18-740/640
Computer Architecture
Lecture 14: Memory Resource Management I

Prof. Onur Mutlu
Carnegie Mellon University

Fall 2015, 10/26/2015

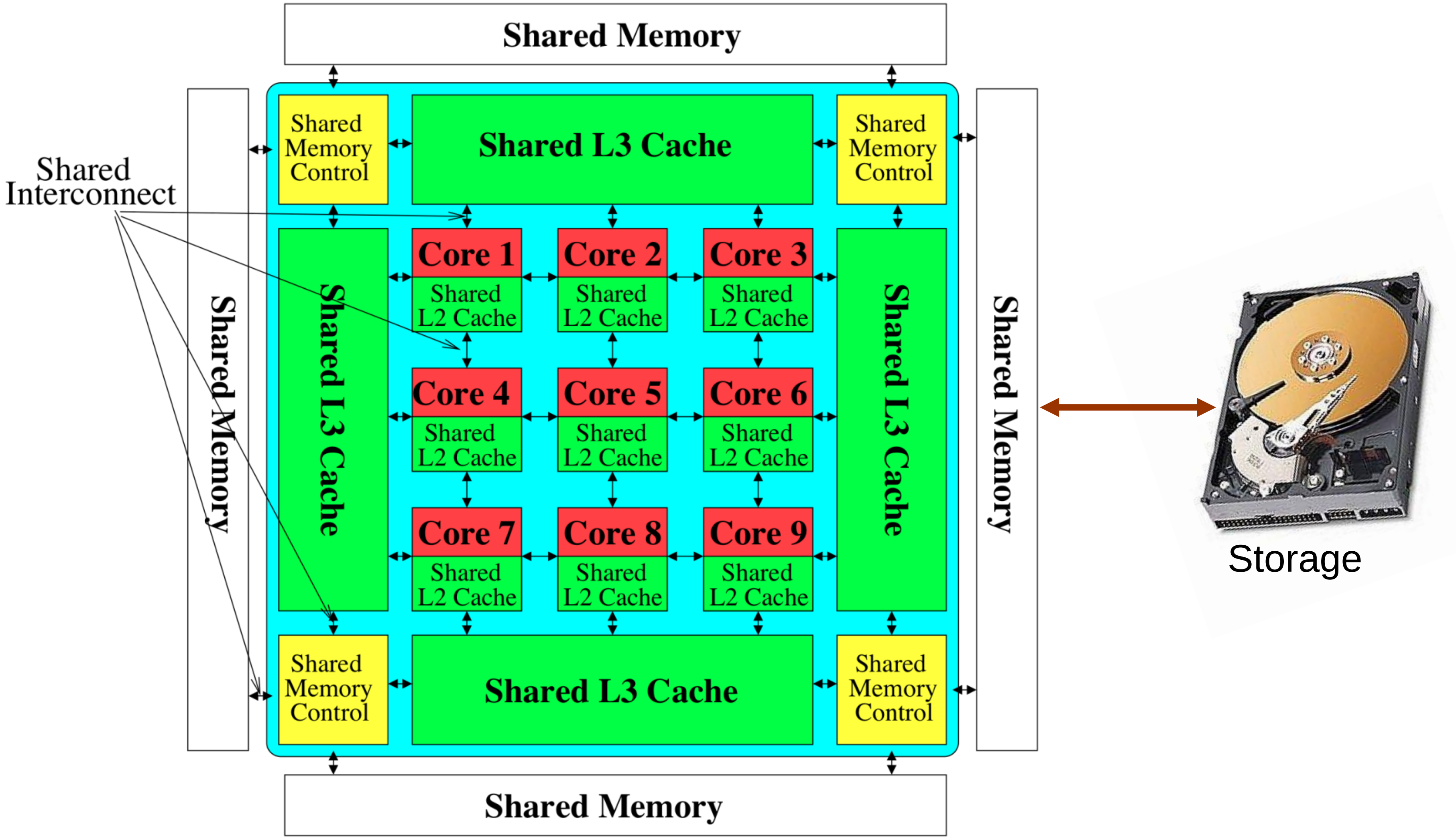
Required Readings

- Required Reading Assignment:
  - Qureshi and Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," MICRO 2006.

- Recommended References:
  - 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.
  - Lin et al., "Gaining Insights into Multi-Core Cache Partitioning: Bridging the Gap between Simulation and Real Systems," HPCA 2008.

Shared Resource Design for Multi-Core Systems

Memory System: A ***Shared Resource*** View



Resource Sharing Concept

- 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, shared data kept in the same cache in SMT processors

+ Compatible with the shared memory model

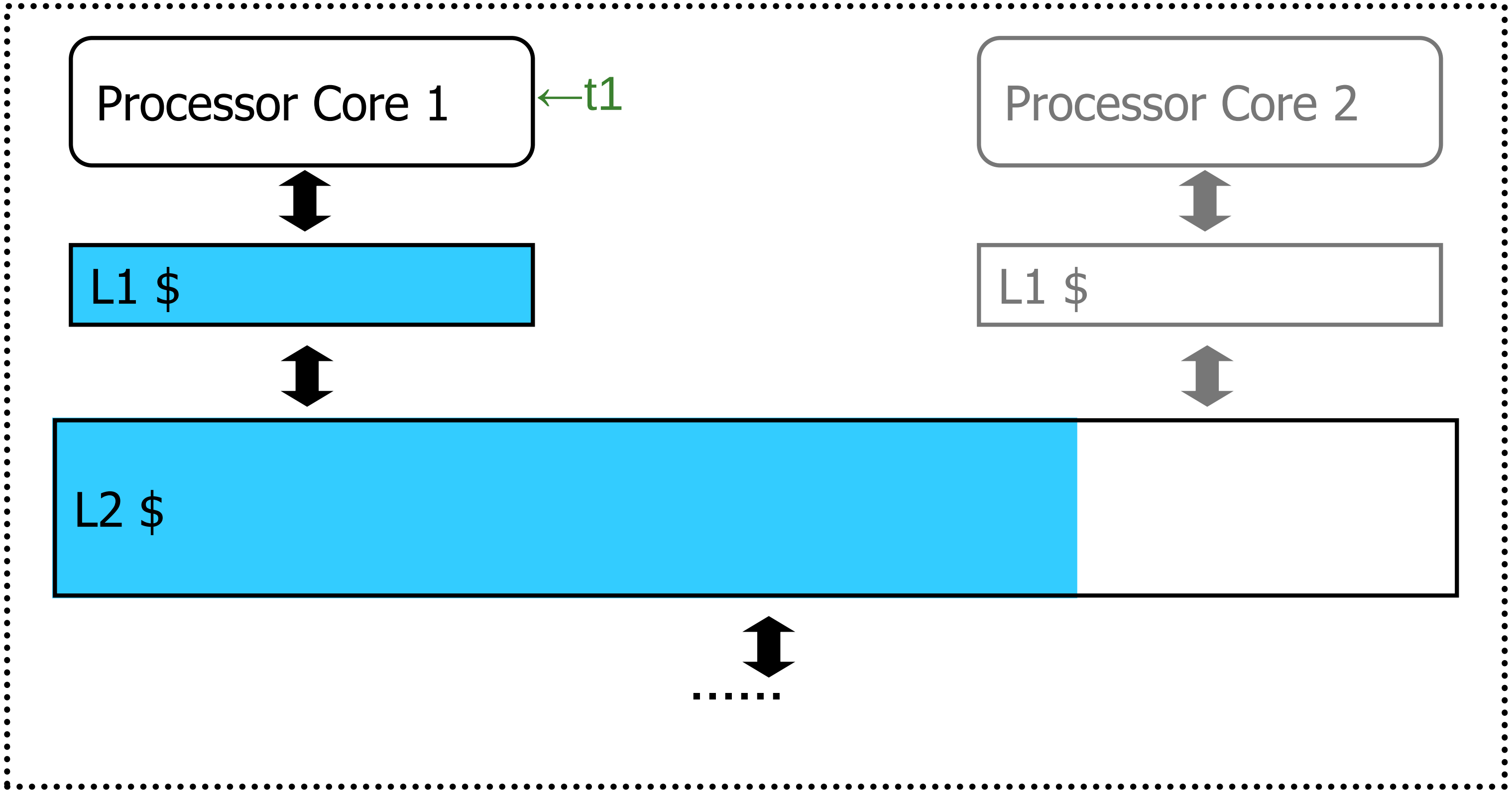
Resource Sharing Disadvantages

- 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

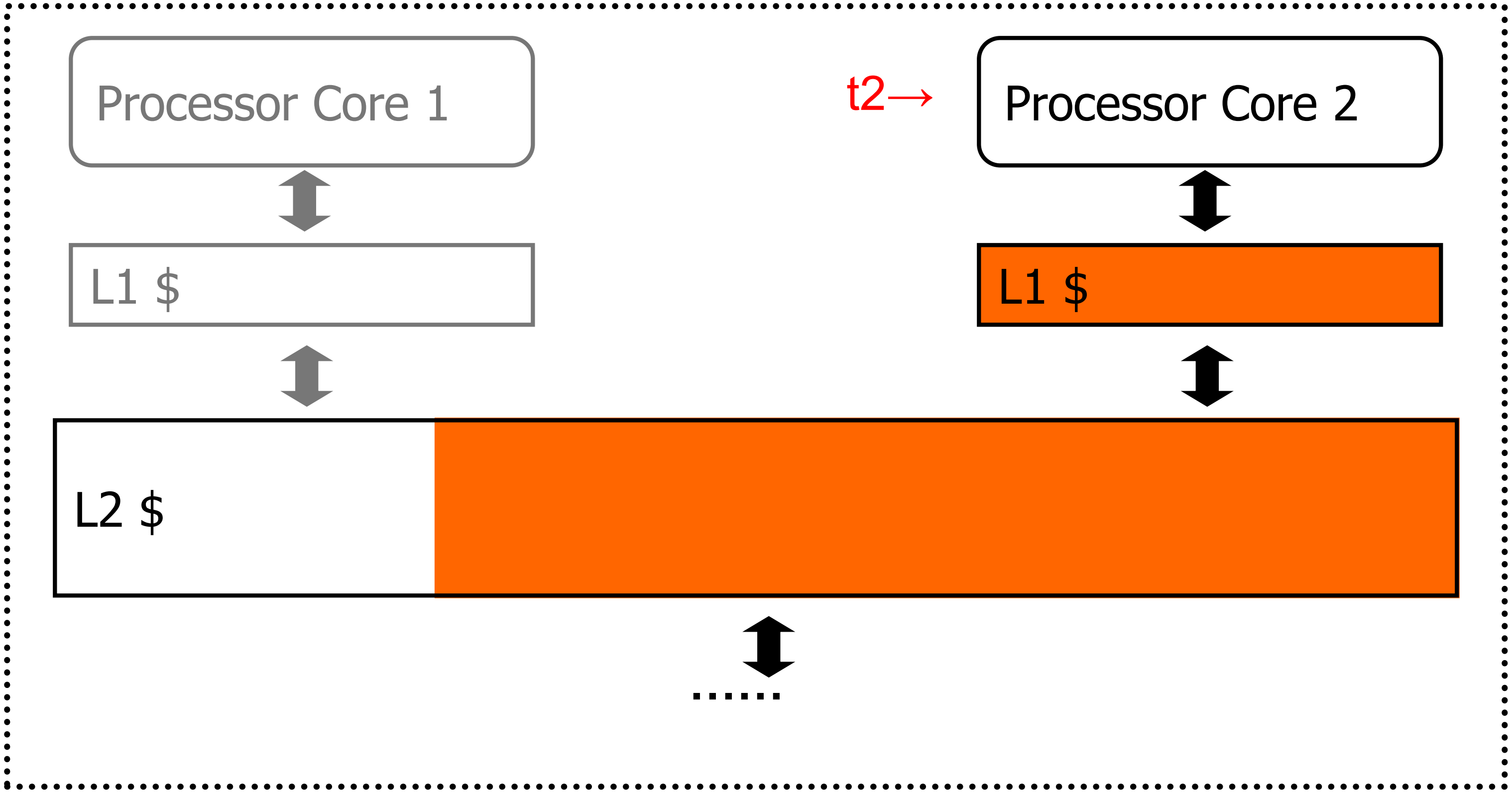
Need to efficiently and fairly utilize shared resources

Example: Problem with Shared Caches



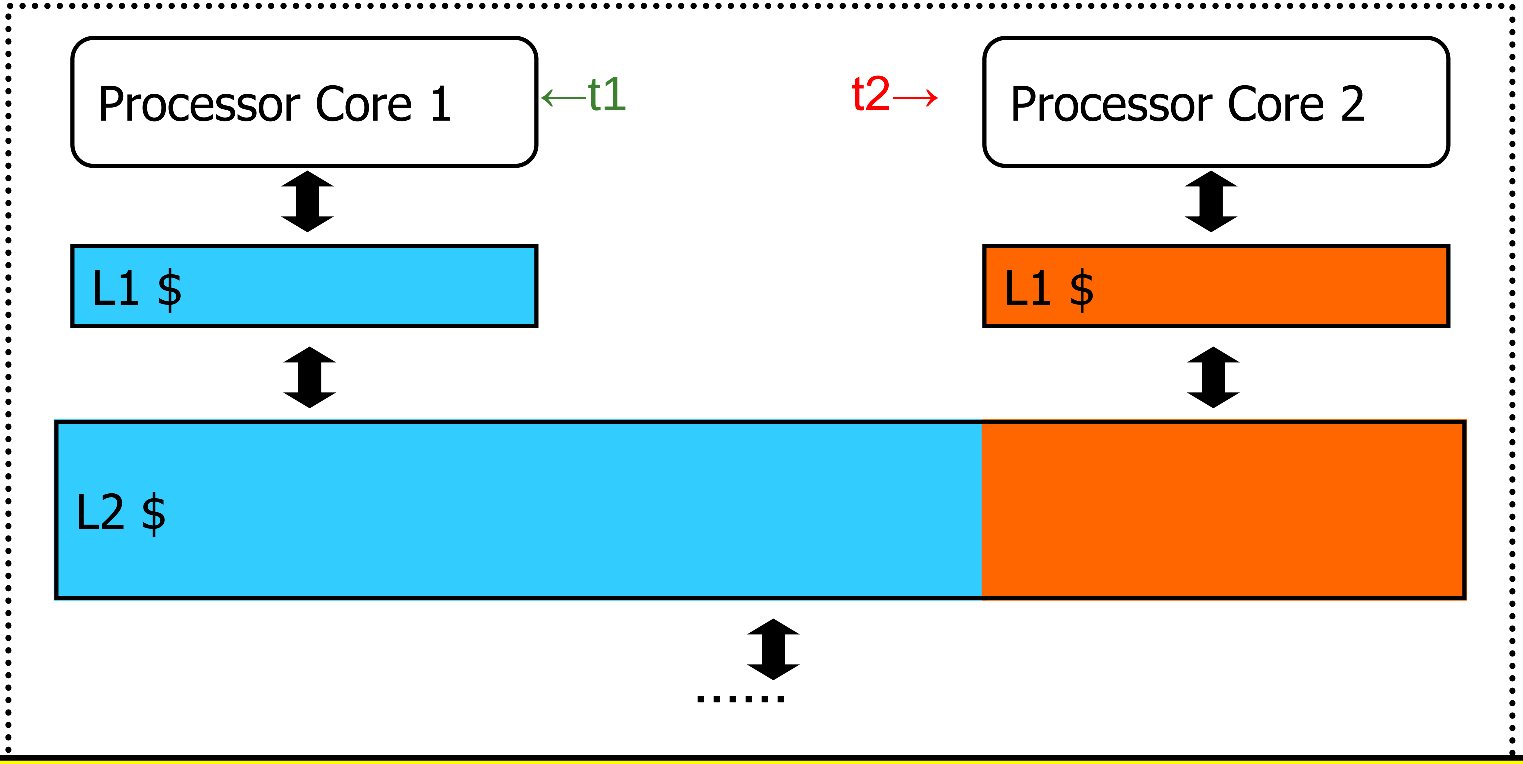
Kim et al., "Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture," PACT 2004.

Example: Problem with Shared Caches



Kim et al., "Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture," PACT 2004.

Example: Problem with Shared Caches



t2's throughput is significantly reduced due to unfair cache sharing.

Kim et al., "Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture," PACT 2004.

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

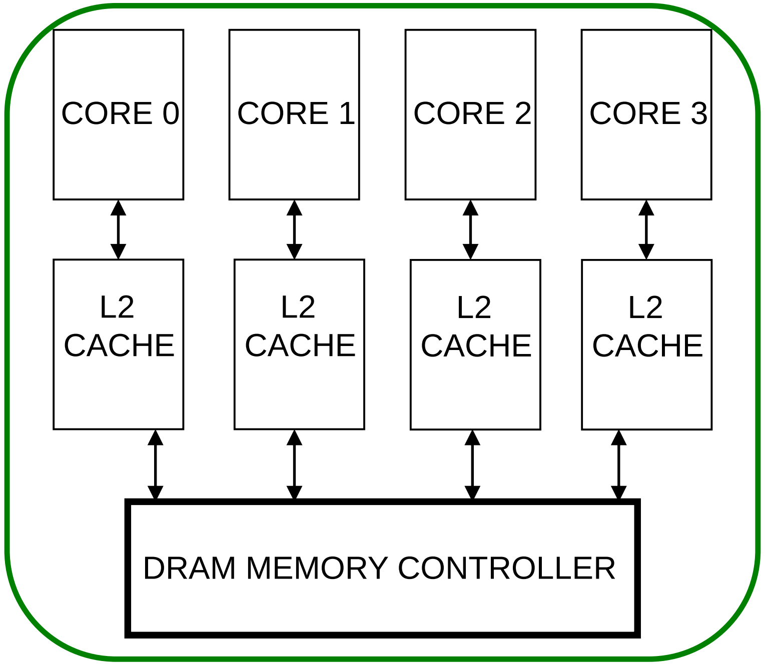
- Memory subsystem (in both Multi-threaded and Multi-core)
  - Non-private caches
  - Interconnects
  - Memory controllers, buses, banks

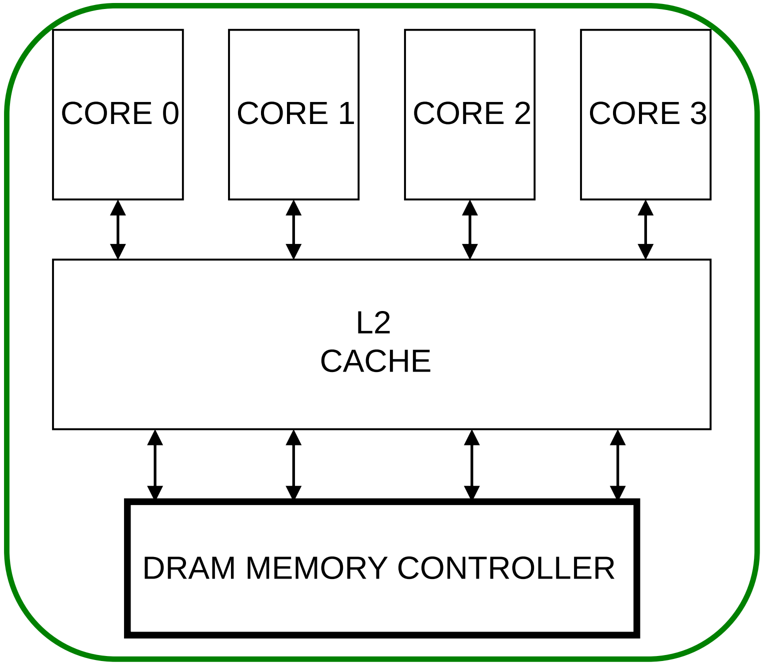
- I/O subsystem (in both Multi-threaded and Multi-core)
  - I/O, DMA controllers
  - Ethernet controllers

- Processor (in Multi-threaded)
  - Pipeline resources
  - L1 caches

Multi-core Issues in Caching

- How does the cache hierarchy change in a multi-core system?
- 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
  - Easier to maintain coherence (a cache block is in a single location)
  - Shared data and locks do not ping pong between caches

- Disadvantages
  - Slower access
  - 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

Handling Shared Caches

- Controlled cache sharing
  - Approach 1: Design shared caches but control the amount of cache allocated to different cores
  - Approach 2: Design "private" caches but spill/receive data from one cache to another

- More efficient cache utilization
  - Minimize the wasted cache space
    - by keeping out useless blocks
    - by keeping in cache blocks that have maximum benefit
    - by minimizing redundant data

Controlled Cache Sharing: Examples

- Utility based cache partitioning
  - 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.

- Fair cache partitioning
  - Kim et al., "Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture," PACT 2004.

- Shared/private mixed cache mechanisms
  - Qureshi, "Adaptive Spill-Receive for Robust High-Performance Caching in CMPs," HPCA 2009.
  - Hardavellas et al., "Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches," ISCA 2009.

Efficient Cache Utilization: Examples

- 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.

Controlled Shared Caching

Hardware-Based Cache Partitioning

Utility Based Shared Cache Partitioning

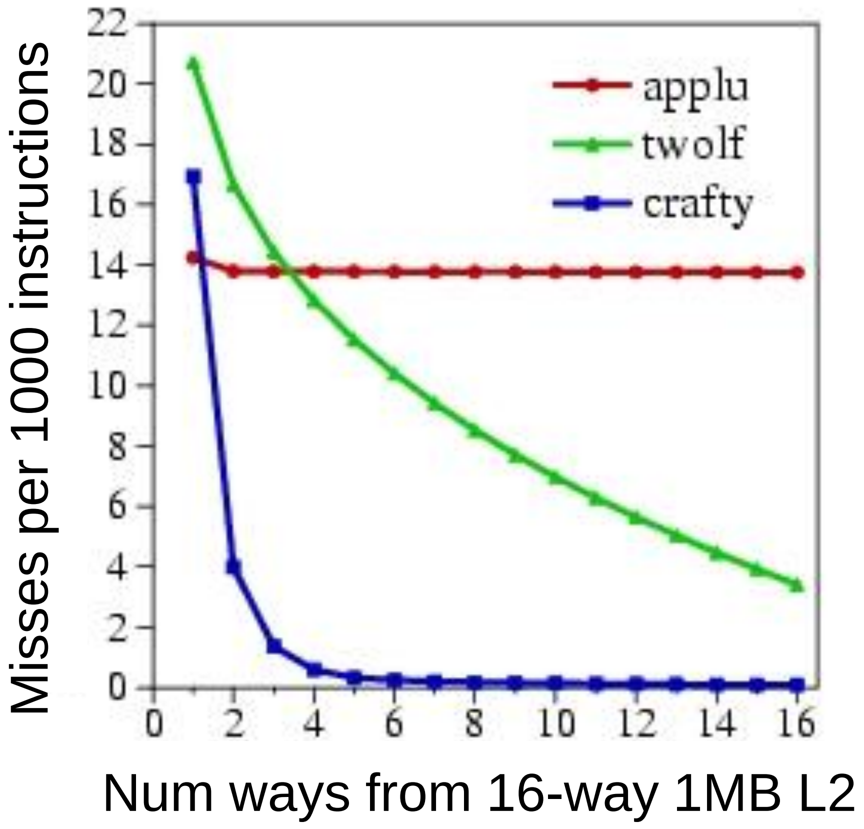
- Goal: Maximize system throughput
- Observation: Not all threads/applications benefit equally from caching → simple LRU replacement not good for system throughput
- Idea: Allocate more cache space to applications that obtain the most benefit from more space

- 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

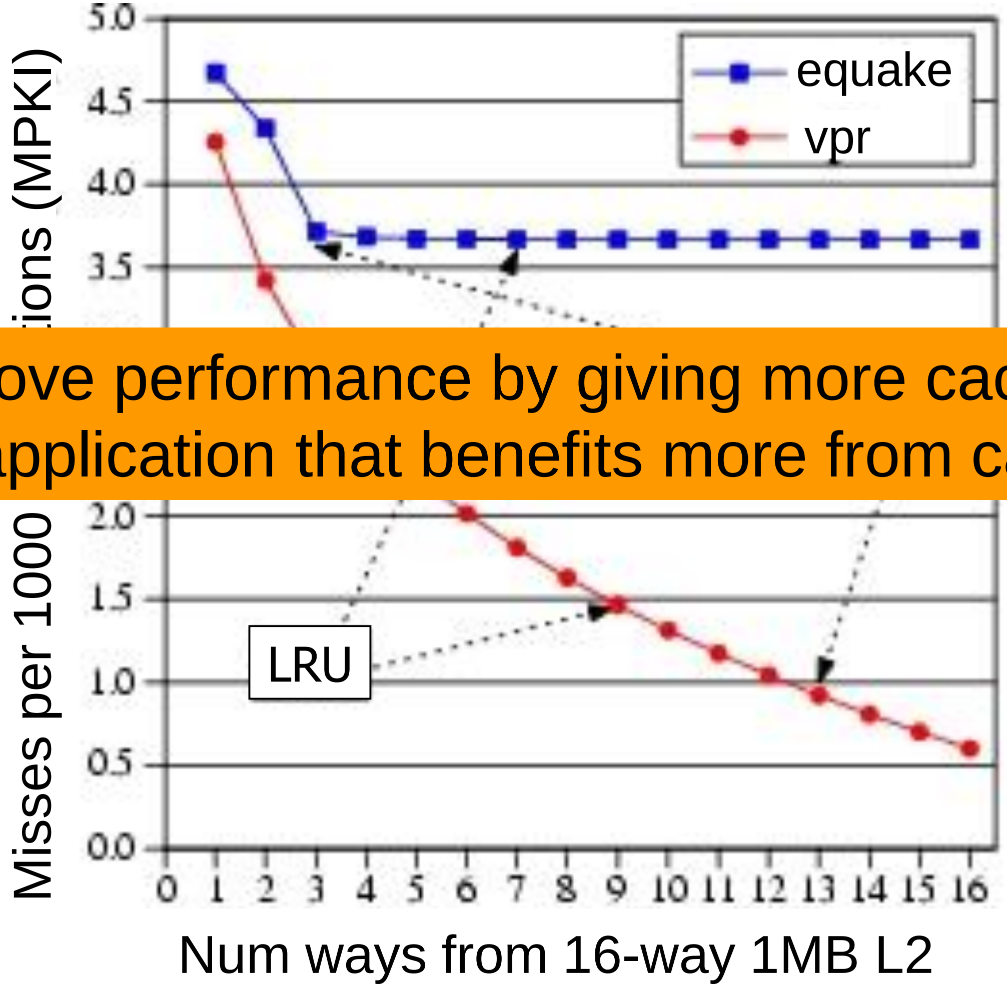


Low Utility

High Utility

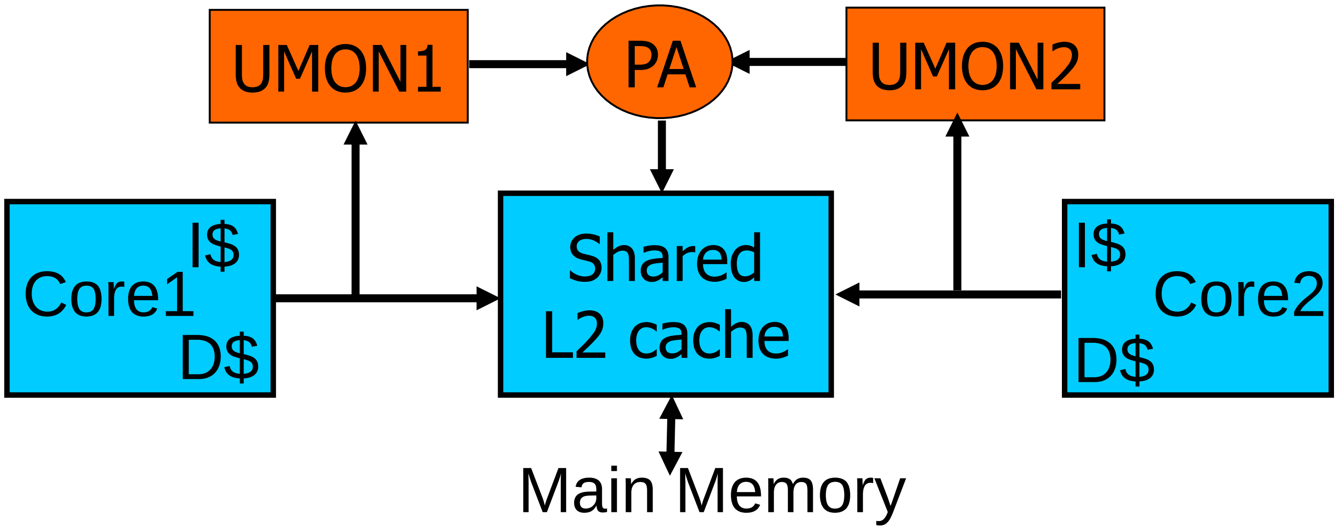
Saturating Utility

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)



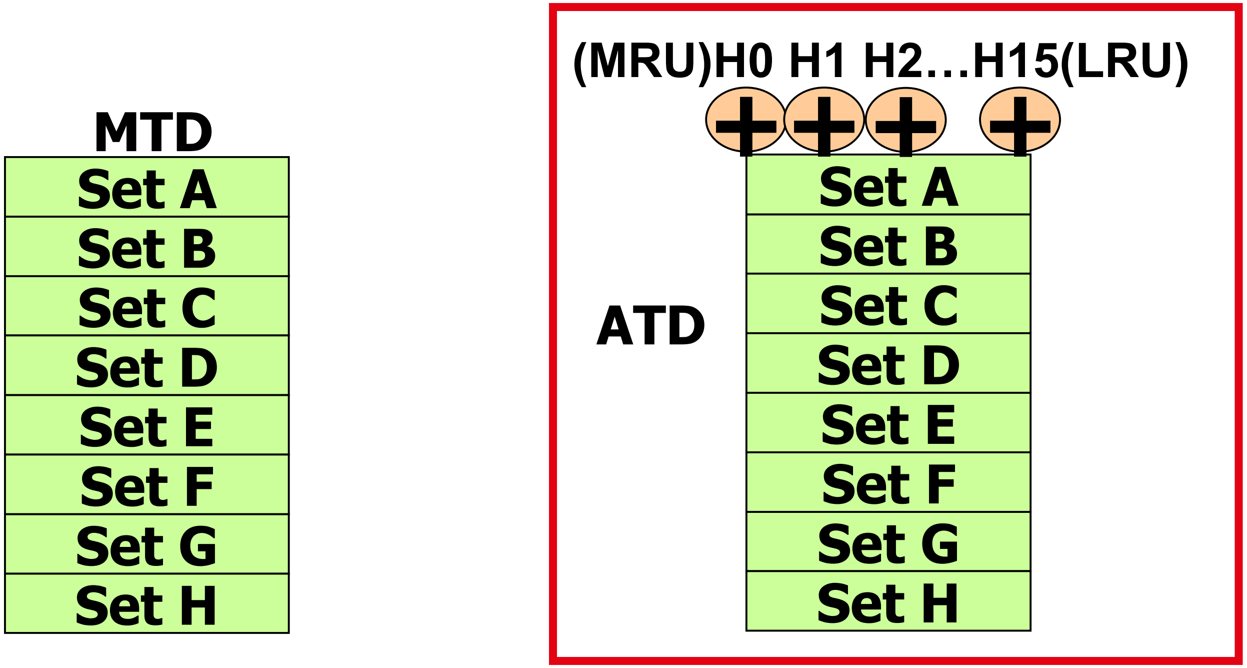
Three components:

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

Utility Monitors

- For each core, simulate LRU policy using ATD
- 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



MTD

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

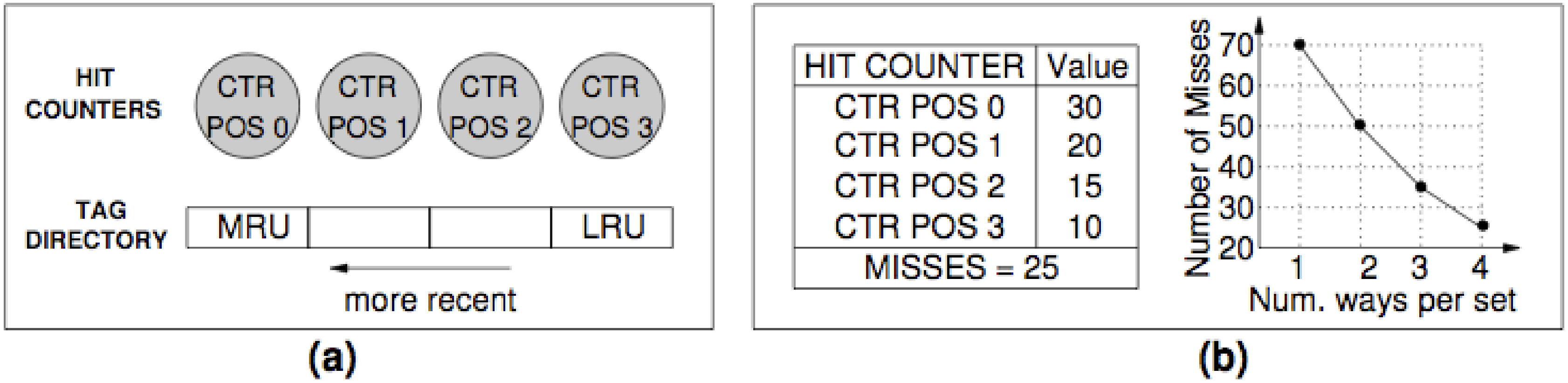
(MRU)H0 H1 H2…H15(LRU)

+ + + +

ATD

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

Utility Monitors

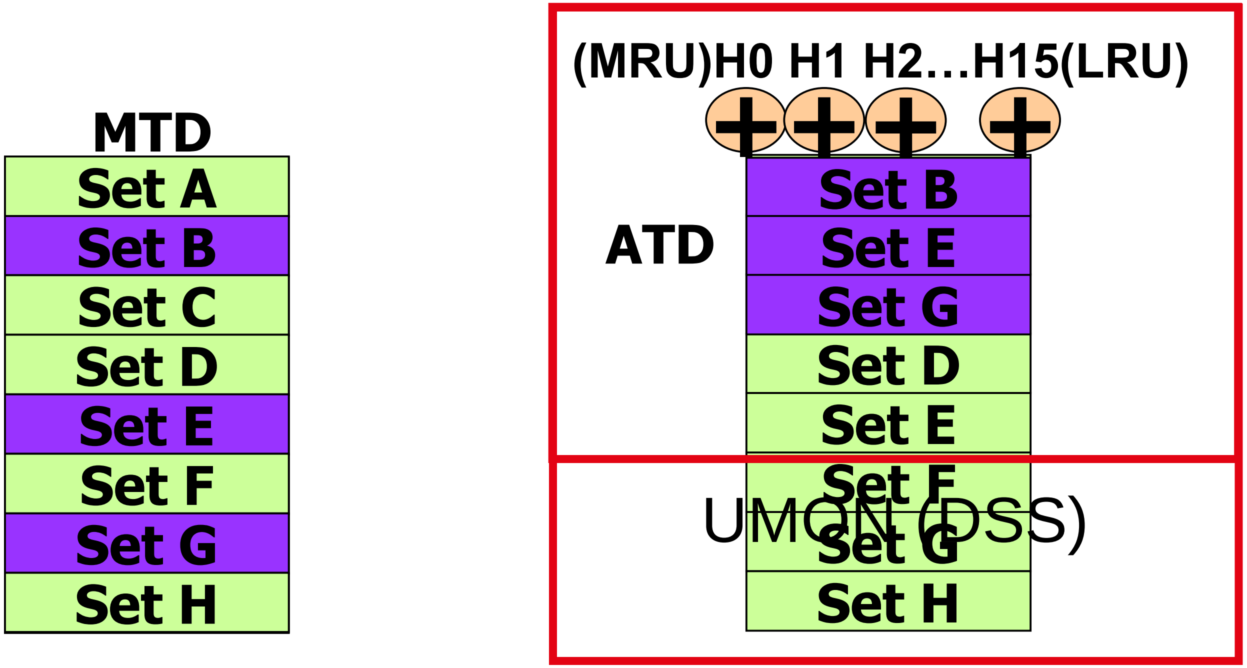


HIT COUNTER	Value
CTR POS 0	30
CTR POS 1	20
CTR POS 2	15
CTR POS 3	10
MISSES = 25	

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
- Dynamic Set Sampling reduces overhead [Qureshi, ISCA'06]
- 32 sets sufficient (analytical bounds)
- Storage < 2kB/UMON



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

Partitioning Algorithm

- Evaluate all possible partitions and select the best

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

  $\text{Hits}_{\text{core1}} = (H_0 + H_1 + \ldots + H_{a-1})$ ---- from UMON1
  $\text{Hits}_{\text{core2}} = (H_0 + H_1 + \ldots + H_{16-a-1})$ ---- from UMON2

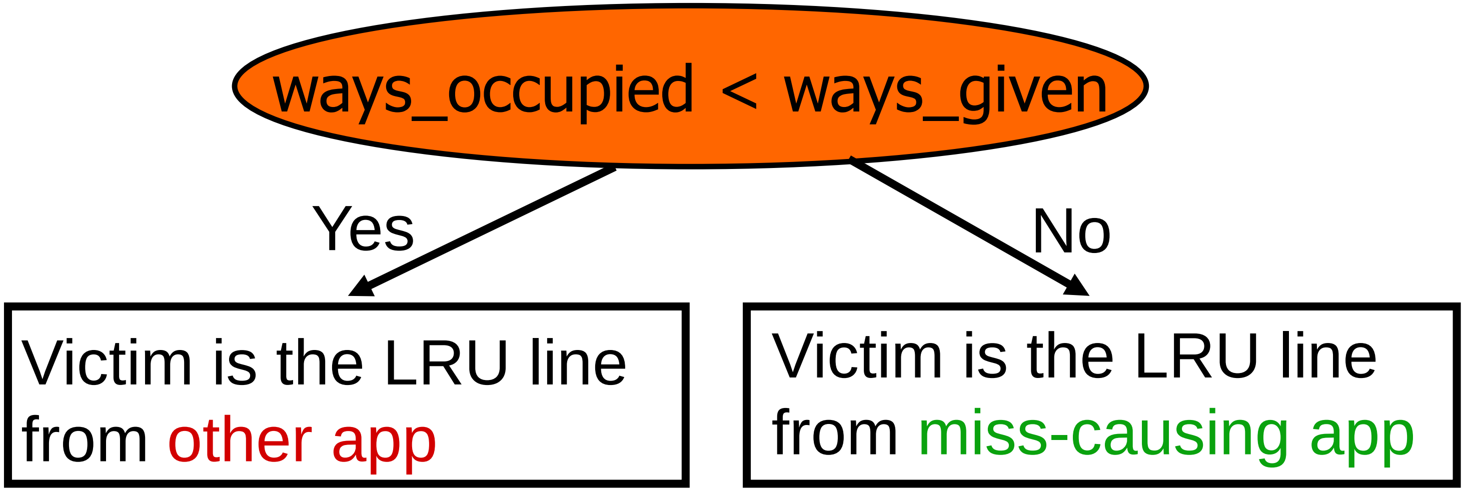
- Select a that maximizes $(\text{Hits}_{\text{core1}} + \text{Hits}_{\text{core2}})$

- Partitioning done once every 5 million cycles

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 = $IPC_1/SingleIPC_1 + IPC_2/SingleIPC_2$

   ➔ correlates with reduction in execution time

2. Throughput

   ➔ perf = $IPC_1 + IPC_2$

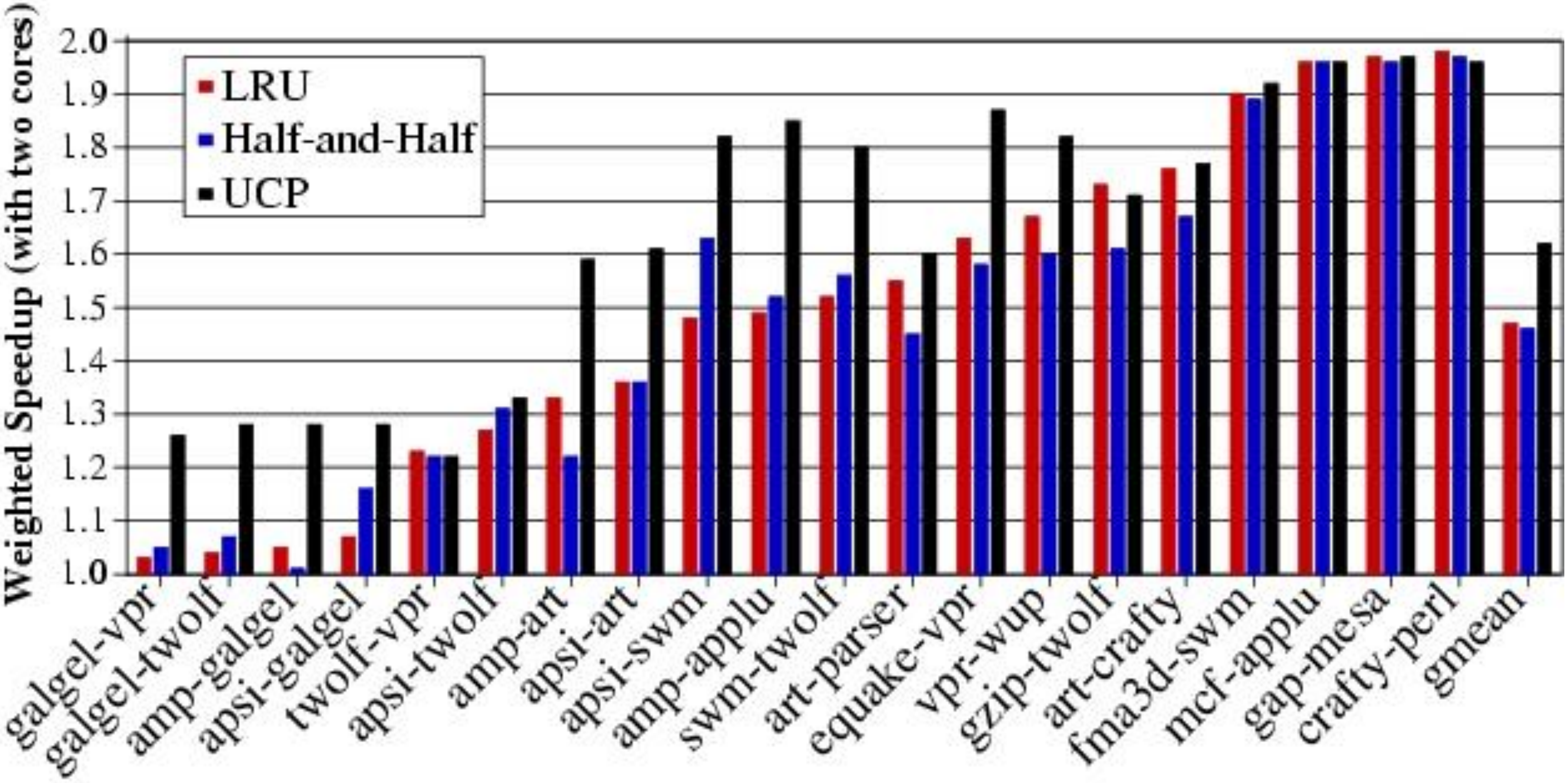
   ➔ can be unfair to low-IPC application

3. Hmean-fairness

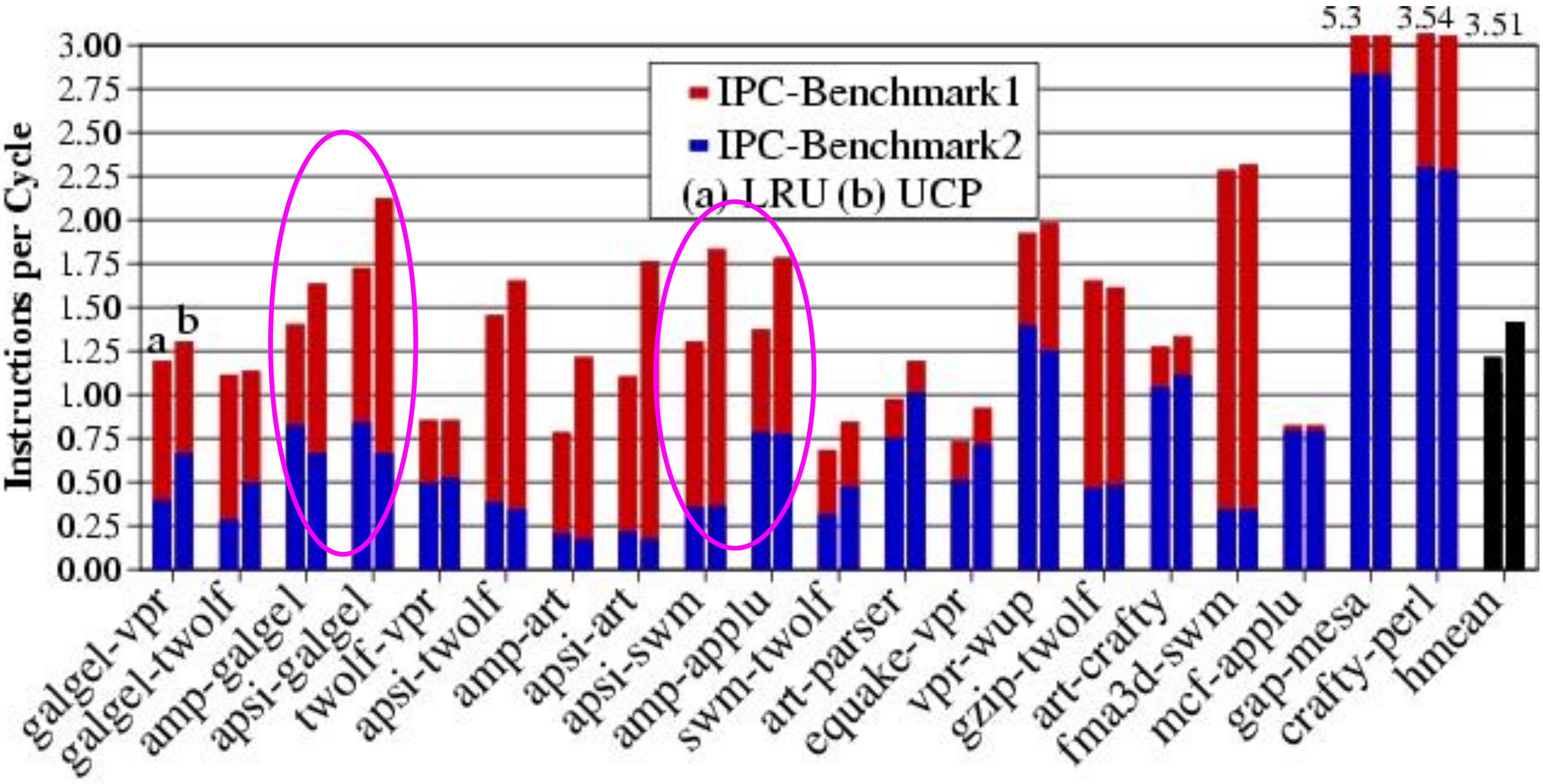
   ➔ perf = hmean($IPC_1/SingleIPC_1$, $IPC_2/SingleIPC_2$)

   ➔ 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 to many cores
- 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

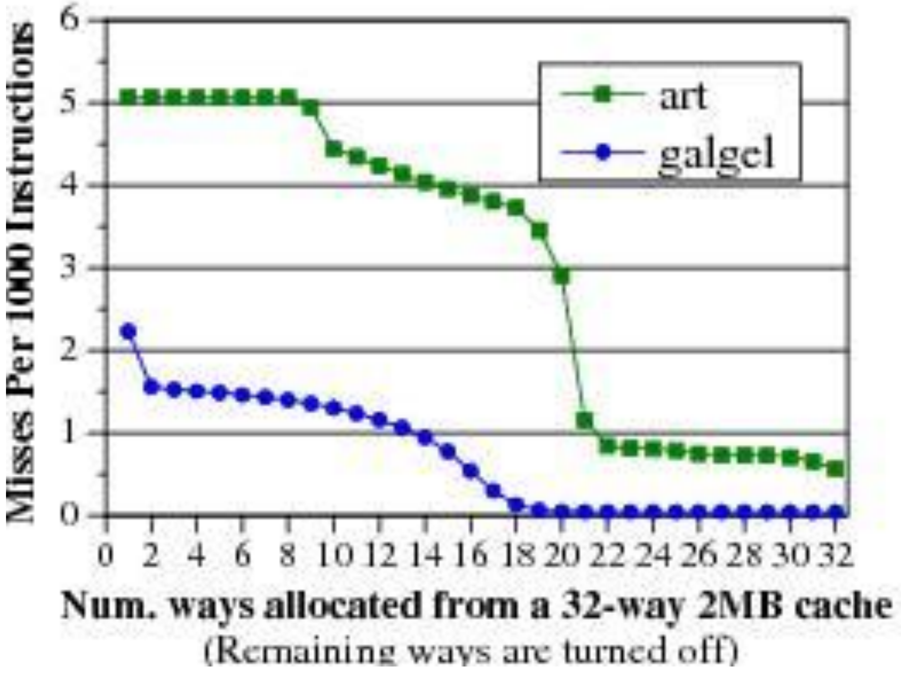
- Problem NP hard → need scalable partitioning algorithm

Greedy Algorithm [Stone+ ToC ’92]

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

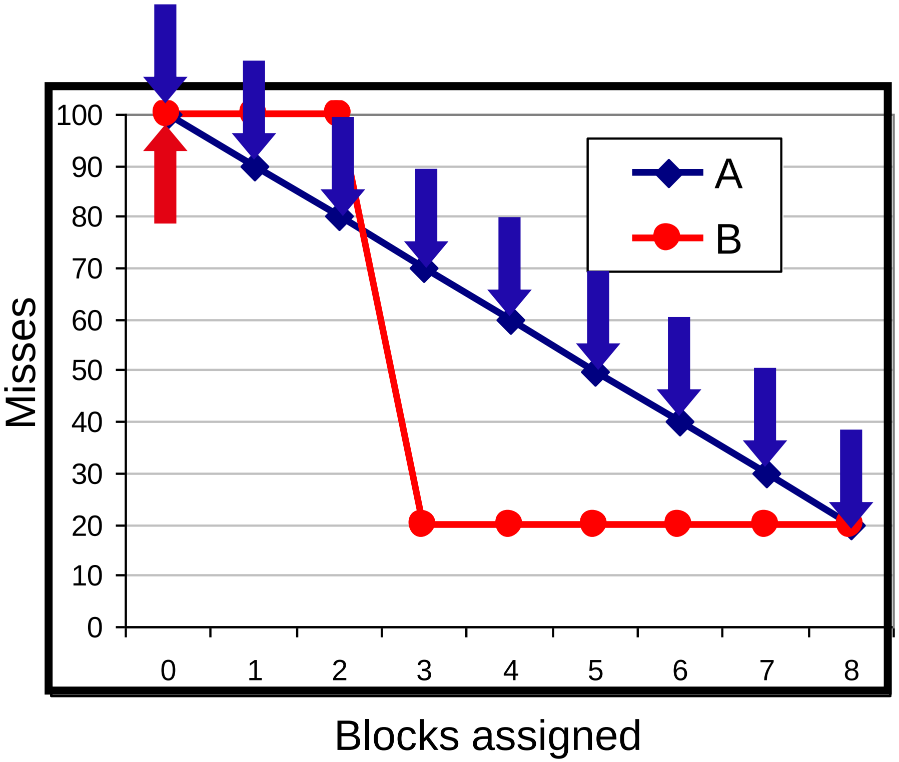
- Optimal partitioning when utility curves are convex

- Pathological behavior for non-convex curves



Stone et al., “Optimal Partitioning of Cache Memory,” IEEE ToC 1992.

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

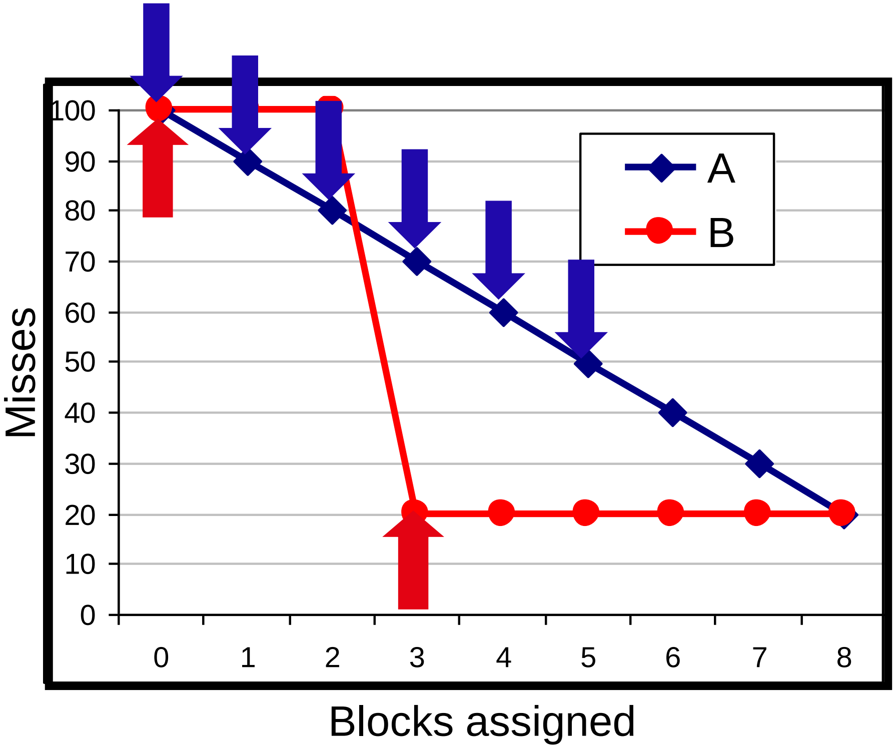
- Marginal Utility (MU) = Utility per cache resource
  - $MU_a^b = U_a^b/(b-a)$

- GA considers MU for 1 block.
- LA (Lookahead Algorithm) 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 until all blocks are 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

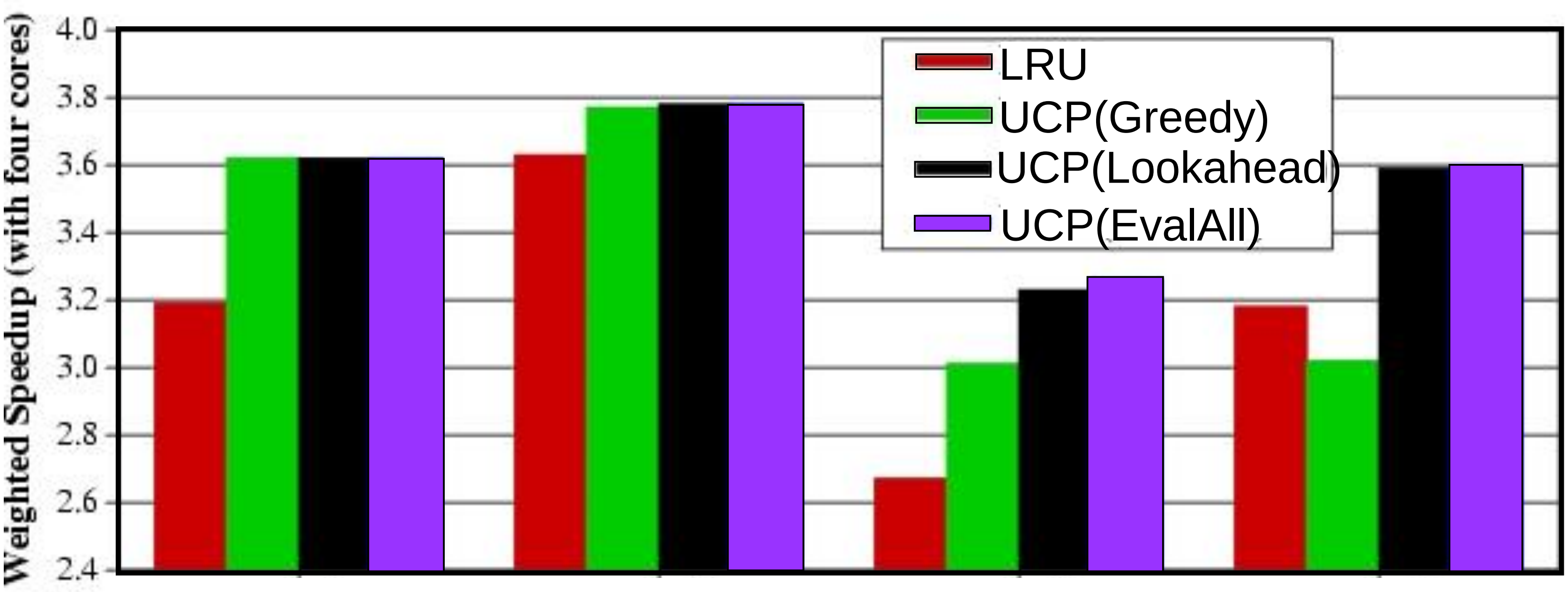
A gets 1 block

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

Time complexity ≈ ways^2/2 (512 ops for 32-ways)

UCP Results

Four cores sharing a 2MB 32-way L2



Mix1
(gap-applu-apsi-gzp)

Mix2
(swm-glg-mesa-prl)

Mix3
(mcf-applu-art-vrtx)

Mix4
(mcf-art-eqk-wupw)

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?

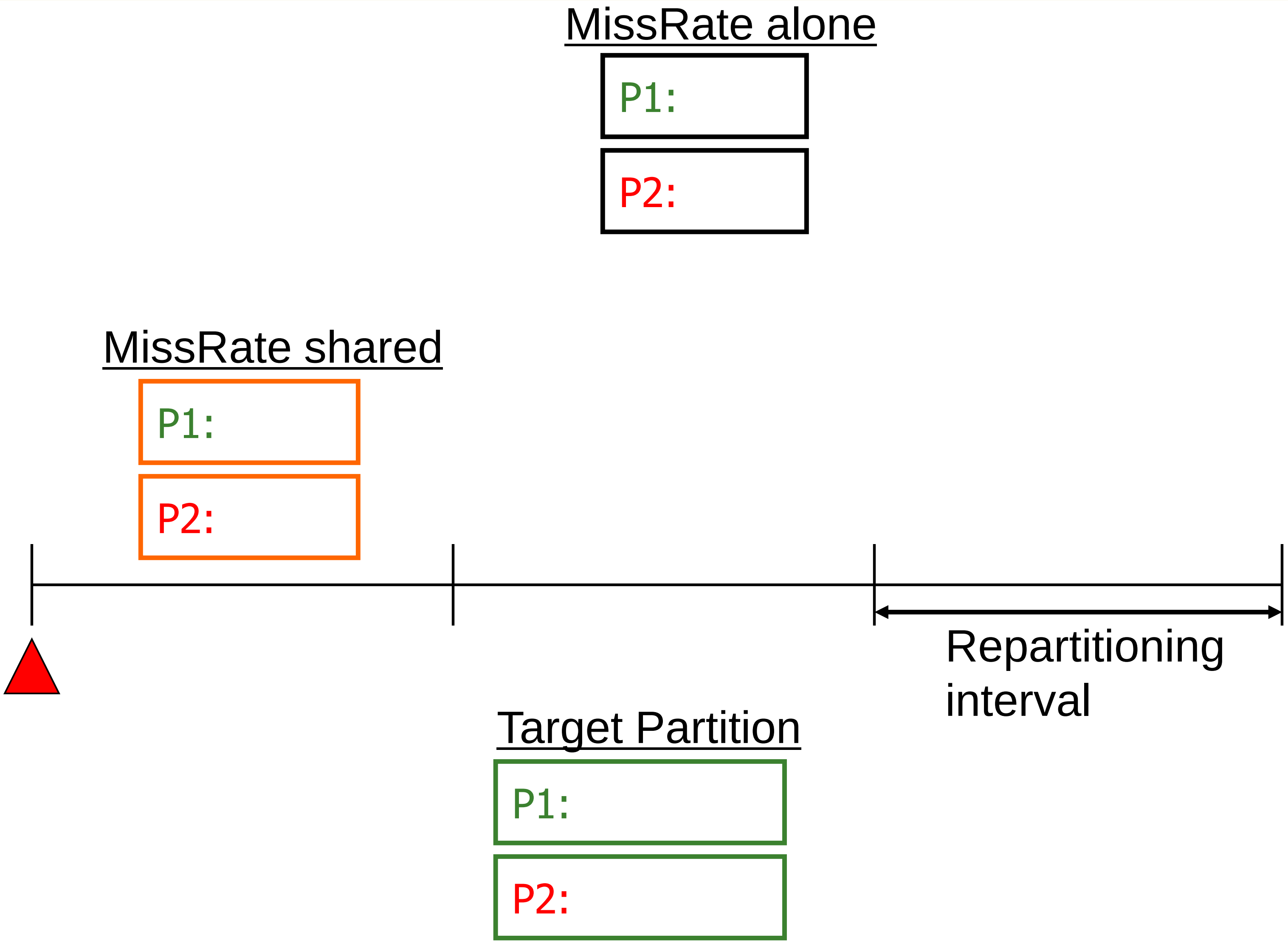
  - What if past behavior is not a good predictor of utility?

Fair Shared Cache Partitioning

- Goal: Equalize the slowdowns of multiple threads sharing the cache
- Idea: Dynamically estimate slowdowns due to sharing and assign cache blocks to balance slowdowns
  - Approximate slowdown with change in miss rate

- Kim et al., "Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture," PACT 2004.

Dynamic Fair Caching Algorithm



MissRate alone

P1:

P2:

MissRate shared

P1:

P2:

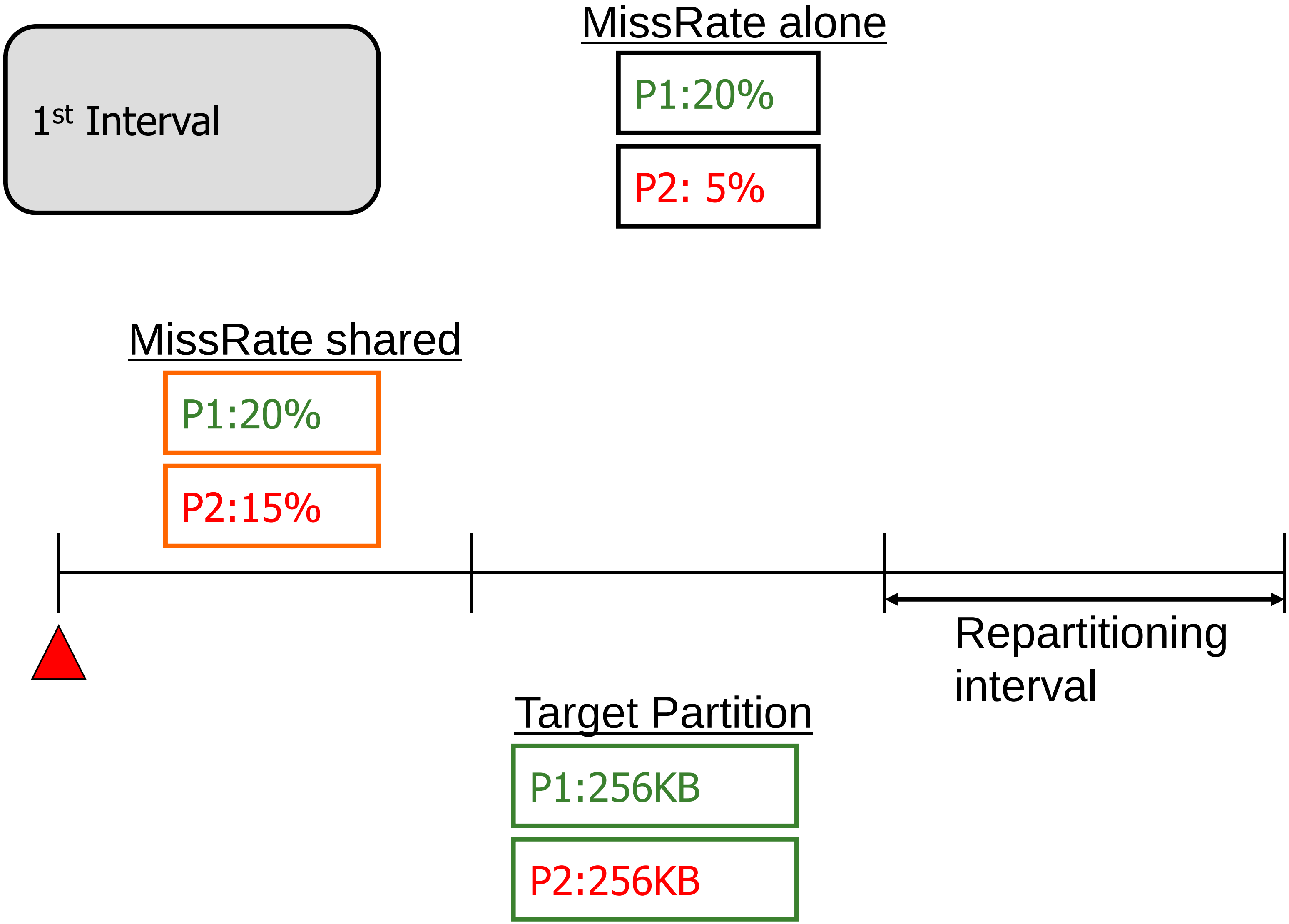
Repartitioning interval

Target Partition

P1:

P2:

Dynamic Fair Caching Algorithm



1st Interval

MissRate alone

P1:20%

P2: 5%

MissRate shared

P1:20%

P2:15%

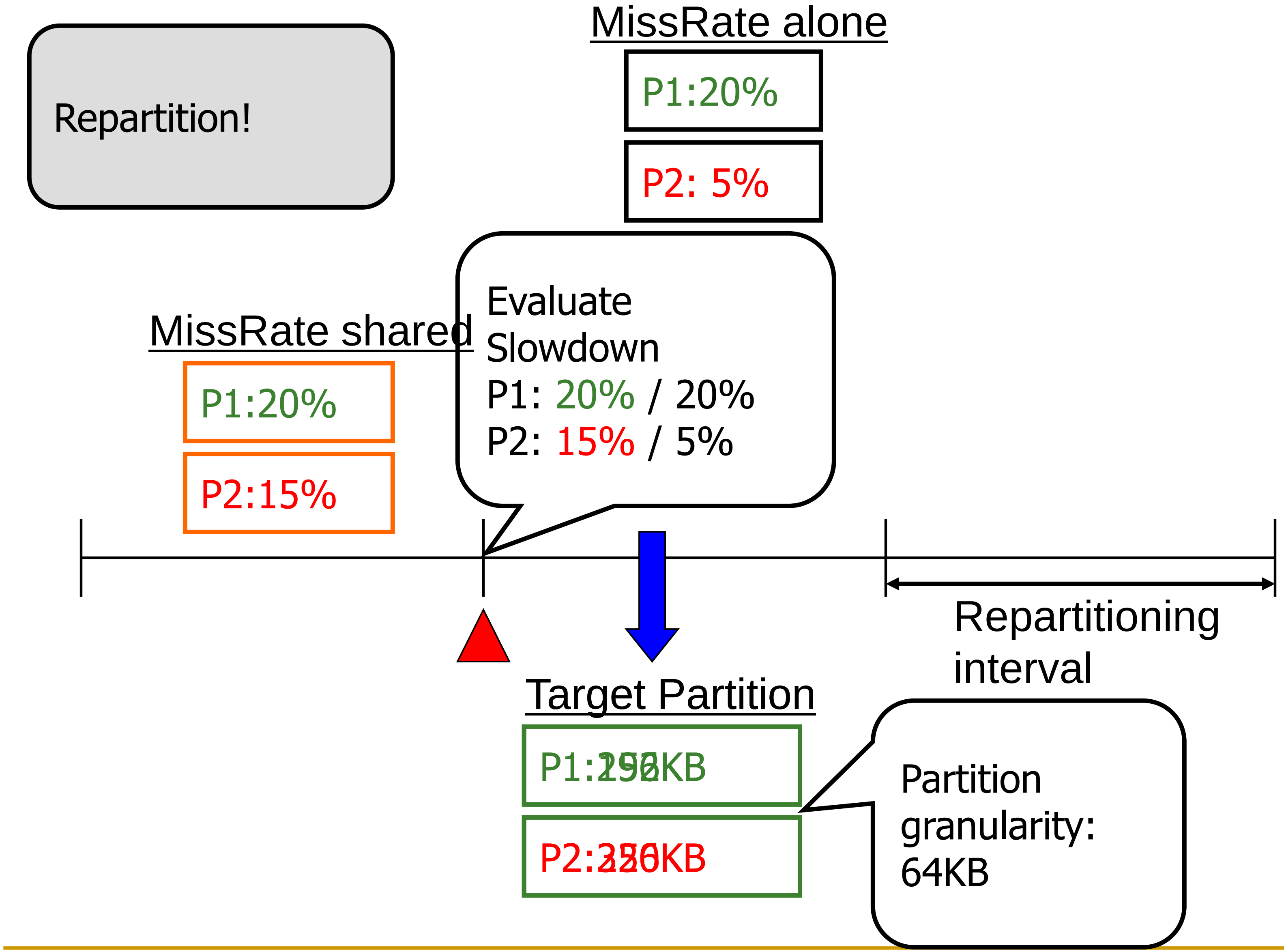
Repartitioning interval

Target Partition

P1:256KB

P2:256KB

Dynamic Fair Caching Algorithm



Repartition!

MissRate alone

P1:20%

P2: 5%

MissRate shared

P1:20%

P2:15%

Evaluate Slowdown
P1: 20% / 20%
P2: 15% / 5%

Repartitioning interval

Target Partition

P1:256KB
P1:192KB

P2:256KB
P2:320KB

Partition granularity: 64KB

Dynamic Fair Caching Algorithm

2nd Interval

MissRate alone

P1:20%

P2: 5%

MissRate shared

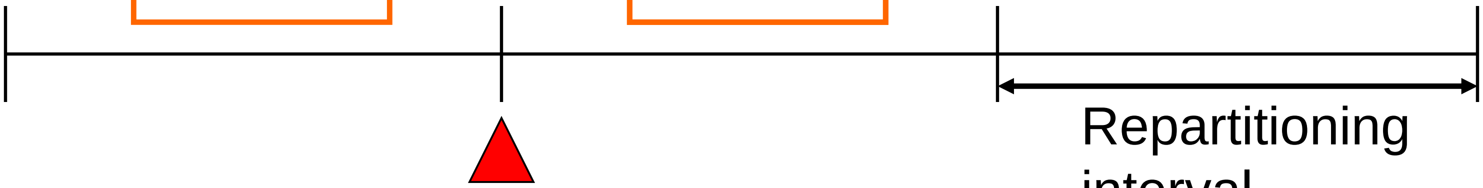
P1:20%

P2:15%

MissRate shared

P1:20%

P2:10%



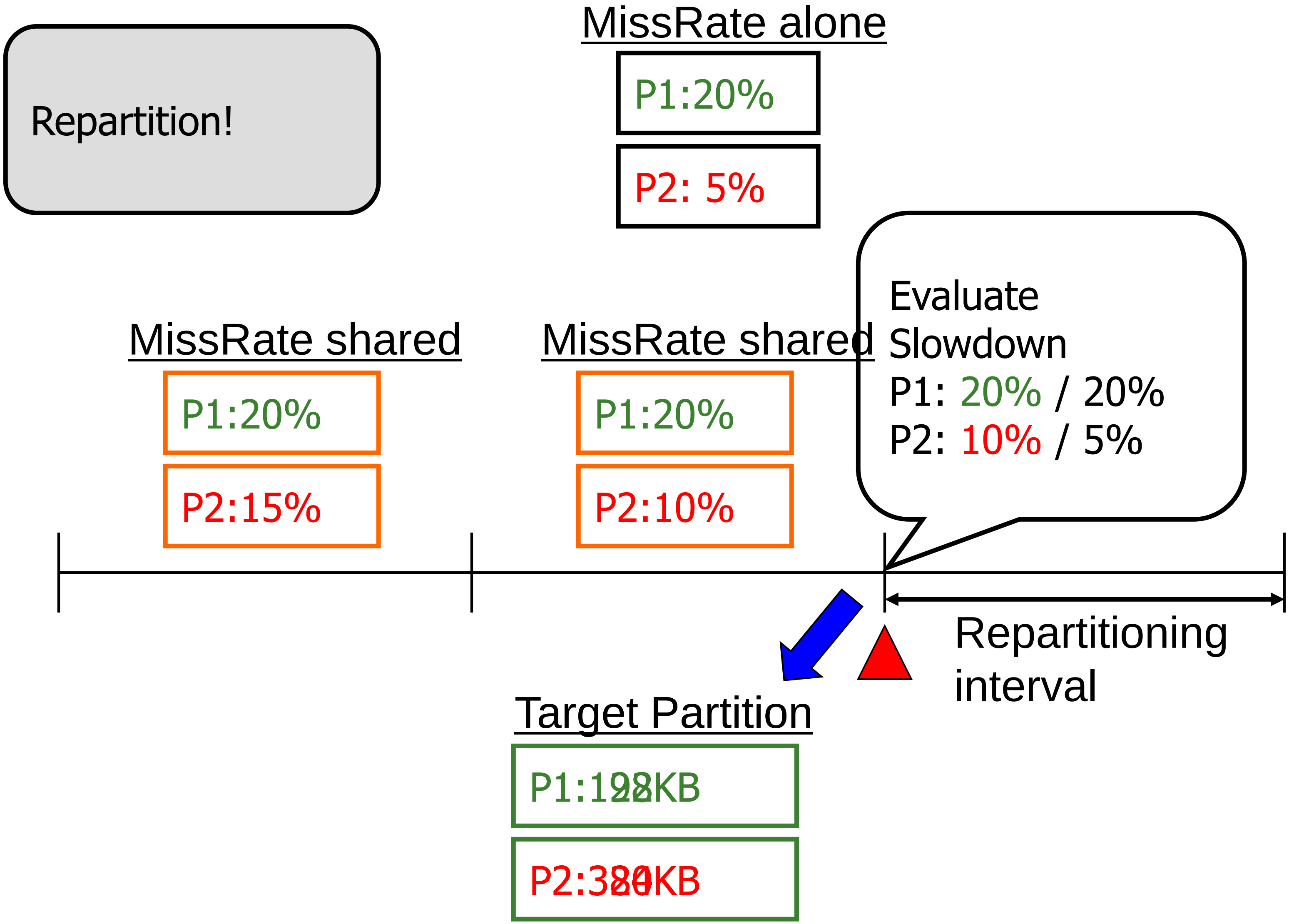
Repartitioning interval

Target Partition

P1:192KB

P2:320KB

Dynamic Fair Caching Algorithm



Repartition!

MissRate alone

P1:20%

P2: 5%

MissRate shared

P1:20%

P2:15%

MissRate shared

P1:20%

P2:10%

Evaluate Slowdown
P1: 20% / 20%
P2: 10% / 5%

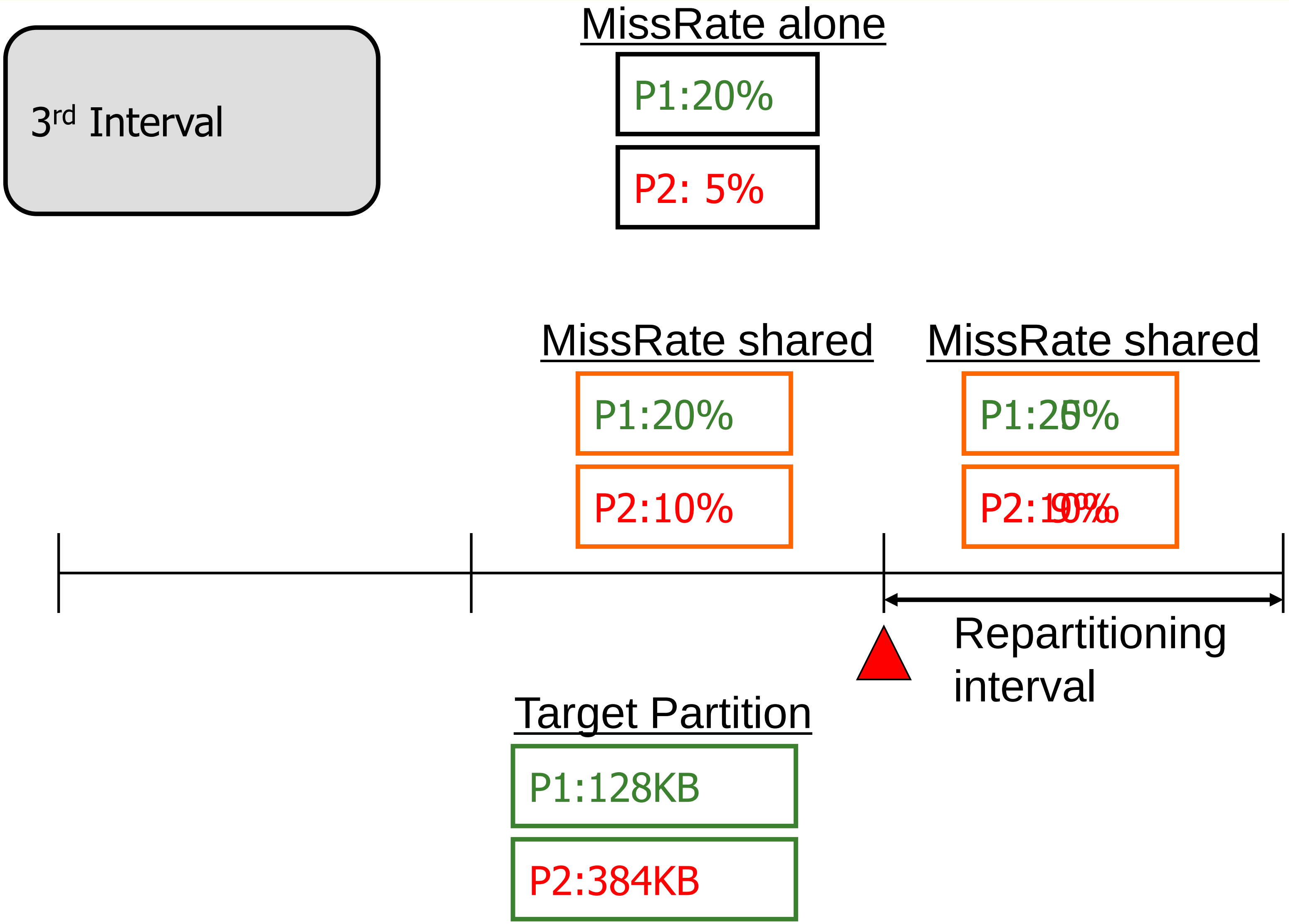
Repartitioning interval

Target Partition

P1:192KB
P1:128KB

P2:320KB
P2:384KB

Dynamic Fair Caching Algorithm



3rd Interval

MissRate alone

P1:20%

P2: 5%

MissRate shared

P1:20%

P2:10%

MissRate shared

P1:20%
P1:25%

P2:10%
P2:9%

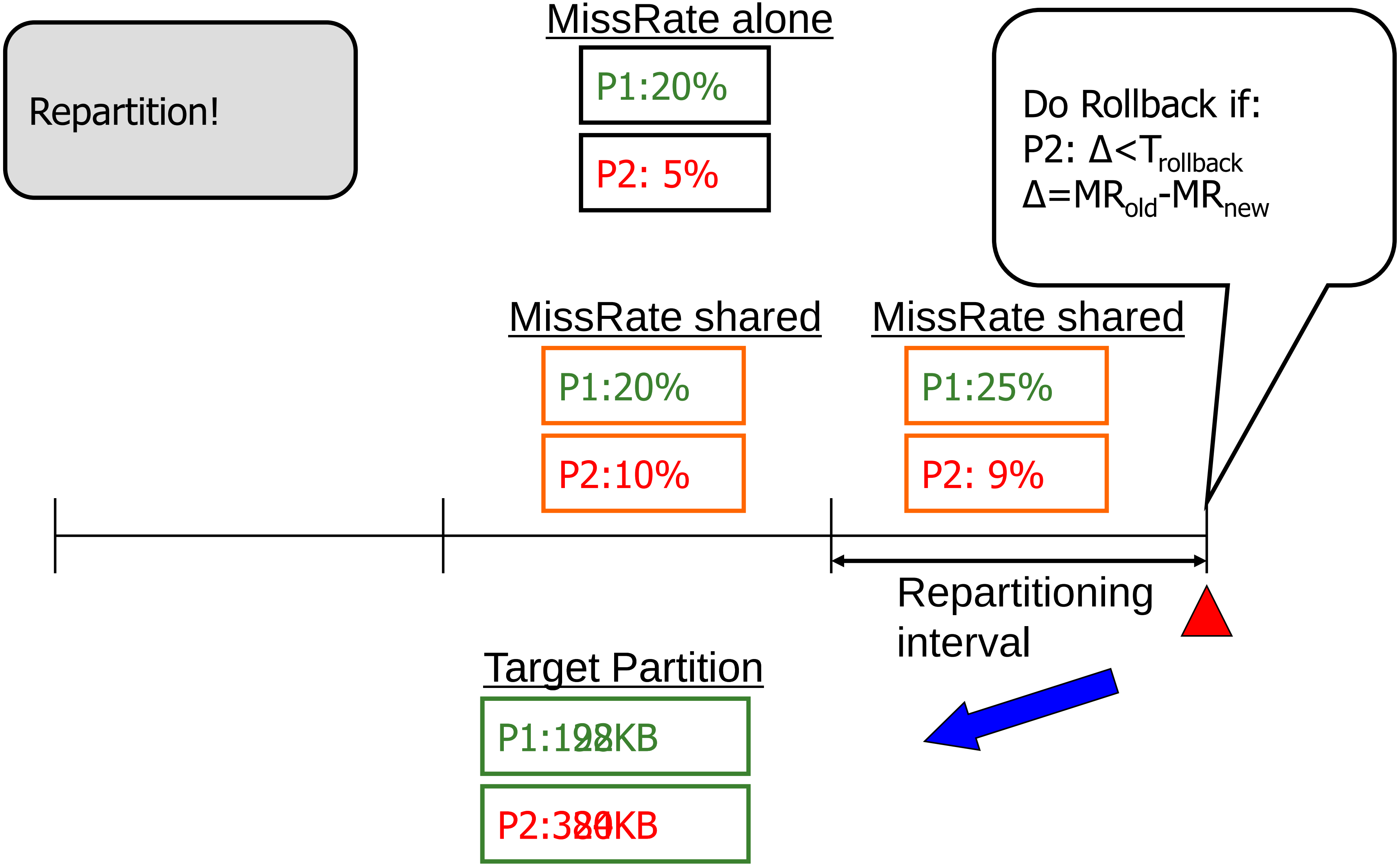
Repartitioning interval

Target Partition

P1:128KB

P2:384KB

Dynamic Fair Caching Algorithm



Repartition!

MissRate alone

P1:20%
P2: 5%

Do Rollback if:
P2: Δ<$T_{rollback}$
Δ=MR_{old}-MR_{new}

MissRate shared

P1:20%
P2:10%

MissRate shared

P1:25%
P2: 9%

Repartitioning interval

Target Partition

P1:192KB P1:128KB
P2:320KB P2:384KB

Advantages/Disadvantages of the Approach

- Advantages

  + Reduced starvation

  + Better average throughput

  + Block granularity partitioning

- Disadvantages and Limitations

  - Alone miss rate estimation can be incorrect

  - Scalable to many cores?

  - Is this the best (or a good) fairness metric?

  - Does this provide performance isolation in cache?

Software-Based Shared Cache Partitioning

Software-Based Shared Cache Management

- Assume no hardware support (demand based cache sharing, i.e. LRU replacement)
- How can the OS best utilize the cache?

- Cache sharing aware thread scheduling
  - Schedule workloads that "play nicely" together in the cache
    - E.g., working sets together fit in the cache
    - Requires static/dynamic profiling of application behavior
    - Fedorova et al., "Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler," PACT 2007.

- Cache sharing aware page coloring
  - Dynamically monitor miss rate over an interval and change virtual to physical mapping to minimize miss rate
    - Try out different partitions

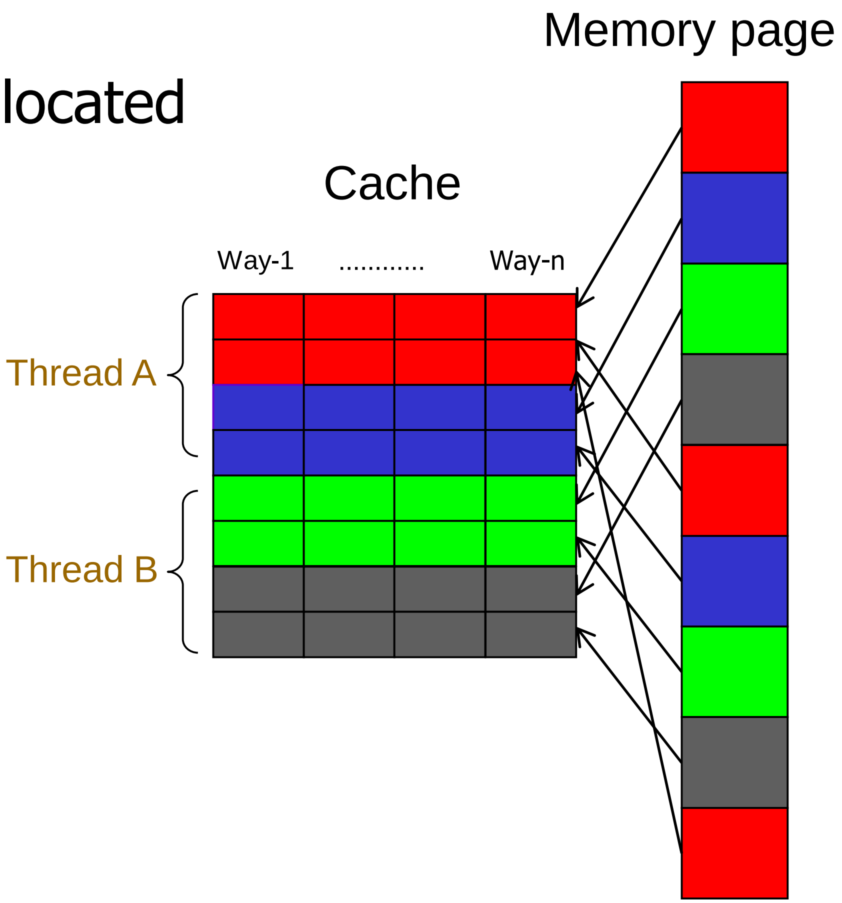
OS Based Cache Partitioning

- Lin et al., "Gaining Insights into Multi-Core Cache Partitioning: Bridging the Gap between Simulation and Real Systems," HPCA 2008.
- Cho and Jin, "Managing Distributed, Shared L2 Caches through OS-Level Page Allocation," MICRO 2006.

- Static cache partitioning
  - Predetermines the amount of cache blocks allocated to each program at the beginning of its execution
  - Divides shared cache to multiple regions and partitions cache regions through OS page address mapping
- Dynamic cache partitioning
  - Adjusts cache quota among processes dynamically
  - Page re-coloring
  - Dynamically changes processes' cache usage through OS page address re-mapping

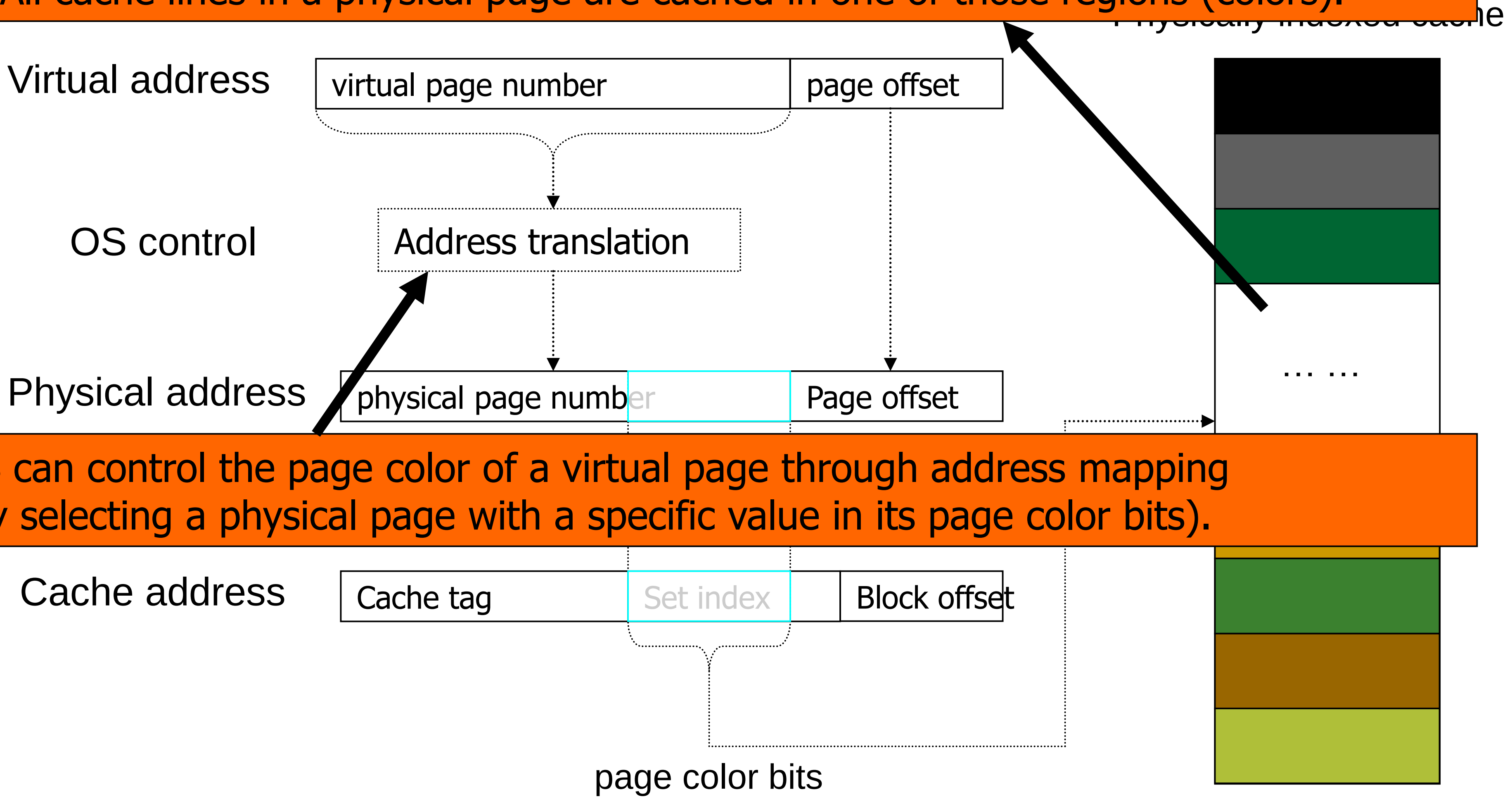
Page Coloring

- Physical memory divided into colors
- Colors map to different cache sets
- Cache partitioning
  - Ensure two threads are allocated pages of different colors



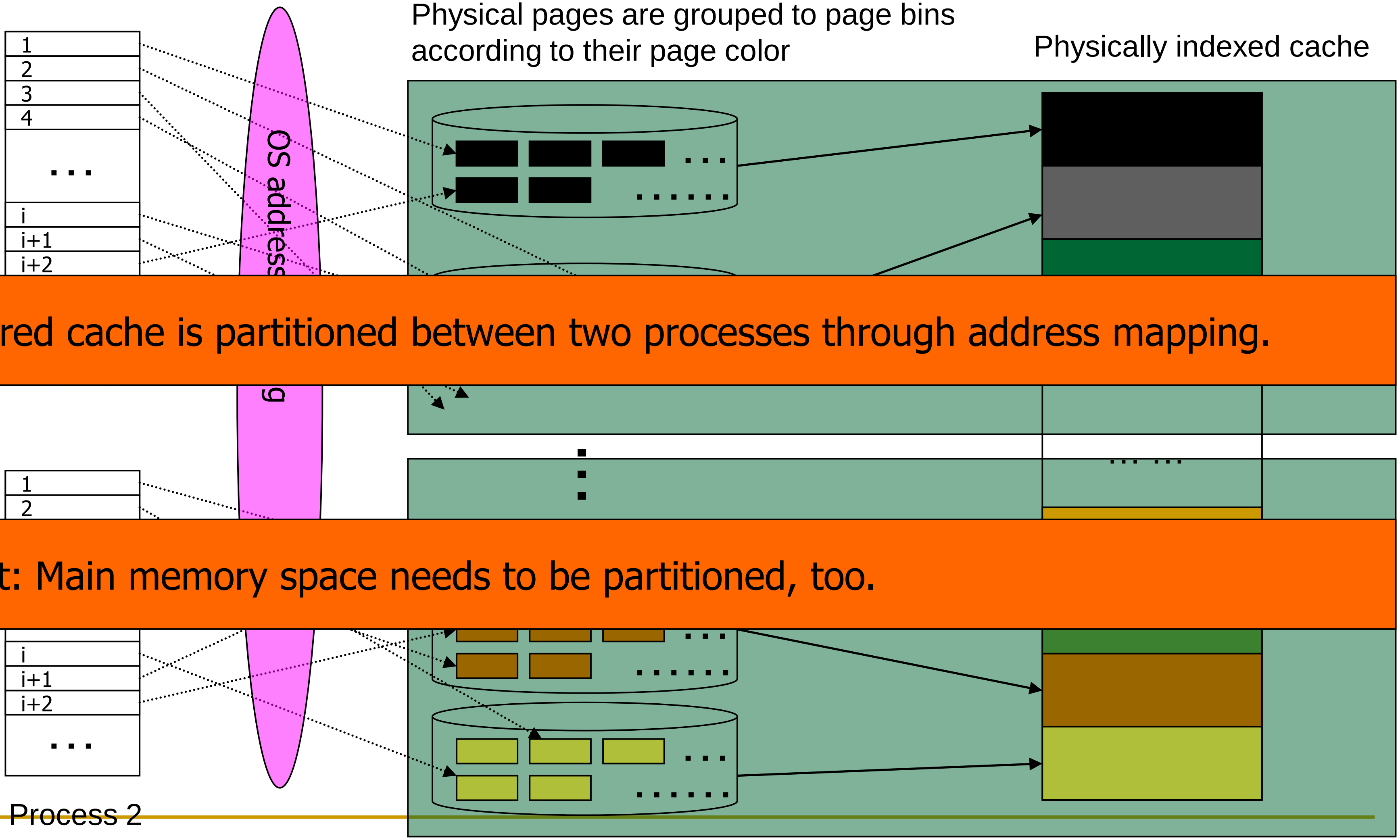
Page Coloring

- Physically indexed caches are divided into multiple regions (colors).
- All cache lines in a physical page are cached in one of those regions (colors).



OS can control the page color of a virtual page through address mapping
(by selecting a physical page with a specific value in its page color bits).

Static Cache Partitioning using Page Coloring



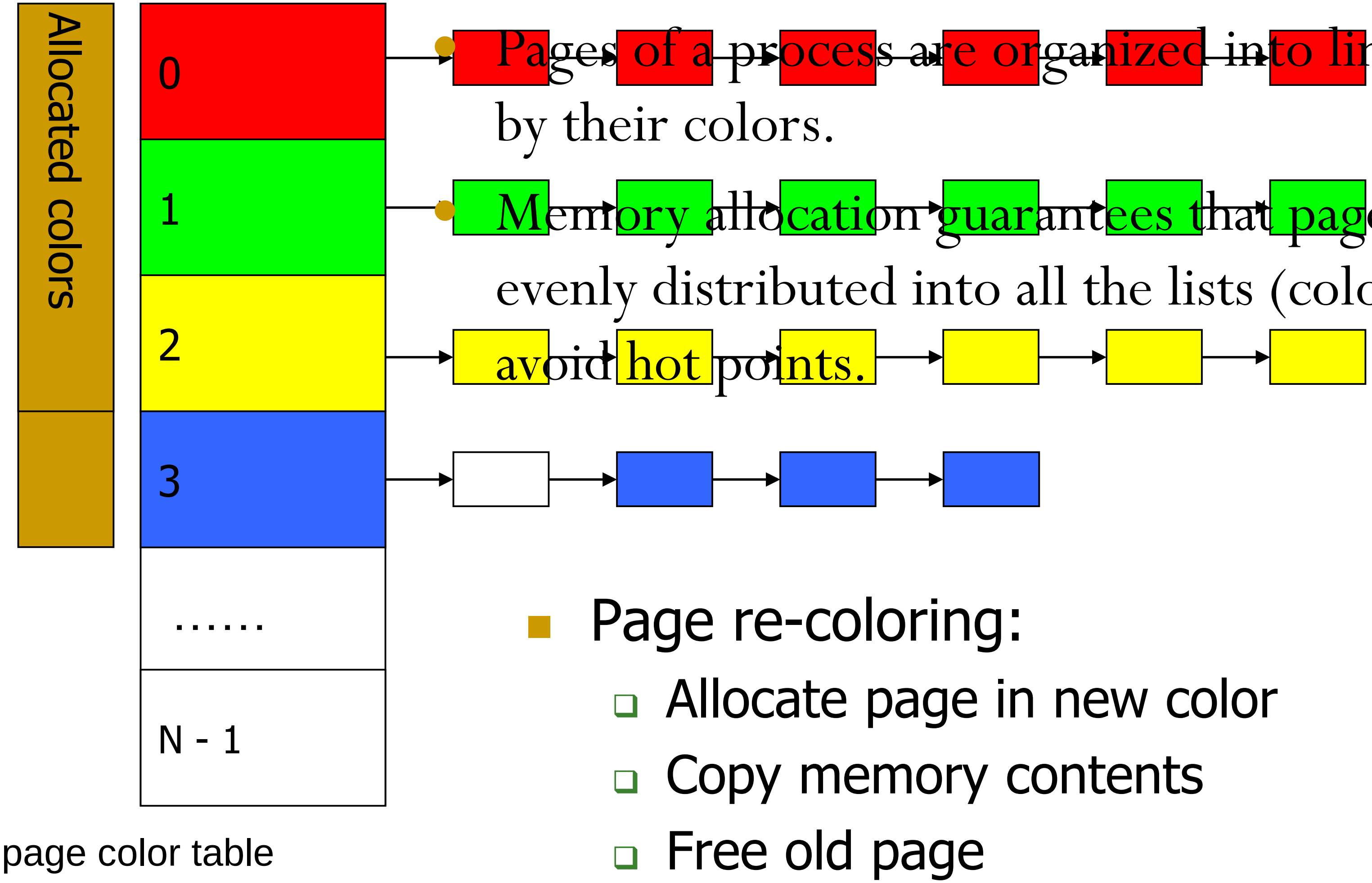
Physical pages are grouped to page bins according to their page color

Physically indexed cache

Shared cache is partitioned between two processes through address mapping.

Cost: Main memory space needs to be partitioned, too.

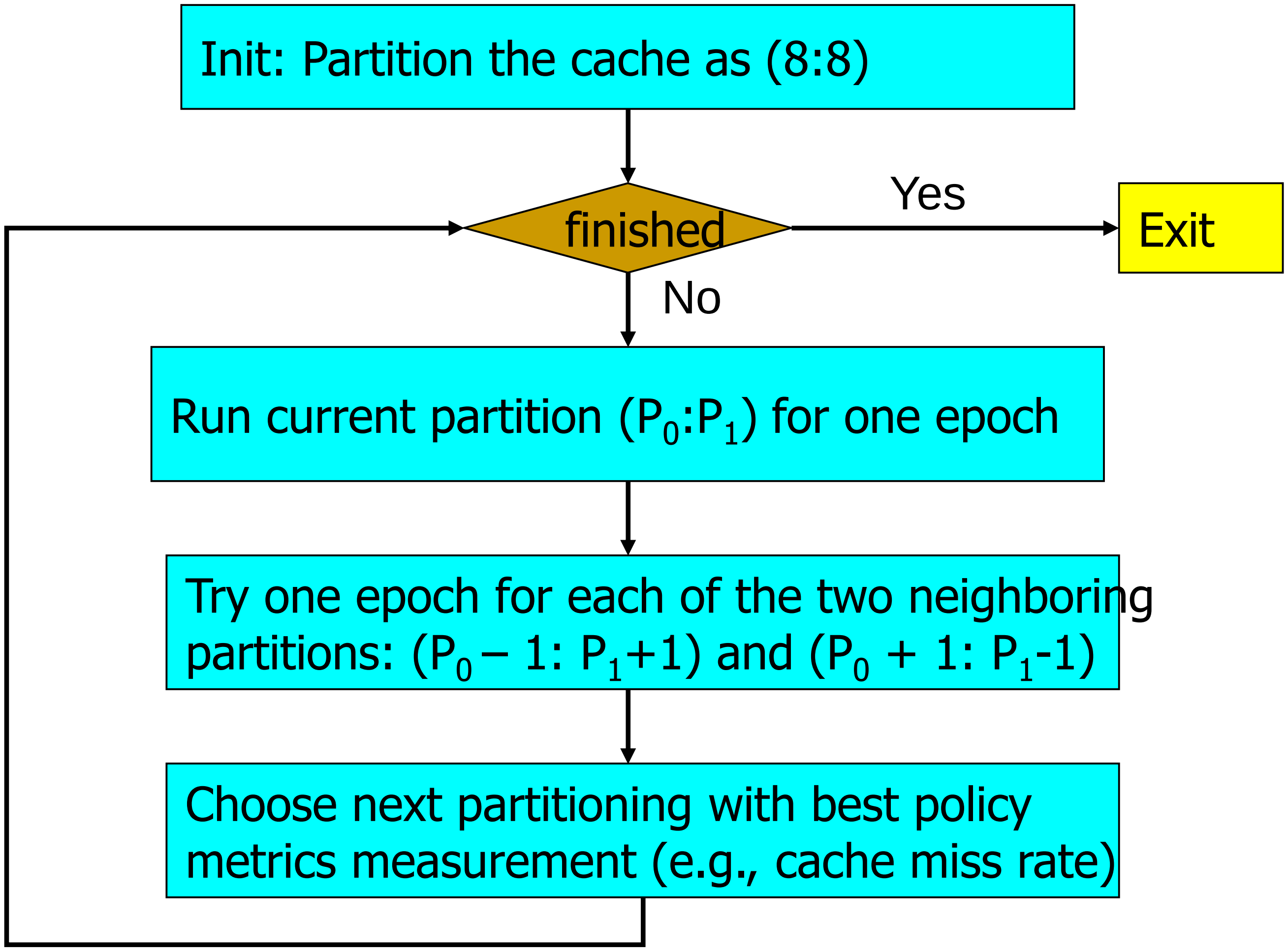
Dynamic Cache Partitioning via Page Re-Coloring



- Pages of a process are organized into linked lists by their colors.
- Memory allocation guarantees that pages are evenly distributed into all the lists (colors) to avoid hot points.

- Page re-coloring:
  - Allocate page in new color
  - Copy memory contents
  - Free old page

Dynamic Partitioning in a Dual-Core System



Init: Partition the cache as (8:8)

finished

Yes

Exit

No

Run current partition (P_0:P_1) for one epoch

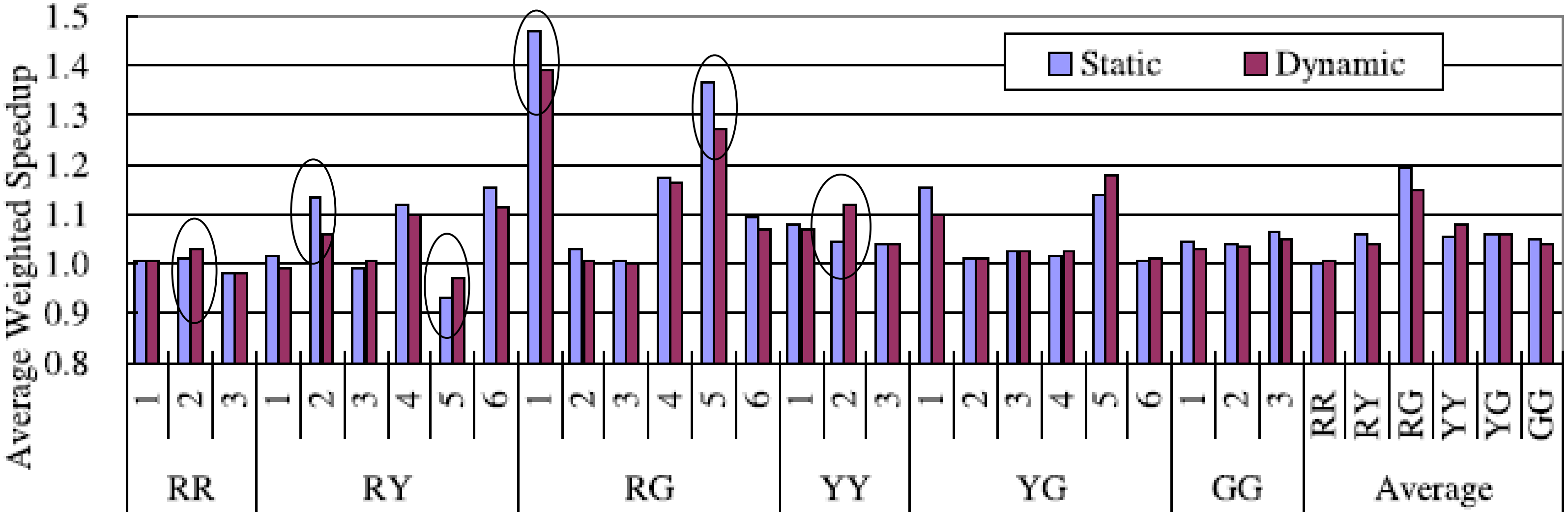
Try one epoch for each of the two neighboring partitions: (P_0 – 1: P_1+1) and (P_0 + 1: P_1-1)

Choose next partitioning with best policy metrics measurement (e.g., cache miss rate)

Experimental Environment

- Dell PowerEdge1950
  - Two-way SMP, Intel dual-core Xeon 5160
  - Shared 4MB L2 cache, 16-way
  - 8GB Fully Buffered DIMM
- Red Hat Enterprise Linux 4.0
  - 2.6.20.3 kernel
  - Performance counter tools from HP (Pfmon)
  - Divide L2 cache into 16 colors

Performance – Static & Dynamic



- Aim to minimize combined miss rate
- For RG-type, and some RY-type:
  - Static partitioning outperforms dynamic partitioning
- For RR- and RY-type, and some RY-type
  - Dynamic partitioning outperforms static partitioning

Software vs. Hardware Cache Management

- Software advantages

  + No need to change hardware

  + Easier to upgrade/change algorithm (not burned into hardware)

- Disadvantages

  - Large granularity of partitioning (page-based versus way/block)

  - Limited page colors → reduced performance per application (limited physical memory space!), reduced flexibility

  - Changing partition size has high overhead → page mapping changes

  - Adaptivity is slow: hardware can adapt every cycle (possibly)

  - Not enough information exposed to software (e.g., number of misses due to inter-thread conflict)

Private/Shared Caching

Private/Shared Caching

- Example: Adaptive spill/receive caching

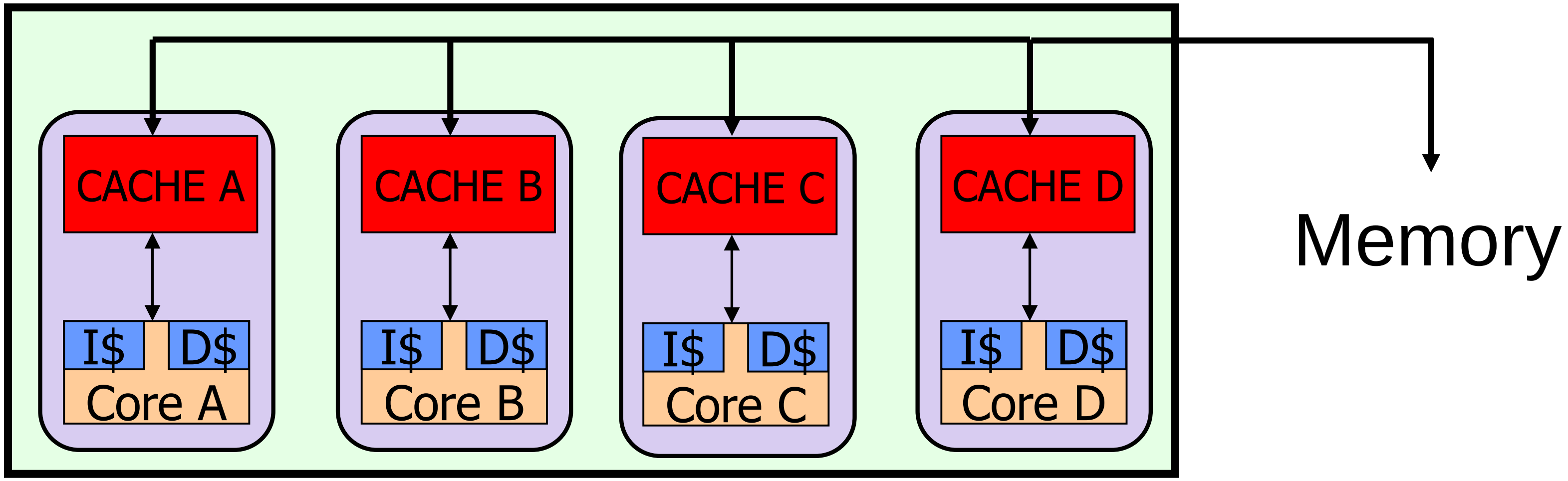
- Goal: Achieve the benefits of private caches (low latency, performance isolation) while sharing cache capacity across cores

- Idea: Start with a private cache design (for performance isolation), but dynamically steal space from other cores that do not need all their private caches
  - Some caches can spill their data to other cores' caches dynamically

- Qureshi, "Adaptive Spill-Receive for Robust High-Performance Caching in CMPs," HPCA 2009.

Revisiting Private Caches on CMP

Private caches avoid the need for shared interconnect
++ fast latency, tiled design, performance isolation

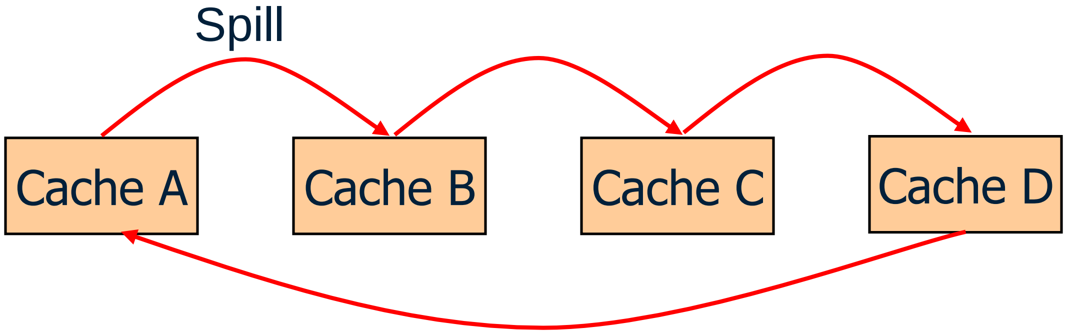


Problem: When one core needs more cache and other core has spare cache, private-cache CMPs cannot share capacity

Cache Line Spilling

Spill evicted line from one cache to neighbor cache

- Co-operative caching (CC) [Chang+ ISCA' 06]



Problem with CC:

1. Performance depends on the parameter (spill probability)
2. All caches spill as well as receive ➔ Limited improvement

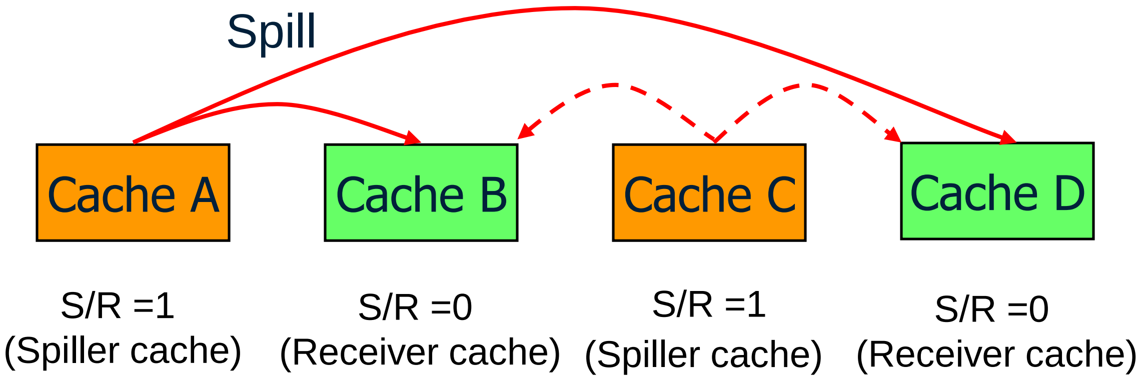
Goal: Robust High-Performance Capacity Sharing with Negligible Overhead

Chang and Sohi, "Cooperative Caching for Chip Multiprocessors," ISCA 2006.

Spill-Receive Architecture

Each Cache is either a Spiller or Receiver but not both

- Lines from spiller cache are spilled to one of the receivers
- Evicted lines from receiver cache are discarded



What is the best N-bit binary string that maximizes the performance of Spill Receive Architecture ➔ Dynamic Spill Receive (DSR)

Qureshi, "Adaptive Spill-Receive for Robust High-Performance Caching in CMPs," HPCA 2009.

Efficient Cache Utilization

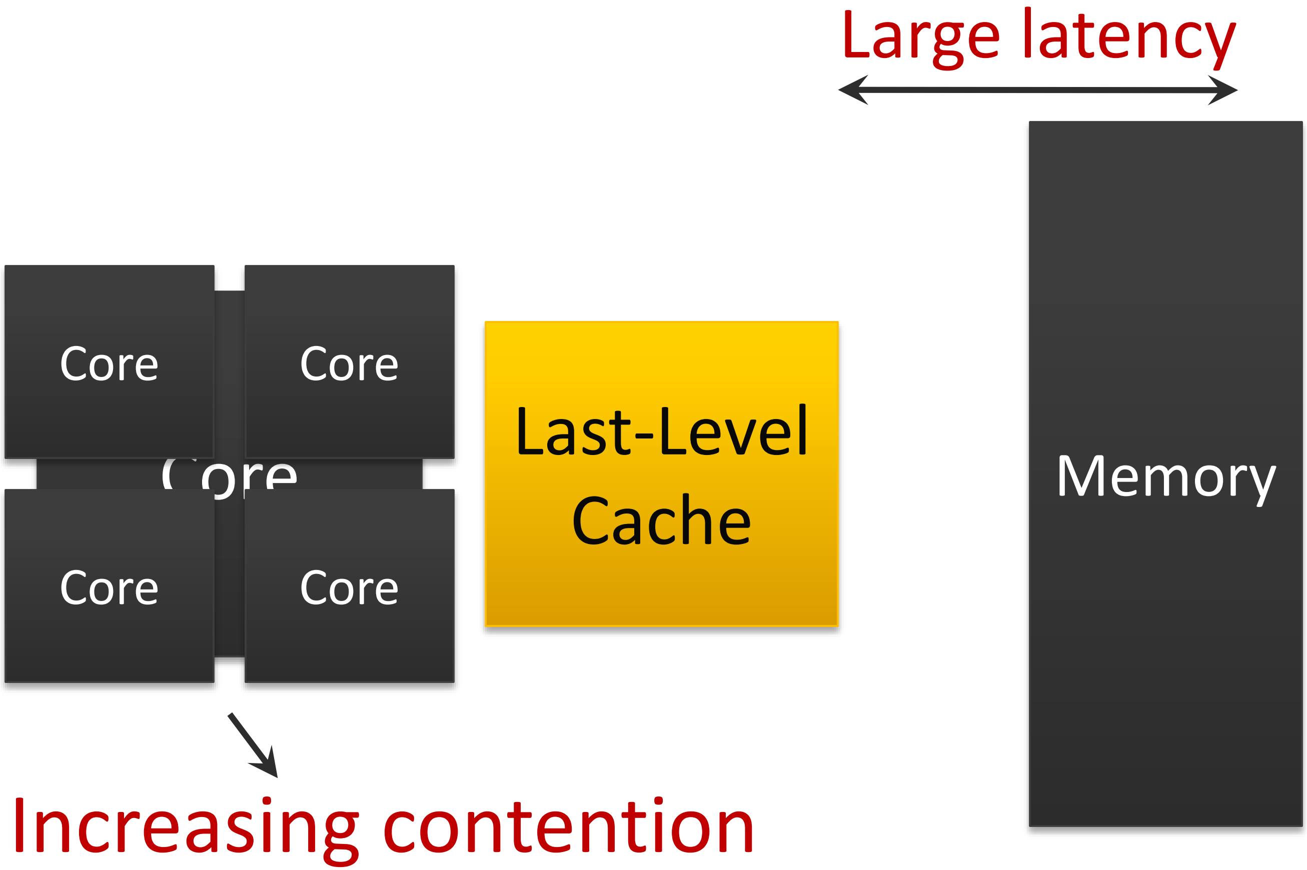
Efficient Cache Utilization: Examples

- 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.

Cache Utilization is Important



Large latency

Core

Core

Core

Core

Core

Last-Level Cache

Memory

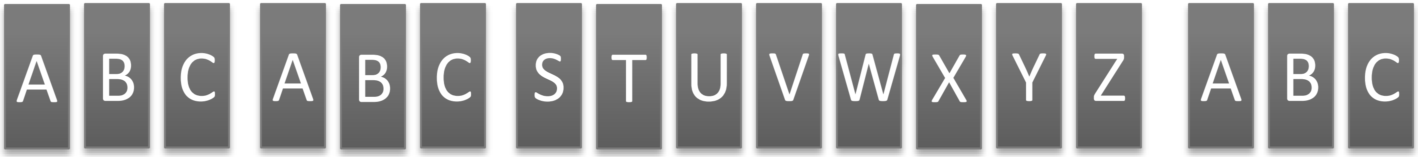
Increasing contention

Effective cache utilization is important

Reuse Behavior of Cache Blocks

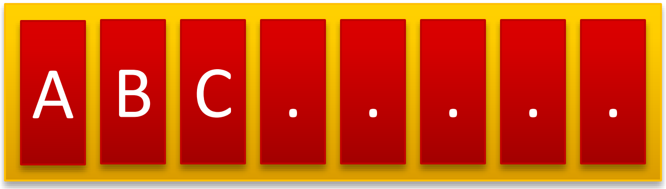
Different blocks have different reuse behavior

Access Sequence:



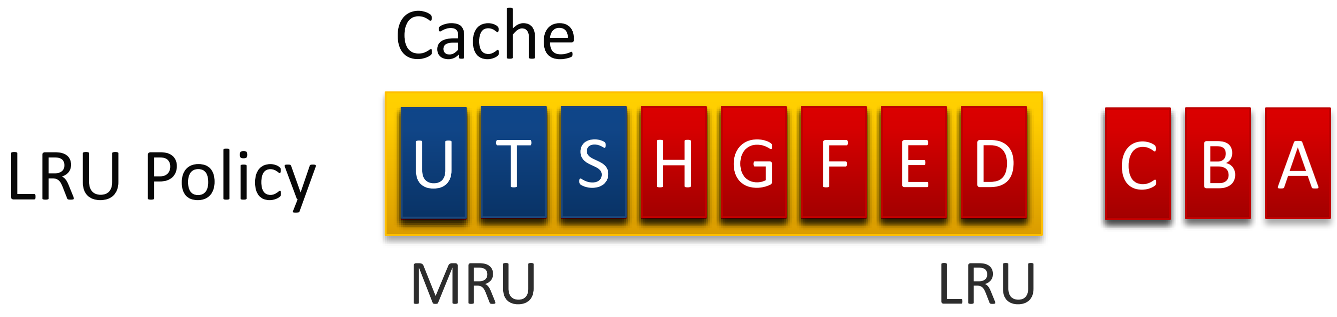
🟥 High-reuse block 🟦 Low-reuse block

Ideal Cache



Cache Pollution

Problem: Low-reuse blocks evict high-reuse blocks



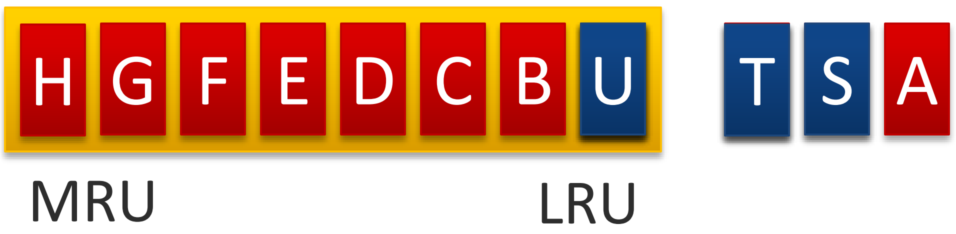
Cache

LRU Policy

MRU

LRU

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

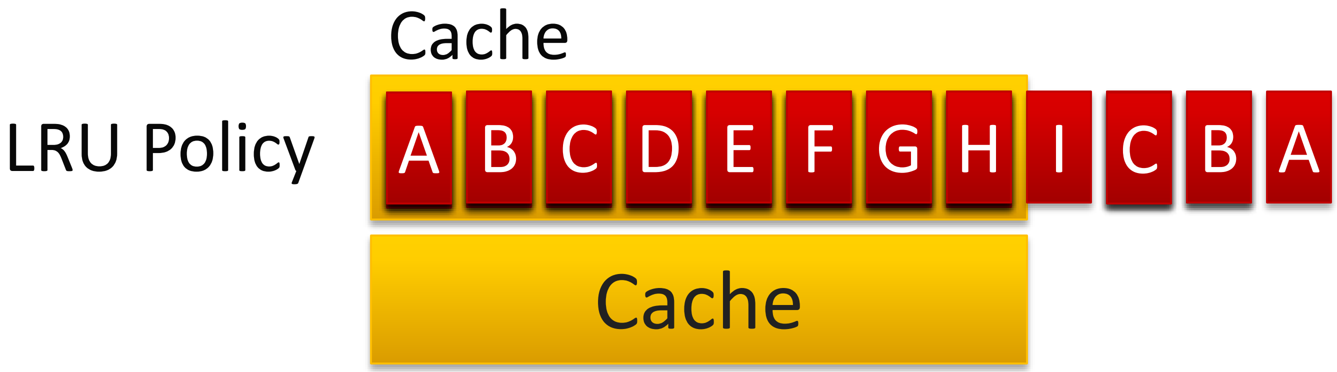


MRU

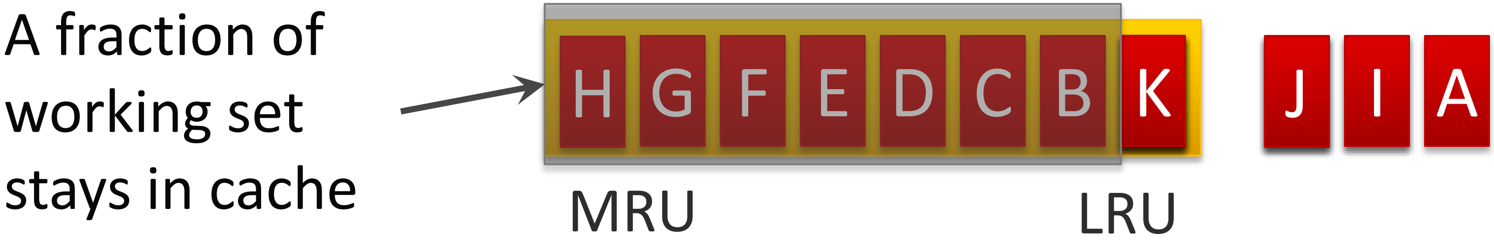
LRU

Cache Thrashing

Problem: High-reuse blocks evict each other



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



Qureshi+, "Adaptive insertion policies for high performance caching," ISCA 2007.

Handling Pollution and Thrashing

Need to address both pollution and thrashing concurrently

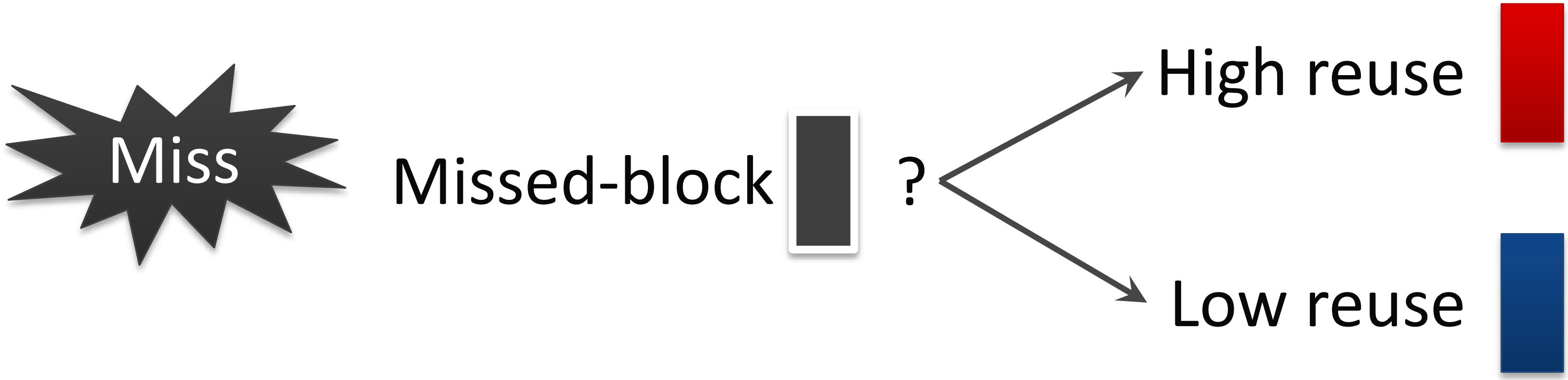
Cache Pollution

Need to distinguish high-reuse blocks from low-reuse blocks

Cache Thrashing

Need to control the number of blocks inserted with high priority into the cache

Reuse Prediction



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

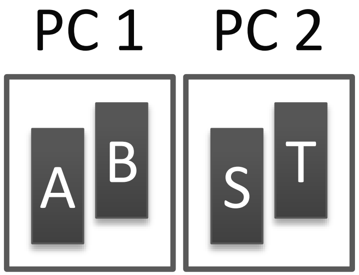
Impractical

1. High storage overhead
2. Look-up latency

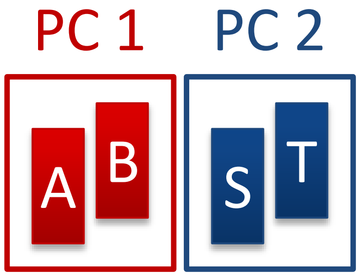
Approaches to Reuse Prediction

Use program counter or memory region information.

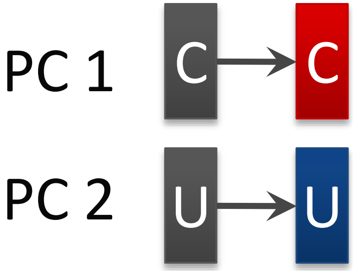
1. Group Blocks



2. Learn group behavior



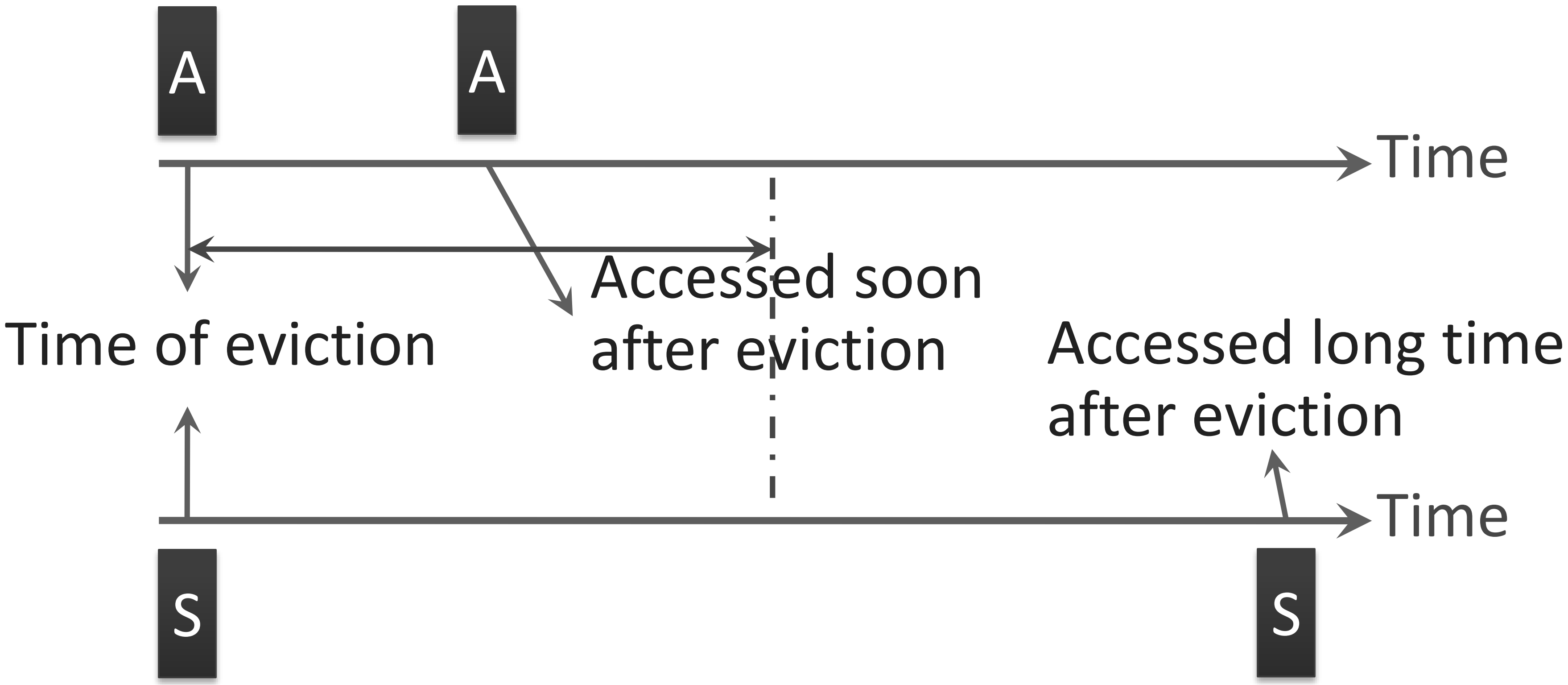
3. Predict reuse



1. Same group ↛ same reuse behavior
2. No control over number of high-reuse blocks

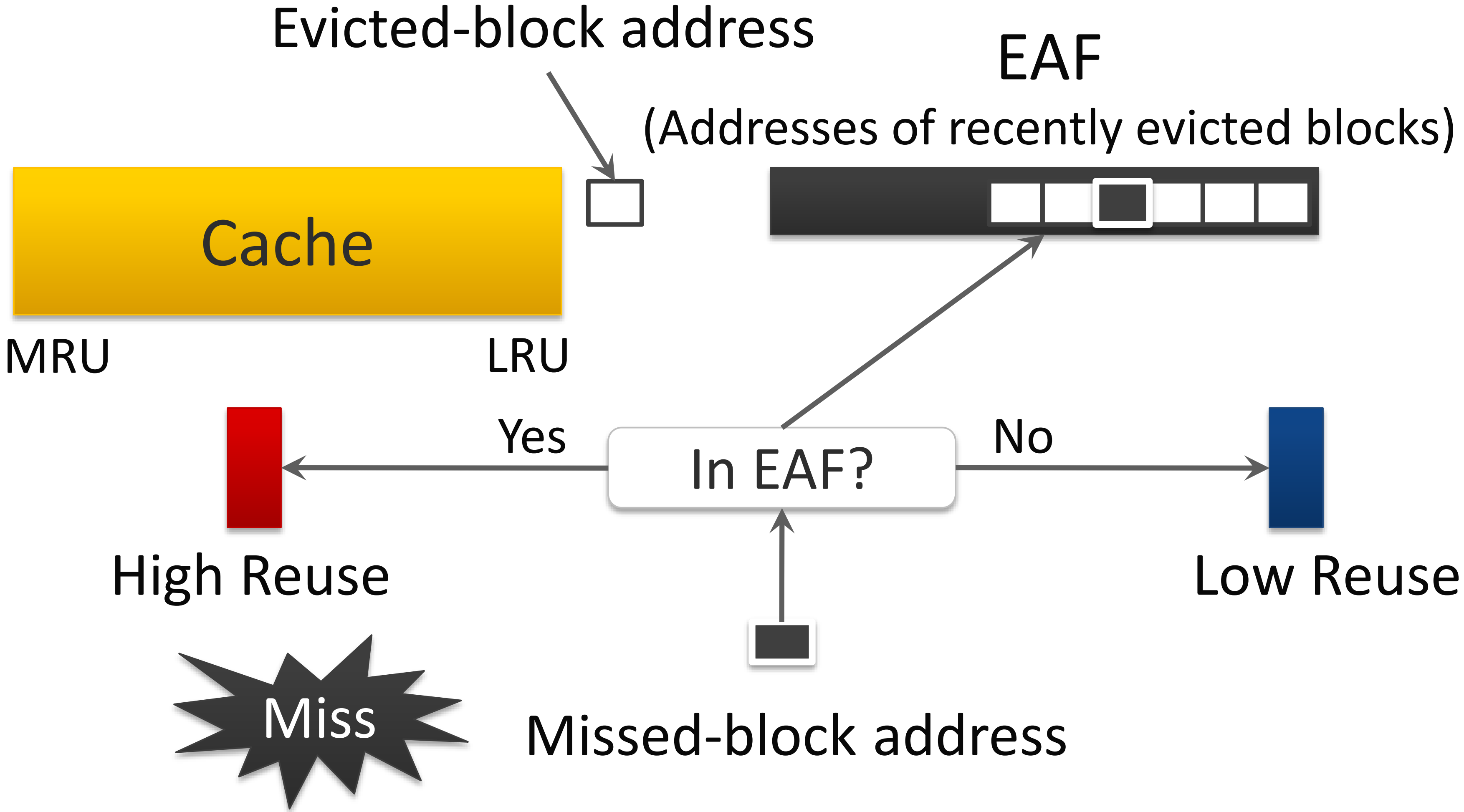
Per-block Reuse Prediction

Use recency of eviction to predict reuse

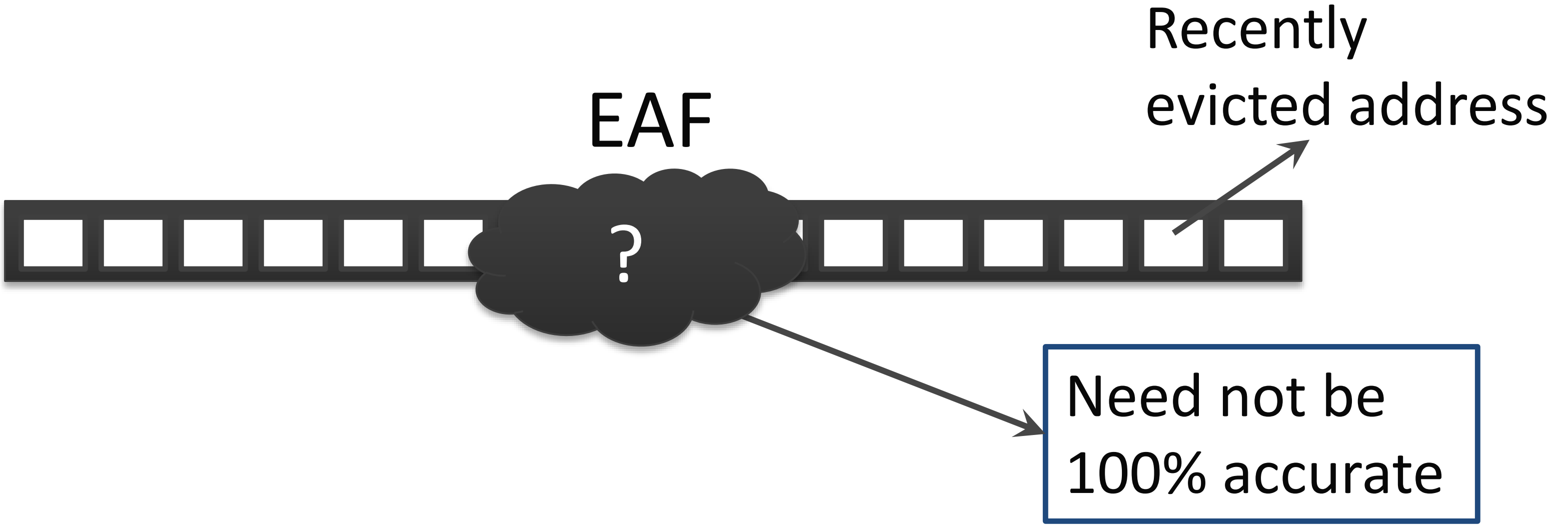


A
A
Time
Time of eviction
Accessed soon after eviction
Accessed long time after eviction
Time
S
S

Evicted-Address Filter (EAF)



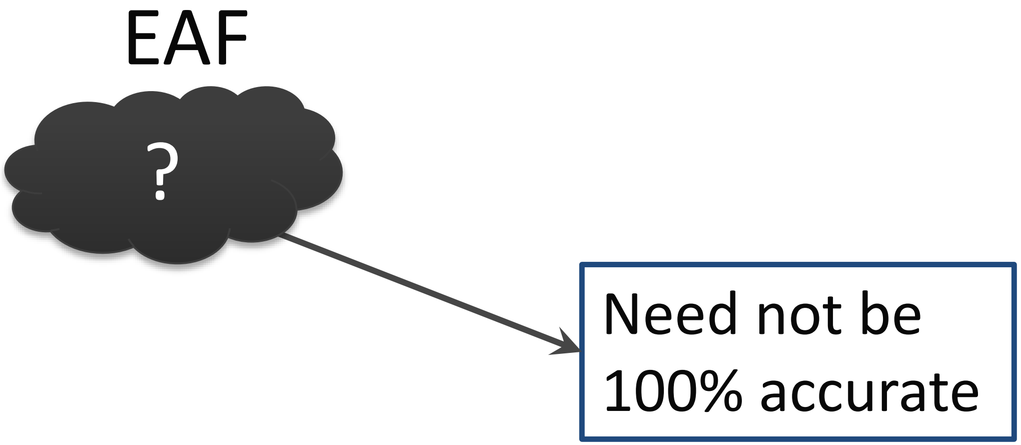
Naïve Implementation: Full Address Tags



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

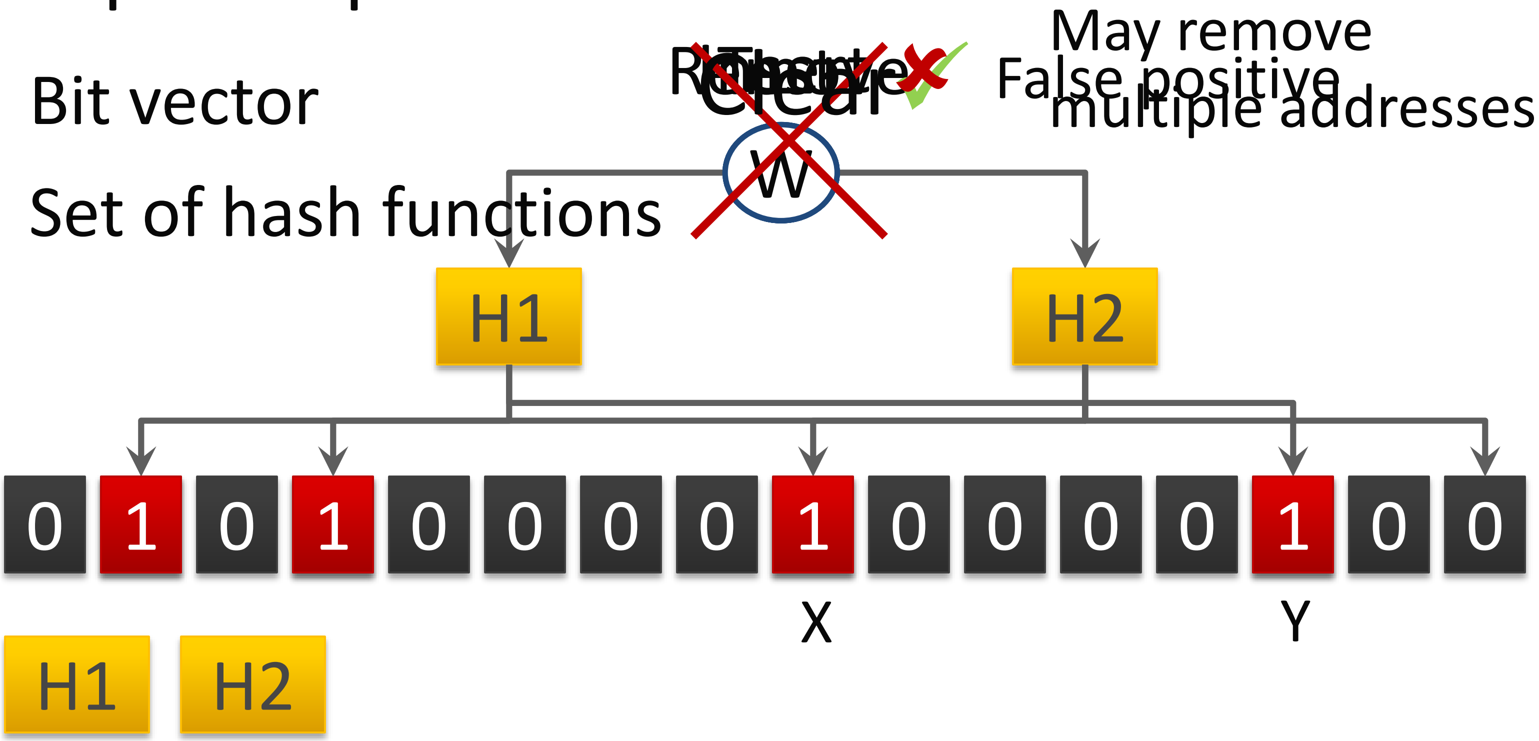
Implement EAF using a **Bloom Filter**

Low storage overhead + energy

Bloom Filter

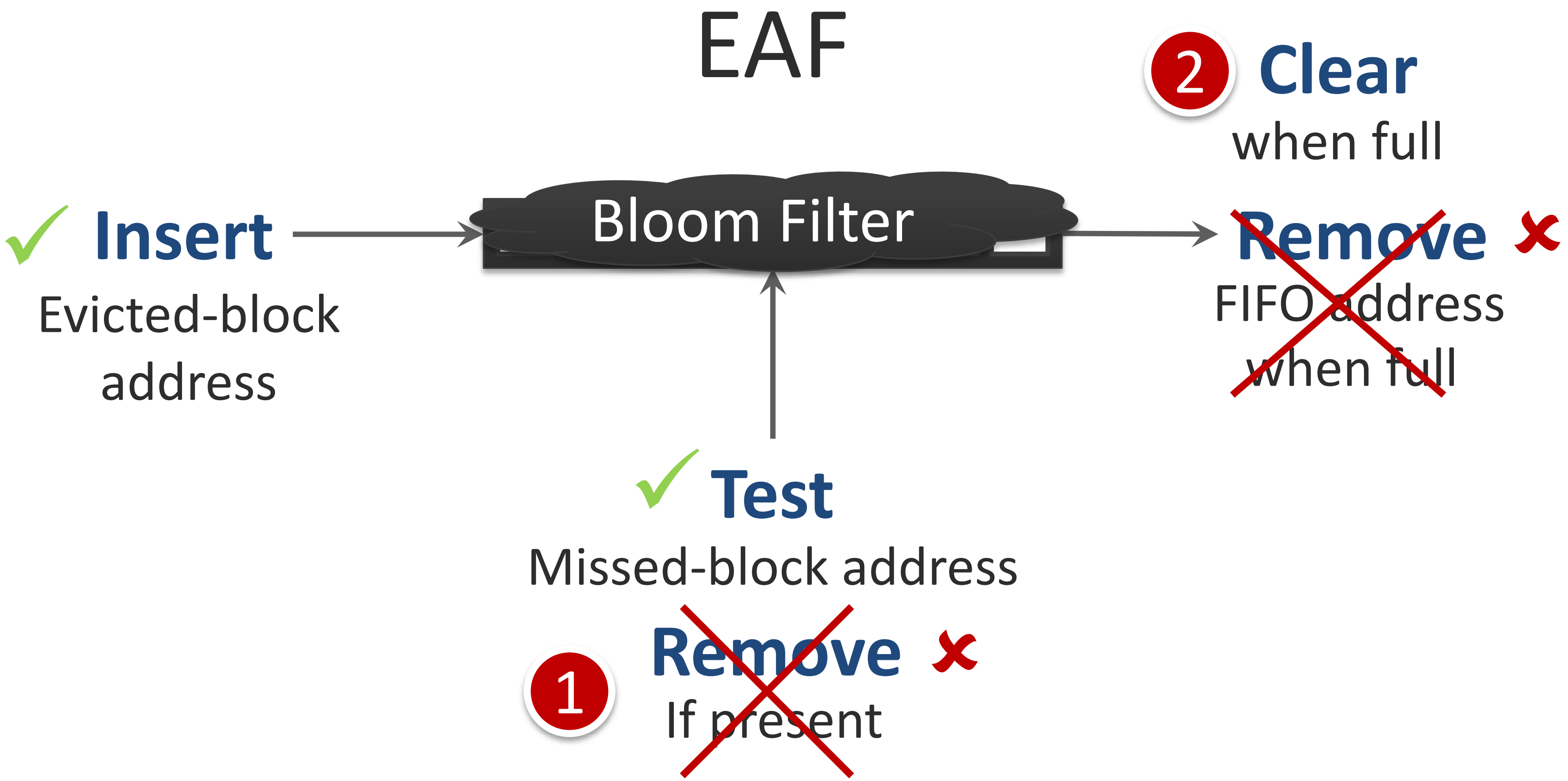
Compact representation of a set

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

② **Clear**

when full

~~**Remove**~~ ✗

~~FIFO address when full~~

① ~~**Remove**~~ ✗

~~If present~~

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

EAF-Cache: Final Design

1 **Cache eviction**

Insert address into filter

Increment counter

Cache

Bloom Filter

Counter

3 **Counter reaches max**

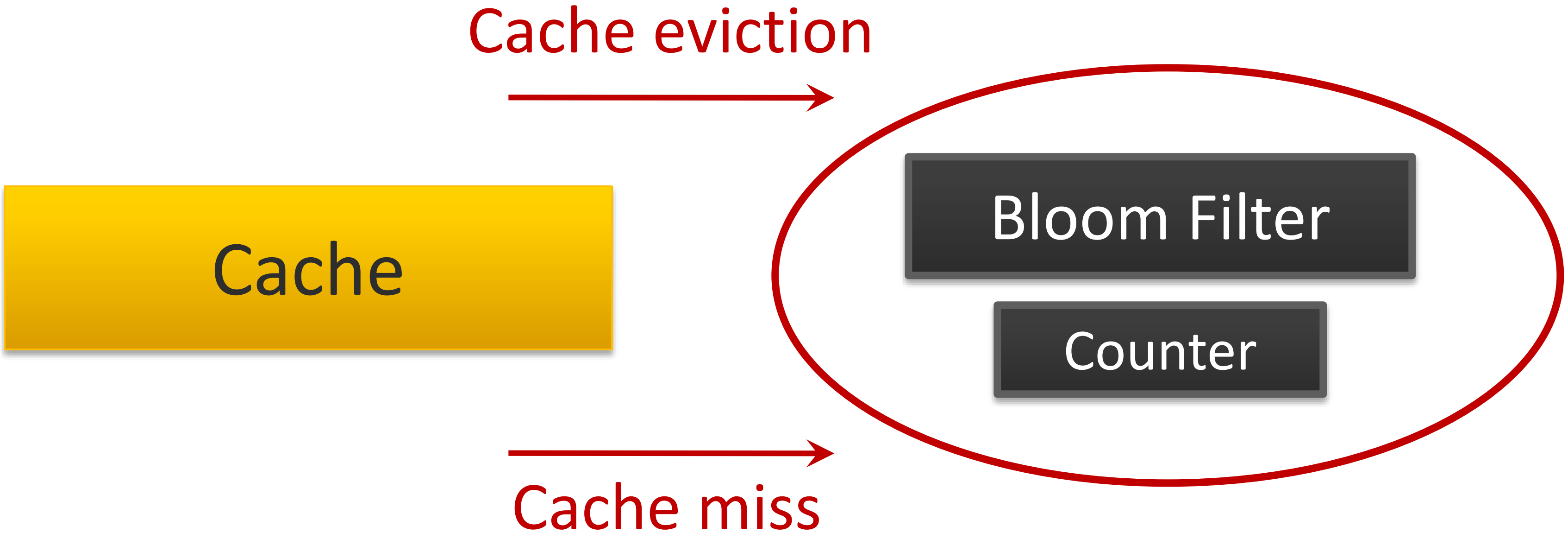
Clear filter and counter

2 **Cache miss**

Test if address is present in filter

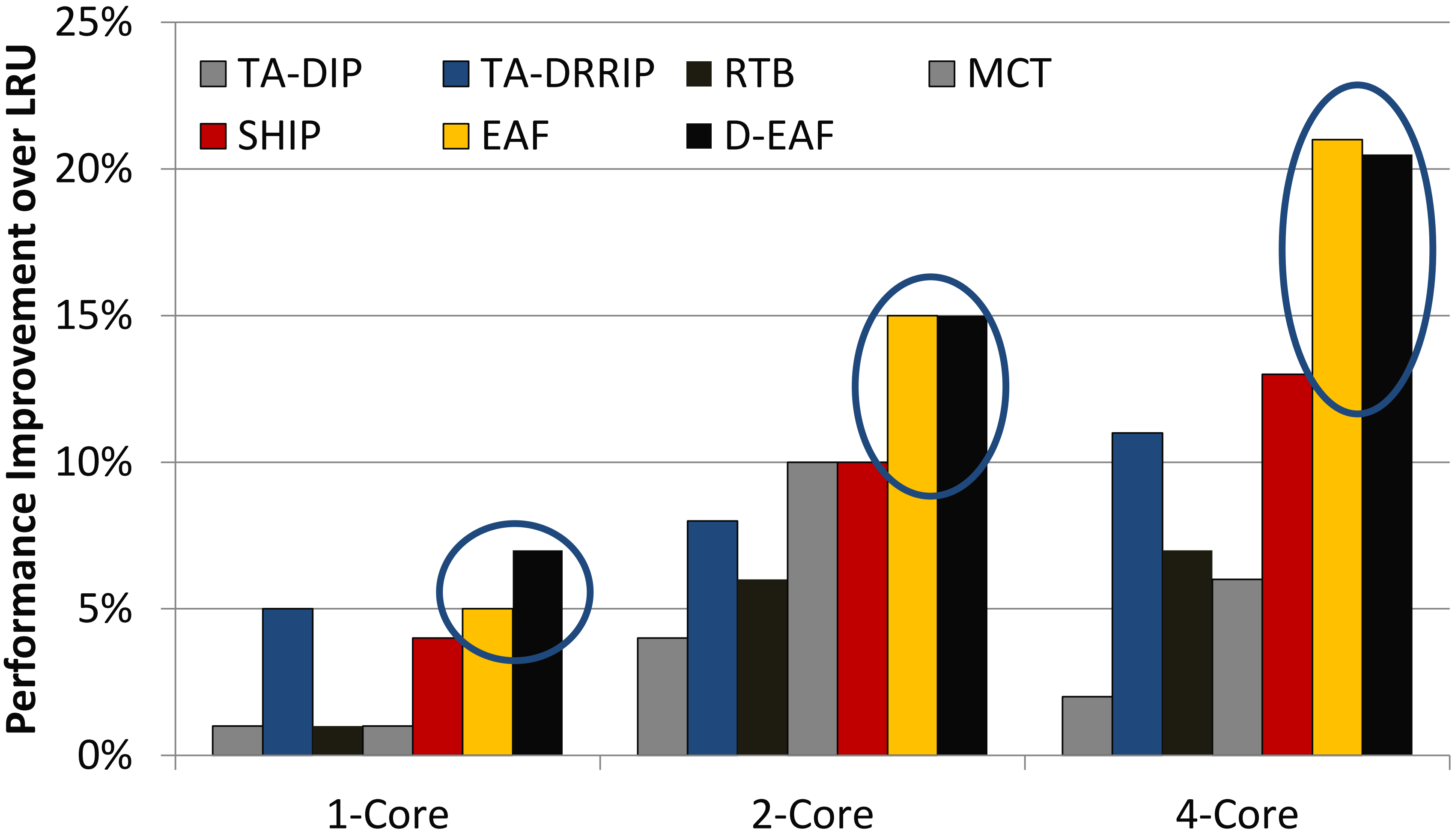
Yes, insert at MRU. No, insert with BIP

EAF: Advantages



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

EAF Performance – Summary



Cache Compression

Motivation for Cache Compression

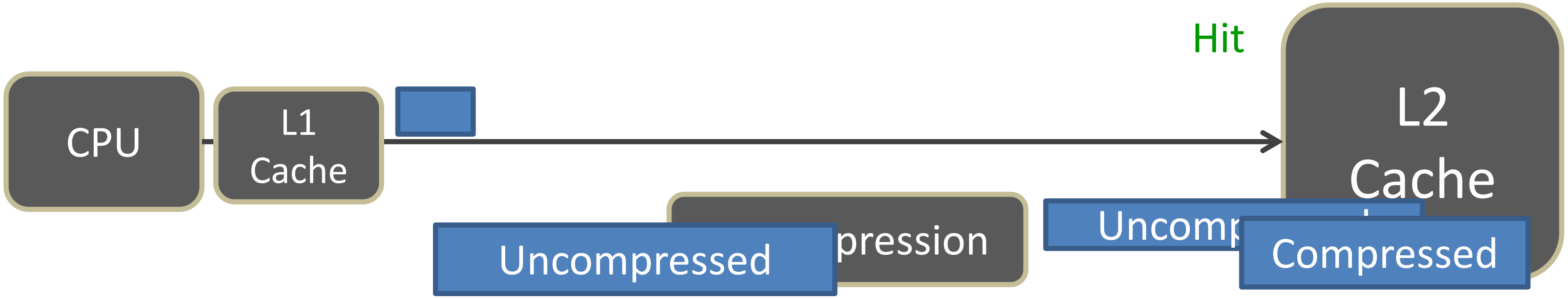
Significant redundancy in data:

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)

Summary of Major Works

Compression Mechanisms	Decompression Latency	Complexity	Compression Ratio
Zero	✓	✓	✗

Summary of Major Works

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

Summary of Major Works

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

Summary of Major Works

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

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

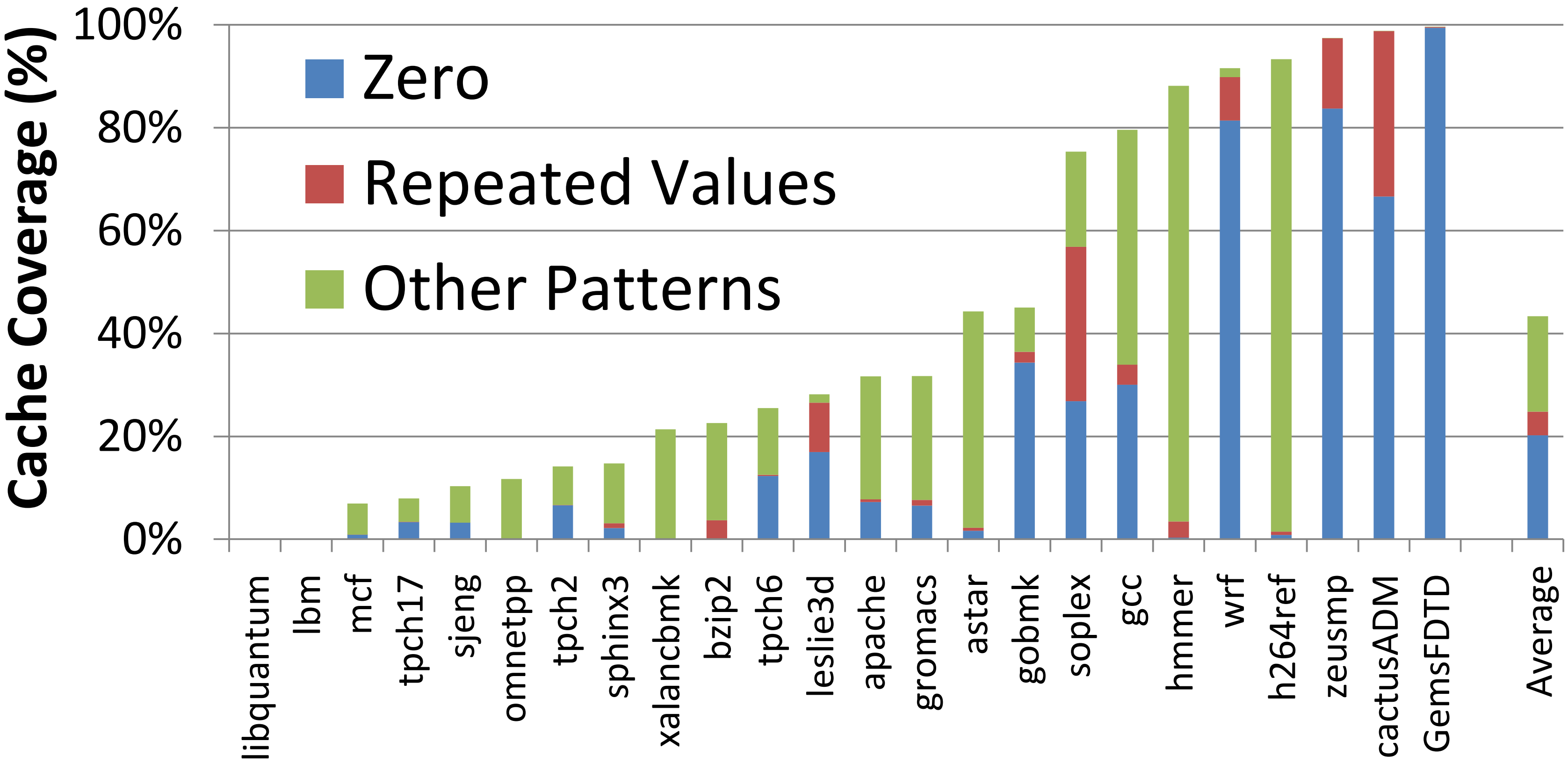
0x*000000***00**	0x*000000***0B**	0x*000000***03**	0x*000000***04**	…

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



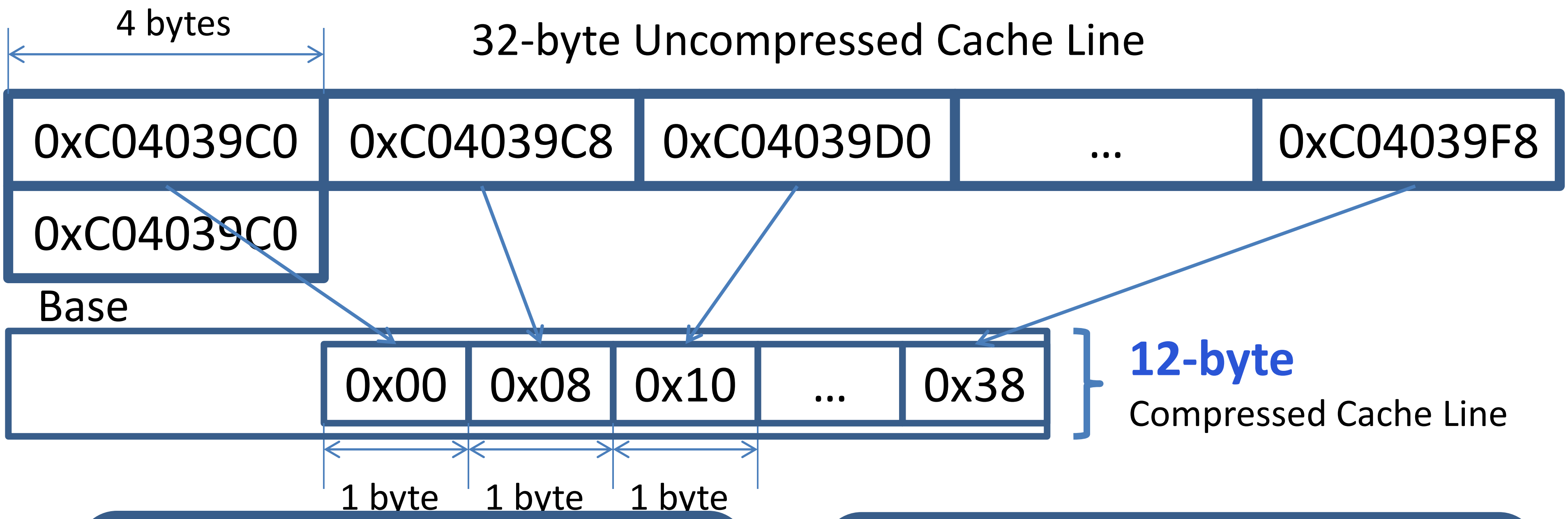
43% of the cache lines belong to key patterns

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



4 bytes

32-byte Uncompressed Cache Line

0xC04039C0 | 0xC04039C8 | 0xC04039D0 | … | 0xC04039F8

0xC04039C0

Base

0x00 | 0x08 | 0x10 | … | 0x38

1 byte 1 byte 1 byte

12-byte

Compressed Cache Line

✓ **Fast Decompression:**
vector addition

✓ **Simple Hardware:**
arithmetic and comparison

✓ **Effective:** good compression ratio

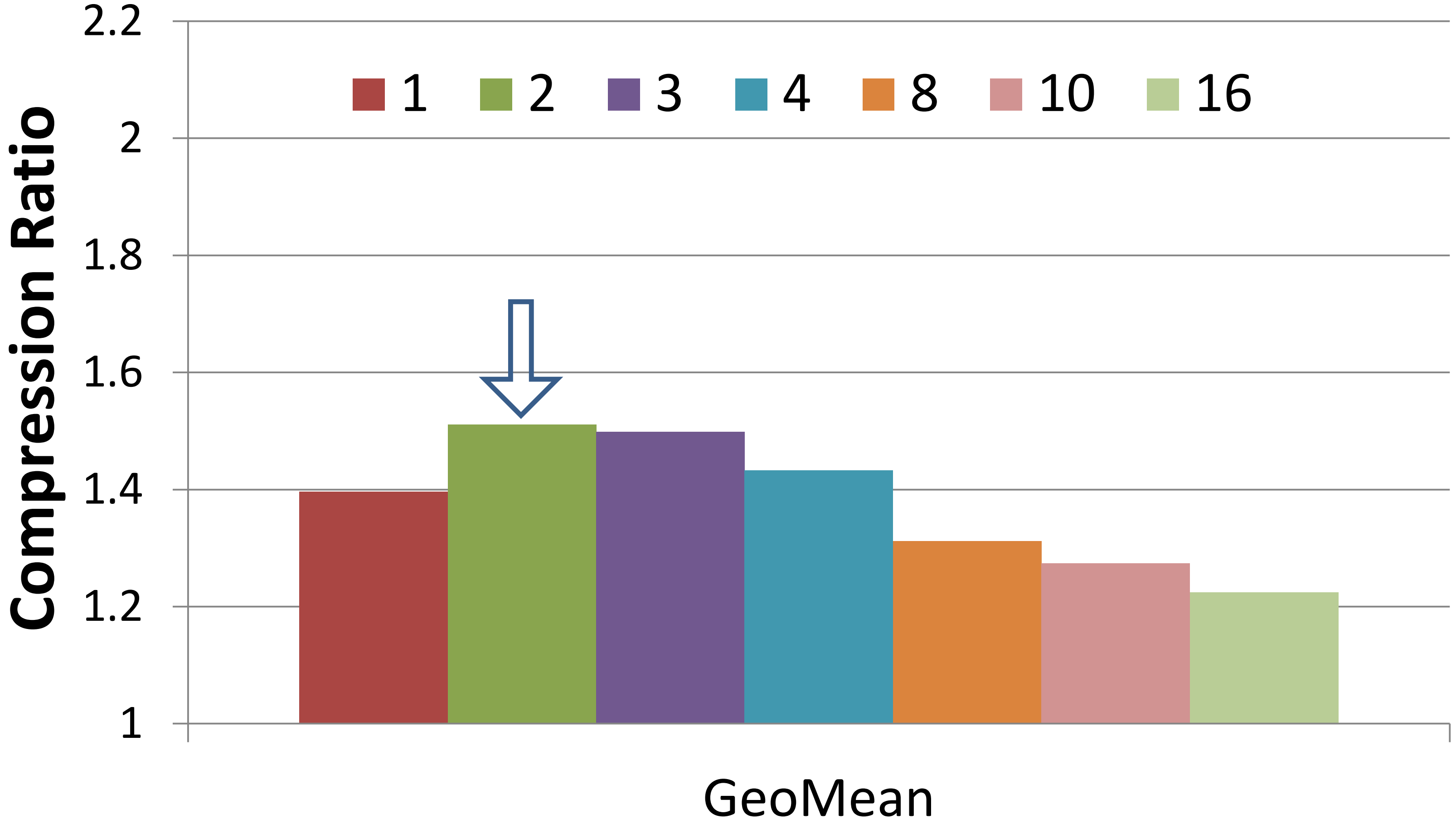
Can We Do Better?

- 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**

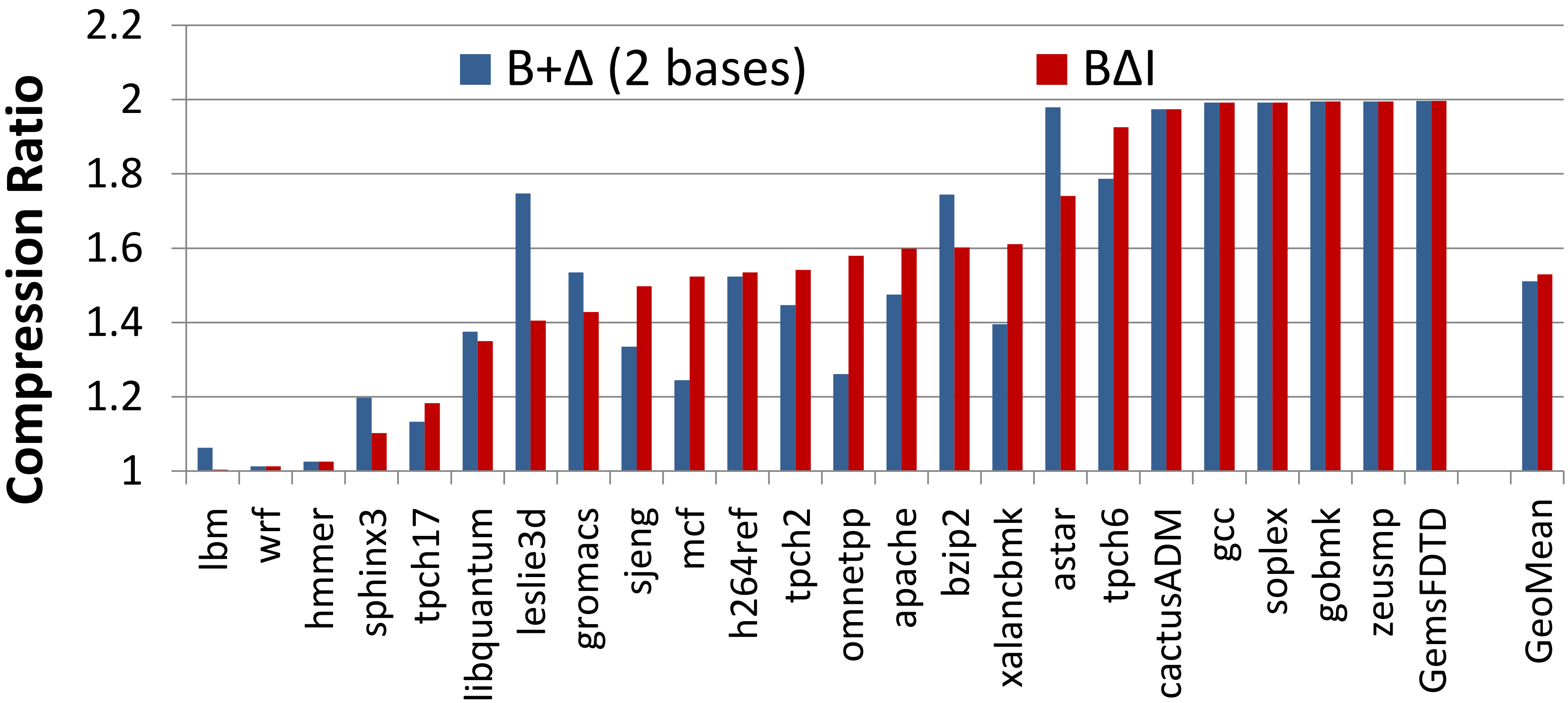
Advantages over 2 arbitrary bases:

– Better compression ratio

– Simpler compression logic

Base-Delta-Immediate (BΔI) Compression

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

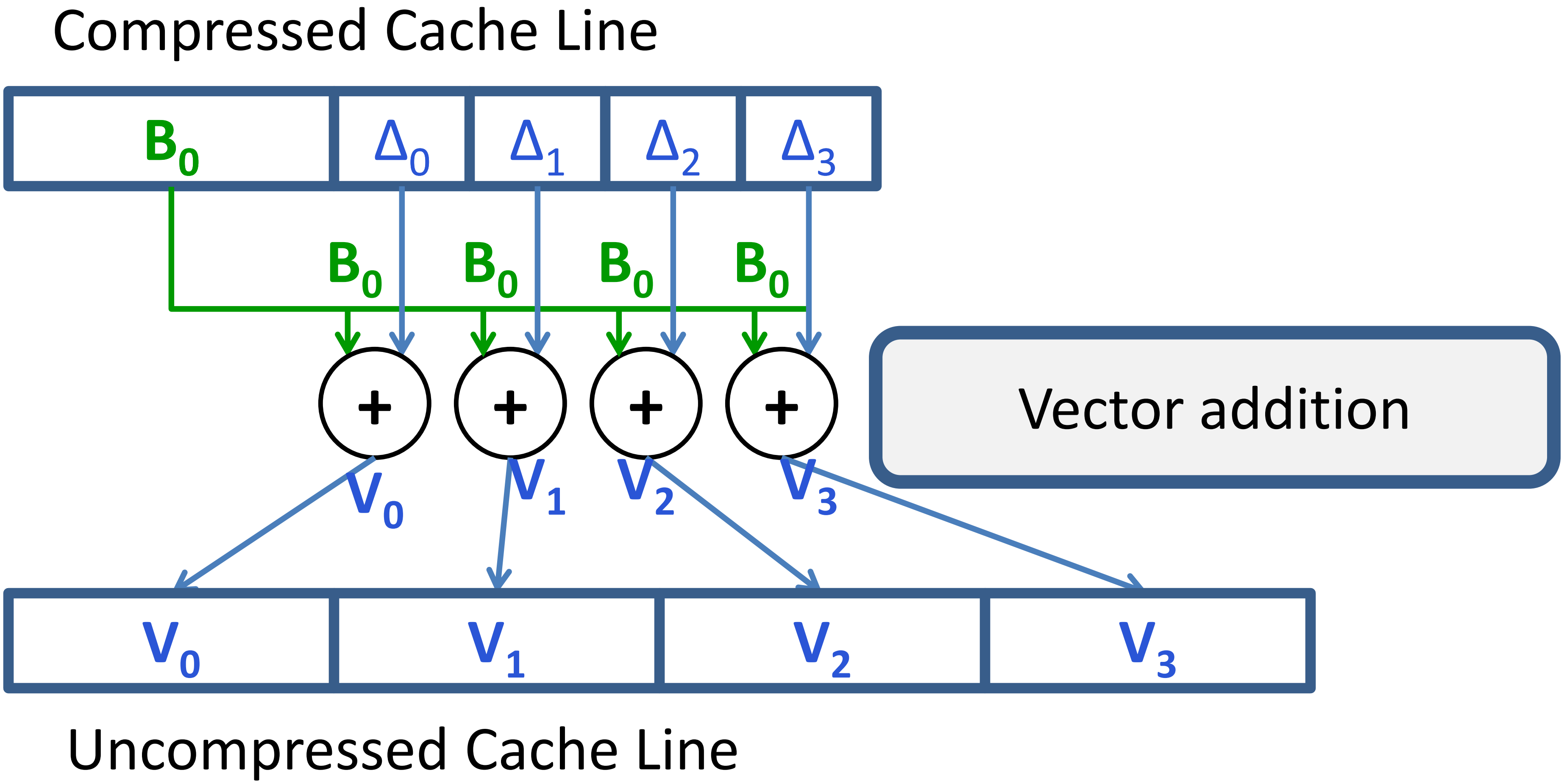


Average compression ratio is close, but **BΔI** is **simpler**

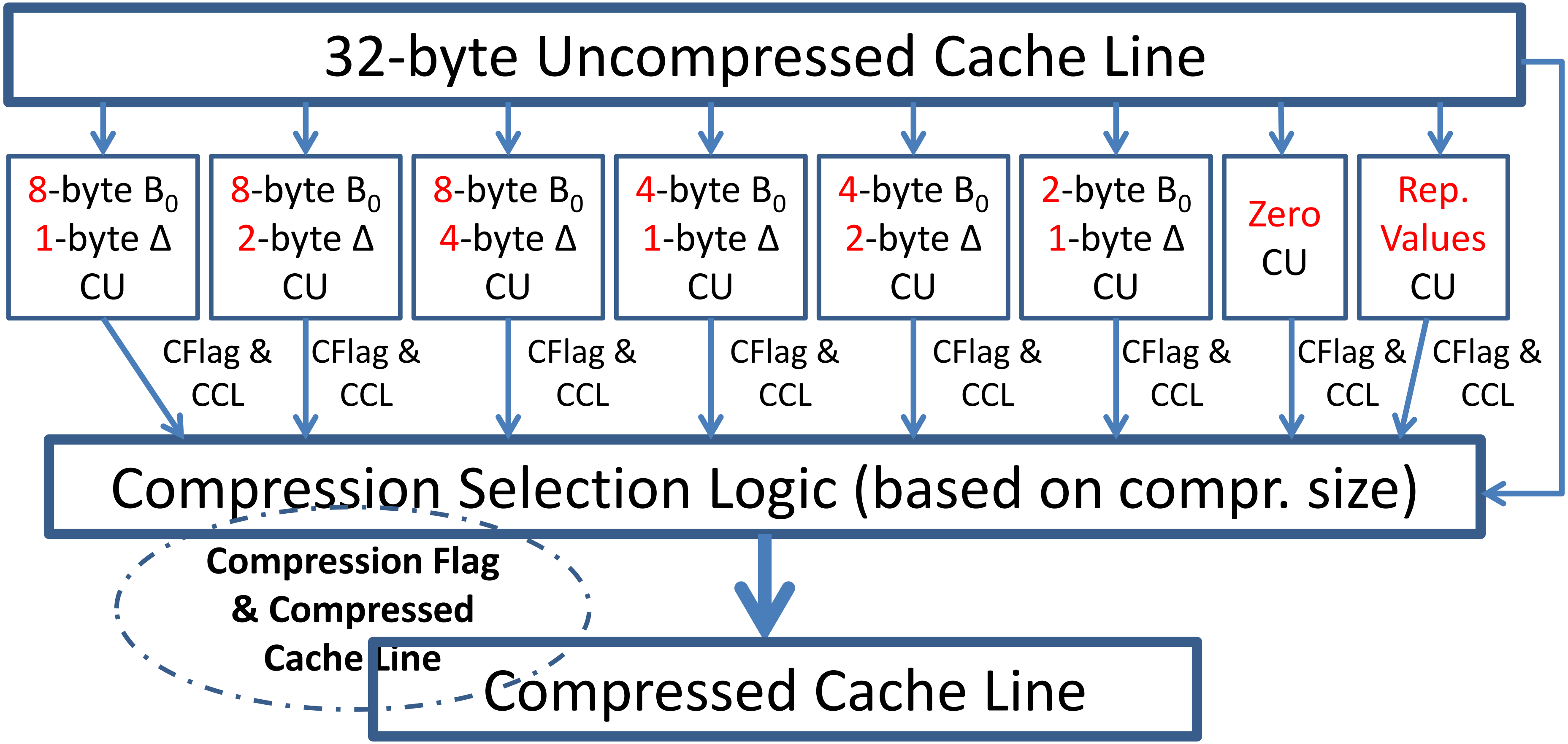
BΔI Cache Compression Implementation

- **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

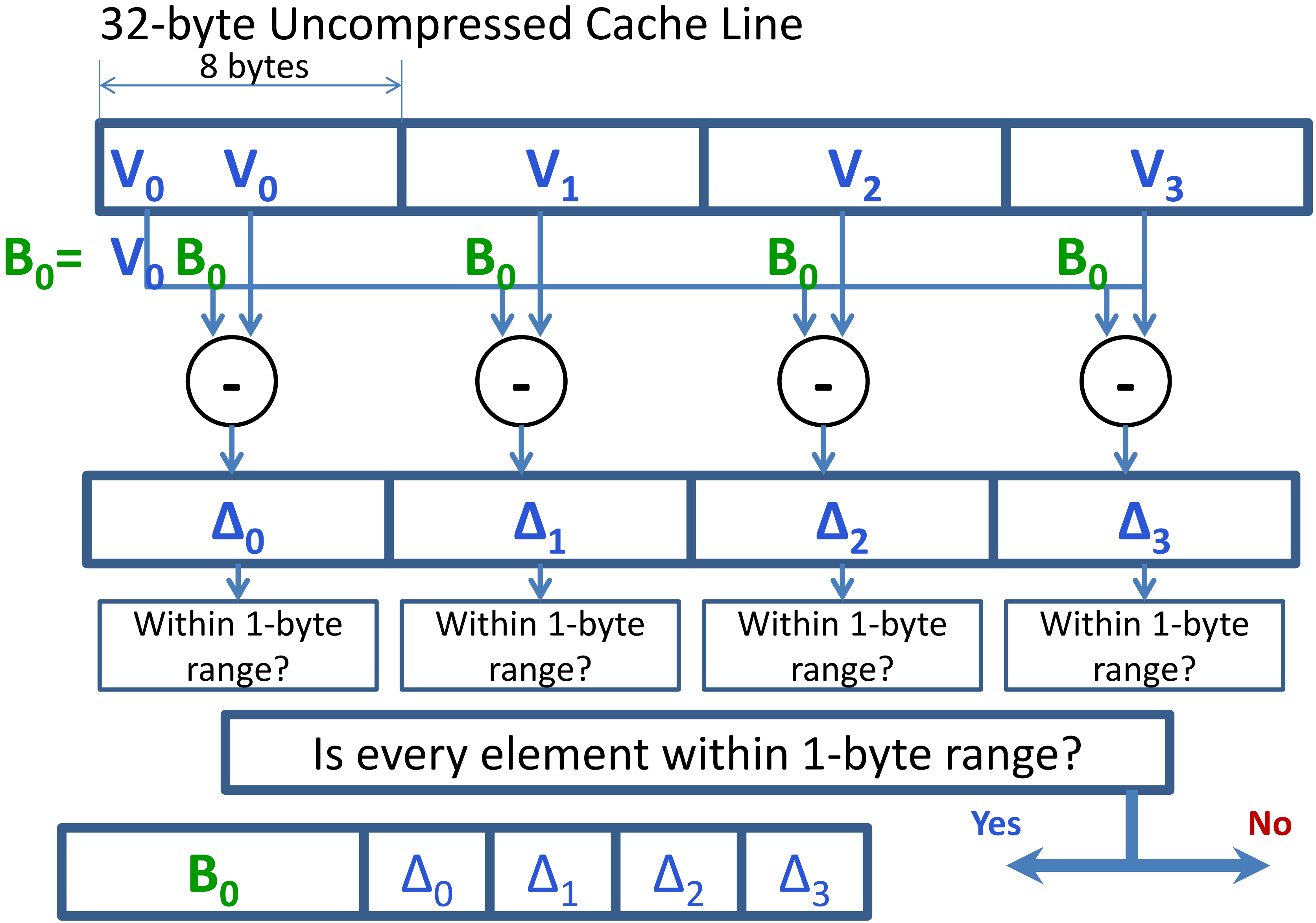
CFlag & CCL

Compression Selection Logic (based on compr. size)

Compression Flag & Compressed Cache Line

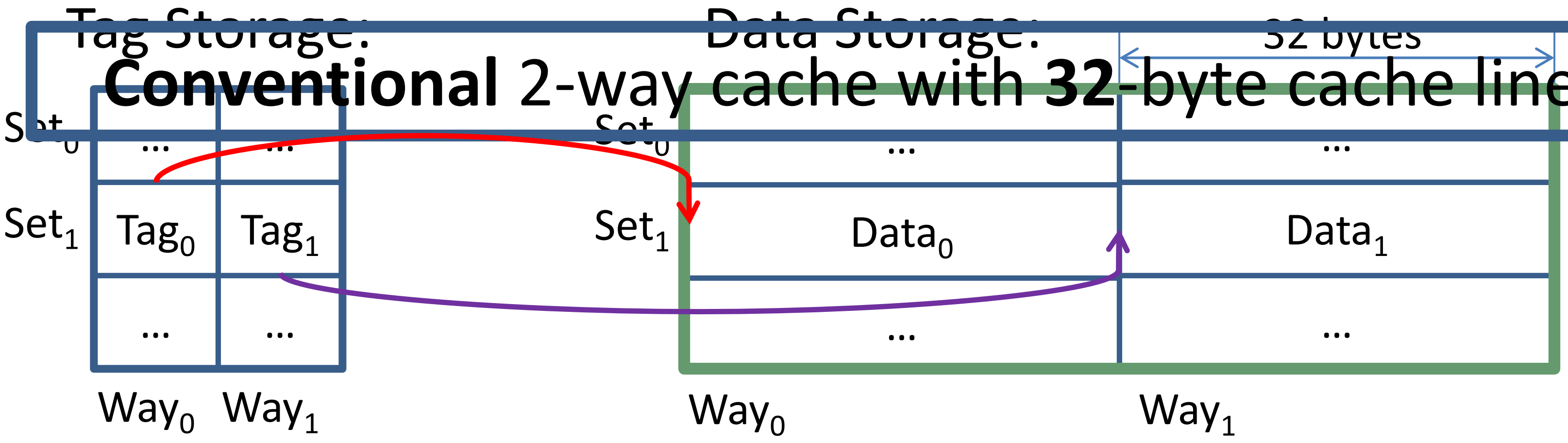
Compressed Cache Line

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

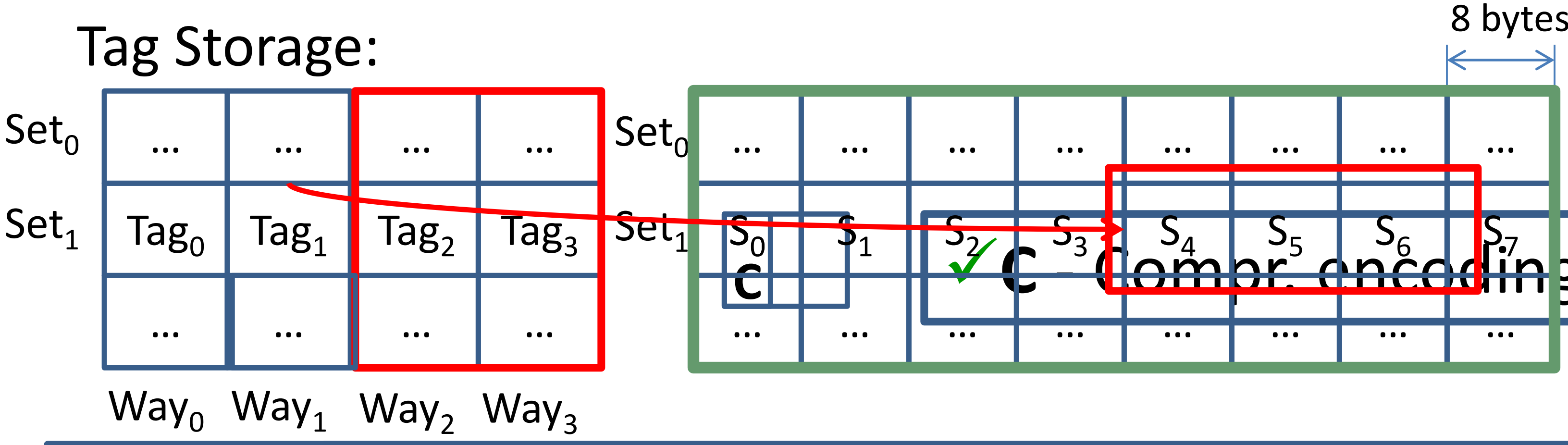


BΔI Cache Organization

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

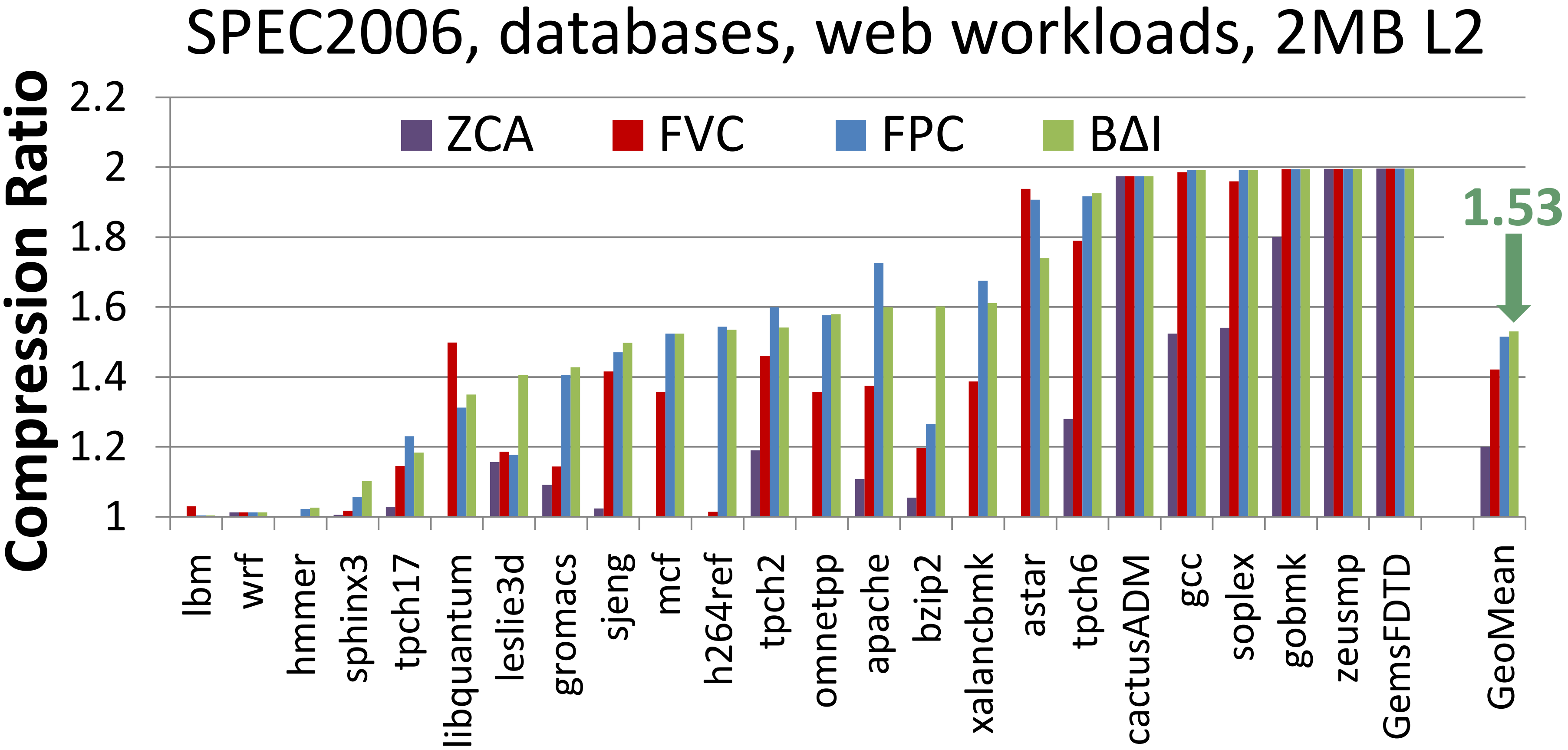
✓Tags map to multiple adjacent segments

2.3% overhead for 2 MB cache

Qualitative Comparison with Prior Work

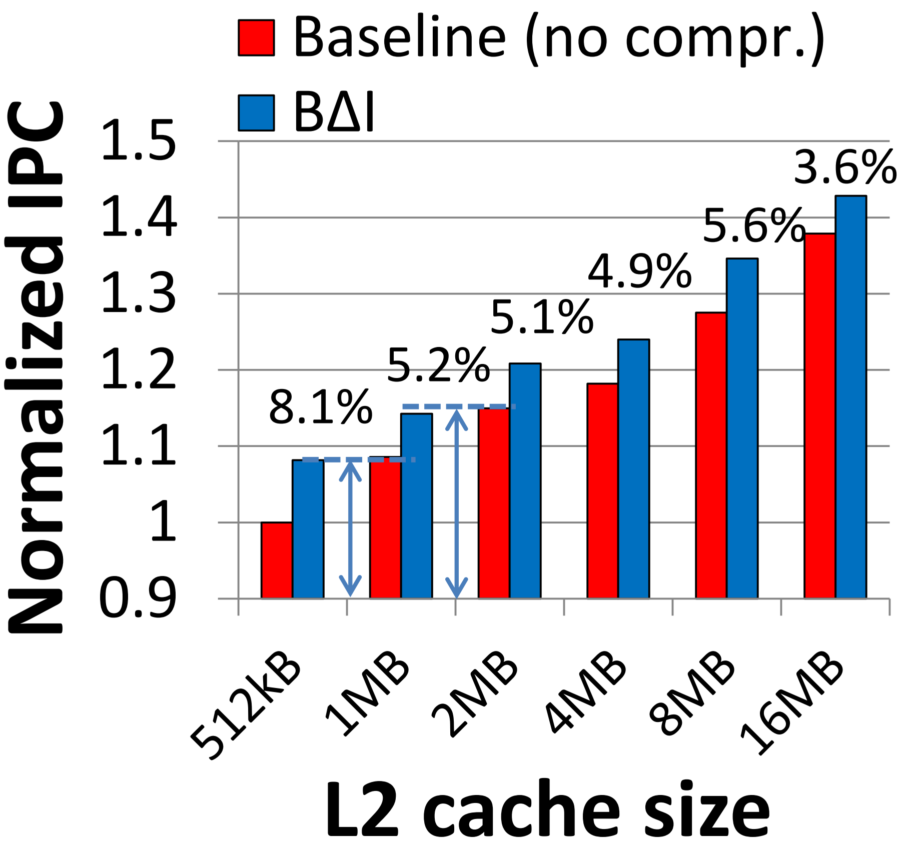
- **Zero-based designs**
  - ZCA *[Dusser+, ICS’09]*: zero-content augmented cache
  - ZVC *[Islam+, PACT’09]*: zero-value cancelling
  - Limited applicability (only zero values)
- **FVC** *[Yang+, MICRO’00]*: frequent value compression
  - High decompression latency and complexity
- **Pattern-based 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)

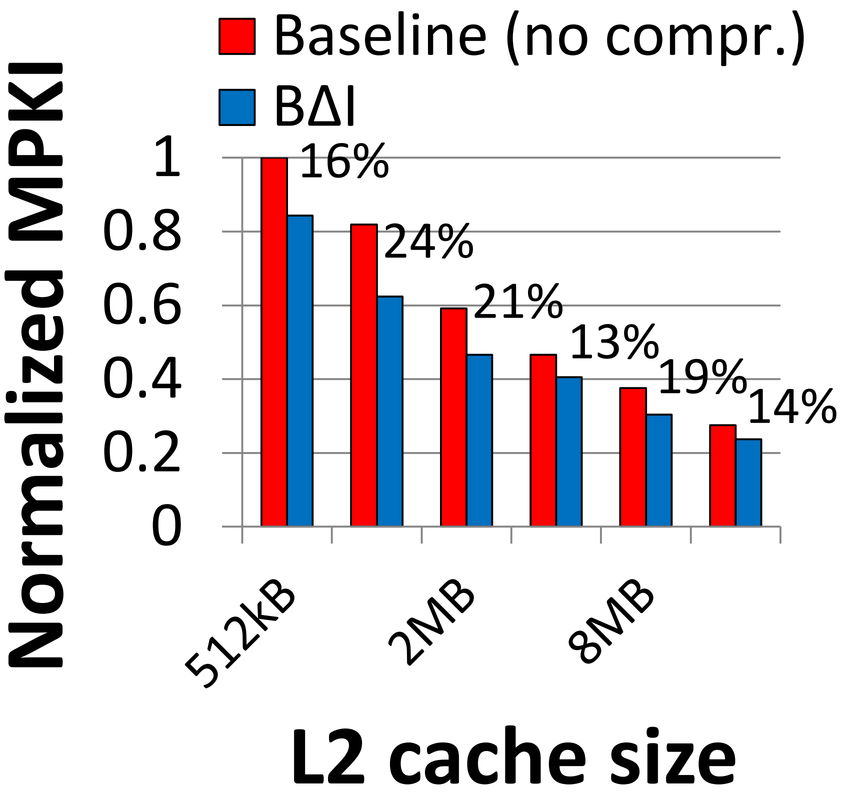
Cache Compression Ratios



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)

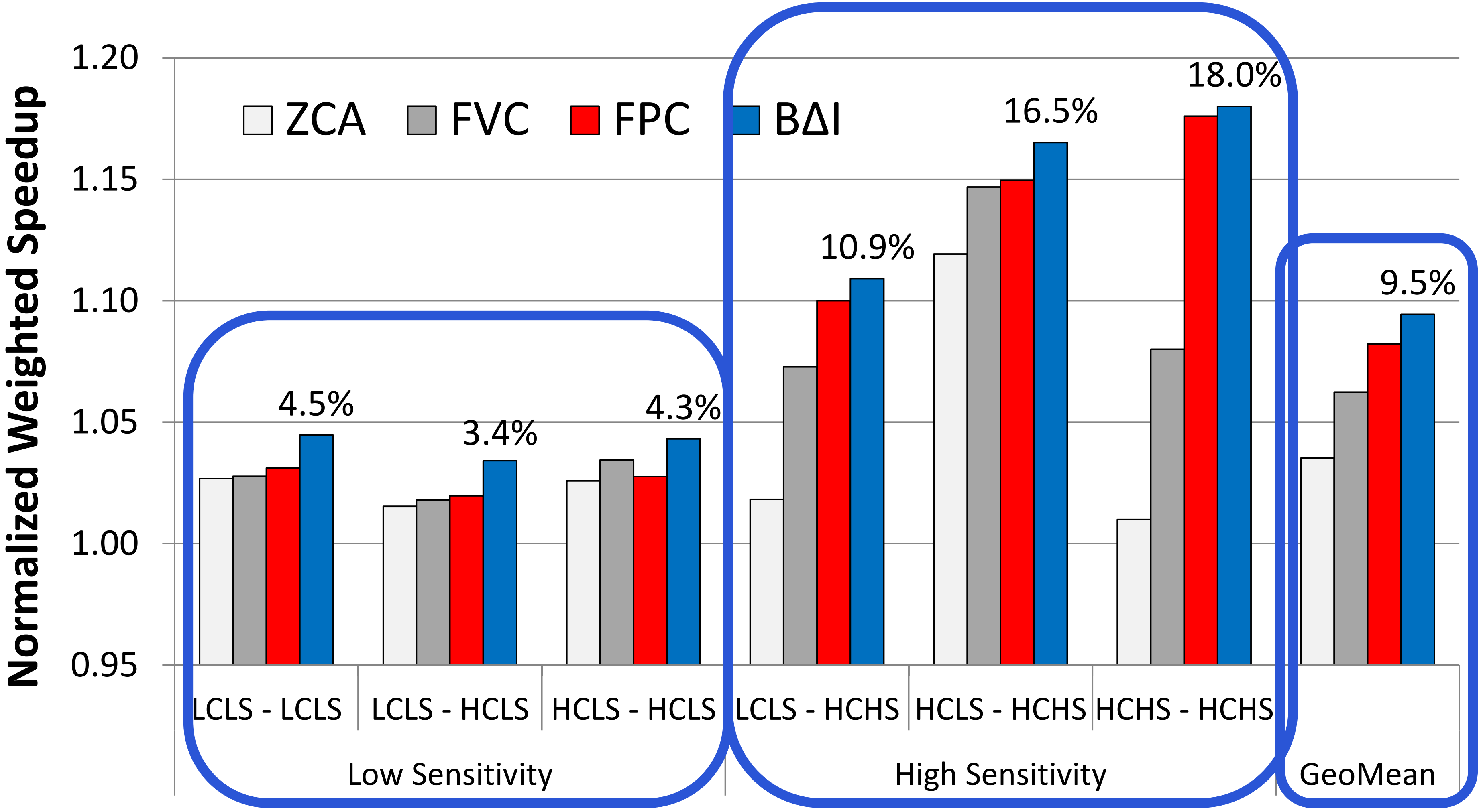
  Sensitivity: performance gain with more cache

  (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%**)

Readings for Lecture 15 (Next Monday)

- Required Reading Assignment:
  - Mutlu and Moscibroda, "Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems," ISCA 2008.

- Recommended References:
  - Muralidhara et al., "Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning," MICRO 2011.
  - Ebrahimi et al., "Parallel Application Memory Scheduling," MICRO 2011.
  - Wang et al., "A-DRM: Architecture-aware Distributed Resource Management of Virtualized Clusters," VEE 2015.

Guest Lecture on Wednesday (10/28)

- Bryan Black, AMD
  - 3D die stacking technology

18-740/640
Computer Architecture
Lecture 14: Memory Resource Management I

Prof. Onur Mutlu

Carnegie Mellon University

Fall 2015, 10/26/2015