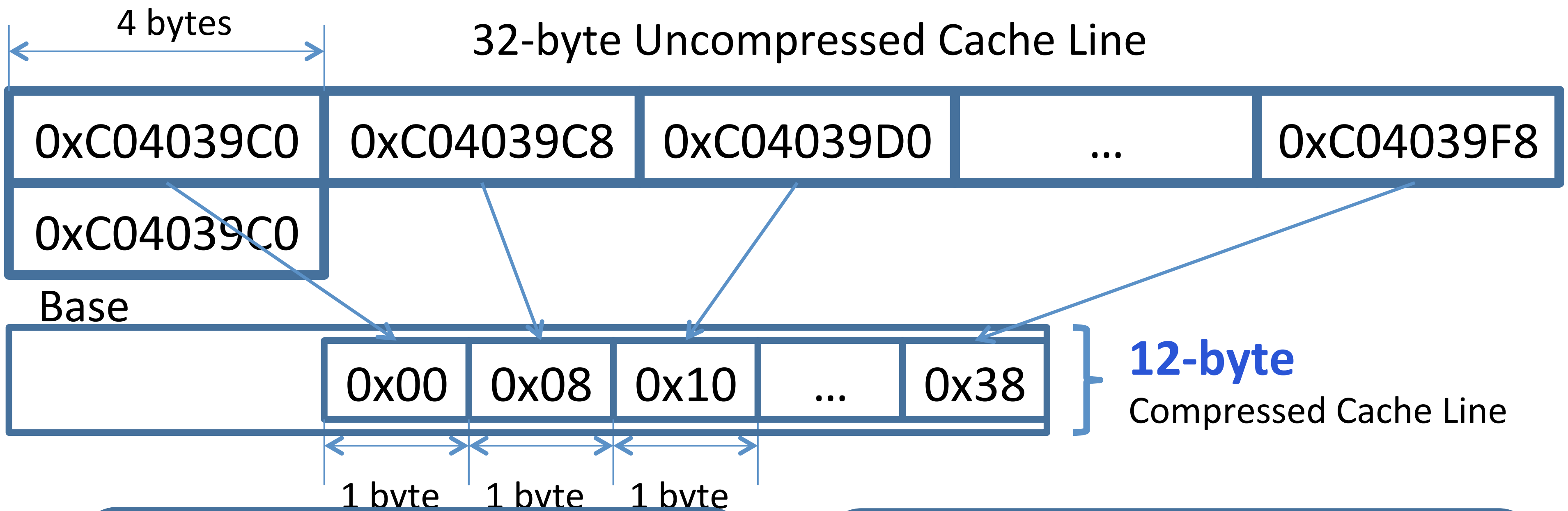
43% of the cache lines belong to key patterns

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

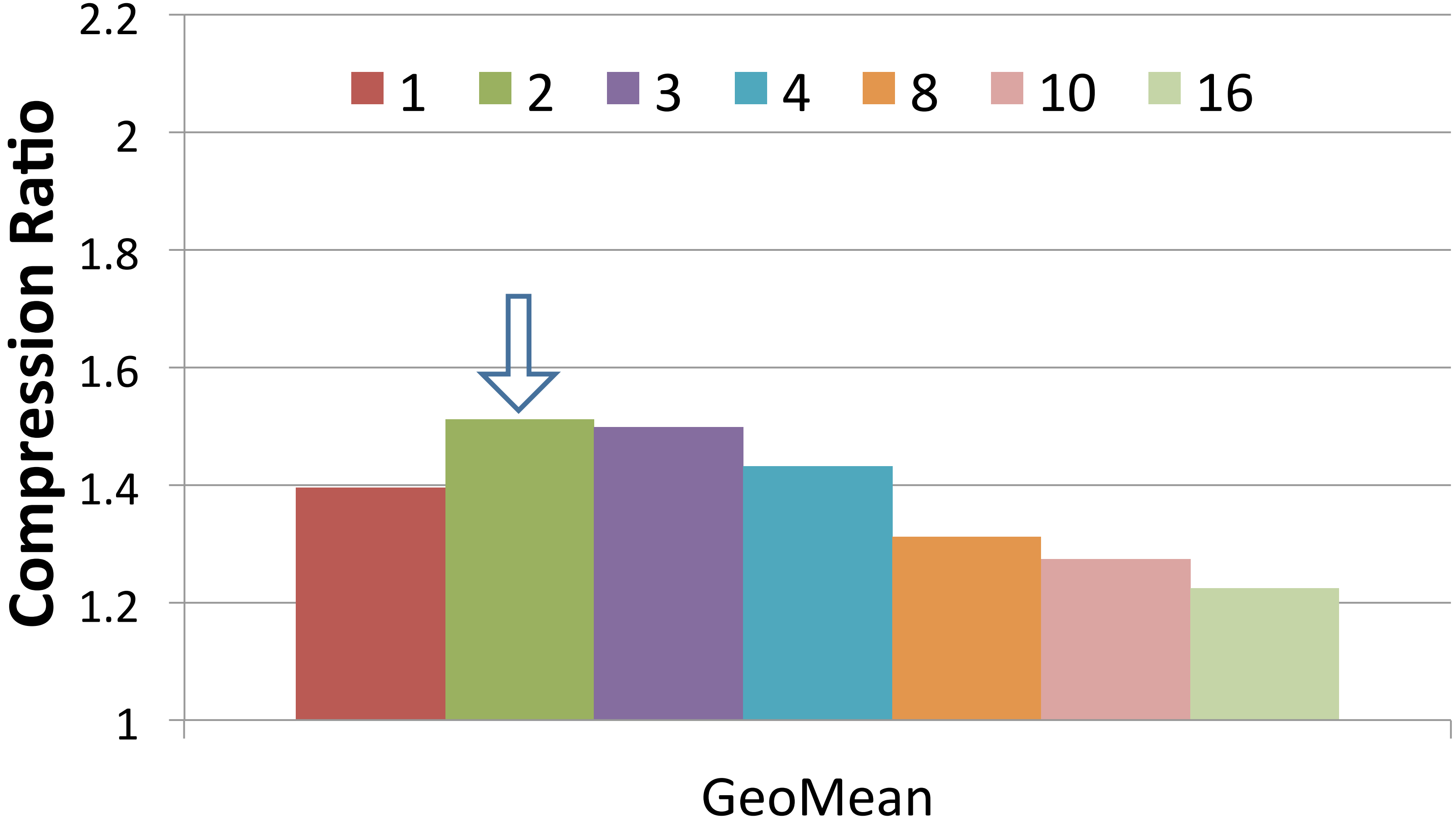
Can We Do Better?

- Uncompressible cache line (with a single base):

0x00000000	**0x09A4**0178	**0x000000**0B	**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**

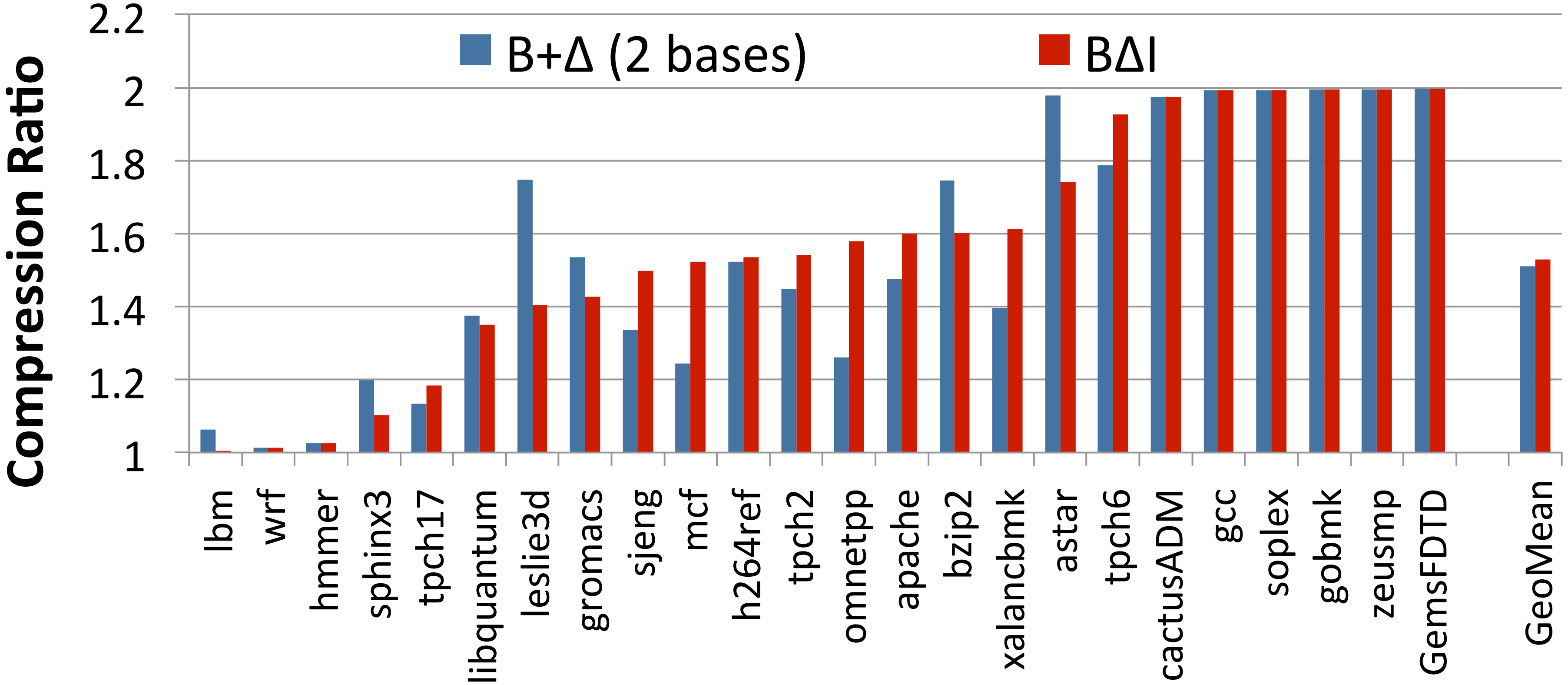
Advantages over 2 arbitrary bases:

– Better compression ratio

– Simpler compression logic

Base-Delta-Immediate (BΔI) Compression

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

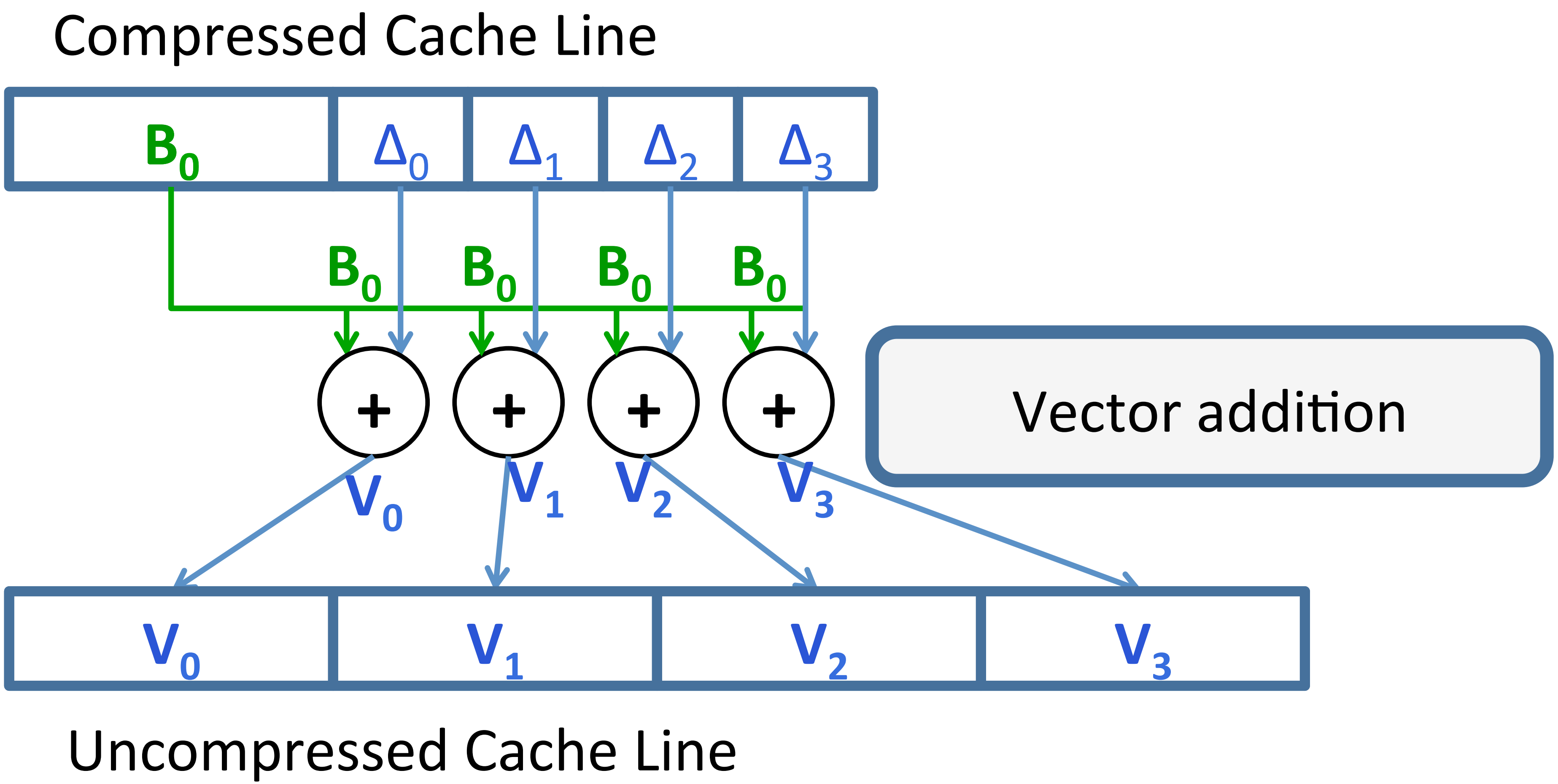


Average compression ratio is close, but **BΔI** is **simpler**

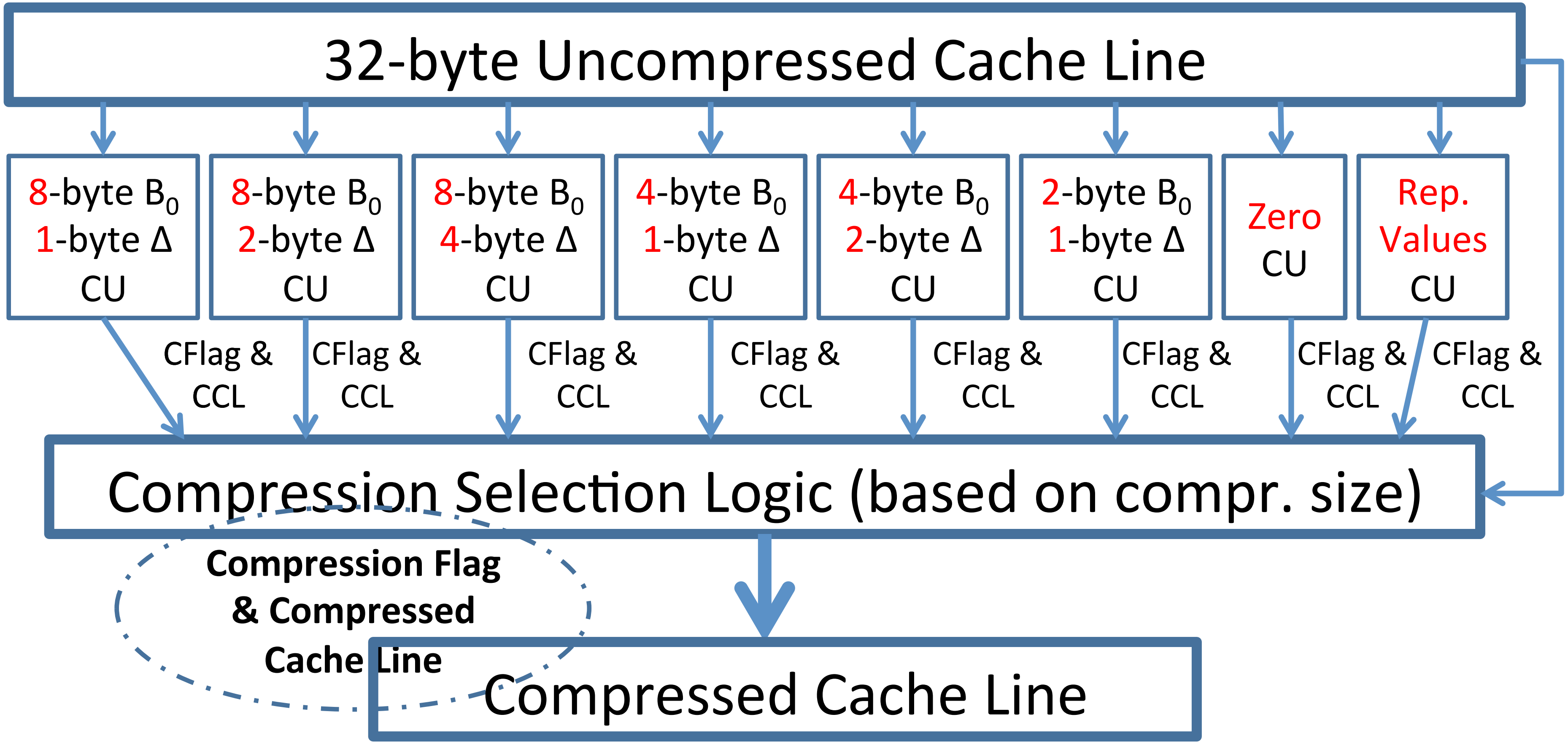
BΔI Implementation

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

BΔI Decompressor Design



BΔI Compressor Design



32-byte Uncompressed Cache Line

8-byte B_0 1-byte Δ CU

8-byte B_0 2-byte Δ CU

8-byte B_0 4-byte Δ CU

4-byte B_0 1-byte Δ CU

4-byte B_0 2-byte Δ CU

2-byte B_0 1-byte Δ CU

Zero CU

Rep. Values CU

CFlag & CCL

CFlag & CCL

CFlag & CCL

CFlag & CCL

CFlag & CCL

CFlag & CCL

CFlag & CCL

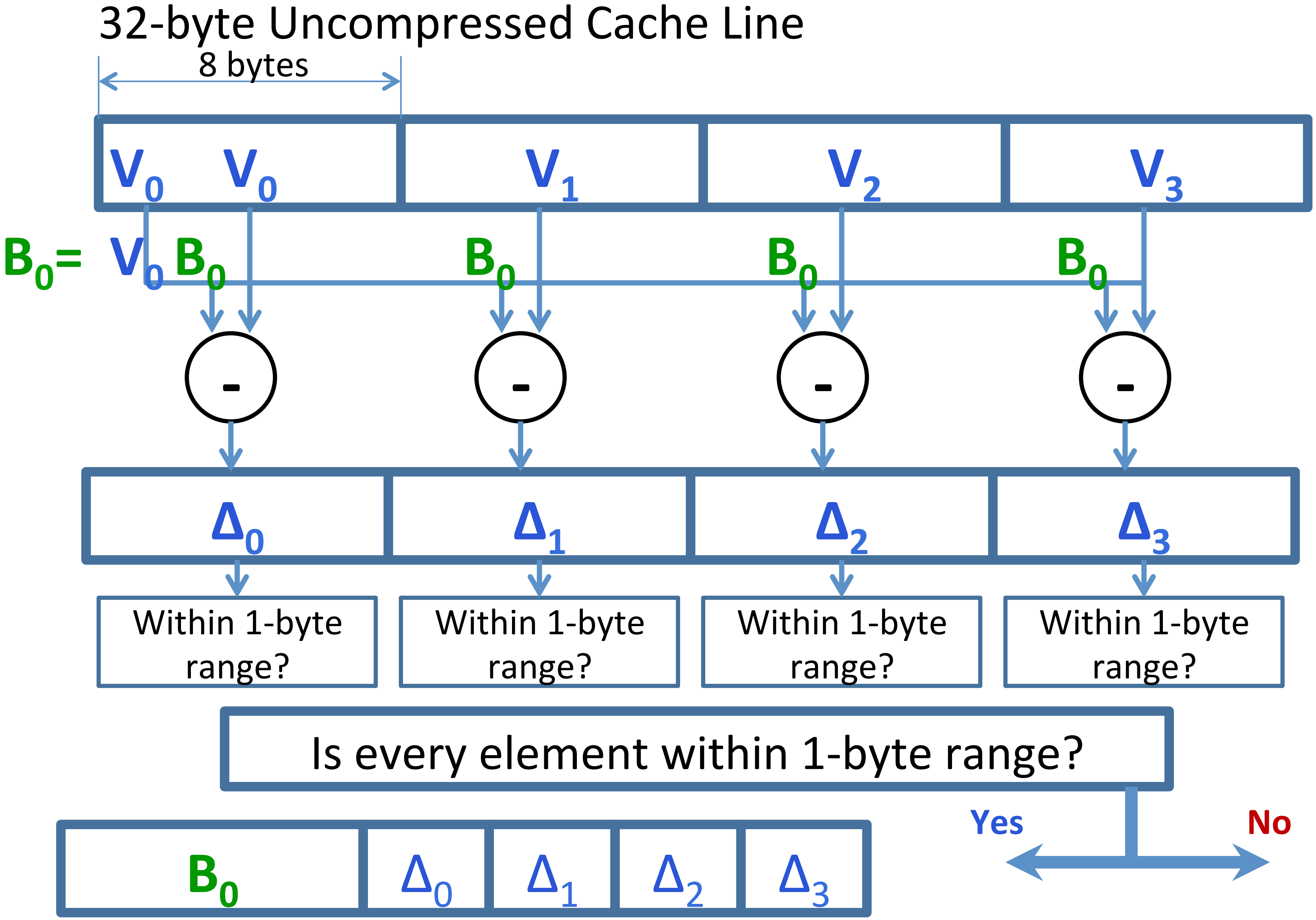
CFlag & CCL

Compression Selection Logic (based on compr. size)

Compression Flag & Compressed Cache Line

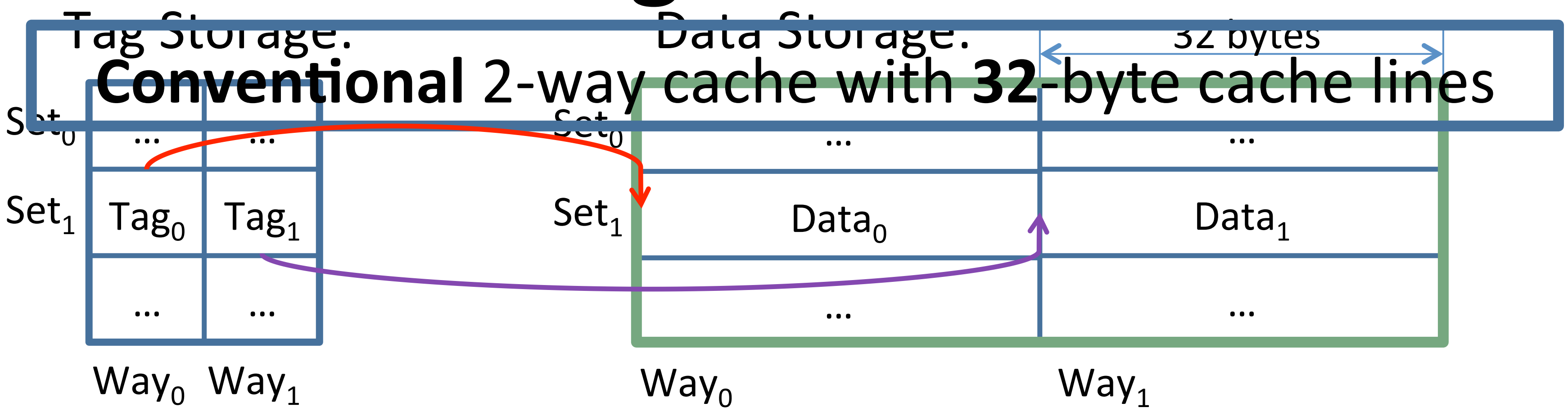
Compressed Cache Line

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

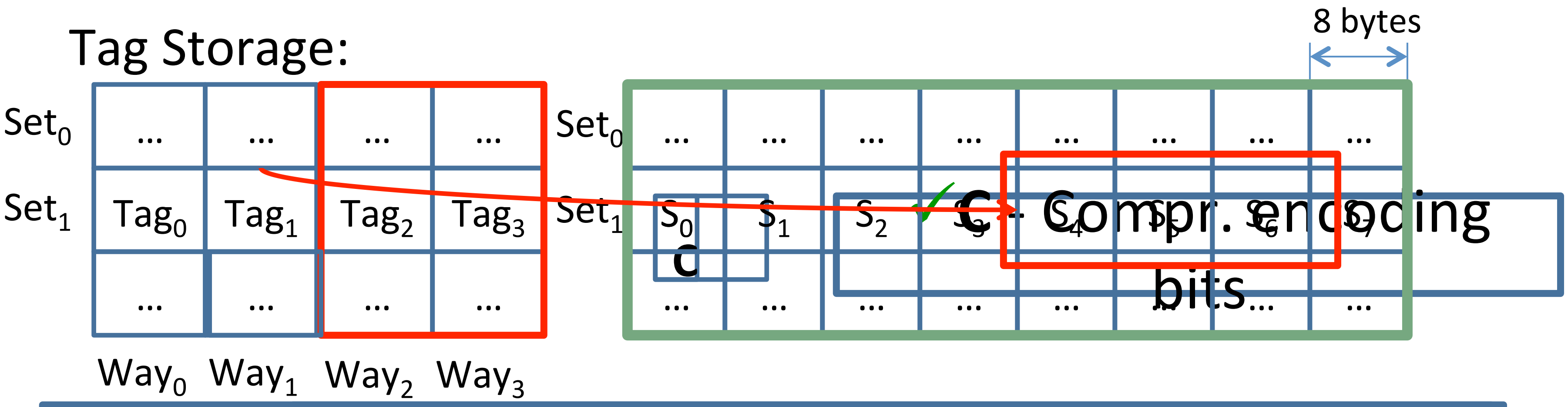


BΔI Cache Organization

Conventional 2-way cache with **32**-byte cache lines



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



✓**C** - Compr. encoding bits

✓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)

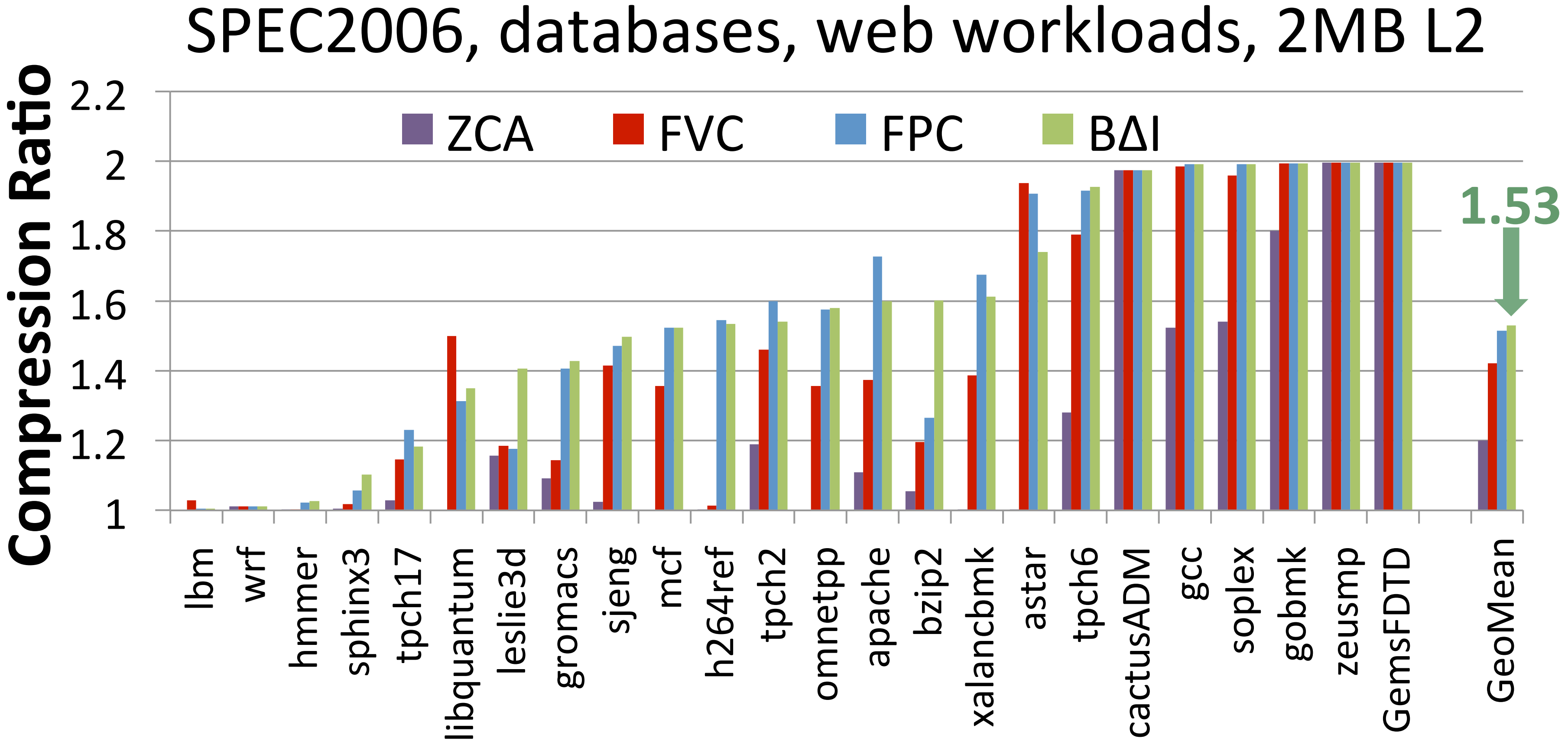
Outline

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

Methodology

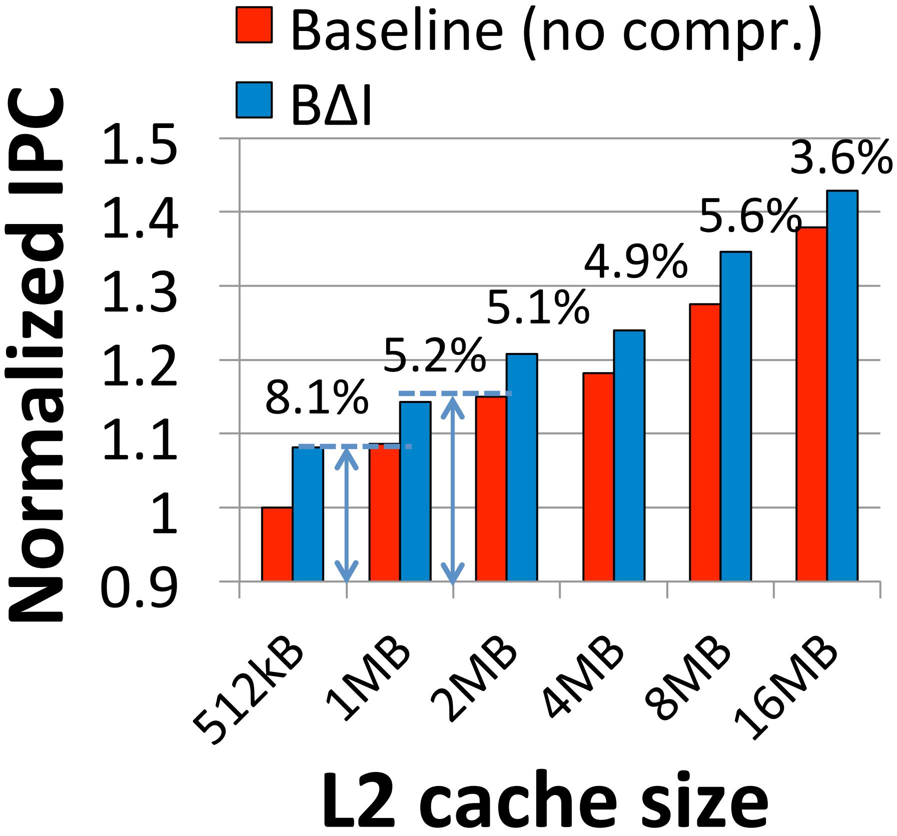
- **Simulator**
  - x86 event-driven simulator based on Simics *[Magnusson+, Computer'02]*
- **Workloads**
  - SPEC2006 benchmarks, TPC, Apache web server
  - 1 – 4 core simulations for 1 billion representative instructions
- **System Parameters**
  - L1/L2/L3 cache latencies from CACTI *[Thoziyoor+,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

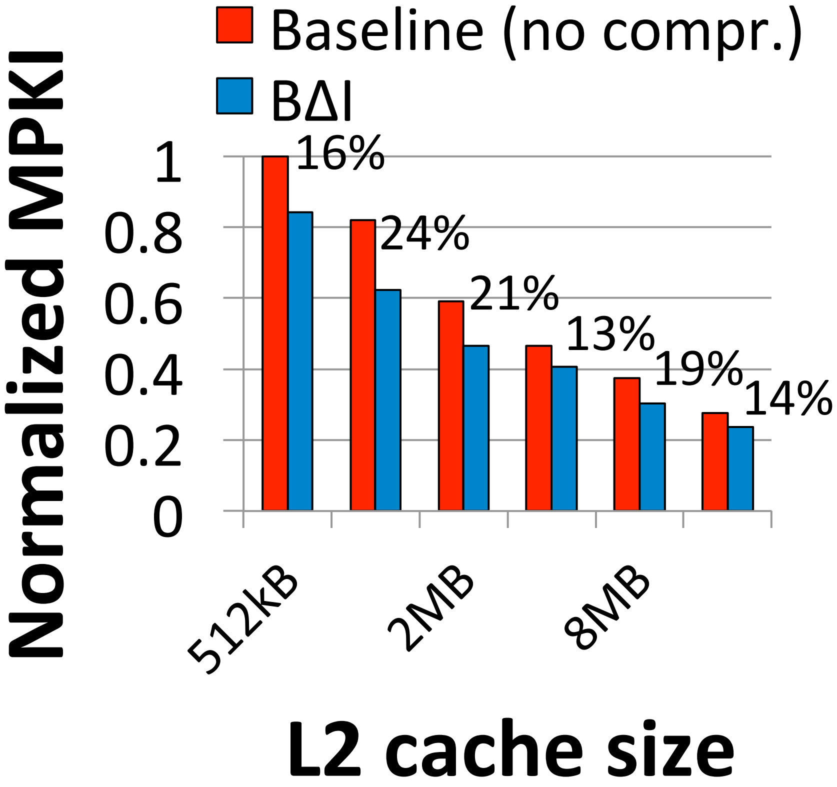
Compression Ratio: BΔI vs. Prior Work



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)

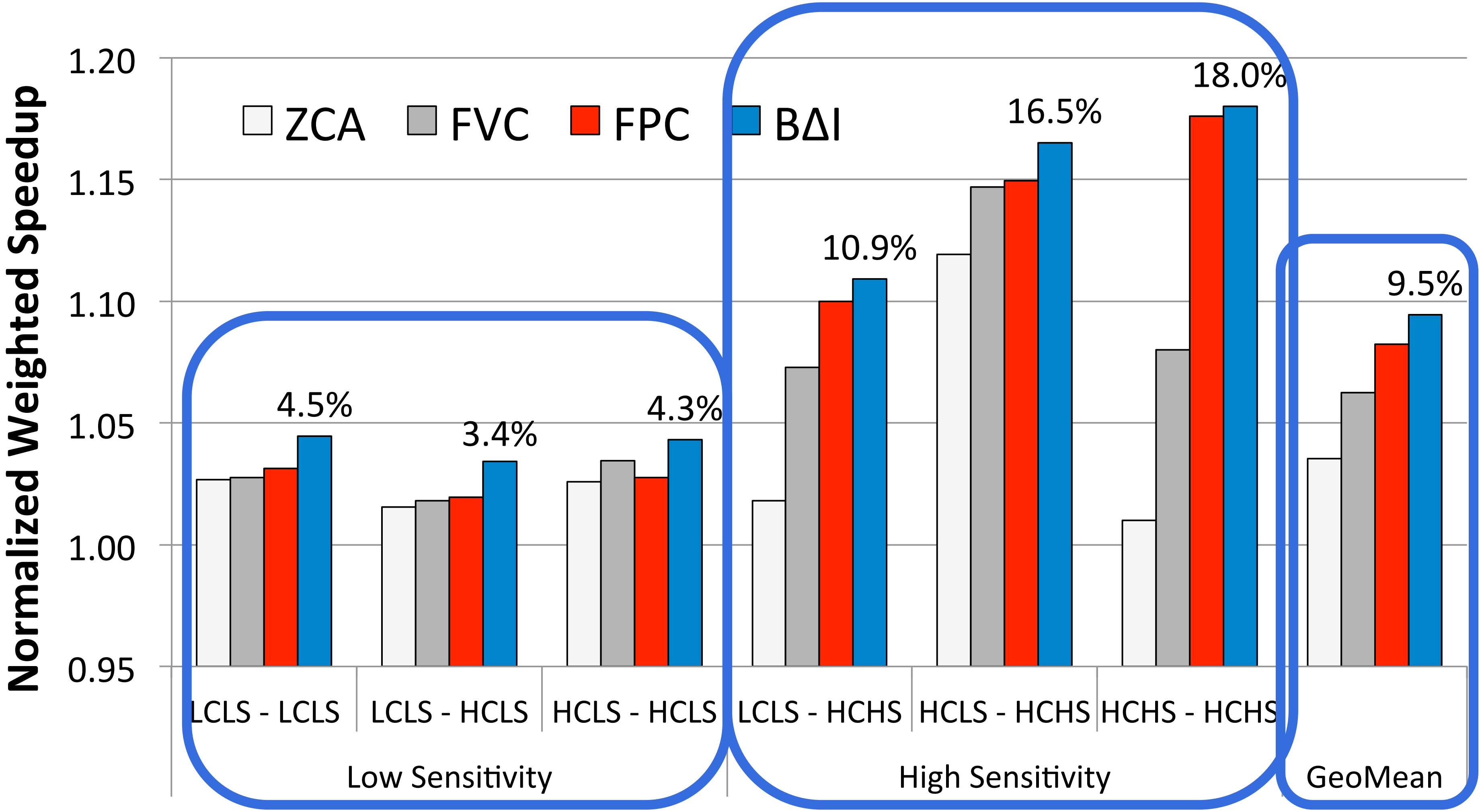
  Sensitivity: performance gain with more cache

  (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