Computer Architecture: (Shared) Cache Management

Prof. Onur Mutlu

Carnegie Mellon University

Readings

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

- Recommended
  - Pekhimenko et al., "Linearly Compressed Pages: A Main Memory Compression Framework with Low Complexity and Low Latency," MICRO 2013.

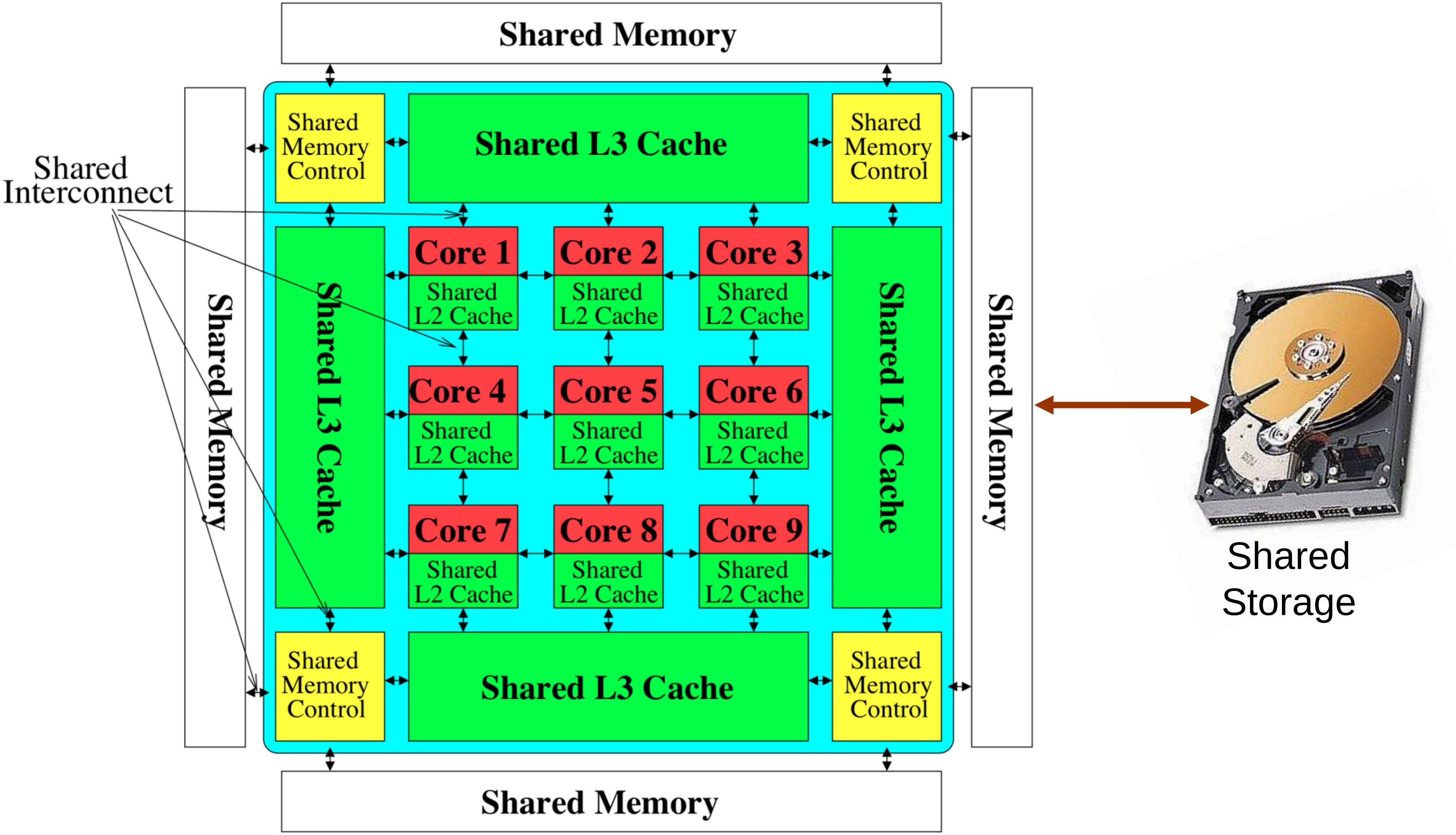
Related Videos

- Cache basics:
  - http://www.youtube.com/watch?v=TpMdBrM1hVc&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=23

- Advanced caches:
  - http://www.youtube.com/watch?v=TboaFbjTd-E&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=24

Shared Resource Design for Multi-Core Systems

The Multi-Core 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

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

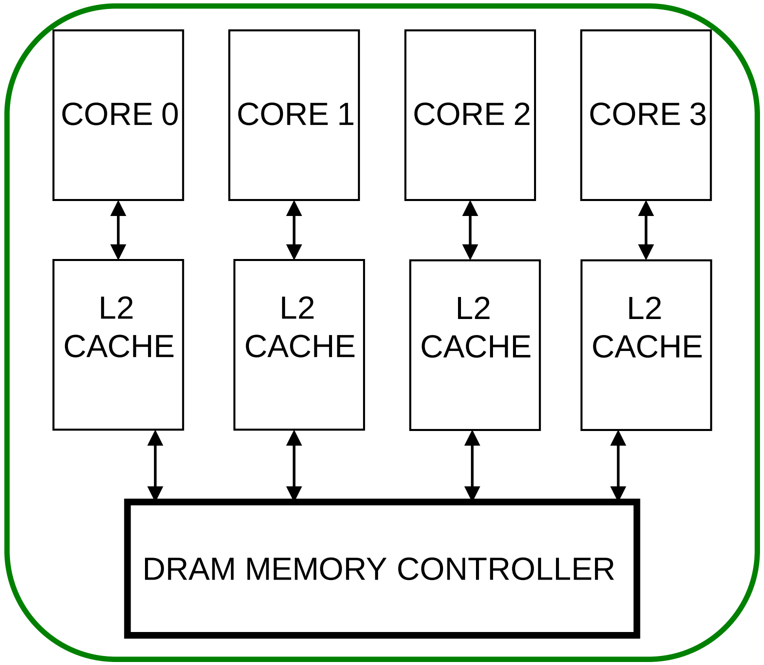
- Memory subsystem (in both MT and CMP)
  - Non-private caches
  - Interconnects
  - Memory controllers, buses, banks

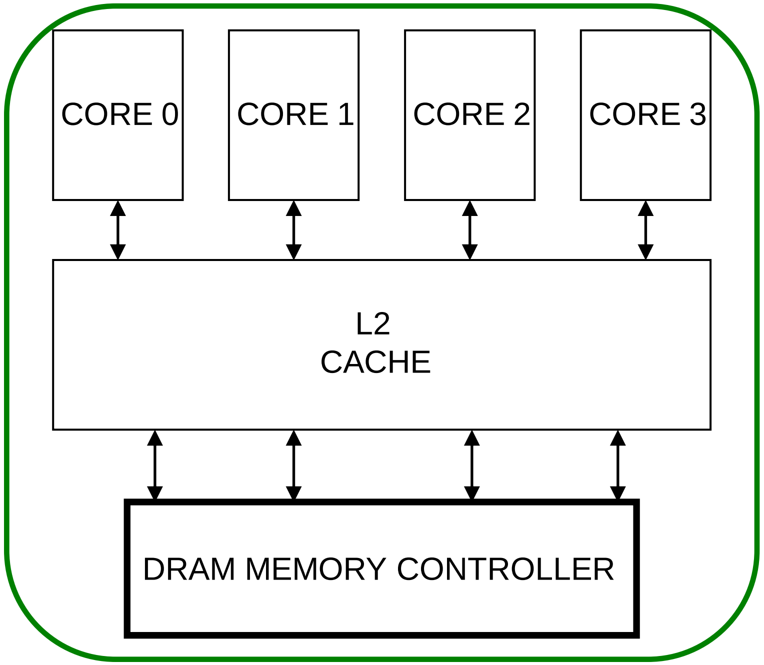
- I/O subsystem (in both MT and CMP)
  - I/O, DMA controllers
  - Ethernet controllers

- Processor (in MT)
  - 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

Controlled Cache Sharing

- 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

- 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.

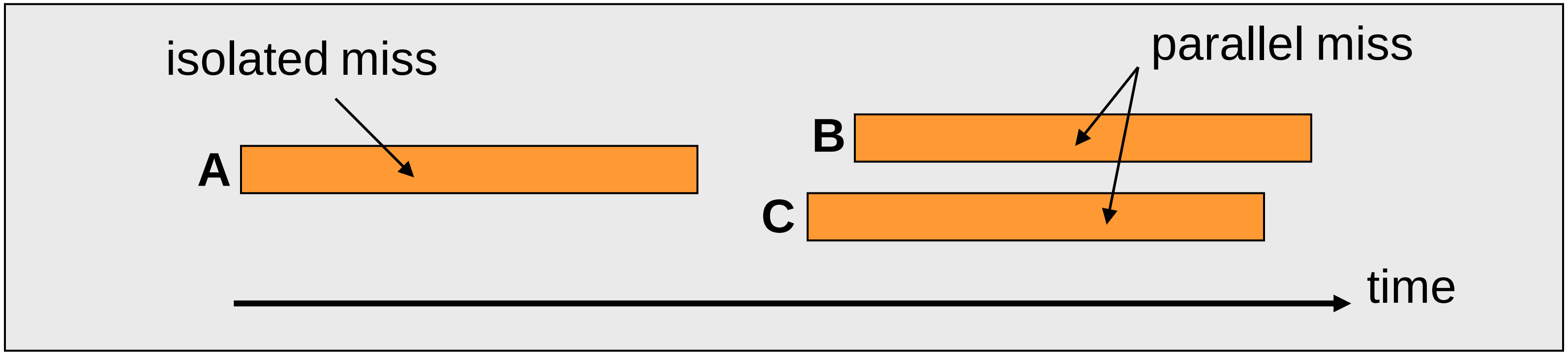
- Pekhimenko et al., "Base-Delta-Immediate Compression: Practical Data Compression for On-Chip Caches," PACT 2012.

- Pekhimenko et al., "Linearly Compressed Pages: A Main Memory Compression Framework with Low Complexity and Low Latency," SAFARI Technical Report 2013.

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

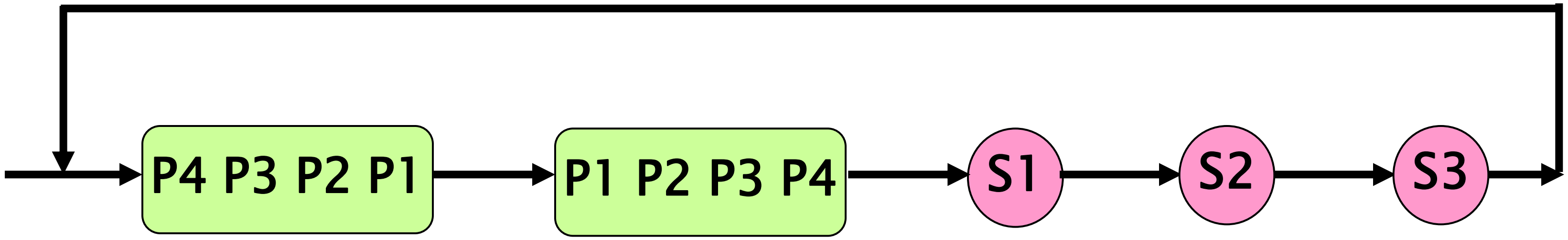
- Traditional cache replacement policies try to reduce miss count

- 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



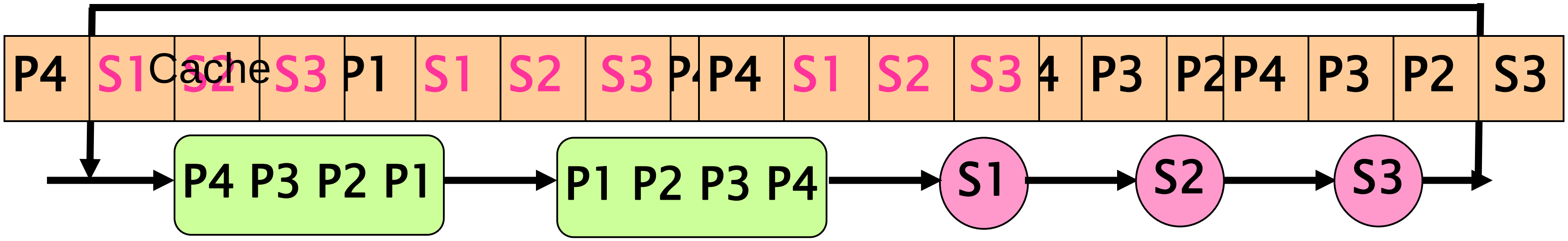
Misses to blocks P1, P2, P3, P4 can be parallel
Misses to blocks S1, S2, and S3 are isolated

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


P4 S1 Cache S2 S3 P1 S1 S2 S3 P4 P4 S1 S2 S3 4 P3 P2 P4 P3 P2 S3

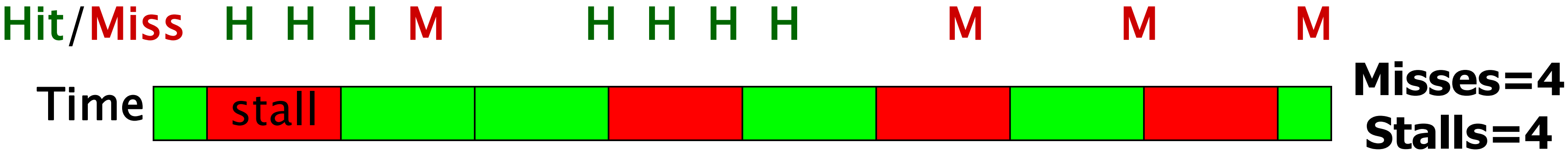
P4 P3 P2 P1

P1 P2 P3 P4

S1

S2

S3

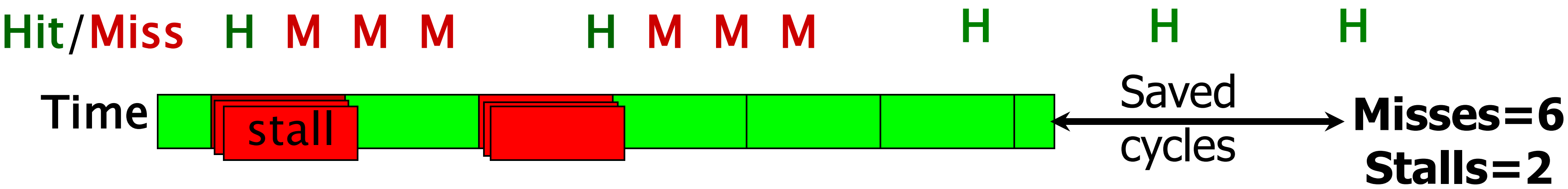

Hit/Miss H H H M H H H H M M M

Time stall

Misses=4

Stalls=4

Belady’s OPT replacement


Hit/Miss H M M M H M M M H H H

Time stall

Saved cycles

Misses=6

Stalls=2

MLP-Aware replacement

Motivation

- MLP varies. Some misses more costly than others

- MLP-aware replacement can improve performance by reducing costly misses

Outline

- ❑ Introduction

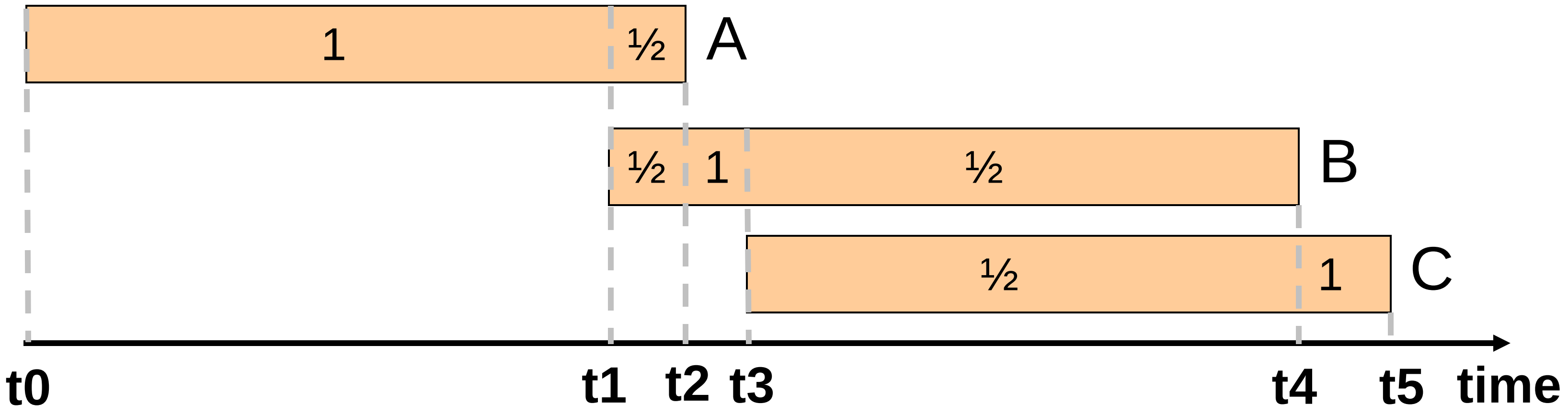
- ❑ MLP-Aware Cache Replacement
  - ▪ Model for Computing Cost
  - ▪ Repeatability of Cost
  - ▪ A Cost-Sensitive Replacement Policy

- ❑ 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

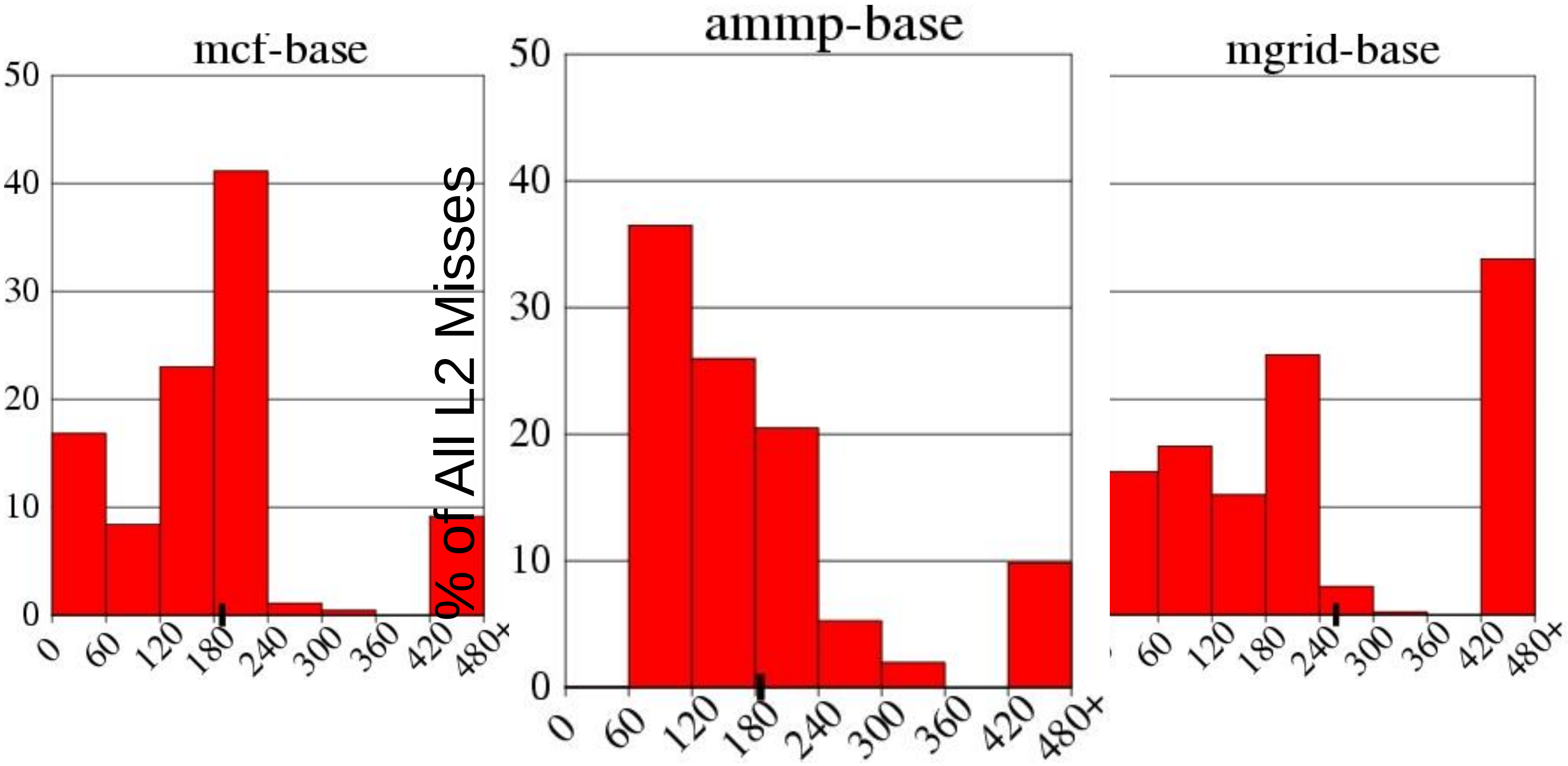
$$\boxed{\textbf{mlp-cost += (1/N)}}$$

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?

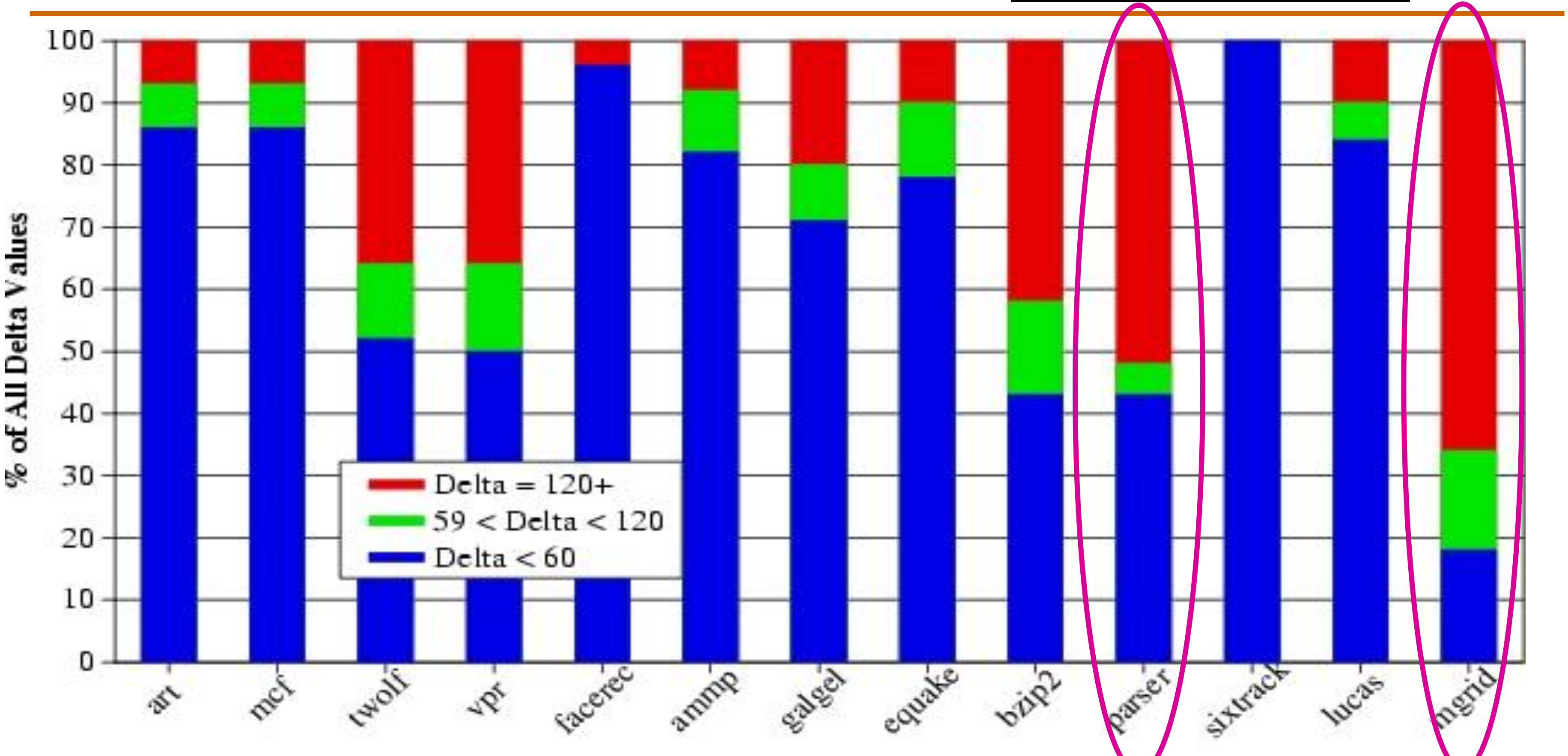
Repeatability of Cost

- ❑ 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

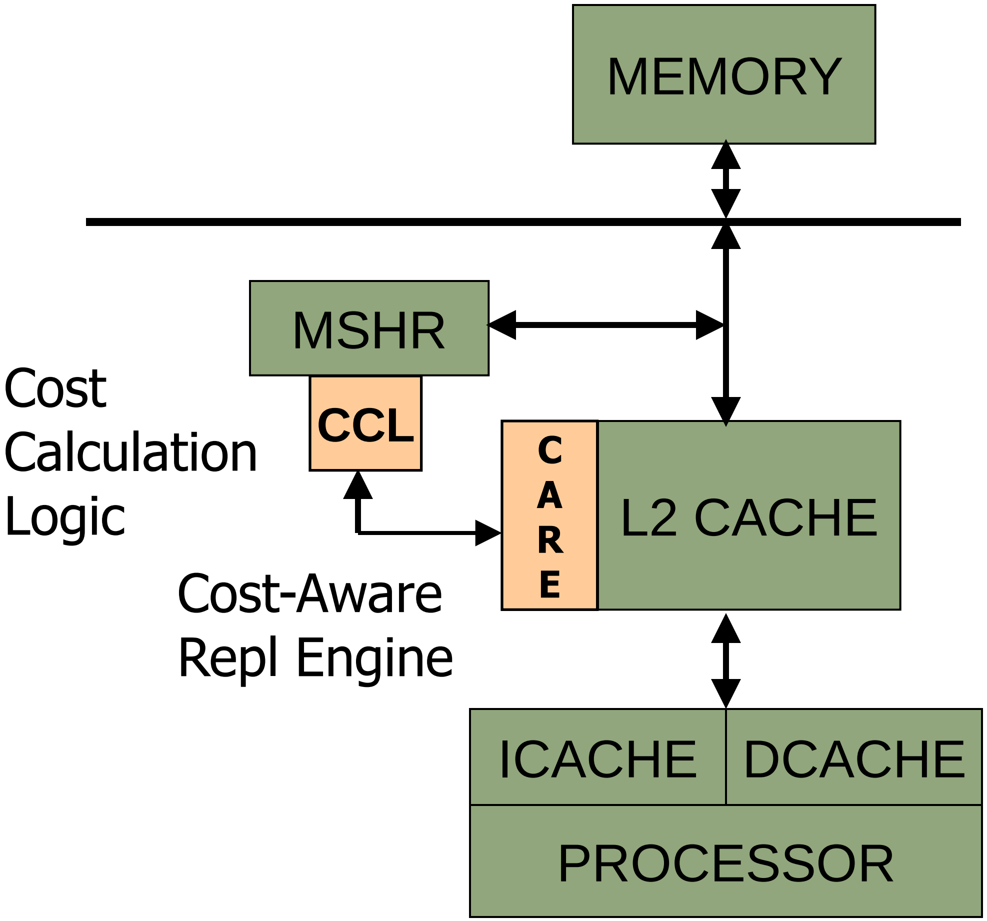
🟦 $\delta < 60$ 🟥 $\delta > 120$

🟩 $59 < \delta < 120$



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

The Framework



Cost Calculation Logic

Cost-Aware Repl Engine

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

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

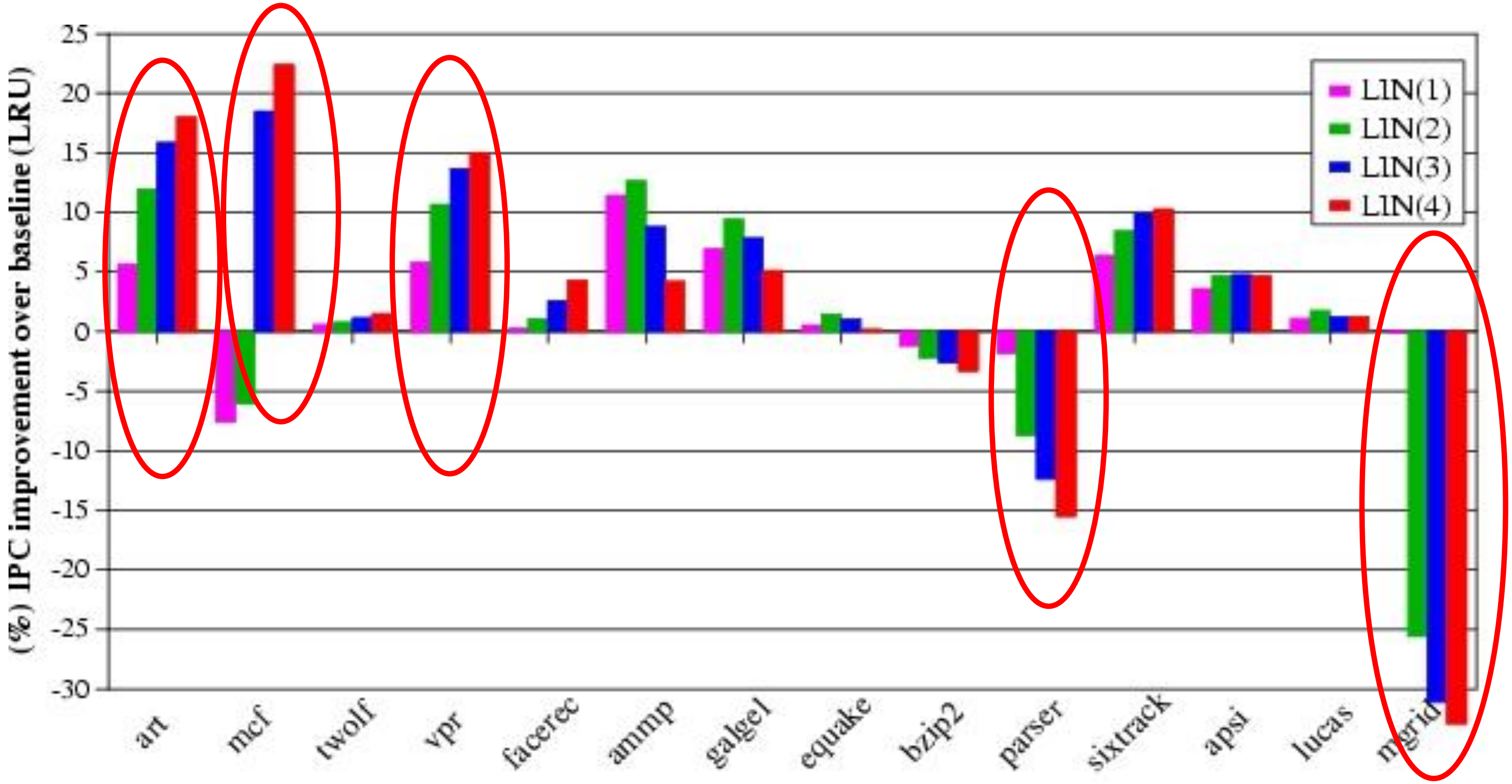
- ❑ Decisions based only on cost and no recency hurt performance. Cache stores useless high cost blocks

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

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

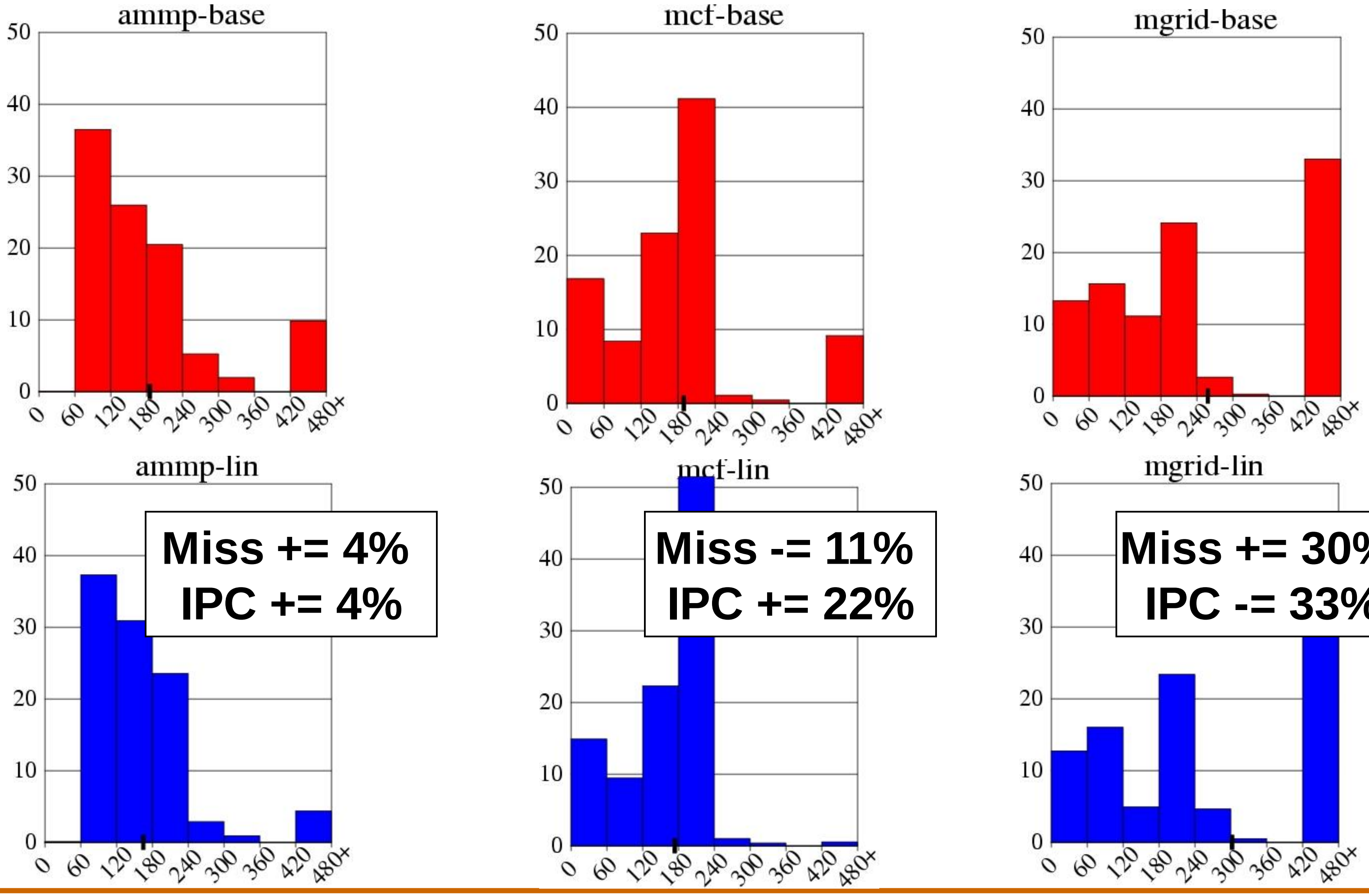
S = significance of cost. Recency(i) = position in LRU stack
cost(i) = quantized cost

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

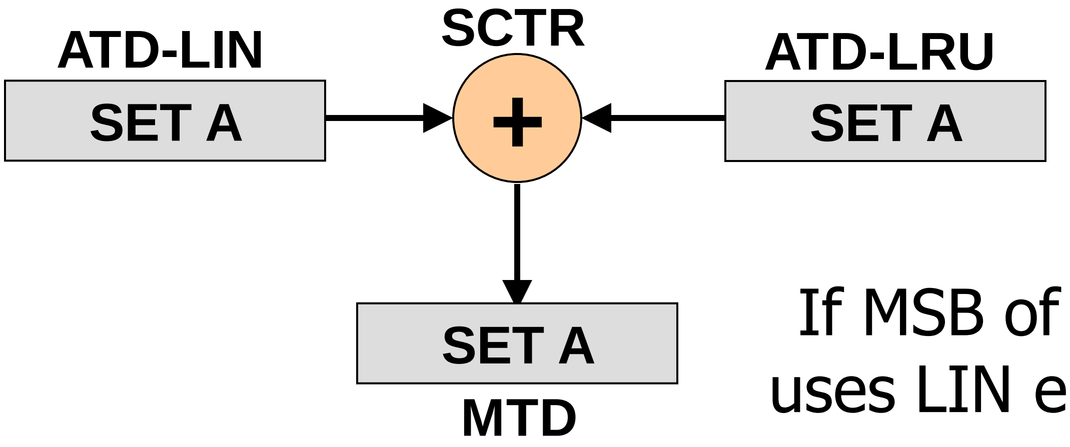
- ❑ Introduction

- ❑ 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



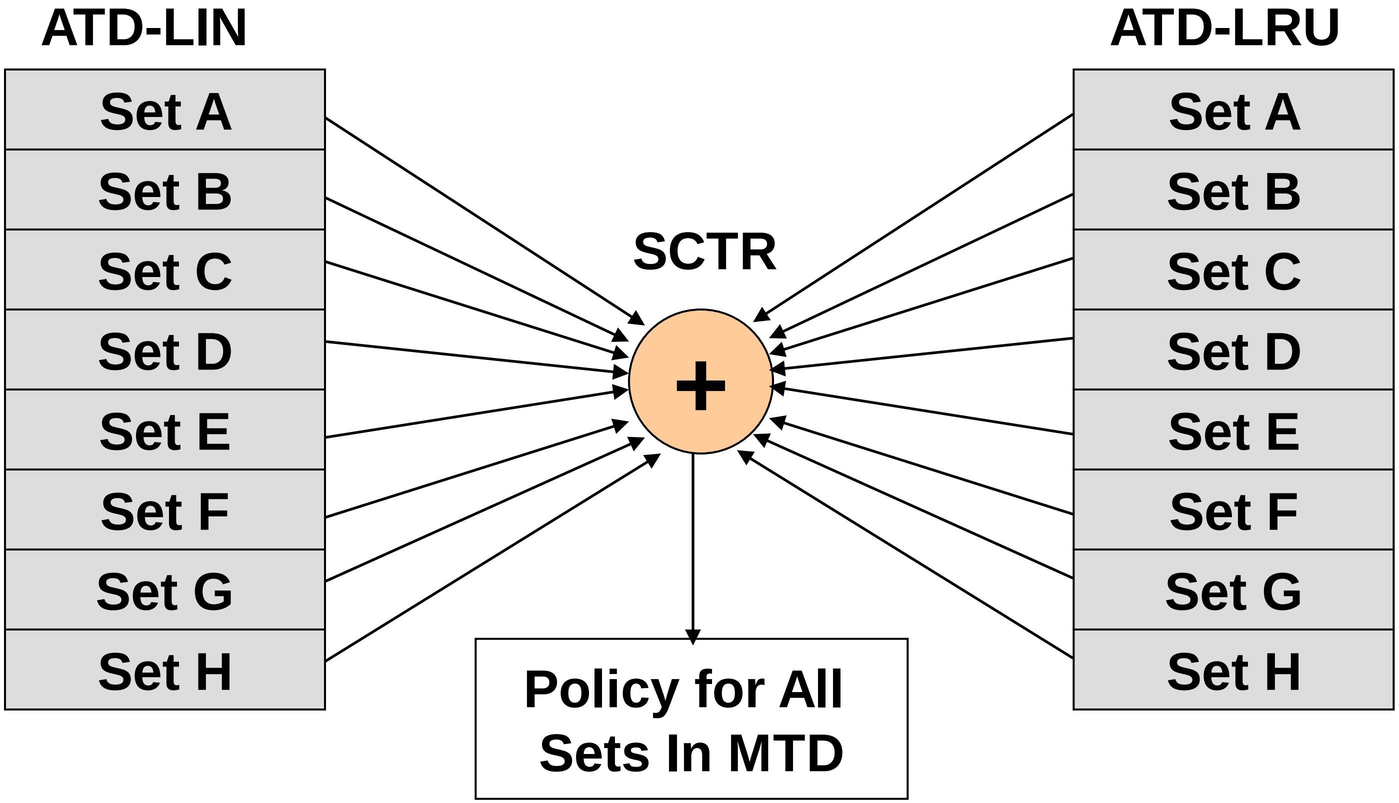
If MSB of SCTR is 1, MTD uses LIN else MTD use LRU

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

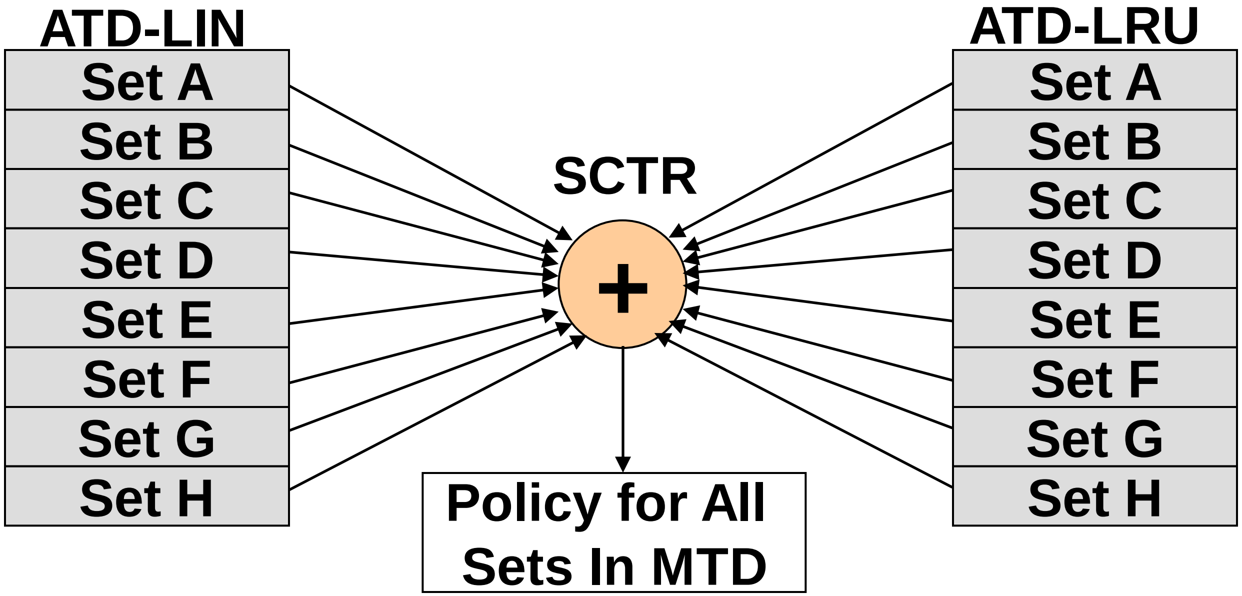
Implementing TSEL on a per-set basis is expensive

Counter overhead can be reduced by using a global counter



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?

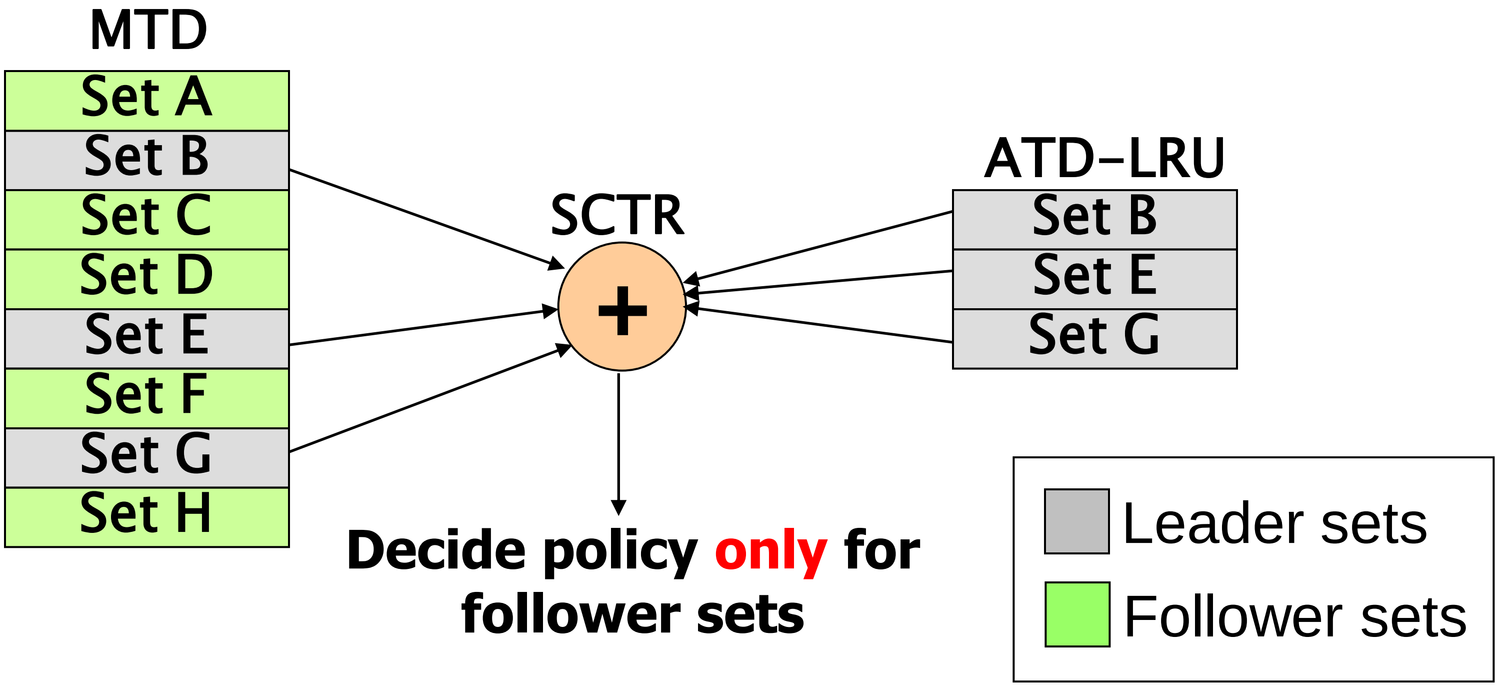
❑ Bounds using analytical model and simulation (in paper)

❑ 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

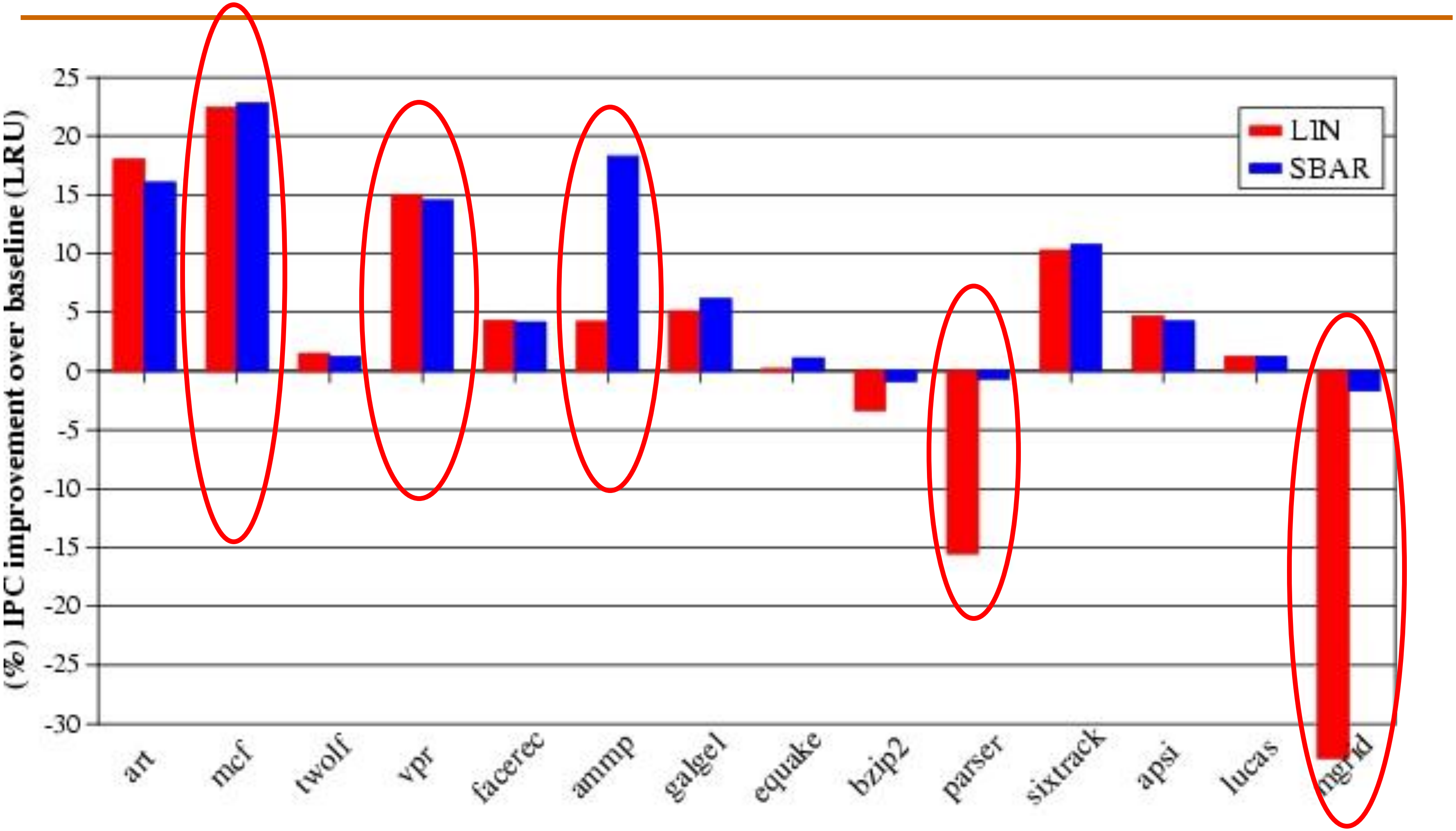
ATD overhead can further be reduced by using MTD to always simulate one of the policies (say LIN)

Sampling Based Adaptive Replacement (SBAR)

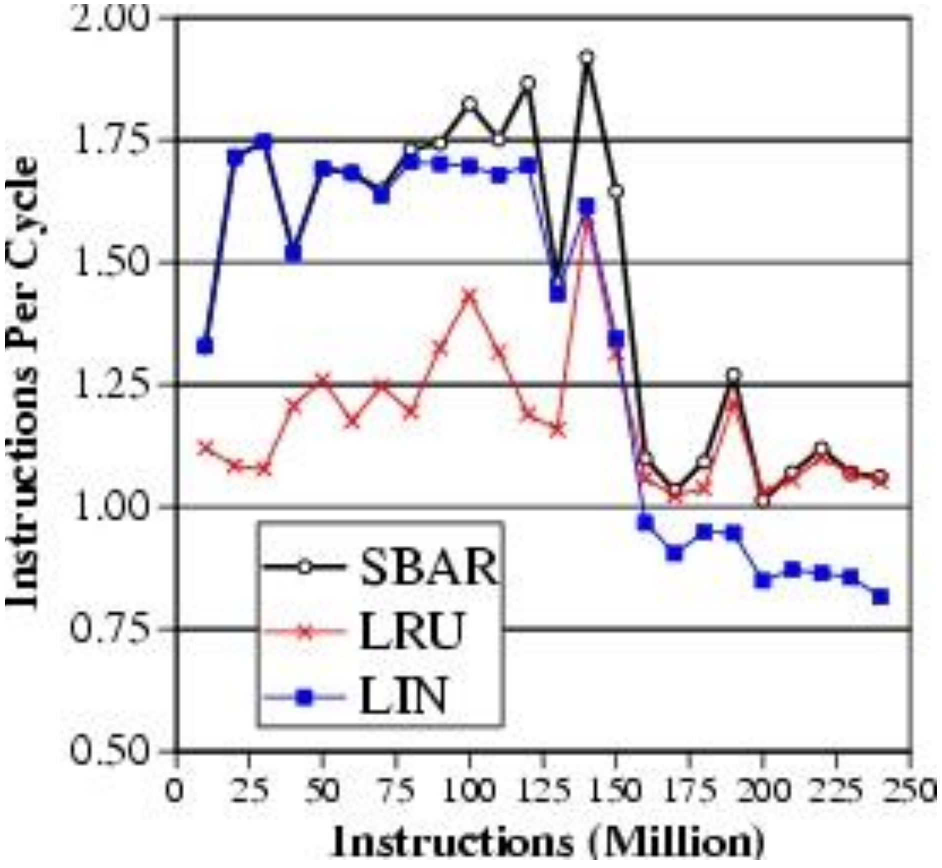


The storage overhead of SBAR is less than 2KB
(0.2% of the baseline 1MB cache)

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

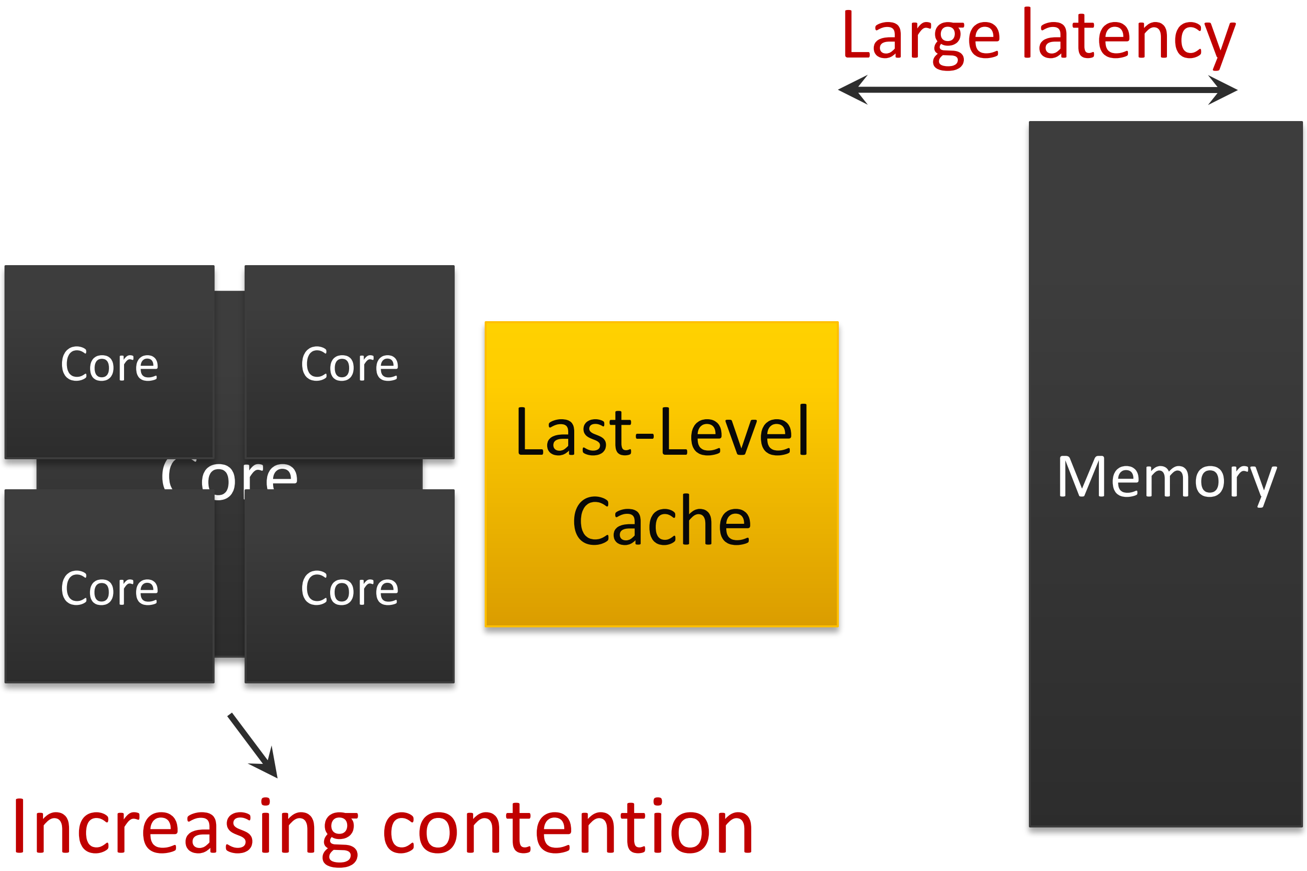
The Evicted-Address Filter

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

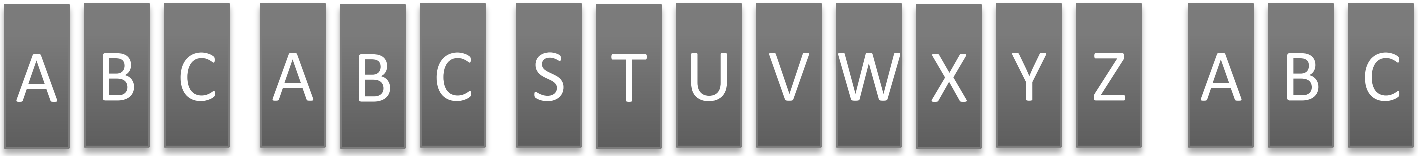


Effective cache utilization is important

Reuse Behavior of Cache Blocks

Different blocks have different reuse behavior

Access Sequence:

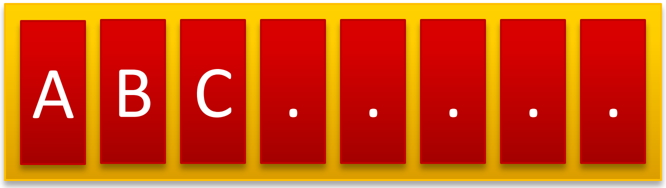


A B C A B C S T U V W X Y Z A B C

🟥 High-reuse block

🟦 Low-reuse block

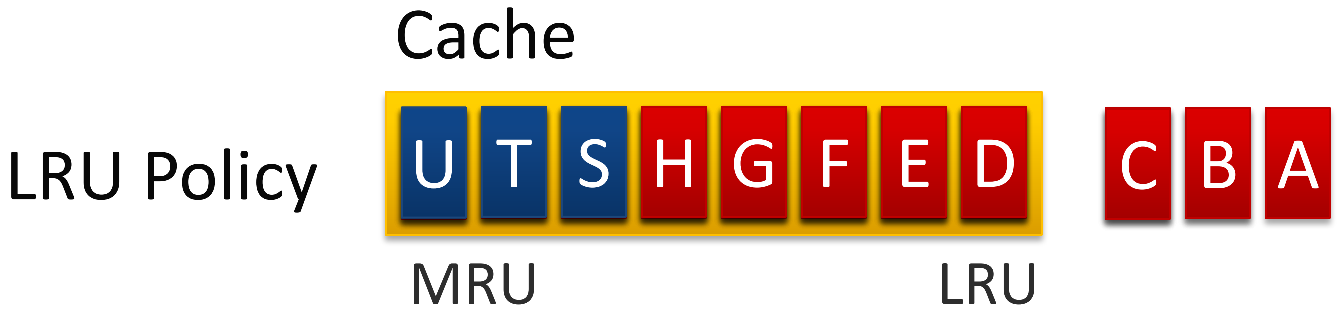
Ideal Cache



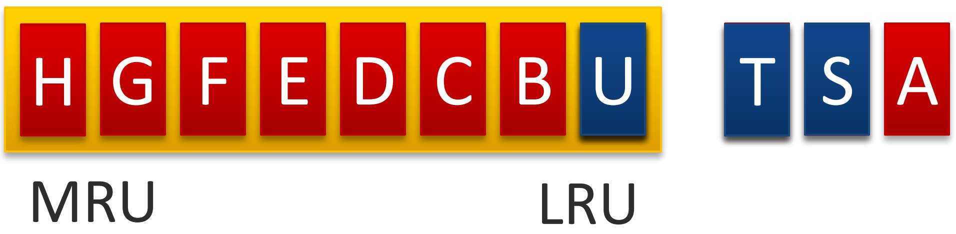
A B C . . . . .

Cache Pollution

Problem: Low-reuse blocks evict high-reuse blocks



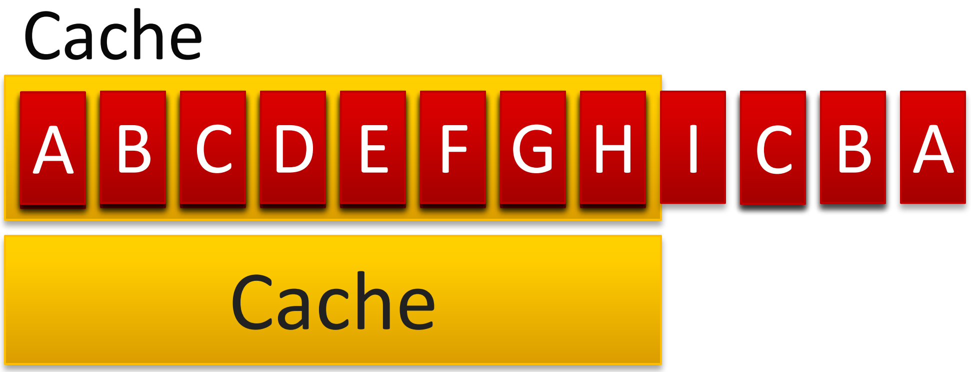
Prior work: Predict reuse behavior of missed blocks.
Insert low-reuse blocks at LRU position.



Cache Thrashing

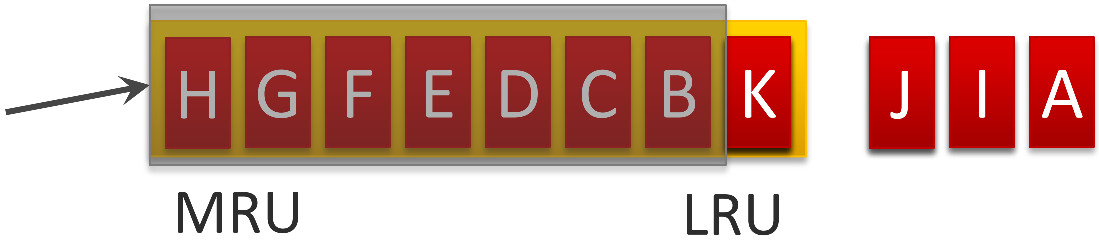
Problem: High-reuse blocks evict each other

LRU Policy



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

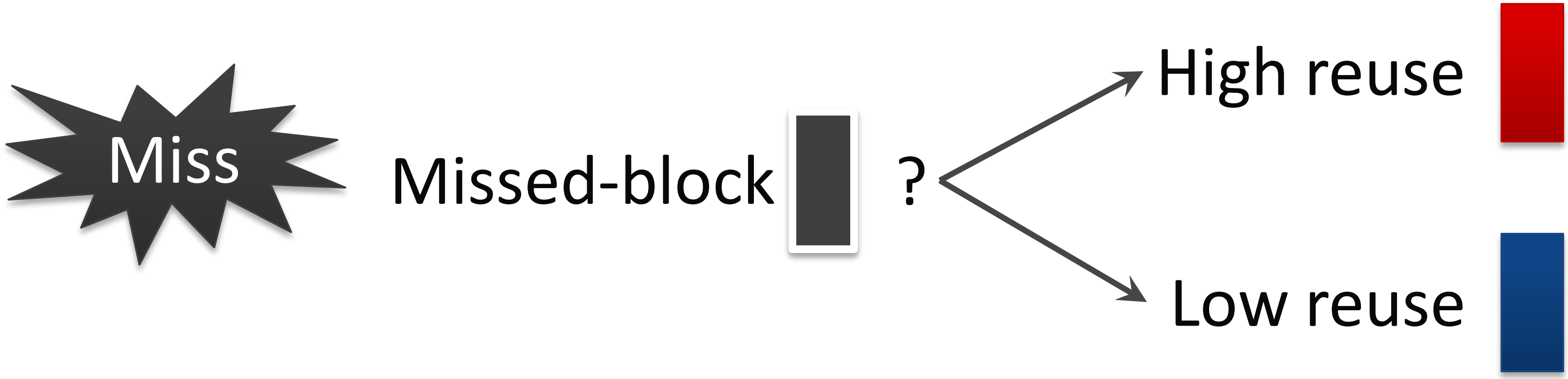
No mechanism to distinguish high-reuse blocks from low-reuse blocks

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

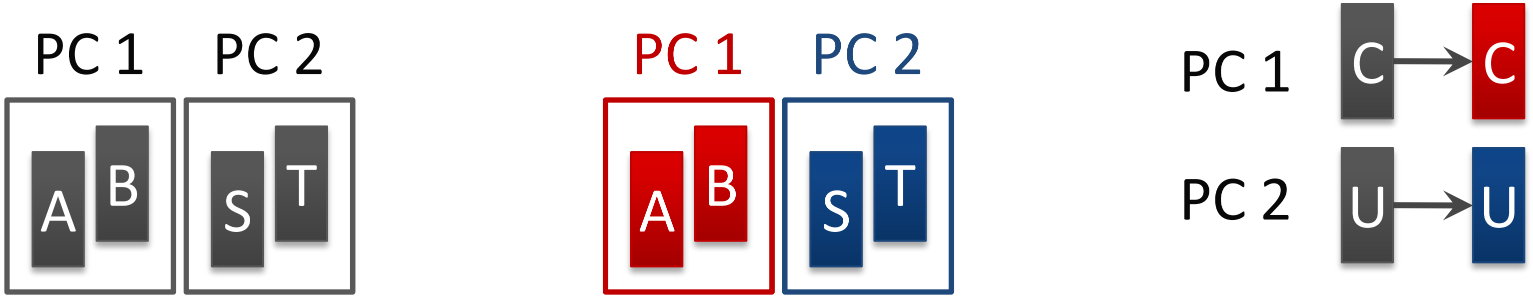
Prior Work on Reuse Prediction

Use program counter or memory region information.

1. Group Blocks

2. Learn group behavior

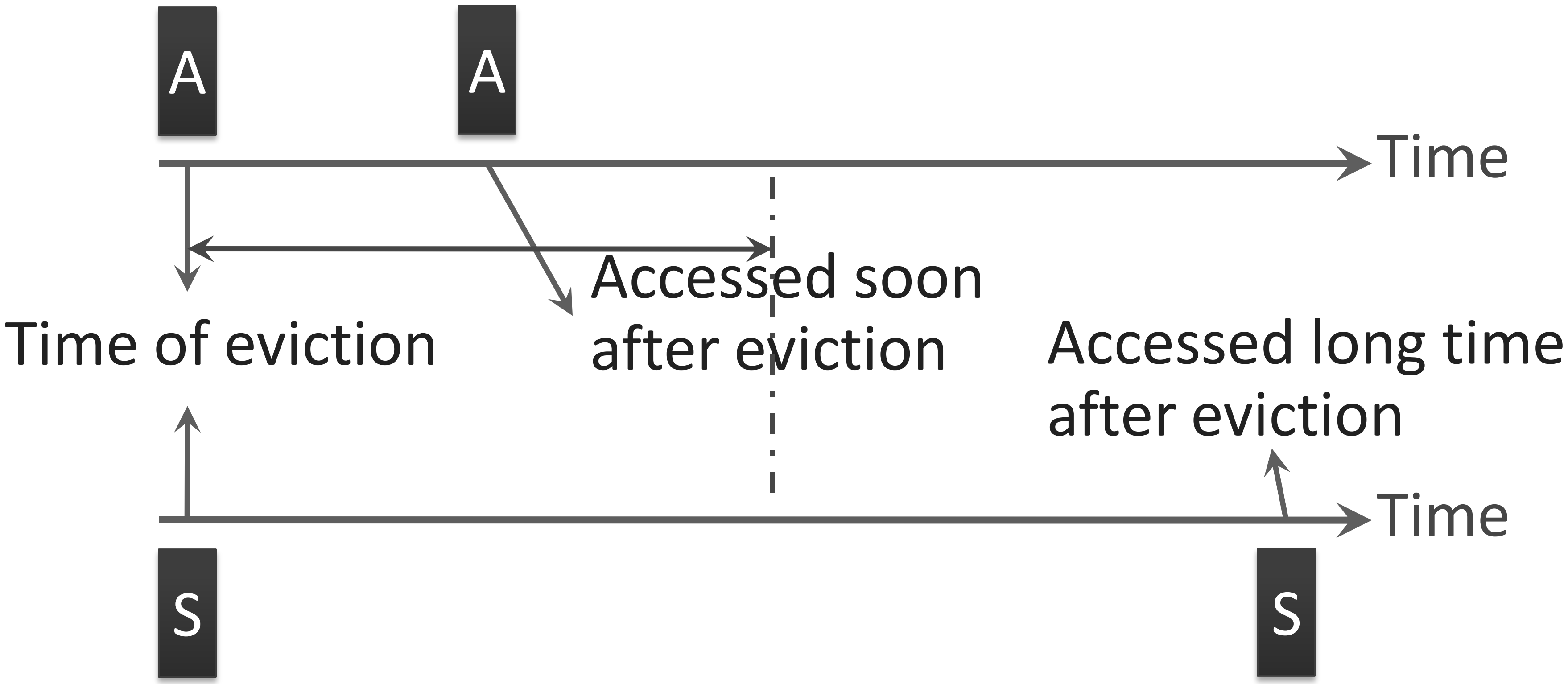
3. Predict reuse



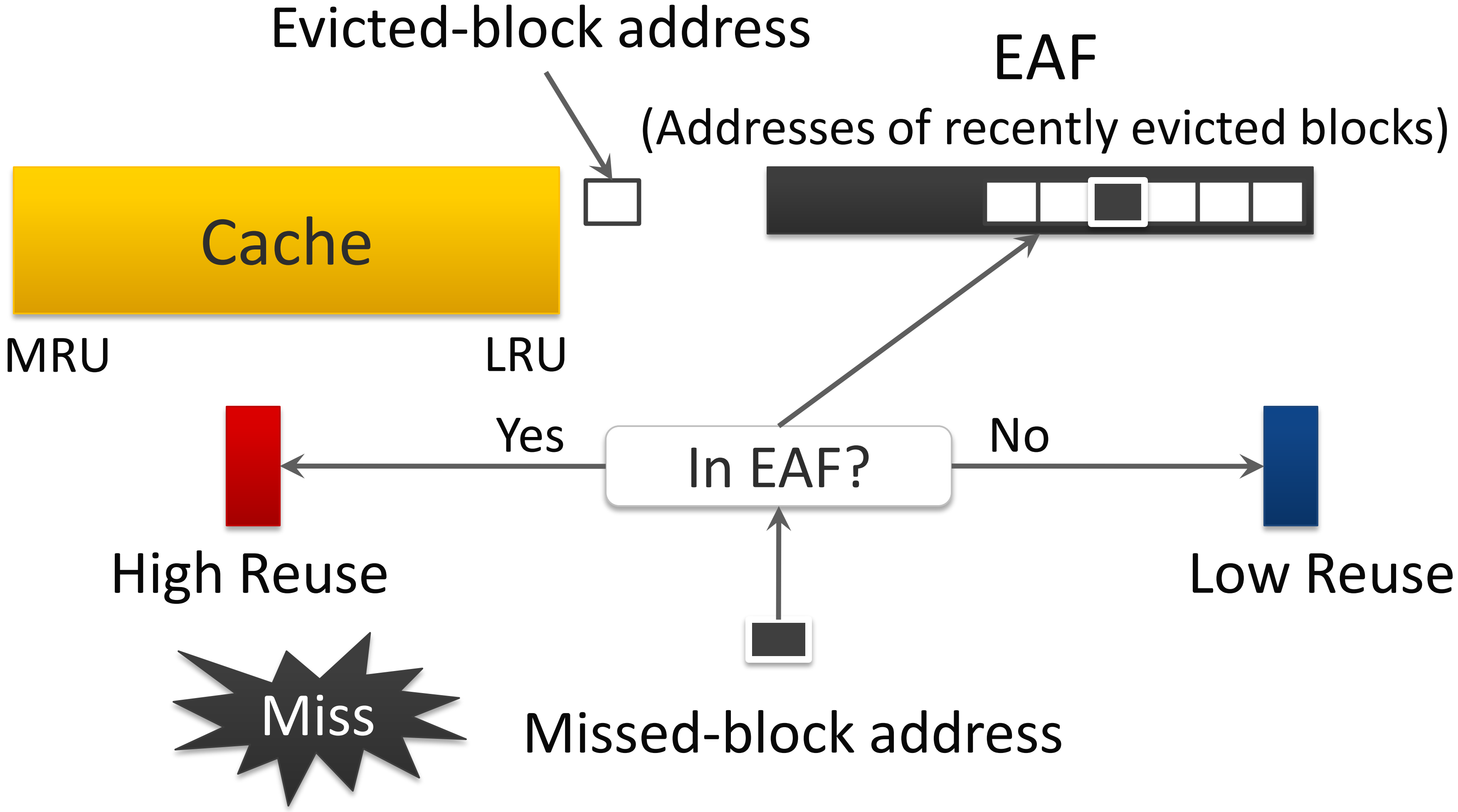
1. Same group ↛ same reuse behavior
2. No control over number of high-reuse blocks

Our Approach: Per-block Prediction

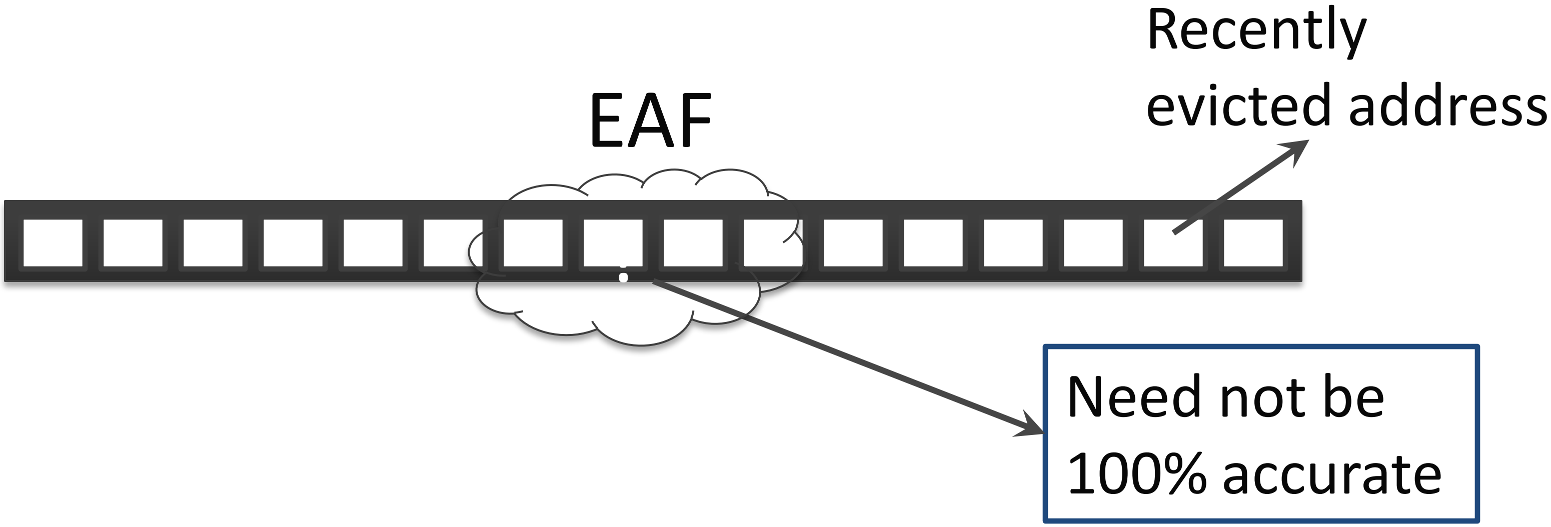
💡 Use recency of eviction to predict reuse


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

Evicted-Address Filter (EAF)

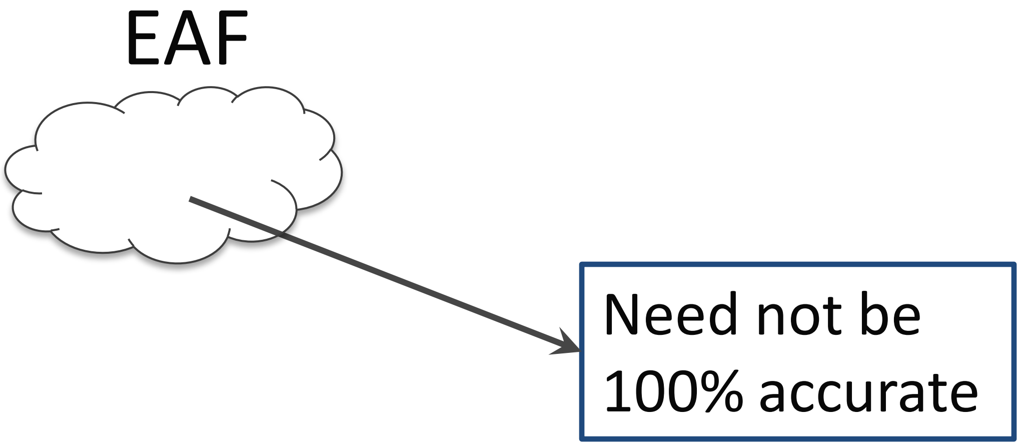


Naïve Implementation: Full Address Tags



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

Low-Cost Implementation: Bloom Filter


EAF

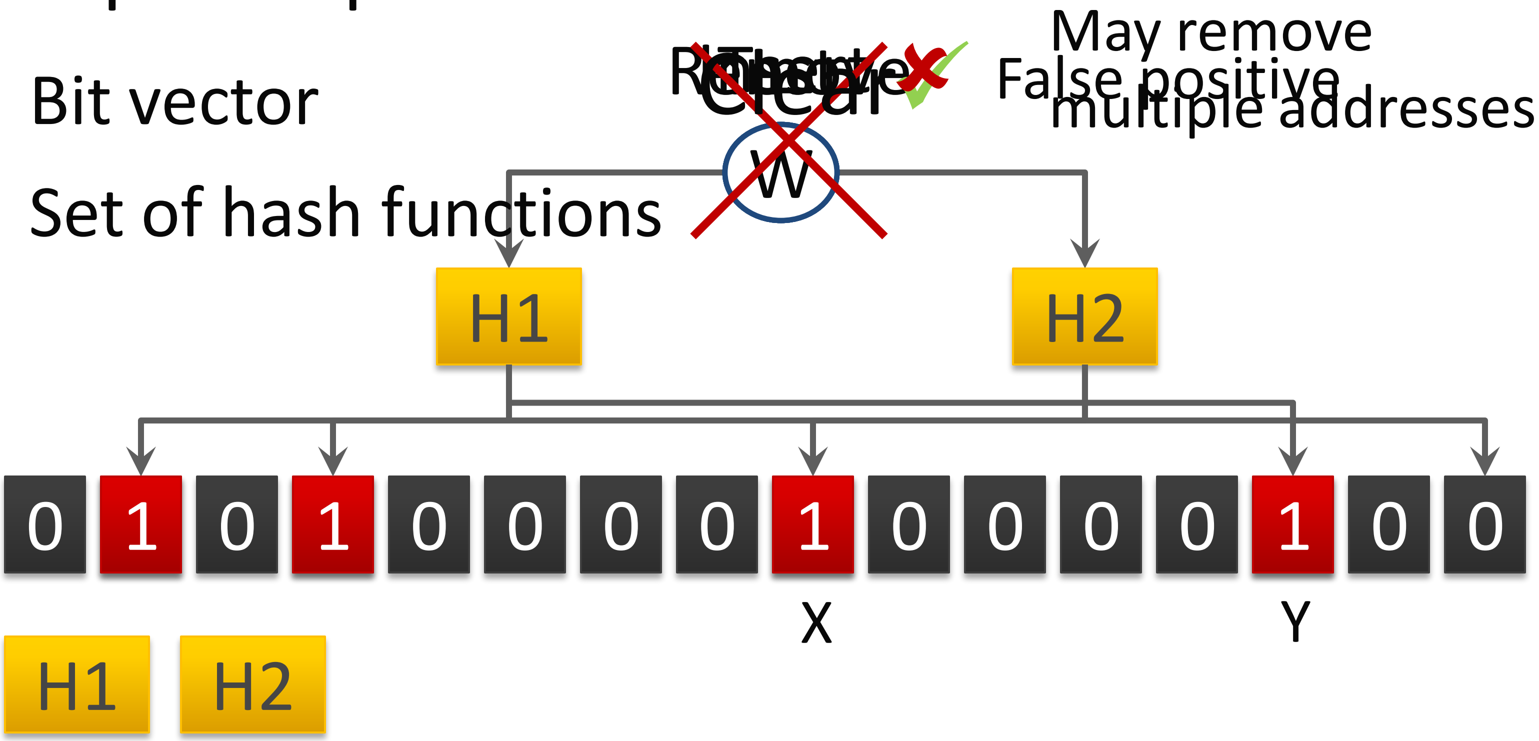
Need not be 100% accurate

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

Bloom Filter

Compact representation of a set

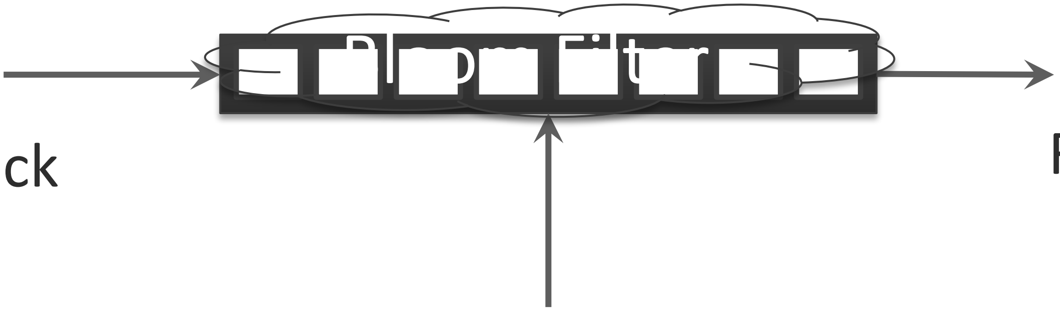
1. Bit vector
2. Set of hash functions



Inserted Elements: X Y

EAF using a Bloom Filter

EAF



✓ **Insert**

Evicted-block address

✓ **Test**

Missed-block address

1 ~~**Remove**~~ ✗

~~If present~~

2 **Clear**

when full

~~**Remove**~~ ✗

~~FIFO address when full~~

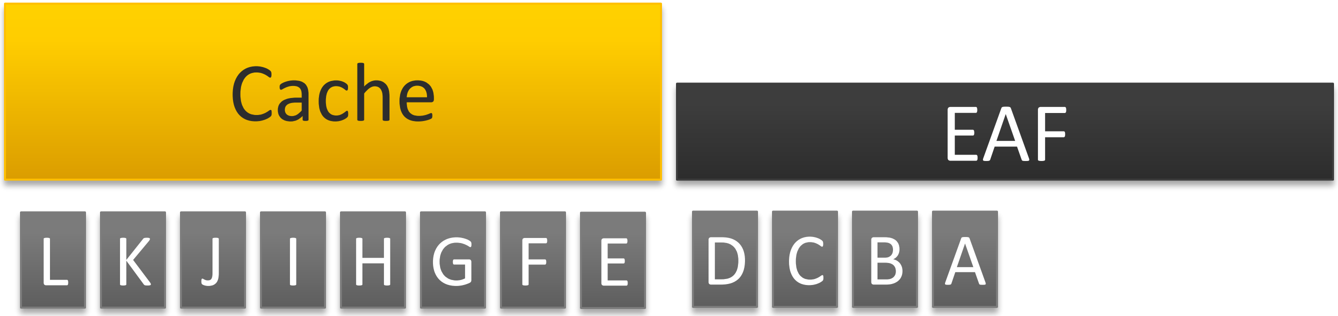
Bloom-filter EAF: 4x reduction in storage overhead, 1.47% compared to cache size

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

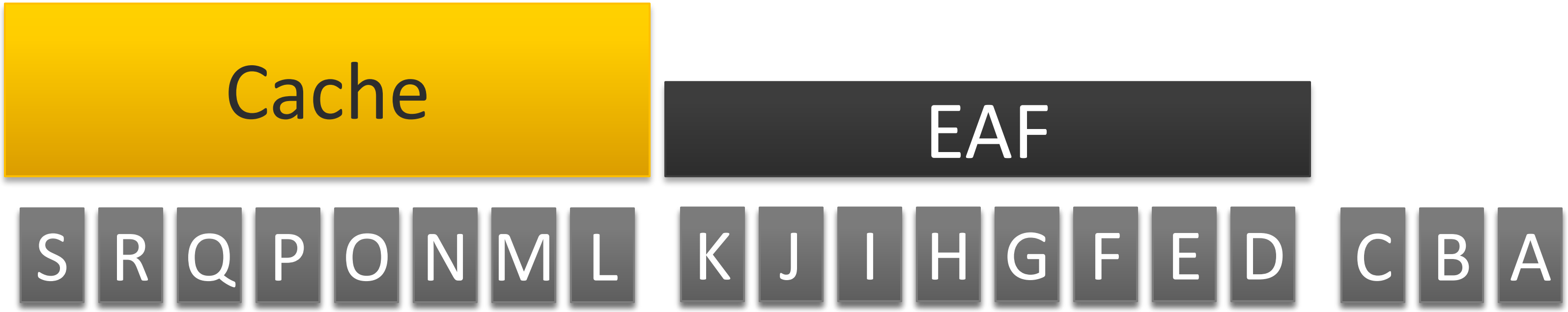


Cache

EAF

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

2 Cache + EAF < Working Set



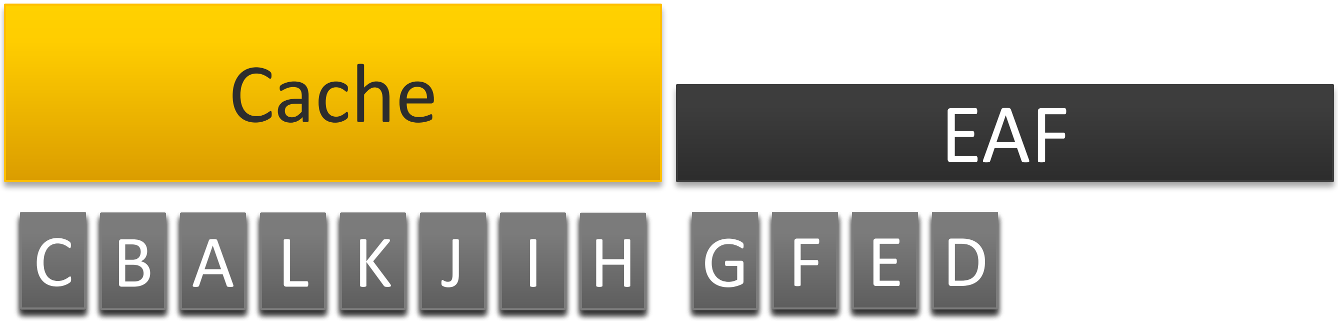
Cache

EAF

S R Q P O N M L K J I H G F E D C B A

Large Working Set: Case 1

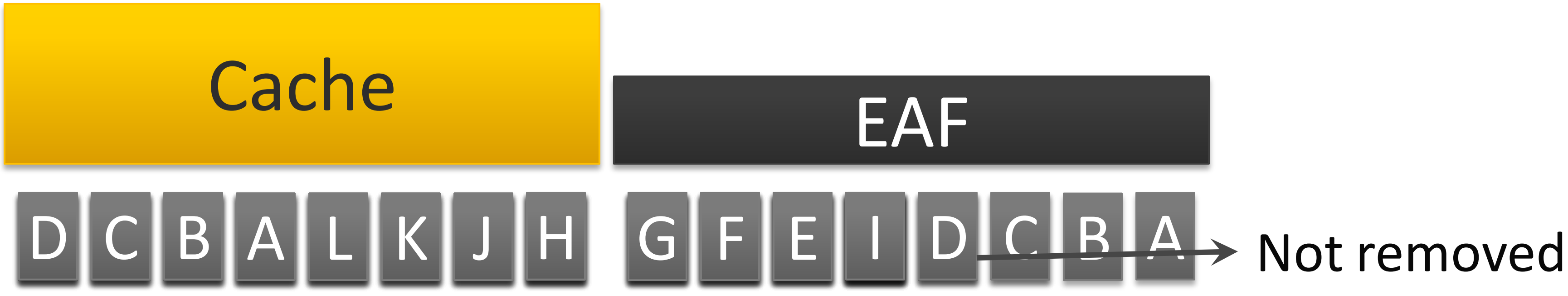
Cache < Working set < Cache + EAF



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

Large Working Set: Case 1

Cache < Working set < Cache + EAF



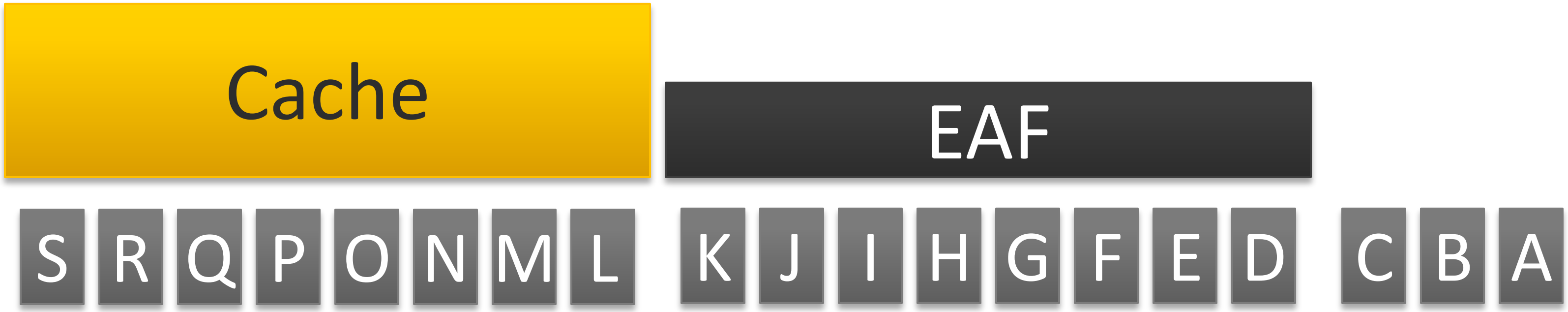
Not present in the EAF

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



Cache

EAF

S R Q P O N M L K J I H G F E D C B A

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**

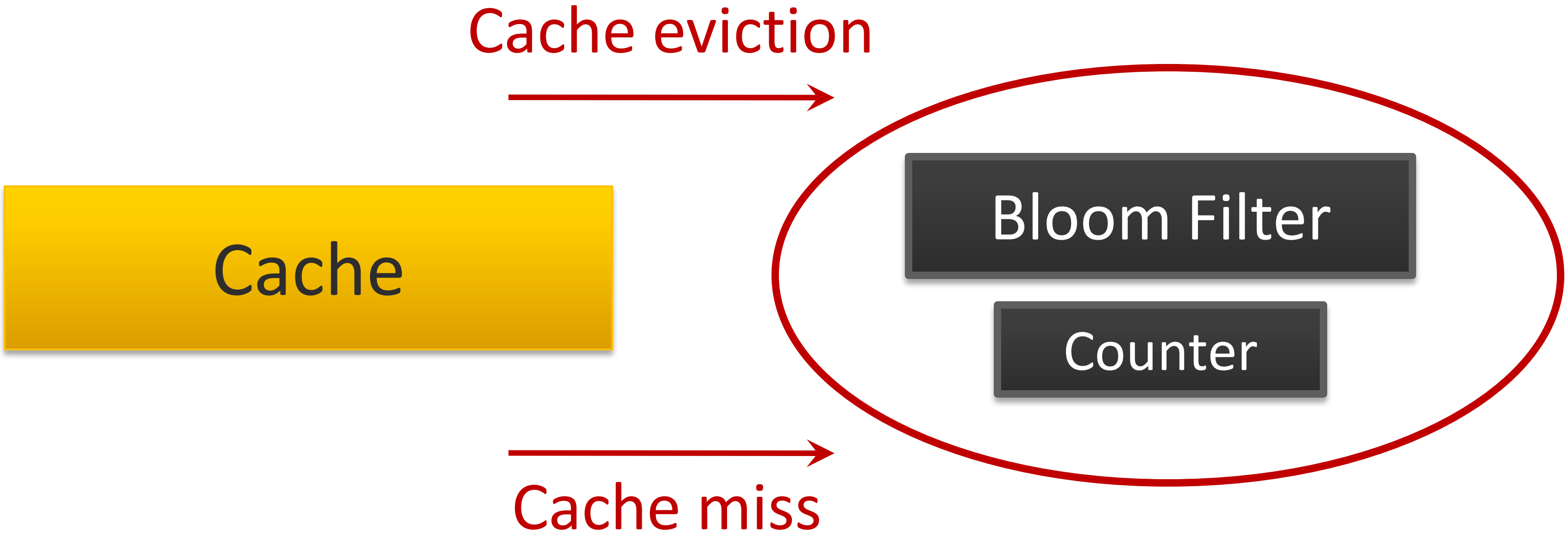
Test if address is present in filter

Yes, insert at MRU. No, insert with BIP

Outline

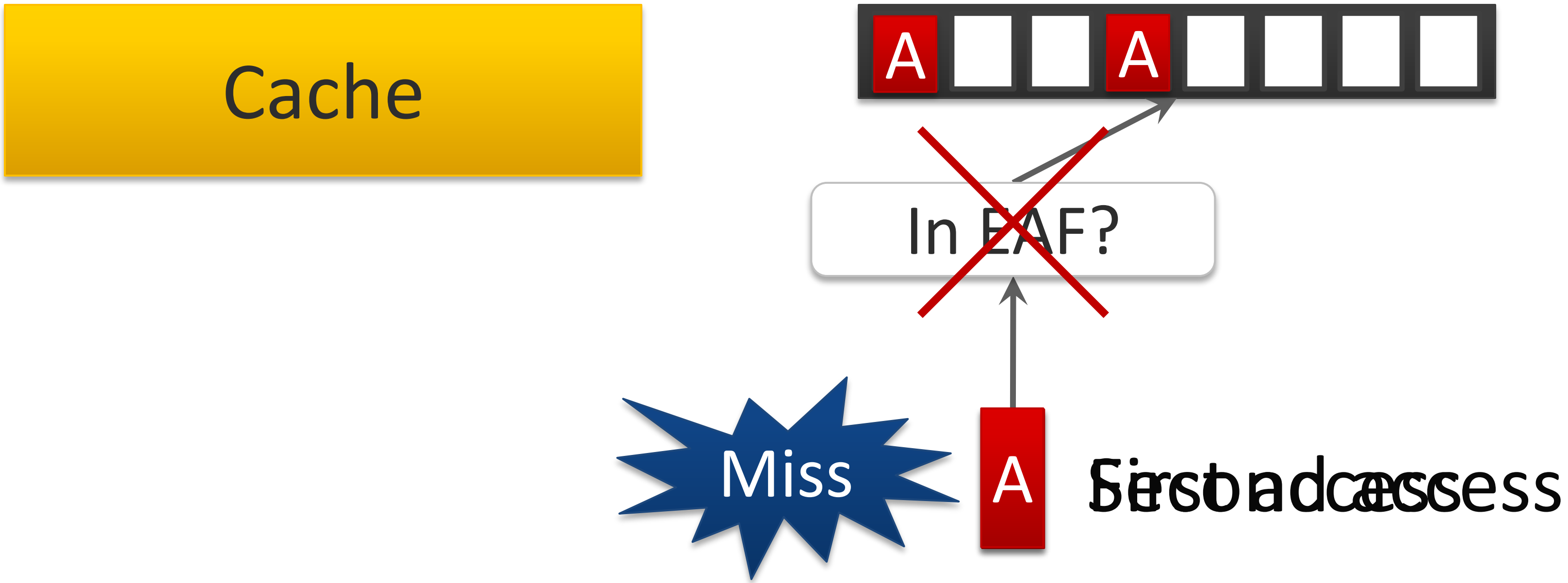
- Background and Motivation
- Evicted-Address Filter
  – Reuse Prediction
  – Thrash Resistance
- Final Design
- Advantages and Disadvantages
- Evaluation
- Conclusion

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

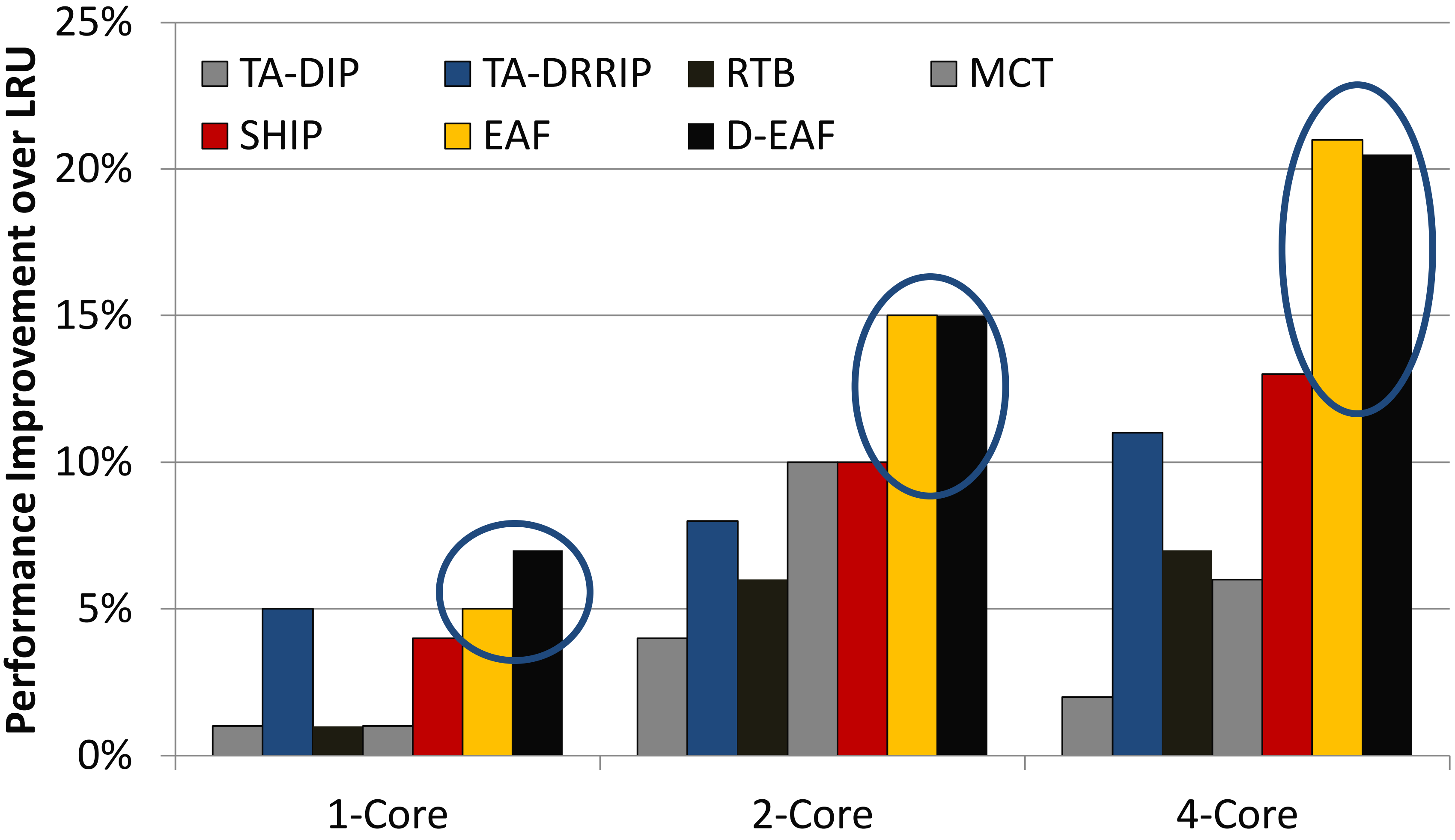
Addressing Cache Thrashing

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

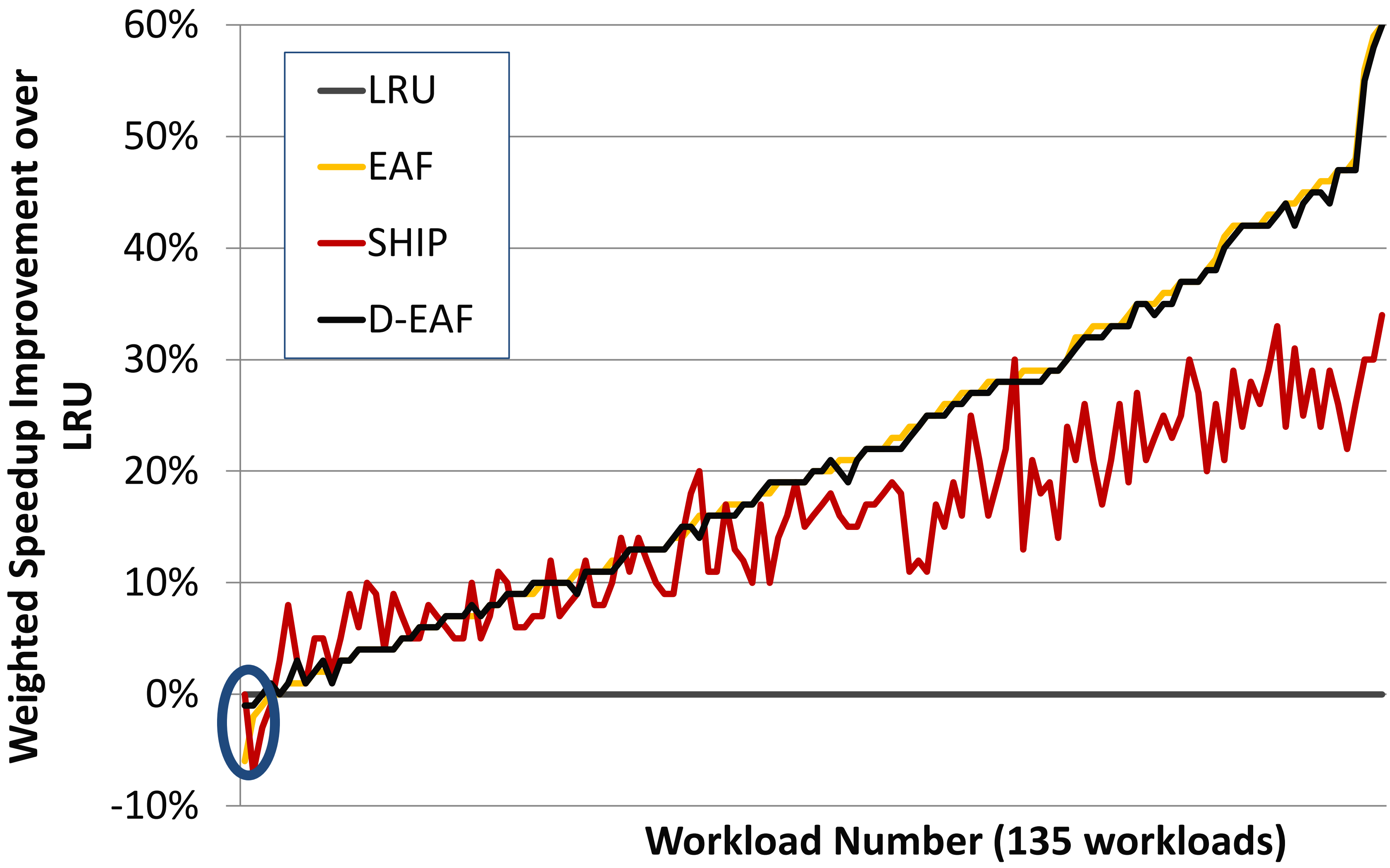
- Use set dueling to determine thrashing applications

- No mechanism to filter low-reuse blocks ⟹ Pollution

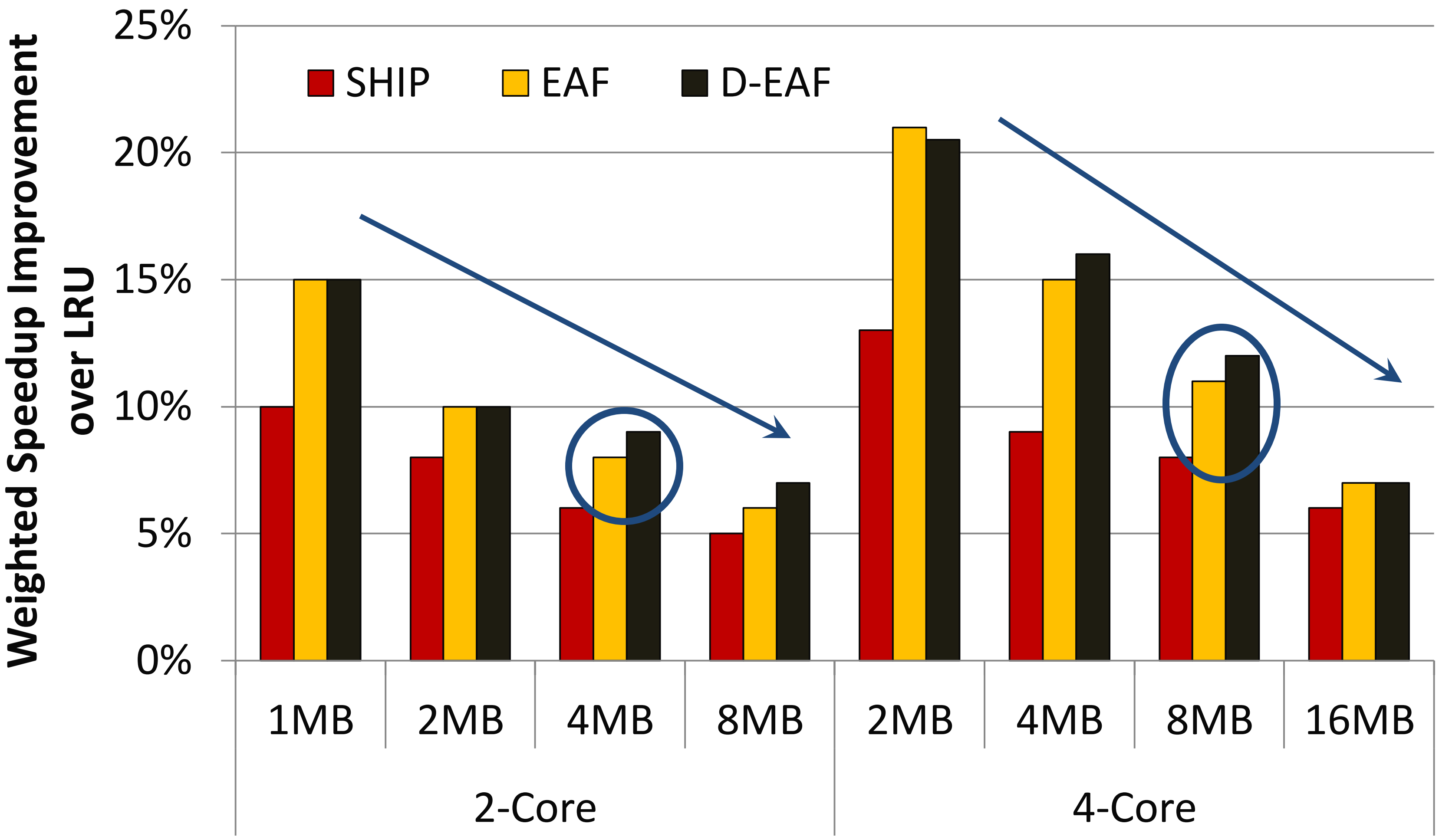
Results – Summary



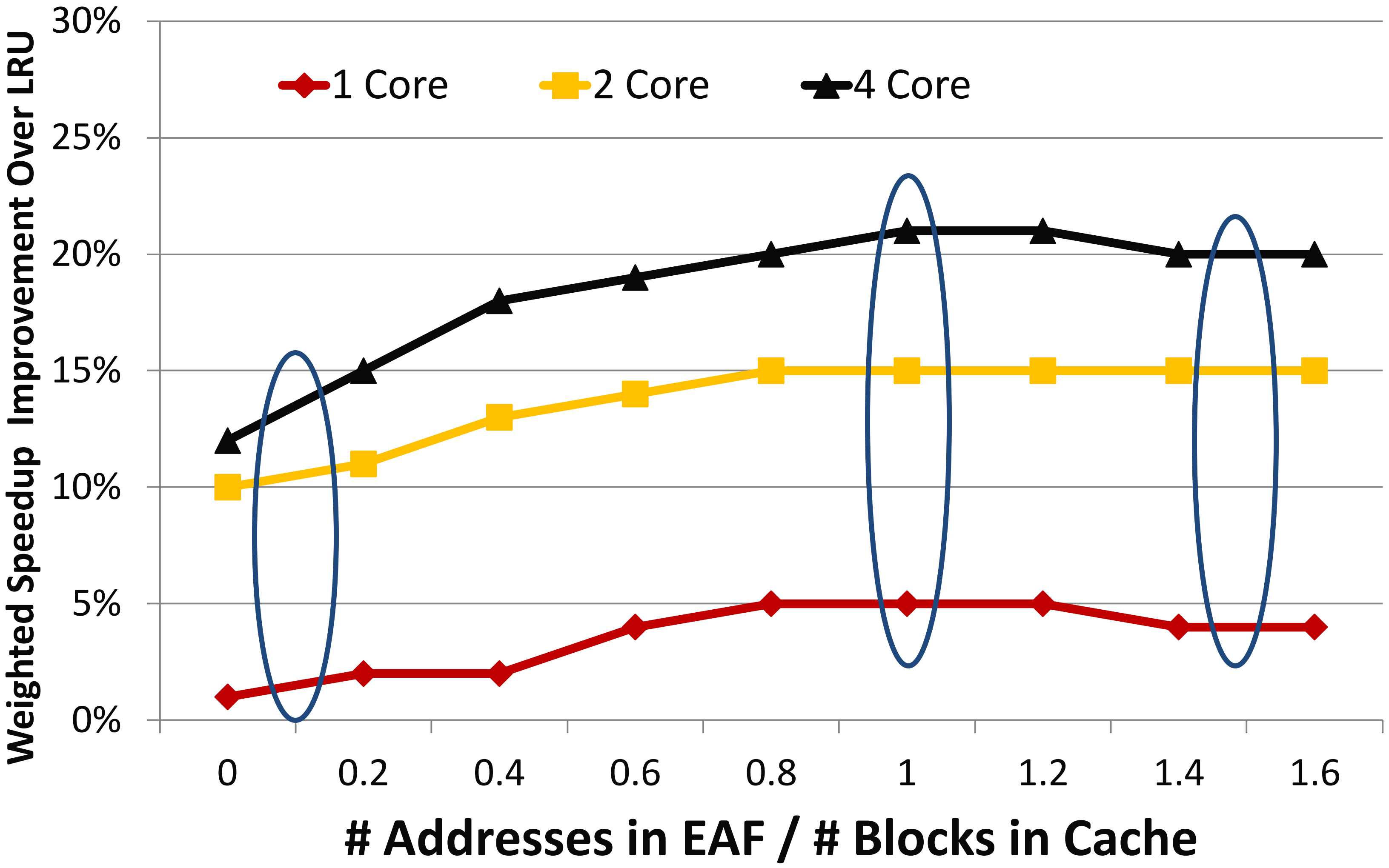
4-Core: Performance



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

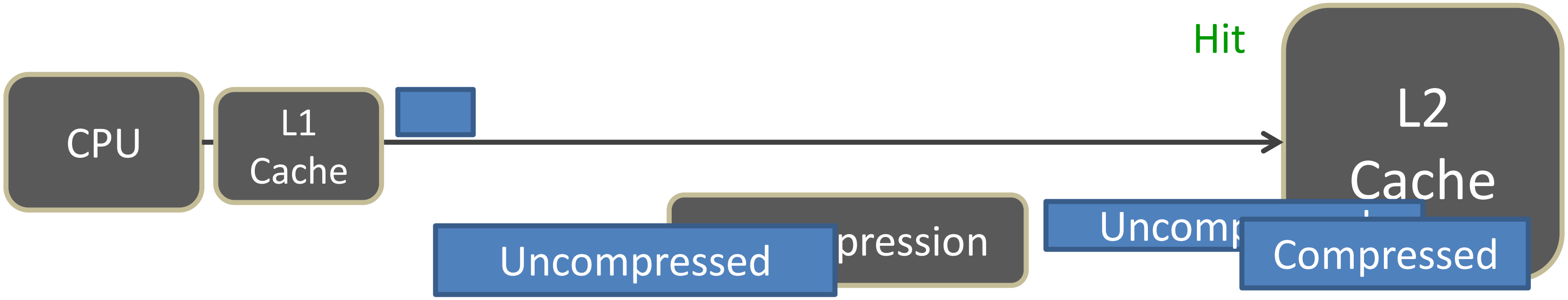
Significant redundancy in data:

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

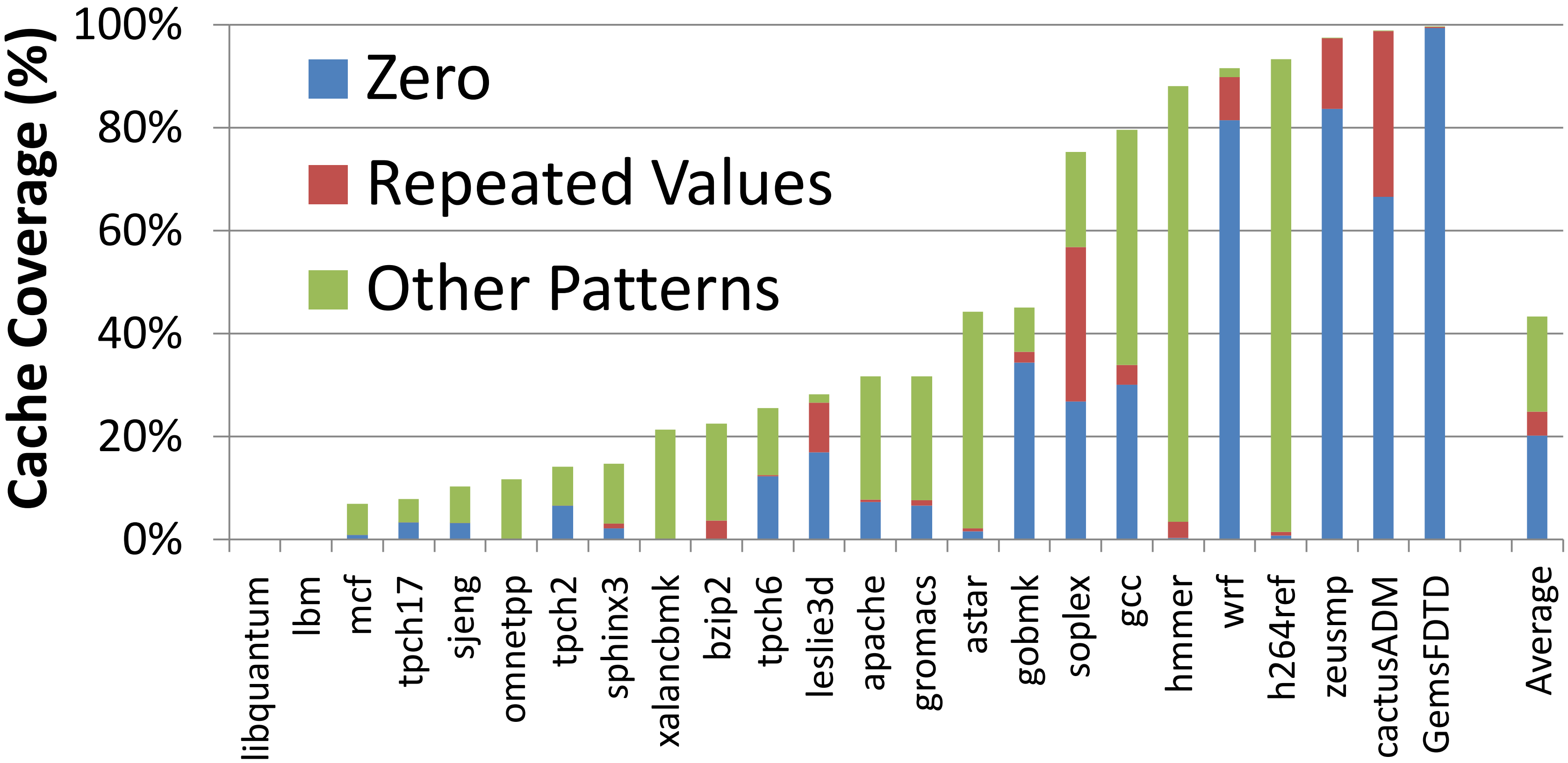
0x*000000***00**	0x*000000***0B**	0x*000000***03**	0x*000000***04**	…

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



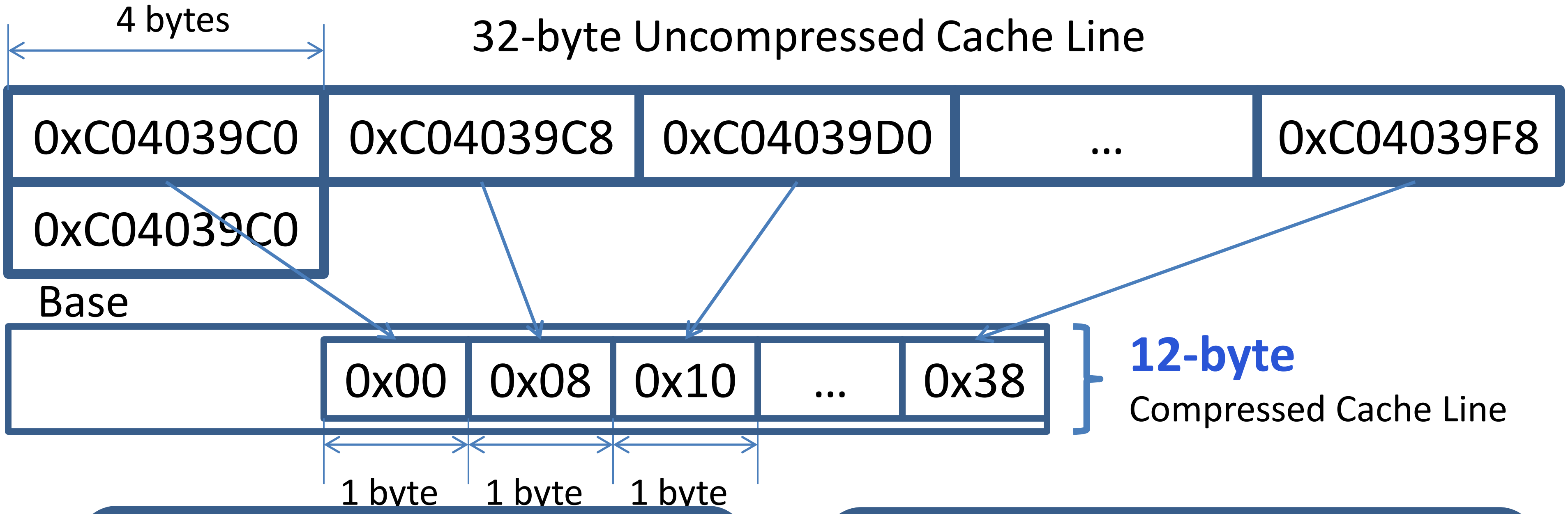
43% of the cache lines belong to key patterns

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



4 bytes

32-byte Uncompressed Cache Line

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

0xC04039C0

Base

0x00 | 0x08 | 0x10 | … | 0x38

12-byte

Compressed Cache Line

1 byte

1 byte

1 byte

✓ **Fast Decompression:**
vector addition

✓ **Simple Hardware:**
arithmetic and comparison

✓ **Effective:** good compression ratio

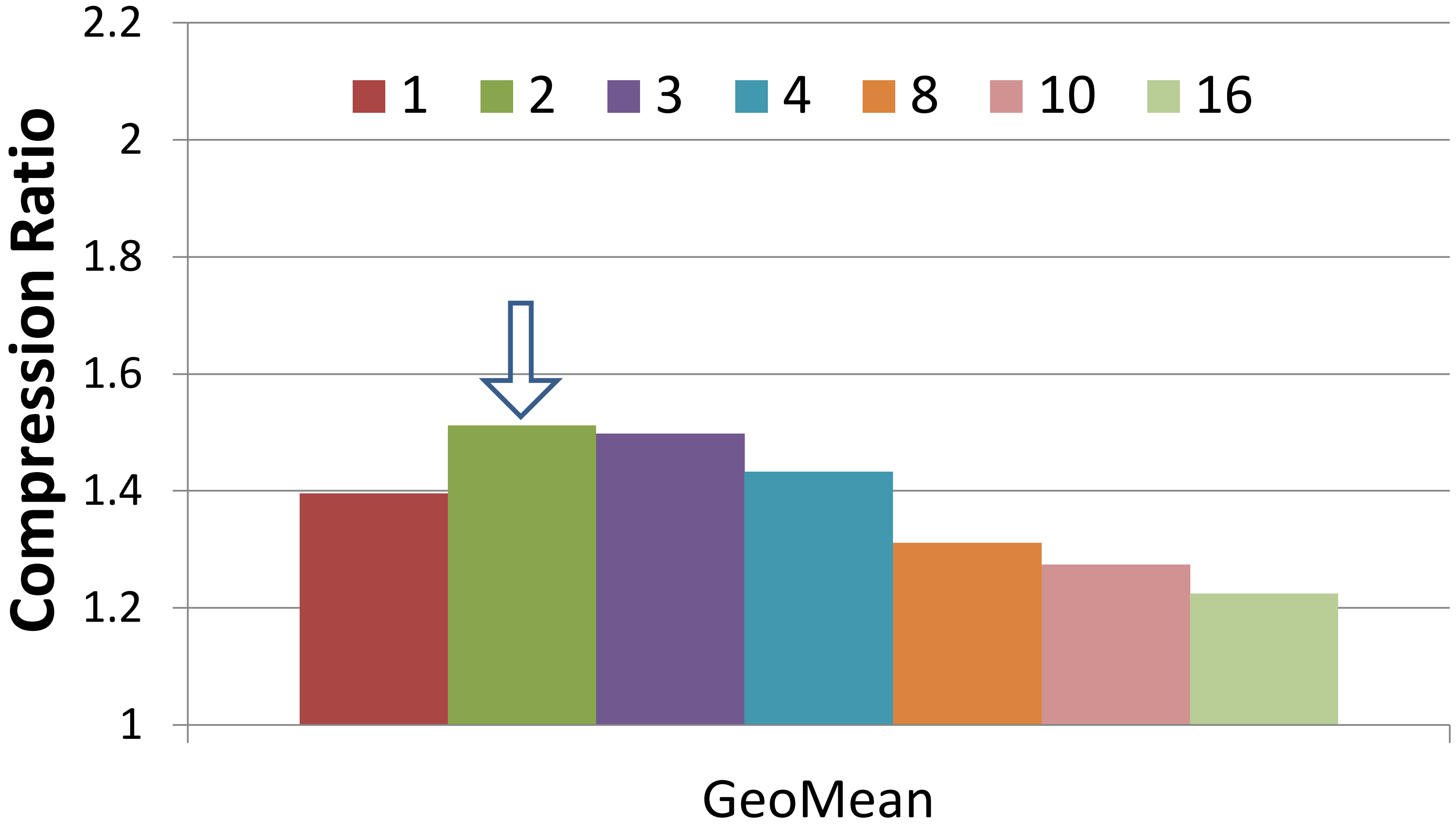
Can We Do Better?

- 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**

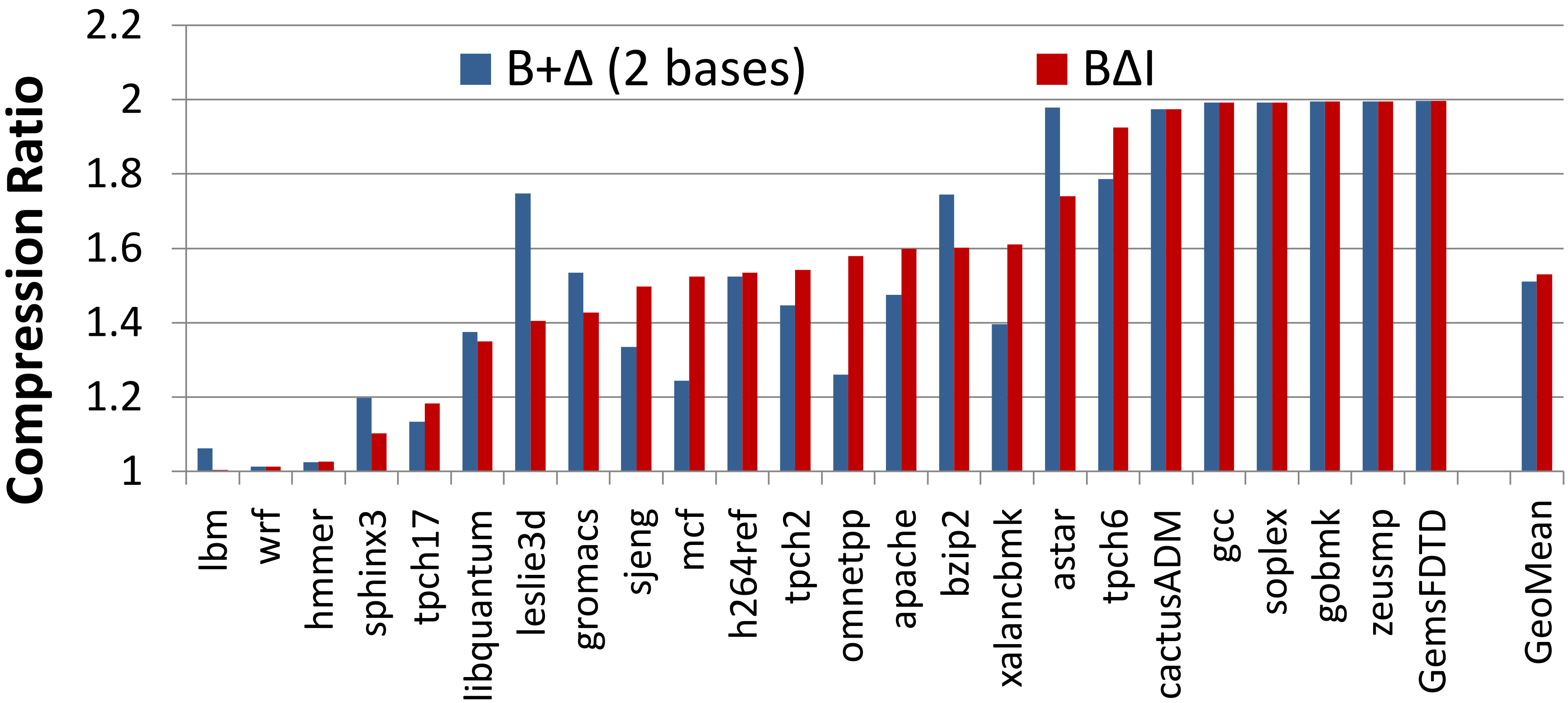
Advantages over 2 arbitrary bases:

– Better compression ratio

– Simpler compression logic

Base-Delta-Immediate (BΔI) Compression

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

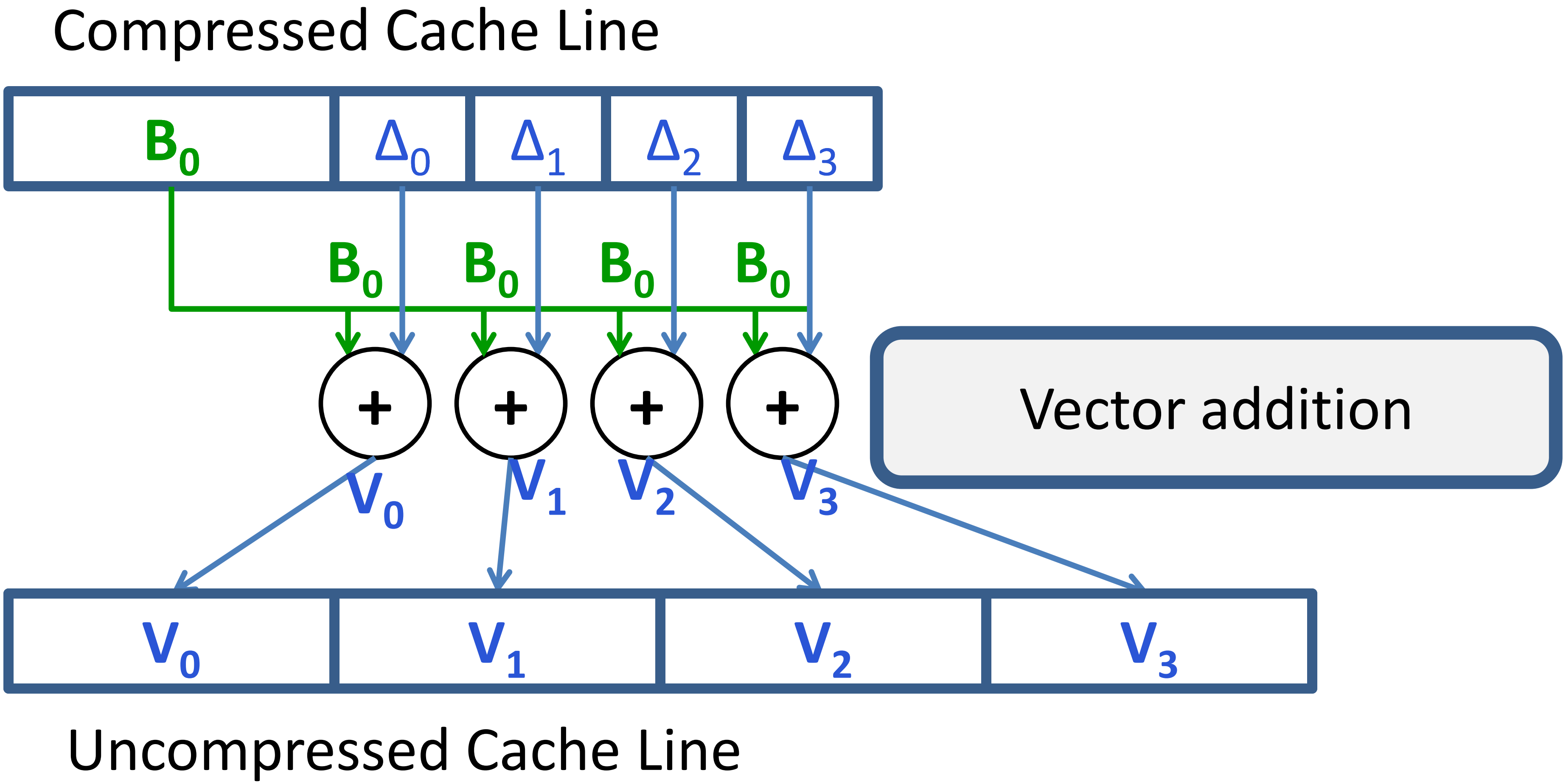


Average compression ratio is close, but **BΔI** is **simpler**

BΔI Implementation

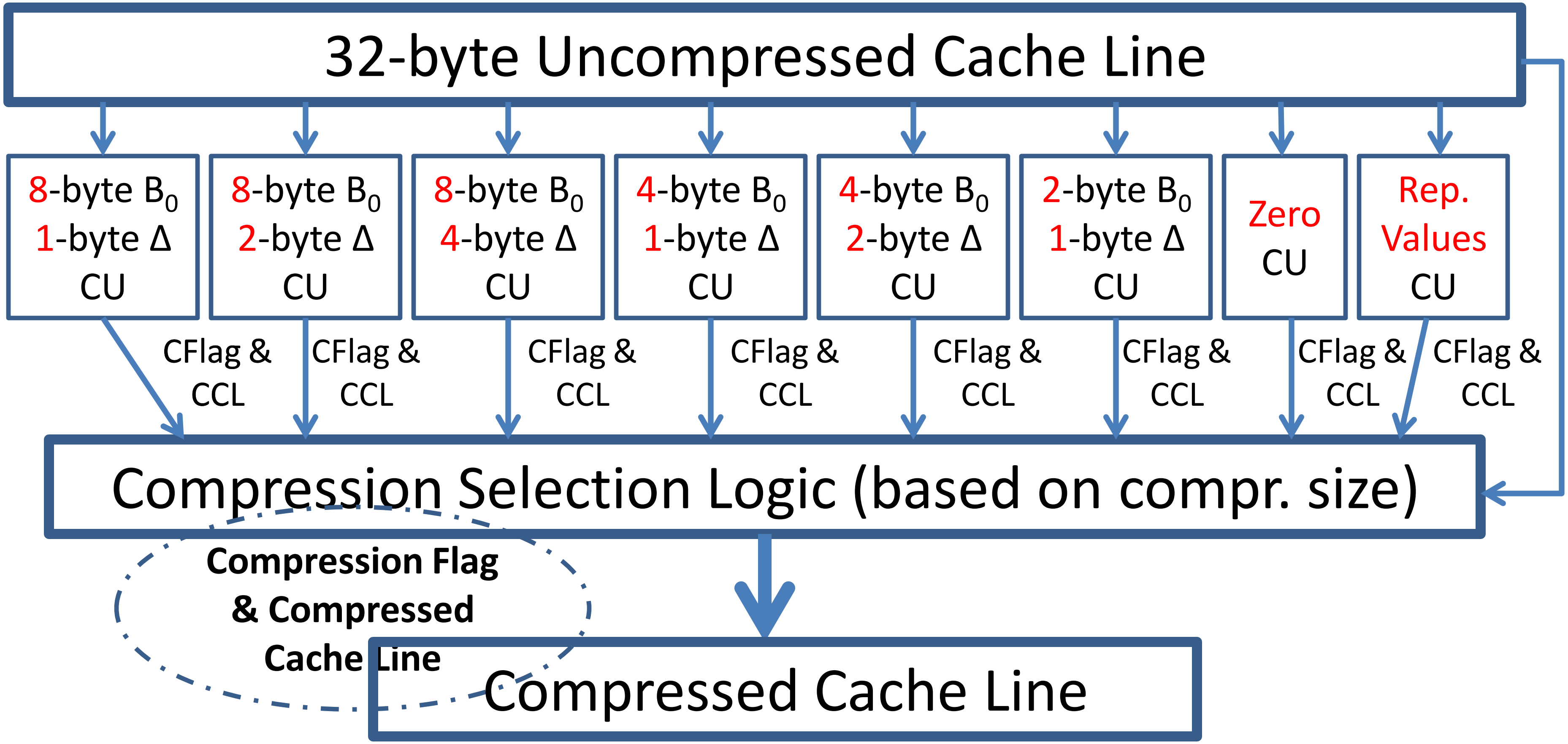
- **Decompressor Design**
  - Low latency
- **Compressor Design**
  - Low cost and complexity
- **BΔI Cache Organization**
  - Modest complexity

BΔI Decompressor Design



Vector addition

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

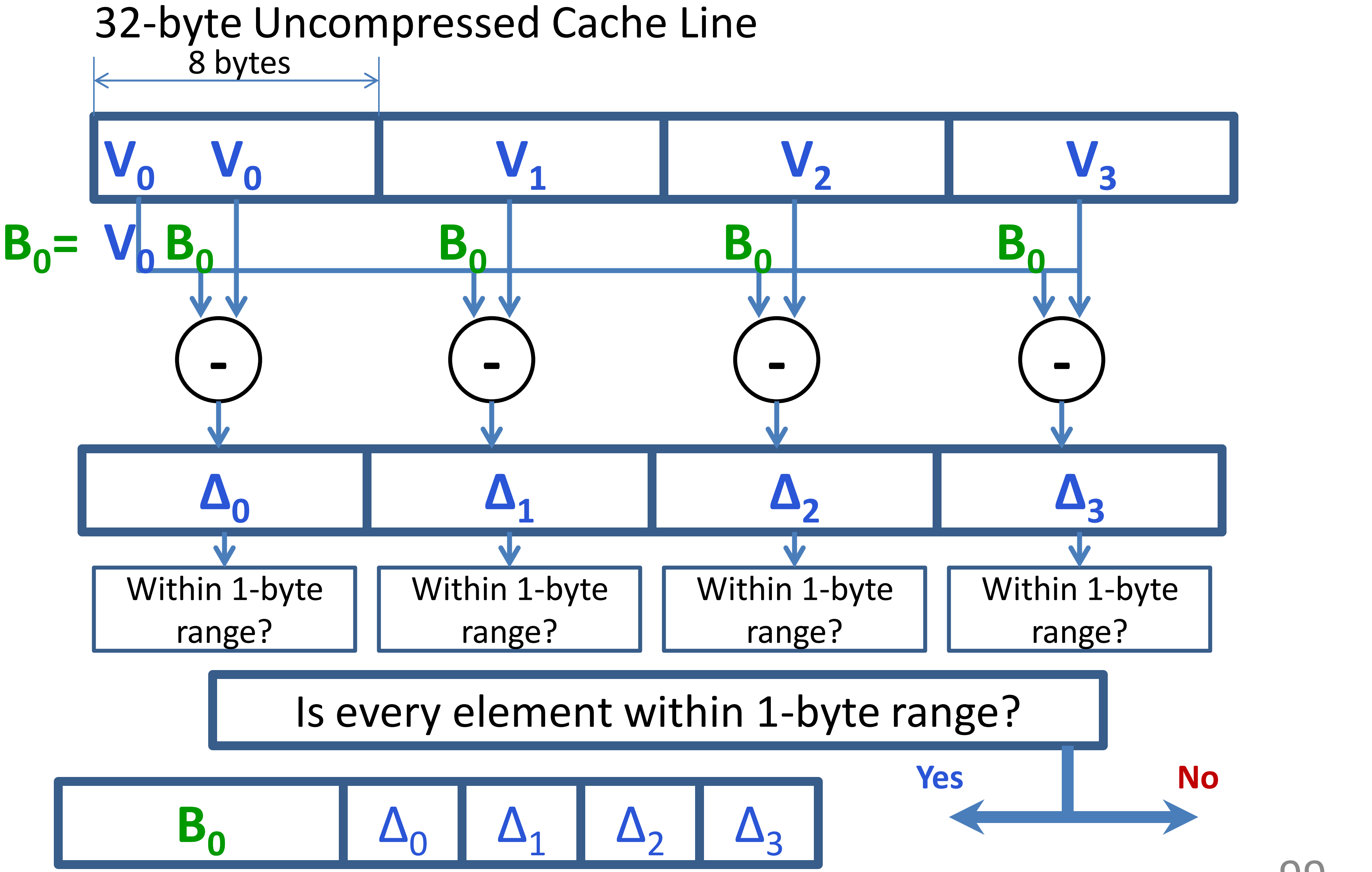
CFlag & CCL

Compression Selection Logic (based on compr. size)

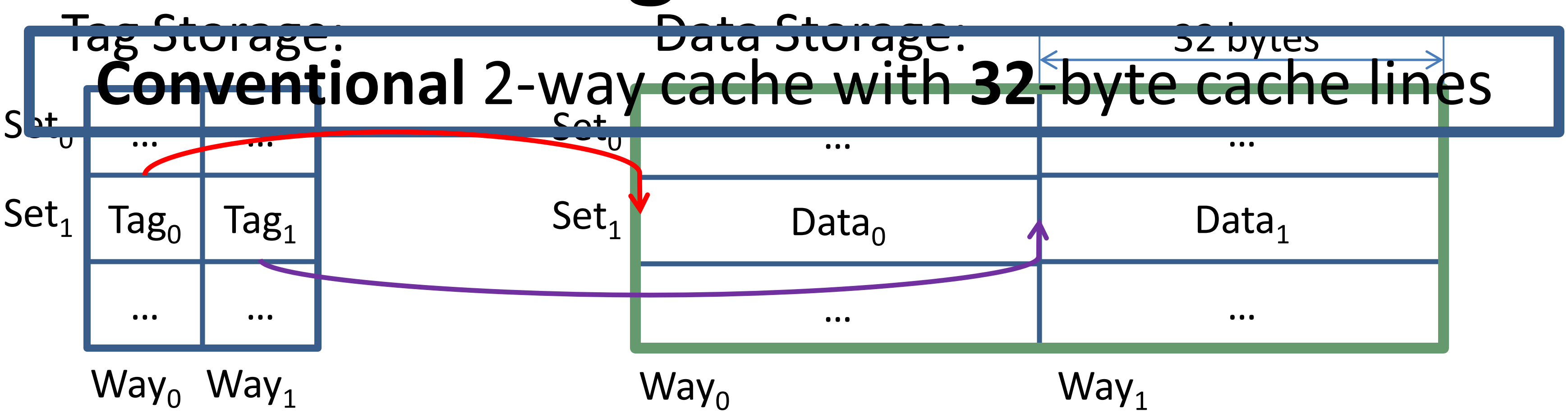
Compression Flag & Compressed Cache Line

Compressed Cache Line

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

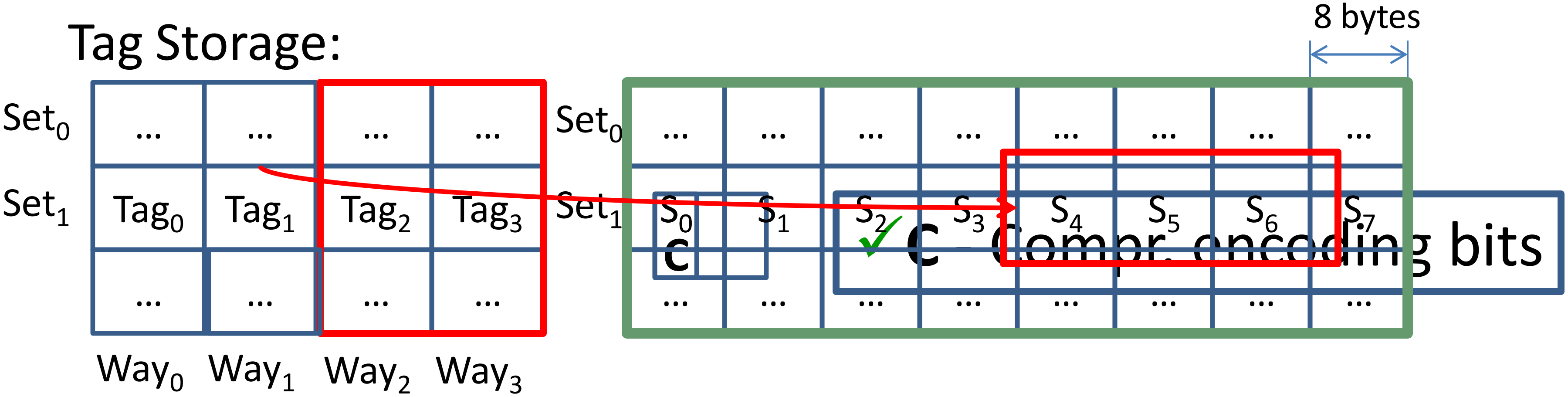


BΔI Cache Organization



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)

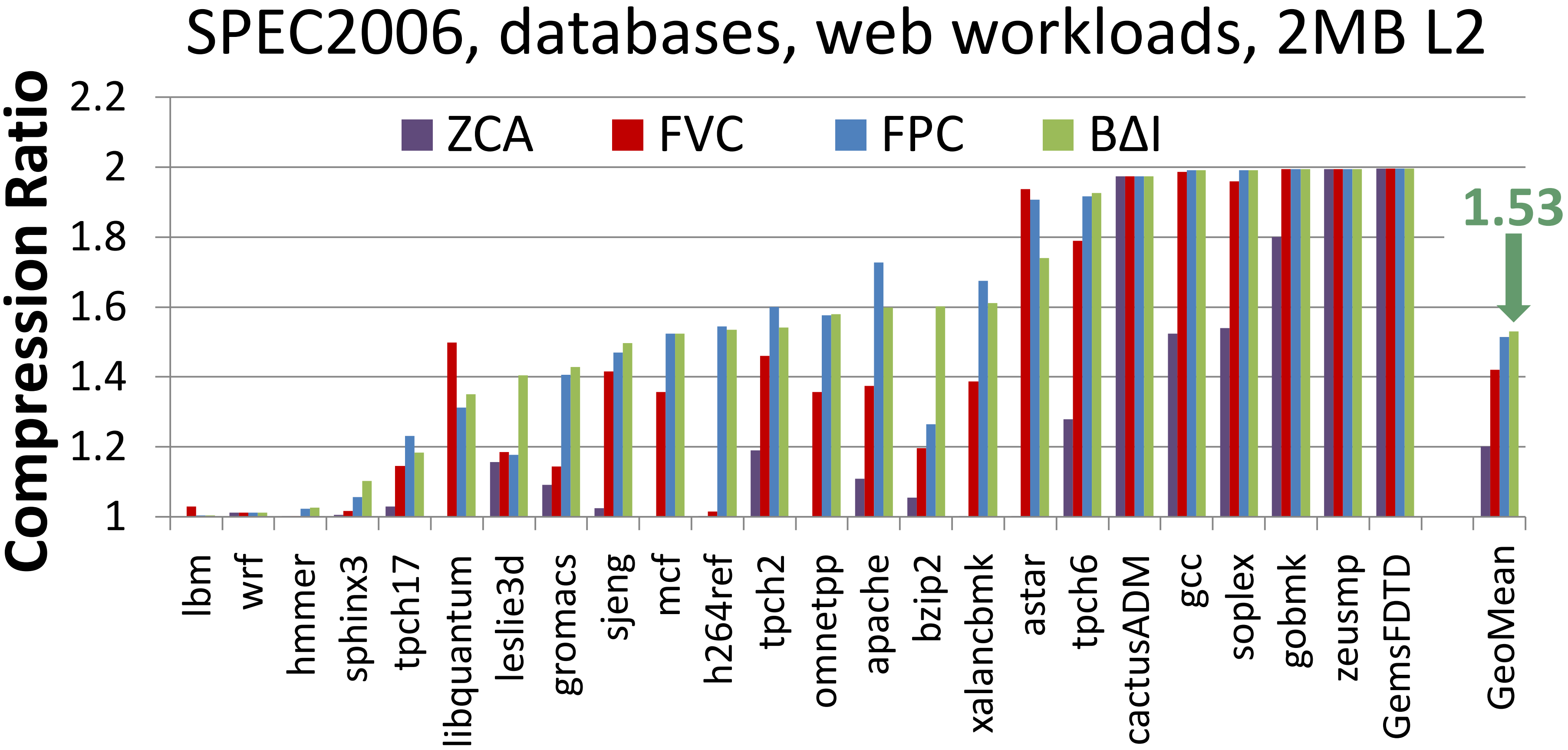
Outline

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

Methodology

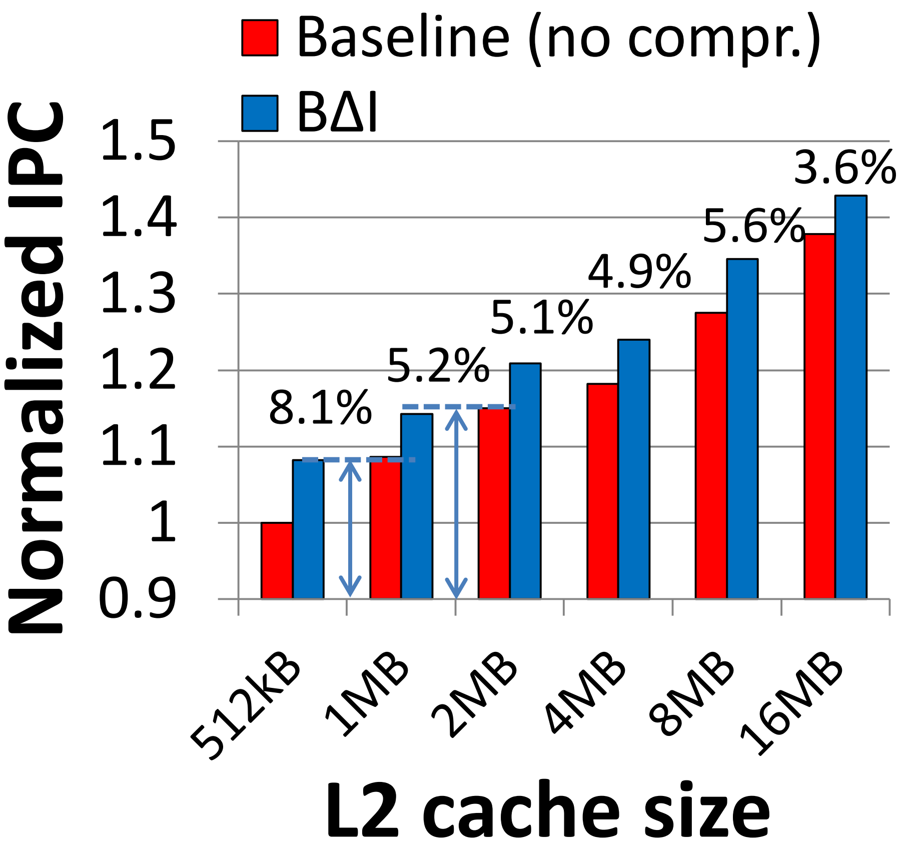
- **Simulator**
  - x86 event-driven simulator based on Simics *[Magnusson+, Computer'02]*
- **Workloads**
  - SPEC2006 benchmarks, TPC, Apache web server
  - 1 – 4 core simulations for 1 billion representative instructions
- **System Parameters**
  - L1/L2/L3 cache latencies from CACTI *[Thoziyoor+,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

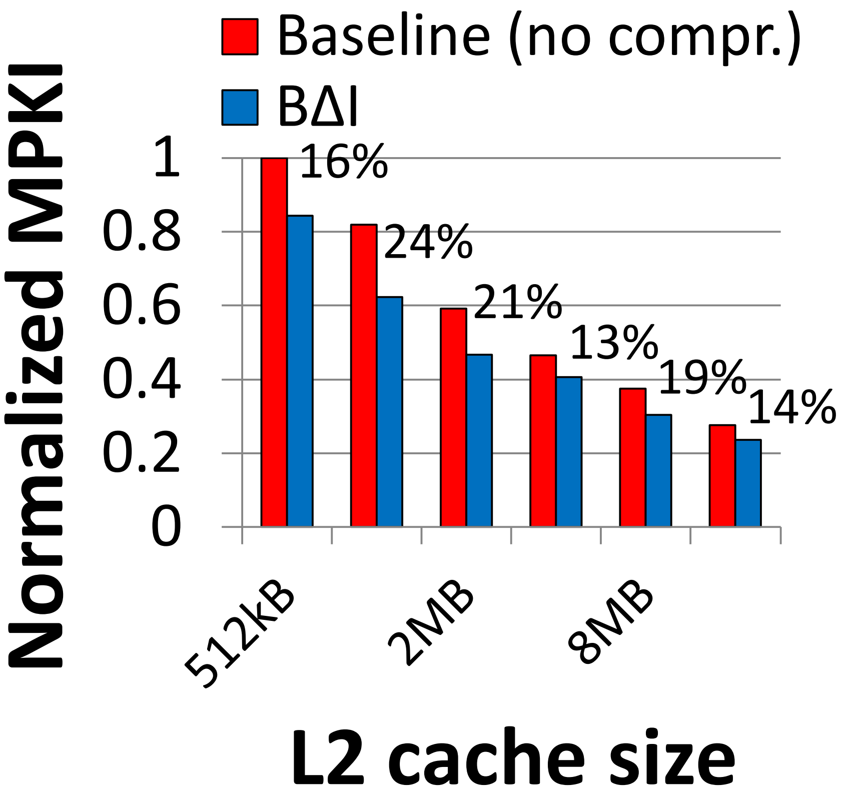
Compression Ratio: BΔI vs. Prior Work



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)

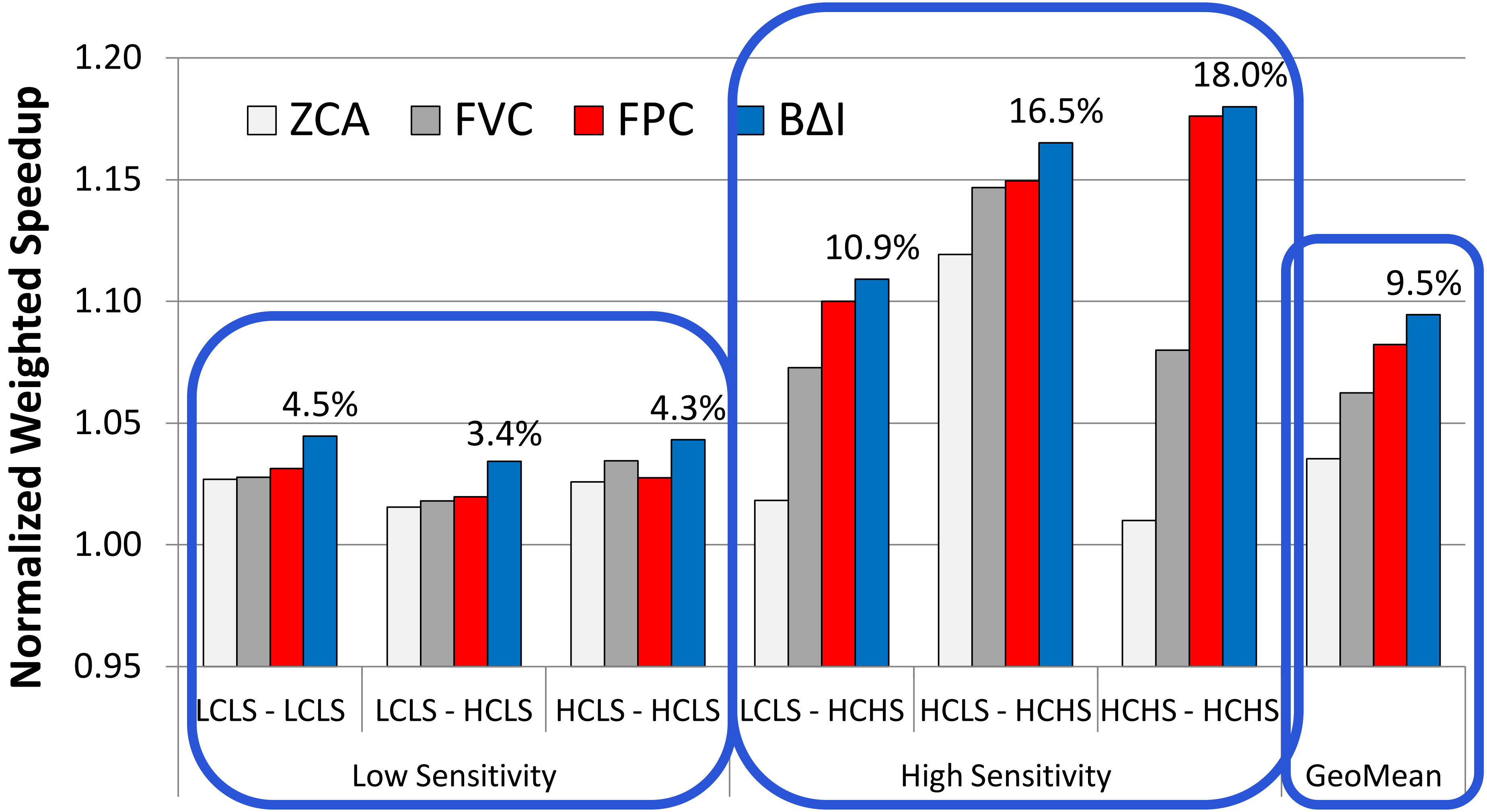
  Sensitivity: performance gain with more cache

  (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

Linearly Compressed Pages

Gennady Pekhimenko, Vivek Seshadri, Yoongu Kim, Hongyi Xin, Onur Mutlu, Michael A. Kozuch, Phillip B. Gibbons, and Todd C. Mowry, **"Linearly Compressed Pages: A Main Memory Compression Framework with Low Complexity and Low Latency"** *SAFARI Technical Report*, TR-SAFARI-2012-005, Carnegie Mellon University, September 2012.

Executive Summary

- Main memory is a limited shared resource
- **Observation**: Significant data redundancy
- **Idea**: Compress data in main memory
- **Problem**: How to avoid latency increase?
- **Solution**: Linearly Compressed Pages (LCP): fixed-size cache line granularity compression

1. Increases capacity (**69%** on average)
2. Decreases bandwidth consumption (**46%**)
3. Improves overall performance (**9.5%**)

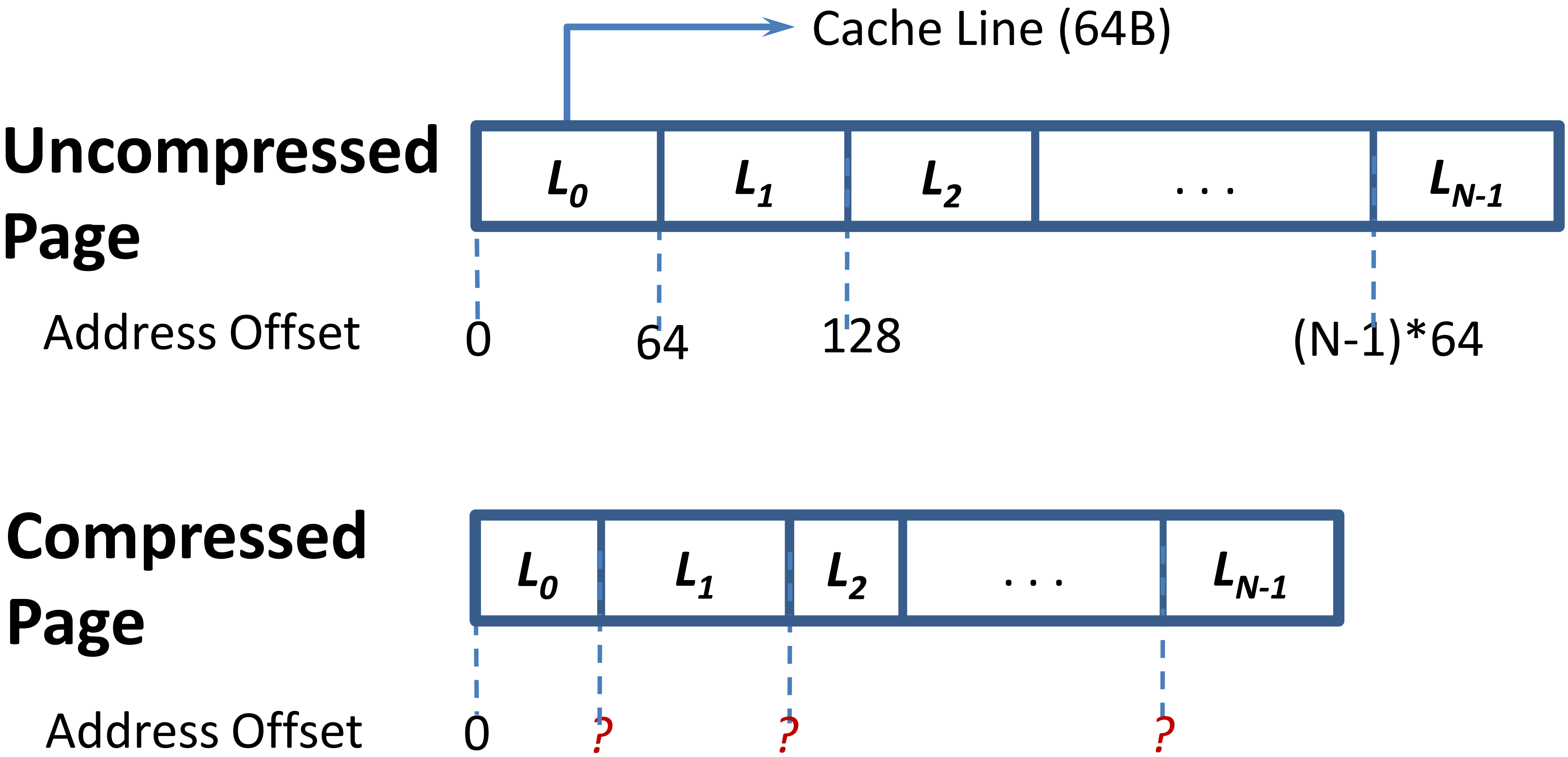
Challenges in Main Memory Compression

1. Address Computation

2. Mapping and Fragmentation

3. Physically Tagged Caches

Address Computation



Cache Line (64B)

Uncompressed Page

L_0 L_1 L_2 . . . L_{N-1}

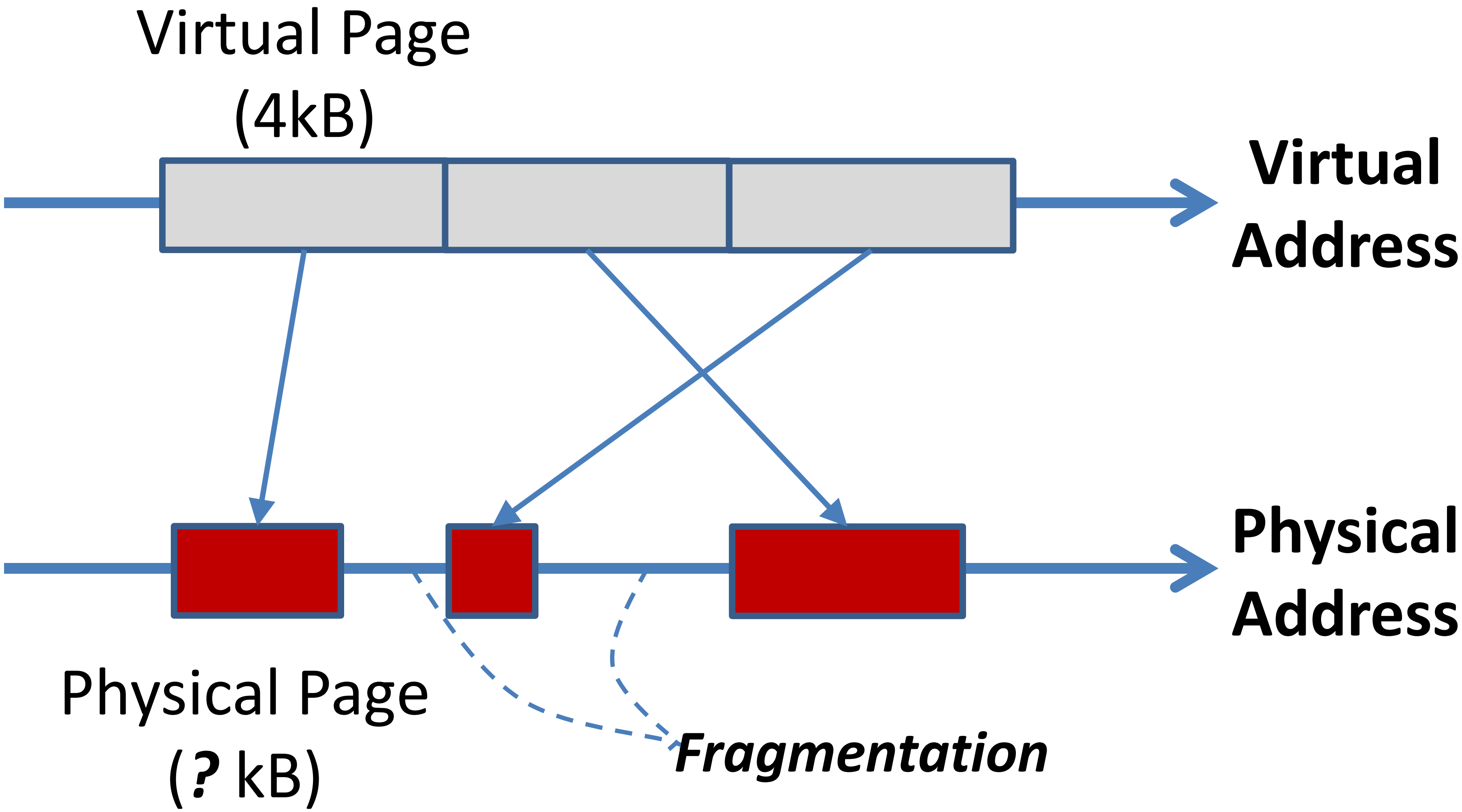
Address Offset 0 64 128 (N-1)*64

Compressed Page

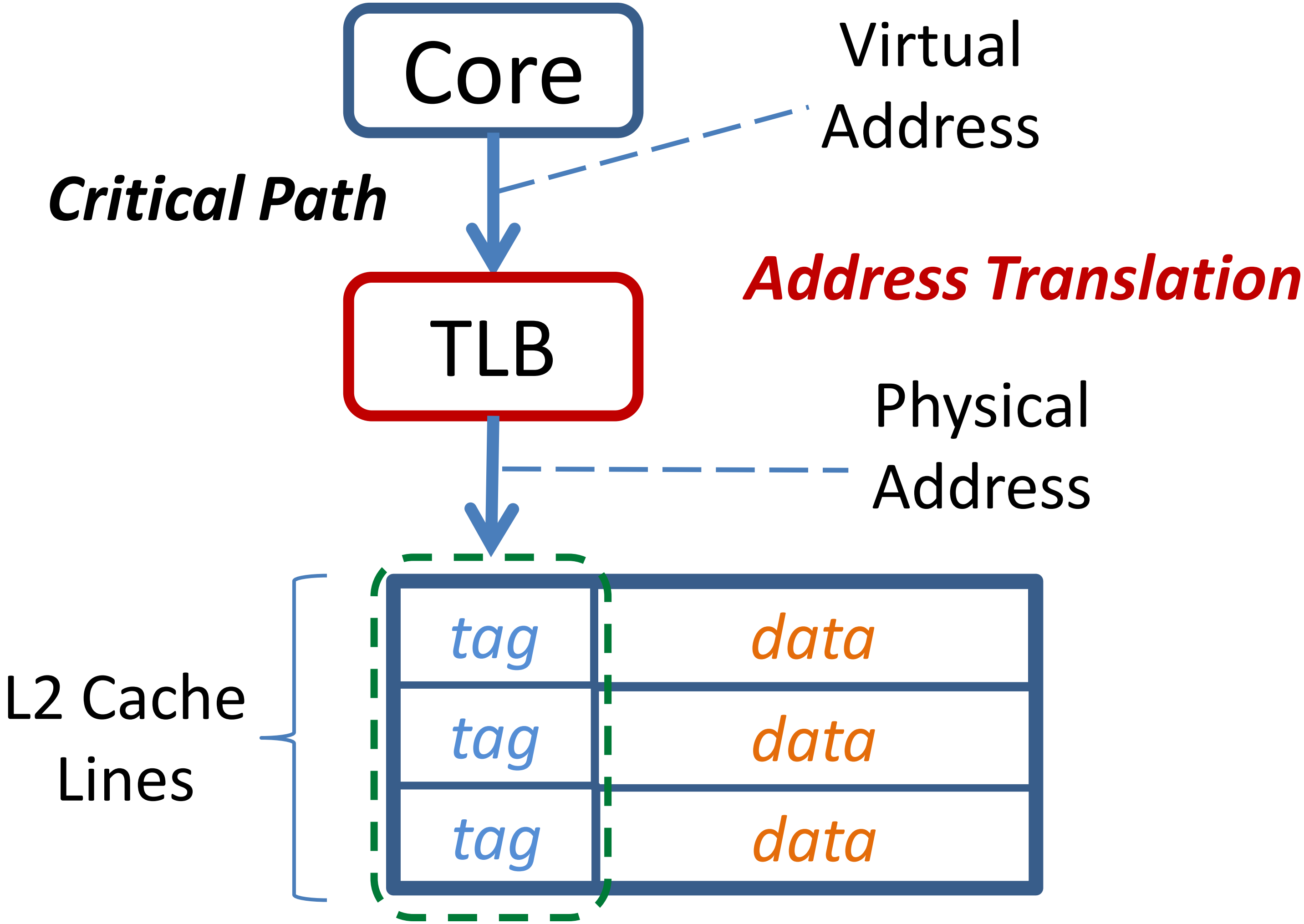
L_0 L_1 L_2 . . . L_{N-1}

Address Offset 0 ? ? ?

Mapping and Fragmentation



Physically Tagged Caches



Shortcomings of Prior Work

Compression Mechanisms	Access Latency	Decompression Latency	Complexity	Compression Ratio
IBM MXT *[IBM J.R.D. ’01]*	✗	✗	✗	✓

Shortcomings of Prior Work

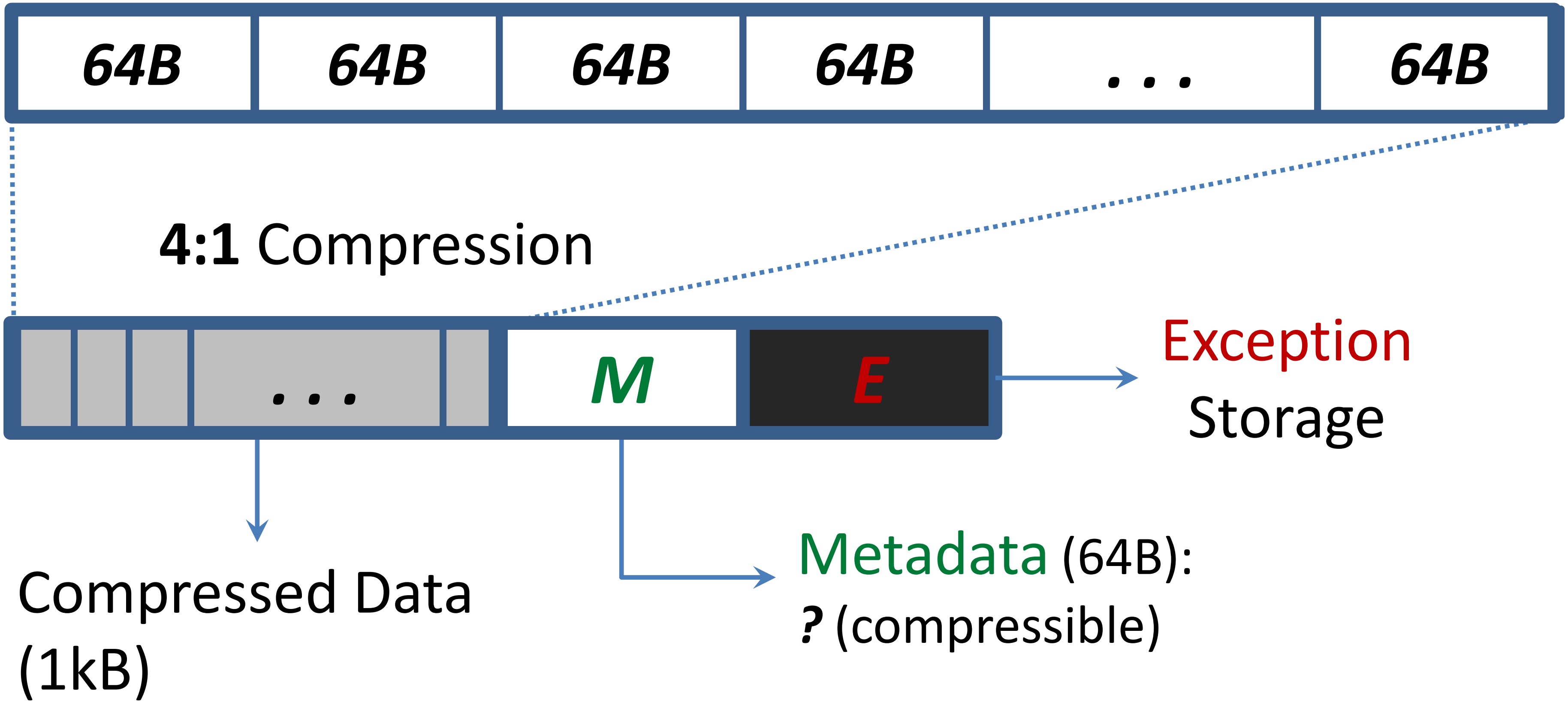
Compression Mechanisms	Access Latency	Decompression Latency	Complexity	Compression Ratio
IBM MXT *[IBM J.R.D. ’01]*	✗	✗	✗	✓
Robust Main Memory Compression *[ISCA’05]*	✗	✓	✗	✓

Shortcomings of Prior Work

Compression Mechanisms	Access Latency	Decompression Latency	Complexity	Compression Ratio
IBM MXT *[IBM J.R.D. '01]*	✗	✗	✗	✓
Robust Main Memory Compression *[ISCA'05]*	✗	✓	✗	✓
LCP: Our Proposal	✓	✓	✓	✓

Linearly Compressed Pages (LCP): Key Idea

Uncompressed Page (4kB: 64***64B**)



4:1 Compression

Exception Storage

Compressed Data (1kB)

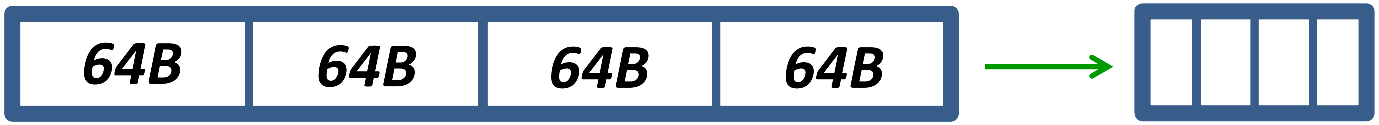
Metadata (64B):
? (compressible)

LCP Overview

- Page Table entry extension
  - compression type and size
  - extended physical base address
- Operating System management support
  - **4** memory pools (512B, 1kB, 2kB, 4kB)
- Changes to cache tagging logic
  - physical page base address + **cache line index** (within a page)
- Handling page overflows
- Compression algorithms: **BDI** *[PACT'12]* , **FPC** *[ISCA'04]*

LCP Optimizations

- **Metadata** cache
  - Avoids additional requests to metadata
- Memory bandwidth reduction:



1 transfer instead of 4

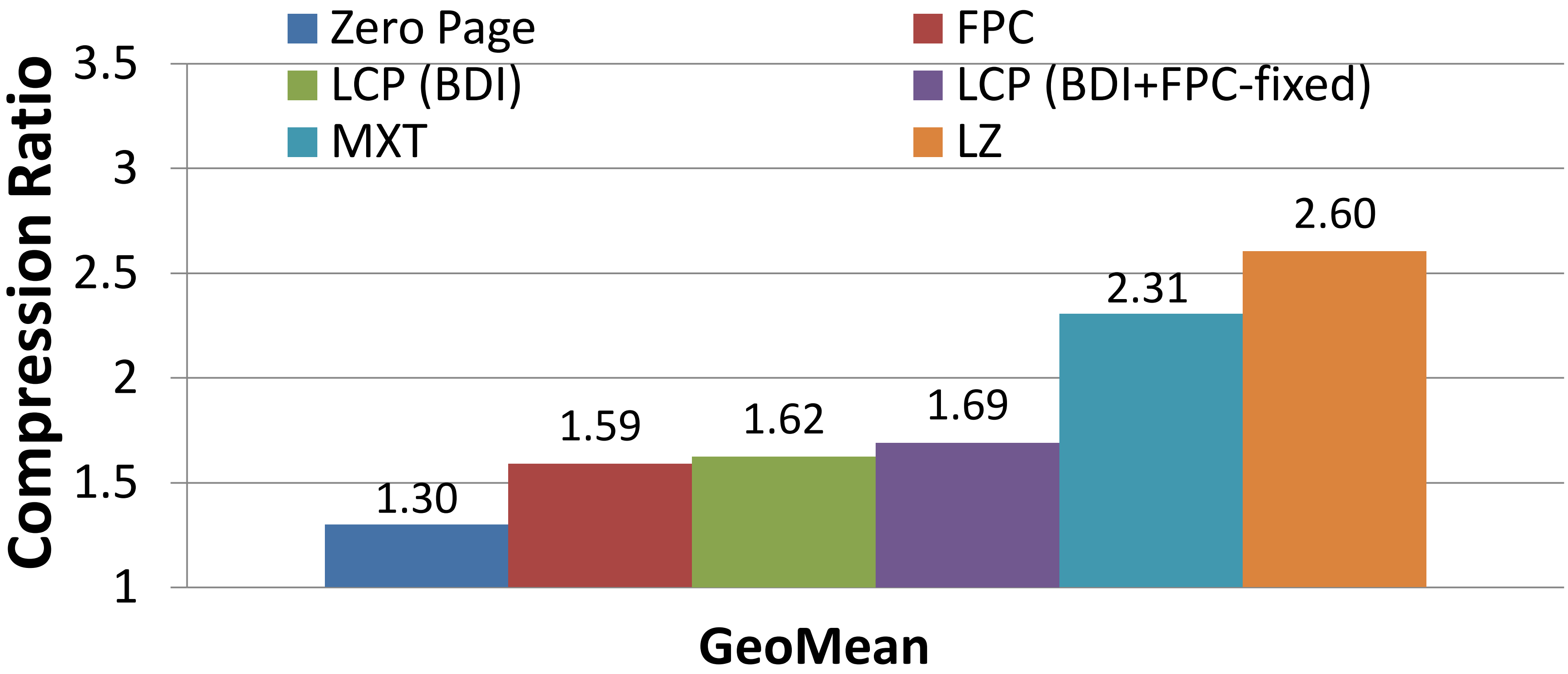
- Zero pages and zero cache lines
  - Handled separately in TLB (1-bit) and in metadata (1-bit per cache line)
- Integration with cache compression
  - BDI and FPC

Methodology

- **Simulator**
  - x86 event-driven simulators
    - Simics-based *[Magnusson+, Computer'02]* for CPU
    - Multi2Sim *[Ubal+, PACT'12]* for GPU
- **Workloads**
  - SPEC2006 benchmarks, TPC, Apache web server, GPGPU applications
- **System Parameters**
  - L1/L2/L3 cache latencies from CACTI *[Thoziyoor+, ISCA'08]*
  - 512kB - 16MB L2, simple memory model

Compression Ratio Comparison

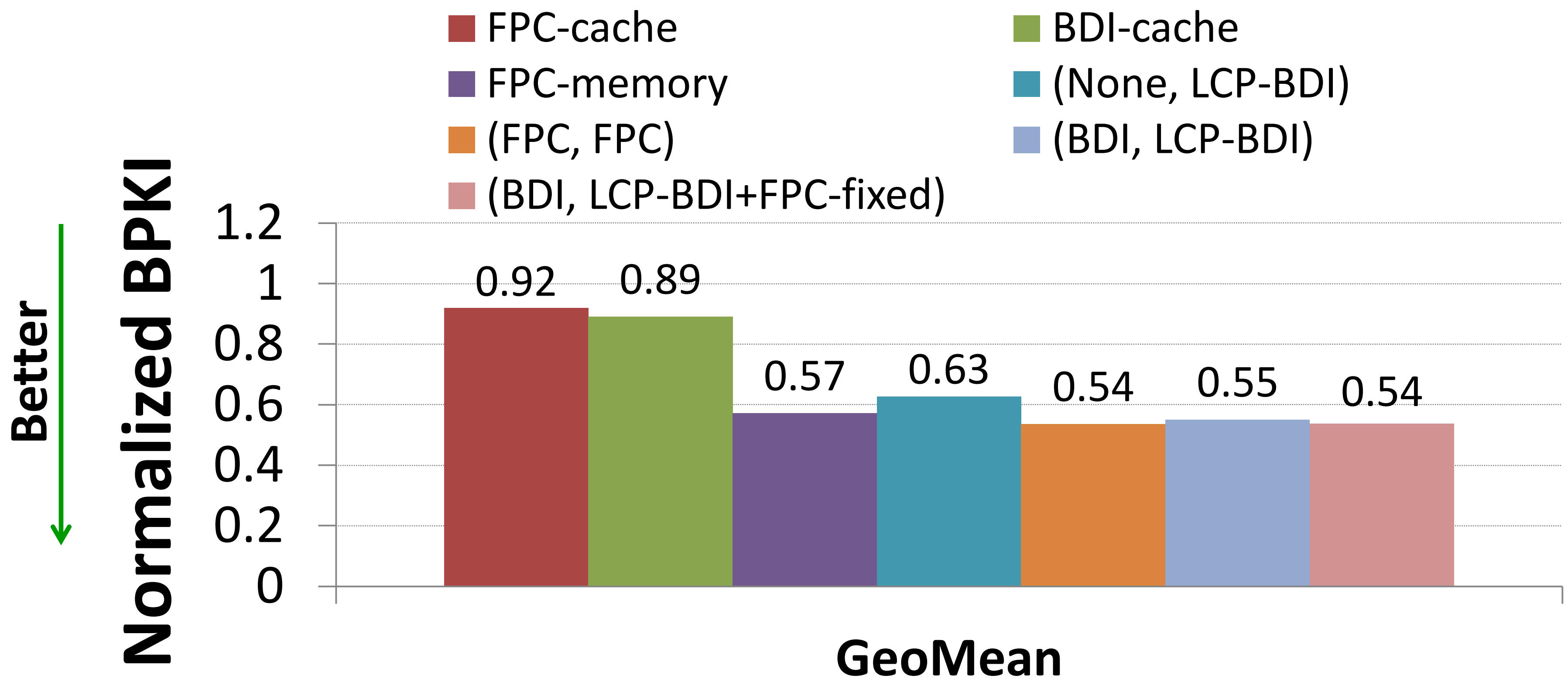
SPEC2006, databases, web workloads, 2MB L2 cache



LCP-based frameworks achieve competitive average compression ratios with prior work

Bandwidth Consumption Decrease

SPEC2006, databases, web workloads, 2MB L2 cache



LCP frameworks significantly reduce bandwidth (**46**%)

Performance Improvement

Cores	LCP-BDI	(BDI, LCP-BDI)	(BDI, LCP-BDI+FPC-fixed)
1	6.1%	**9.5%**	9.3%
2	13.9%	**23.7%**	23.6%
4	10.7%	**22.6%**	22.5%

LCP frameworks significantly improve performance

Conclusion

- A new main memory compression framework called **LCP (Linearly Compressed Pages)**
  - **Key idea: fixed size** for compressed cache lines within a page and **fixed compression algorithm** per page

- LCP evaluation:
  - Increases capacity (**69%** on average)
  - Decreases bandwidth consumption (**46%**)
  - Improves overall performance (**9.5%**)
  - Decreases energy of the off-chip bus (**37**%)

Controlled Shared Caching

Controlled Cache Sharing

- 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.

Utility Based Shared Cache Partitioning

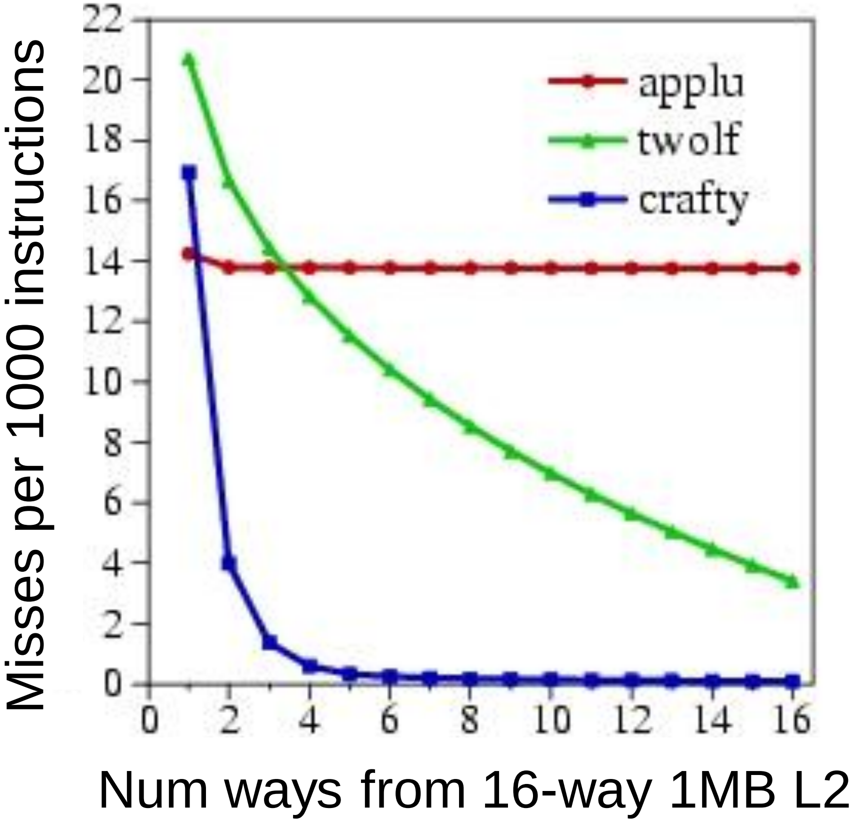
- Goal: Maximize system throughput
- Observation: Not all threads/applications benefit equally from caching → simple LRU replacement not good for system throughput
- Idea: Allocate more cache space to applications that obtain the most benefit from more space

- 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

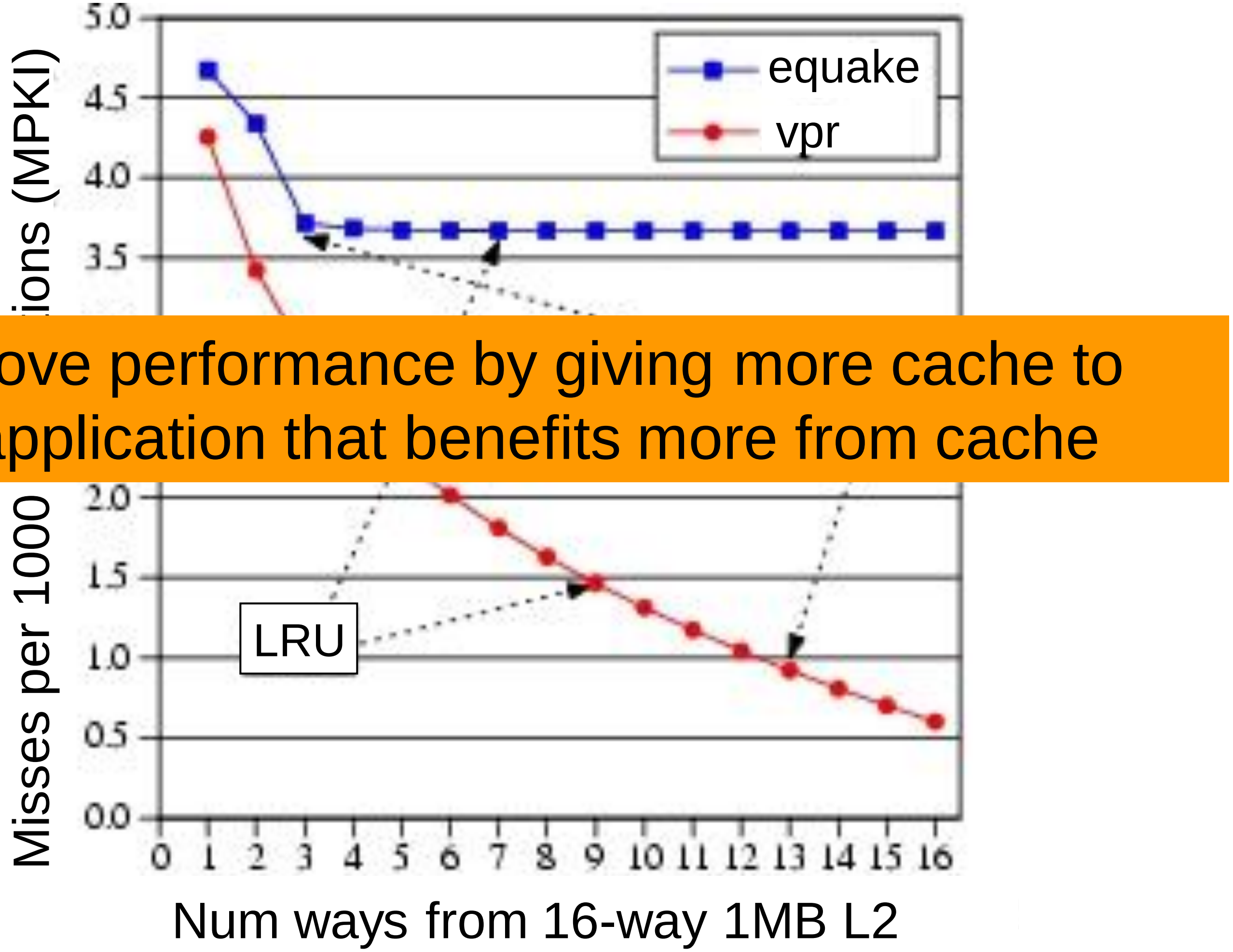


Low Utility

High Utility

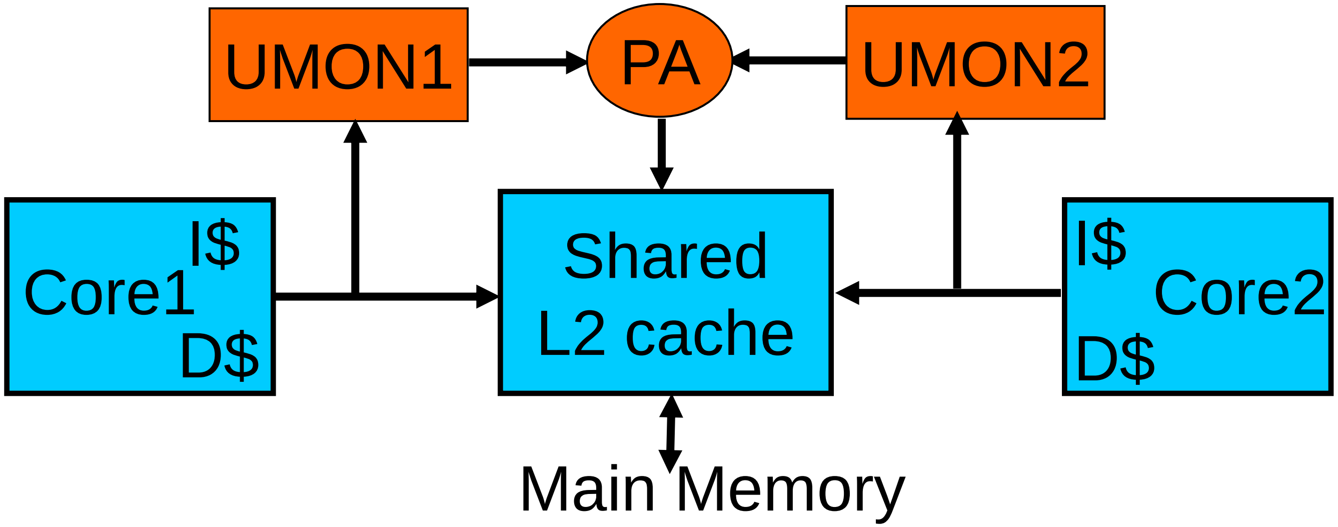
Saturating Utility

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)

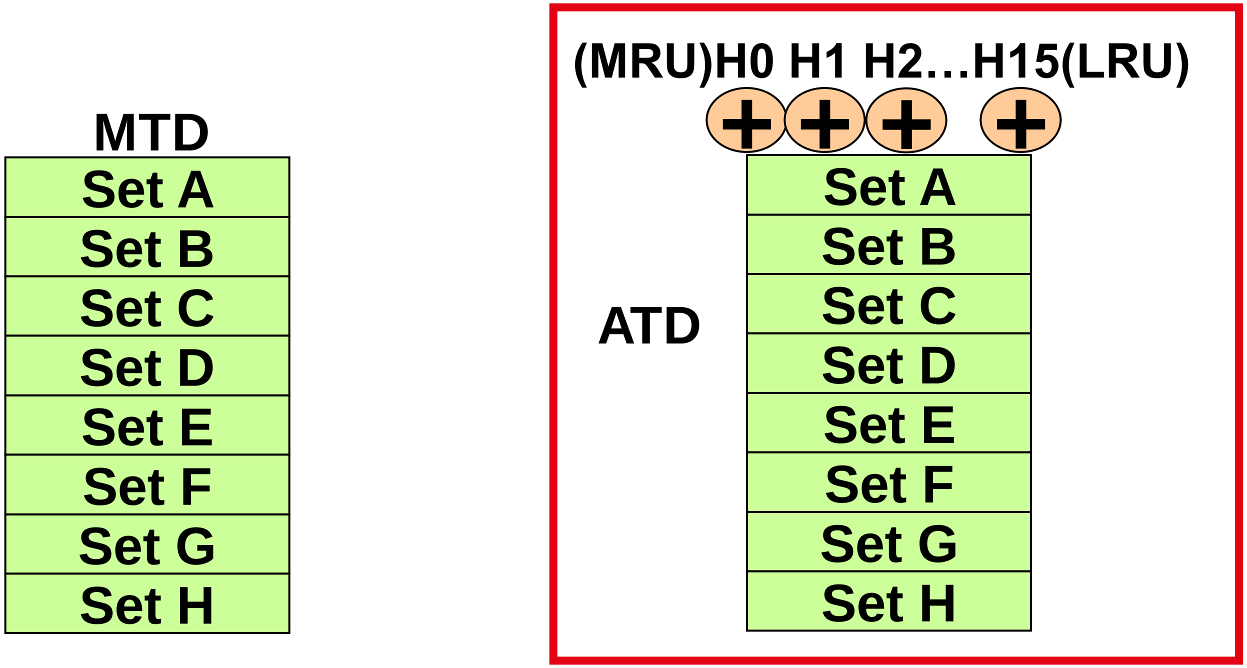


Three components:

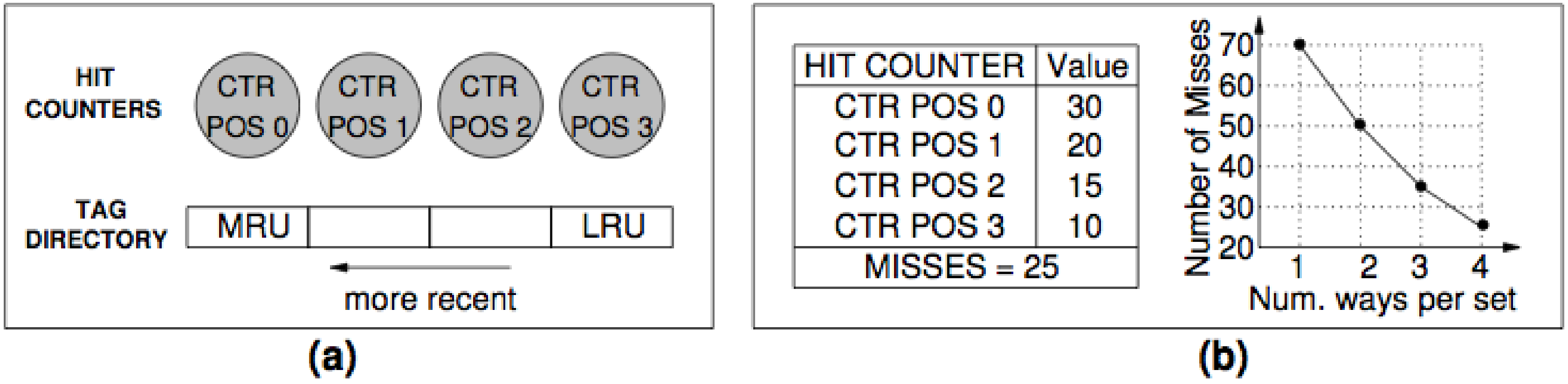
- ❑ Utility Monitors (UMON) per core
- ❑ Partitioning Algorithm (PA)
- ❑ Replacement support to enforce partitions

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



Utility Monitors

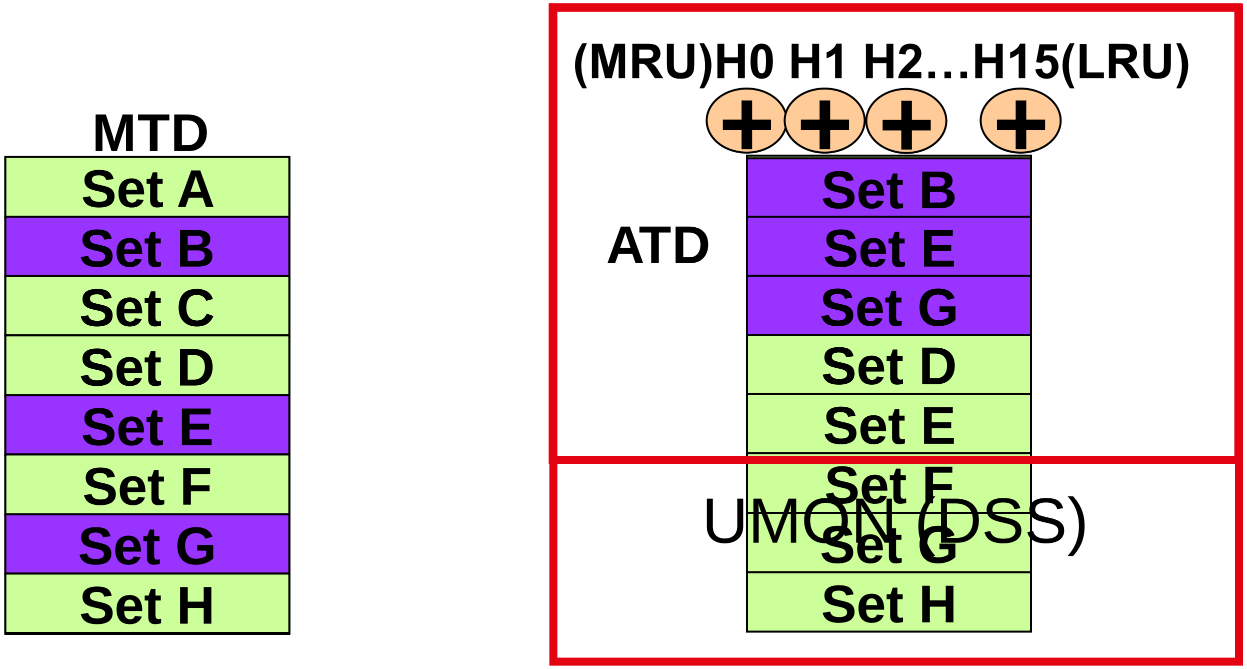


HIT COUNTER	Value
CTR POS 0	30
CTR POS 1	20
CTR POS 2	15
CTR POS 3	10
MISSES = 25	

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



Partitioning Algorithm

- Evaluate all possible partitions and select the best

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

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

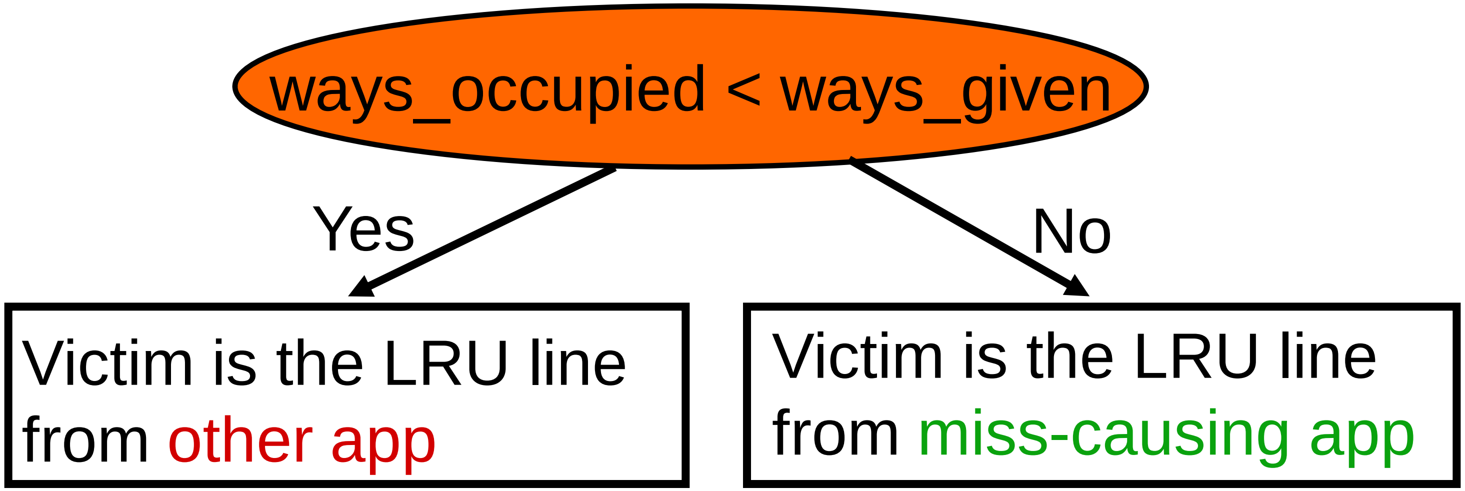
- Select a that maximizes $(Hits_{core1} + Hits_{core2})$

- 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 = $IPC_1 + IPC_2$

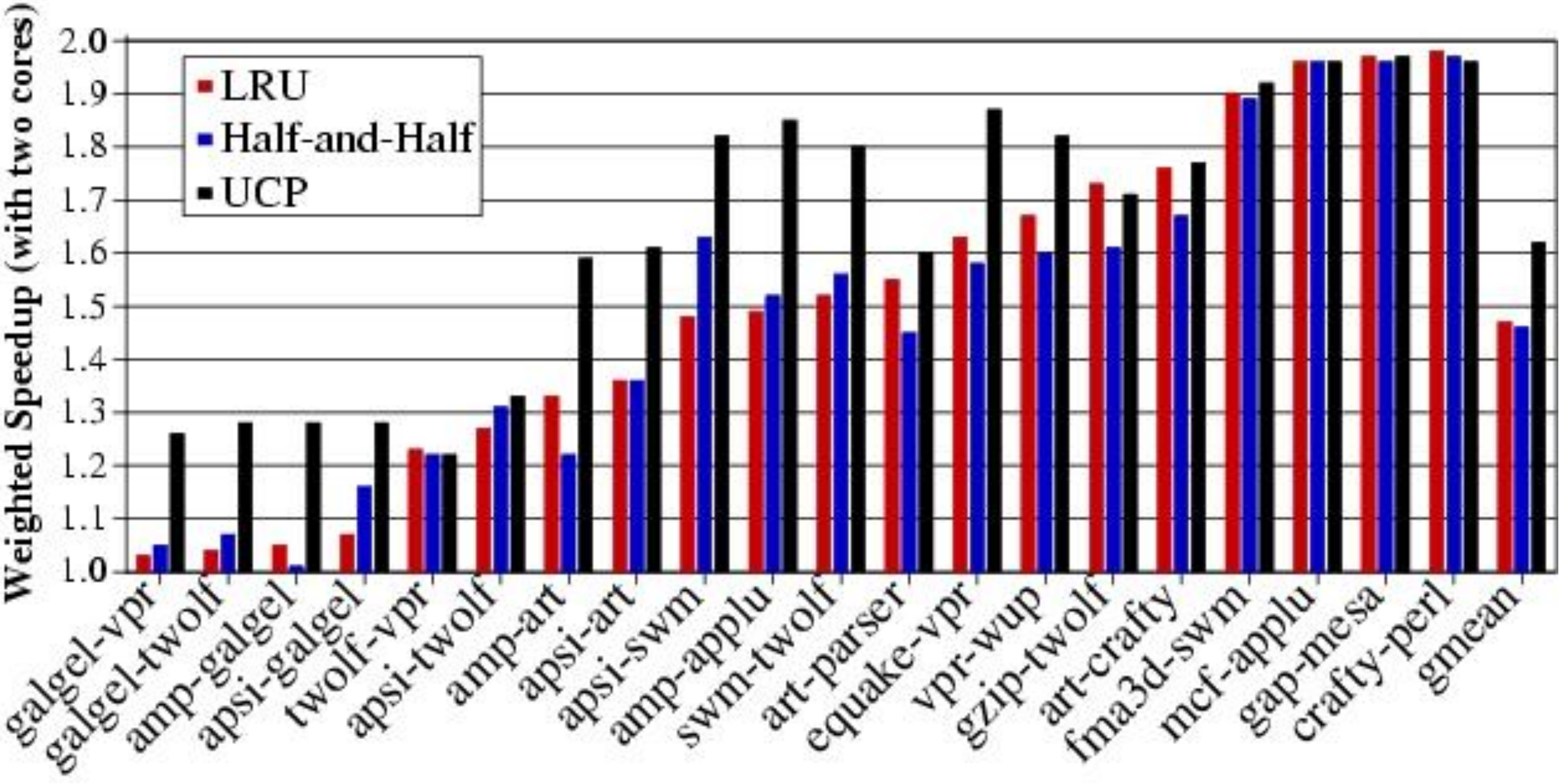
   ➔ can be unfair to low-IPC application

3. Hmean-fairness

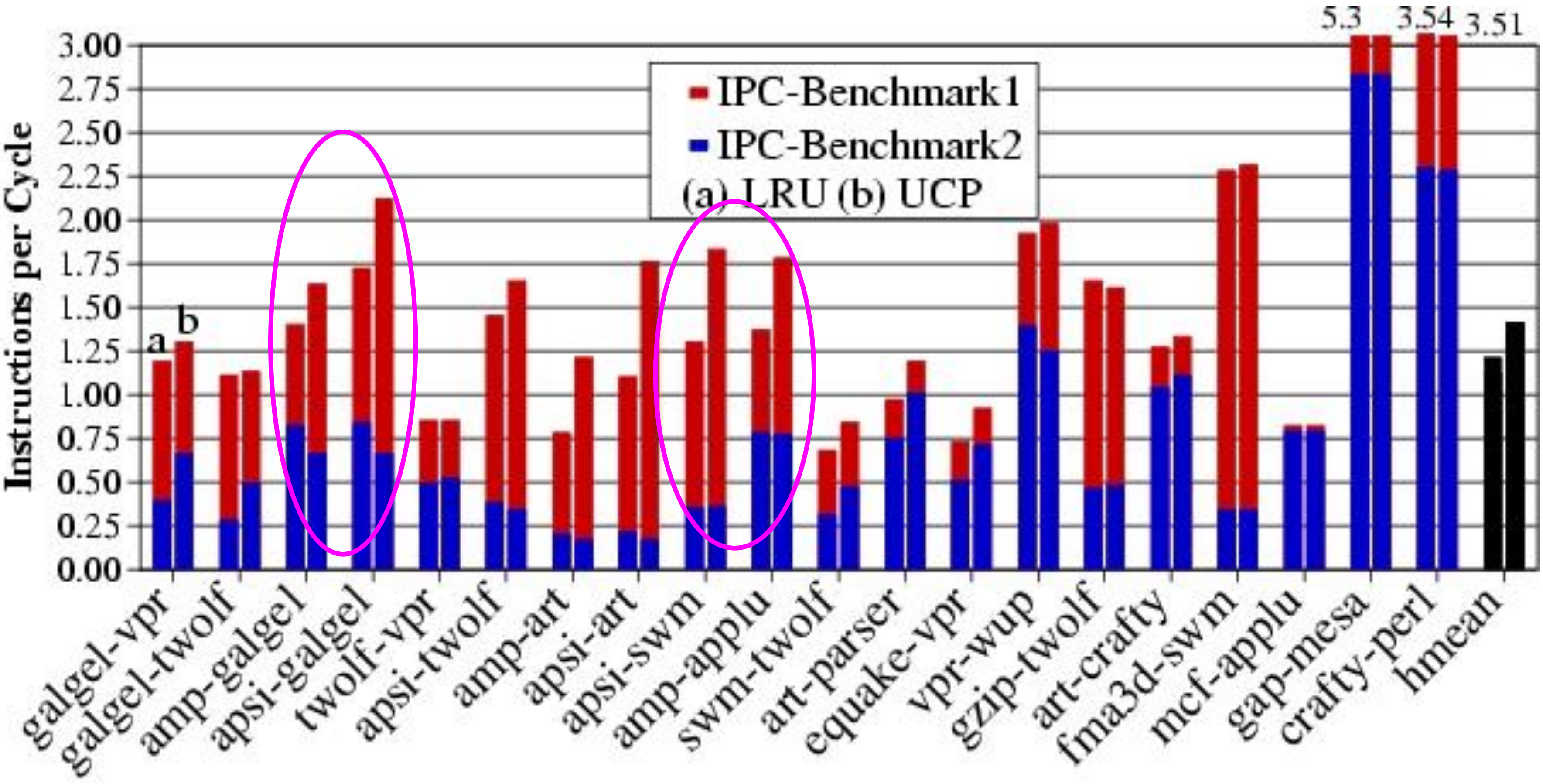
   ➔ perf = hmean($IPC_1/SingleIPC_1$, $IPC_2/SingleIPC_2$)

   ➔ 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
- 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

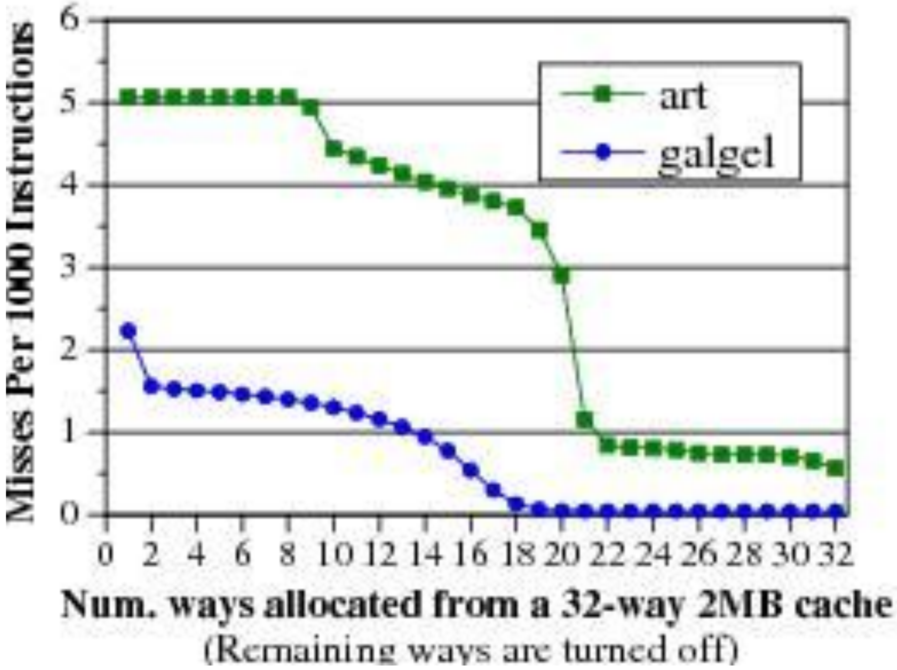
- Problem NP hard → need scalable partitioning algorithm

Greedy Algorithm [Stone+ ToC ’ 92]

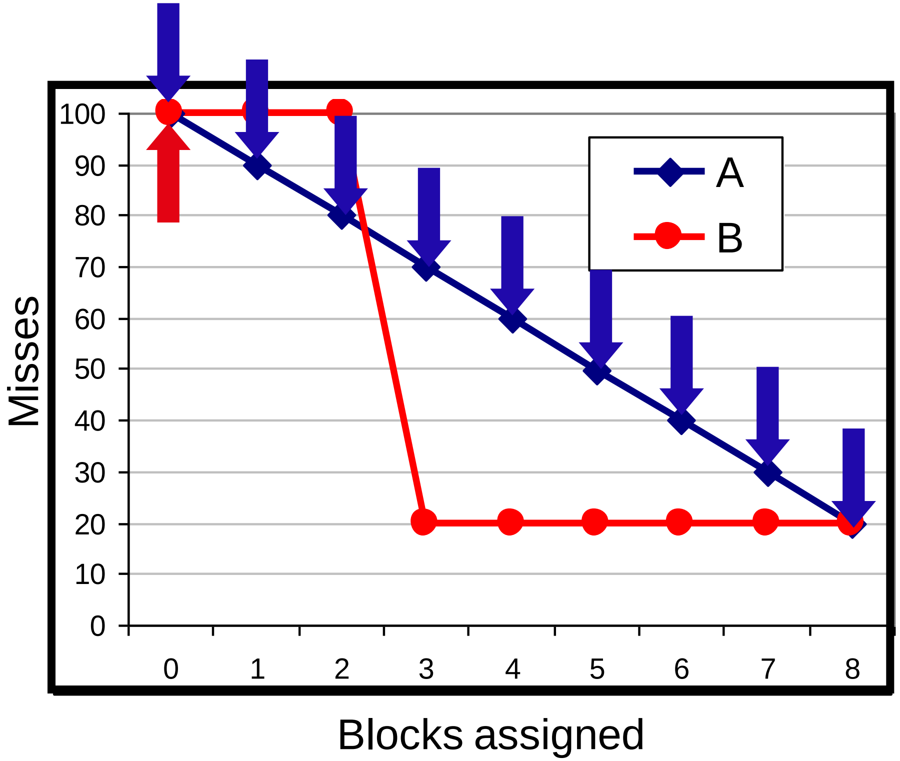
- GA allocates 1 block to the app that has the max utility for one block. Repeat till all blocks allocated

- 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

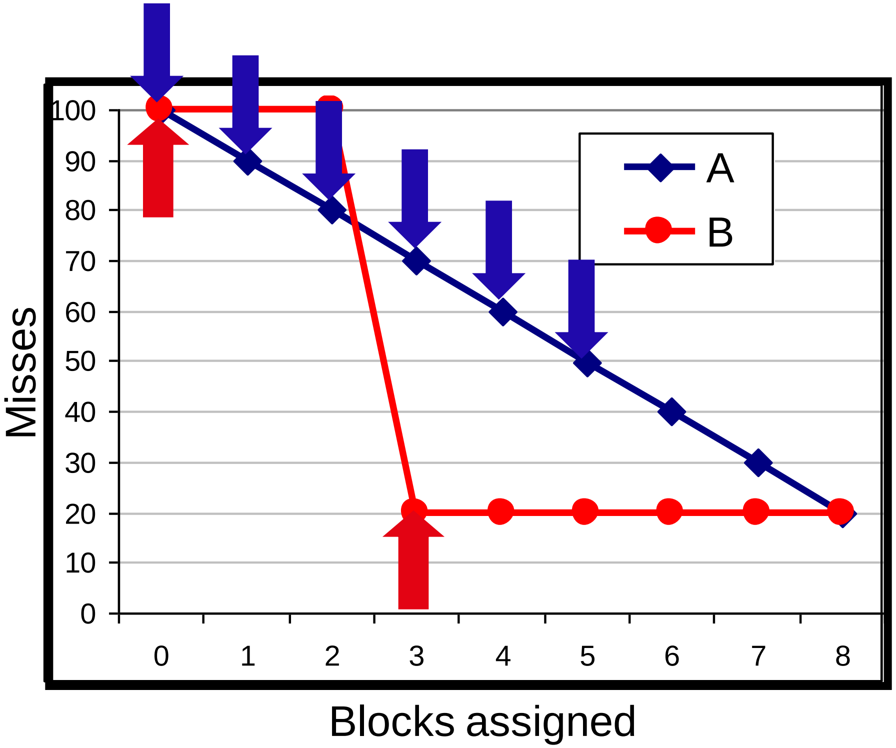
- Marginal Utility (MU) = Utility per cache resource
  - $MU_a^b = U_a^b/(b-a)$

- 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

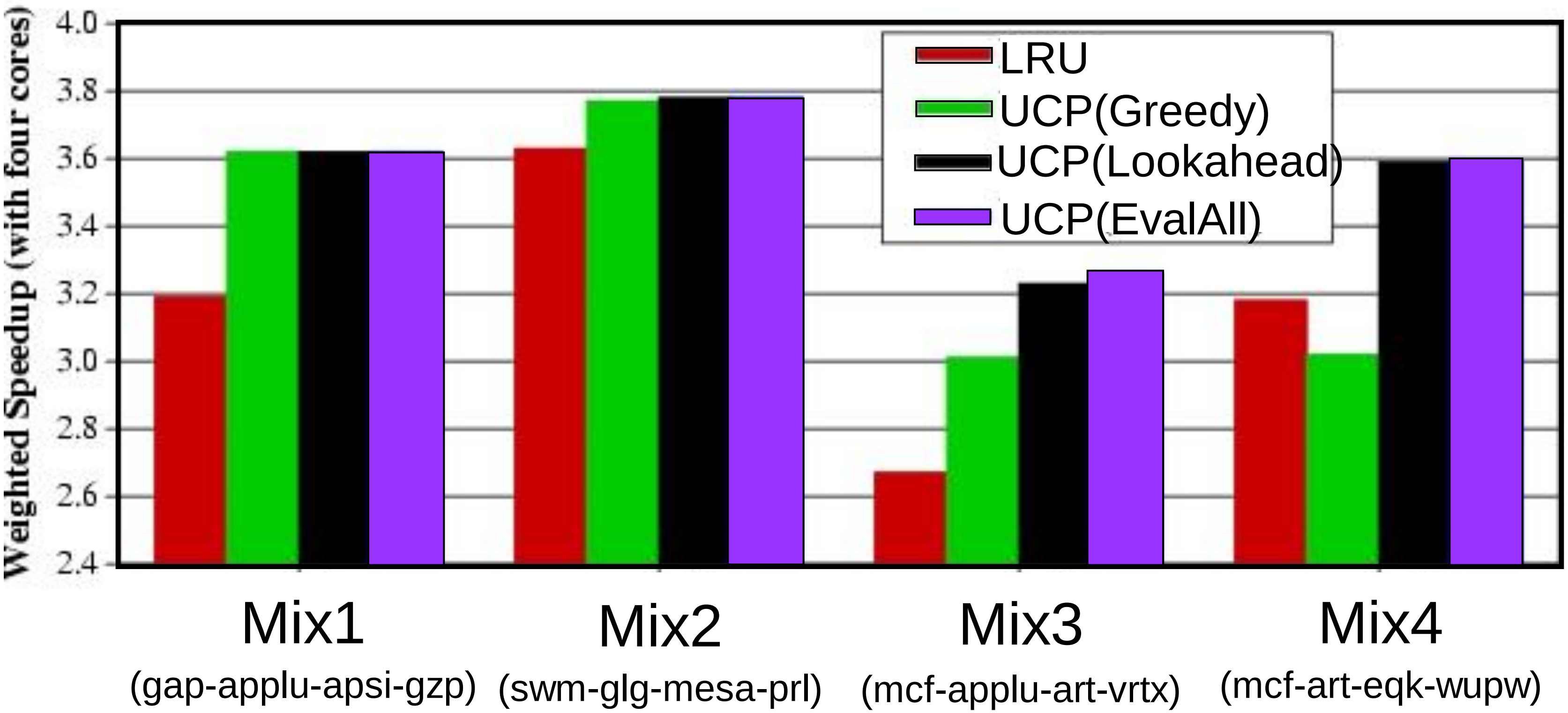
A gets 1 block

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?

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

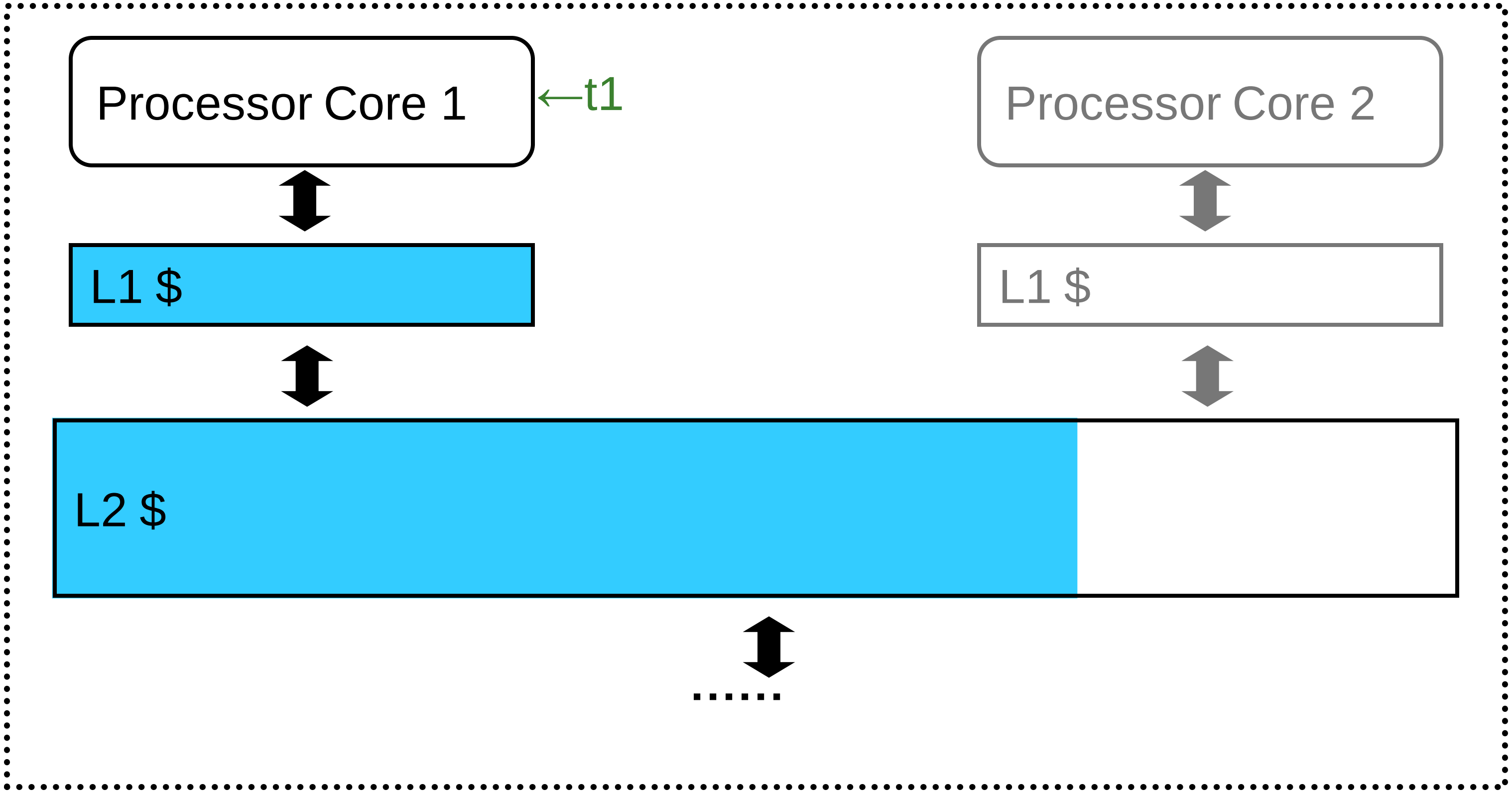
- Approximate slowdown with change in miss rate

  + Simple

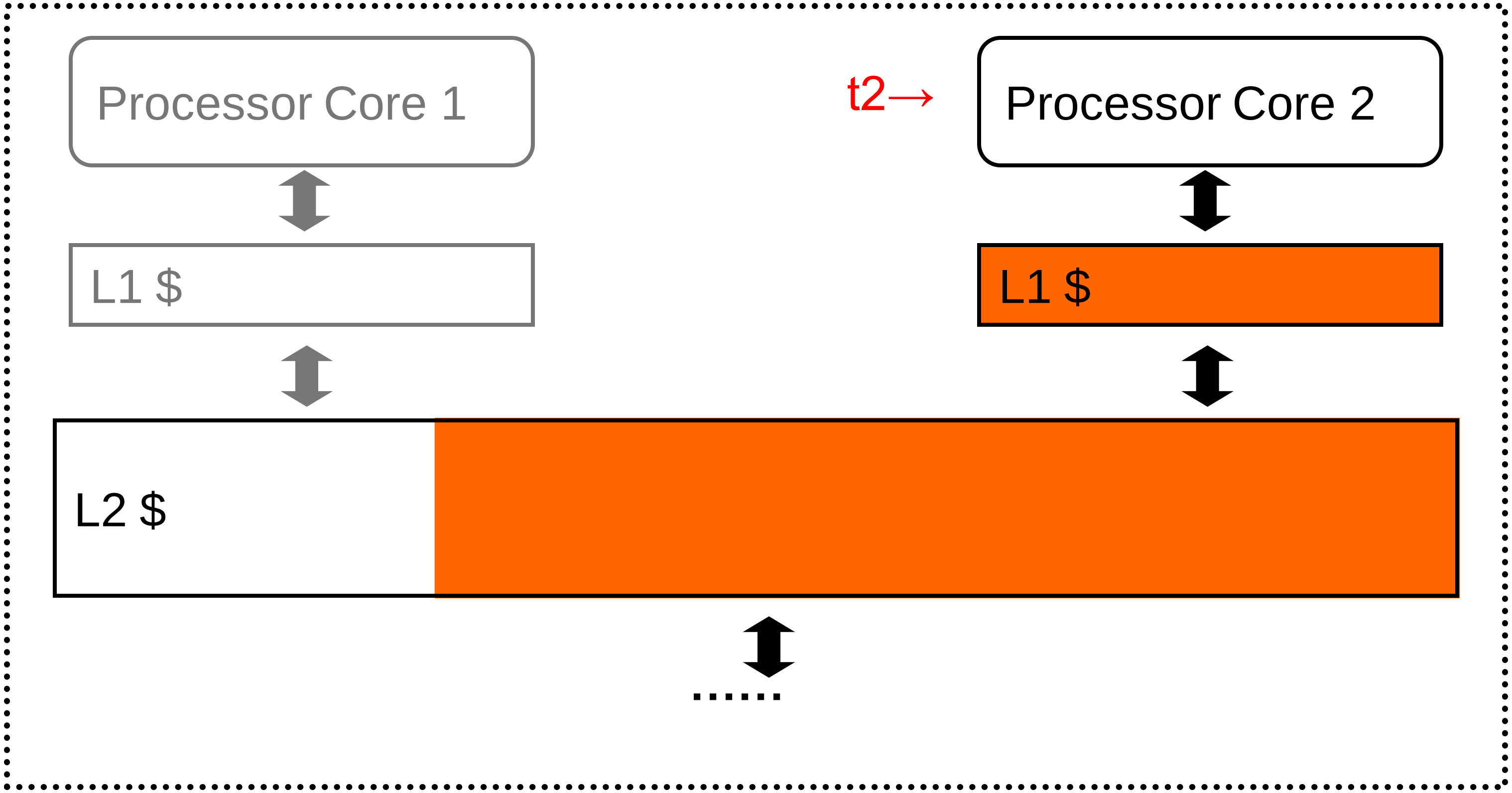
  - Not accurate. Why?

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

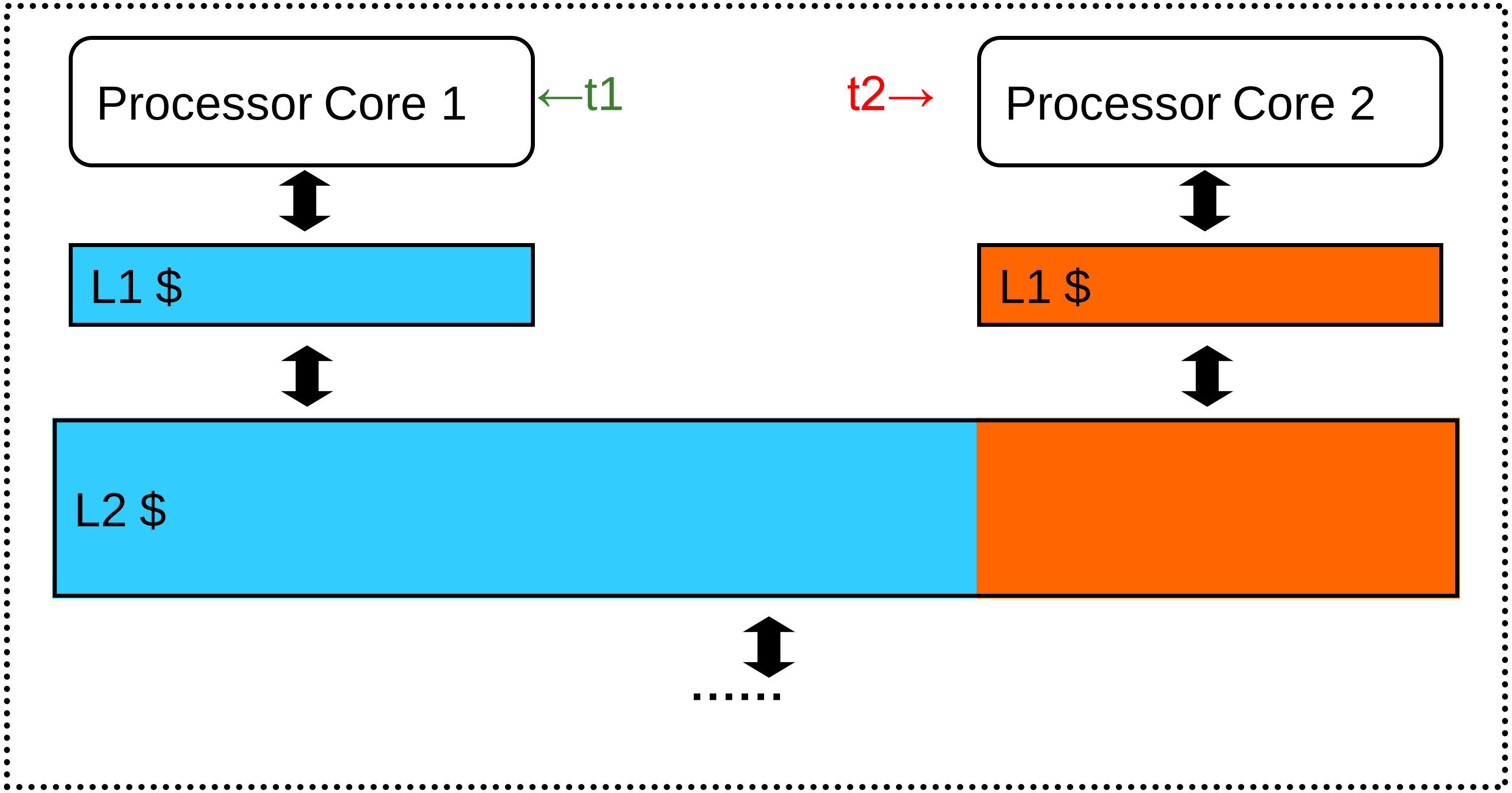
Problem with Shared Caches



Problem with Shared Caches

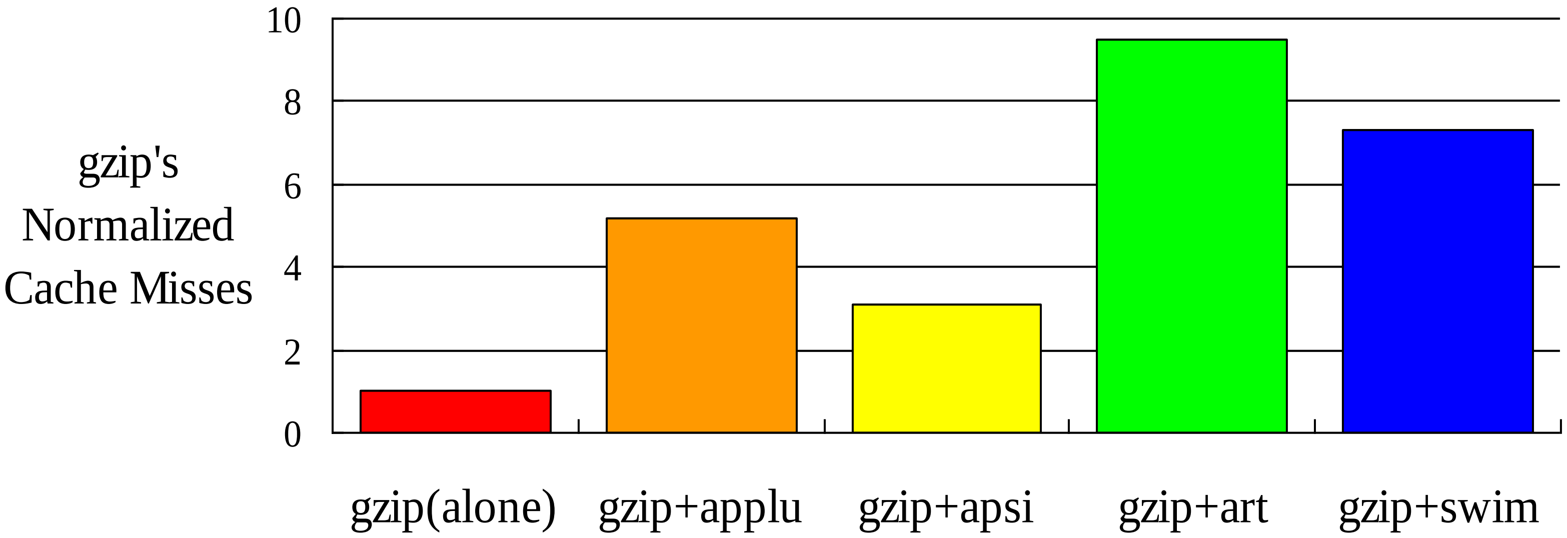


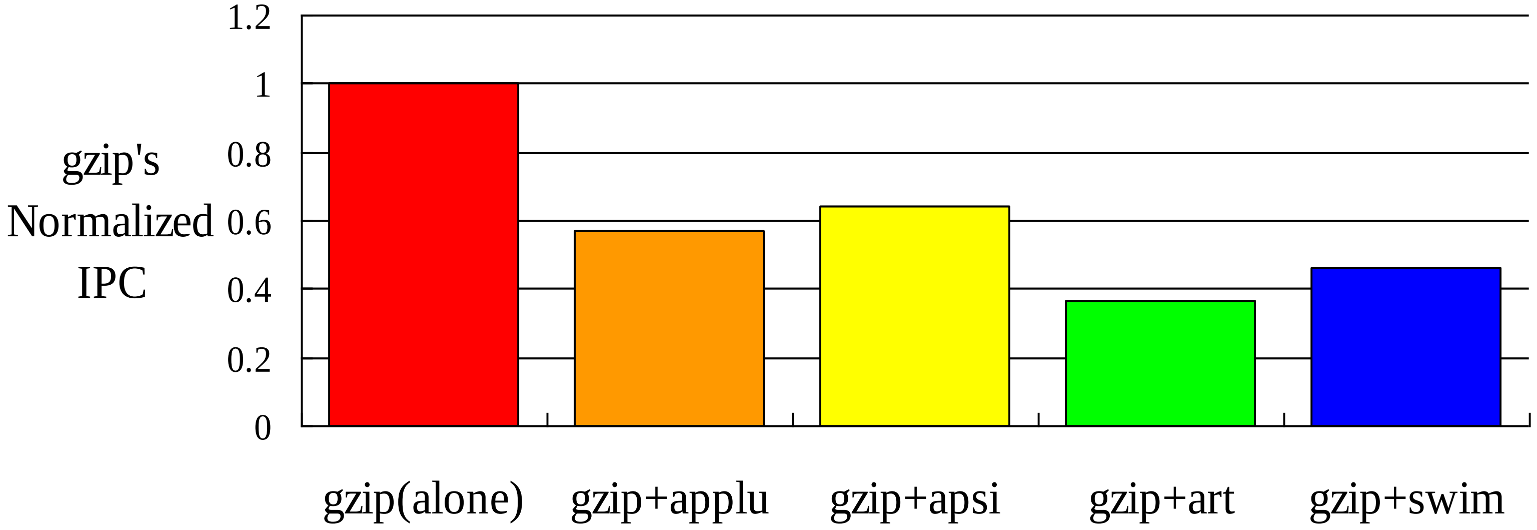
Problem with Shared Caches



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

Problem with Shared Caches





Fairness Metrics

- Uniform slowdown

$$\frac{T\_shared_i}{T\_alone_i} = \frac{T\_shared_j}{T\_alone_j}$$

- Minimize:
  - Ideally:

$$\boxed{M_0^{ij} = \left|X_i - X_j\right|, \; where \; X_i = \frac{T\_shared_i}{T\_alone_i}}$$

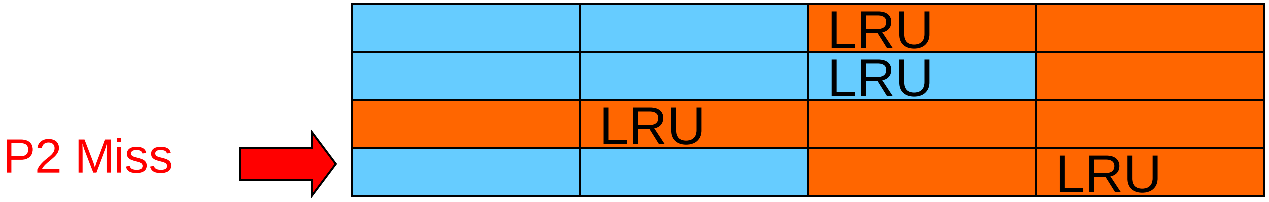
$$M_1^{ij} = \left|X_i - X_j\right|, \; where \; X_i = \frac{Miss\_shared_i}{Miss\_alone_i}$$

$$M_3^{ij} = \left|X_i - X_j\right|, \; where \; X_i = \frac{MissRate\_shared_i}{MissRate\_alone_i}$$

Block-Granularity Partitioning

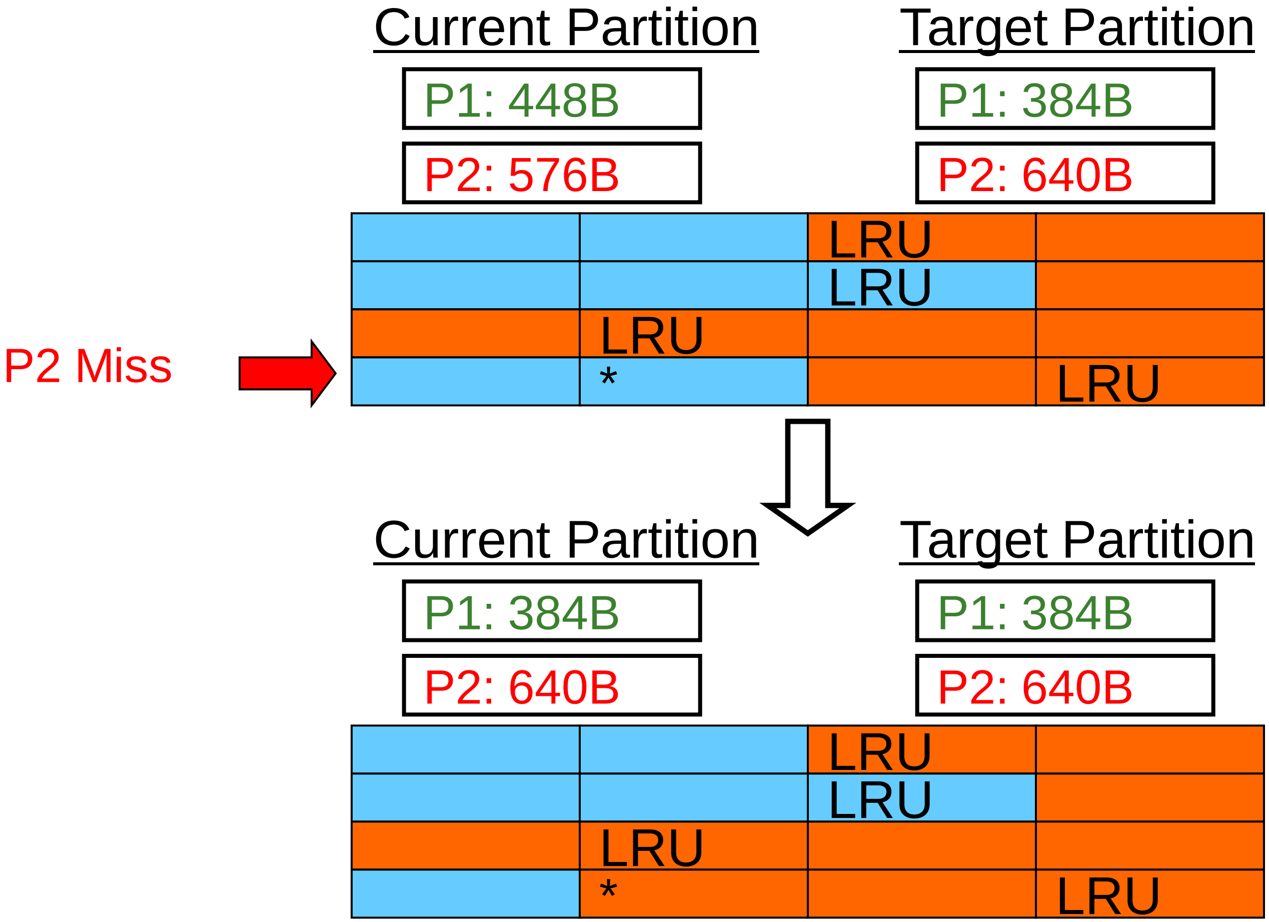
- Modified LRU cache replacement policy
  - G. Suh, et. al., HPCA 2002

Current Partition	Target Partition
P1: 448B	P1: 384B
P2: 576B	P2: 640B



Block-Granularity Partitioning

- Modified LRU cache replacement policy
  - G. Suh, et. al., HPCA 2002



Current Partition

P1: 448B

P2: 576B

Target Partition

P1: 384B

P2: 640B

P2 Miss

🟦	🟦	🟧 LRU	🟧
🟦	🟦	🟦 LRU	🟧
🟧	🟧 LRU	🟧	🟧
🟦	🟦 *	🟧	🟧 LRU

Current Partition

P1: 384B

P2: 640B

Target Partition

P1: 384B

P2: 640B

🟦	🟦	🟧 LRU	🟧
🟦	🟦	🟦 LRU	🟧
🟧	🟧 LRU	🟧	🟧
🟦	🟧 *	🟧	🟧 LRU

Dynamic Fair Caching Algorithm

Ex) Optimizing
M3 metric

MissRate alone

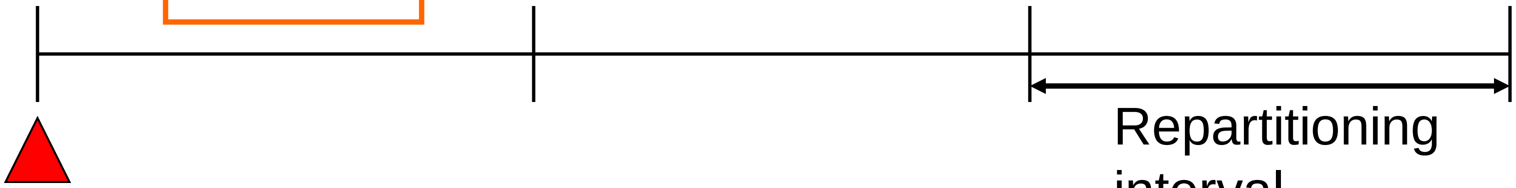
P1:

P2:

MissRate shared

P1:

P2:


Repartitioning
interval

Target Partition

P1:

P2:

Dynamic Fair Caching Algorithm

1st Interval

MissRate alone

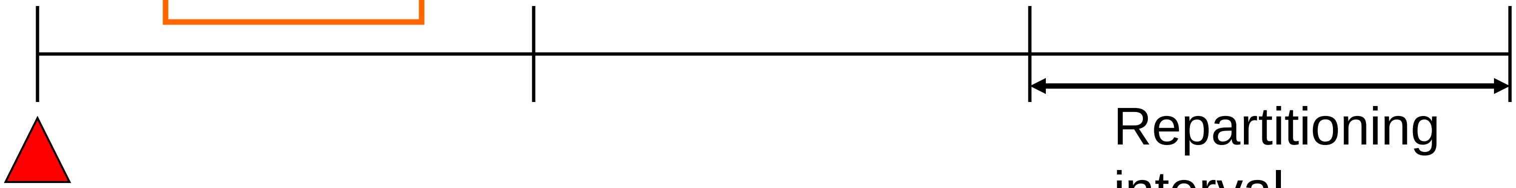
P1:20%

P2: 5%

MissRate shared

P1:20%

P2:15%



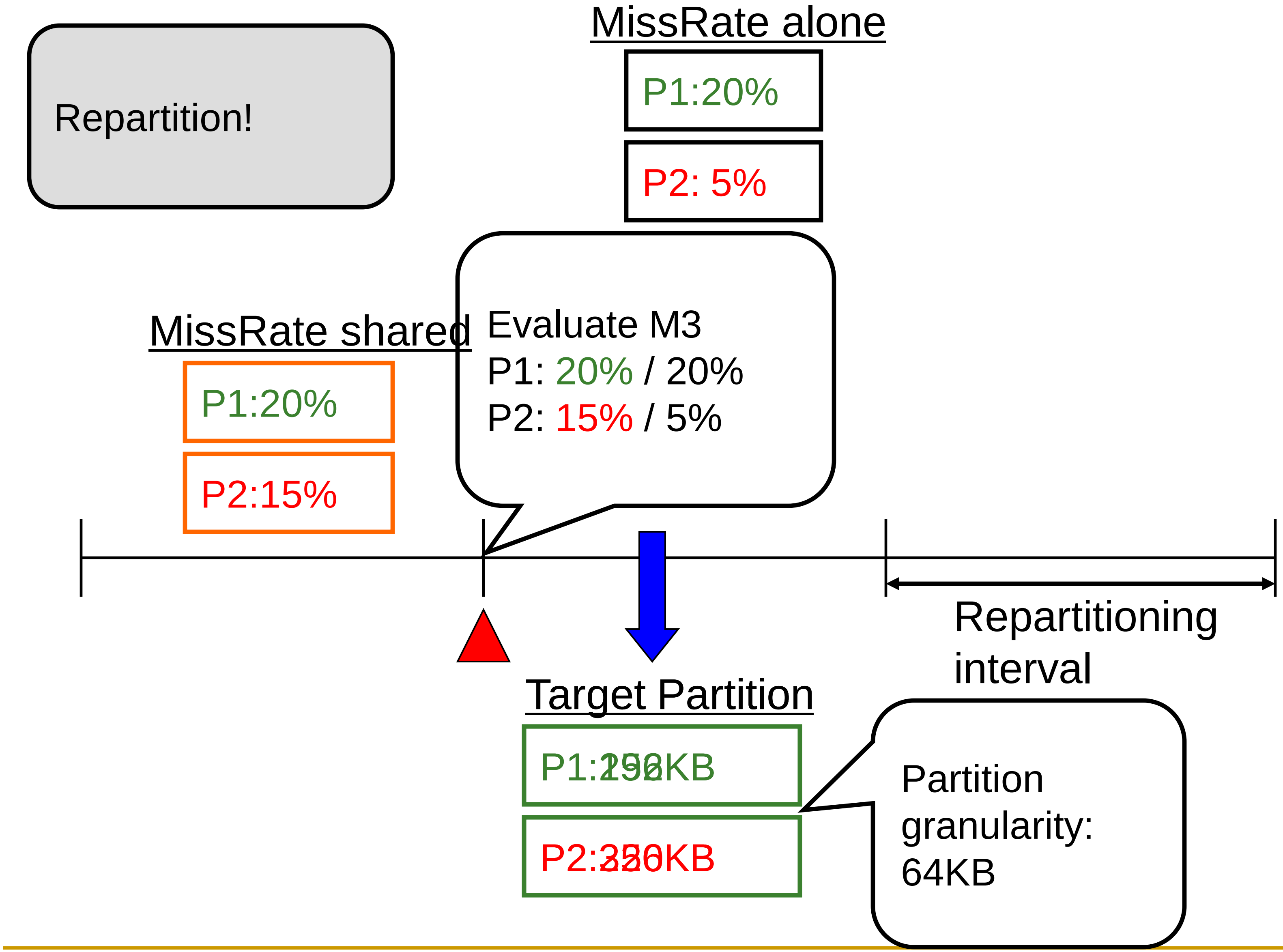
Repartitioning interval

Target Partition

P1:256KB

P2:256KB

Dynamic Fair Caching Algorithm



Repartition!

MissRate alone

P1:20%

P2: 5%

MissRate shared

P1:20%

P2:15%

Evaluate M3
P1: 20% / 20%
P2: 15% / 5%

Repartitioning interval

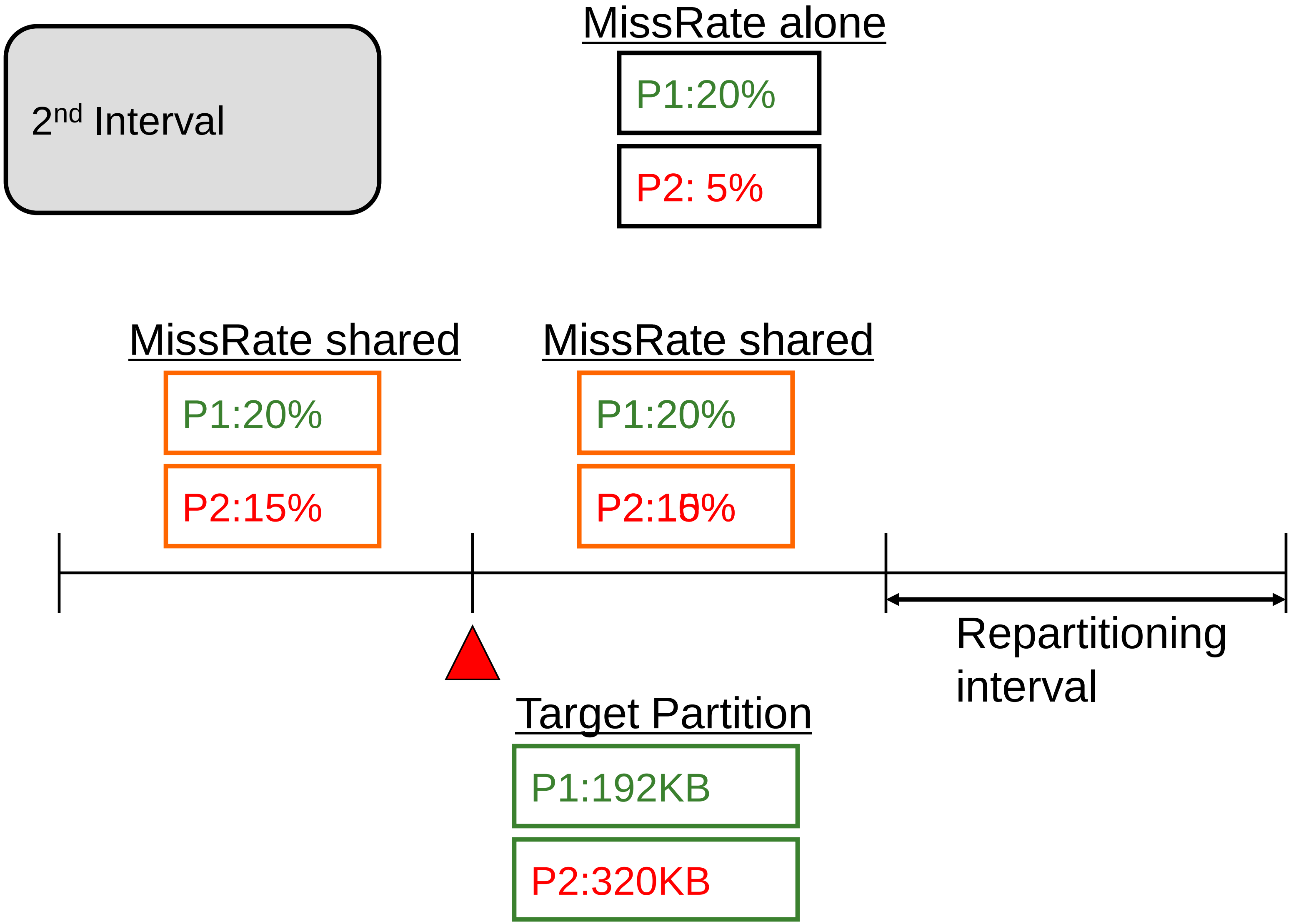
Target Partition

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%

P2:15%

P2:10%

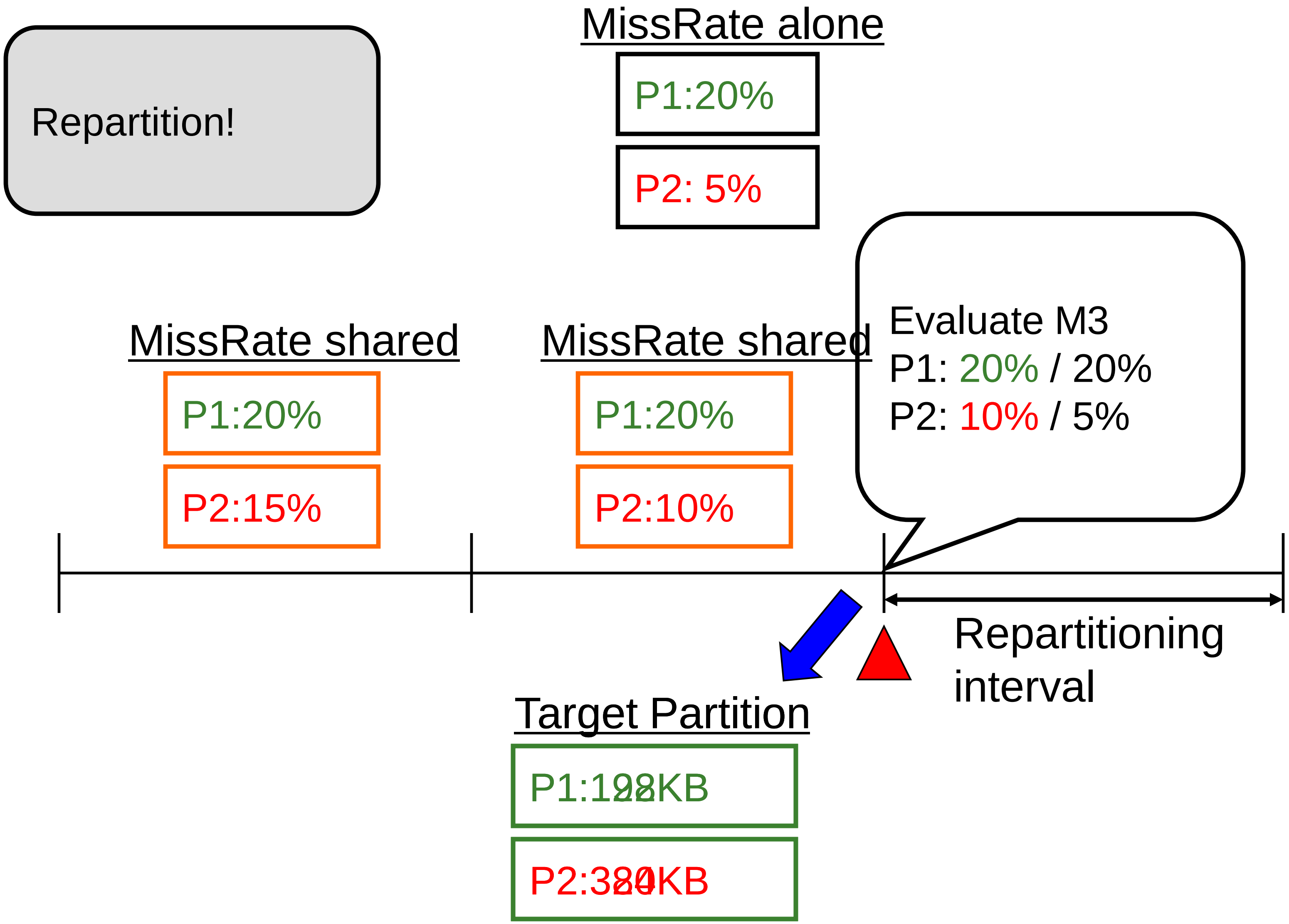
Repartitioning interval

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 M3
P1: 20% / 20%
P2: 10% / 5%

Repartitioning interval

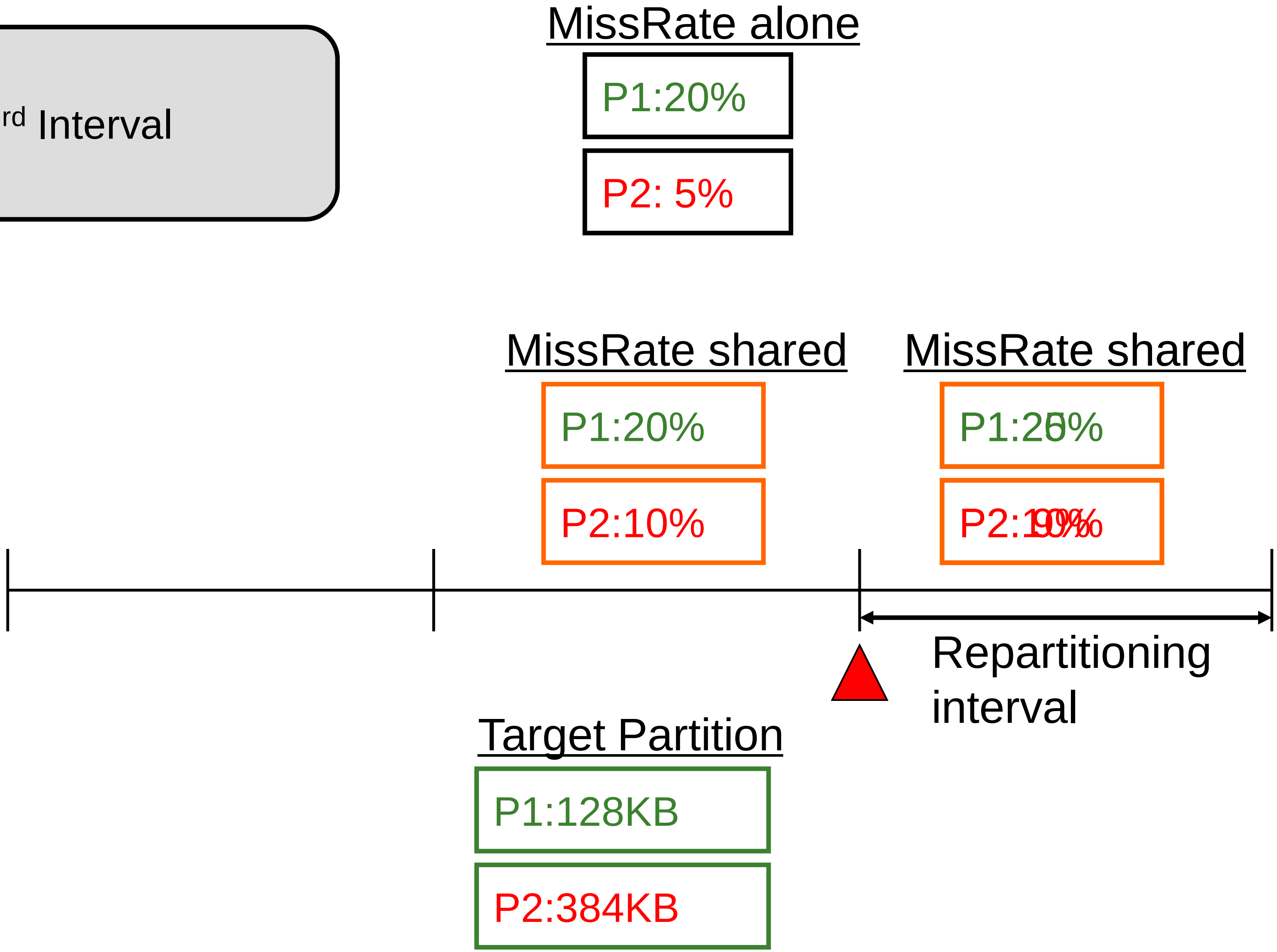
Target Partition

P1:192KB
P1:128KB

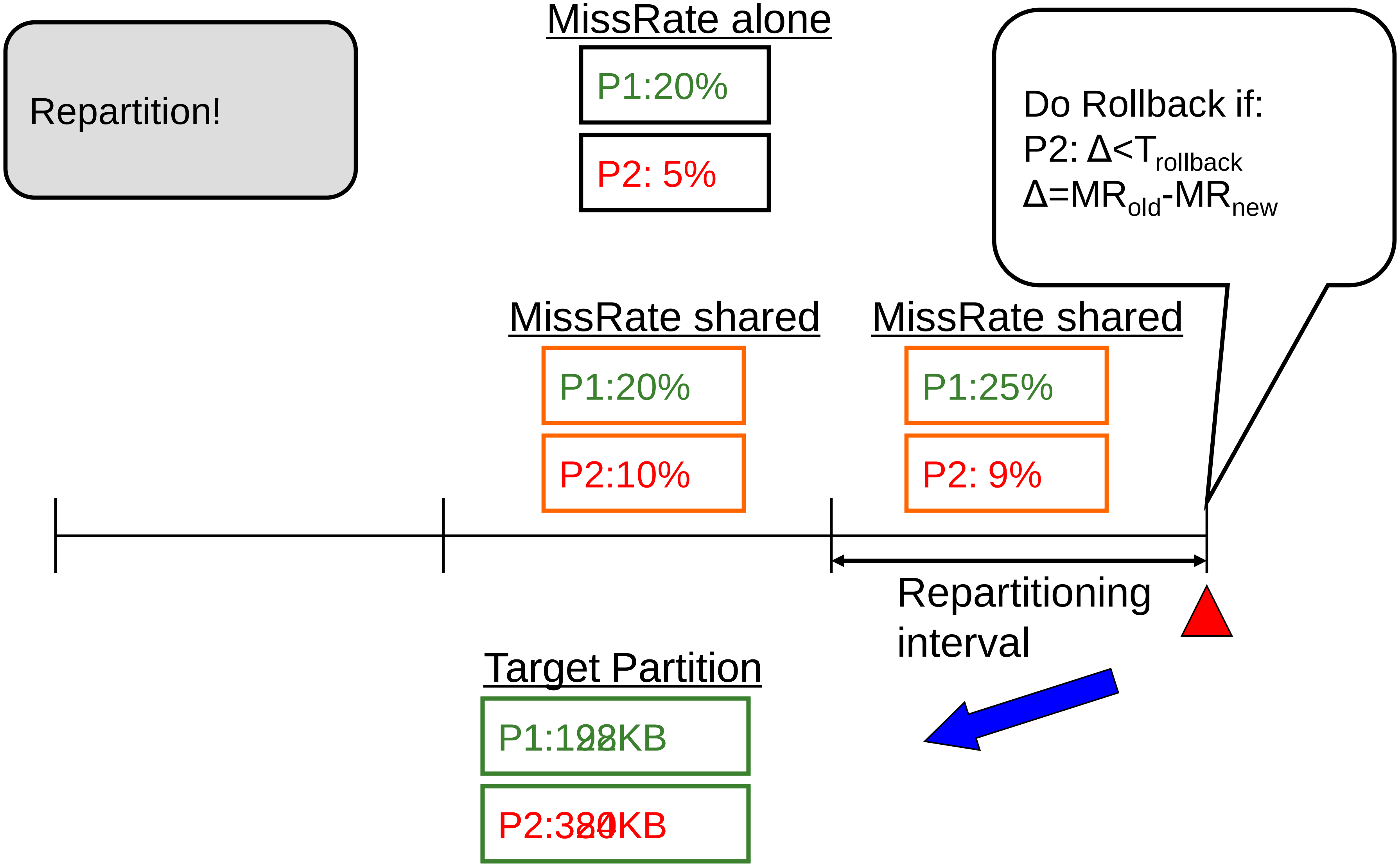
P2:320KB
P2:384KB

Dynamic Fair Caching Algorithm

3rd Interval



Dynamic Fair Caching Algorithm



Repartition!

MissRate alone

P1:20%

P2: 5%

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

MissRate shared

P1:20%

P2:10%

MissRate shared

P1:25%

P2: 9%

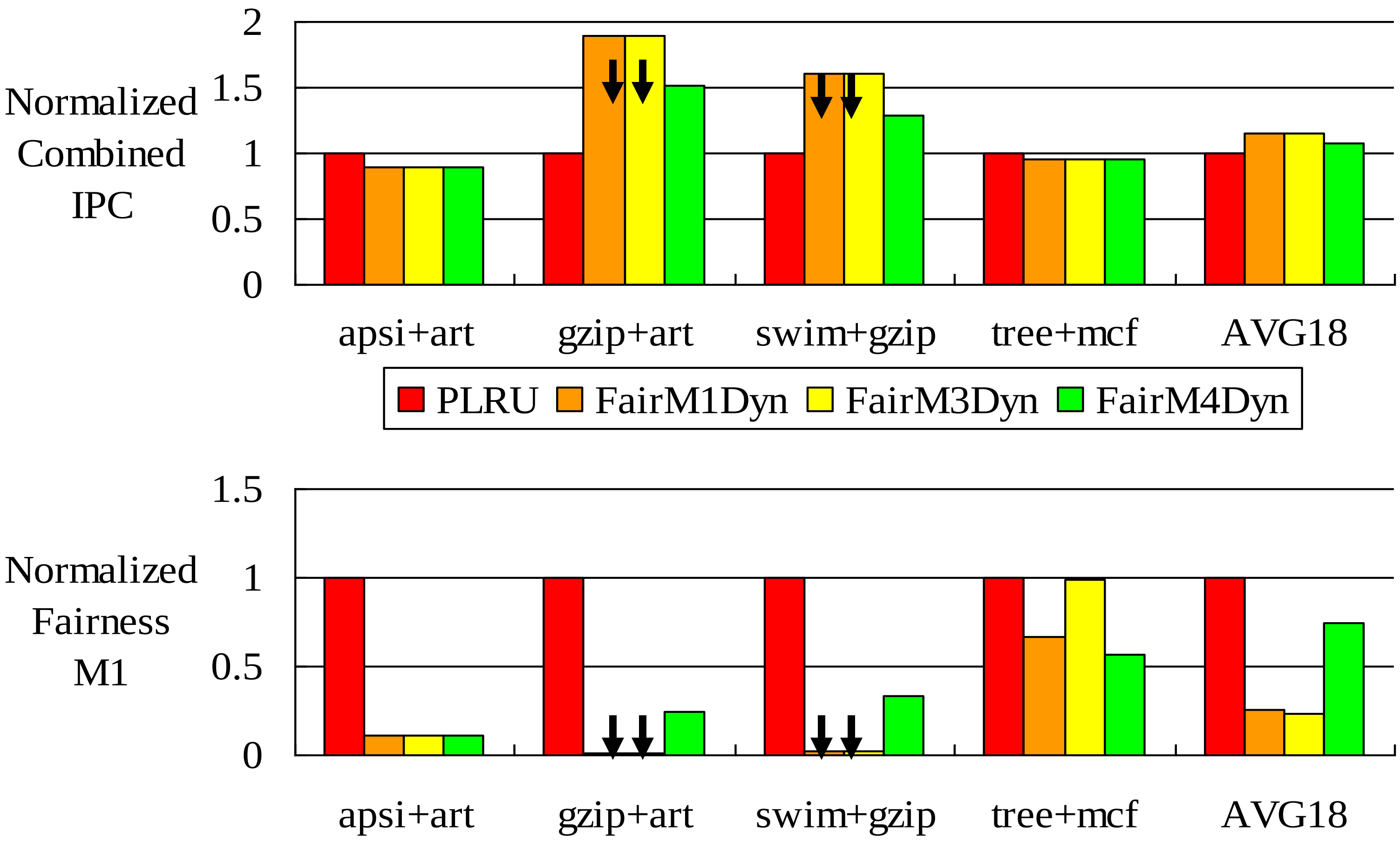
Repartitioning interval

Target Partition

P1:192KB
P1:128KB

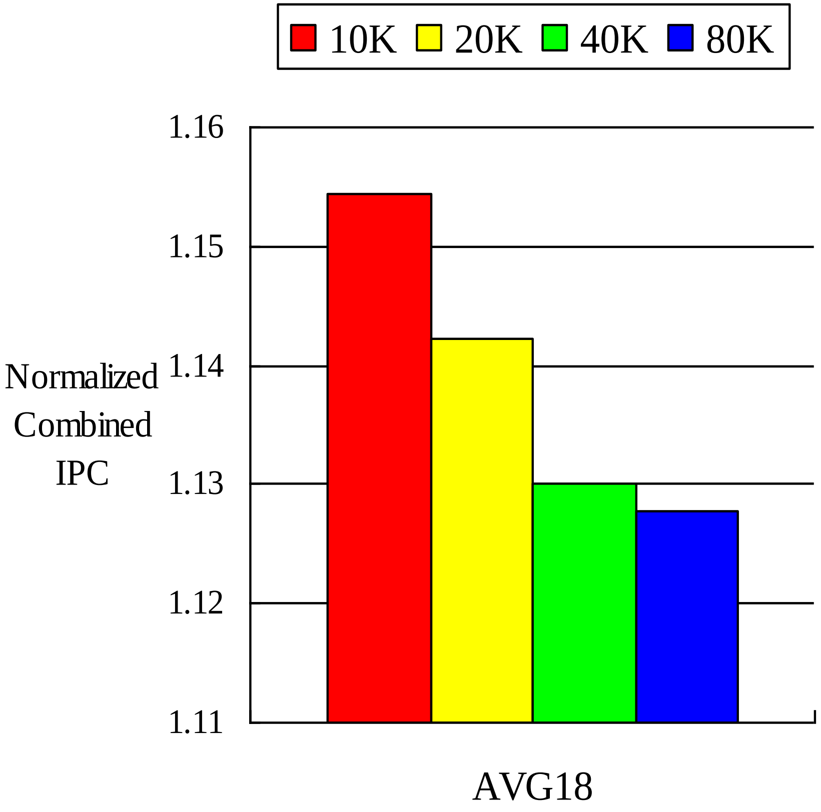
P2:320KB
P2:384KB

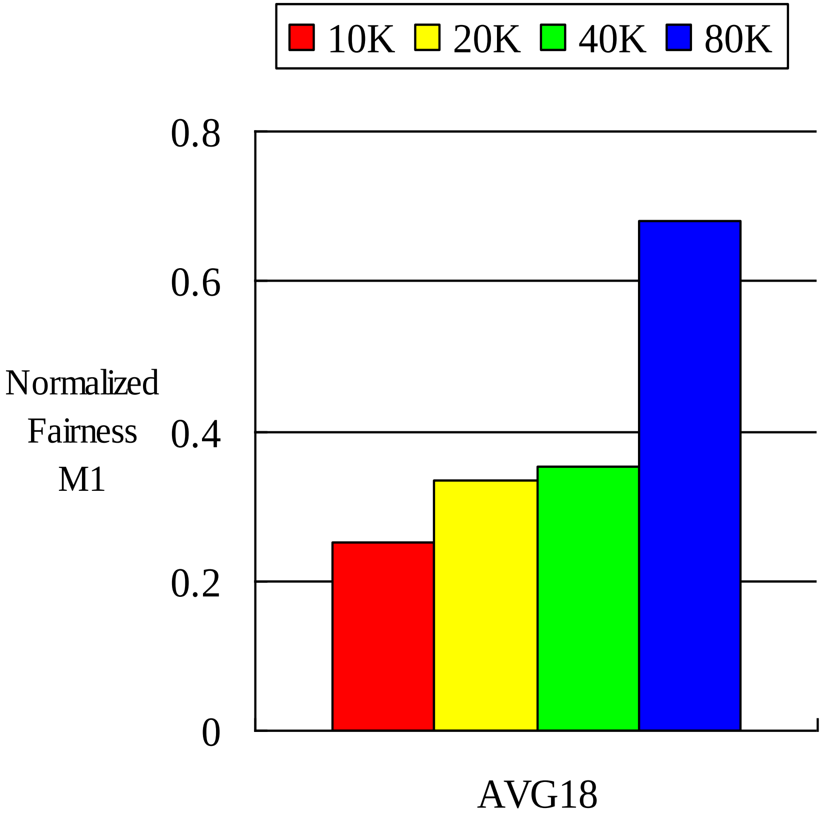
Dynamic Fair Caching Results



- Improves both fairness and throughput

Effect of Partitioning Interval





- Fine-grained partitioning is important for both fairness and throughput

Benefits of Fair Caching

- Problems of unfair cache sharing
  - Sub-optimal throughput
  - Thread starvation
  - Priority inversion
  - Thread-mix dependent performance

- Benefits of fair caching
  - Better fairness
  - Better throughput
  - Fair caching likely simplifies OS scheduler design

Advantages/Disadvantages of the Approach

- Advantages

  + No (reduced) starvation

  + Better average throughput

- Disadvantages

  - Scalable to many cores?

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

  - Does this provide performance isolation in cache?

  - Alone miss rate estimation can be incorrect (estimation interval different from enforcement interval)

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

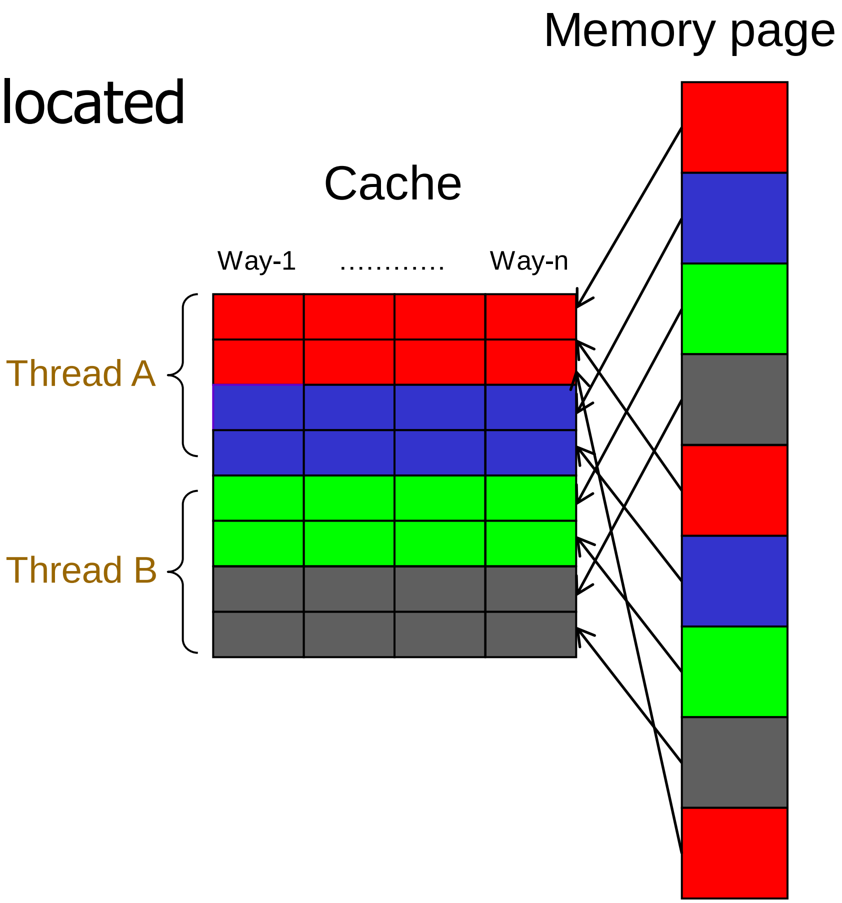
OS Based Cache Partitioning

- 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

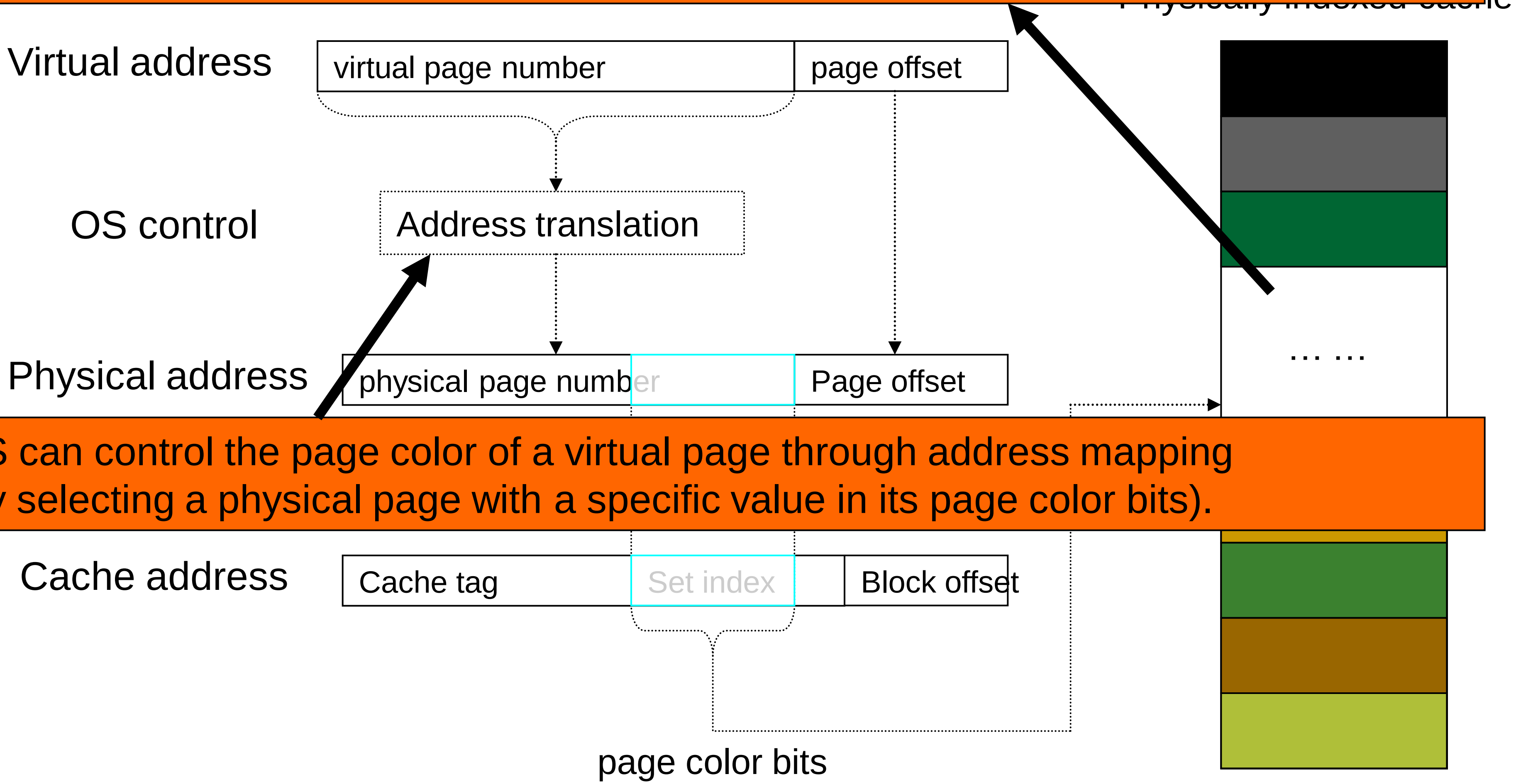
Page Coloring

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



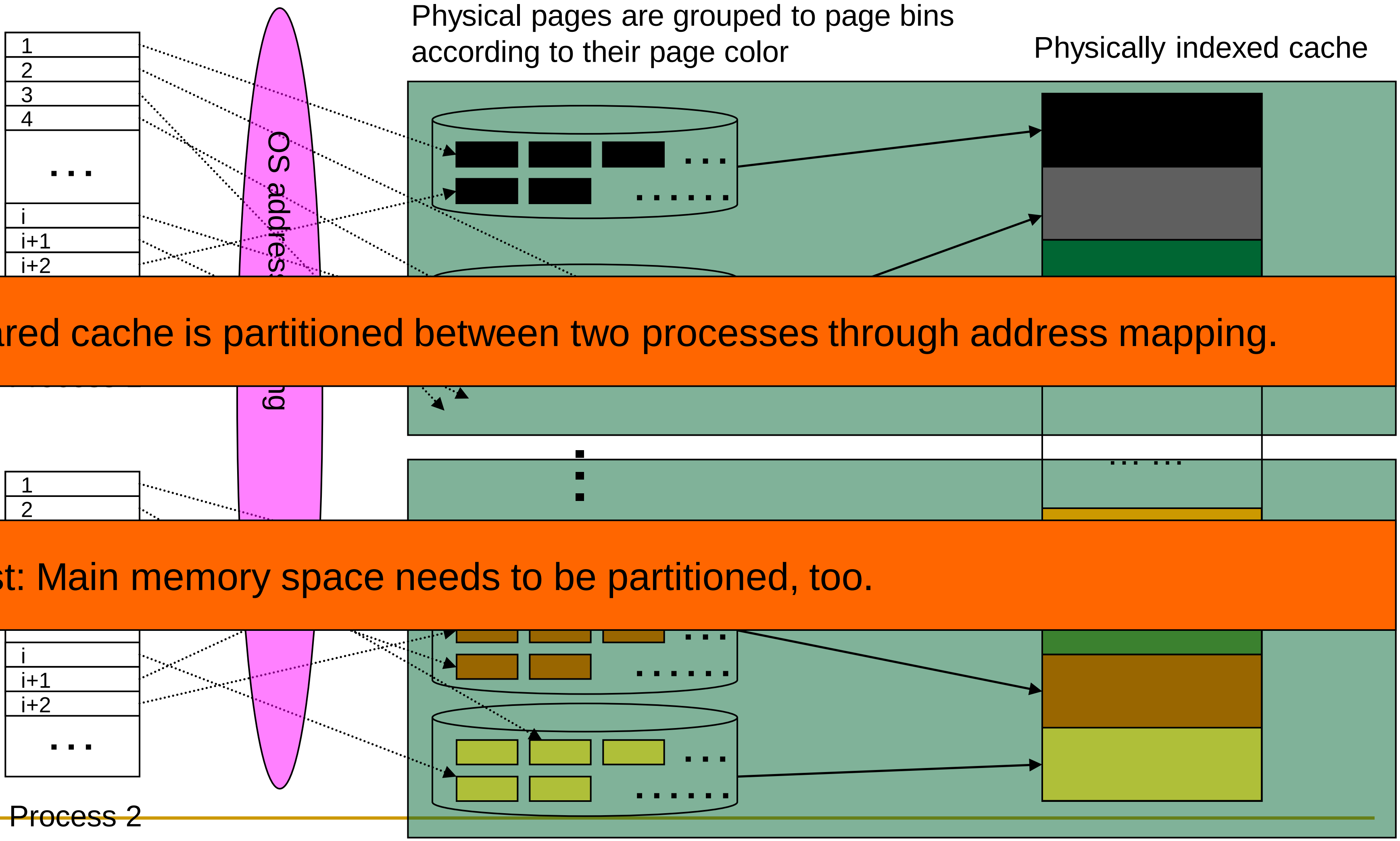
Page Coloring

- 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).

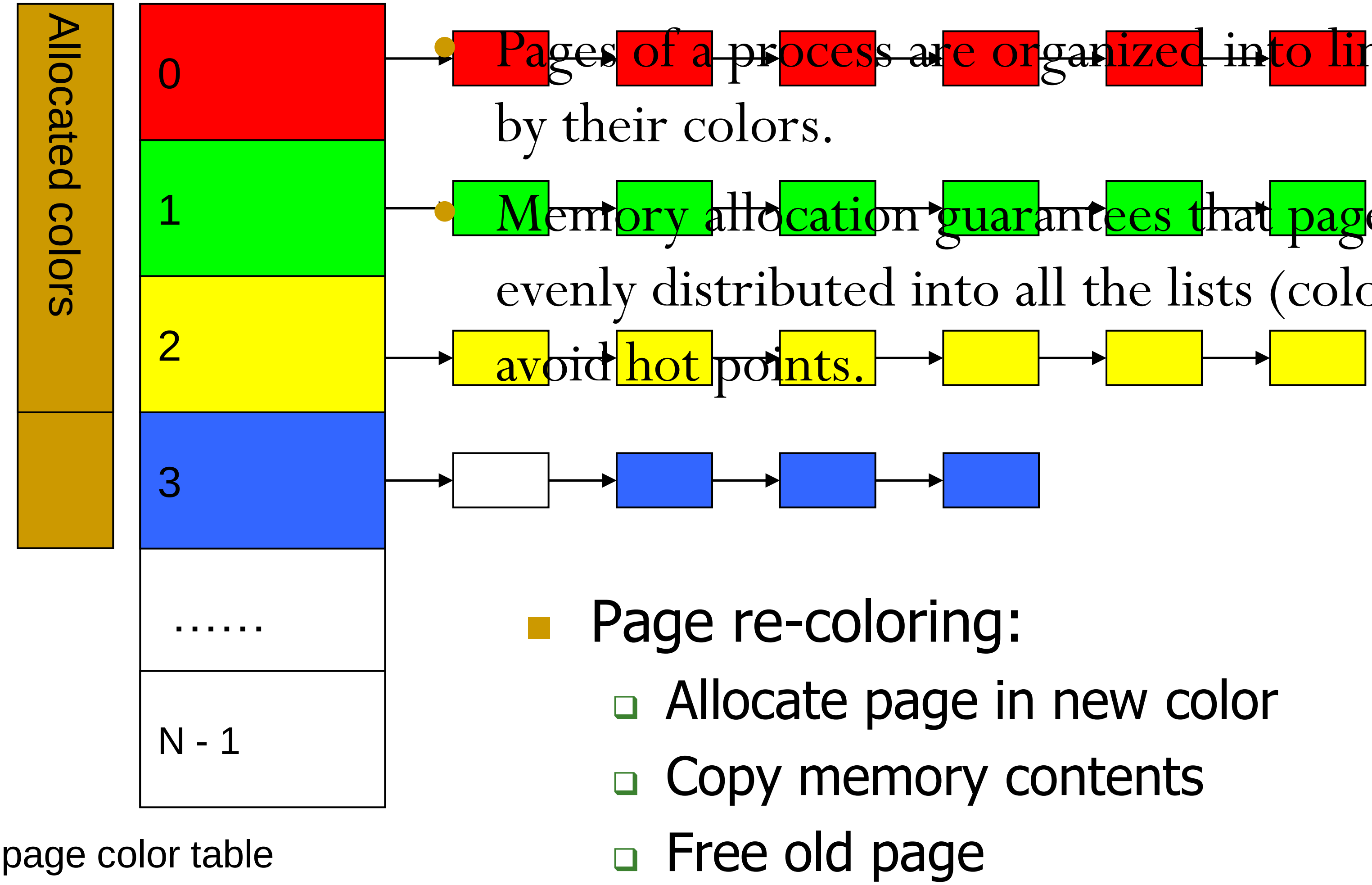
Static Cache Partitioning using Page Coloring



Shared cache is partitioned between two processes through address mapping.

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

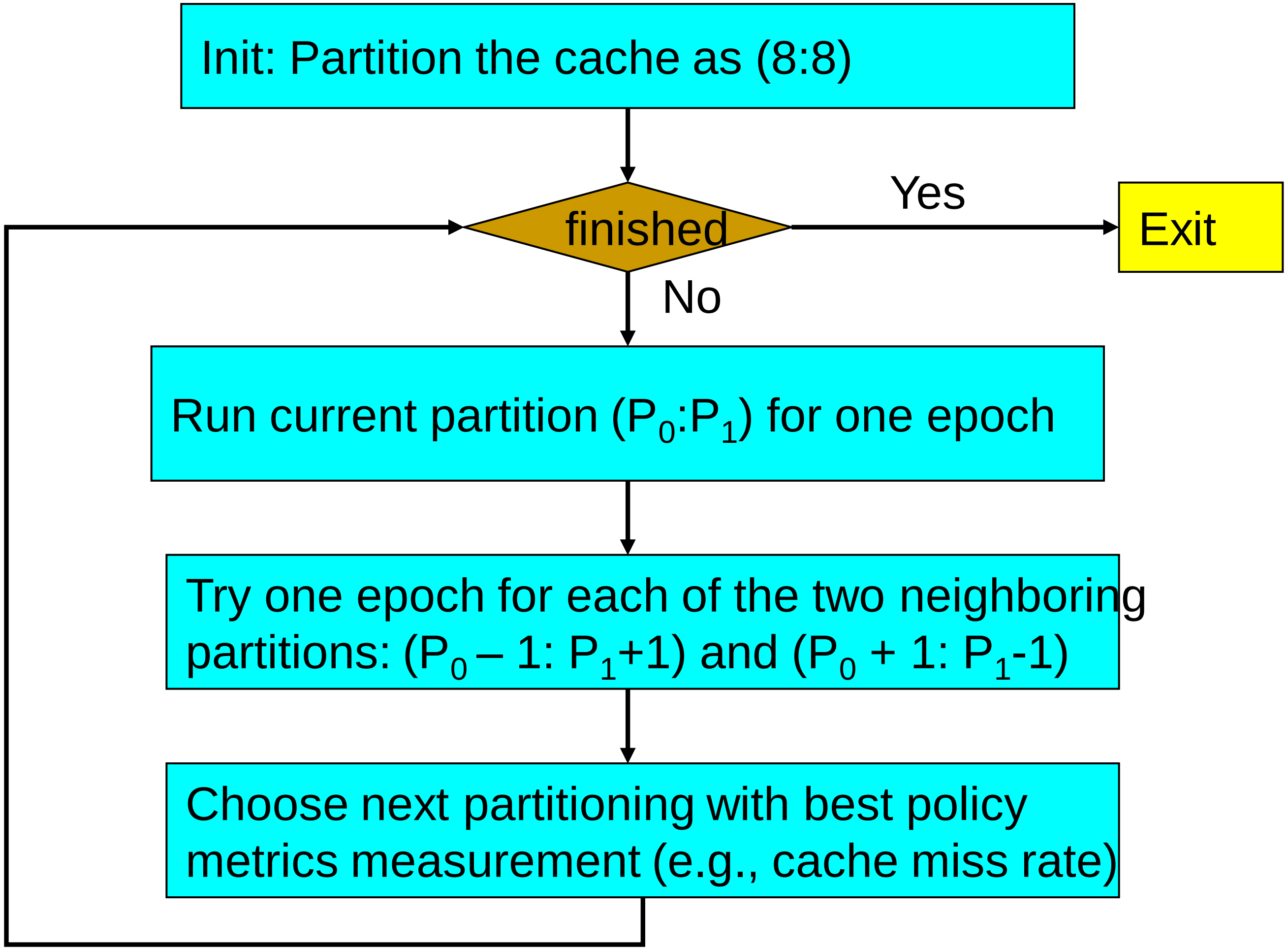
Dynamic Cache Partitioning via Page Re-Coloring



- 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 Dual Core

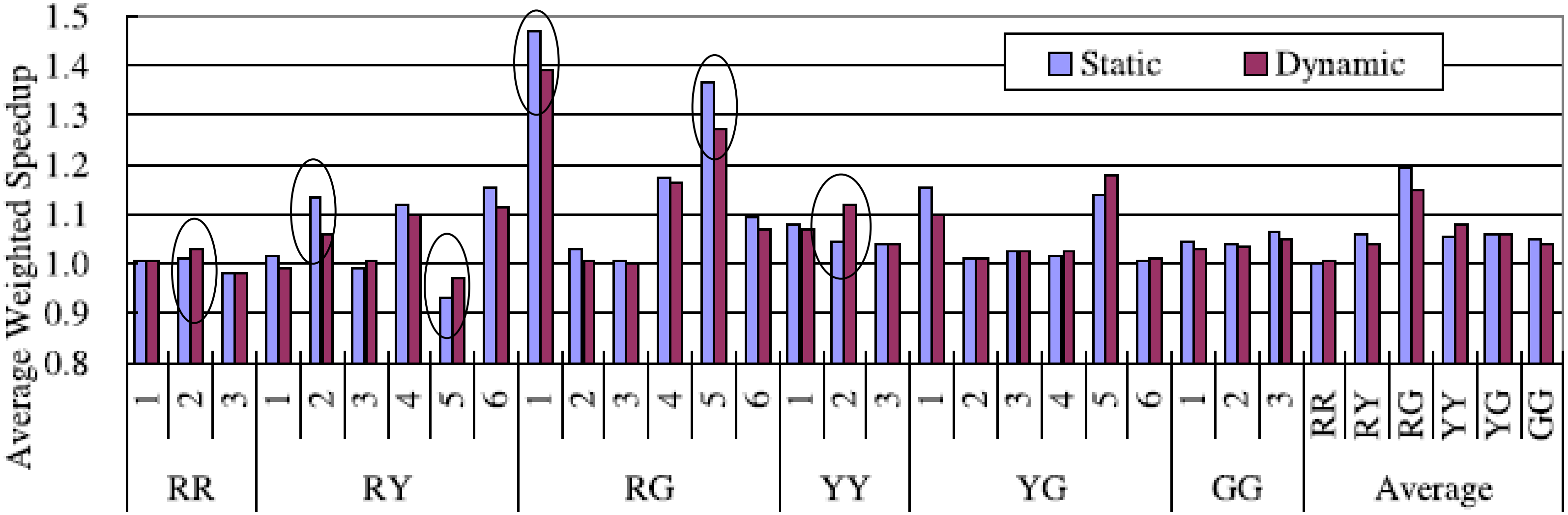


Init: Partition the cache as (8:8)
finished
Yes
Exit
No
Run current partition (P_0:P_1) for one epoch
Try one epoch for each of the two neighboring partitions: (P_0 – 1: P_1+1) and (P_0 + 1: P_1-1)
Choose next partitioning with best policy metrics measurement (e.g., cache miss rate)

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

  - Less flexible: large granularity (page-based instead of 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)

Computer Architecture: (Shared) Cache Management

Prof. Onur Mutlu

Carnegie Mellon University

Backup slides

Referenced Readings (I)

- 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.
- Pekhimenko et al., "Linearly Compressed Pages: A Main Memory Compression Framework with Low Complexity and Low Latency," SAFARI Technical Report 2013.
- Qureshi et al., "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.
- Kim et al., "Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture," PACT 2004.

Referenced Readings (II)

- Fedorova et al., "Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler," PACT 2007.
- 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.
- 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.

Private/Shared Caching

Private/Shared Caching

- Example: Adaptive spill/receive caching

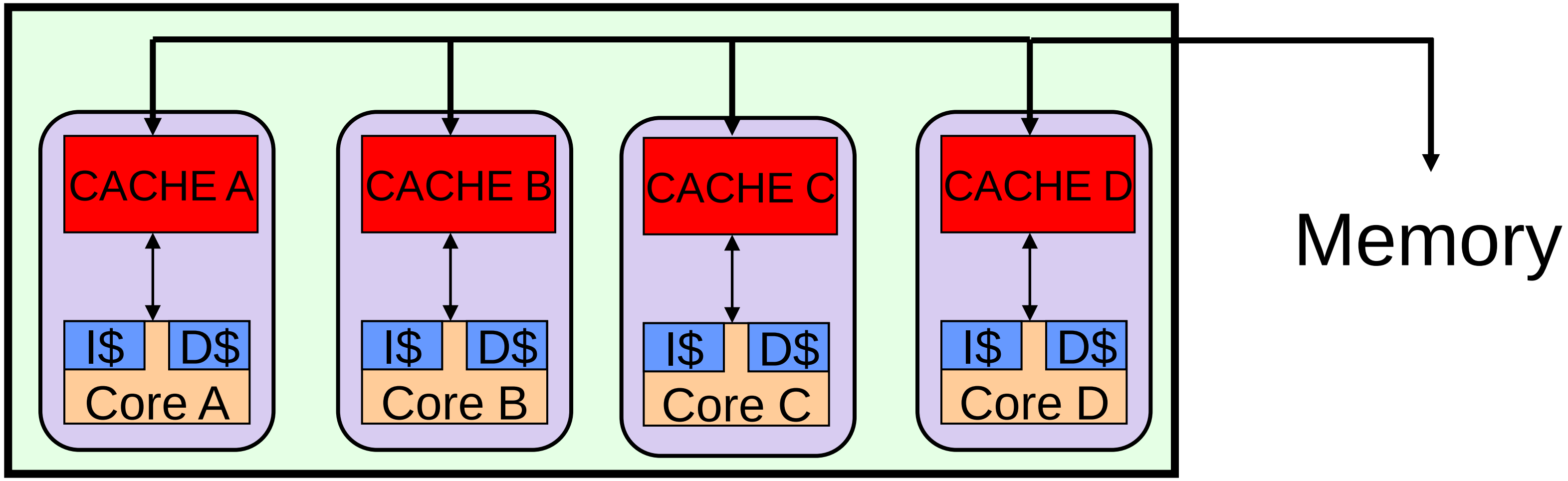
- Goal: Achieve the benefits of private caches (low latency, performance isolation) while sharing cache capacity across cores

- 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

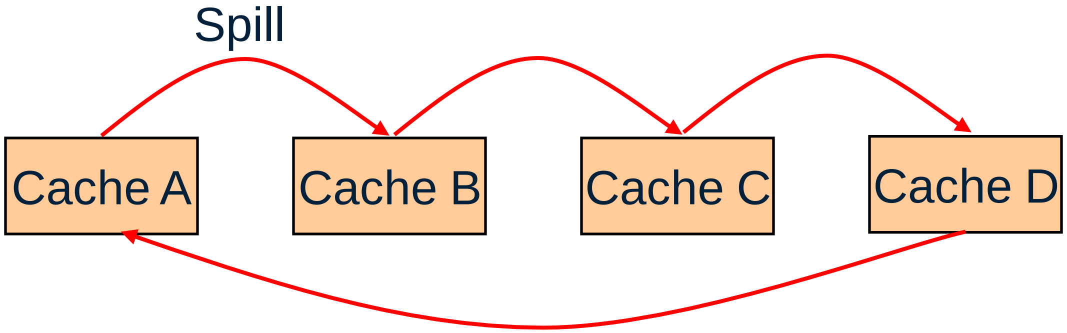


Problem: When one core needs more cache and other core has spare cache, private-cache CMPs cannot share capacity

Cache Line Spilling

Spill evicted line from one cache to neighbor cache

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



Problem with CC:

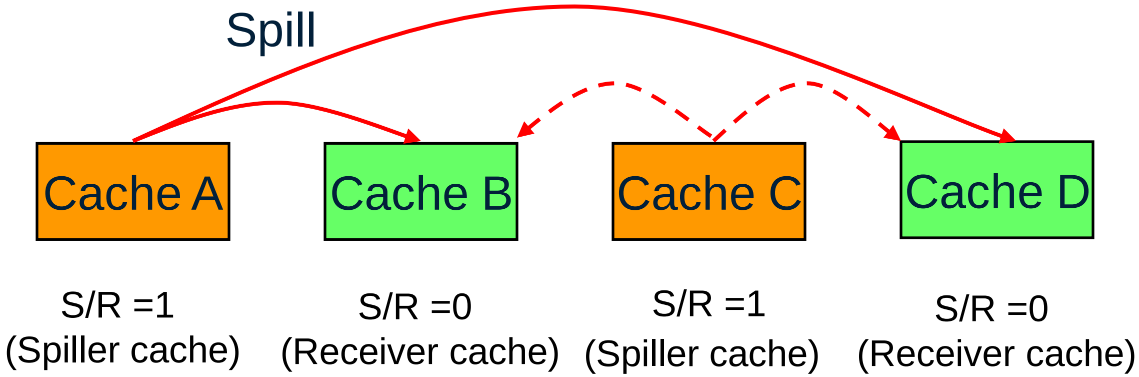
1. Performance depends on the parameter (spill probability)
2. All caches spill as well as receive ➔ Limited improvement

Goal: Robust High-Performance Capacity Sharing with Negligible Overhead

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)

Dynamic Spill-Receive via "Set Dueling"

Divide the cache in three:

- Spiller sets
- Receiver sets
- Follower sets (winner of spiller, receiver)

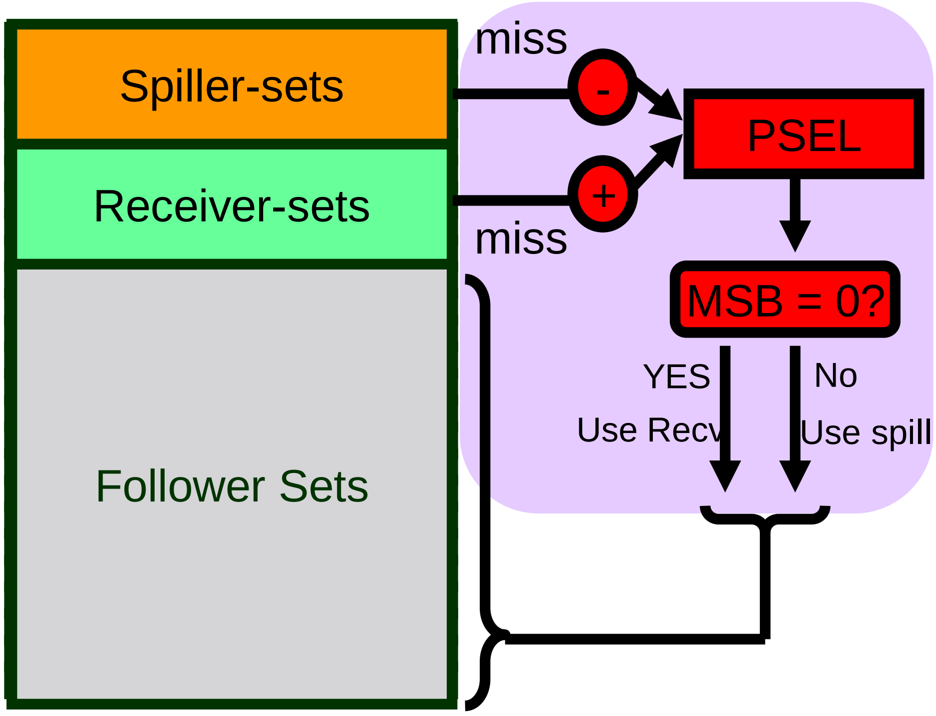
n-bit PSEL counter

misses to spiller-sets: PSEL--

misses to receiver-set: PSEL++

MSB of PSEL decides policy for Follower sets:

- MSB = 0, Use spill
- MSB = 1, Use receive

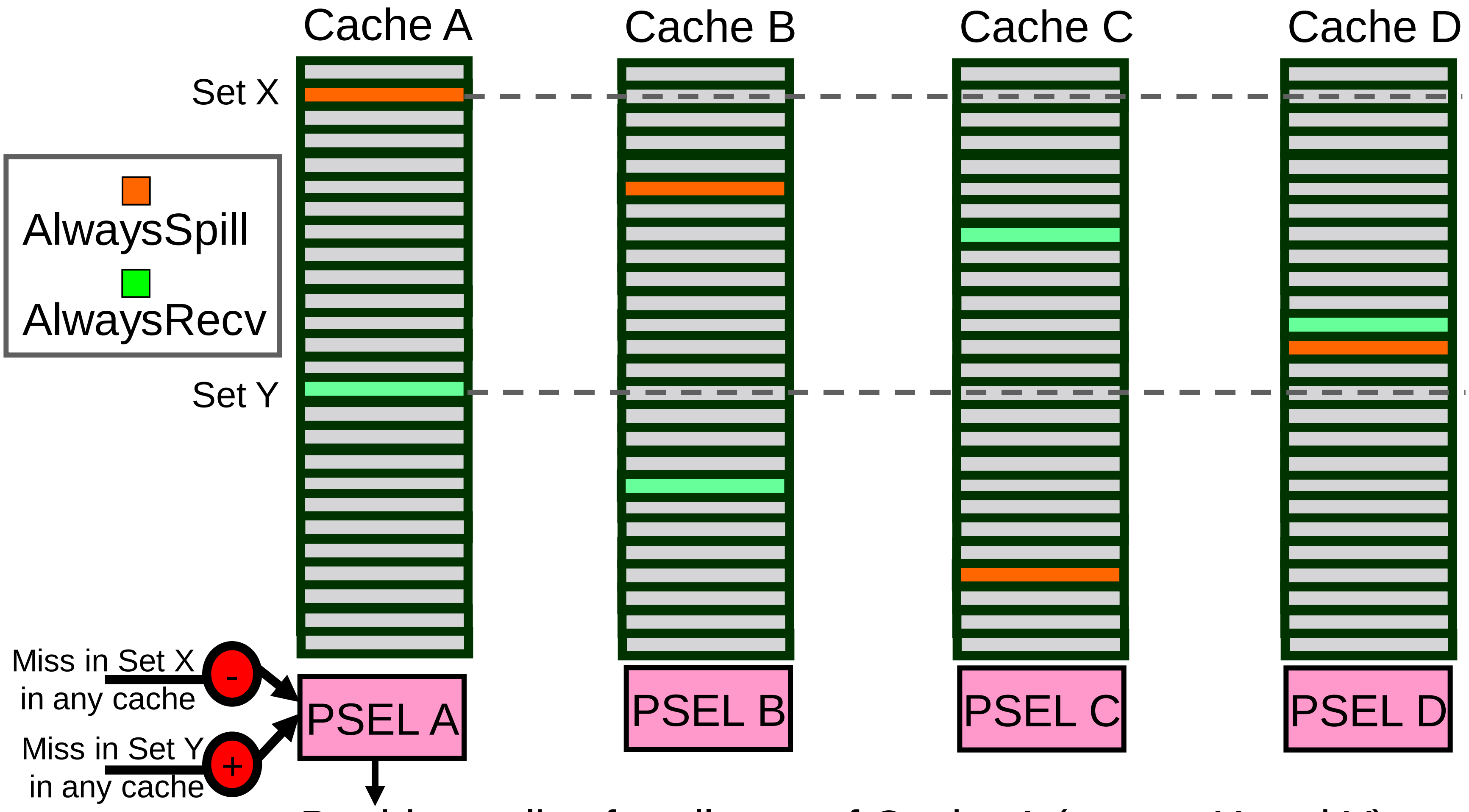


monitor → choose → apply

(using a single counter)

Dynamic Spill-Receive Architecture

Each cache learns whether it should act as a spiller or receiver



Cache A

Cache B

Cache C

Cache D

Set X

Set Y

🟧 AlwaysSpill

🟩 AlwaysRecv

Miss in Set X in any cache -

Miss in Set Y in any cache +

PSEL A

PSEL B

PSEL C

PSEL D

Decides policy for all sets of Cache A (except X and Y)

Experimental Setup

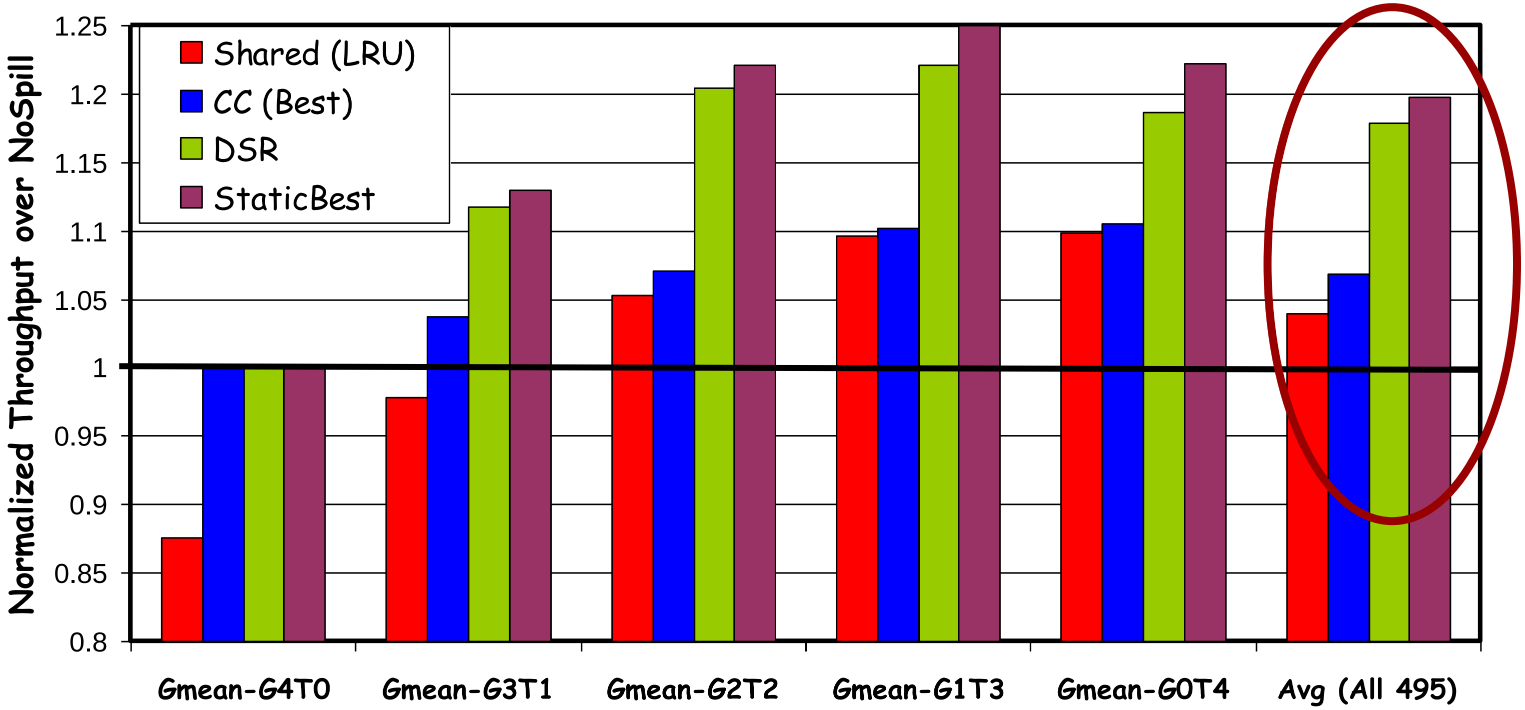
- Baseline Study:
  - 4-core CMP with in-order cores
  - Private Cache Hierarchy: 16KB L1, 1MB L2
  - 10 cycle latency for local hits, 40 cycles for remote hits

- Benchmarks:
  - 6 benchmarks that have extra cache: “Givers” (G)
  - 6 benchmarks that benefit from more cache: “Takers” (T)
  - All 4-thread combinations of 12 benchmarks: 495 total

Five types of workloads:

G4T0	G3T1	G2T2	G1T3	G0T4

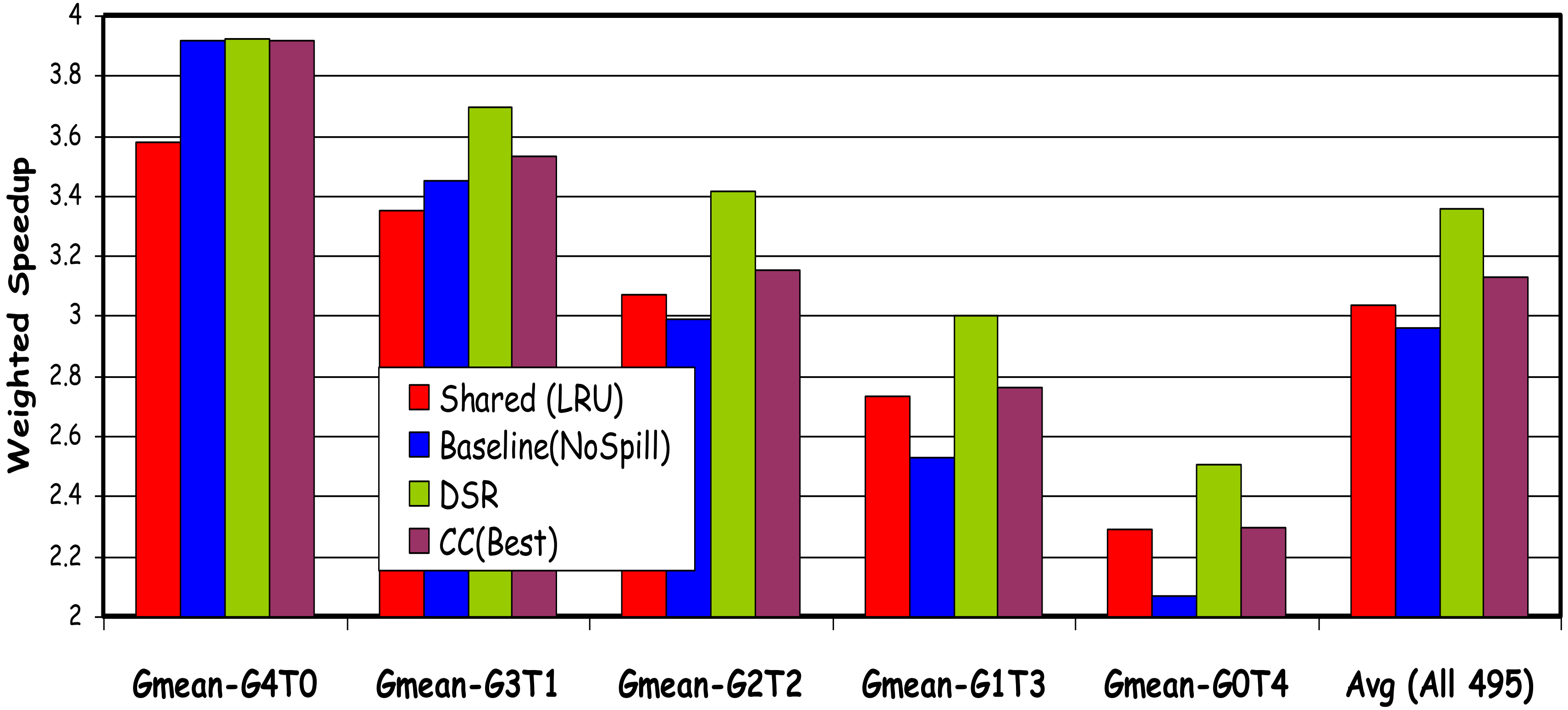
Results for Throughput



On average, DSR improves throughput by 18%, co-operative caching by 7%
DSR provides 90% of the benefit of knowing the best decisions a priori

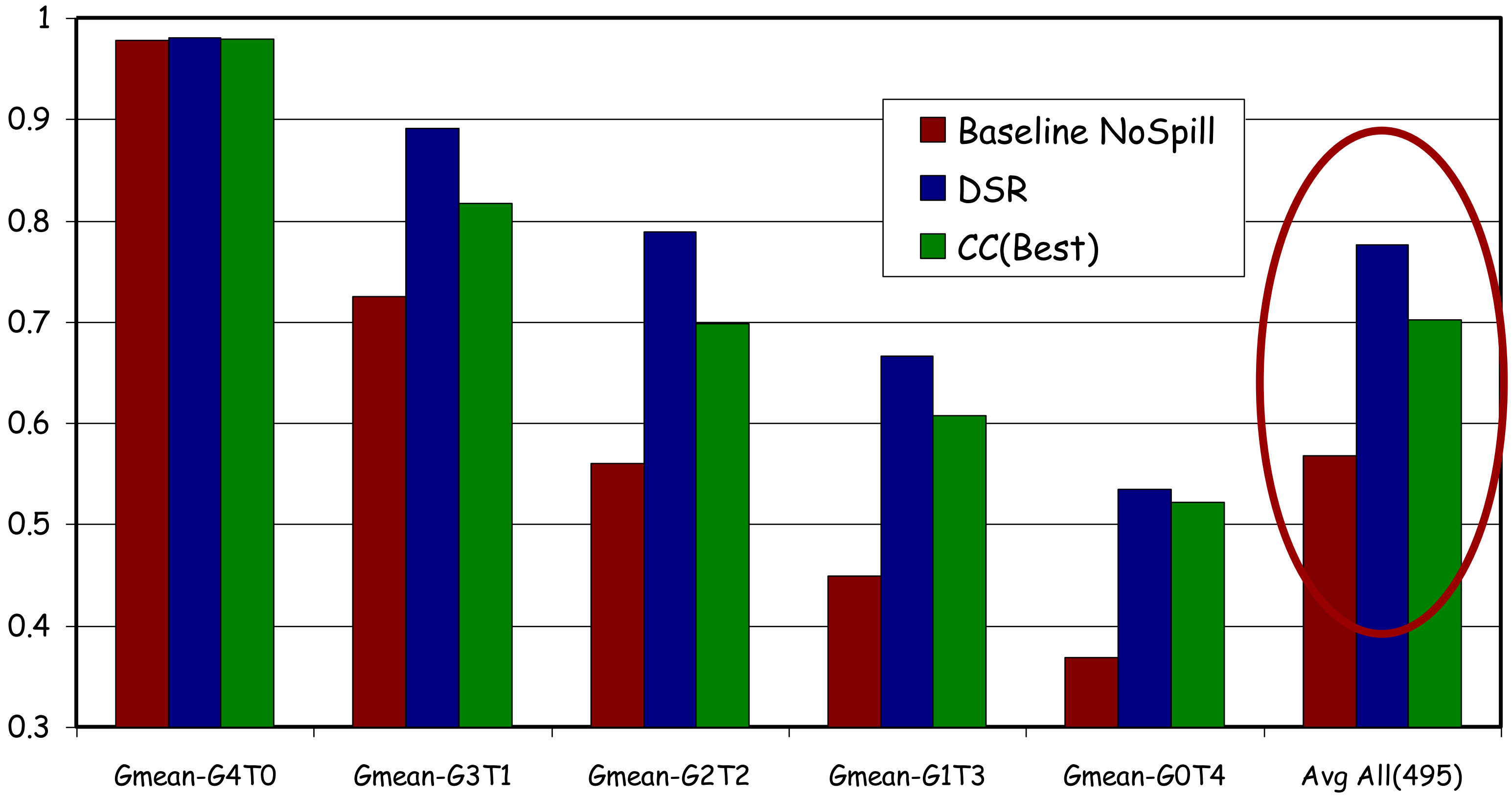
* DSR implemented with 32 dedicated sets and 10 bit PSEL counters

Results for Weighted Speedup



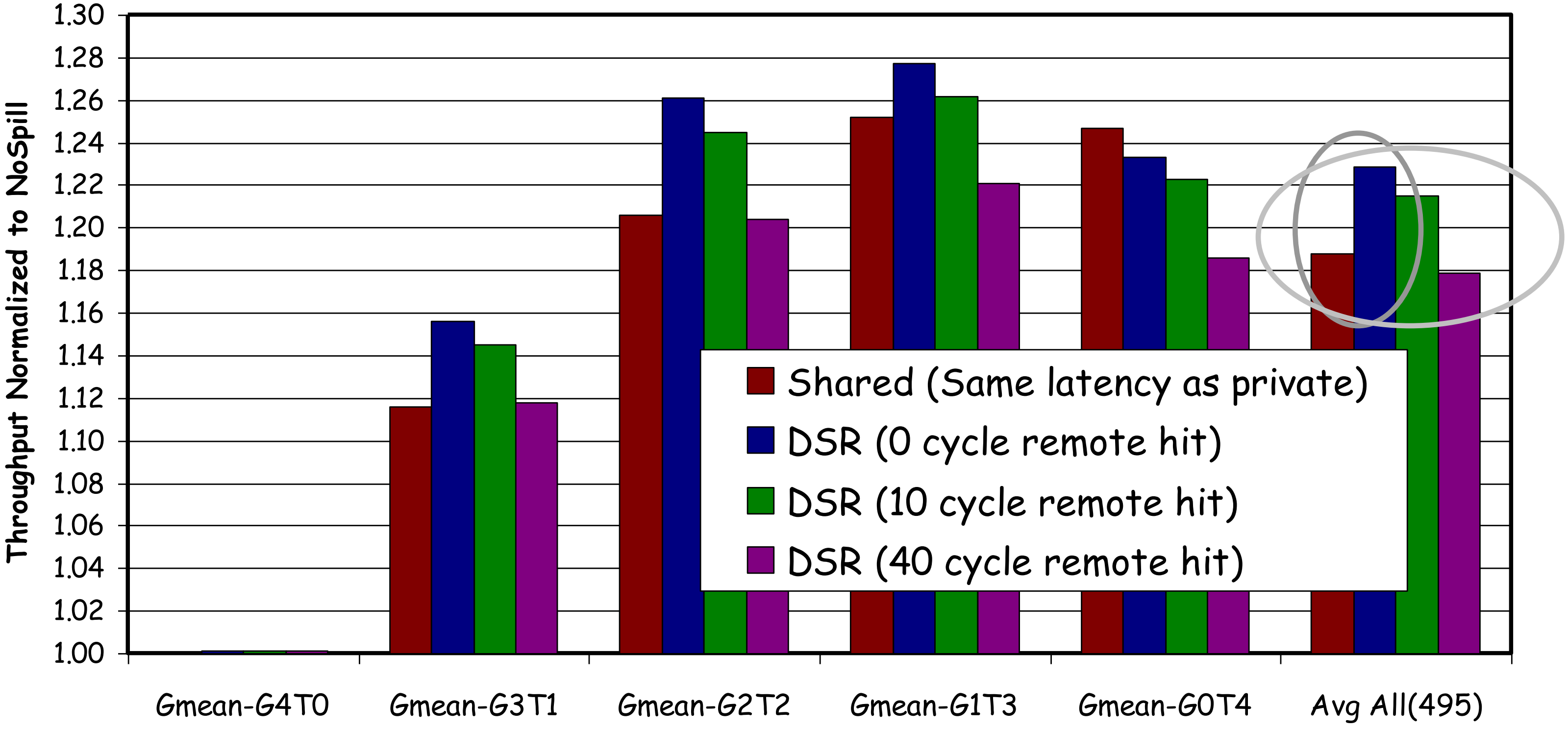
On average, DSR improves weighted speedup by 13%

Results for Hmean Speedup



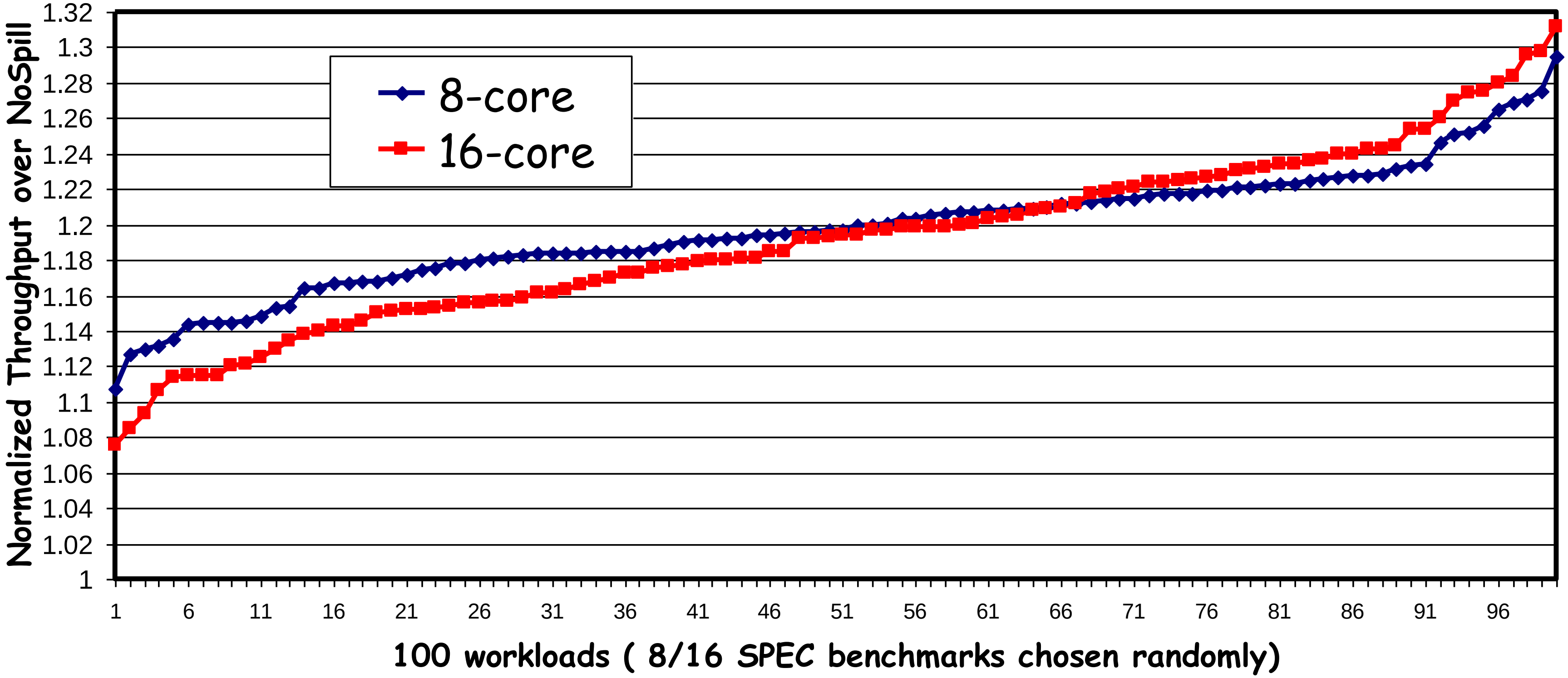
On average, DSR improves Hmean Fairness from 0.58 to 0.78

DSR vs. Faster Shared Cache



DSR (with 40 cycle extra for remote hits) performs similar to shared cache with zero latency overhead and crossbar interconnect

Scalability of DSR



DSR improves average throughput by 19% for both systems
(No performance degradation for any of the workloads)

Quality of Service with DSR

For 1 % of the 495x4 =1980 apps, DSR causes IPC loss of > 5%

In some cases, important to ensure that performance does not degrade compared to dedicated private cache ➔ QoS

DSR can ensure QoS: change PSEL counters by weight of miss:

$$\Delta\text{Miss} = \text{MissesWithDSR} - \text{MissesWithNoSpill}$$

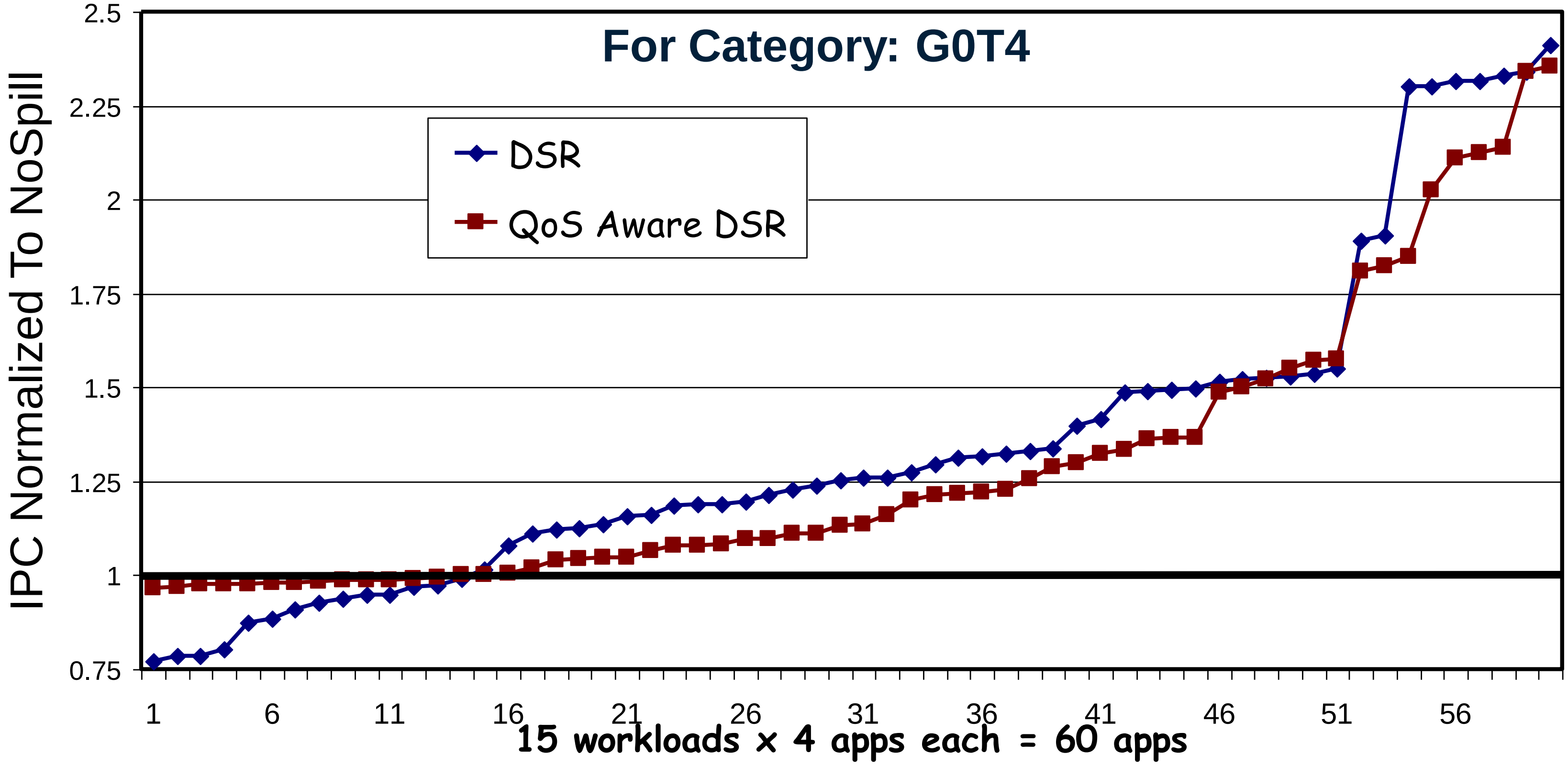
Estimated by Spiller Sets

$$\boxed{\text{Weight of Miss} = 1 + \text{Max}(0, f(\Delta\text{Miss}))}$$

Calculate weight every 4M cycles. Needs 3 counters per core

Over time, ΔMiss → 0, if DSR is causing more misses.

IPC of QoS-Aware DSR



IPC curves for other categories almost overlap for the two schemes.
Avg. throughput improvement across all 495 workloads similar (17.5% vs. 18%)

Distributed Caches

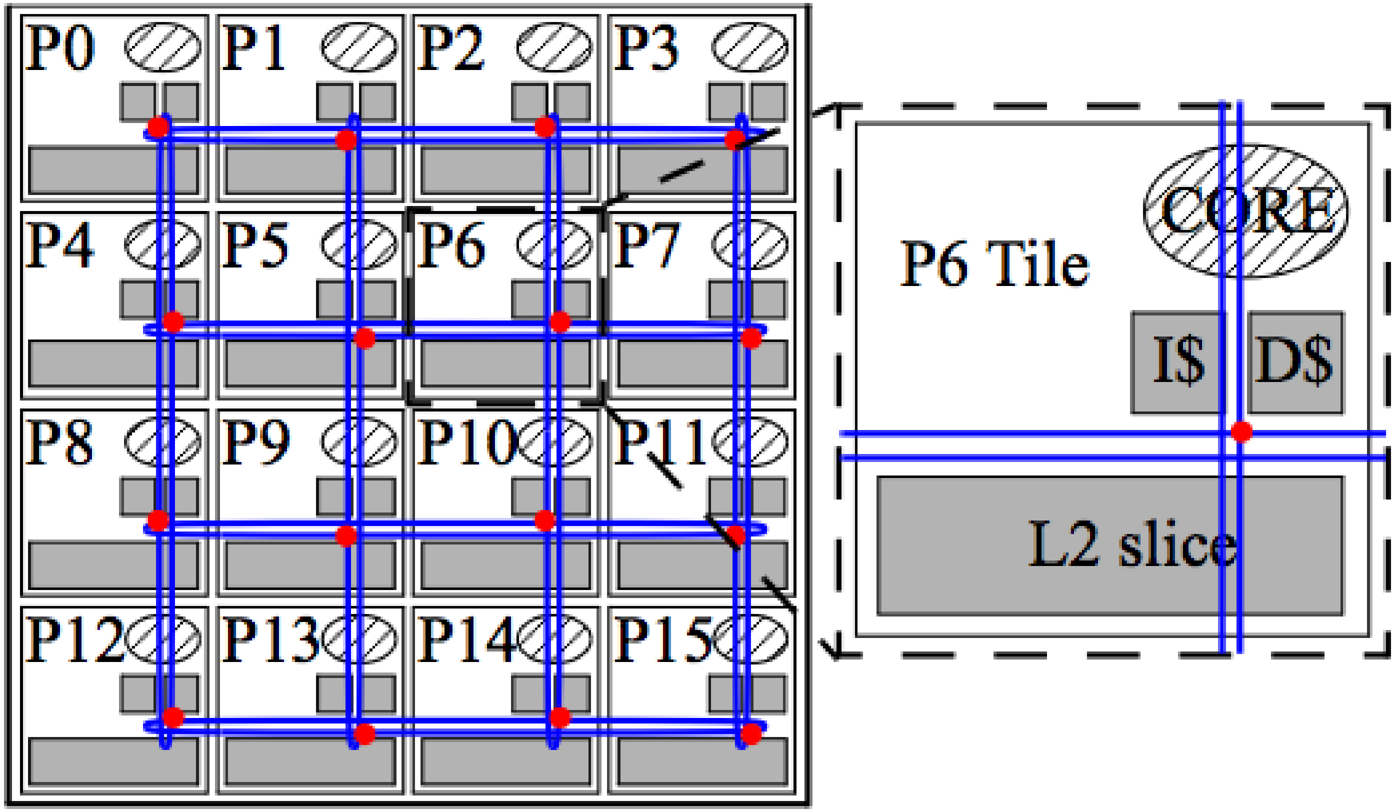
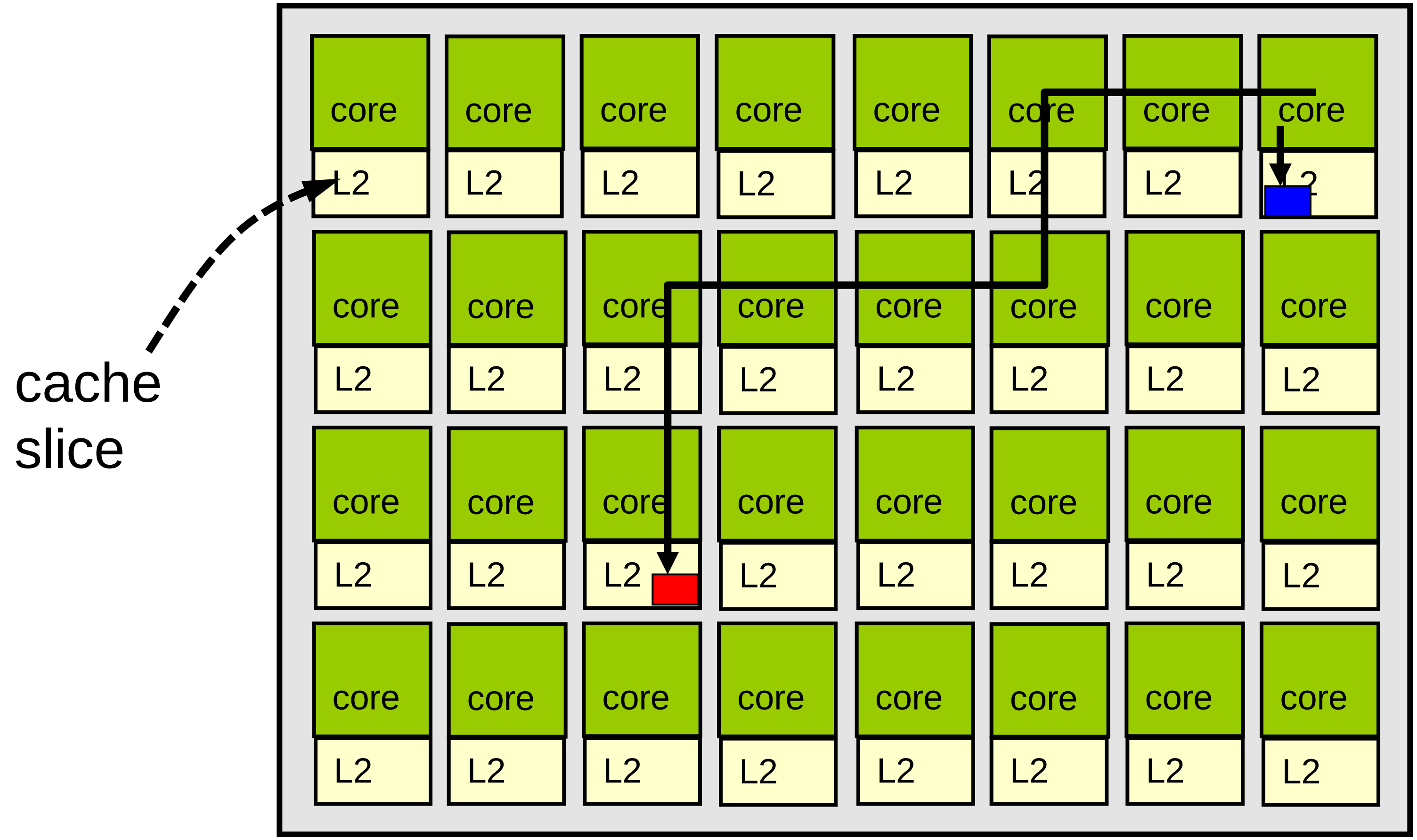


FIGURE 1. Typical tiled architecture. Tiles are interconnected into a 2-D folded torus. Each tile contains a core, L1 instruction and data caches, a shared-L2 cache slice, and a router/switch.

Caching for Parallel Applications



- Data placement determines performance
- Goal: place data on chip close to where they are used

Research Topics

Shared Cache Management: Research Topics

- Scalable partitioning algorithms
  - Distributed caches have different tradeoffs
- Configurable partitioning algorithms
  - Many metrics may need to be optimized at different times or at the same time
  - It is not only about overall performance
- Ability to have high capacity AND high locality (fast access)
- Within vs. across-application prioritization
- Holistic design
  - How to manage caches, NoC, and memory controllers together?
- Cache coherence in shared/private distributed caches