Computer Architecture
Lecture 13: Memory Interference and Quality of Service (II)

Prof. Onur Mutlu
ETH Zürich

Fall 2017

2 November 2017

Summary of Yesterday

- Shared vs. private resources in multi-core systems
- Memory interference and the QoS problem
- Memory scheduling

Agenda for Today

- Memory scheduling wrap-up

- Other approaches to mitigate and control memory interference
  - Source Throttling
  - Data Mapping
  - Thread Scheduling

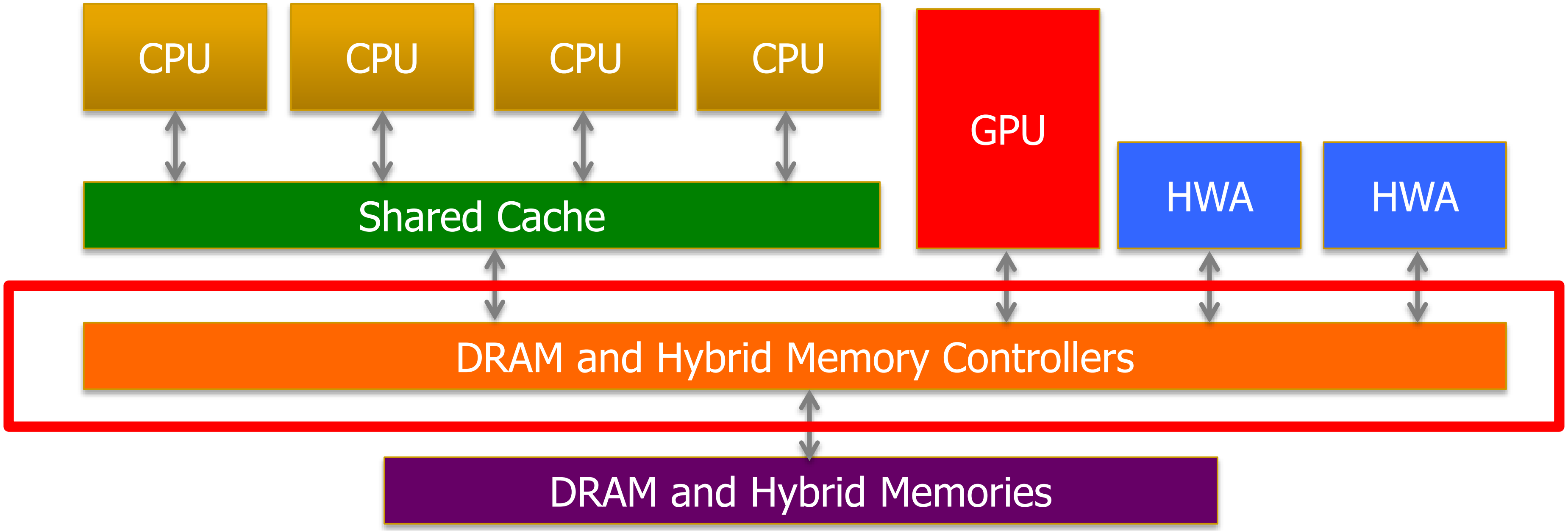
- Multi-Core Cache Management

Quick Summary Papers

- "Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems”
- "The Blacklisting Memory Scheduler: Achieving High Performance and Fairness at Low Cost"
- "Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems”
- "Parallel Application Memory Scheduling”
- "Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning"

Predictable Performance: Strong Memory Service Guarantees

Goal: Predictable Performance in Complex Systems



- Heterogeneous agents: CPUs, GPUs, and HWAs
- Main memory interference between CPUs, GPUs, HWAs

How to allocate resources to heterogeneous agents
to mitigate interference and provide predictable performance?

Strong Memory Service Guarantees

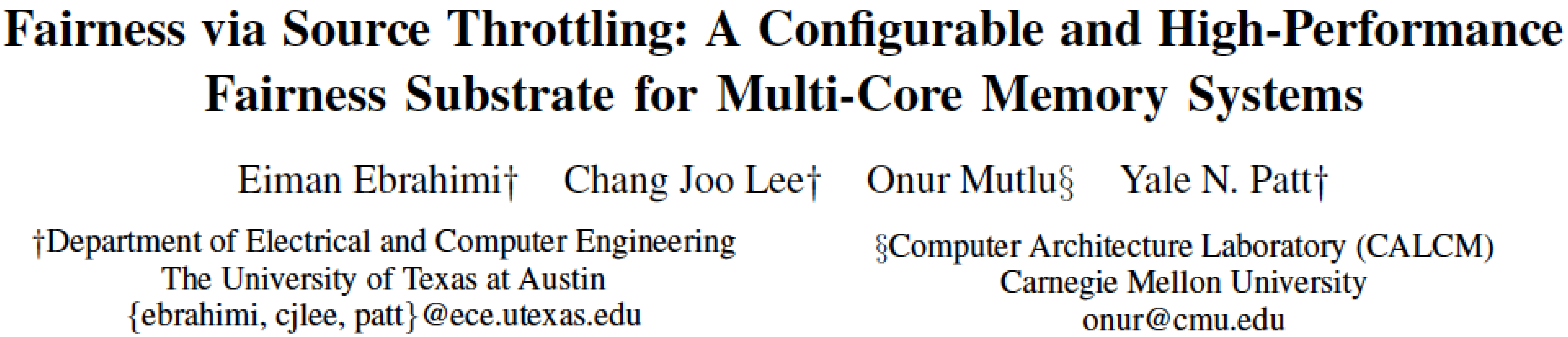
- Goal: Satisfy performance/SLA requirements in the presence of shared main memory, heterogeneous agents, and hybrid memory/storage

- Approach:
  - Develop techniques/models to accurately estimate the performance loss of an application/agent in the presence of resource sharing
  - Develop mechanisms (hardware and software) to enable the resource partitioning/prioritization needed to achieve the required performance levels for all applications
  - All the while providing high system performance

- Subramanian et al., "MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems," HPCA 2013.
- Subramanian et al., "The Application Slowdown Model," MICRO 2015.

Predictable Performance Readings (I)

- Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt, **"Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems"** *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (**ASPLOS**)*, pages 335-346, Pittsburgh, PA, March 2010. Slides (pdf)



Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems

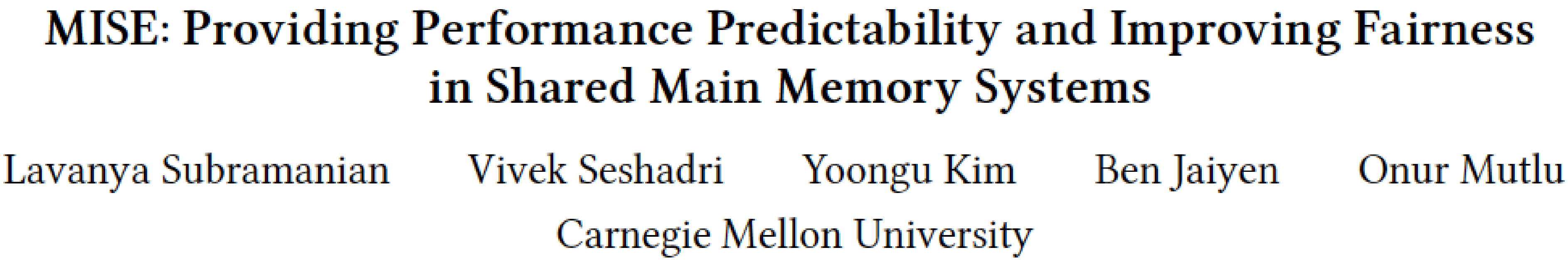
Eiman Ebrahimi[†] Chang Joo Lee[†] Onur Mutlu[§] Yale N. Patt[†]

[†]Department of Electrical and Computer Engineering
The University of Texas at Austin
{ebrahimi, cjlee, patt}@ece.utexas.edu

[§]Computer Architecture Laboratory (CALCM)
Carnegie Mellon University
onur@cmu.edu

Predictable Performance Readings (II)

- Lavanya Subramanian, Vivek Seshadri, Yoongu Kim, Ben Jaiyen, and Onur Mutlu,
"MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems"
Proceedings of the 19th International Symposium on High-Performance Computer Architecture (***HPCA***), Shenzhen, China, February 2013. Slides (pptx)



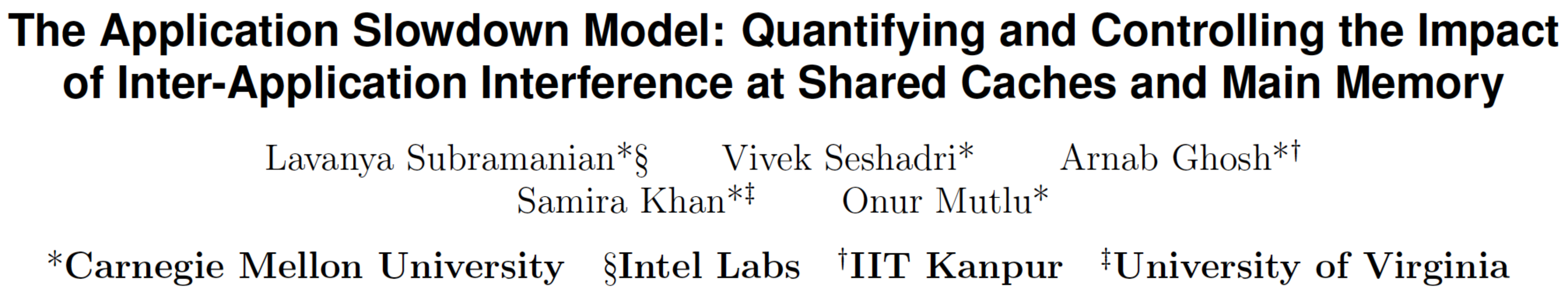
MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems

Lavanya Subramanian Vivek Seshadri Yoongu Kim Ben Jaiyen Onur Mutlu

Carnegie Mellon University

Predictable Performance Readings (III)

- Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, and Onur Mutlu,
"The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-Application Interference at Shared Caches and Main Memory"
*Proceedings of the 48th International Symposium on Microarchitecture (**MICRO**)*, Waikiki, Hawaii, USA, December 2015.
[Slides (pptx) (pdf)] [Lightning Session Slides (pptx) (pdf)] [Poster (pptx) (pdf)]
[Source Code]



The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-Application Interference at Shared Caches and Main Memory

Lavanya Subramanian*§ Vivek Seshadri* Arnab Ghosh*† Samira Khan*‡ Onur Mutlu*

*Carnegie Mellon University §Intel Labs †IIT Kanpur ‡University of Virginia

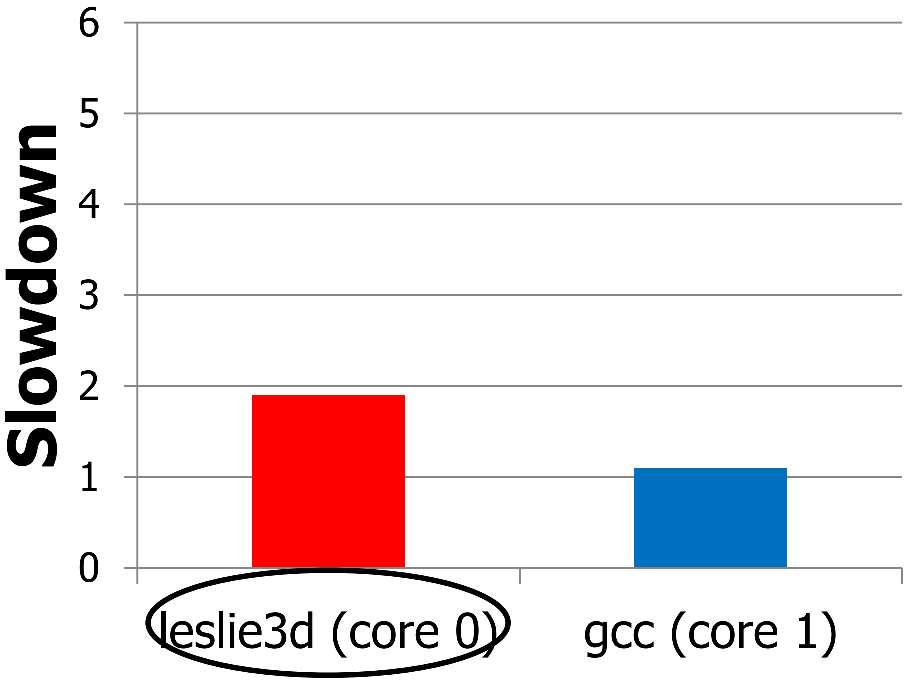
MISE: Providing Performance Predictability in Shared Main Memory Systems

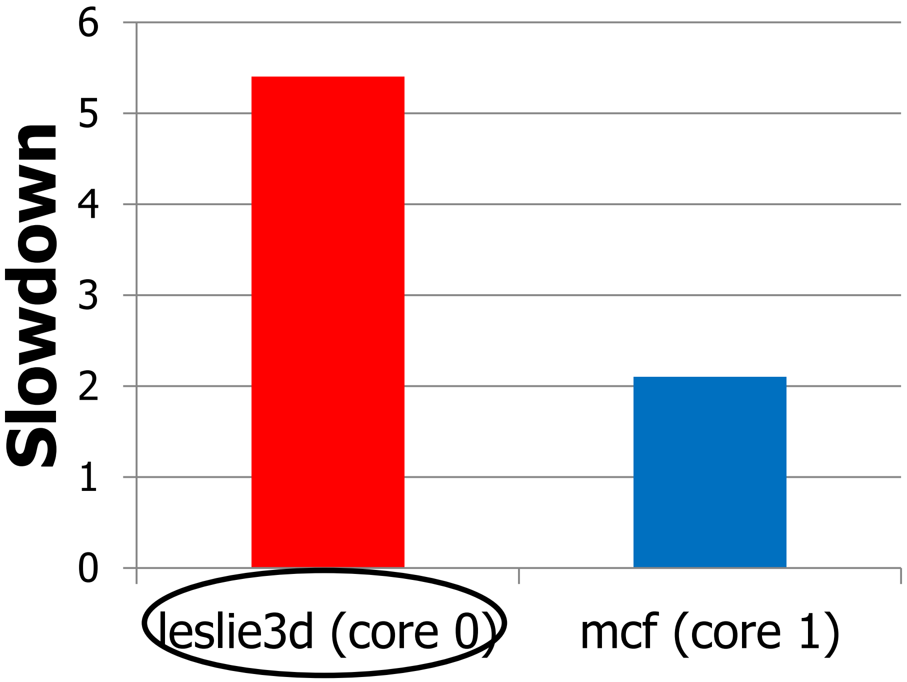
Lavanya Subramanian, Vivek Seshadri,
Yoongu Kim, Ben Jaiyen, Onur Mutlu

SAFARI

Carnegie Mellon

Unpredictable Application Slowdowns





An application’s performance depends on which application it is running with

Need for Predictable Performance

- There is a need for predictable performance
  - When multiple applications share resources
  - Especially if some applications require performance guarantees

Our Goal: Predictable performance in the presence of memory interference

- Example 2: In server systems
  - Different users' jobs consolidated onto the same server
  - Need to provide bounded slowdowns to critical jobs

Outline

1. Estimate Slowdown

2. Control Slowdown

Outline

1. Estimate Slowdown
   - Key Observations
   - Implementation
   - MISE Model: Putting it All Together
   - Evaluating the Model
2. Control Slowdown
   - Providing Soft Slowdown Guarantees
   - Minimizing Maximum Slowdown

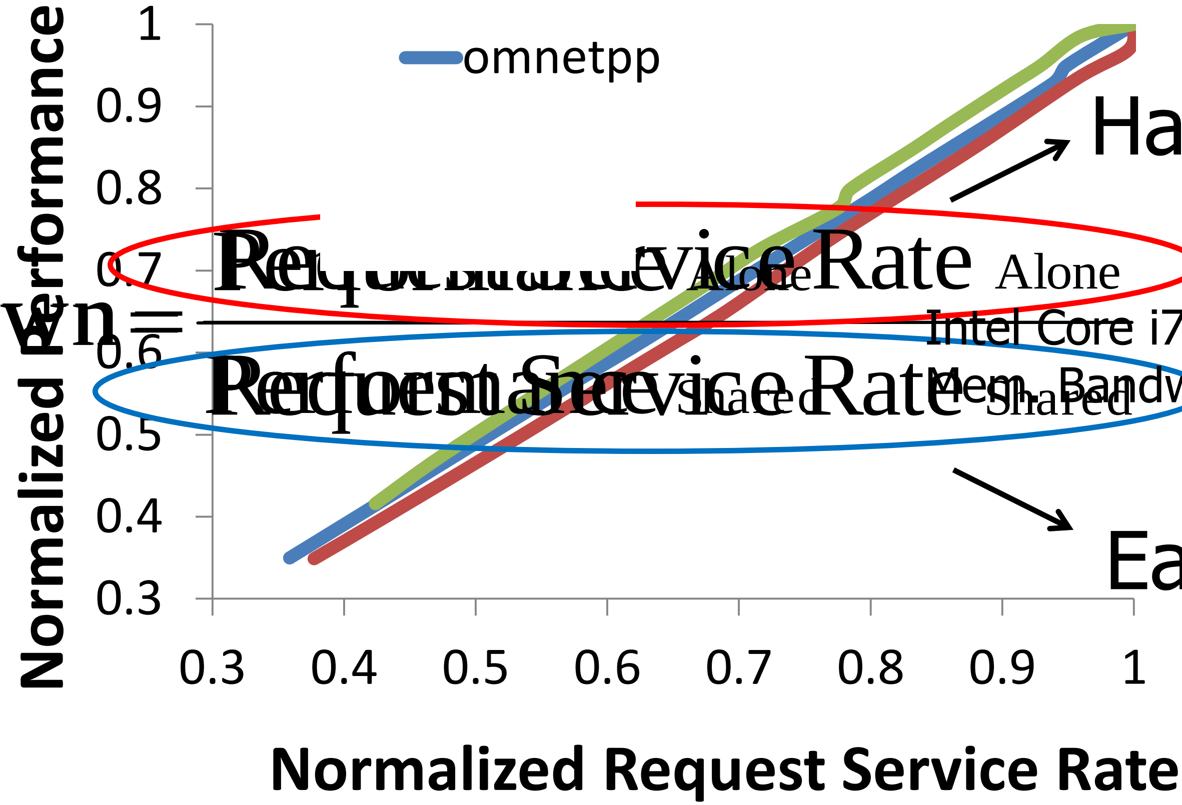
Slowdown: Definition

$$\text{Slowdown} = \frac{\text{Performance}_{\text{Alone}}}{\text{Performance}_{\text{Shared}}}$$

Key Observation 1

For a memory bound application,

Performance ∝ Memory request service rate



$$\text{Slowdown} = \frac{\text{Performance}_{\text{Alone}}}{\text{Performance}_{\text{Shared}}}$$

$$\text{Slowdown} = \frac{\text{Request Service Rate}_{\text{Alone}}}{\text{Request Service Rate}_{\text{Shared}}}$$

Harder

Easy

Intel Core i7, 4 cores

Mem. Bandwidth: 8.5 GB/s

Key Observation 2

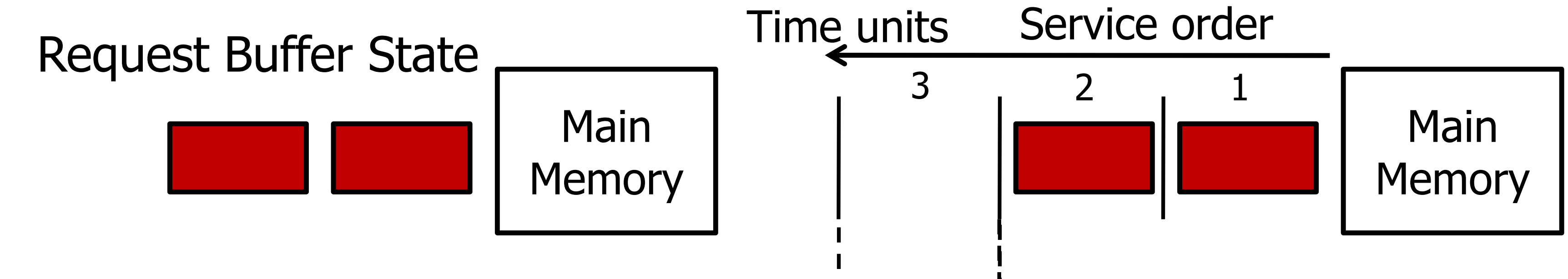
Request Service Rate $_{Alone}$ (RSR$_{Alone}$) of an application can be estimated by giving the application highest priority in accessing memory

Highest priority → Little interference

(almost as if the application were run alone)

Key Observation 2

1. Run alone



Request Buffer State

Main Memory

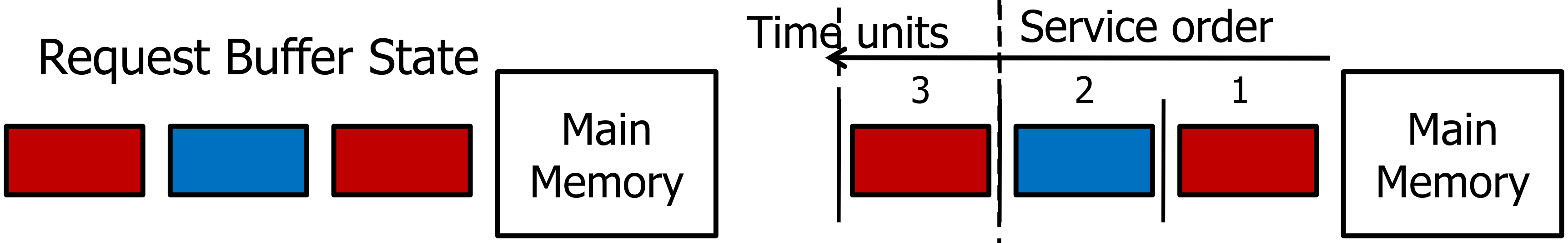
Time units

Service order

3 2 1

Main Memory

2. Run with another application



Request Buffer State

Main Memory

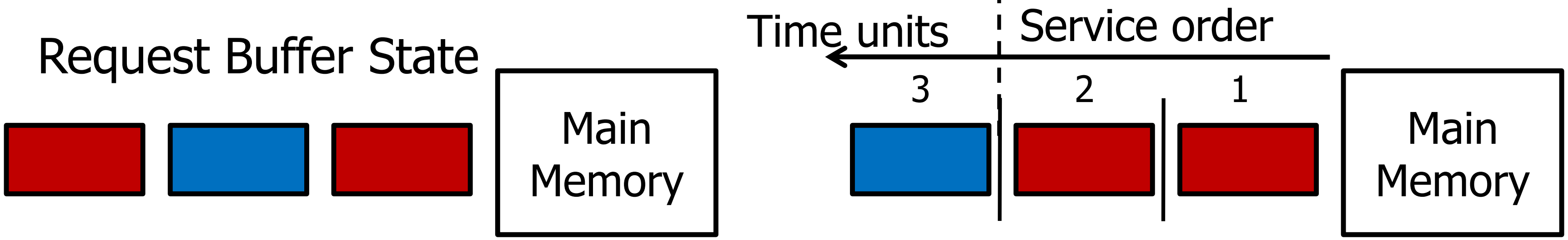
Time units

Service order

3 2 1

Main Memory

3. Run with another application: highest priority



Request Buffer State

Main Memory

Time units

Service order

3 2 1

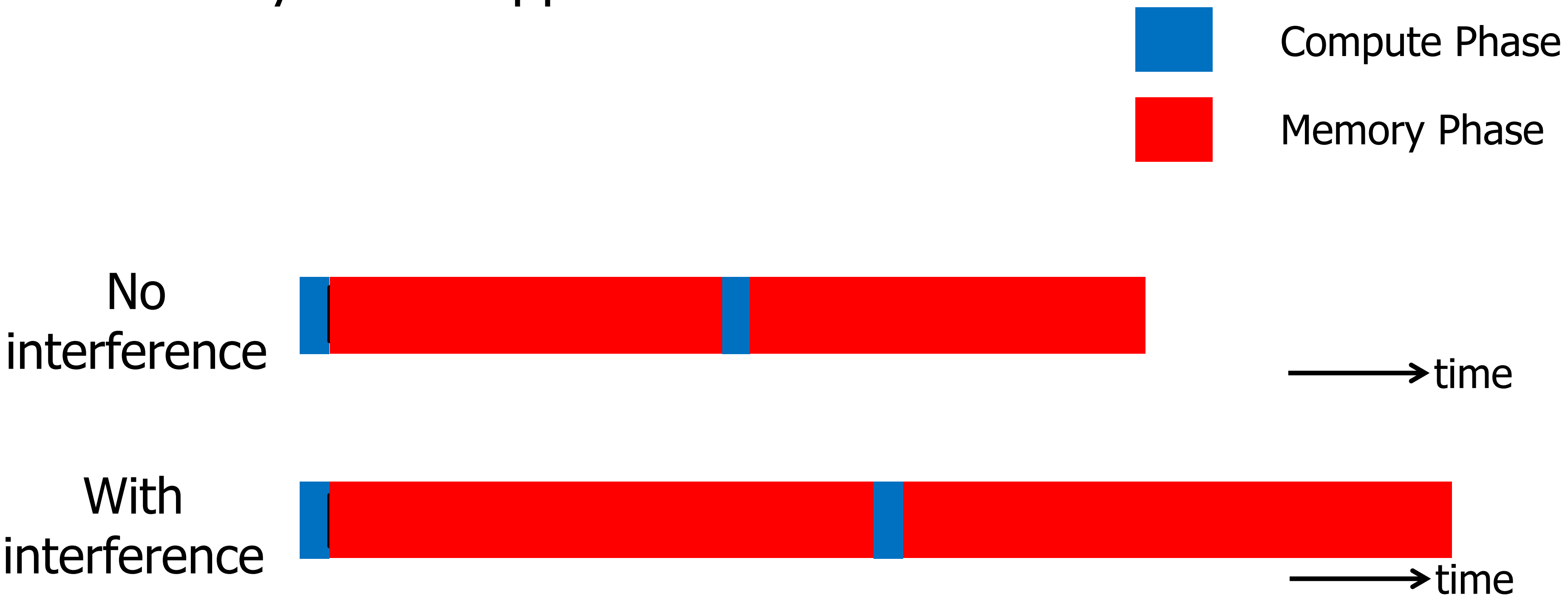
Main Memory

Memory Interference-induced Slowdown Estimation (MISE) model for memory bound applications

$$\text{Slowdown} = \frac{\text{Request Service Rate}_{\text{Alone}}\ (\text{RSR}_{\text{Alone}})}{\text{Request Service Rate}_{\text{Shared}}\ (\text{RSR}_{\text{Shared}})}$$

Key Observation 3

- Memory-bound application



🟦 Compute Phase

🟥 Memory Phase

No interference

time

With interference

time

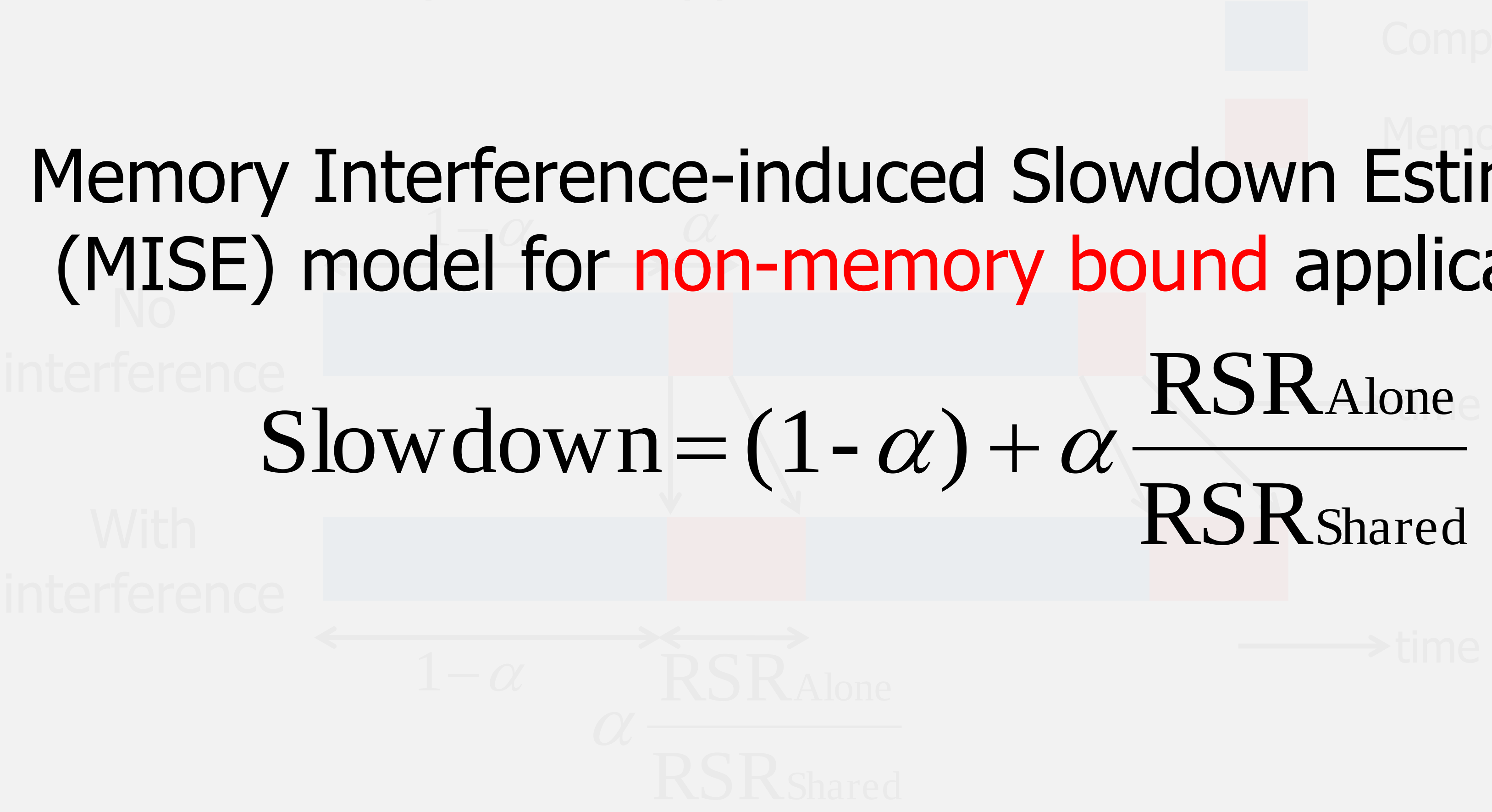
Memory phase slowdown dominates overall slowdown

Key Observation 3

Memory Interference-induced Slowdown Estimation (MISE) model for non-memory bound applications

$$\text{Slowdown} = (1 - \alpha) + \alpha \frac{RSR_{Alone}}{RSR_{Shared}}$$

- Non-memory-bound application

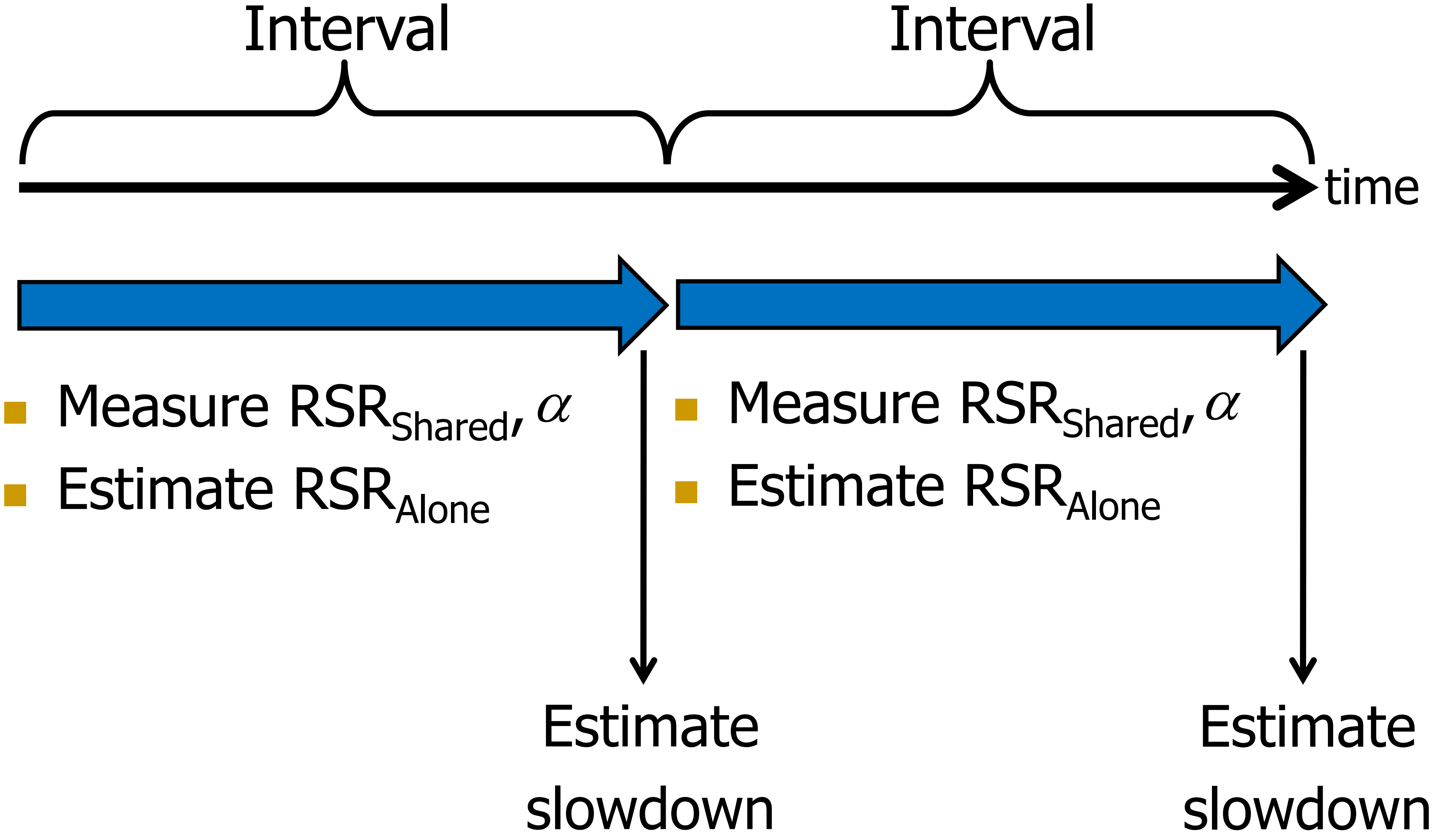


Only memory fraction (α) slows down with interference

Outline

1. Estimate Slowdown
   - ❑ Key Observations
   - ❑ Implementation
   - ❑ MISE Model: Putting it All Together
   - ❑ Evaluating the Model
2. Control Slowdown
   - ❑ Providing Soft Slowdown Guarantees
   - ❑ Minimizing Maximum Slowdown

Interval Based Operation



Interval

Interval

time

- Measure RSR_{Shared}, α
- Estimate RSR_{Alone}

- Measure RSR_{Shared}, α
- Estimate RSR_{Alone}

Estimate slowdown

Estimate slowdown

Measuring $\text{RSR}_{\text{Shared}}$ and α

- Request Service Rate $_{\text{Shared}}$ ($\text{RSR}_{\text{Shared}}$)
  - Per-core counter to track number of requests serviced
  - At the end of each interval, measure

$$\text{RSR}_{\text{Shared}} = \frac{\text{Number of Requests Serviced}}{\text{Interval Length}}$$

- Memory Phase Fraction (a)
  - Count number of stall cycles at the core
  - Compute fraction of cycles stalled for memory

Estimating Request Service Rate $_{\text{Alone}}$ ($\text{RSR}_{\text{Alone}}$)

- Divide each interval into shorter epochs

- At the beginning of each epoch
  - Memory controller randomly picks an application as the highest priority application

Goal: Estimate $\text{RSR}_{\text{Alone}}$

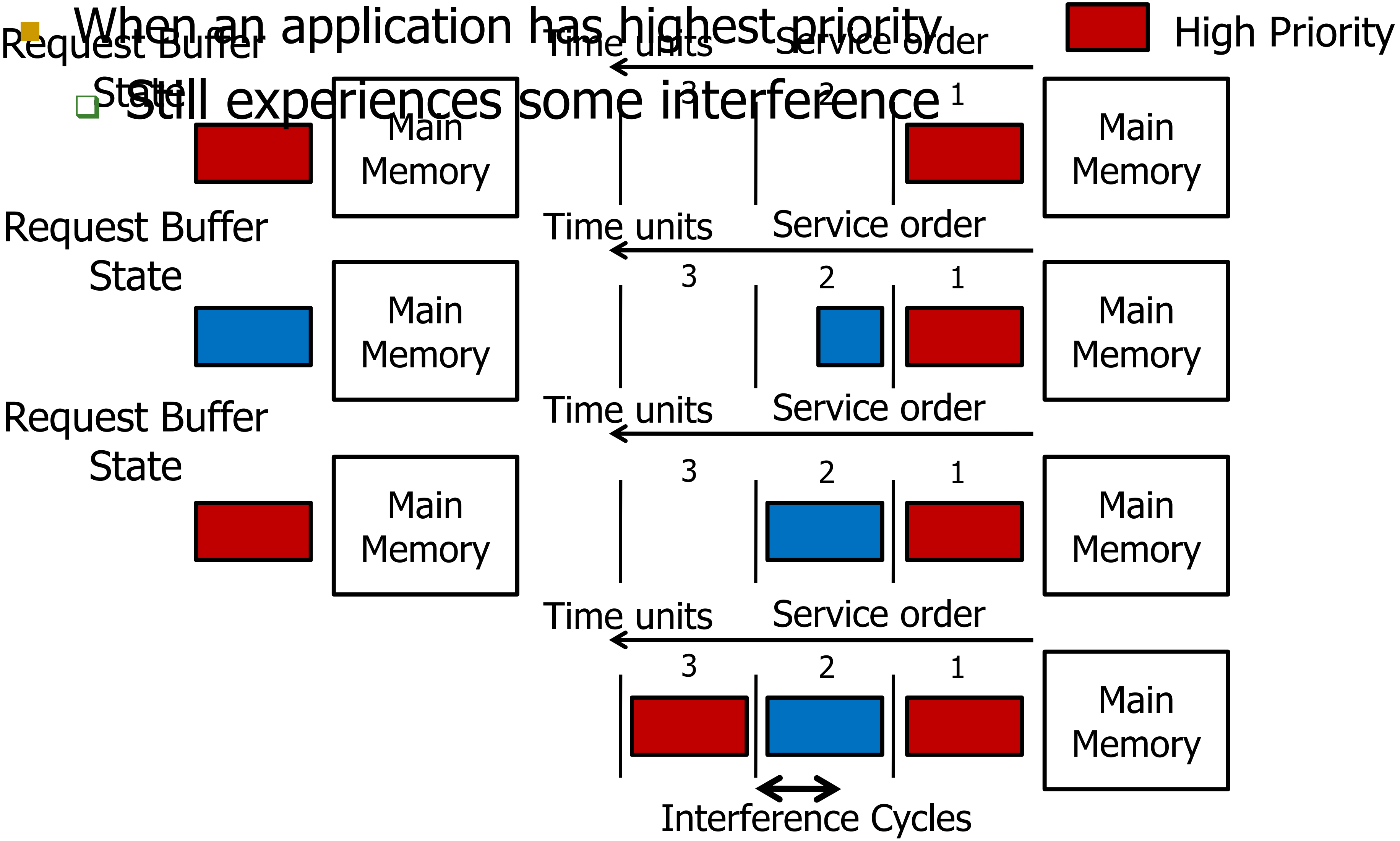
How: Periodically give each application highest priority in accessing memory

- At the end of an interval, for each application, estimate

$$\text{RSR}_{\text{Alone}} = \frac{\text{Number of Requests During High Priority Epochs}}{\text{Number of Cycles Applicatio n Given High Priority}}$$

Inaccuracy in Estimating $\text{RSR}_{\text{Alone}}$

- When an application has highest priority
  - Still experiences some interference



Accounting for Interference in RSR$_{\text{Alone}}$ Estimation

- Solution: Determine and remove interference cycles from RSR$_{\text{Alone}}$ calculation

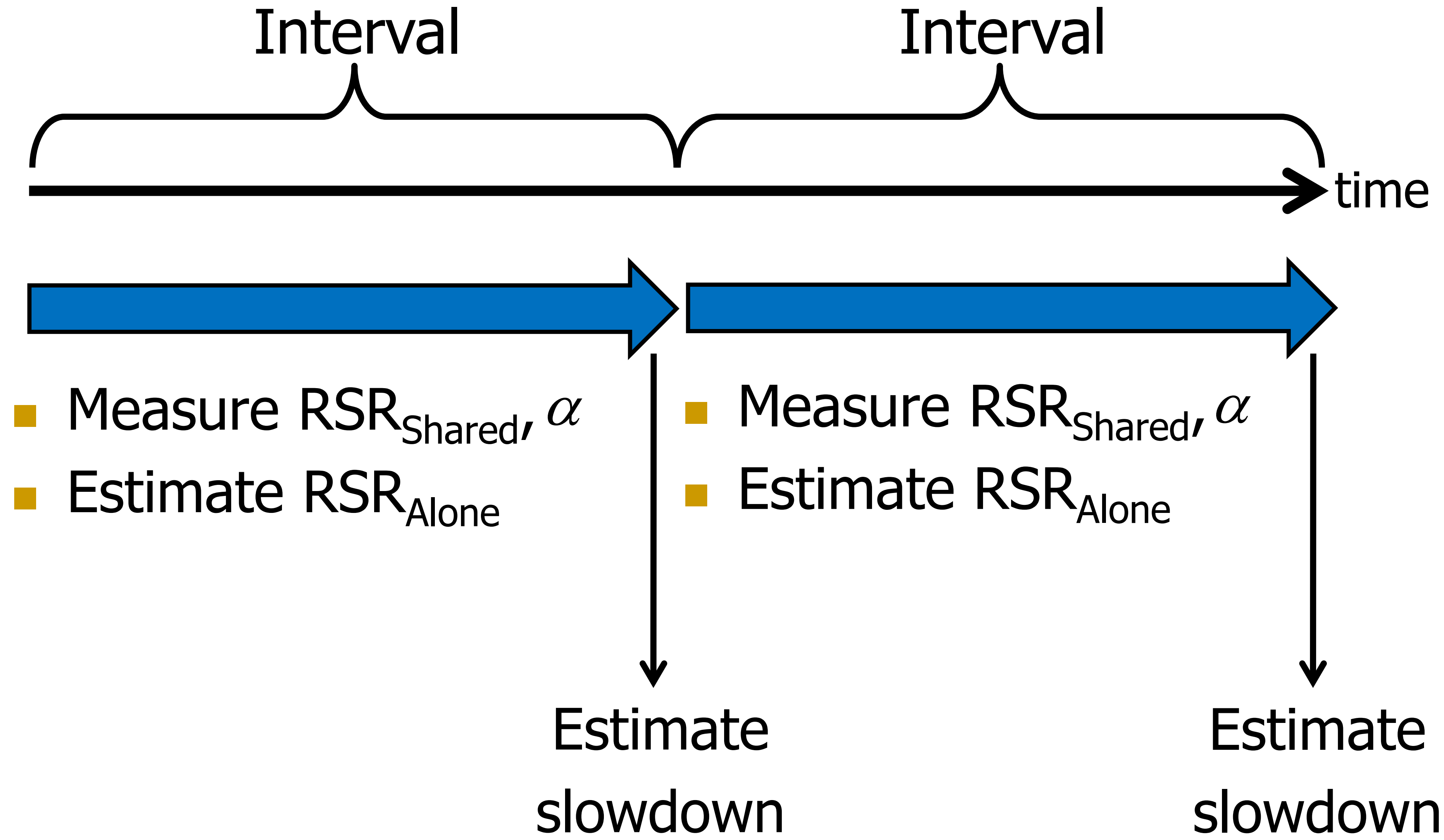
$$\text{RSR}_{\text{Alone}} = \frac{\text{Number of Requests During High Priority Epochs}}{\text{Number of Cycles Application Given High Priority} - \text{Interference Cycles}}$$

- A cycle is an interference cycle if
  - a request from the highest priority application is waiting in the request buffer *and*
  - another application’s request was issued previously

Outline

1. Estimate Slowdown
   - Key Observations
   - Implementation
   - MISE Model: Putting it All Together
   - Evaluating the Model
2. Control Slowdown
   - Providing Soft Slowdown Guarantees
   - Minimizing Maximum Slowdown

MISE Model: Putting it All Together



Interval

Interval

time

- Measure RSR_{Shared}, α
- Estimate RSR_{Alone}

- Measure RSR_{Shared}, α
- Estimate RSR_{Alone}

Estimate slowdown

Estimate slowdown

Outline

1. Estimate Slowdown
   - Key Observations
   - Implementation
   - MISE Model: Putting it All Together
   - Evaluating the Model
2. Control Slowdown
   - Providing Soft Slowdown Guarantees
   - Minimizing Maximum Slowdown

Previous Work on Slowdown Estimation

- Previous work on slowdown estimation
  - **STFM** (Stall Time Fair Memory) Scheduling [Mutlu+, MICRO '07]
  - **FST** (Fairness via Source Throttling) [Ebrahimi+, ASPLOS '10]
  - **Per-thread Cycle Accounting** [Du Bois+, HiPEAC '13]

- Basic Idea:

Hard

$$\text{Slowdown} = \frac{\text{Stall Time}_{\text{Alone}}}{\text{Stall Time}_{\text{Shared}}}$$

Easy

Count number of cycles application receives interference

Two Major Advantages of MISE Over STFM

- Advantage 1:
  - STFM estimates alone performance while an application is receiving interference → Hard
  - MISE estimates alone performance while giving an application the highest priority → Easier

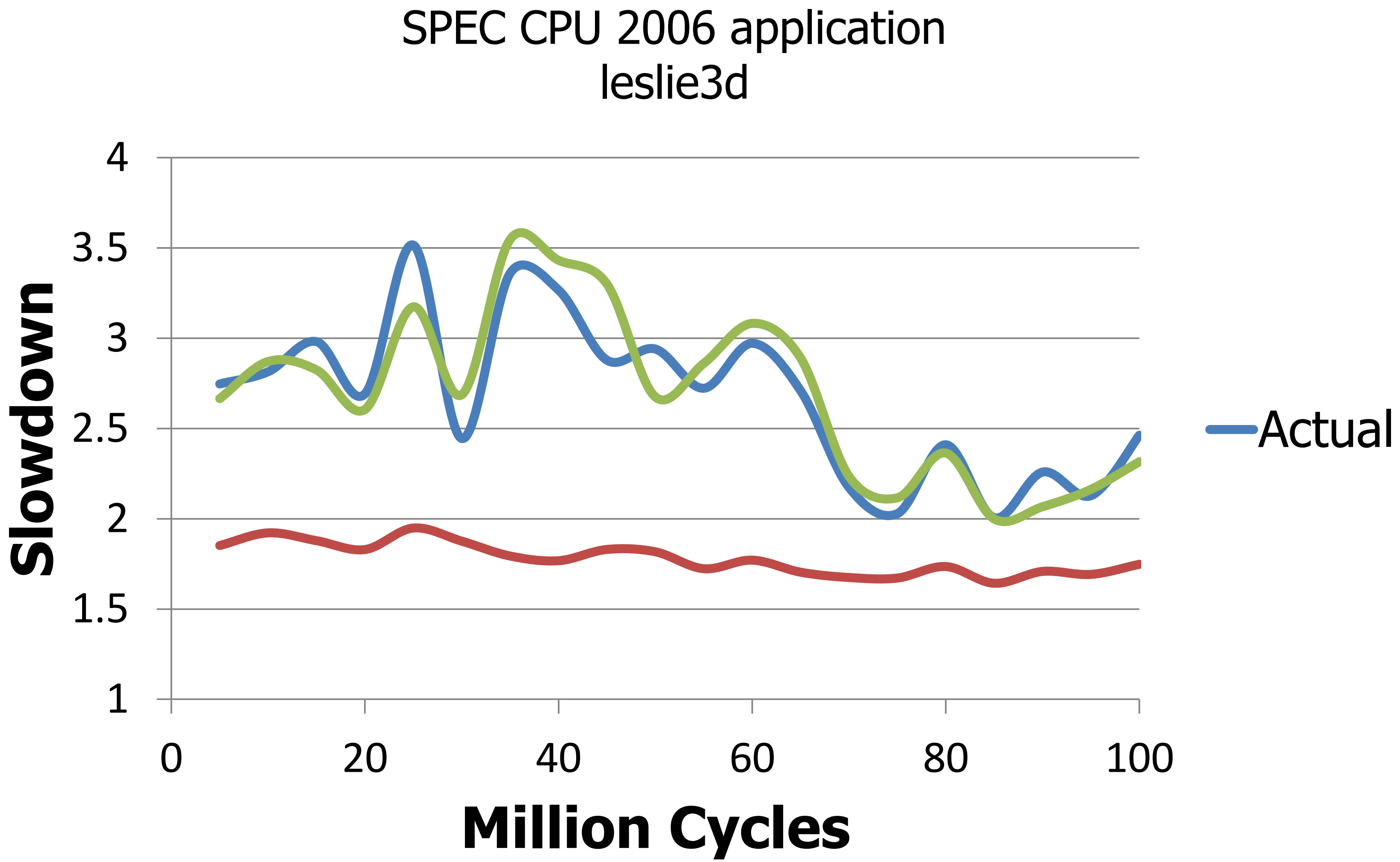
- Advantage 2:
  - STFM does not take into account compute phase for non-memory-bound applications
  - MISE accounts for compute phase → Better accuracy

Methodology

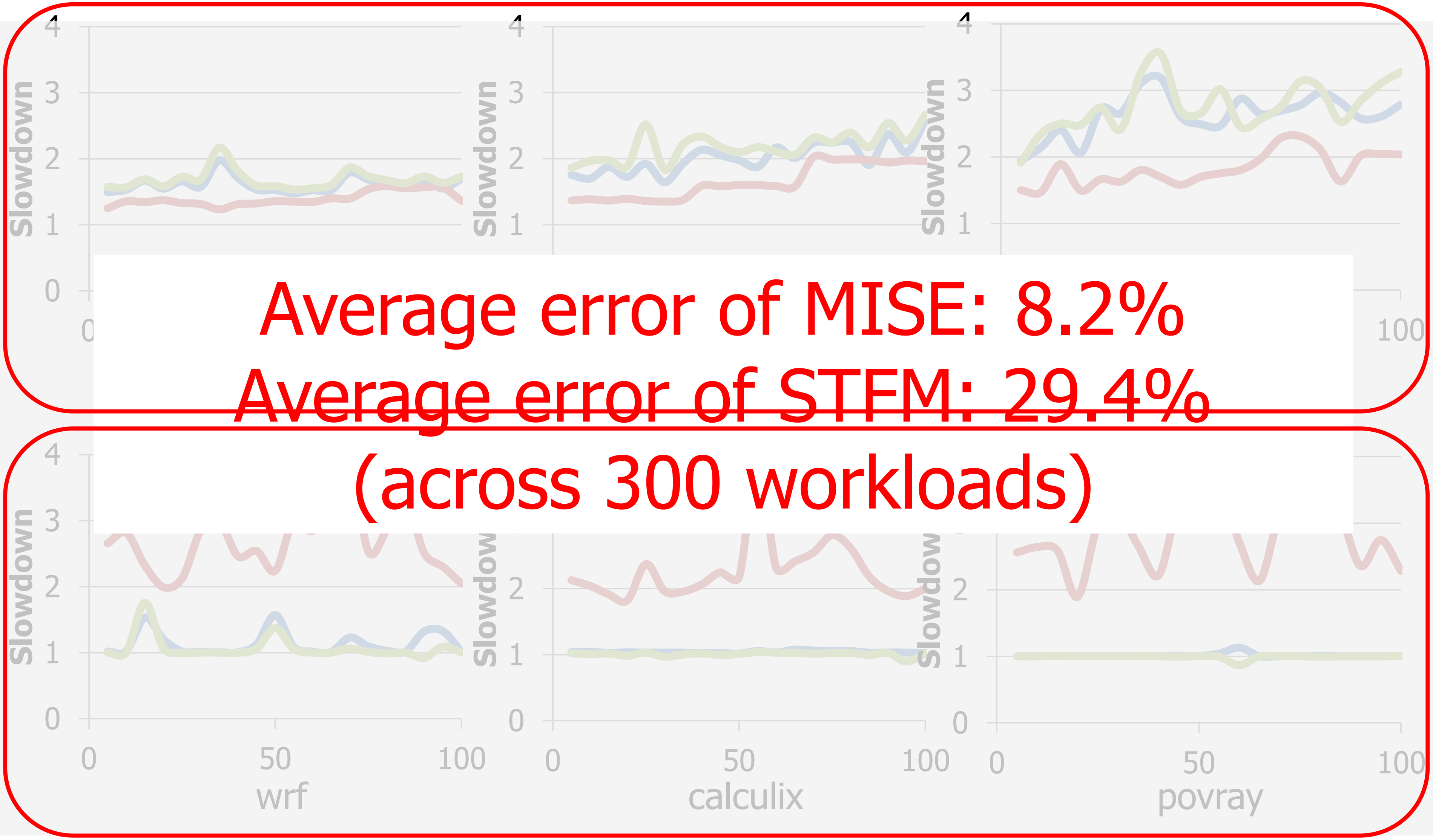
- Configuration of our simulated system
  - 4 cores
  - 1 channel, 8 banks/channel
  - DDR3 1066 DRAM
  - 512 KB private cache/core

- Workloads
  - SPEC CPU2006
  - 300 multi programmed workloads

Quantitative Comparison



Comparison to STFM



Average error of MISE: 8.2%

Average error of STFM: 29.4%

(across 300 workloads)

Outline

1. Estimate Slowdown
   - Key Observations
   - Implementation
   - MISE Model: Putting it All Together
   - Evaluating the Model
2. Control Slowdown
   - Providing Soft Slowdown Guarantees
   - Minimizing Maximum Slowdown

Providing "Soft" Slowdown Guarantees

- Goal
  1. Ensure QoS-critical applications meet a prescribed slowdown bound
  2. Maximize system performance for other applications

- Basic Idea
  - Allocate just enough bandwidth to QoS-critical application
  - Assign remaining bandwidth to other applications

MISE-QoS: Mechanism to Provide Soft QoS

- Assign an initial bandwidth allocation to QoS-critical application
- Estimate slowdown of QoS-critical application using the MISE model
- After every N intervals
  - If slowdown > bound B +/- ε, increase bandwidth allocation
  - If slowdown < bound B +/- ε, decrease bandwidth allocatior
- When slowdown bound not met for N intervals
  - Notify the OS so it can migrate/de-schedule jobs

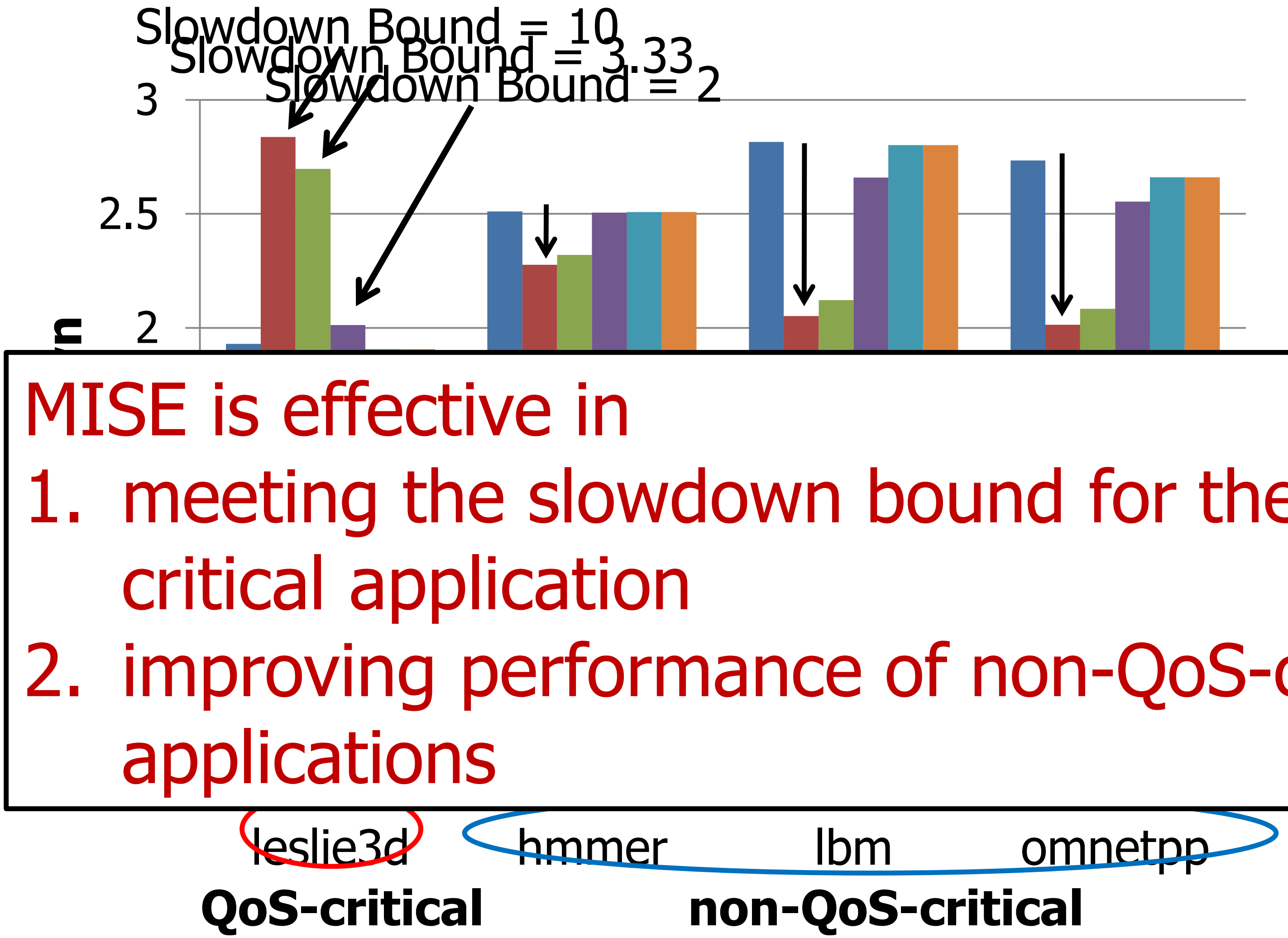
Methodology

- Each application (25 applications in total) considered the QoS-critical application
- Run with 12 sets of co-runners of different memory intensities
- Total of 300 multiprogrammed workloads
- Each workload run with 10 slowdown bound values
- Baseline memory scheduling mechanism
  - Always prioritize QoS-critical application

    [Iyer+, SIGMETRICS 2007]
  - Other applications' requests scheduled in FRFCFS order

    [Zuravleff +, US Patent 1997, Rixner+, ISCA 2000]

A Look at One Workload



QoS-critical

non-QoS-critical

MISE is effective in

1. meeting the slowdown bound for the QoS-critical application
2. improving performance of non-QoS-critical applications

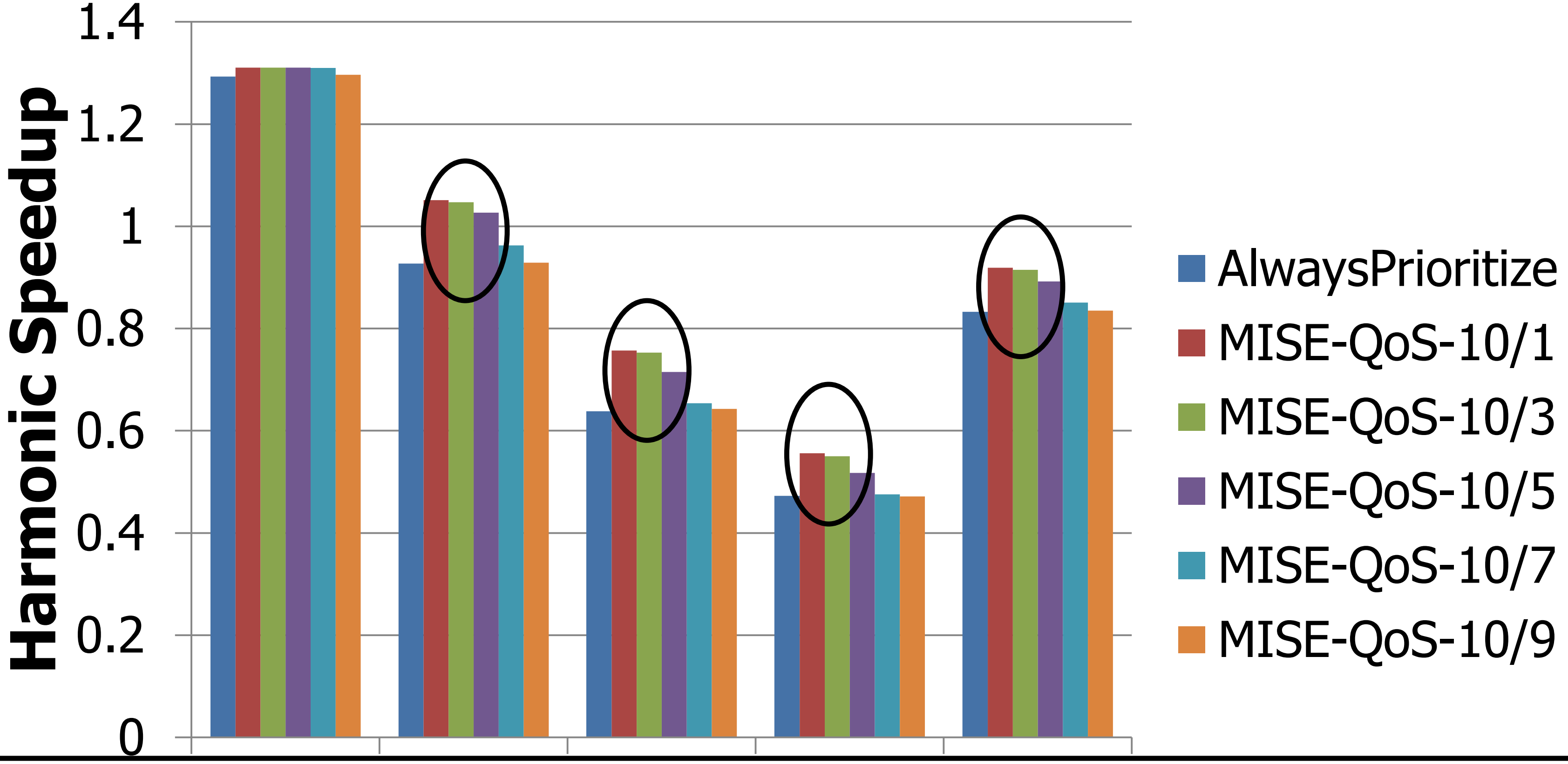
Effectiveness of MISE in Enforcing QoS

Across 3000 data points

	Predicted Met	Predicted Not Met
QoS Bound Met	78.8%	2.1%
QoS Bound Not Met	2.2%	16.9%

MISE-QoS correctly predicts whether or not the bound is met for 95.7% of workloads

Performance of Non-QoS-Critical Applications



When slowdown bound is 10/3
MISE-QoS improves system performance by 10%

Outline

1. Estimate Slowdown
   - Key Observations
   - Implementation
   - MISE Model: Putting it All Together
   - Evaluating the Model
2. Control Slowdown
   - Providing Soft Slowdown Guarantees
   - Minimizing Maximum Slowdown

Other Results in the Paper

- Sensitivity to model parameters
  - Robust across different values of model parameters

- Comparison of STFM and MISE models in enforcing soft slowdown guarantees
  - MISE significantly more effective in enforcing guarantees

- Minimizing maximum slowdown
  - MISE improves fairness across several system configurations

Summary

- Uncontrolled memory interference slows down applications unpredictably
- Goal: Estimate and control slowdowns
- Key contribution
  - MISE: An accurate slowdown estimation model
  - Average error of MISE: 8.2%
- Key Idea
  - Request Service Rate is a proxy for performance
  - Request Service Rate $_{\text{Alone}}$ estimated by giving an application highest priority in accessing memory
- Leverage slowdown estimates to control slowdowns
  - Providing soft slowdown guarantees
  - Minimizing maximum slowdown

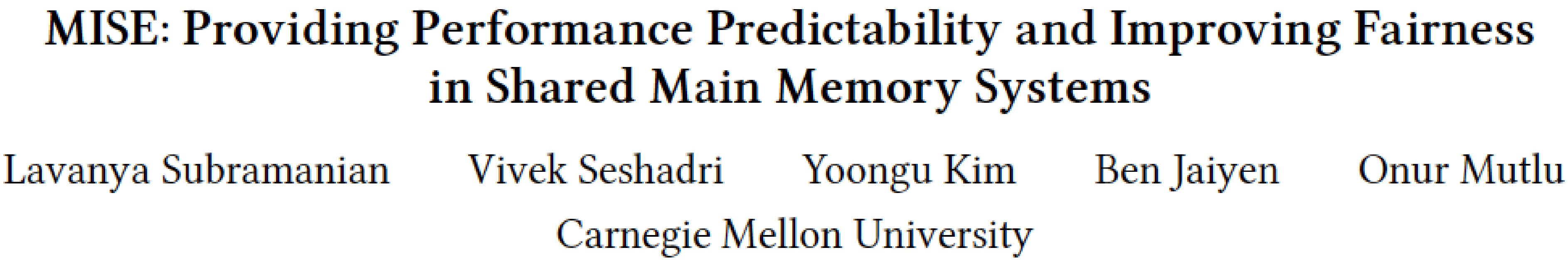
MISE: Pros and Cons

- Upsides:
  - Simple new insight to estimate slowdown
  - Much more accurate slowdown estimations than prior techniques (STFM, FST)
  - Enables a number of QoS mechanisms that can use slowdown estimates to satisfy performance requirements

- Downsides:
  - Slowdown estimation is not perfect - there are still errors
  - Does not take into account caches and other shared resources in slowdown estimation

More on MISE

- Lavanya Subramanian, Vivek Seshadri, Yoongu Kim, Ben Jaiyen, and Onur Mutlu,
"MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems"
*Proceedings of the 19th International Symposium on High-Performance Computer Architecture (**HPCA**)*, Shenzhen, China, February 2013. Slides (pptx)



MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems

Lavanya Subramanian   Vivek Seshadri   Yoongu Kim   Ben Jaiyen   Onur Mutlu

Carnegie Mellon University

Handling Memory Interference In Multithreaded Applications

Eiman Ebrahimi, Rustam Miftakhutdinov, Chris Fallin,
Chang Joo Lee, Onur Mutlu, and Yale N. Patt,
"Parallel Application Memory Scheduling"
Proceedings of the 44th International Symposium on Microarchitecture (***MICRO***),
Porto Alegre, Brazil, December 2011. Slides (pptx)

Multithreaded (Parallel) Applications

- Threads in a multi-threaded application can be inter-dependent
  - As opposed to threads from different applications

- Such threads can synchronize with each other
  - Locks, barriers, pipeline stages, condition variables, semaphores, …

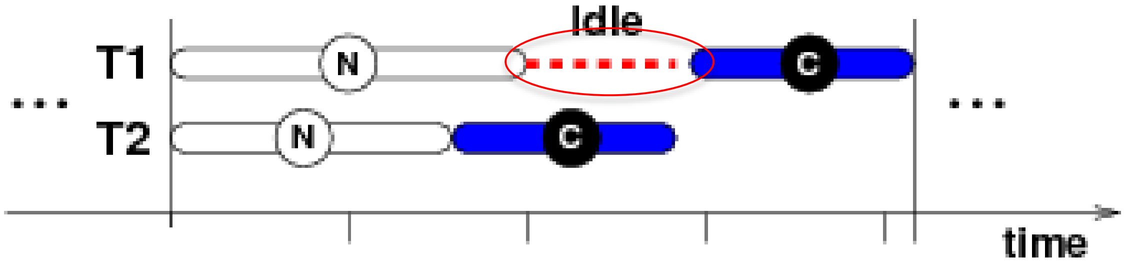
- Some threads can be on the critical path of execution due to synchronization; some threads are not

- Even within a thread, some “code segments” may be on the critical path of execution; some are not

Critical Sections

- Enforce mutually exclusive access to shared data
- Only one thread can be executing it at a time
- Contended critical sections make threads wait → threads causing serialization can be on the critical path

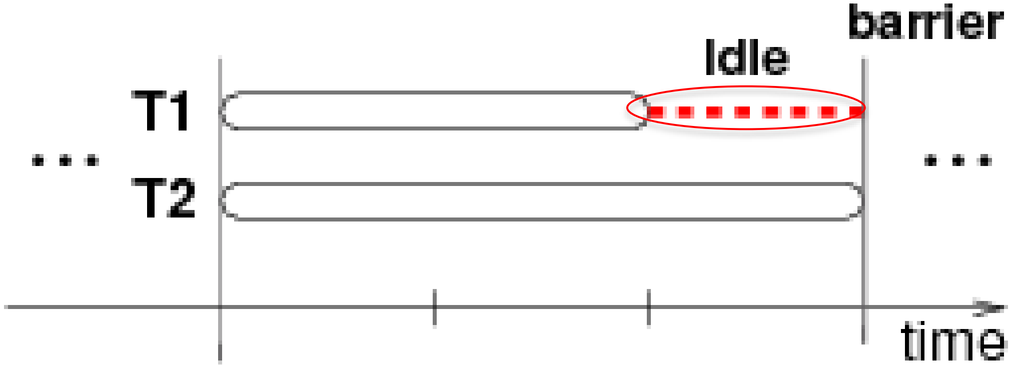
```
Each thread:
    loop {
        Compute                  N
        lock(A)
            Update shared data
        unlock(A)                C
    }
```



Barriers

- Synchronization point
- Threads have to wait until all threads reach the barrier
- Last thread arriving at the barrier is on the critical path

```
Each thread:
    loop1 {
        Compute
    }
    barrier
    loop2 {
        Compute
    }
```



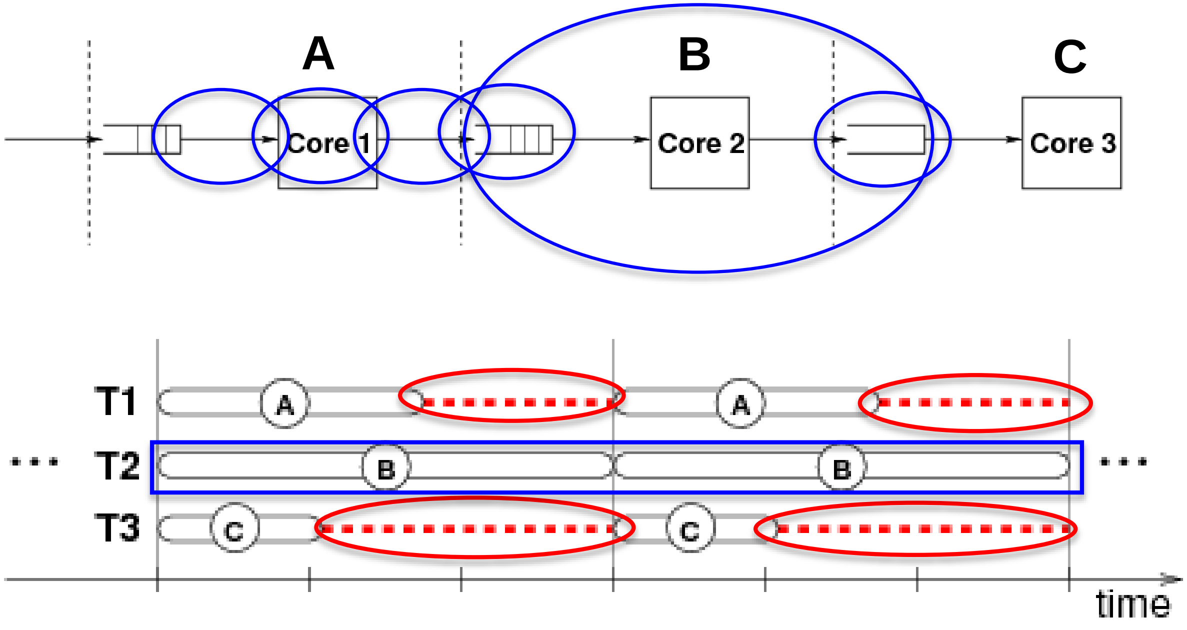
Stages of Pipelined Programs

- Loop iterations are statically divided into code segments called *stages*
- Threads execute stages on different cores
- Thread executing the slowest stage is on the critical path

```
loop {
  Compute1     A

  Compute2     B

  Compute3     C
}
```



Handling Interference in Parallel Applications

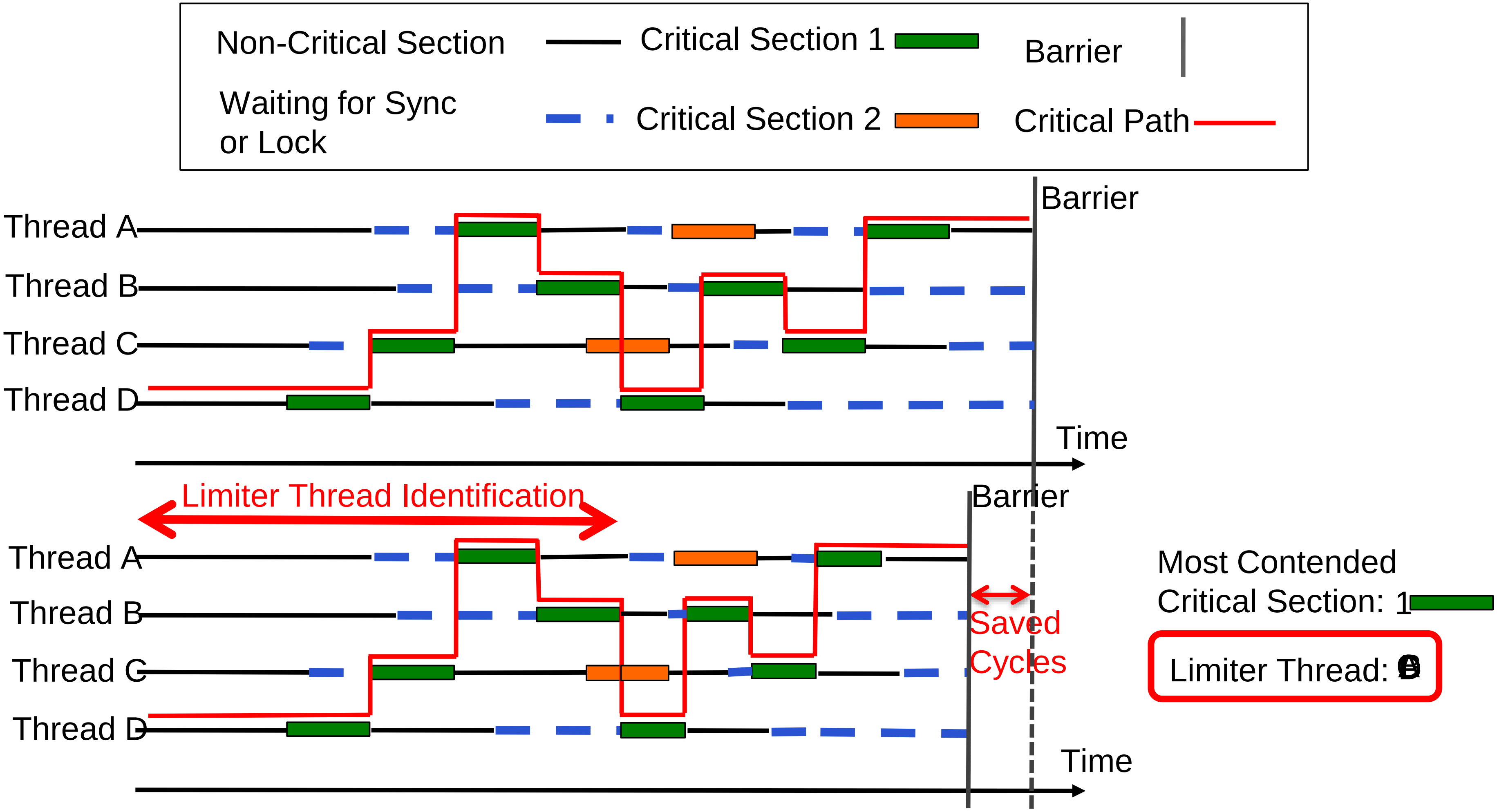
- Threads in a multithreaded application are inter-dependent
- Some threads can be on the critical path of execution due to synchronization; some threads are not
- How do we schedule requests of inter-dependent threads to maximize multithreaded application performance?

- Idea: Estimate limiter threads likely to be on the critical path and prioritize their requests; shuffle priorities of non-limiter threads to reduce memory interference among them [Ebrahimi+, MICRO'11]

- Hardware/software cooperative limiter thread estimation:
  - Thread executing the most contended critical section
  - Thread executing the slowest pipeline stage
  - Thread that is falling behind the most in reaching a barrier

PAMS Micro 2011 Talk

Prioritizing Requests from Limiter Threads



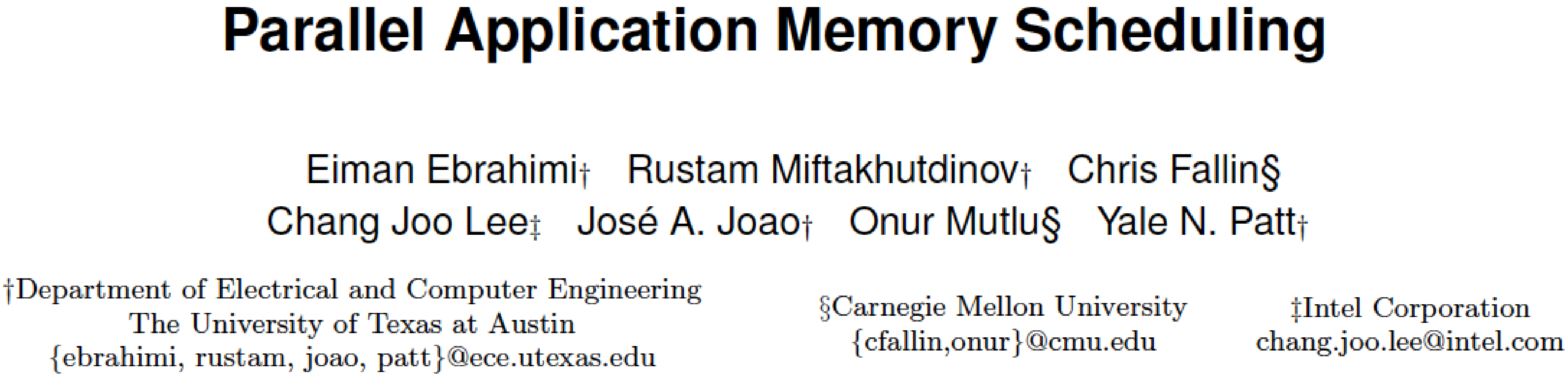
Parallel App Mem Scheduling: Pros and Cons

- Upsides:
  - Improves the performance of multi-threaded applications
  - Provides a mechanism for estimating “limiter threads”
  - Opens a path for slowdown estimation for multi-threaded applications

- Downsides:
  - What if there are multiple multi-threaded applications running together?
  - Limiter thread estimation can become complex

More on PAMS

- Eiman Ebrahimi, Rustam Miftakhutdinov, Chris Fallin, Chang Joo Lee, Onur Mutlu, and Yale N. Patt,
"Parallel Application Memory Scheduling"
*Proceedings of the 44th International Symposium on Microarchitecture (**MICRO**)*, Porto Alegre, Brazil, December 2011. Slides (pptx)



Parallel Application Memory Scheduling

Eiman Ebrahimi[†] Rustam Miftakhutdinov[†] Chris Fallin[§]
Chang Joo Lee[‡] José A. Joao[†] Onur Mutlu[§] Yale N. Patt[†]

[†]Department of Electrical and Computer Engineering
The University of Texas at Austin
{ebrahimi, rustam, joao, patt}@ece.utexas.edu

[§]Carnegie Mellon University
{cfallin,onur}@cmu.edu

[‡]Intel Corporation
chang.joo.lee@intel.com

Other Ways of Handling Memory Interference

Fundamental Interference Control Techniques

- Goal: to reduce/control inter-thread memory interference

1. Prioritization or request scheduling

2. Data mapping to banks/channels/ranks

3. Core/source throttling

4. Application/thread scheduling

Designing QoS-Aware Memory Systems: Approaches

- Smart resources: Design each shared resource to have a configurable interference control/reduction mechanism
  - QoS-aware memory controllers
  - QoS-aware interconnects
  - QoS-aware caches

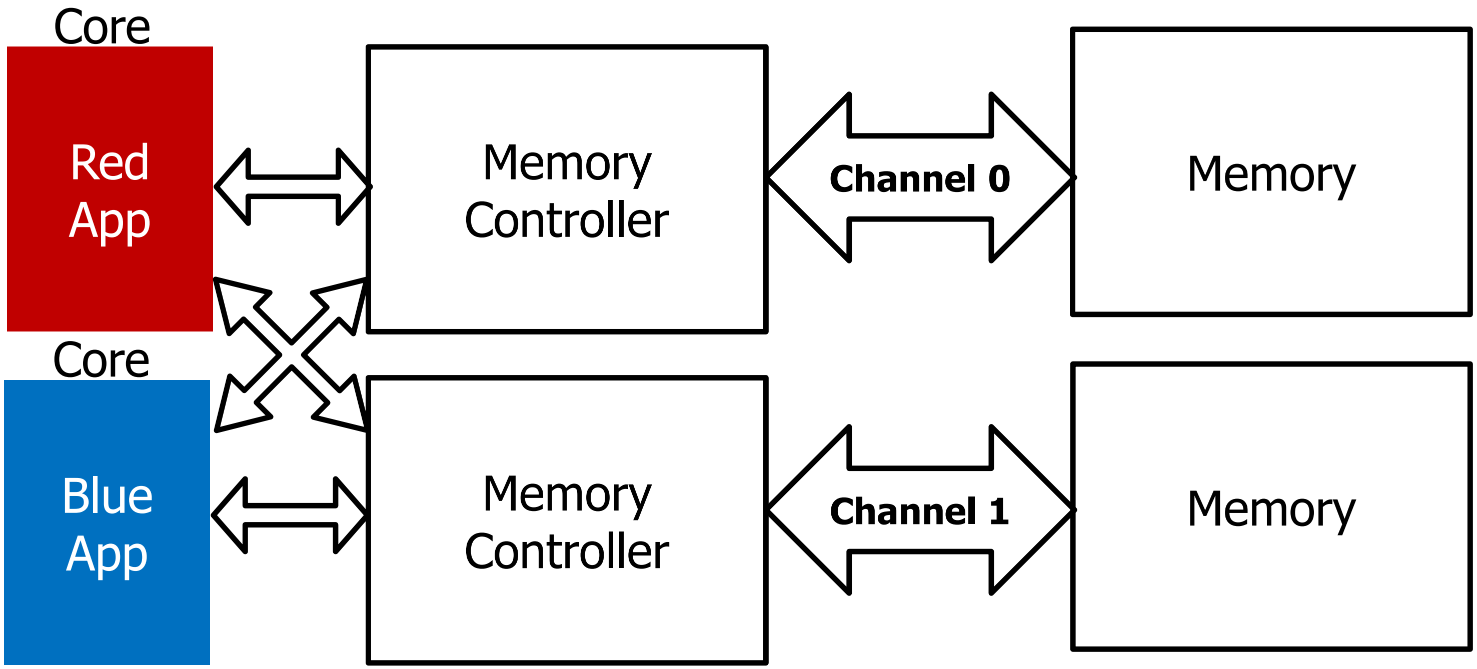
- Dumb resources: Keep each resource free-for-all, but reduce/control interference by injection control or data mapping
  - Source throttling to control access to memory system
  - QoS-aware data mapping to memory controllers
  - QoS-aware thread scheduling to cores

Memory Channel Partitioning

Sai Prashanth Muralidhara, Lavanya Subramanian, Onur Mutlu, Mahmut Kandemir, and Thomas Moscibroda,
"Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning"
44th International Symposium on Microarchitecture (***MICRO***),
Porto Alegre, Brazil, December 2011. Slides (pptx)

MCP Micro 2011 Talk

Observation: Modern Systems Have Multiple Channels

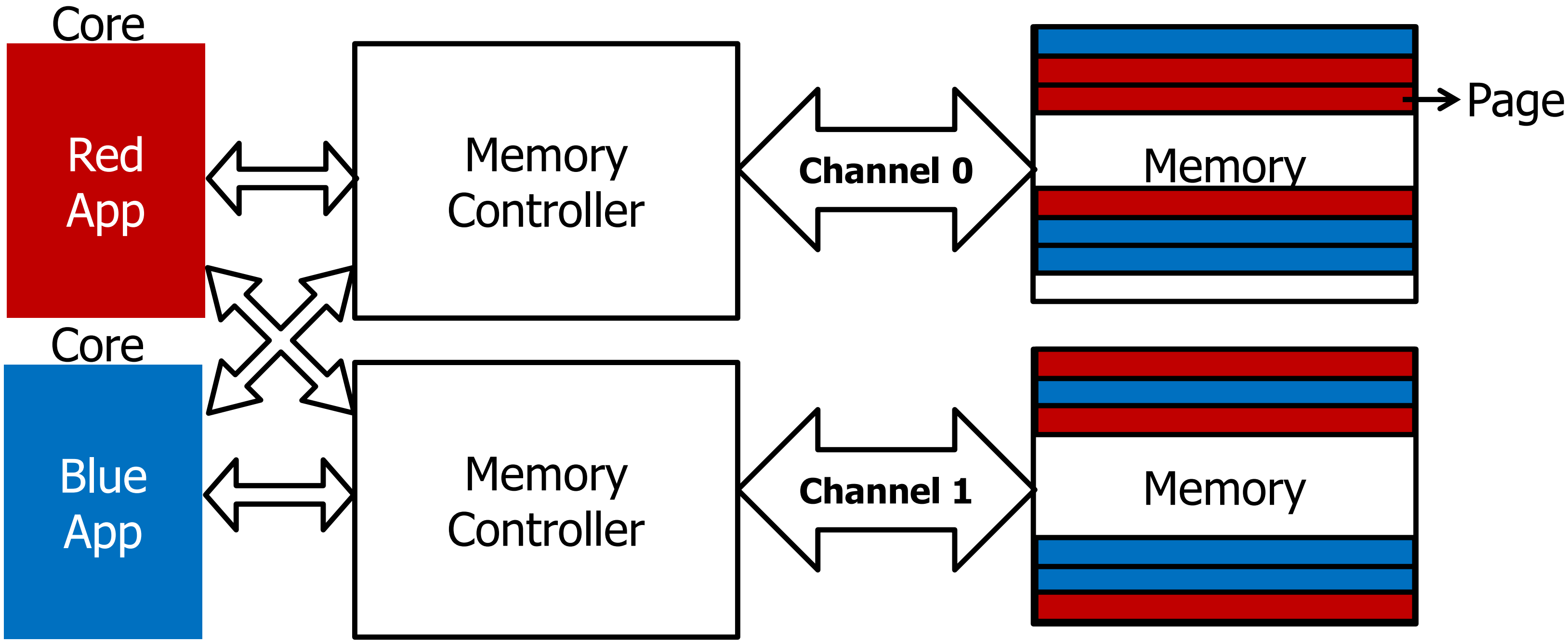


A new degree of freedom

Mapping data across multiple channels

Muralidhara et al., "Memory Channel Partitioning," MICRO'11.

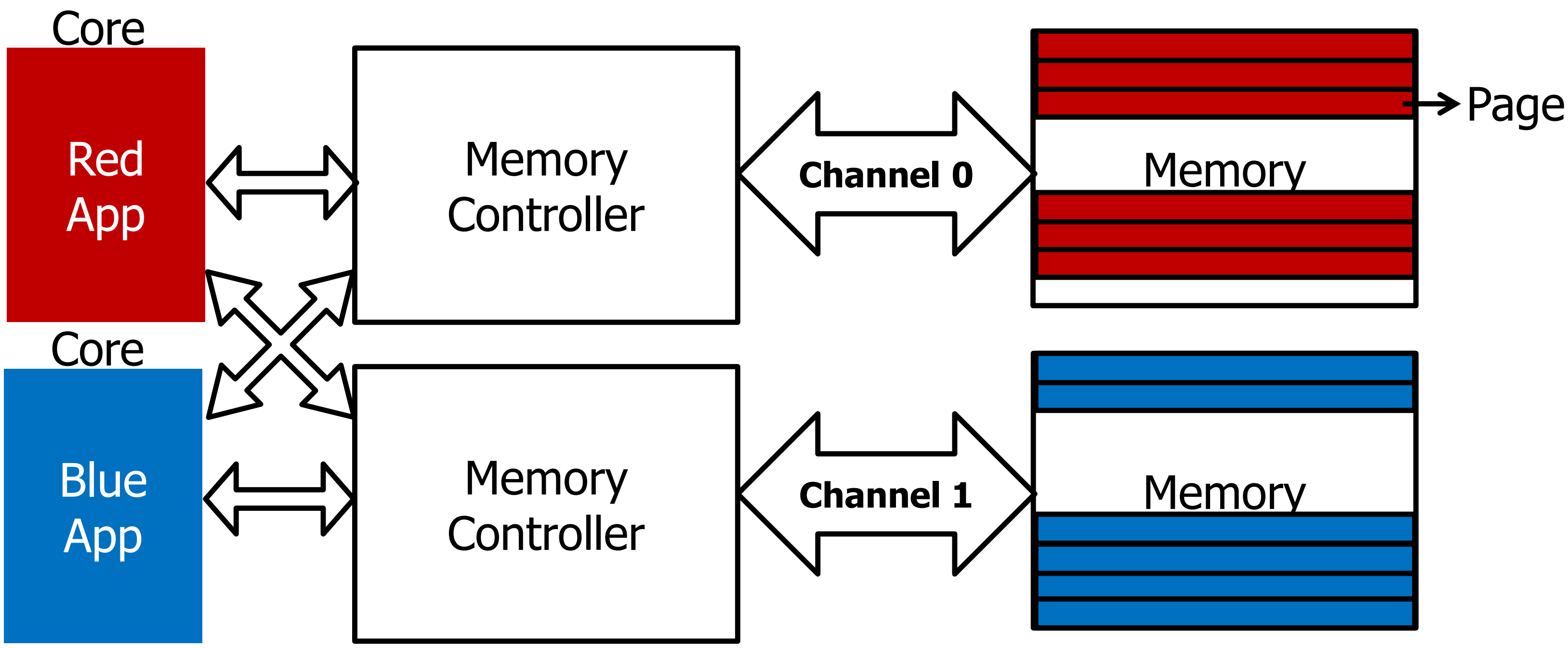
Data Mapping in Current Systems



Causes interference between applications' requests

Muralidhara et al., "Memory Channel Partitioning," MICRO'11.

Partitioning Channels Between Applications



Eliminates interference between applications' requests

Muralidhara et al., "Memory Channel Partitioning," MICRO'11.

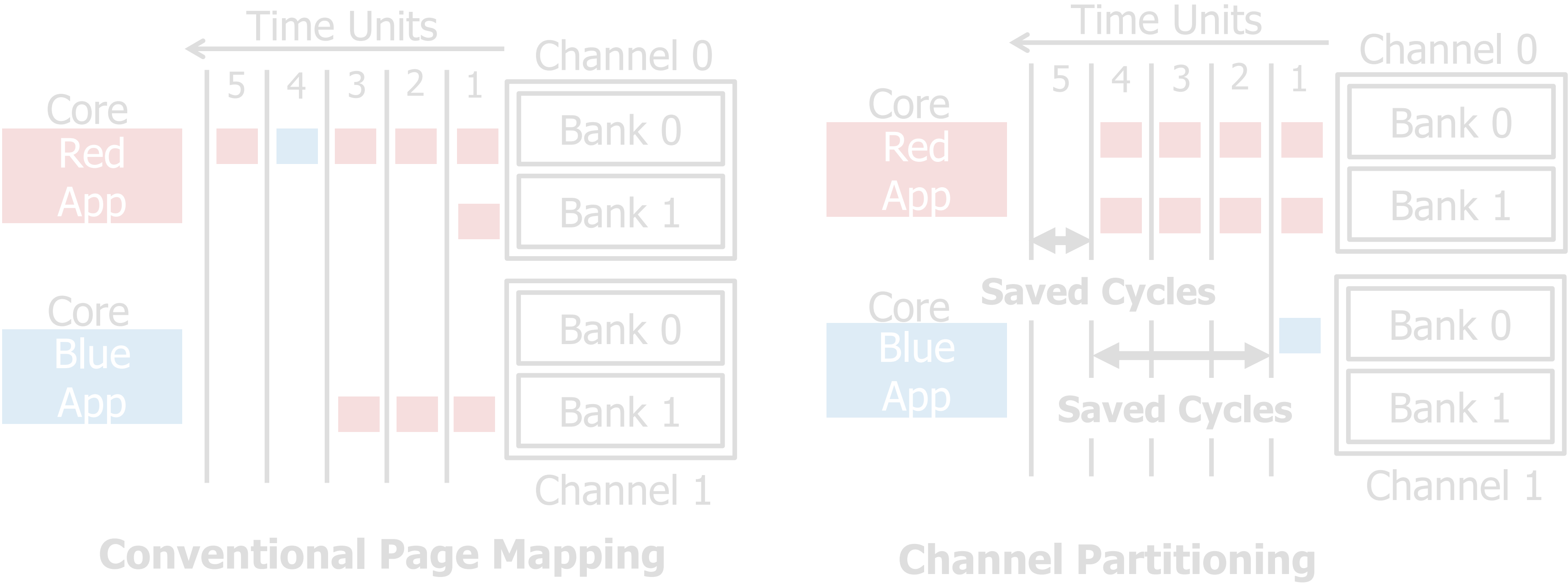
Overview: Memory Channel Partitioning (MCP)

- Goal
  - Eliminate harmful interference between applications
- Basic Idea
  - Map the data of badly-interfering applications to different channels
- Key Principles
  - Separate low and high memory-intensity applications
  - Separate low and high row-buffer locality applications

Muralidhara et al., "Memory Channel Partitioning," MICRO'11.

Key Insight 1: Separate by Memory Intensity

High memory-intensity applications interfere with low memory-intensity applications in shared memory channels



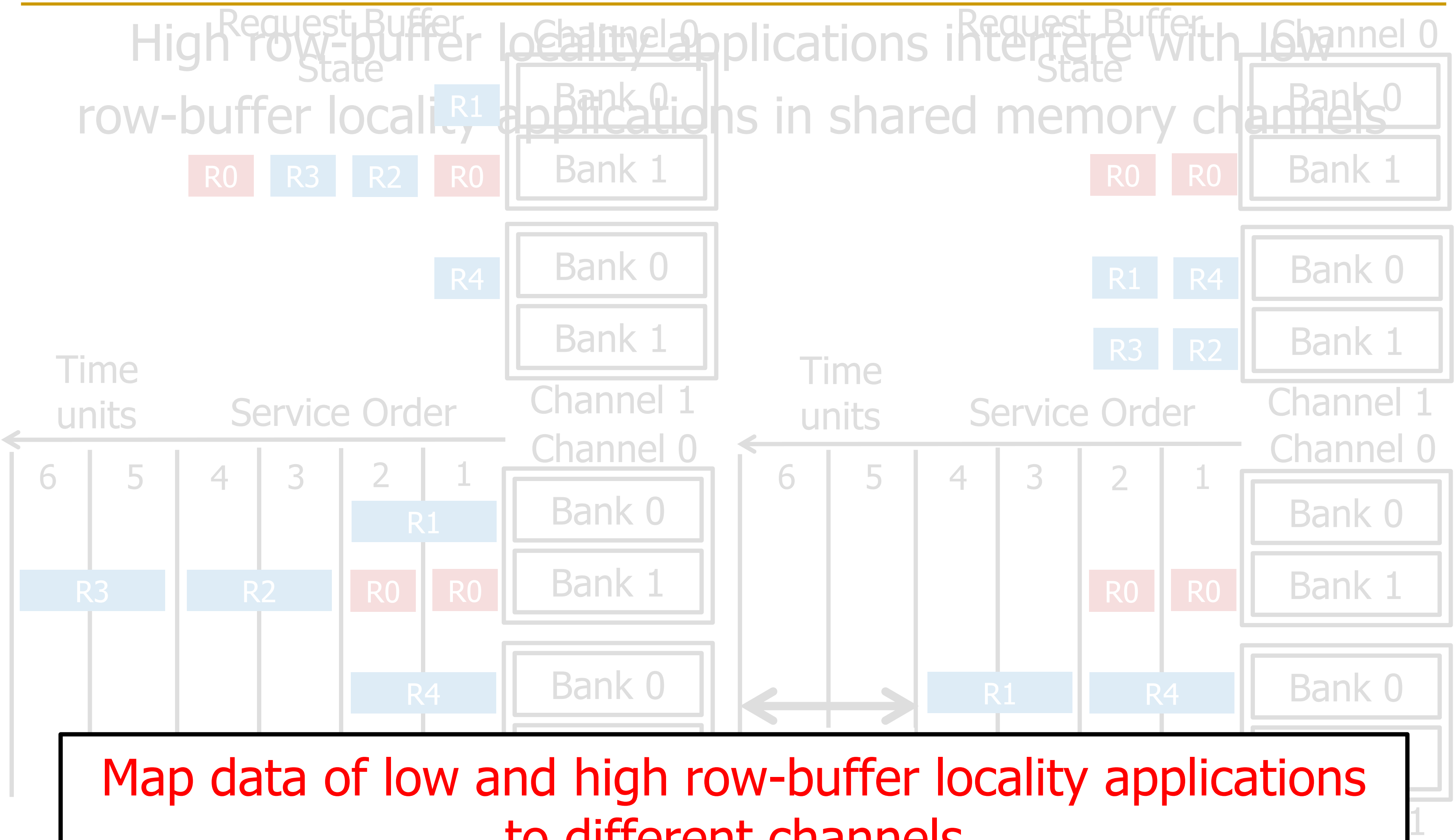
Conventional Page Mapping

Channel Partitioning

Map data of low and high memory-intensity applications to different channels

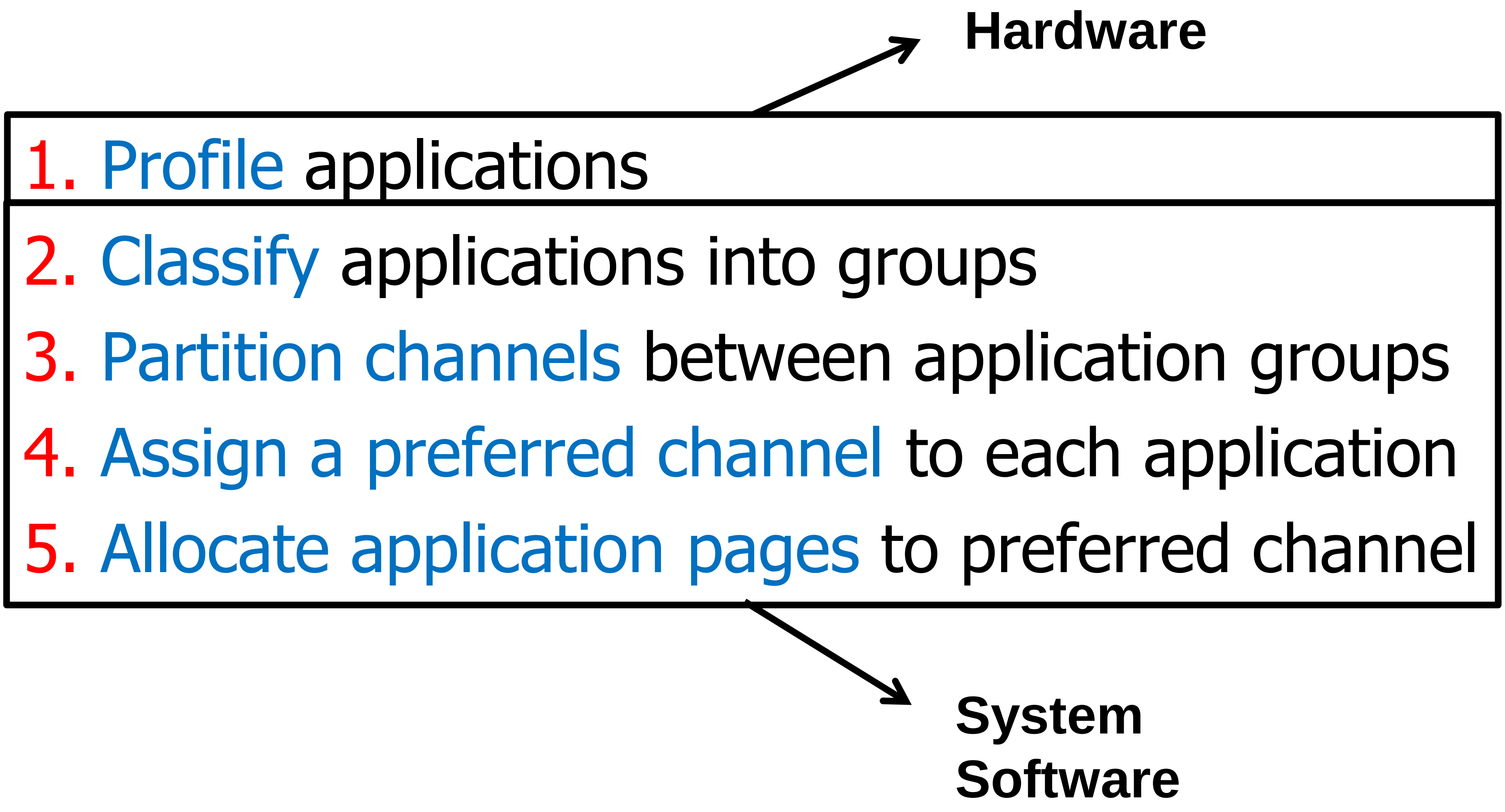
Key Insight 2: Separate by Row-Buffer Locality

High row-buffer locality applications interfere with low row-buffer locality applications in shared memory channels



Map data of low and high row-buffer locality applications to different channels

Memory Channel Partitioning (MCP) Mechanism



Hardware

1. Profile applications

2. Classify applications into groups

3. Partition channels between application groups

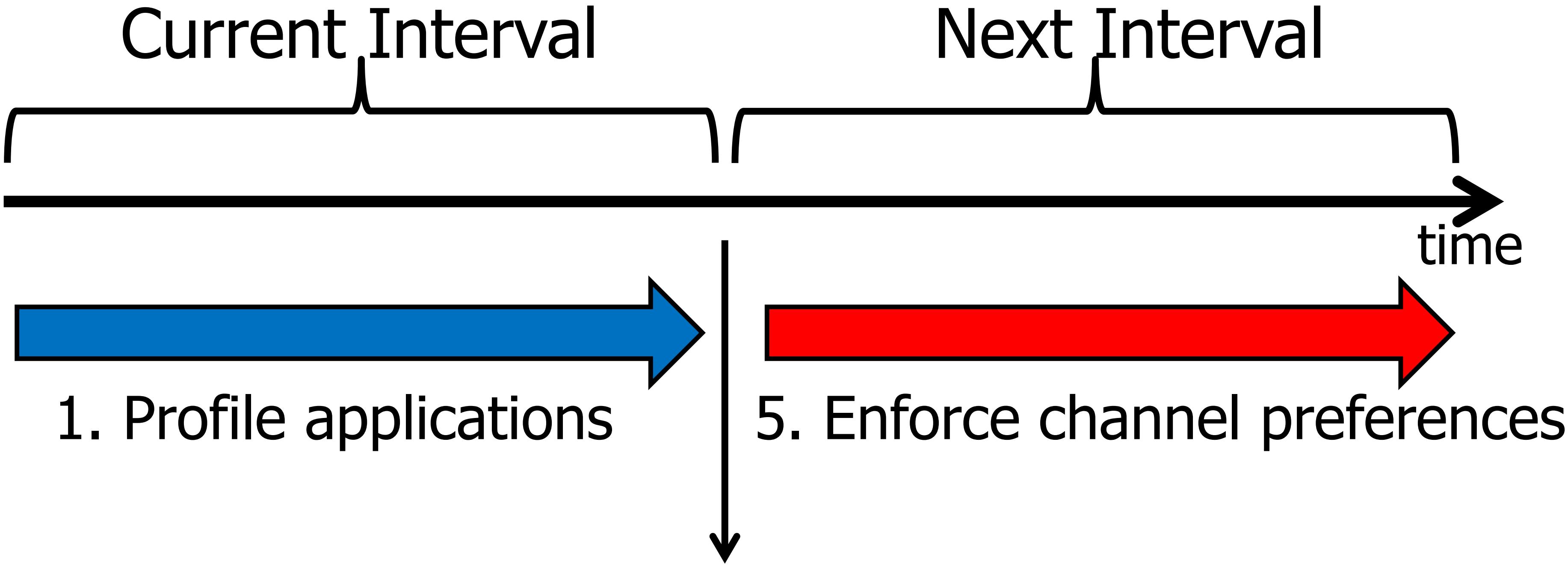
4. Assign a preferred channel to each application

5. Allocate application pages to preferred channel

System Software

Muralidhara et al., “Memory Channel Partitioning,” MICRO’11.

Interval Based Operation



Current Interval

Next Interval

time

1. Profile applications

5. Enforce channel preferences

2. Classify applications into groups
3. Partition channels between groups
4. Assign preferred channel to applications

Observations

- Applications with very low memory-intensity rarely access memory
  → Dedicating channels to them results in precious memory bandwidth waste

- They have the most potential to keep their cores busy
  → We would really like to prioritize them

- They interfere minimally with other applications
  → Prioritizing them does not hurt others

Integrated Memory Partitioning and Scheduling (IMPS)

- Always prioritize very low memory-intensity applications in the memory scheduler

- Use memory channel partitioning to mitigate interference between other applications

Muralidhara et al., "Memory Channel Partitioning," MICRO'11.

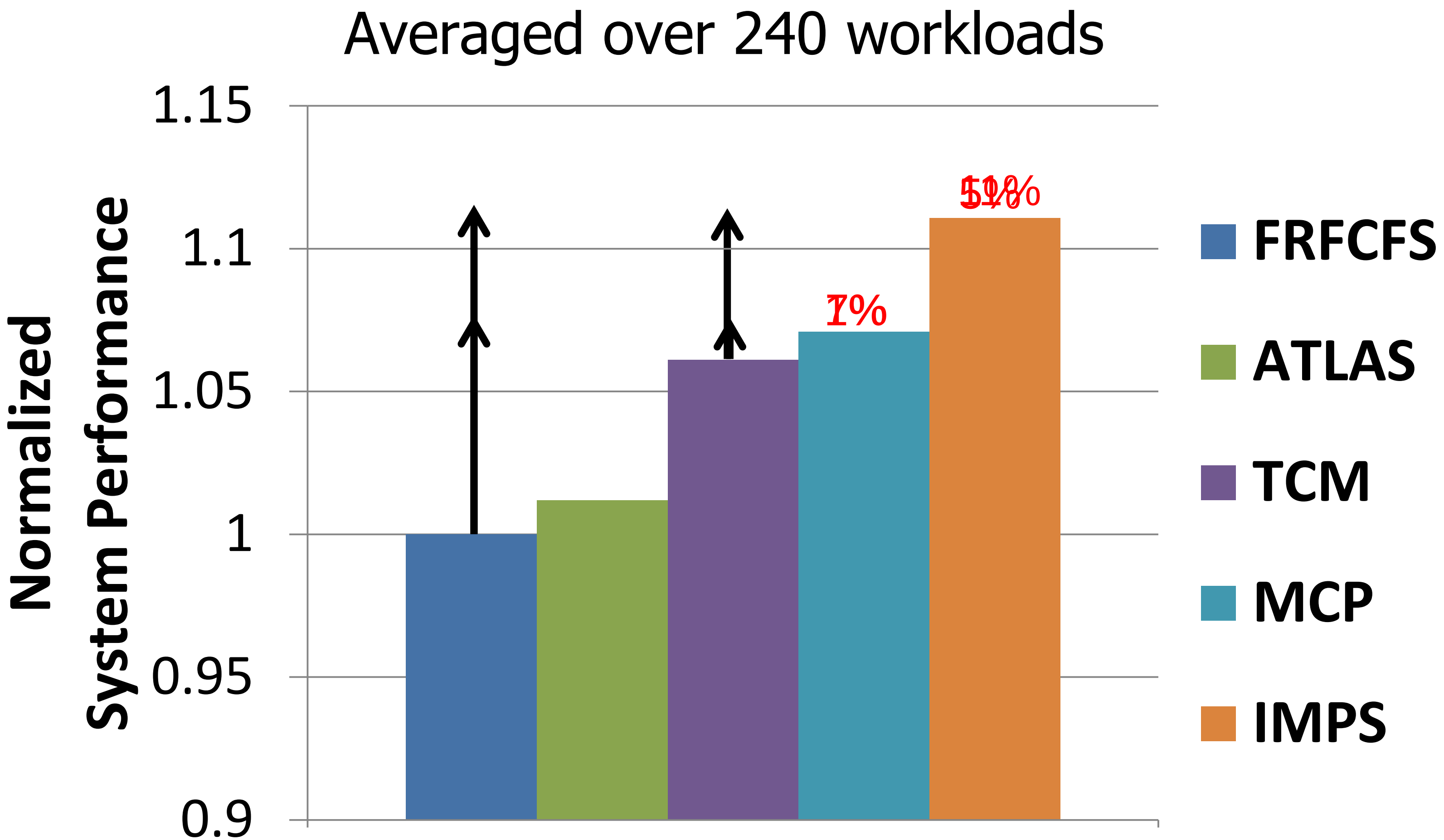
Hardware Cost

- Memory Channel Partitioning (MCP)
  - Only profiling counters in hardware
  - No modifications to memory scheduling logic
  - 1.5 KB storage cost for a 24-core, 4-channel system

- Integrated Memory Partitioning and Scheduling (IMPS)
  - A single bit per request
  - Scheduler prioritizes based on this single bit

Muralidhara et al., “Memory Channel Partitioning,” MICRO’11.

Performance of Channel Partitioning



Better system performance than the best previous scheduler at lower hardware cost

Combining Multiple Interference Control Techniques

- Combined interference control techniques can mitigate interference much more than a single technique alone can do

- The key challenge is:
  - Deciding what technique to apply when
  - Partitioning work appropriately between software and hardware

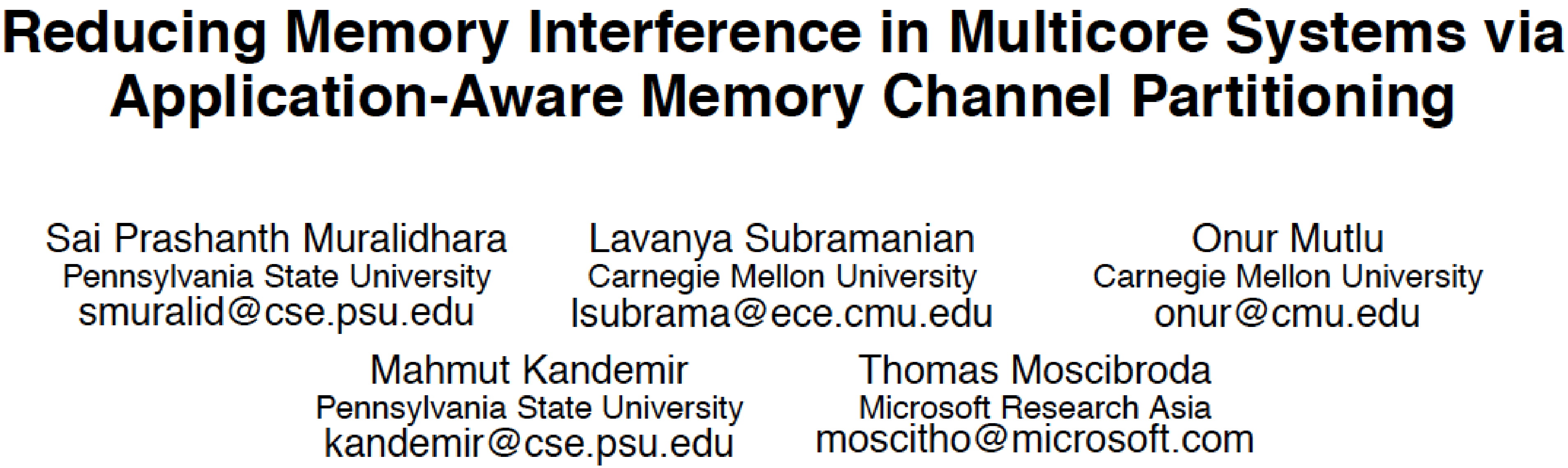
MCP and IMPS: Pros and Cons

- Upsides:
  - Keeps the memory scheduling hardware simple
  - Combines multiple interference reduction techniques
  - Can provide performance isolation across applications mapped to different channels
  - General idea of partitioning can be extended to smaller granularities in the memory hierarchy: banks, subarrays, etc.

- Downsides:
  - Reacting is difficult if workload changes behavior after profiling
  - Overhead of moving pages between channels restricts benefits

More on Memory Channel Partitioning

- Sai Prashanth Muralidhara, Lavanya Subramanian, Onur Mutlu, Mahmut Kandemir, and Thomas Moscibroda, **"Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning"** *Proceedings of the 44th International Symposium on Microarchitecture* (***MICRO***), Porto Alegre, Brazil, December 2011. Slides (pptx)



Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning

Sai Prashanth Muralidhara
Pennsylvania State University
smuralid@cse.psu.edu

Lavanya Subramanian
Carnegie Mellon University
lsubrama@ece.cmu.edu

Onur Mutlu
Carnegie Mellon University
onur@cmu.edu

Mahmut Kandemir
Pennsylvania State University
kandemir@cse.psu.edu

Thomas Moscibroda
Microsoft Research Asia
moscitho@microsoft.com

Fundamental Interference Control Techniques

- Goal: to reduce/control inter-thread memory interference

1. Prioritization or request scheduling

2. Data mapping to banks/channels/ranks

3. Core/source throttling

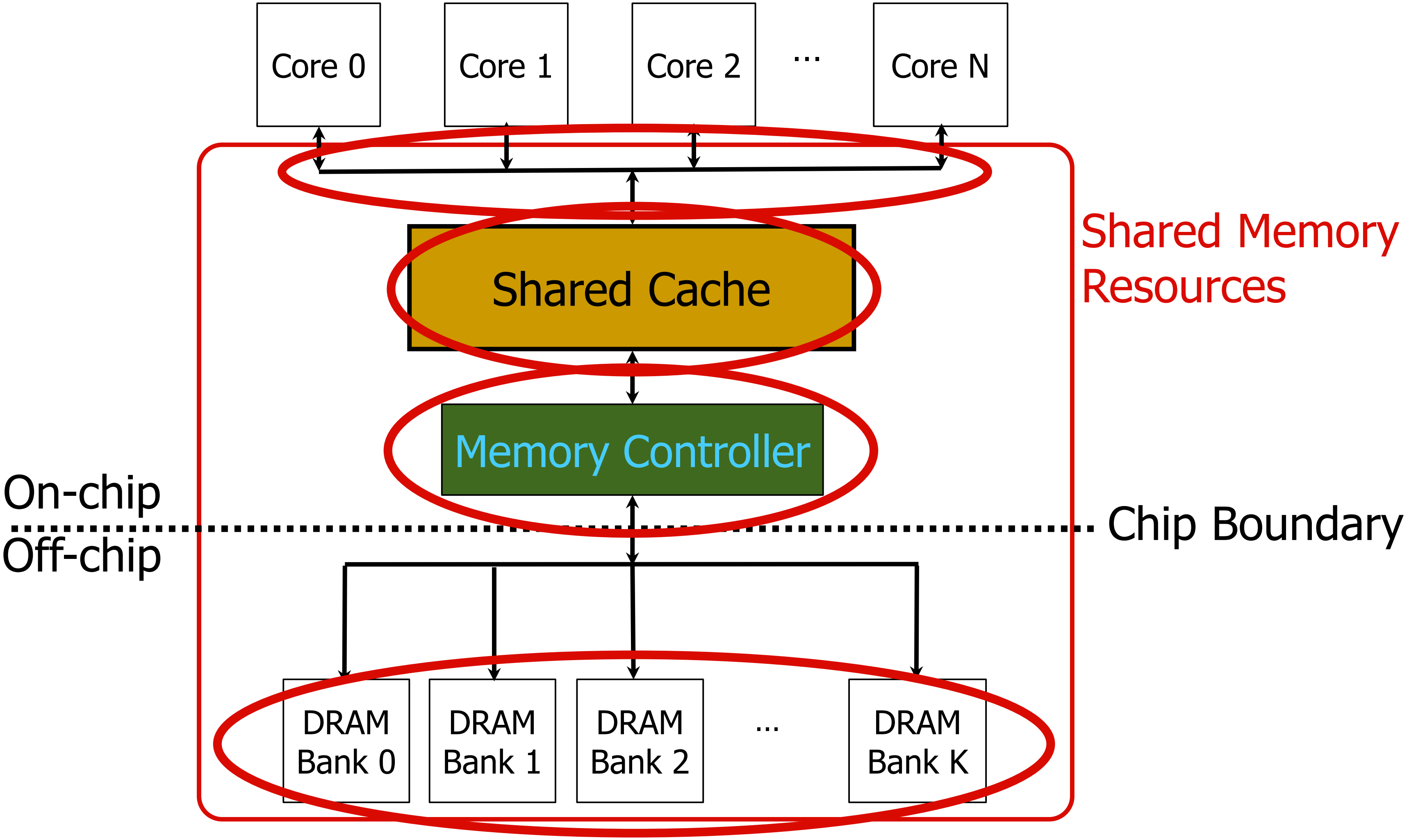
4. Application/thread scheduling

Fairness via Source Throttling

Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt,
"Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems"
15th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (***ASPLOS***),
pages 335-346, Pittsburgh, PA, March 2010. Slides (pdf)

FST ASPLOS 2010 Talk

Many Shared Resources



The Problem with "Smart Resources"

- Independent interference control mechanisms in caches, interconnect, and memory can contradict each other

- Explicitly coordinating mechanisms for different resources requires complex implementation

- How do we enable fair sharing of the entire memory system by controlling interference in a coordinated manner?

Source Throttling: A Fairness Substrate

- Key idea: Manage inter-thread interference at the cores (sources), not at the shared resources

- Dynamically estimate unfairness in the memory system
- Feed back this information into a controller
- Throttle cores' memory access rates accordingly
  - Whom to throttle and by how much depends on performance target (throughput, fairness, per-thread QoS, etc)
  - E.g., if unfairness > system-software-specified target then
    throttle down core causing unfairness &
    throttle up core that was unfairly treated

- Ebrahimi et al., "Fairness via Source Throttling," ASPLOS'10, TOCS'12.

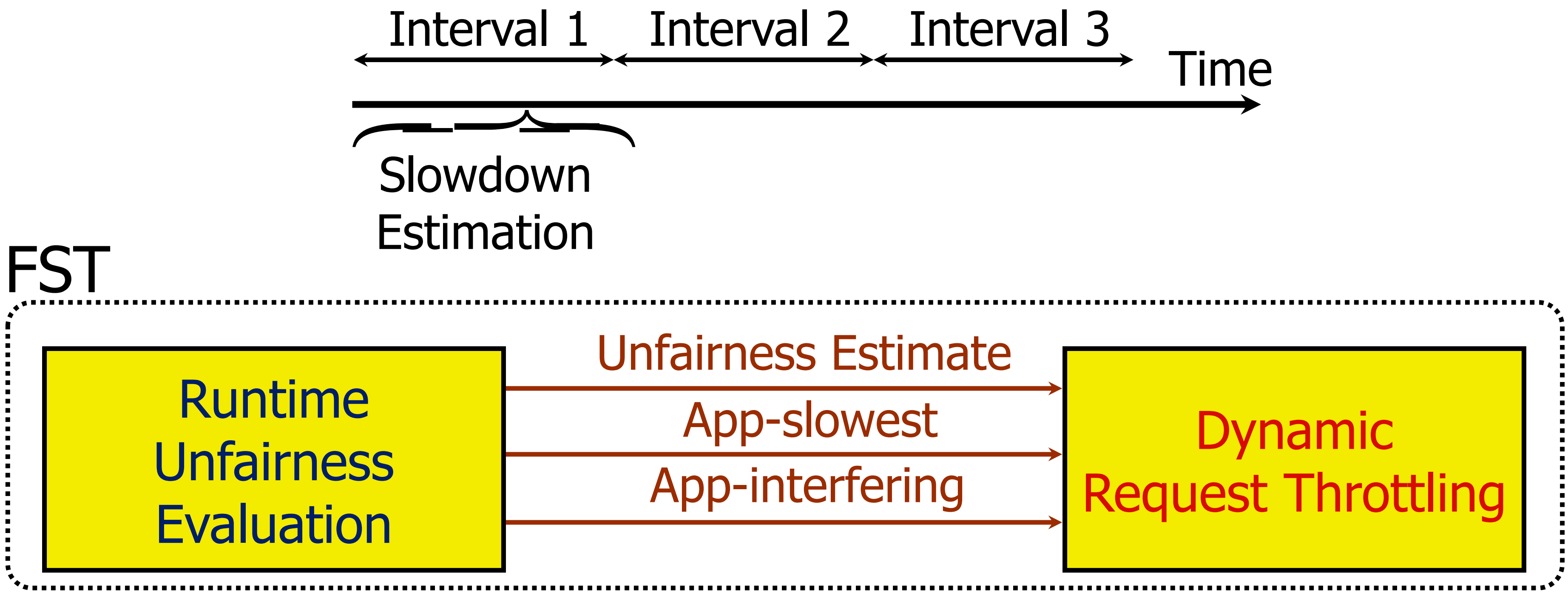
Fairness via Source Throttling (FST)

- Two components (interval-based)

- Run-time unfairness evaluation (in hardware)
  - Dynamically estimates the unfairness (application slowdowns) in the memory system
  - Estimates which application is slowing down which other

- Dynamic request throttling (hardware or software)
  - Adjusts how aggressively each core makes requests to the shared resources
  - Throttles down request rates of cores causing unfairness
    - Limit miss buffers, limit injection rate

Fairness via Source Throttling (FST) [ASPLOS'10]



1- Estimating system unfairness
2- Find app. with the highest slowdown (App-slowest)
3- Find app. causing most interference for App-slowest (App-interfering)

```
if (Unfairness Estimate >Target)
{
 1-Throttle down App-interfering
   (limit injection rate and parallelism)
 2-Throttle up App-slowest
}
```

Dynamic Request Throttling

- Goal: Adjust how aggressively each core makes requests to the shared memory system

- Mechanisms:
  - Miss Status Holding Register (MSHR) quota
    - Controls the number of concurrent requests accessing shared resources from each application
  - Request injection frequency
    - Controls how often memory requests are issued to the last level cache from the MSHRs

Dynamic Request Throttling

- Throttling level assigned to each core determines both MSHR quota and request injection rate

Throttling level	MSHR quota	Request Injection Rate
100%	128	Every cycle
50%	64	Every other cycle
25%	32	Once every 4 cycles
10%	12	Once every 10 cycles
5%	6	Once every 20 cycles
4%	5	Once every 25 cycles
3%	3	Once every 30 cycles

Total # of MSHRs: 128

System Software Support

- Different fairness objectives can be configured by system software
  - Keep maximum slowdown in check
    - Estimated Max Slowdown < Target Max Slowdown
  - Keep slowdown of particular applications in check to achieve a particular performance target
    - Estimated Slowdown(i) < Target Slowdown(i)

- Support for thread priorities
  - Weighted Slowdown(i) = Estimated Slowdown(i) x Weight(i)

Source Throttling Results: Takeaways

- Source throttling alone provides better performance than a combination of "smart" memory scheduling and fair caching
  - Decisions made at the memory scheduler and the cache sometimes contradict each other

- Neither source throttling alone nor "smart resources" alone provides the best performance

- Combined approaches are even more powerful
  - Source throttling and resource-based interference control

Source Throttling: Ups and Downs

- Advantages

  + Core/request throttling is easy to implement: no need to change the memory scheduling algorithm

  + Can be a general way of handling shared resource contention

  + Can reduce overall load/contention in the memory system

- Disadvantages

  - Requires slowdown estimations → difficult to estimate

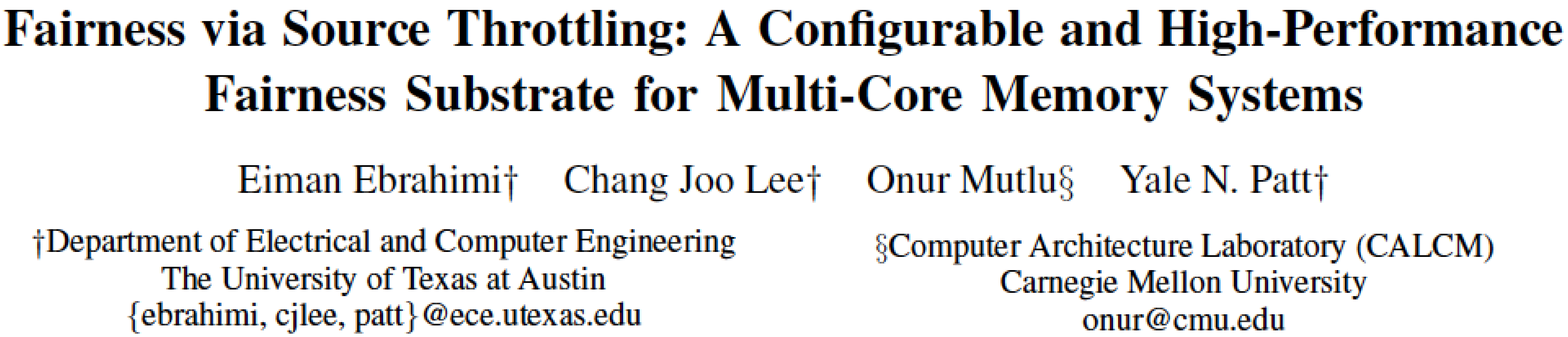
  - Thresholds can become difficult to optimize

  → throughput loss due to too much throttling

  → can be difficult to find an overall-good configuration

More on Source Throttling (I)

- Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt, **"Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems"**
  *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (**ASPLOS**)*, pages 335-346, Pittsburgh, PA, March 2010.
  Slides (pdf)



Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems

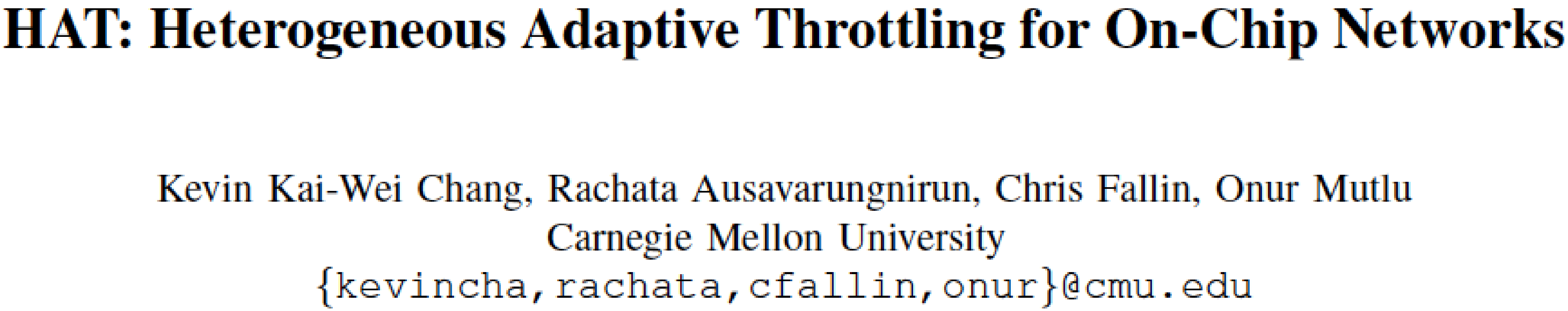
Eiman Ebrahimi† Chang Joo Lee† Onur Mutlu§ Yale N. Patt†

†Department of Electrical and Computer Engineering
The University of Texas at Austin
{ebrahimi, cjlee, patt}@ece.utexas.edu

§Computer Architecture Laboratory (CALCM)
Carnegie Mellon University
onur@cmu.edu

More on Source Throttling (II)

- Kevin Chang, Rachata Ausavarungnirun, Chris Fallin, and Onur Mutlu, **"HAT: Heterogeneous Adaptive Throttling for On-Chip Networks"**
  *Proceedings of the 24th International Symposium on Computer Architecture and High Performance Computing (**SBAC-PAD**)*, New York, NY, October 2012. Slides (pptx) (pdf)

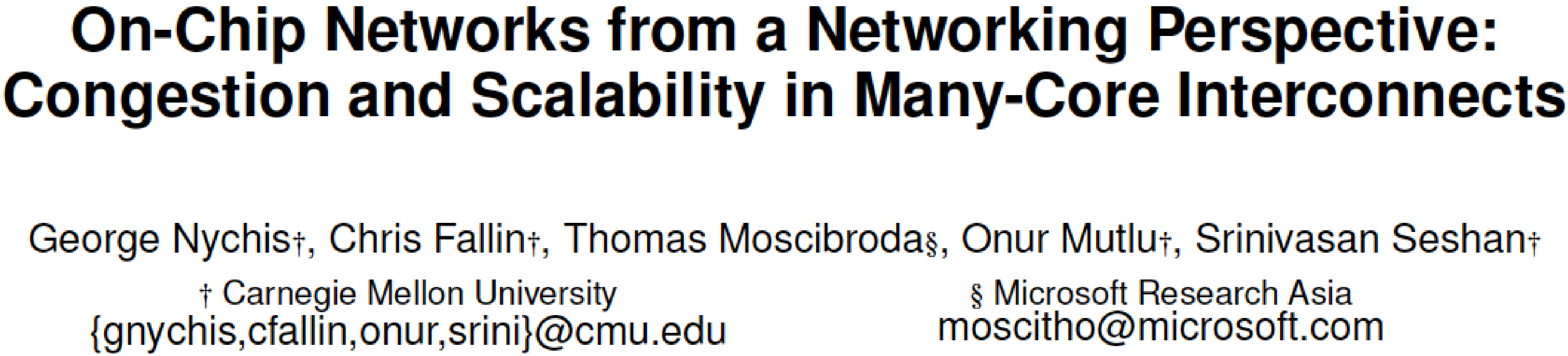


HAT: Heterogeneous Adaptive Throttling for On-Chip Networks

Kevin Kai-Wei Chang, Rachata Ausavarungnirun, Chris Fallin, Onur Mutlu
Carnegie Mellon University
{kevincha,rachata,cfallin,onur}@cmu.edu

More on Source Throttling (III)

- George Nychis, Chris Fallin, Thomas Moscibroda, Onur Mutlu, and Srinivasan Seshan, **"On-Chip Networks from a Networking Perspective: Congestion and Scalability in Many-core Interconnects"** *Proceedings of the 2012 ACM SIGCOMM Conference (**SIGCOMM**)*, Helsinki, Finland, August 2012. Slides (pptx)



On-Chip Networks from a Networking Perspective: Congestion and Scalability in Many-Core Interconnects

George Nychis†, Chris Fallin†, Thomas Moscibroda§, Onur Mutlu†, Srinivasan Seshan†

† Carnegie Mellon University
{gnychis,cfallin,onur,srini}@cmu.edu

§ Microsoft Research Asia
moscitho@microsoft.com

Fundamental Interference Control Techniques

- Goal: to reduce/control interference

1. Prioritization or request scheduling

2. Data mapping to banks/channels/ranks

3. Core/source throttling

4. Application/thread scheduling

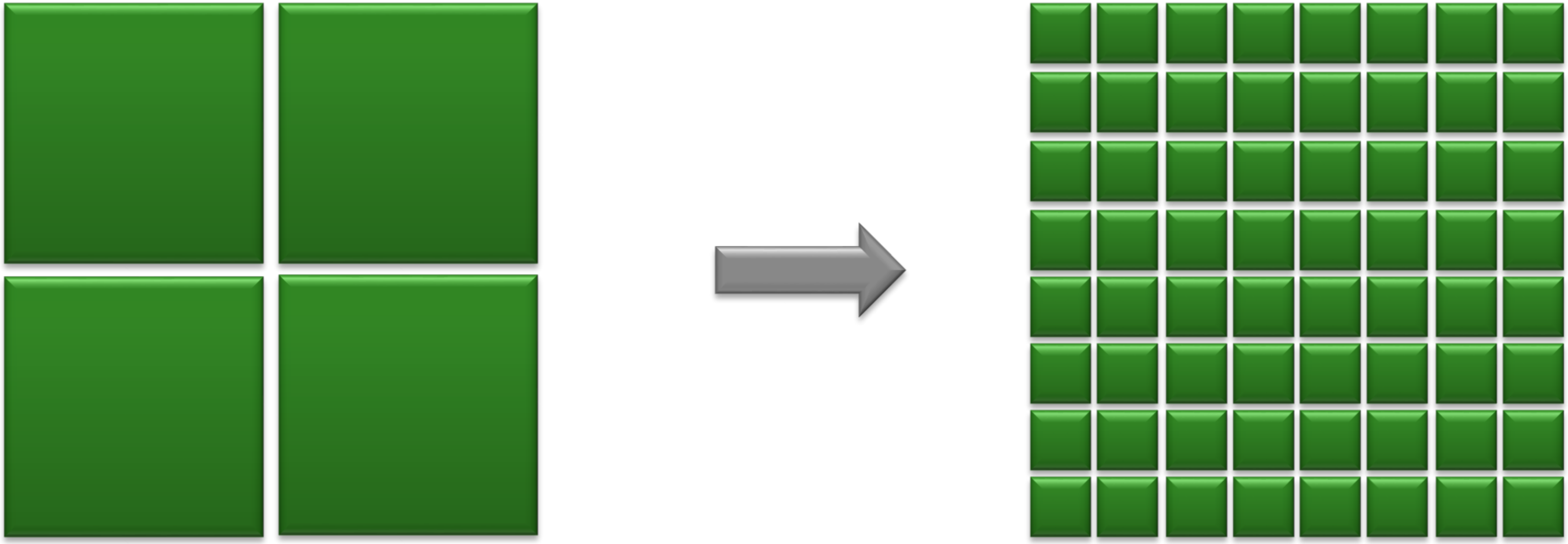
Idea: Pick threads that do not badly interfere with each other to be scheduled together on cores sharing the memory system

Application-to-Core Mapping to Reduce Interference

- Reetuparna Das, Rachata Ausavarungnirun, Onur Mutlu, Akhilesh Kumar, and Mani Azimi,
  "Application-to-Core Mapping Policies to Reduce Memory System Interference in Multi-Core Systems"
  Proceedings of the 19th International Symposium on High-Performance Computer Architecture (***HPCA***), Shenzhen, China, February 2013.
  Slides (pptx)

- Key ideas:
  - Cluster threads to memory controllers (to reduce across chip interference)
  - Isolate interference-sensitive (low-intensity) applications in a separate cluster (to reduce interference from high-intensity applications)
  - Place applications that benefit from memory bandwidth closer to the controller

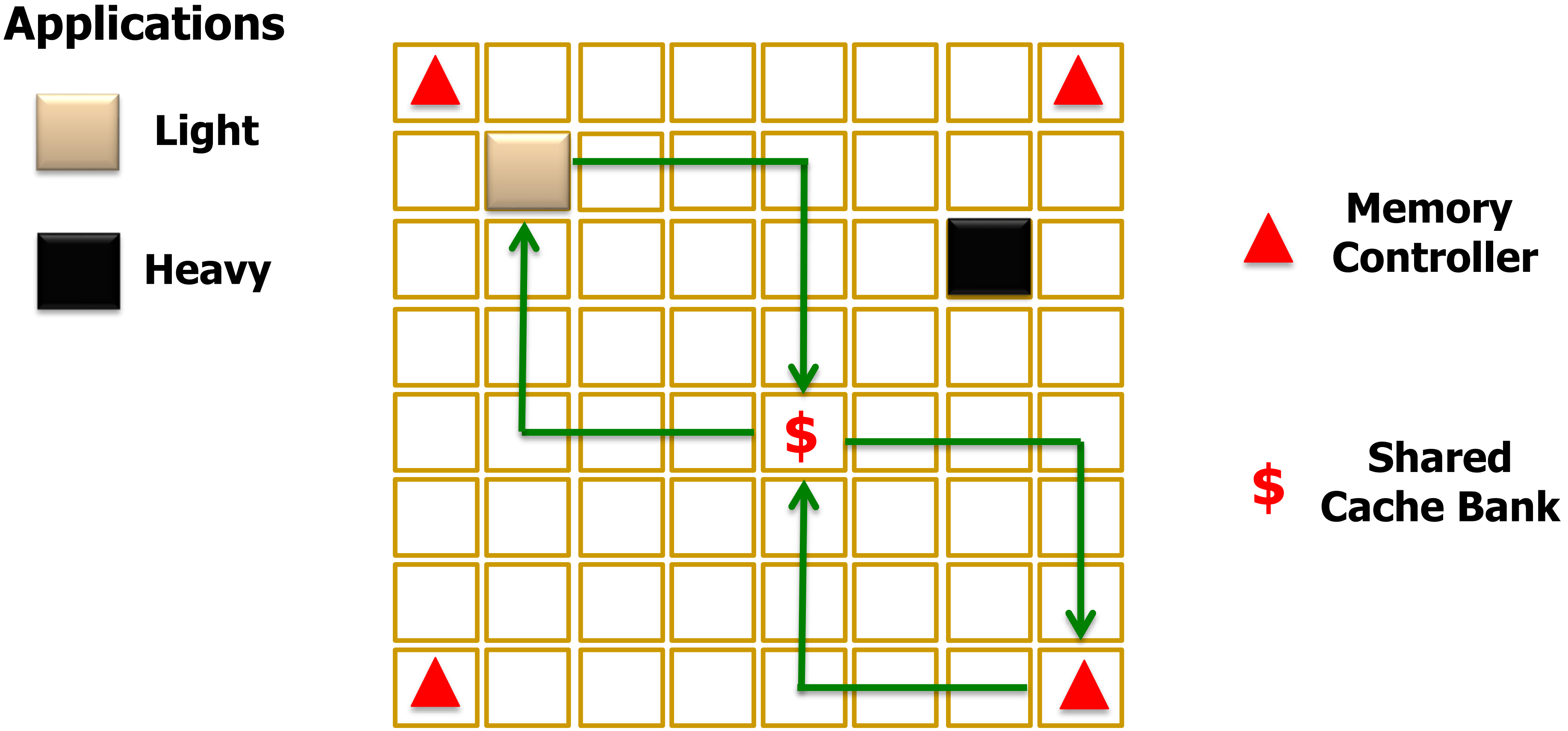
Multi-Core to Many-Core



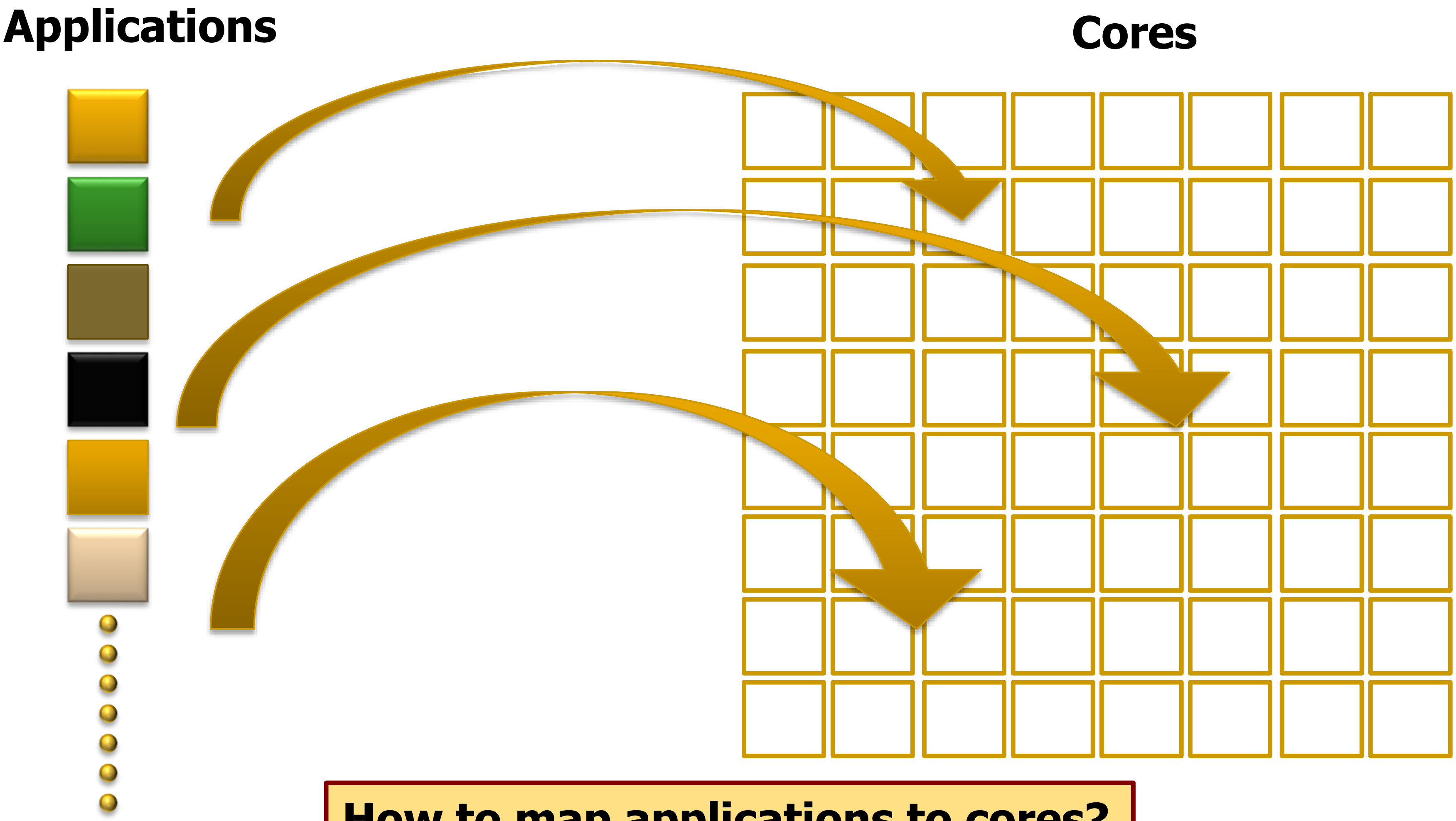
Multi-Core

Many-Core

Many-Core On-Chip Communication

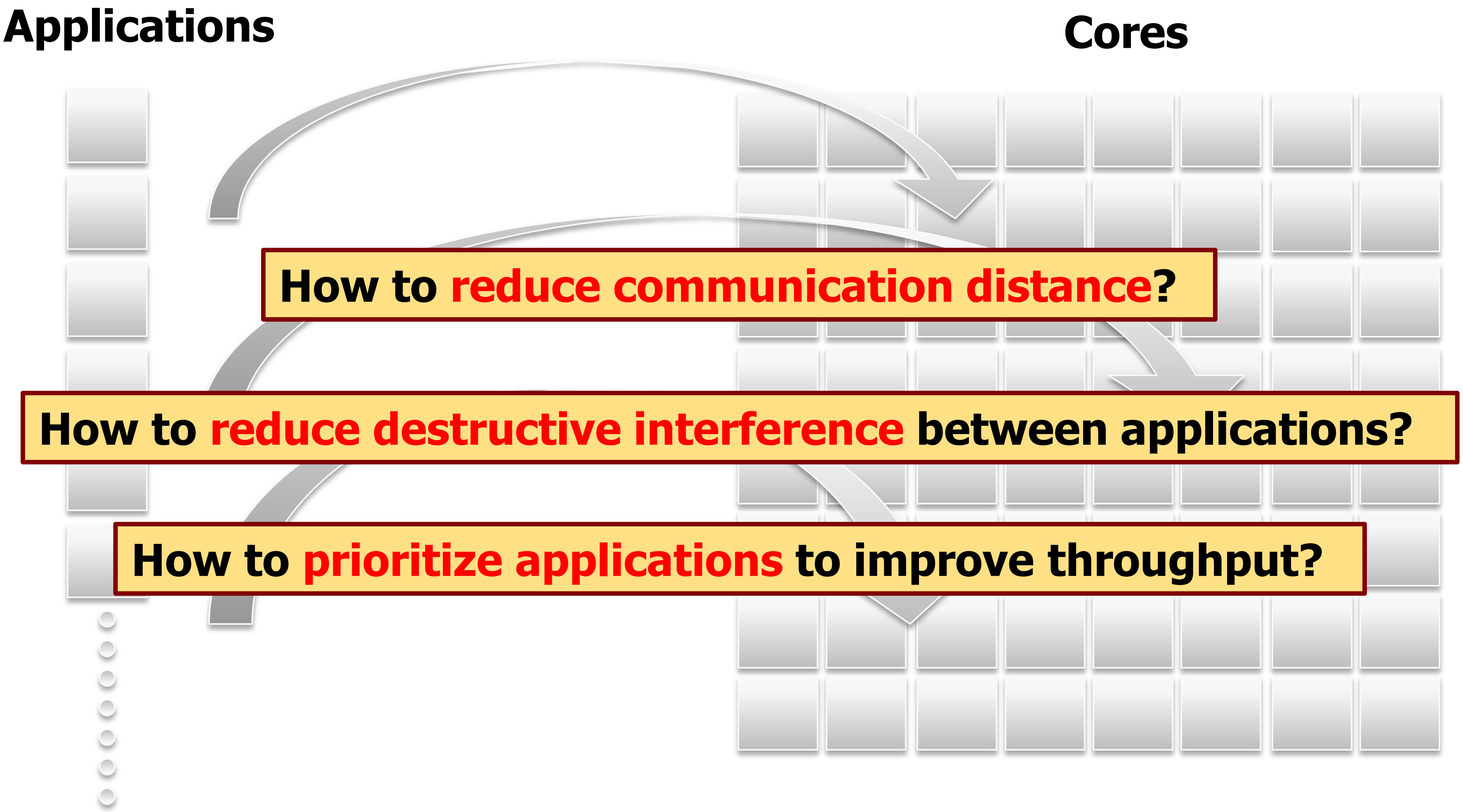


Problem: Spatial Task Scheduling



How to map applications to cores?

Challenges in Spatial Task Scheduling

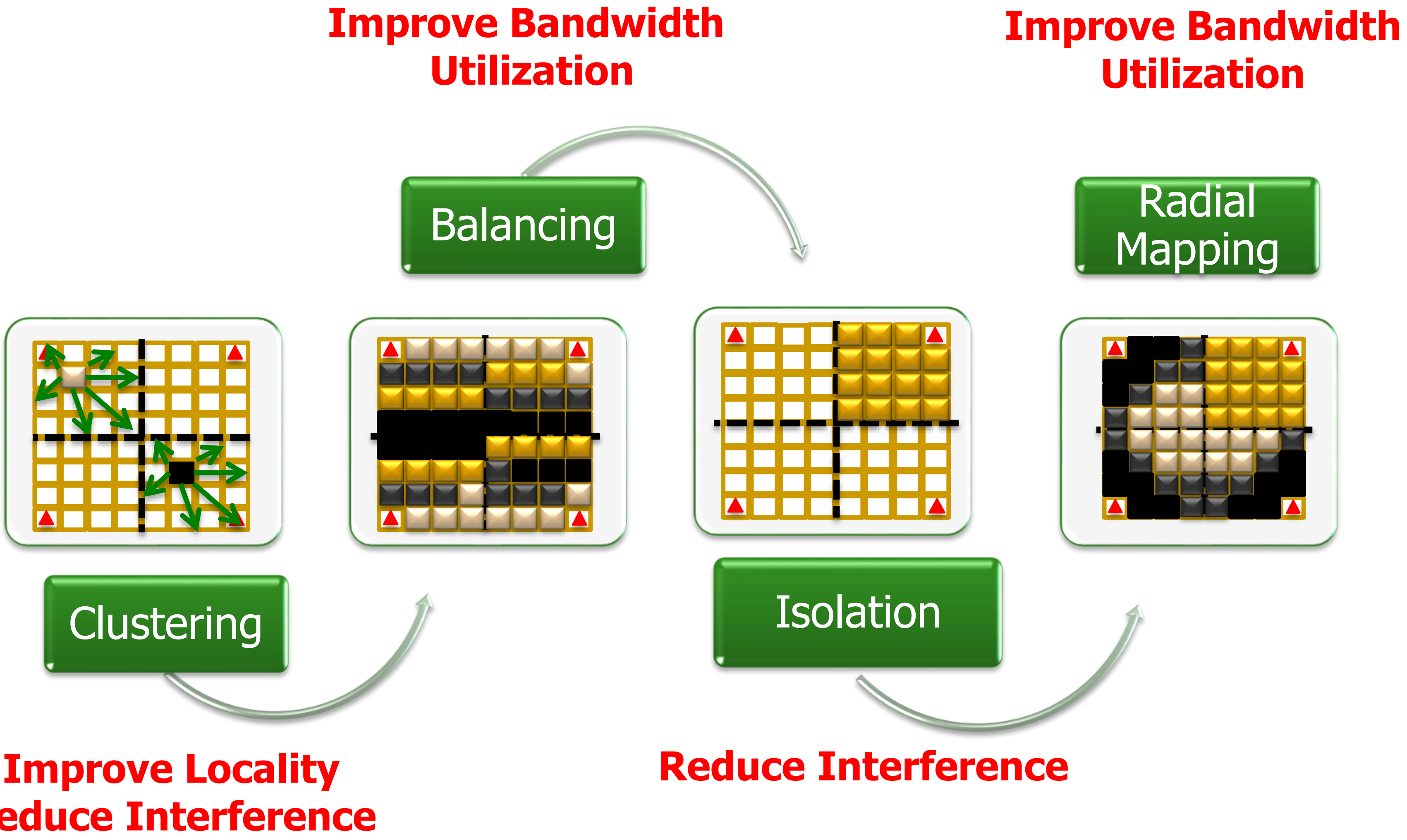


How to **reduce communication distance**?

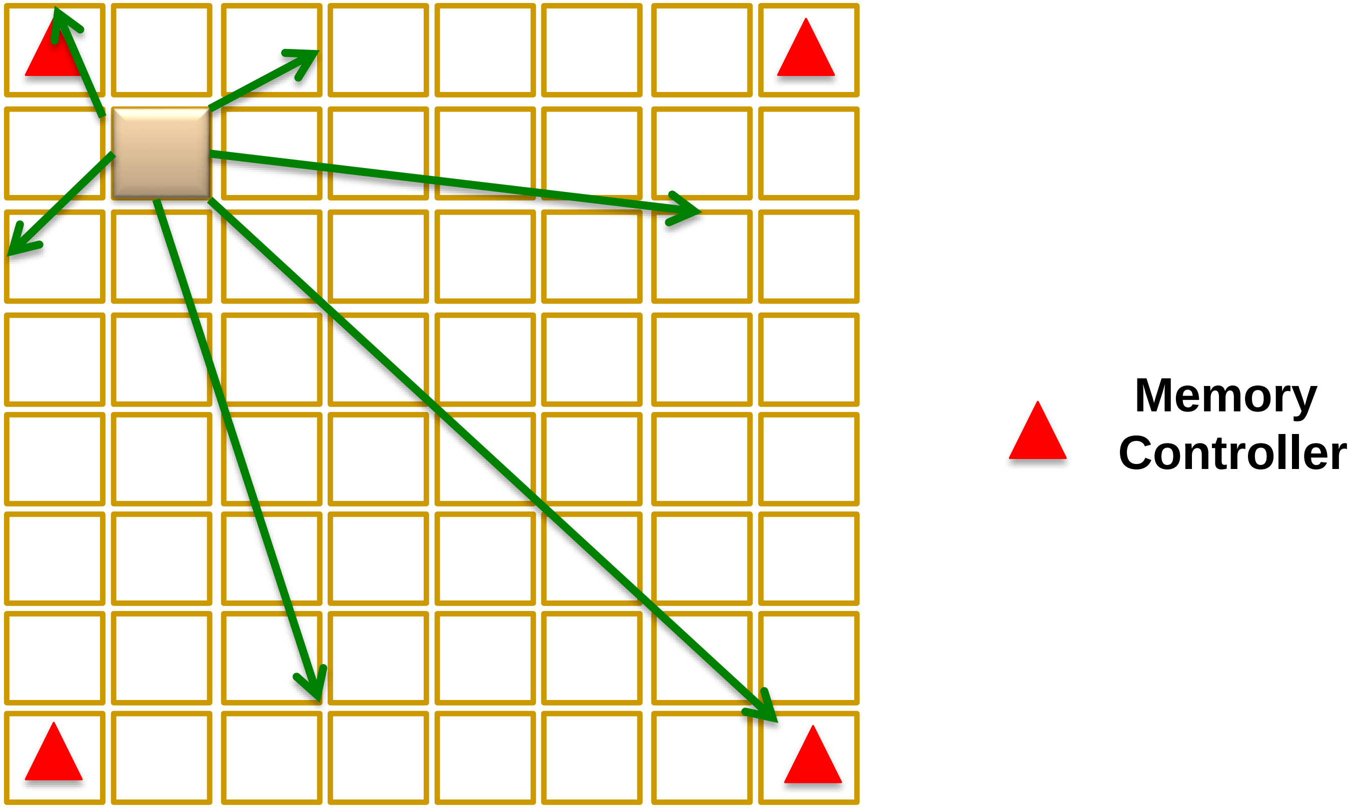
How to **reduce destructive interference** between applications?

How to **prioritize applications** to improve throughput?

Application-to-Core Mapping



Step 1 — Clustering

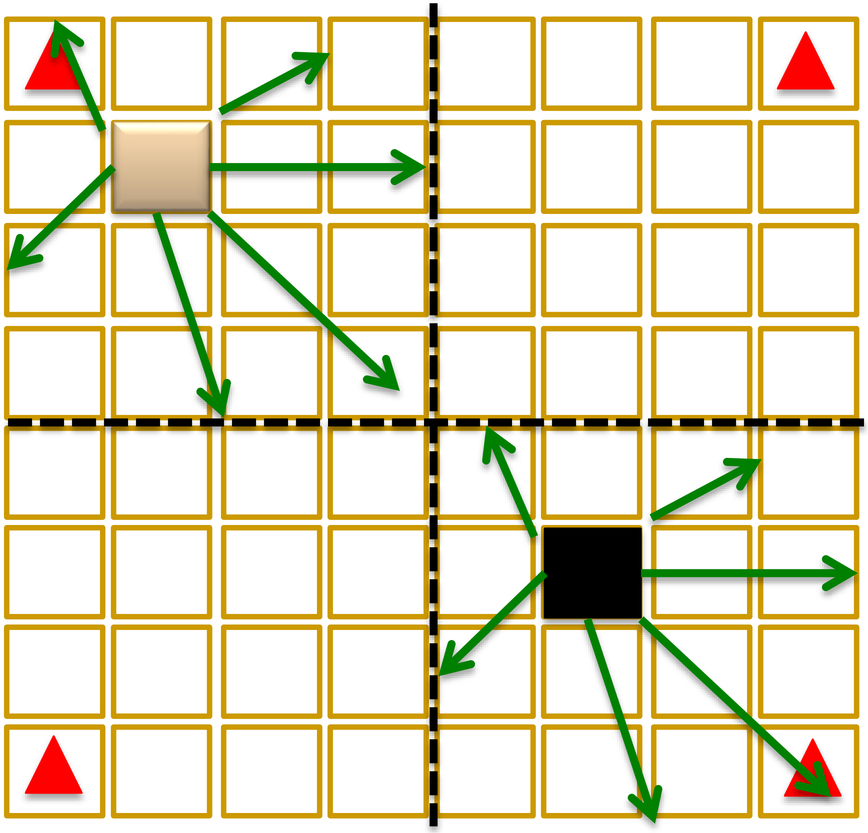


Inefficient data mapping to memory and caches

Step 1 — Clustering

Cluster 0

Cluster 2



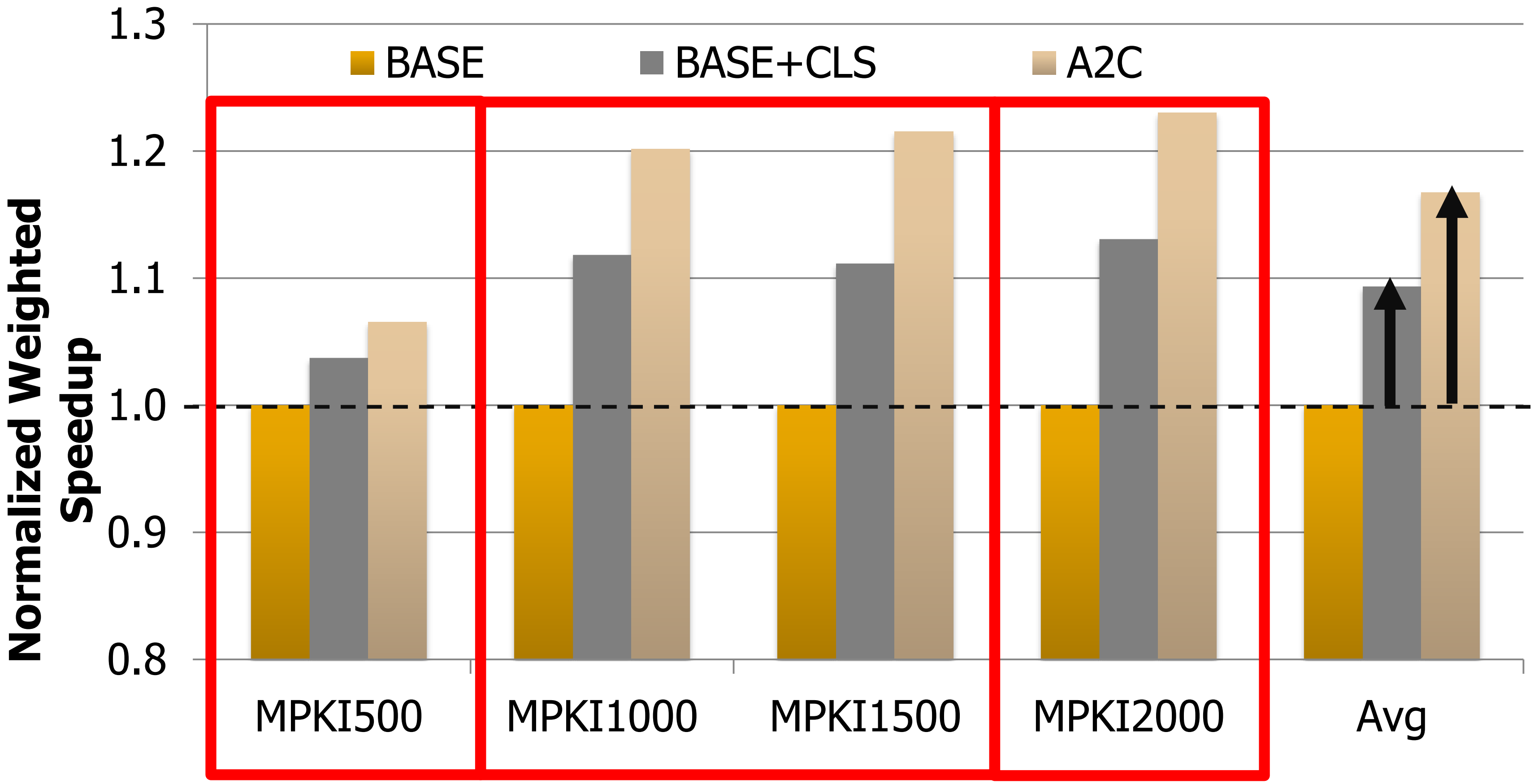
Cluster 1

Cluster 3

Improved Locality

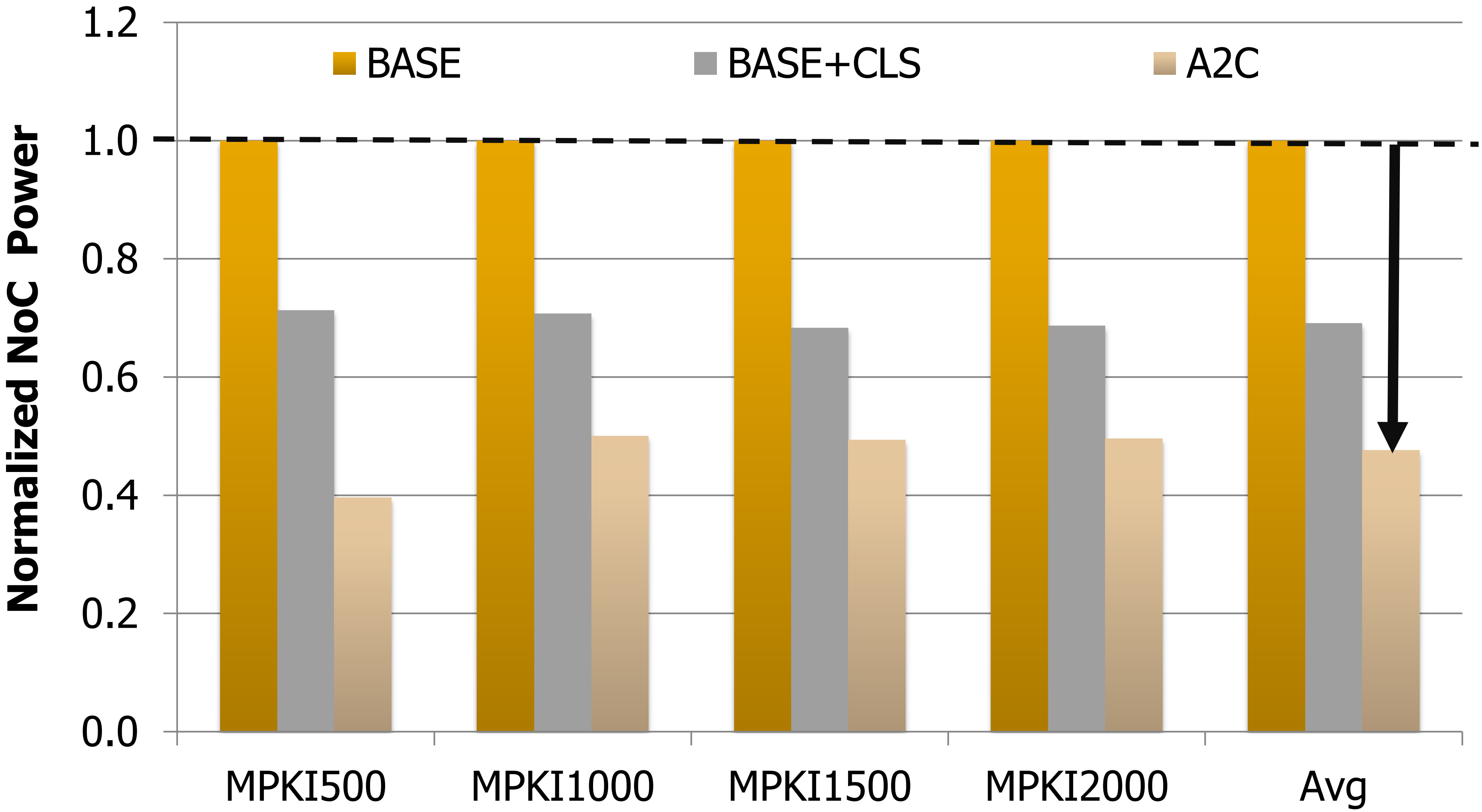
Reduced Interference

System Performance



System performance improves by 17%

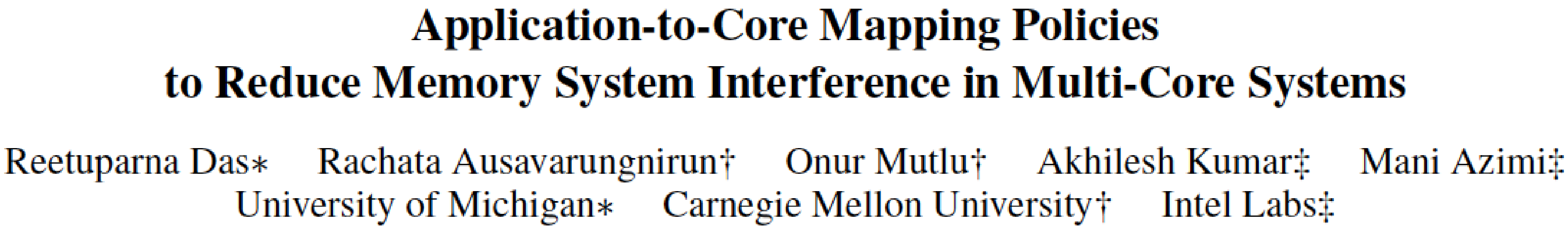
Network Power



Average network power consumption reduces by 52%

More on App-to-Core Mapping

- Reetuparna Das, Rachata Ausavarungnirun, Onur Mutlu, Akhilesh Kumar, and Mani Azimi, **"Application-to-Core Mapping Policies to Reduce Memory System Interference in Multi-Core Systems"** *Proceedings of the 19th International Symposium on High-Performance Computer Architecture (**HPCA**)*, Shenzhen, China, February 2013. Slides (pptx)



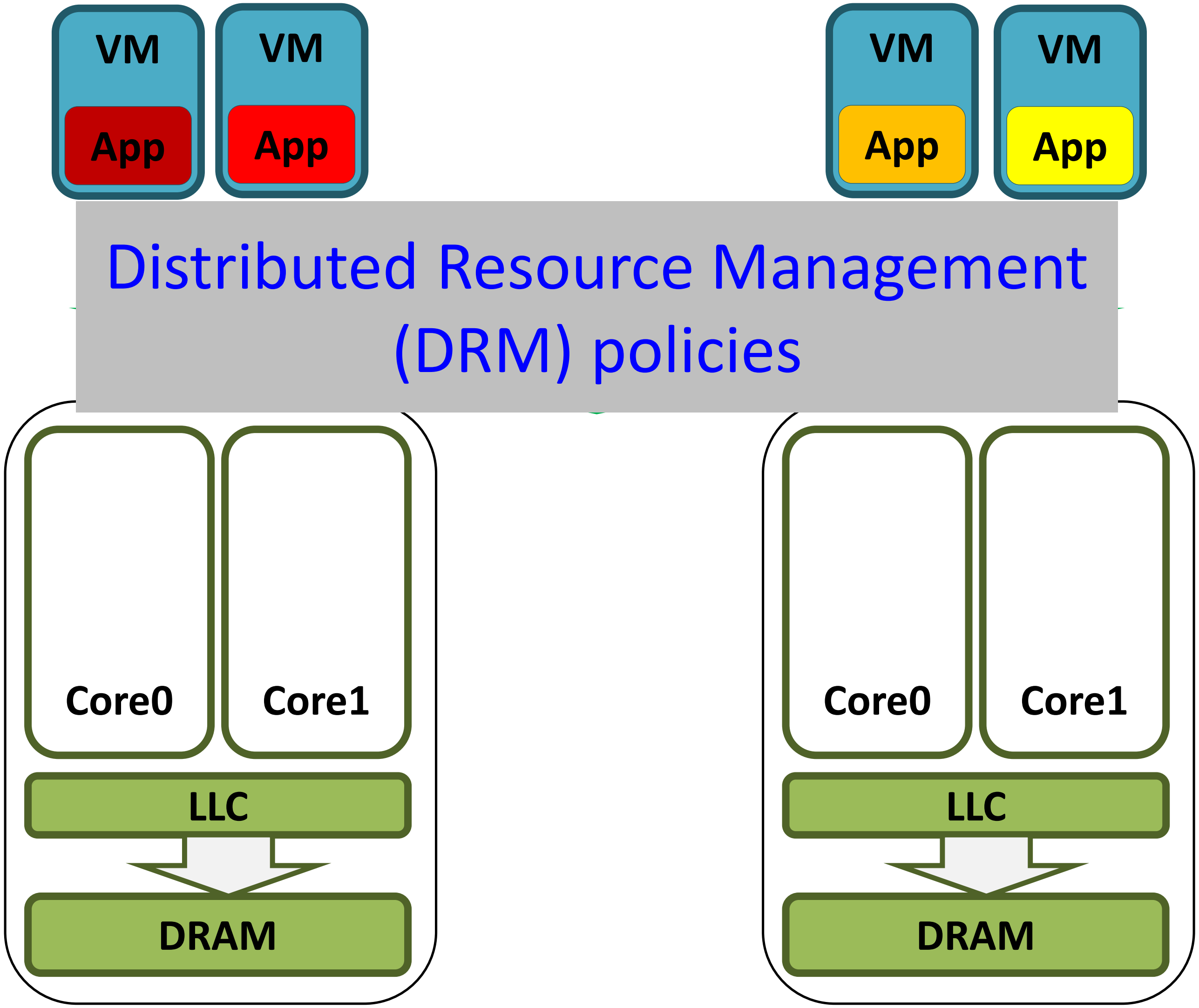
Application-to-Core Mapping Policies
to Reduce Memory System Interference in Multi-Core Systems

Reetuparna Das∗ Rachata Ausavarungnirun† Onur Mutlu† Akhilesh Kumar‡ Mani Azimi‡
University of Michigan∗ Carnegie Mellon University† Intel Labs‡

Interference-Aware Thread Scheduling

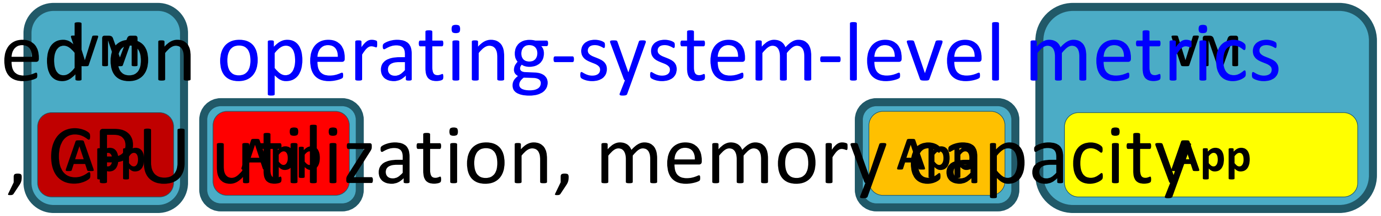
- An example from scheduling in compute clusters (data centers)
- Data centers can be running virtual machines

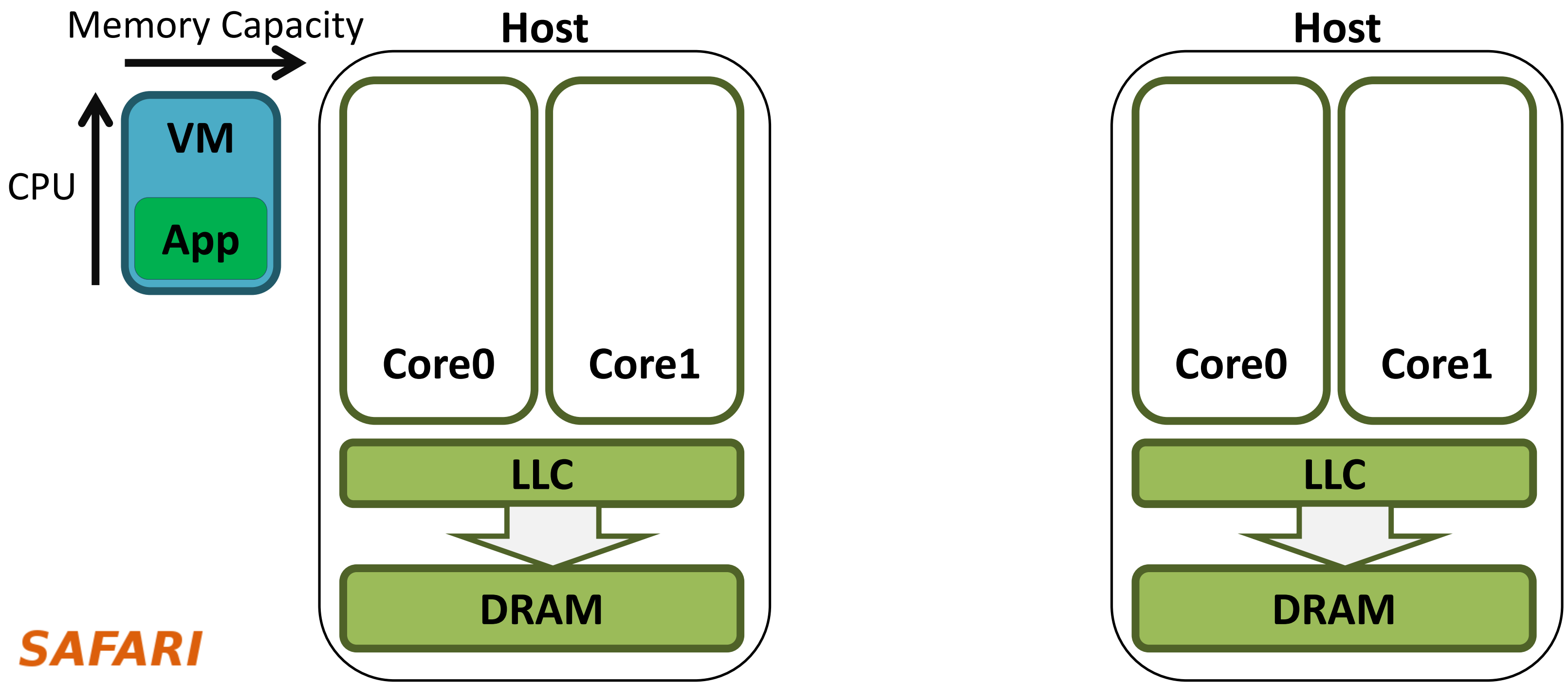
Virtualized Cluster



Conventional DRM Policies

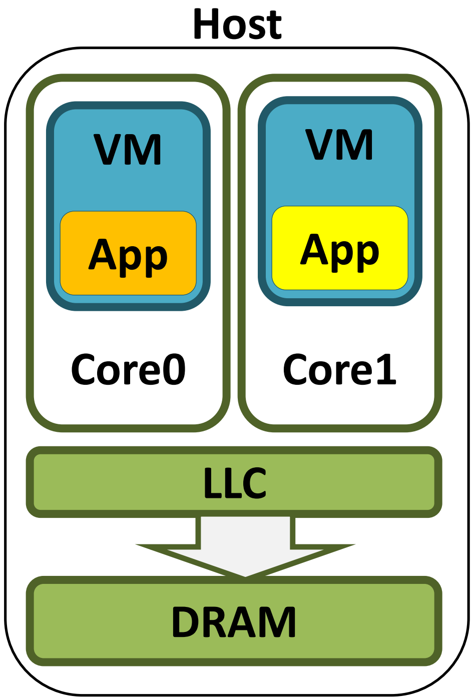
Based on operating-system-level metrics e.g., CPU utilization, memory capacity demand





Microarchitecture-level Interference

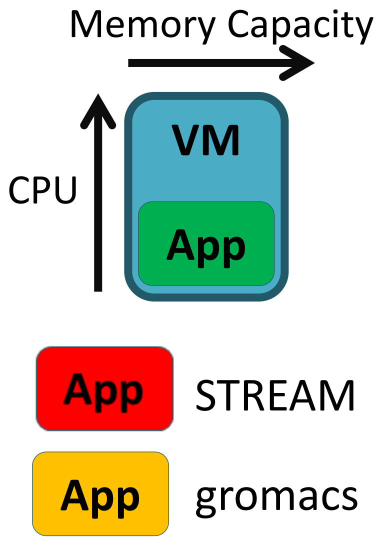
- VMs within a host compete for:
  - Shared cache capacity
  - Shared memory bandwidth

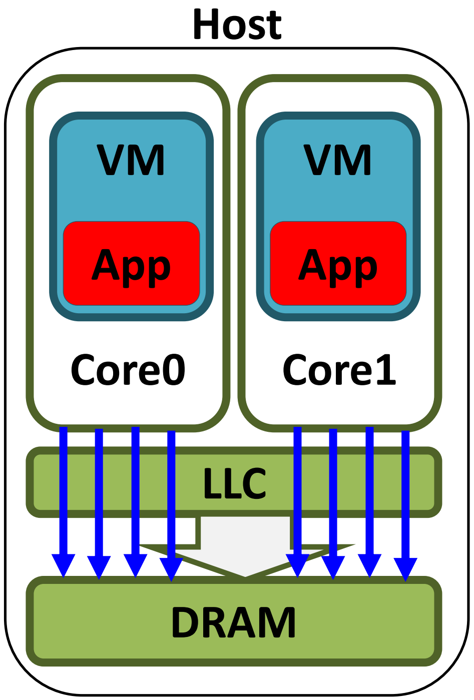


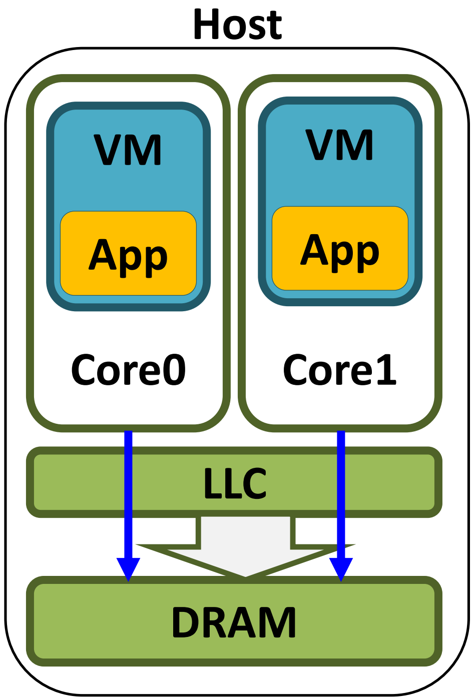
Can operating-system-level metrics capture the microarchitecture-level resource interference?

Microarchitecture Unawareness

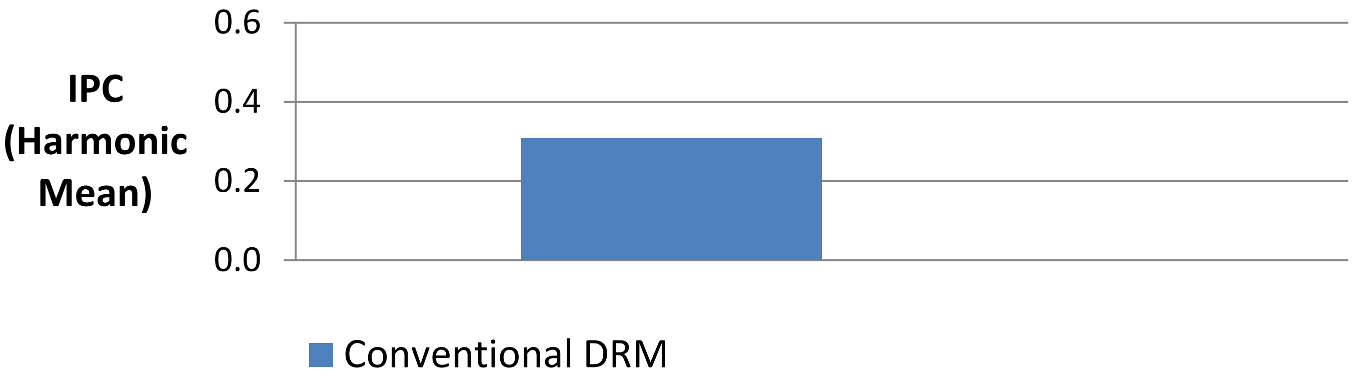
VM	Operating-system-level metrics		Microarchitecture-level metrics	
	CPU Utilization	Memory Capacity	LLC Hit Ratio	Memory Bandwidth
🟥 App	92%	369 MB	**2%**	**2267 MB/s**
🟨 App	93%	348 MB	**98%**	**1 MB/s**

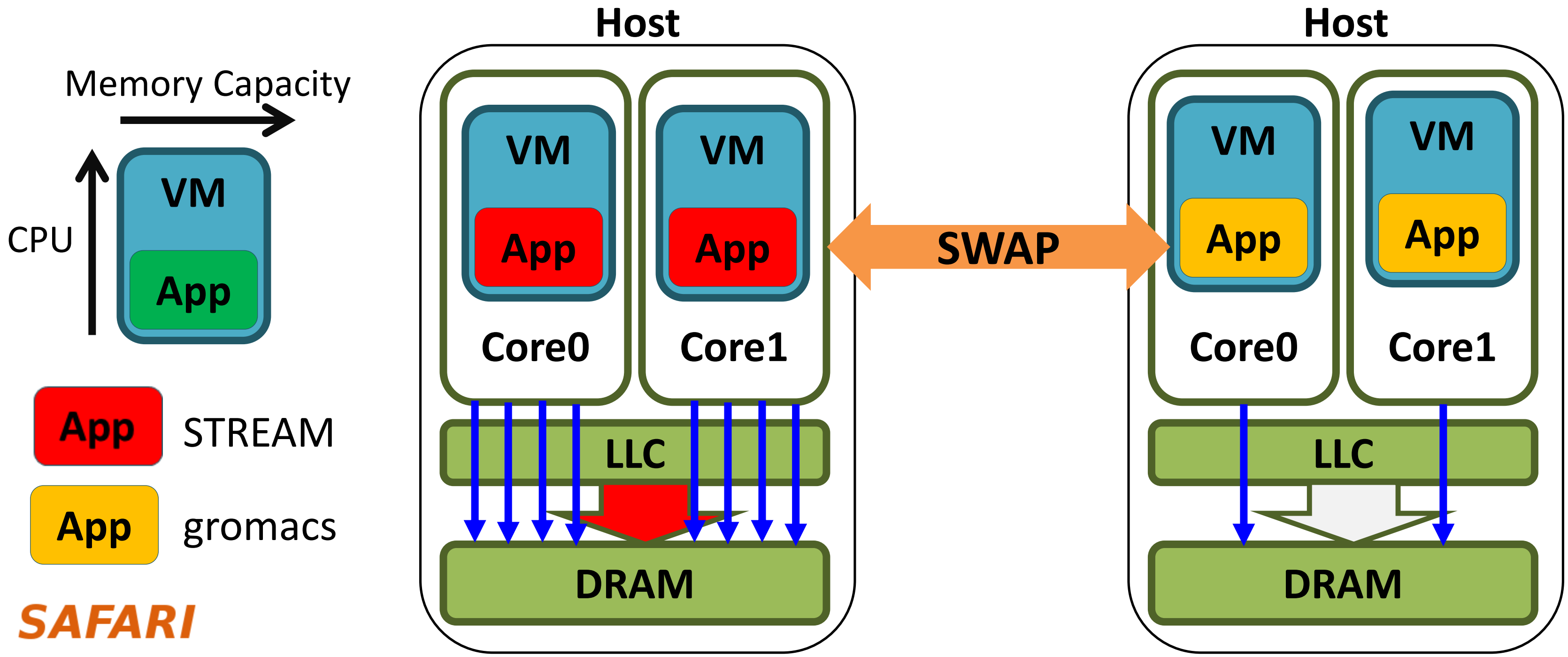




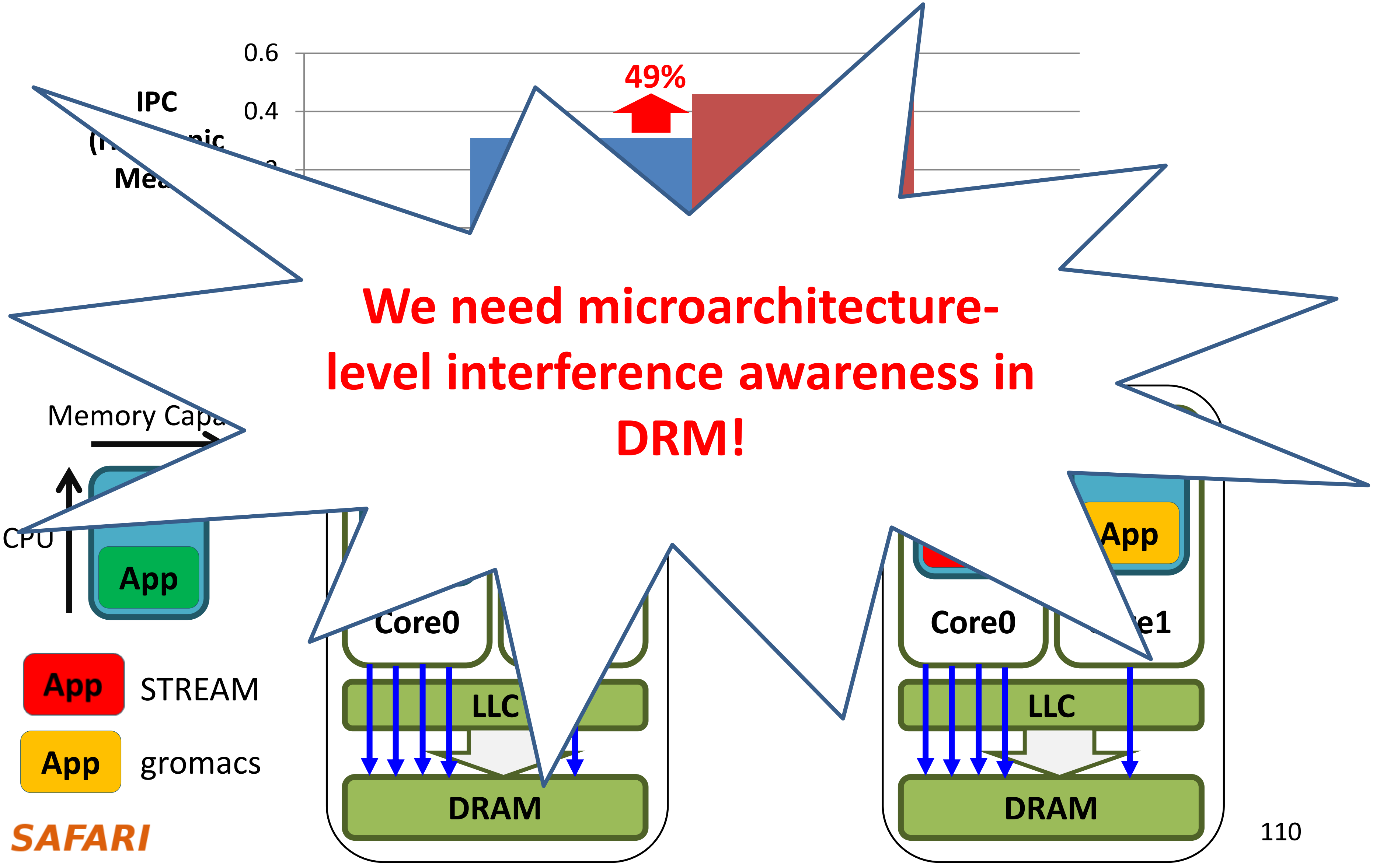


Impact on Performance





Impact on Performance



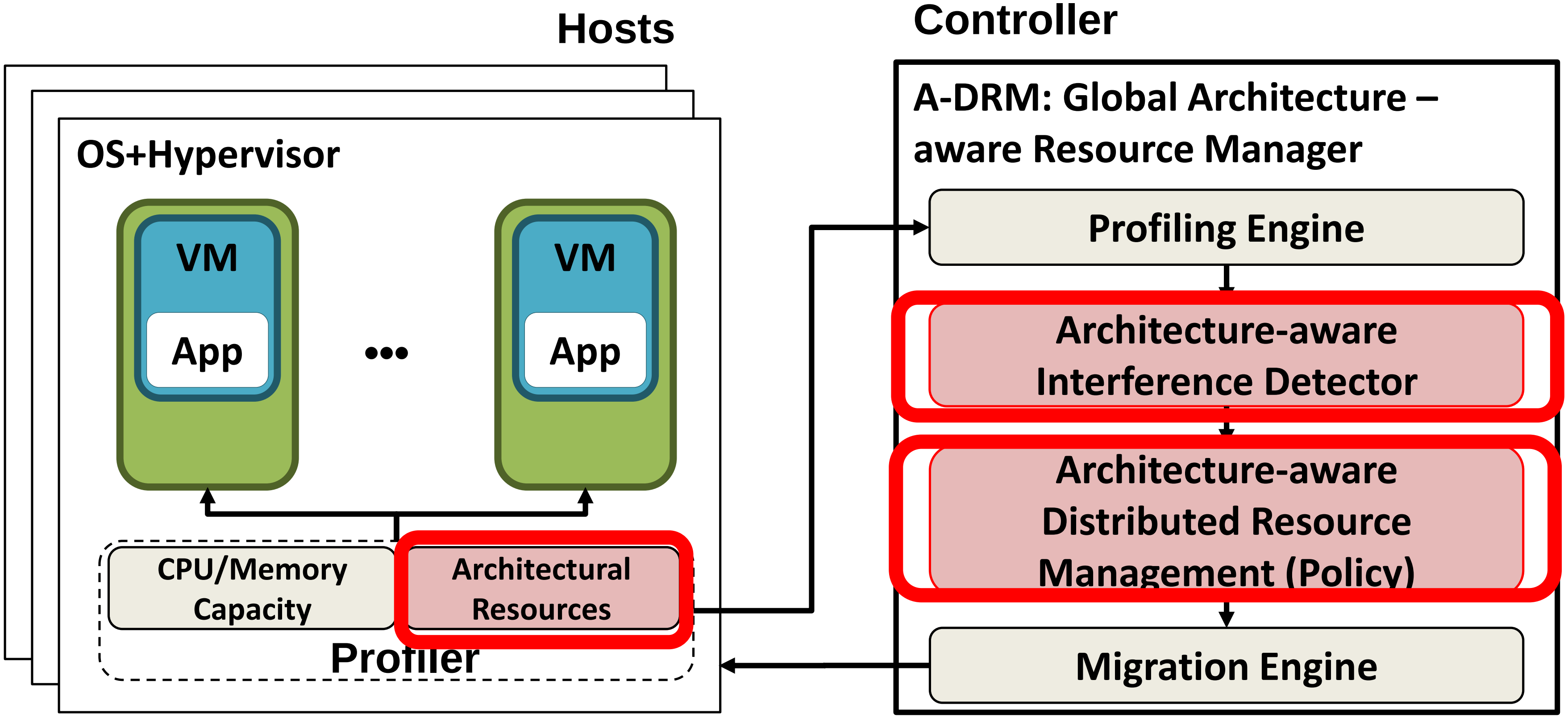
We need microarchitecture-level interference awareness in DRM!

A-DRM: Architecture-aware DRM

- **Goal**: Take into account microarchitecture-level shared resource interference
  - Shared cache capacity
  - Shared memory bandwidth

- **Key Idea**:
  - Monitor and detect microarchitecture-level shared resource interference
  - Balance microarchitecture-level resource usage across cluster to minimize memory interference while maximizing system performance

A-DRM: Architecture-aware DRM



Hosts

Controller

OS+Hypervisor

VM

App

•••

VM

App

CPU/Memory Capacity

Architectural Resources

Profiler

A-DRM: Global Architecture – aware Resource Manager

Profiling Engine

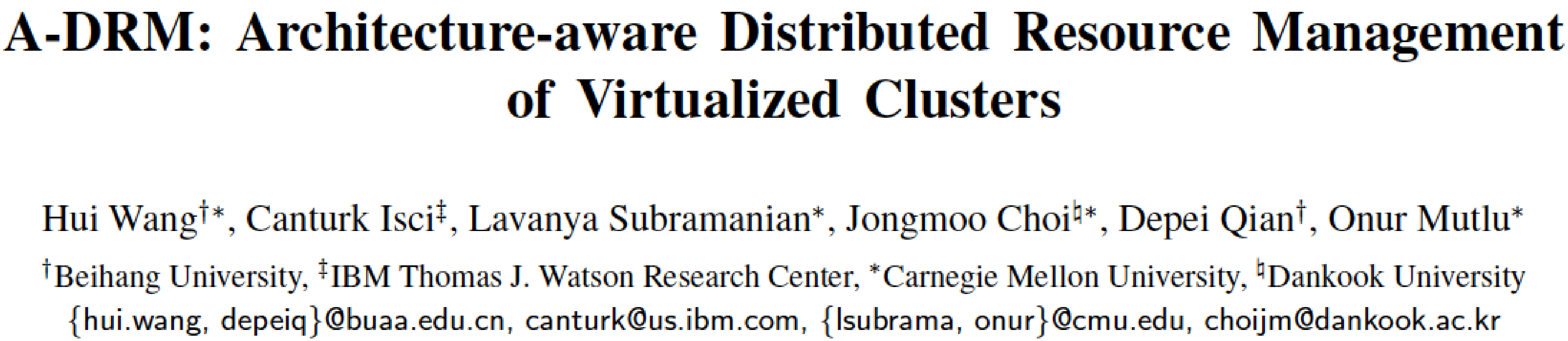
Architecture-aware Interference Detector

Architecture-aware Distributed Resource Management (Policy)

Migration Engine

More on Architecture-Aware DRM

- Hui Wang, Canturk Isci, Lavanya Subramanian, Jongmoo Choi, Depei Qian, and Onur Mutlu,
  "A-DRM: Architecture-aware Distributed Resource Management of Virtualized Clusters"
  *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (**VEE**)*, Istanbul, Turkey, March 2015.
  [Slides (pptx) (pdf)]



A-DRM: Architecture-aware Distributed Resource Management of Virtualized Clusters

Hui Wang[†*], Canturk Isci[‡], Lavanya Subramanian[*], Jongmoo Choi[♮*], Depei Qian[†], Onur Mutlu[*]

[†]Beihang University, [‡]IBM Thomas J. Watson Research Center, [*]Carnegie Mellon University, [♮]Dankook University

{hui.wang, depeiq}@buaa.edu.cn, canturk@us.ibm.com, {lsubrama, onur}@cmu.edu, choijm@dankook.ac.kr

Interference-Aware Thread Scheduling

- Advantages

  + Can eliminate/minimize interference by scheduling "symbiotic applications" together (as opposed to just managing the interference)

  + Less intrusive to hardware (less need to modify the hardware resources)

- Disadvantages and Limitations

  -- High overhead to migrate threads and data between cores and machines

  -- Does not work (well) if all threads are similar and they interfere

Summary: Fundamental Interference Control Techniques

- Goal: to reduce/control interference

1. Prioritization or request scheduling

2. Data mapping to banks/channels/ranks

3. Core/source throttling

4. Application/thread scheduling

Best is to combine all. How would you do that?

Summary: Memory QoS Approaches and Techniques

- Approaches: Smart vs. dumb resources
  - Smart resources: QoS-aware memory scheduling
  - Dumb resources: Source throttling; channel partitioning
  - Both approaches are effective in reducing interference
  - No single best approach for all workloads

- Techniques: Request/thread scheduling, source throttling, memory partitioning
  - All approaches are effective in reducing interference
  - Can be applied at different levels: hardware vs. software
  - No single best technique for all workloads

- Combined approaches and techniques are the most powerful
  - Integrated Memory Channel Partitioning and Scheduling [MICRO'11]

MCP Micro 2011 Talk

Summary: Memory Interference and QoS

- QoS-unaware memory → uncontrollable and unpredictable system

- Providing QoS awareness improves performance, predictability, fairness, and utilization of the memory system

- Discussed many new techniques to:
  - Minimize memory interference
  - Provide predictable performance

- Many new research ideas needed for integrated techniques and closing the interaction with software

What Did We Not Cover?

- Prefetch-aware shared resource management
- DRAM-controller-cache co-design
- Cache interference management
- Interconnect interference management
- Write-read scheduling
- …

Some Other Ideas …

Decoupled DMA w/ Dual-Port DRAM

[PACT 2015]

Isolating CPU and IO Traffic by Leveraging a Dual-Data-Port DRAM

Decoupled Direct Memory Access

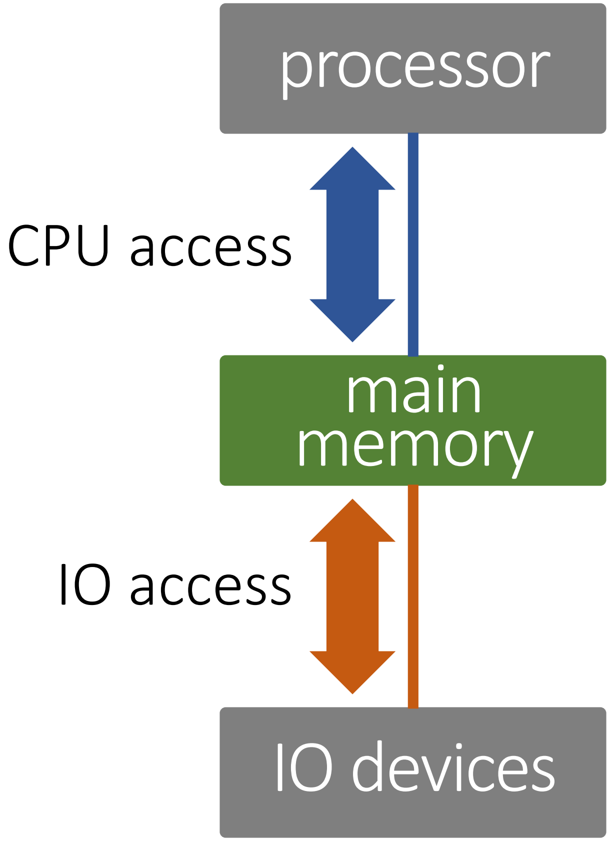
Donghyuk Lee

Lavanya Subramanian, Rachata Ausavarungnirun,
Jongmoo Choi, Onur Mutlu

SAFARI

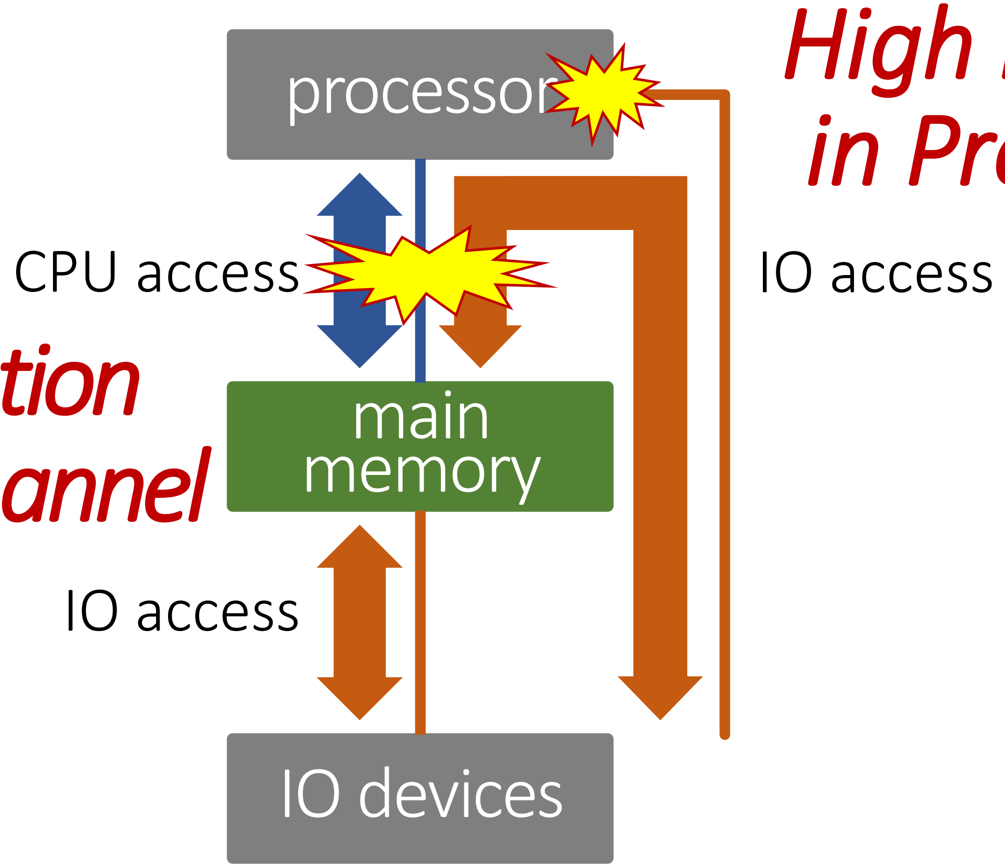
Carnegie Mellon

Logical System Organization



Main memory connects processor and IO devices as an ***intermediate layer***

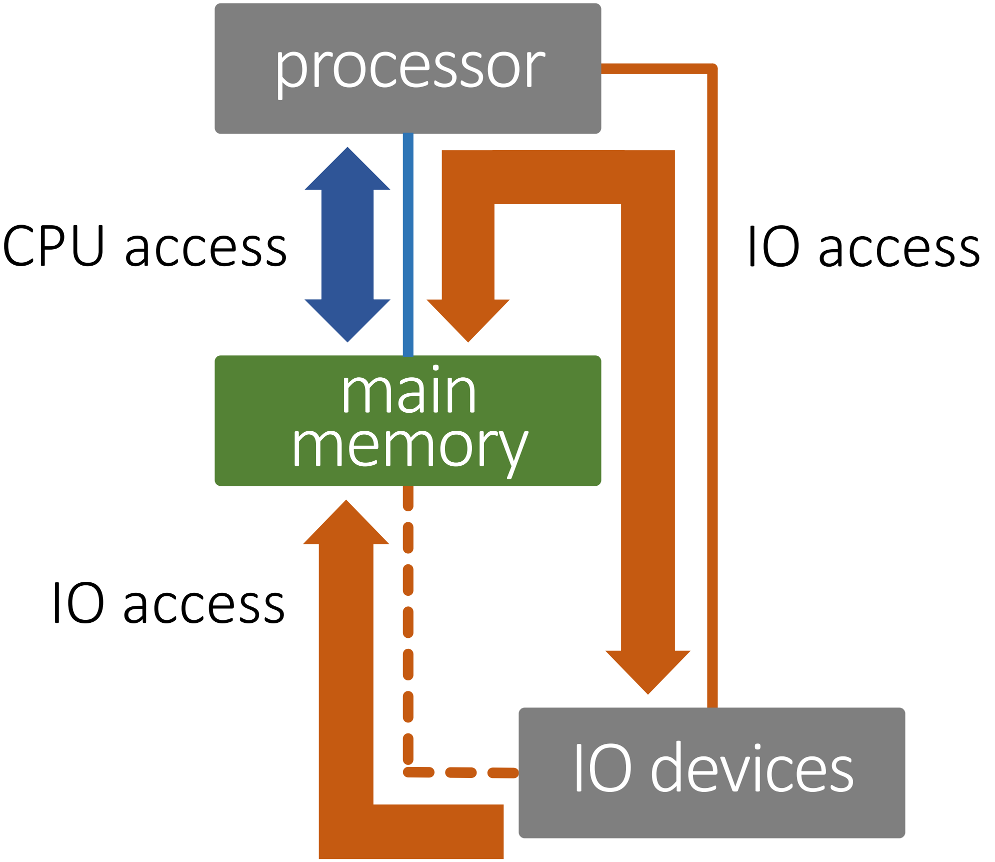
Physical System Implementation



High Pin Cost in Processor

High Contention in Memory Channel

Our Approach



Enabling IO channel,
decoupled & *isolated* from CPU channel

Executive Summary

- Problem
  - CPU and IO accesses contend for the shared memory channel
- Our Approach: *Decoupled Direct Memory Access (DDMA)*
  - Design new DRAM architecture with two independent data ports → *Dual-Data-Port DRAM*
  - Connect one port to CPU and the other port to IO devices → *Decouple CPU and IO accesses*
- Application
  - Communication between compute units (e.g., CPU – GPU)
  - In-memory communication (e.g., bulk in-memory copy/init.)
  - Memory-storage communication (e.g., page fault, IO prefetch)
- Result
  - Significant *performance improvement* (20% in 2 ch. & 2 rank system)
  - *CPU pin count reduction* (4.5%)

Outline

1. Problem

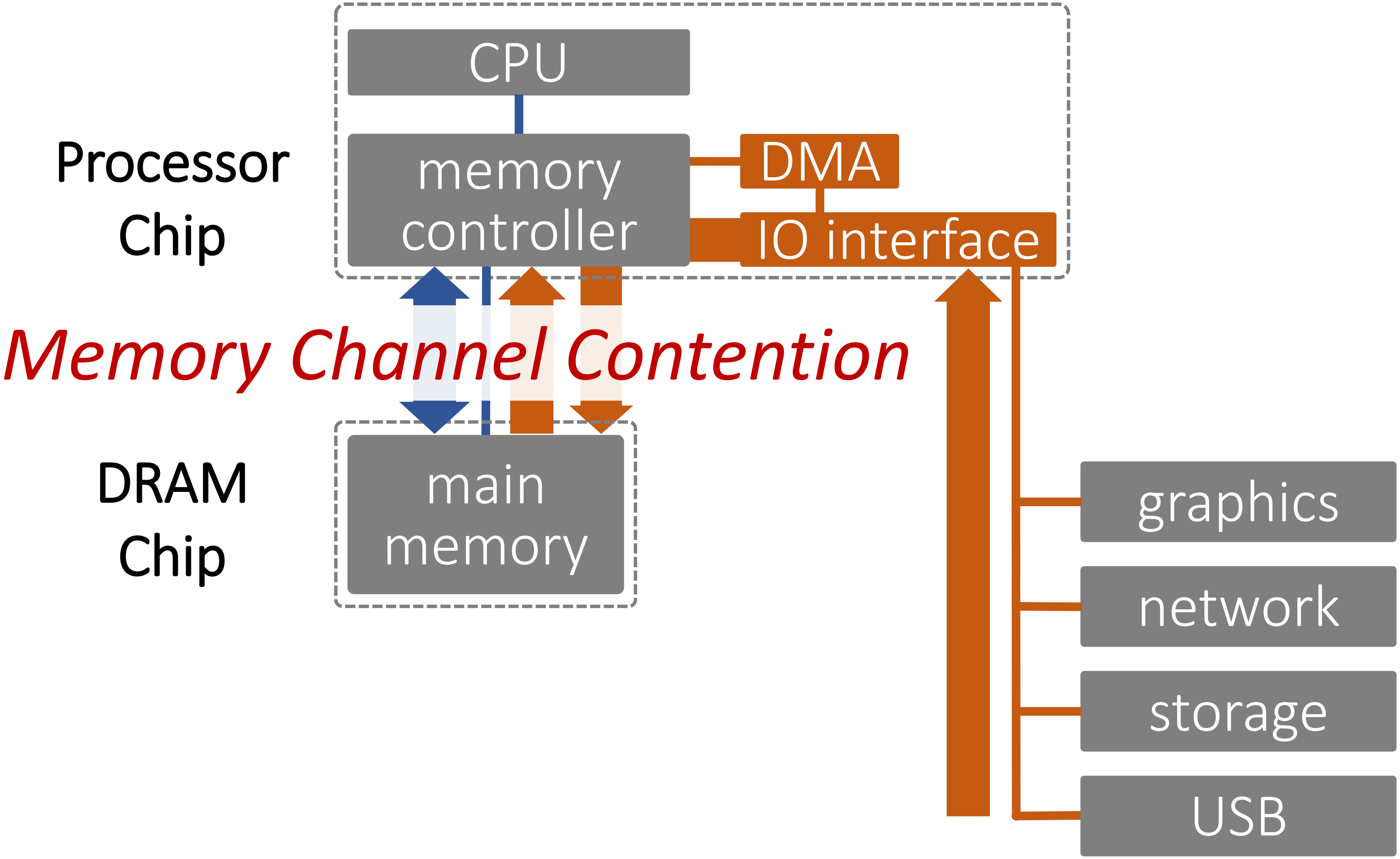
2. Our Approach

3. Dual-Data-Port DRAM

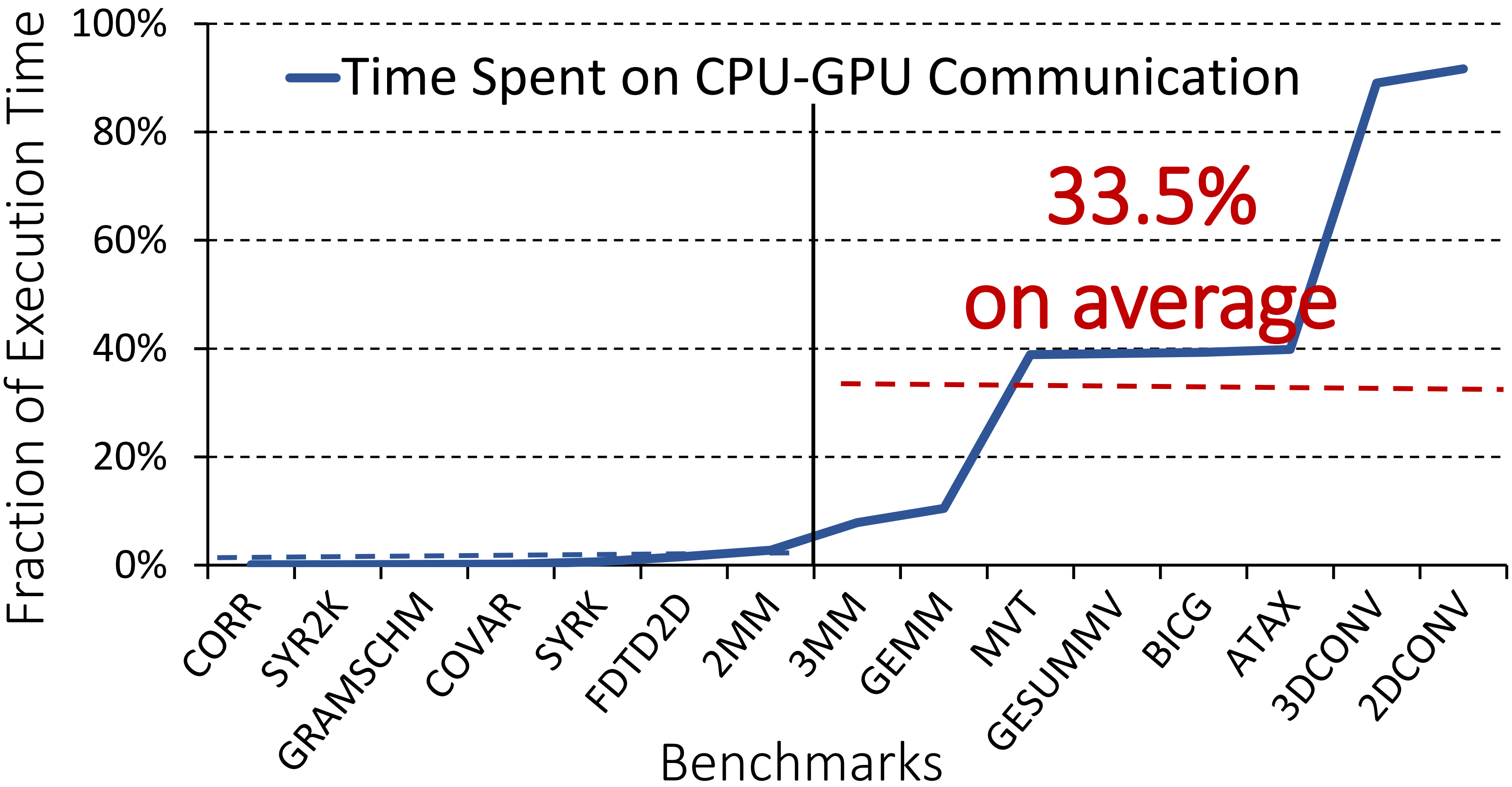
4. Applications for DDMA

5. Evaluation

Problem 1: Memory Channel Contention

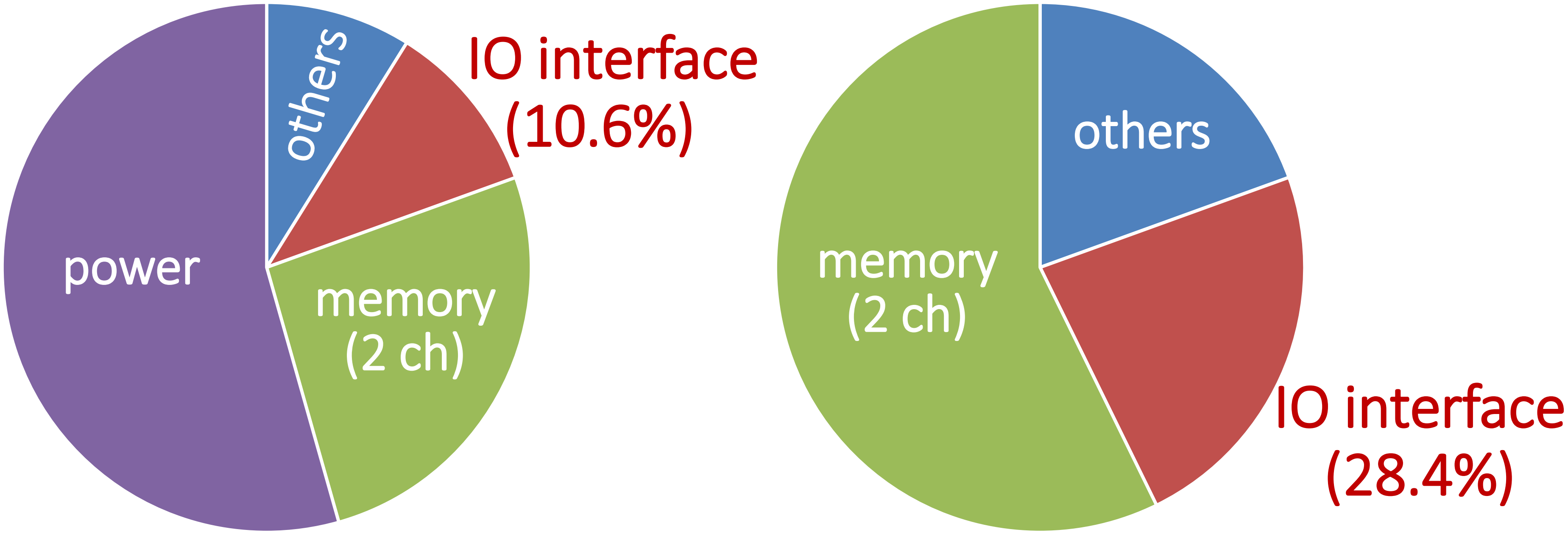


Problem 1: Memory Channel Contention



A large fraction of the execution time is spent on IO accesses

Problem 2: High Cost for IO Interfaces



959 pins in total

Processor Pin Count
(w/ power pins)

359 pins in total

Processor Pin Count
(w/o power pins)

Integrating IO interface on the processor chip leads to ***high area cost***

Shared Memory Channel

- Memory channel contention for IO access and CPU access

- High area cost for integrating IO interfaces on processor chip

Outline

1. Problem

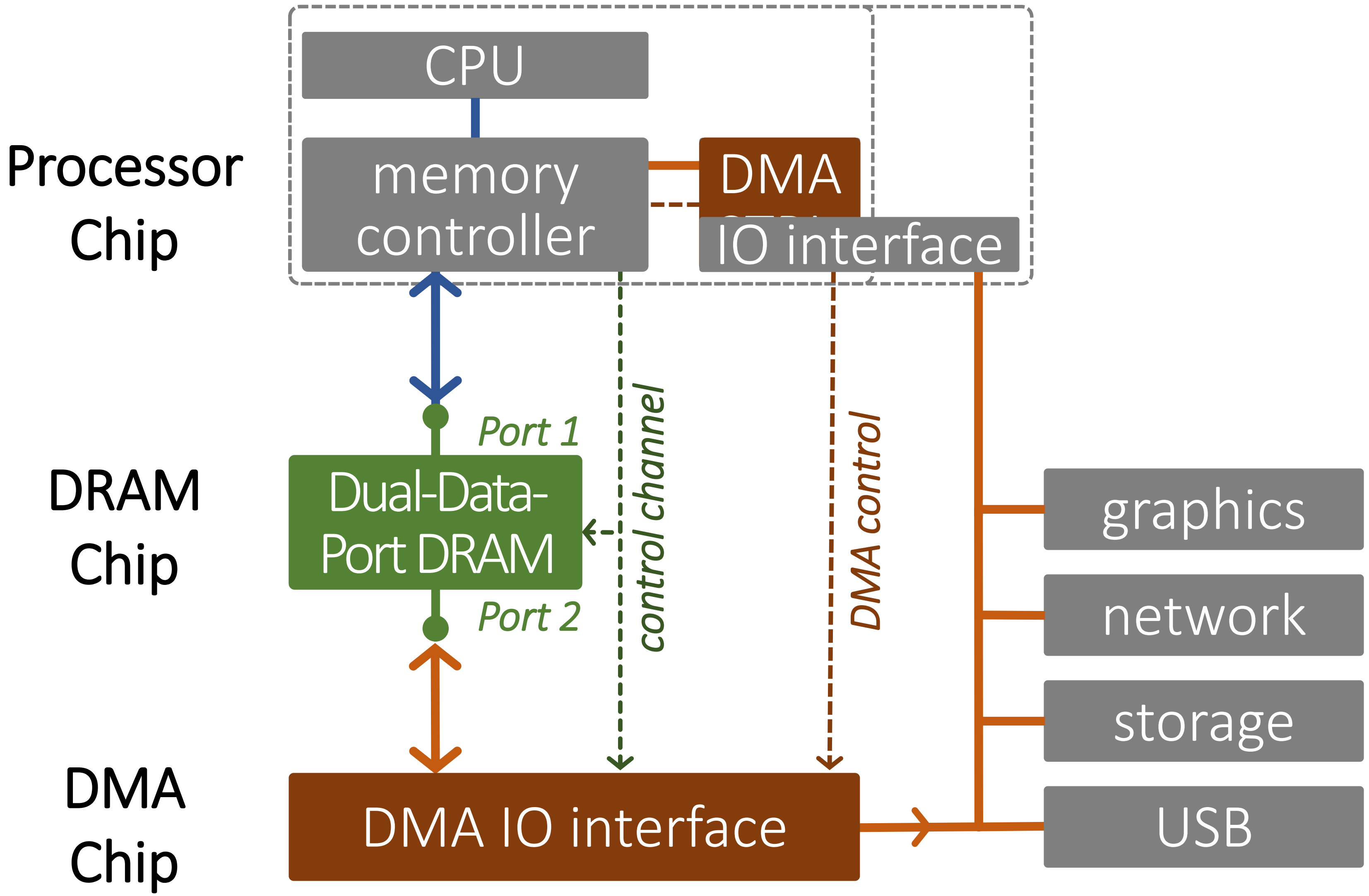
2. Our Approach

3. Dual-Data-Port DRAM

4. Applications for DDMA

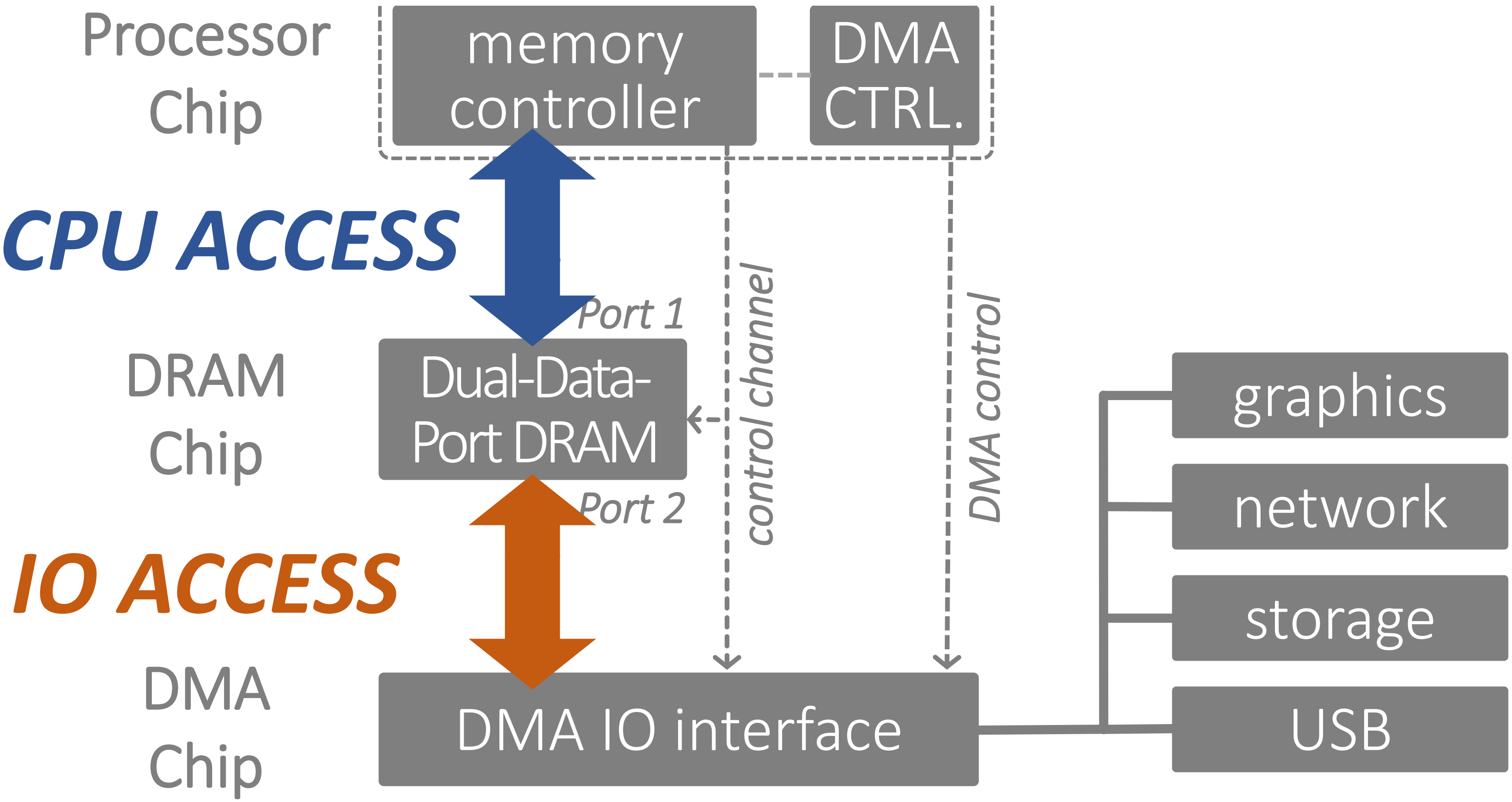
5. Evaluation

Our Approach



Our Approach

Decoupled Direct Memory Access



Outline

1. Problem

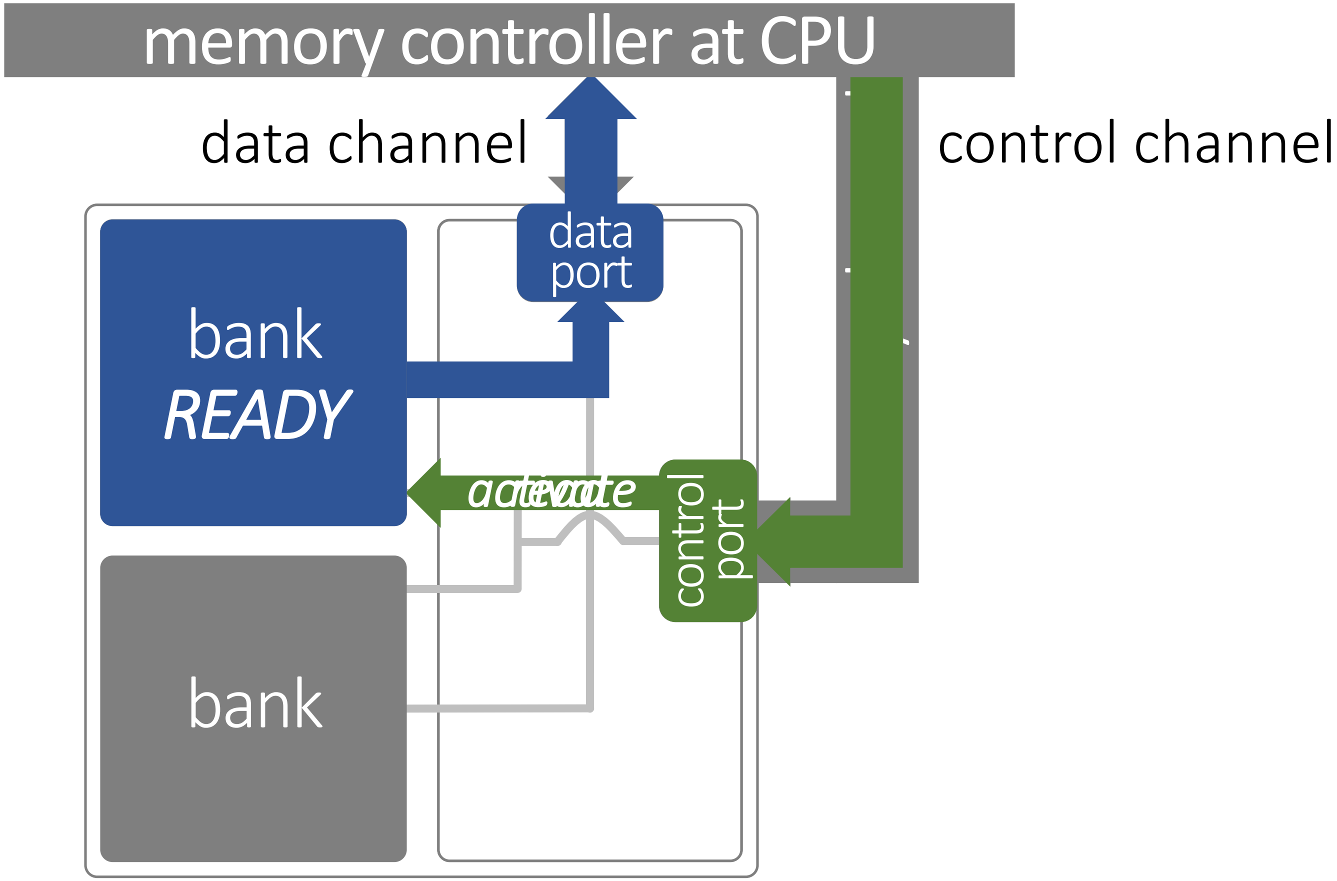
2. Our Approach

3. Dual-Data-Port DRAM

4. Applications for DDMA

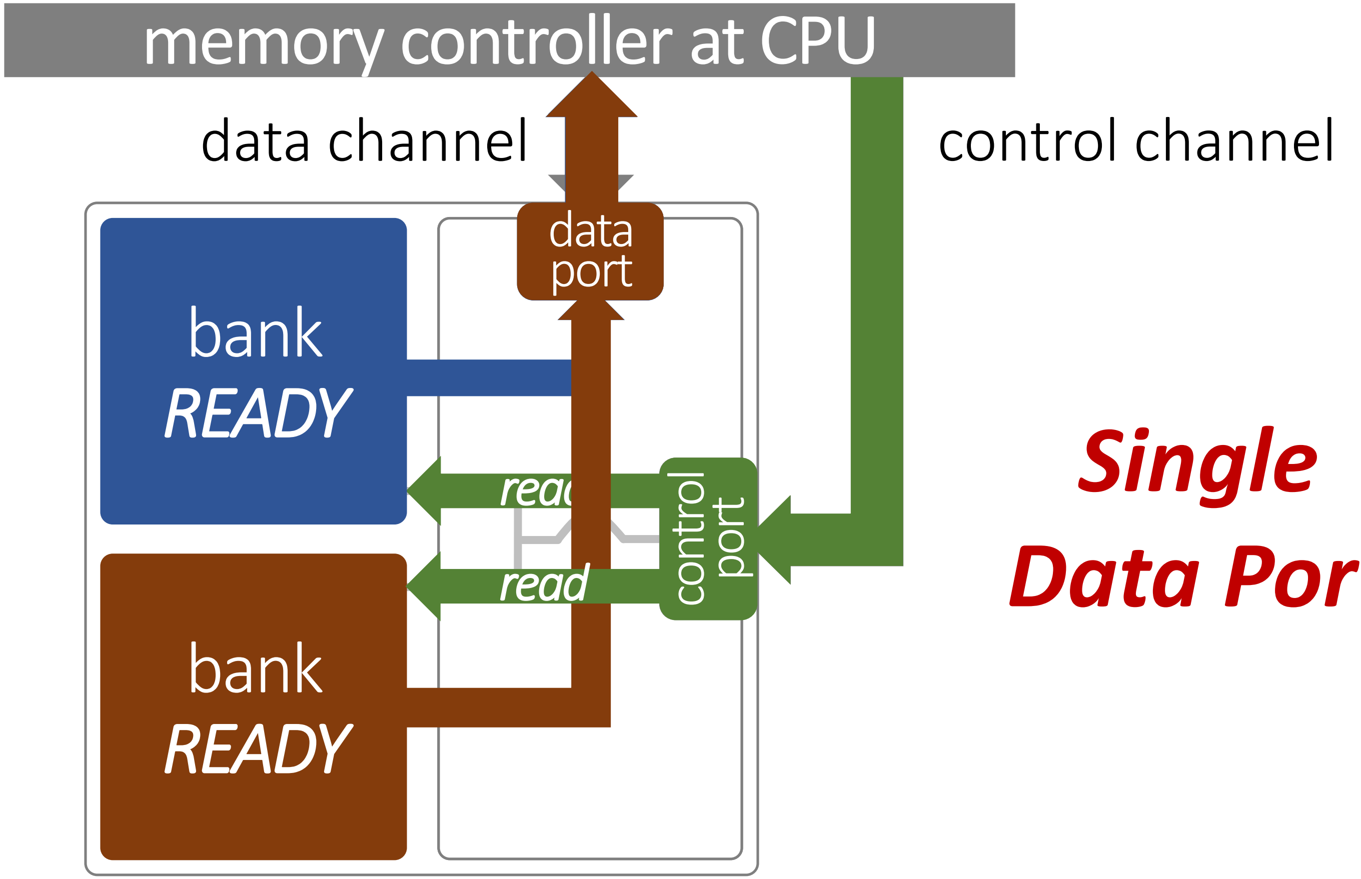
5. Evaluation

Background: DRAM Operation



DRAM peripheral logic: *i) controls banks*, and *ii) transfers data* over memory channel

Problem: Single Data Port

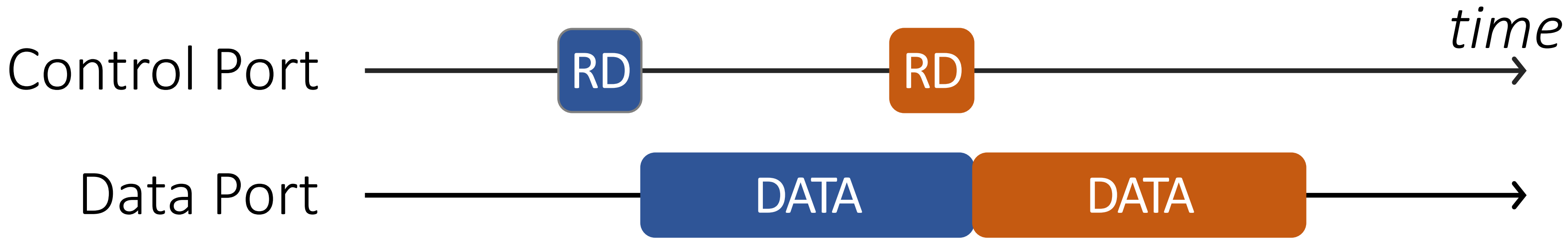


Many Banks

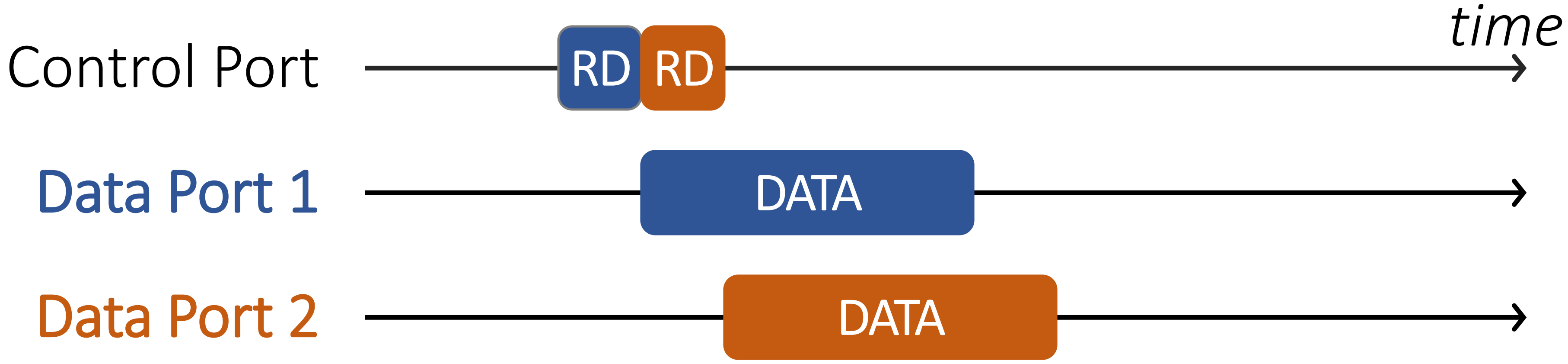
Single Data Port

Requests are served *serially*
due to ***single data port***

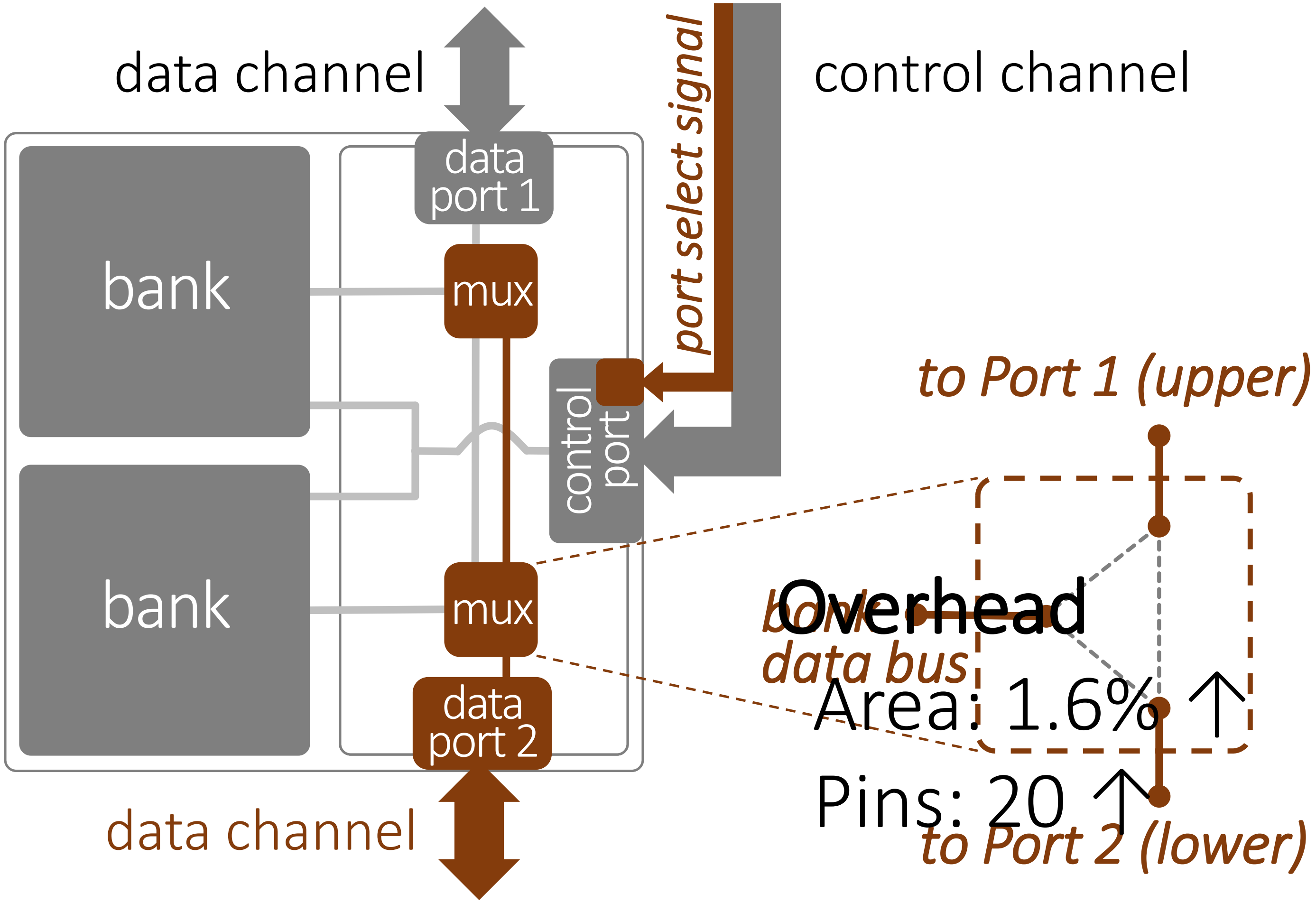
Problem: Single Data Port



What about a DRAM with **two data ports**?



Dual-Data-Port DRAM



Overhead

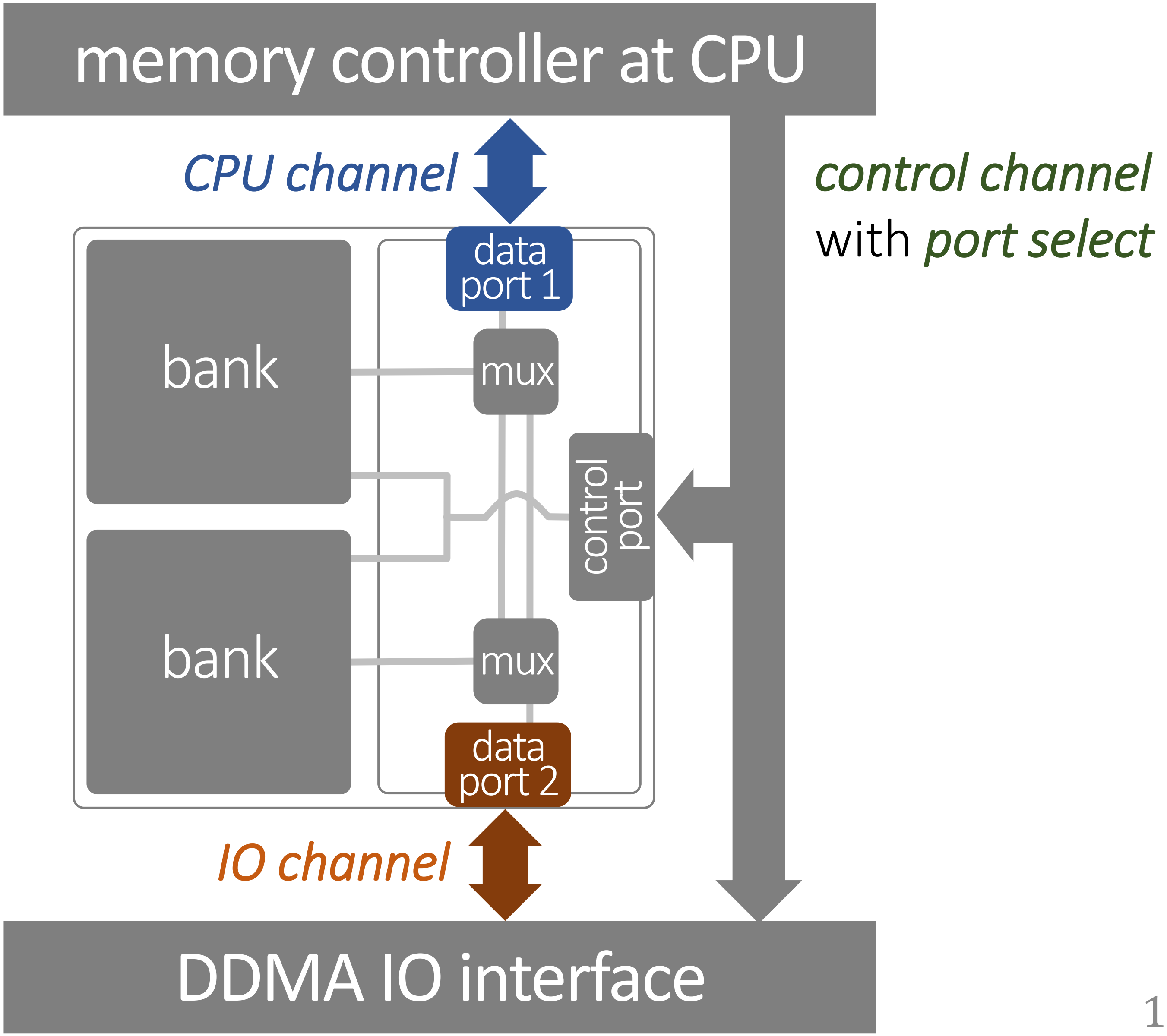
Area: 1.6% ↑

Pins: 20 ↑

twice the bandwidth & *independent data ports*

with low overhead

DDP-DRAM Memory System



SAFARI

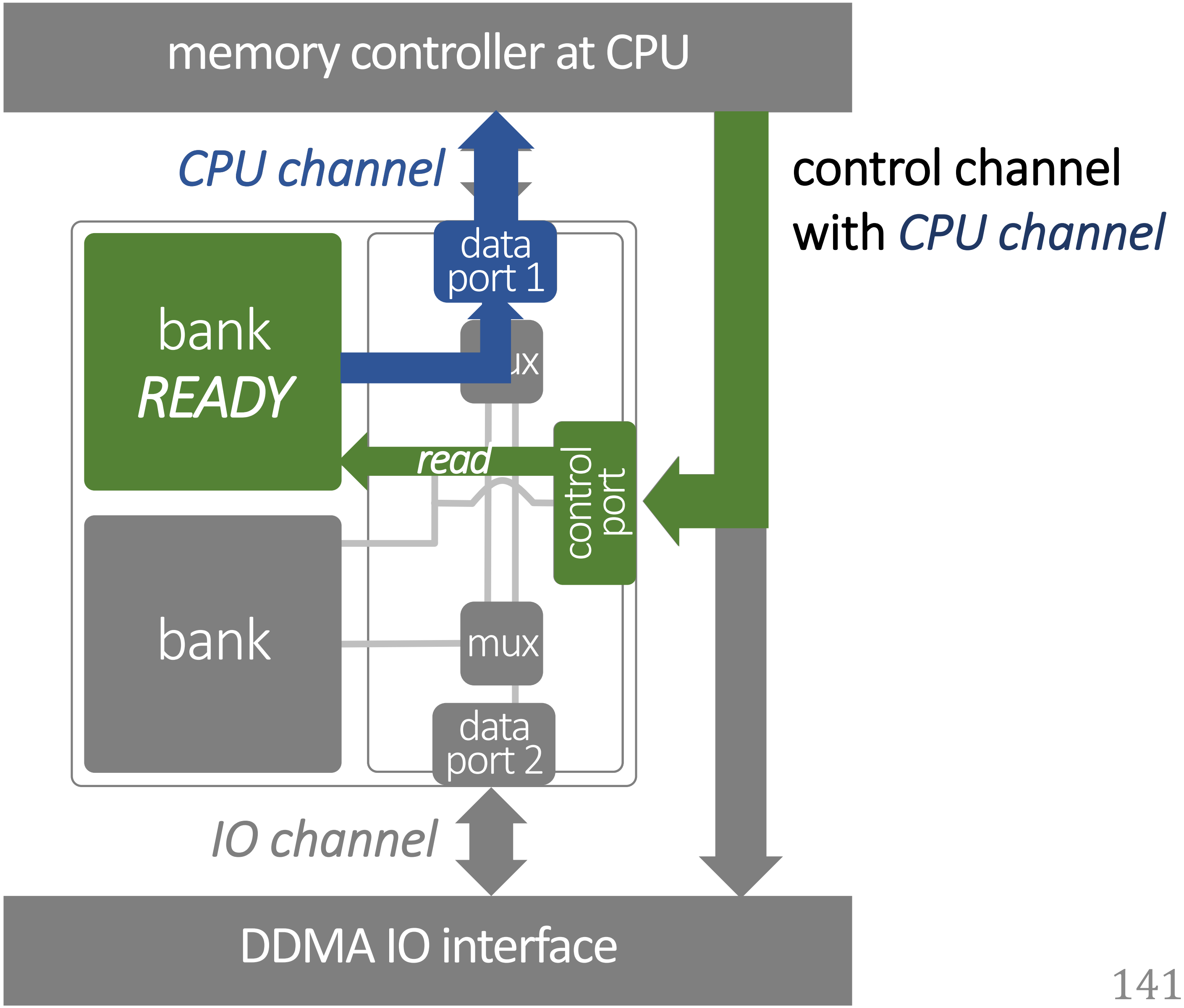
Three Data Transfer Modes

- **CPU Access**: Access through CPU channel
  - DRAM read/write with CPU port selection

- **IO Access**: Access through IO channel
  - DRAM read/write with IO port selection

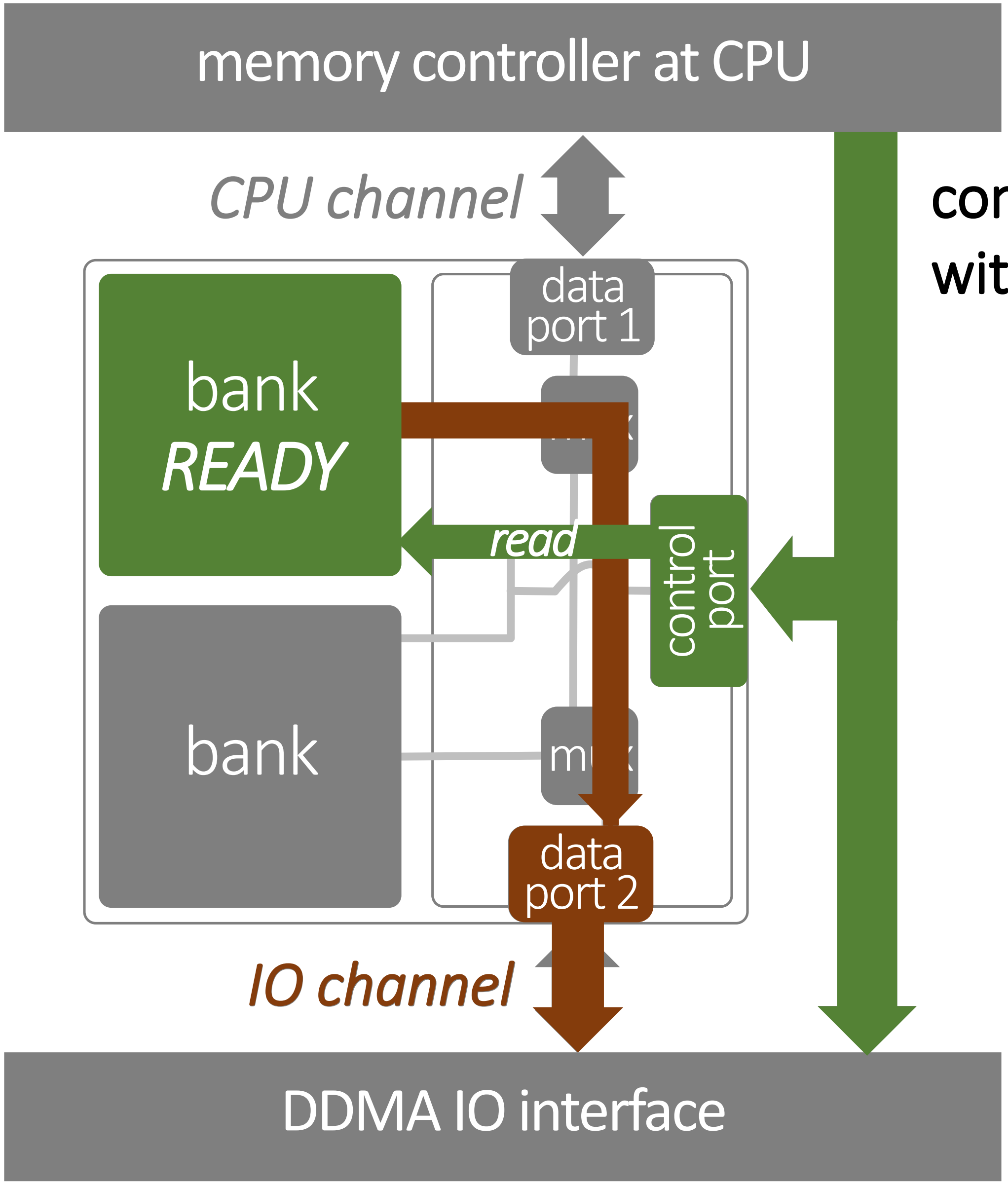
- **Port Bypass**: Direct transfer between channels
  - DRAM access with port bypass selection

1. CPU Access Mode



SAFARI

2. IO Access Mode

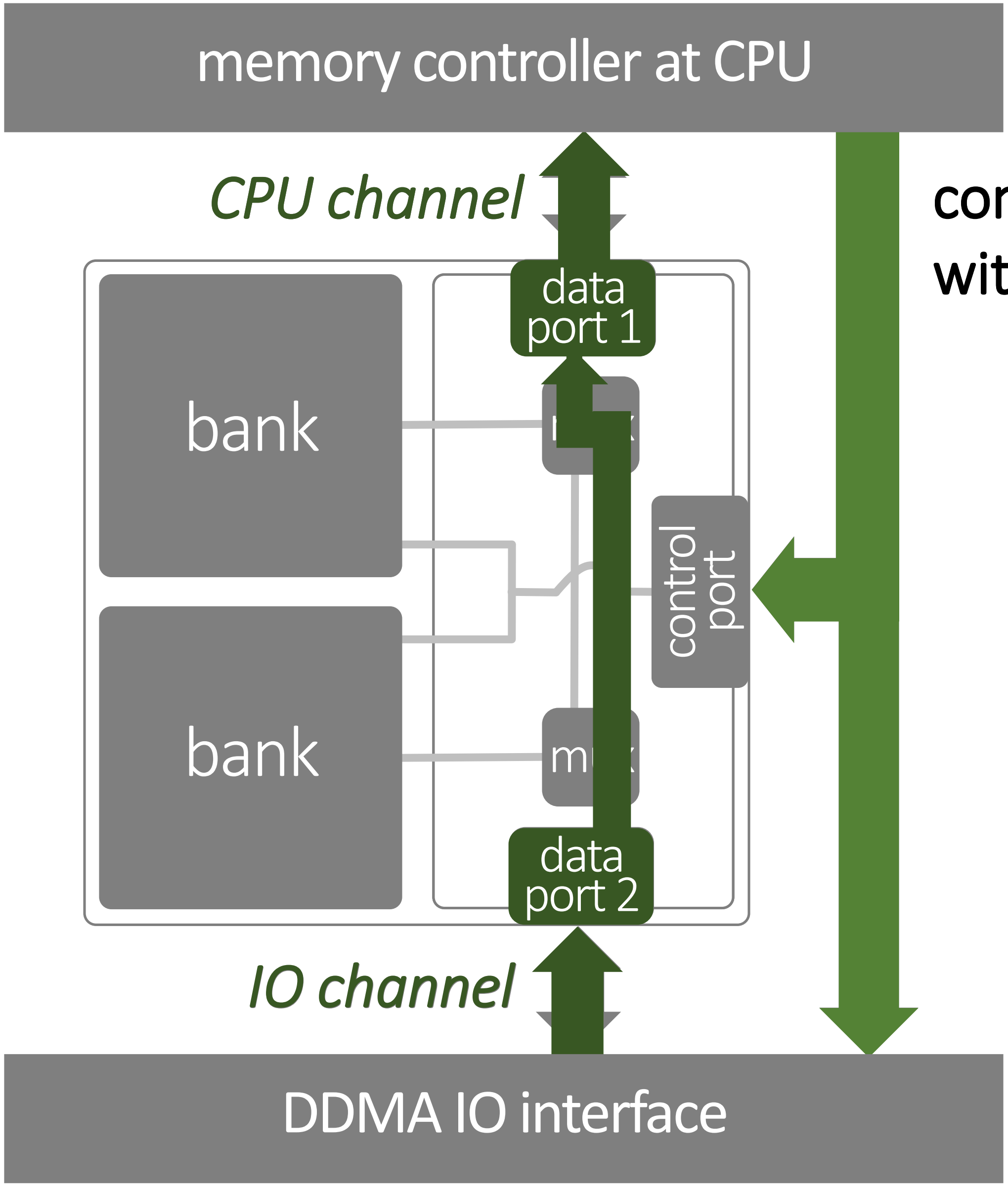


control channel
with port select

IO channel

SAFARI

3. Port Bypass Mode



control channel
with port select.
port bypass

Outline

1. Problem

2. Our Approach

3. Dual-Data-Port DRAM

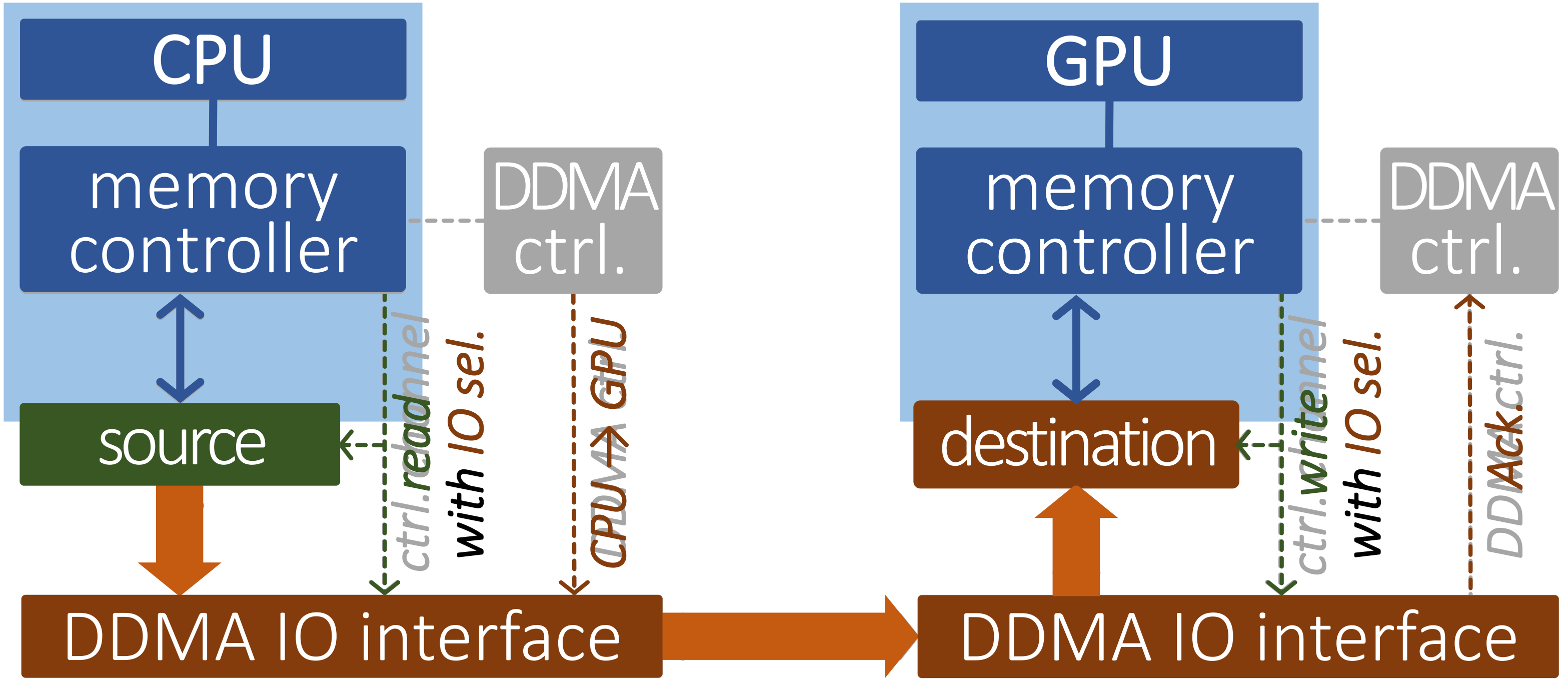
4. Applications for DDMA

5. Evaluation

Three Applications for DDMA

- Communication b/w Compute Units
  - CPU-GPU communication
- In-Memory Communication and Initialization
  - Bulk page copy/initialization
- Communication b/w Memory and Storage
  - Serving page fault/file read & write

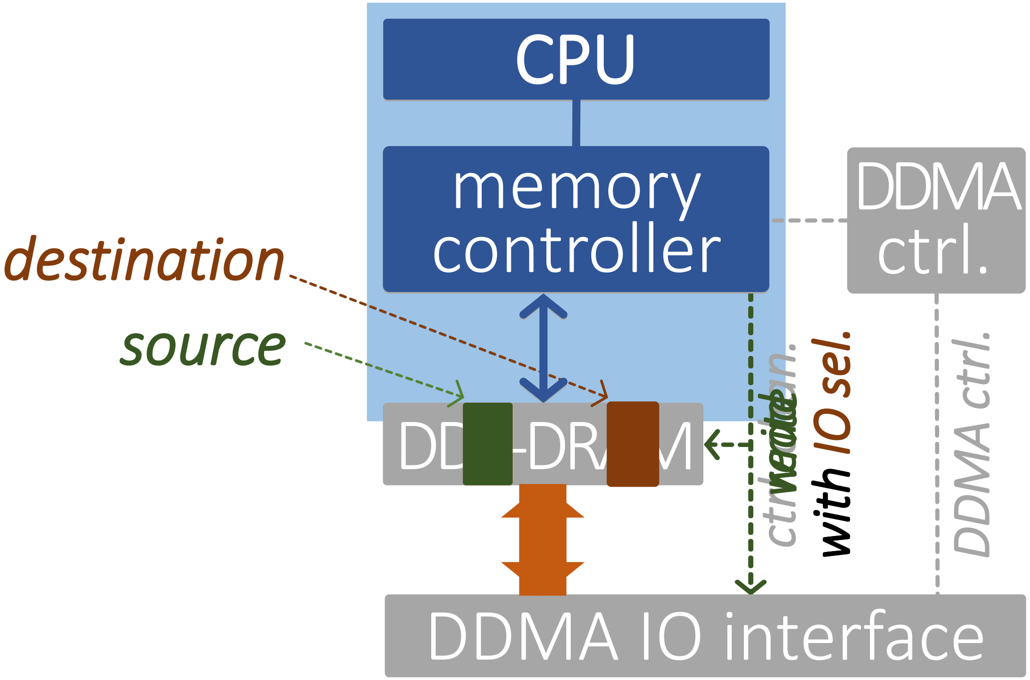
1. Compute Unit ↔ Compute Unit



Transfer data through DDMA

without interfering w/ CPU/GPU memory accesses

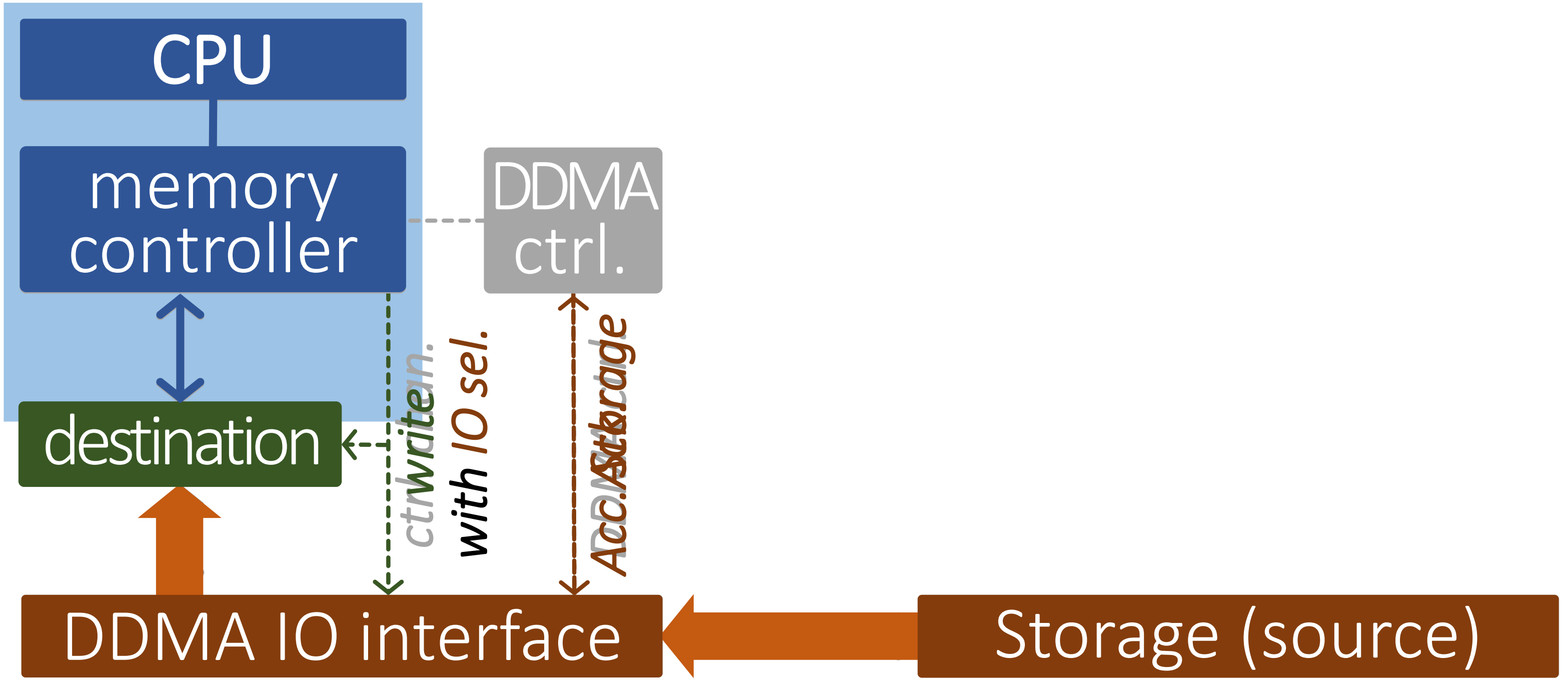
2. In-Memory Communication



Transfer data in DRAM through DDAM

without interfering with CPU memory accesses

3. Memory ↔ Storage



Transfer data from storage through DDMA
without interfering with CPU memory accesses

Outline

1. Problem

2. Our Approach

3. Dual-Data-Port DRAM

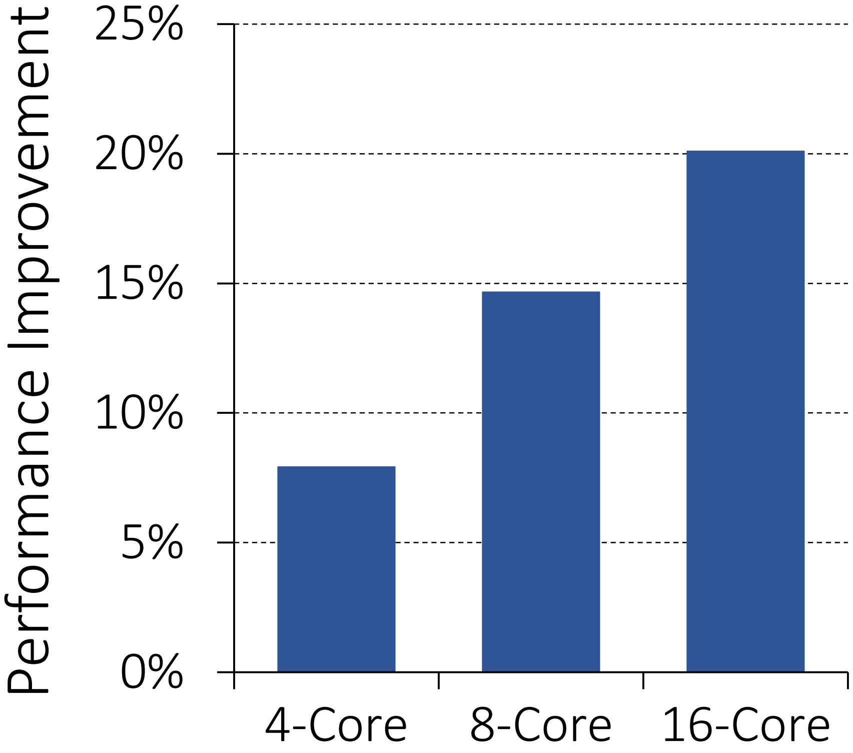
4. Applications for DDMA

5. Evaluation

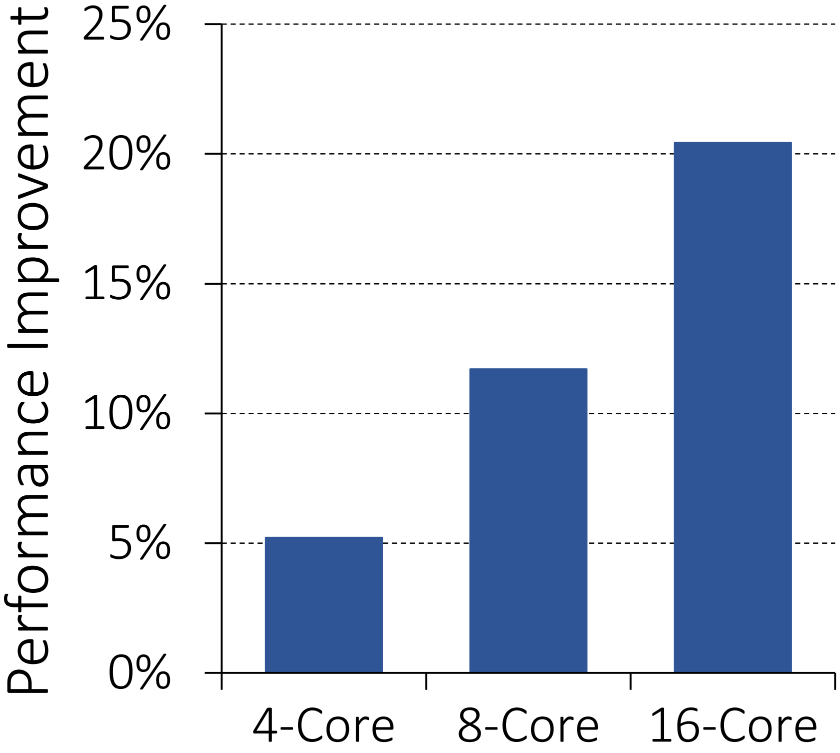
Evaluation Methods

- System
  - Processor: 4 – 16 cores
  - LLC: 16-way associative, 512KB private cache-slice/core
  - Memory: 1 – 4 ranks and 1 – 4 channels
- Workloads
  - **Memory intensive**:
    SPEC CPU2006, TPC, stream (31 benchmarks)
  - **CPU-GPU communication intensive**:
    polybench (8 benchmarks)
  - **In-memory communication intensive**:
    apache, bootup, compiler, filecopy, mysql, fork, shell, memcached (8 in total)

Performance (2 Channel, 2 Rank)



CPU-GPU Comm.-Intensive

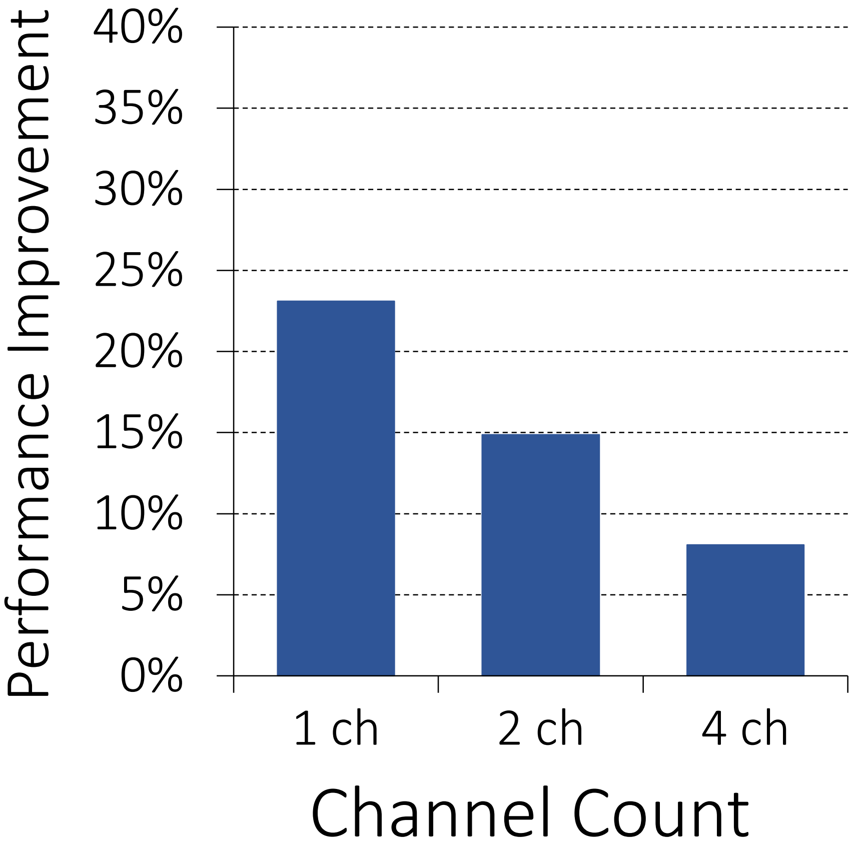


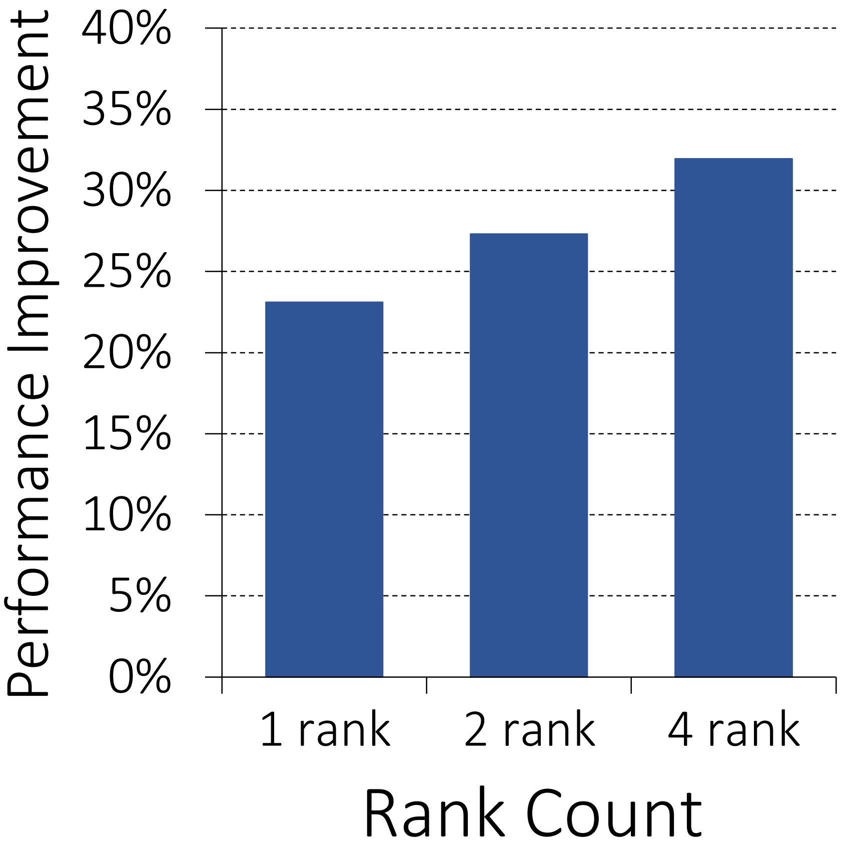
In-Memory Comm.-Intensive

High performance improvement

More performance improvement at ***higher core count***

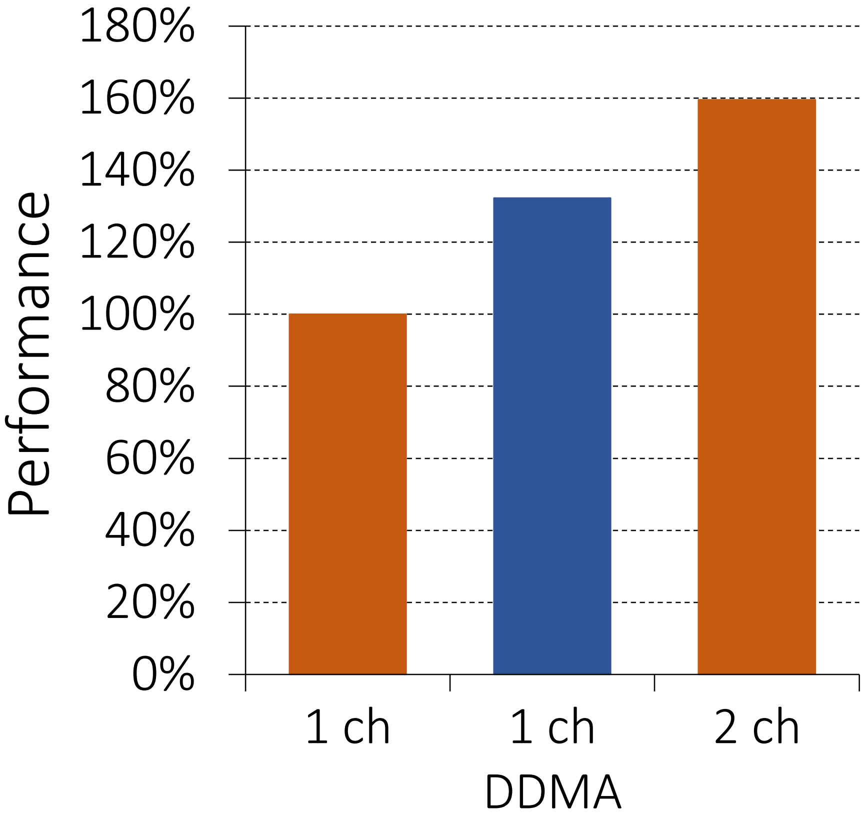
Performance on Various Systems

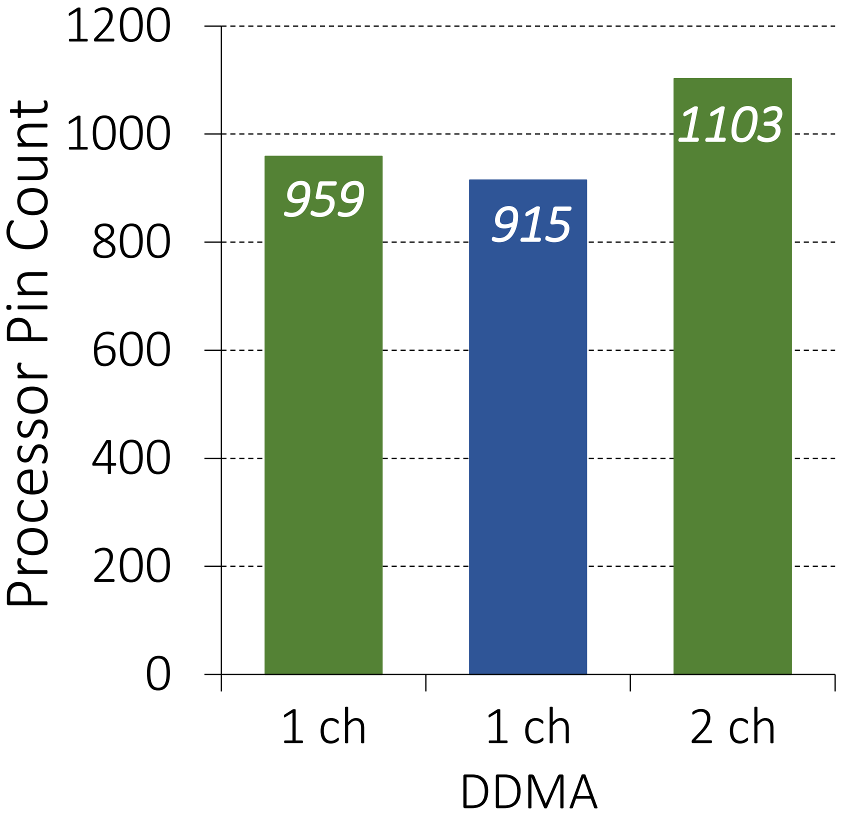




Performance increases with rank count

DDMA vs. Dual Channel



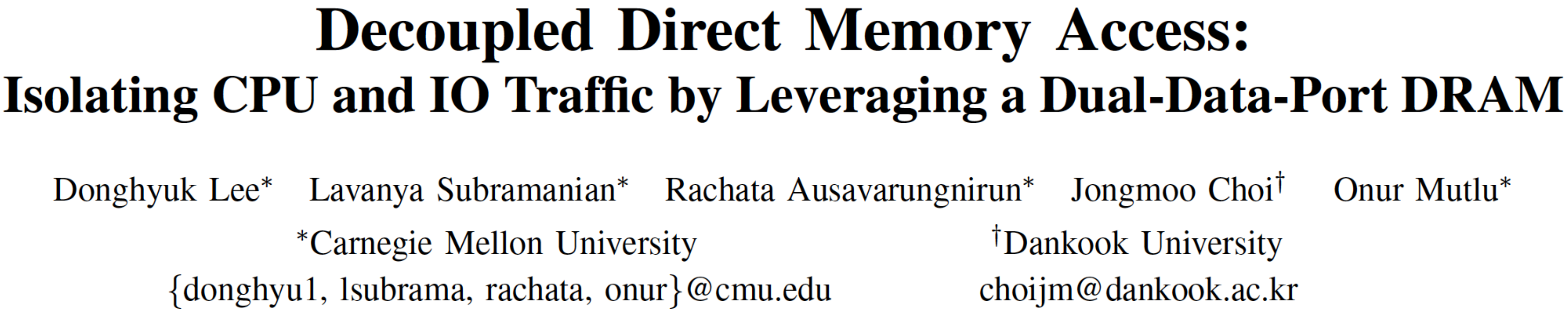


DDMA achieves *higher performance* at *lower processor pin count*

SAFARI

More on Decoupled DMA

- Donghyuk Lee, Lavanya Subramanian, Rachata Ausavarungnirun, Jongmoo Choi, and Onur Mutlu, **"Decoupled Direct Memory Access: Isolating CPU and IO Traffic by Leveraging a Dual-Data-Port DRAM"** *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques* (***PACT***), San Francisco, CA, USA, October 2015. [Slides (pptx) (pdf)]



Decoupled Direct Memory Access:
Isolating CPU and IO Traffic by Leveraging a Dual-Data-Port DRAM

Donghyuk Lee* Lavanya Subramanian* Rachata Ausavarungnirun* Jongmoo Choi† Onur Mutlu*
*Carnegie Mellon University †Dankook University
{donghyu1, lsubrama, rachata, onur}@cmu.edu choijm@dankook.ac.kr

Computer Architecture

Lecture 13: Memory Interference and Quality of Service (II)

Prof. Onur Mutlu

ETH Zürich

Fall 2017

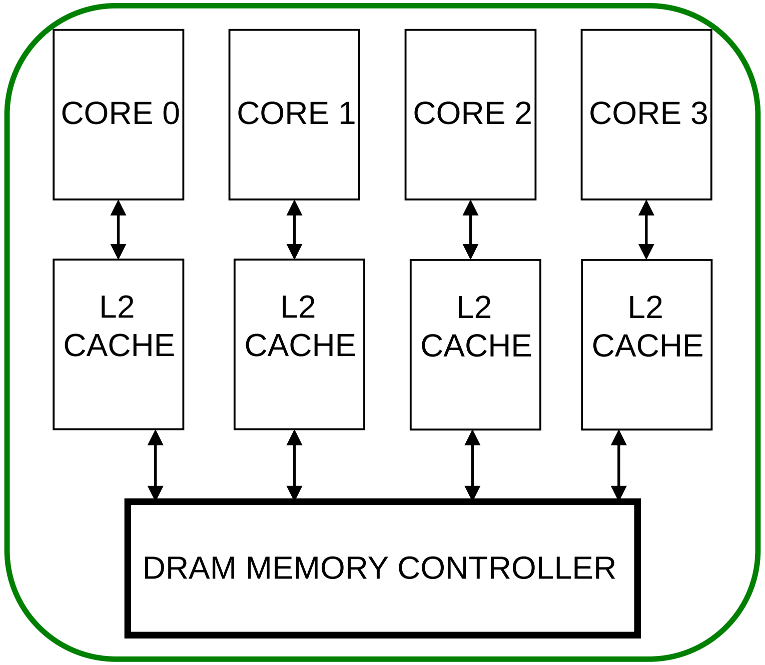
2 November 2017

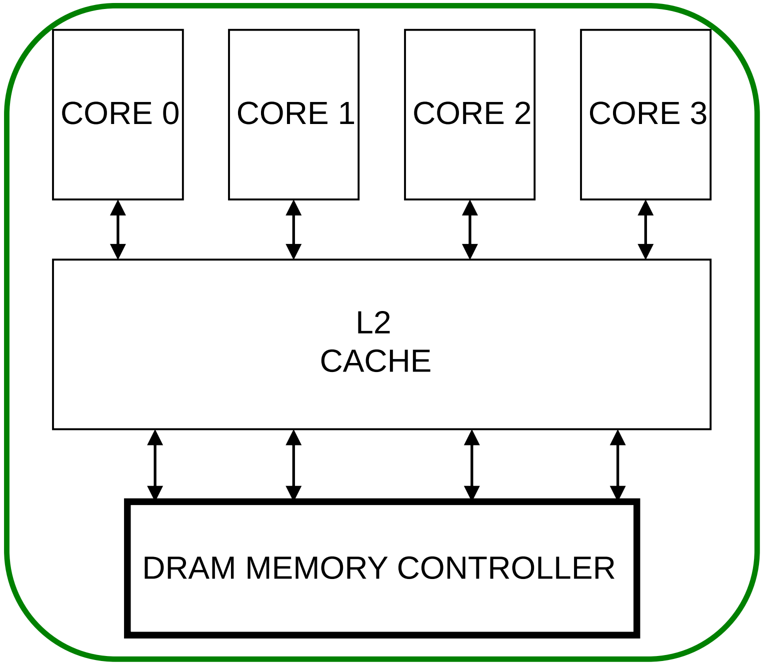
We did not cover the following slides in lecture. These are for your preparation for the next lecture.

Multi-Core Caching Issues

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.

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

- 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

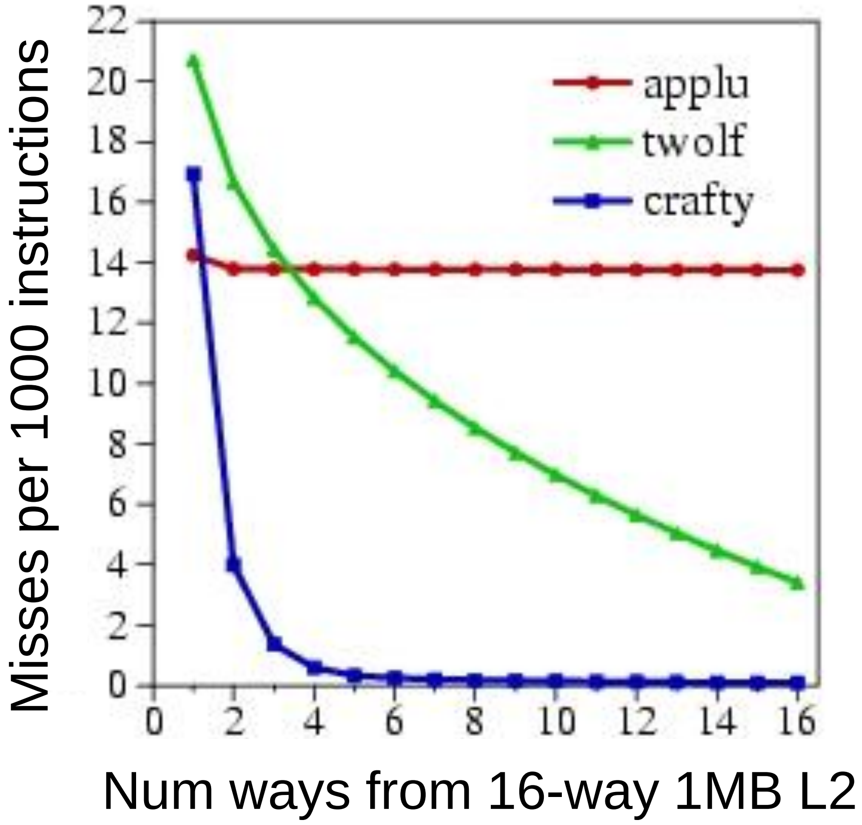
- Goal: Maximize system throughput
- Observation: Not all threads/applications benefit equally from caching → simple LRU replacement not good for system throughput
- Idea: Allocate more cache space to applications that obtain the most benefit from more space

- 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

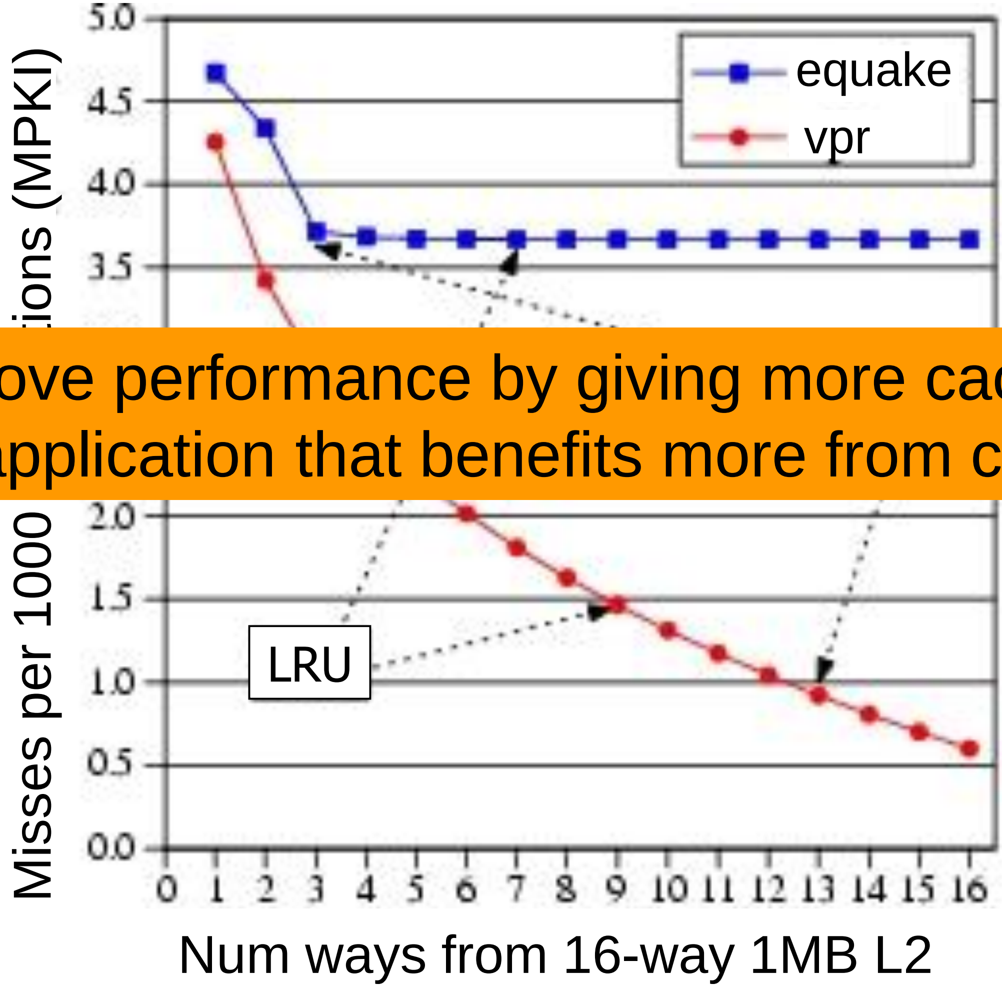


Low Utility

High Utility

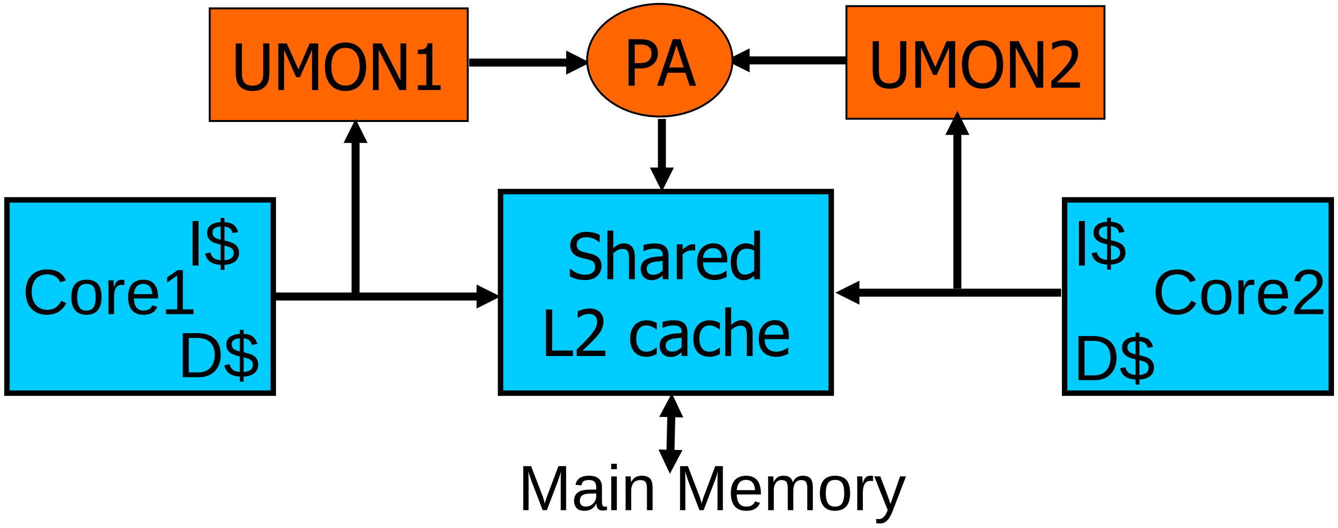
Saturating Utility

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)

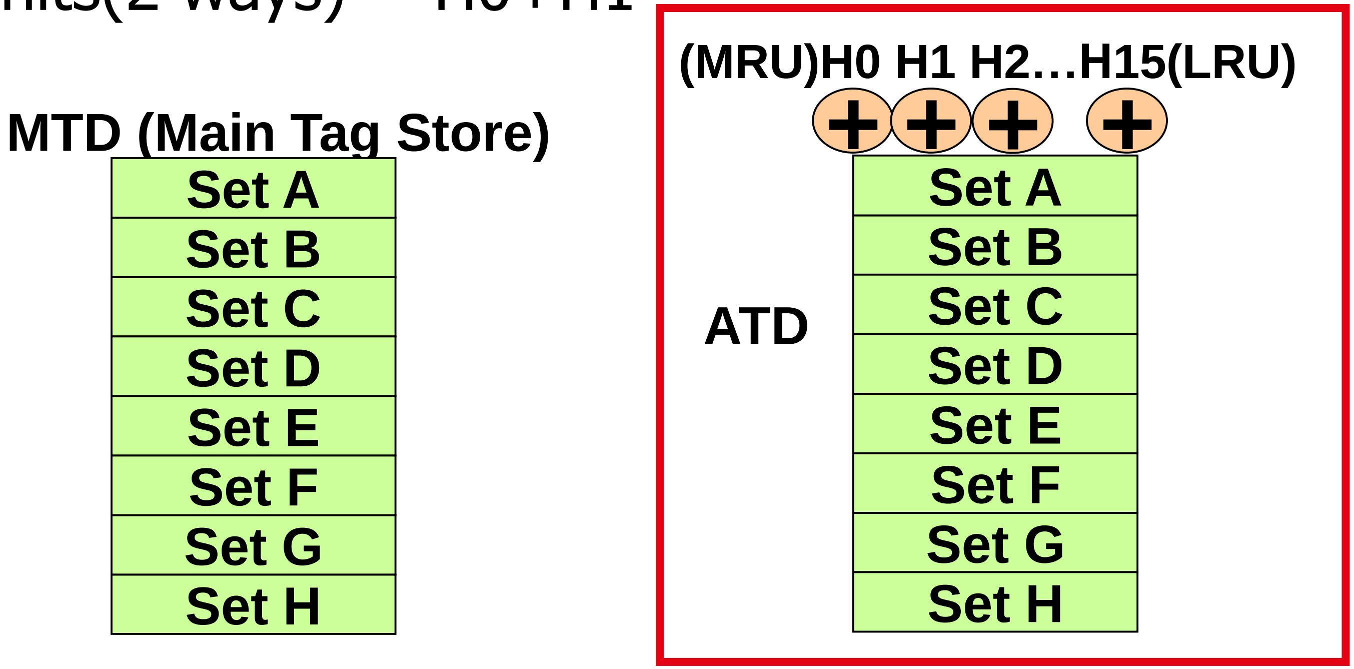


Three components:

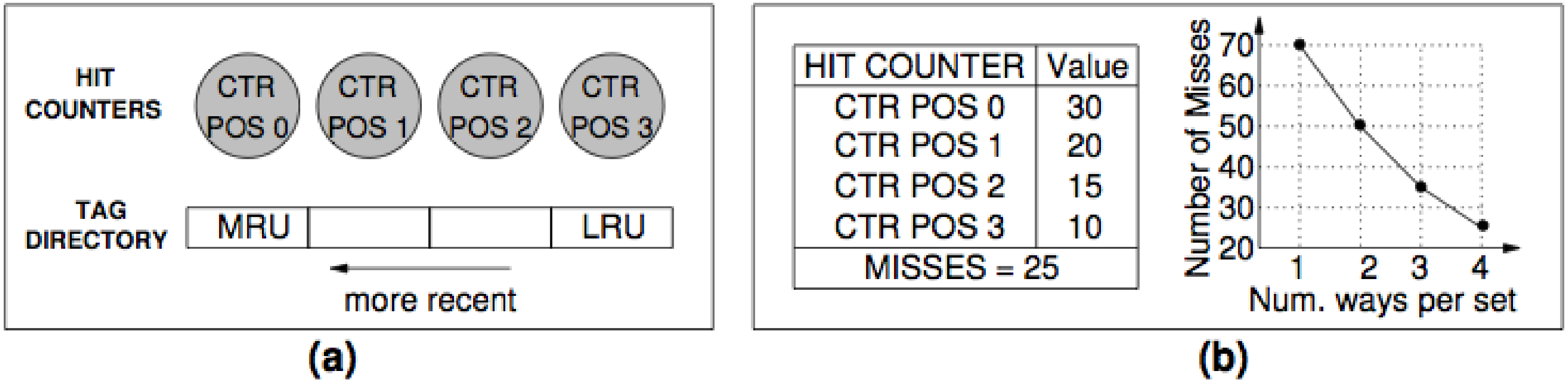
- ❑ Utility Monitors (UMON) per core
- ❑ Partitioning Algorithm (PA)
- ❑ Replacement support to enforce partitions

1. Utility Monitors

- For each core, simulate LRU policy using a separate tag store called ATD (auxiliary tag directory/store)
- Hit counters in ATD to count hits per recency position
- LRU is a stack algorithm: hit counts ➔ utility
  E.g. hits(2 ways) = H0+H1



Utility Monitors

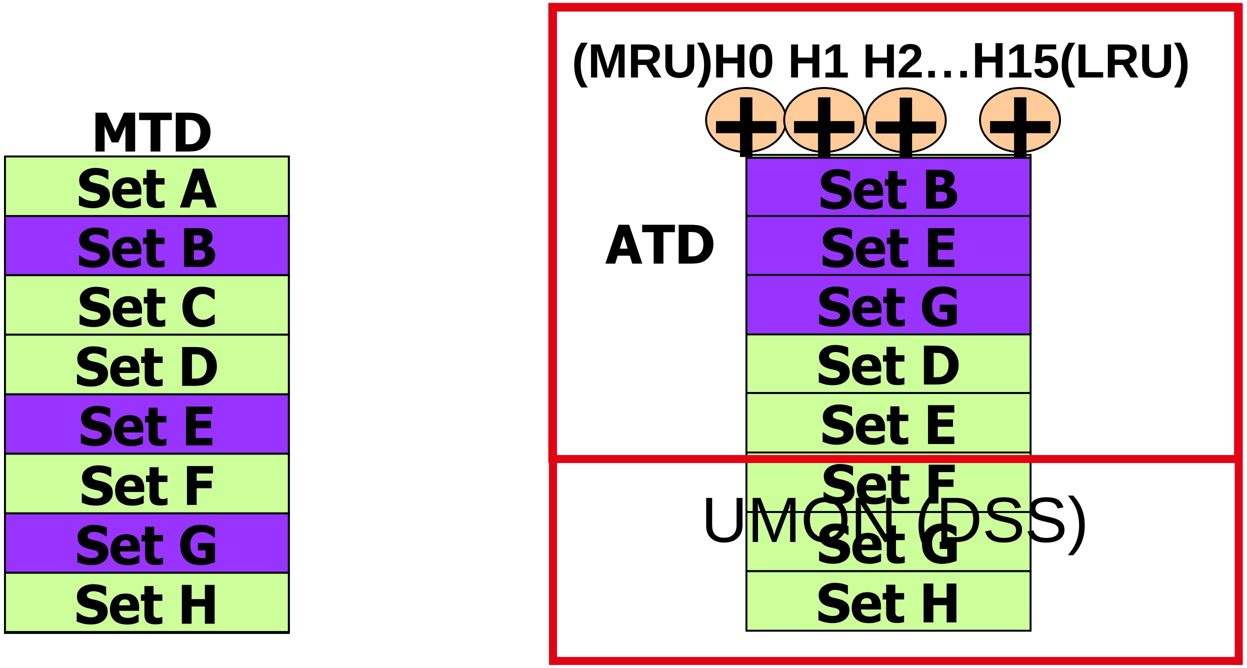


HIT COUNTER	Value
CTR POS 0	30
CTR POS 1	20
CTR POS 2	15
CTR POS 3	10
MISSES = 25	

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

Dynamic Set Sampling

- Extra tags incur hardware and power overhead
- 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.

2. Partitioning Algorithm

- Evaluate all possible partitions and select the best

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

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

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

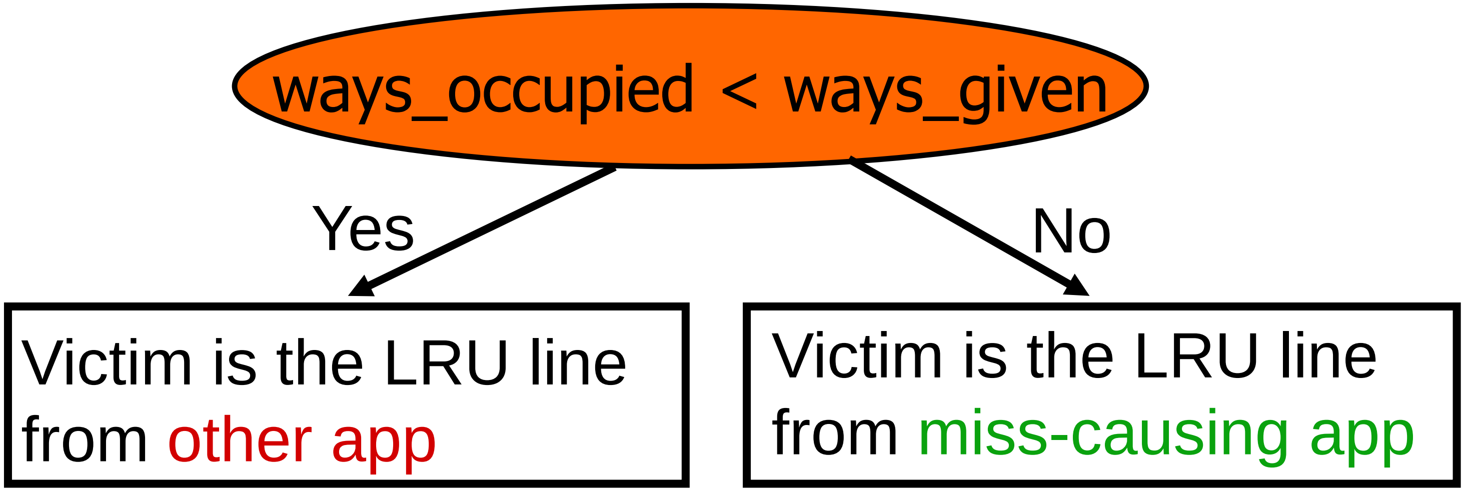
- Select a that maximizes ($\text{Hits}_{\text{core1}} + \text{Hits}_{\text{core2}}$)

- Partitioning done once every 5 million cycles

3. Enforcing Partitions: Way Partitioning

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

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


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

Performance Metrics

- Three metrics for performance:

1. Weighted Speedup (default metric)

   ➔ perf = $IPC_1/SingleIPC_1 + IPC_2/SingleIPC_2$
   ➔ correlates with reduction in execution time

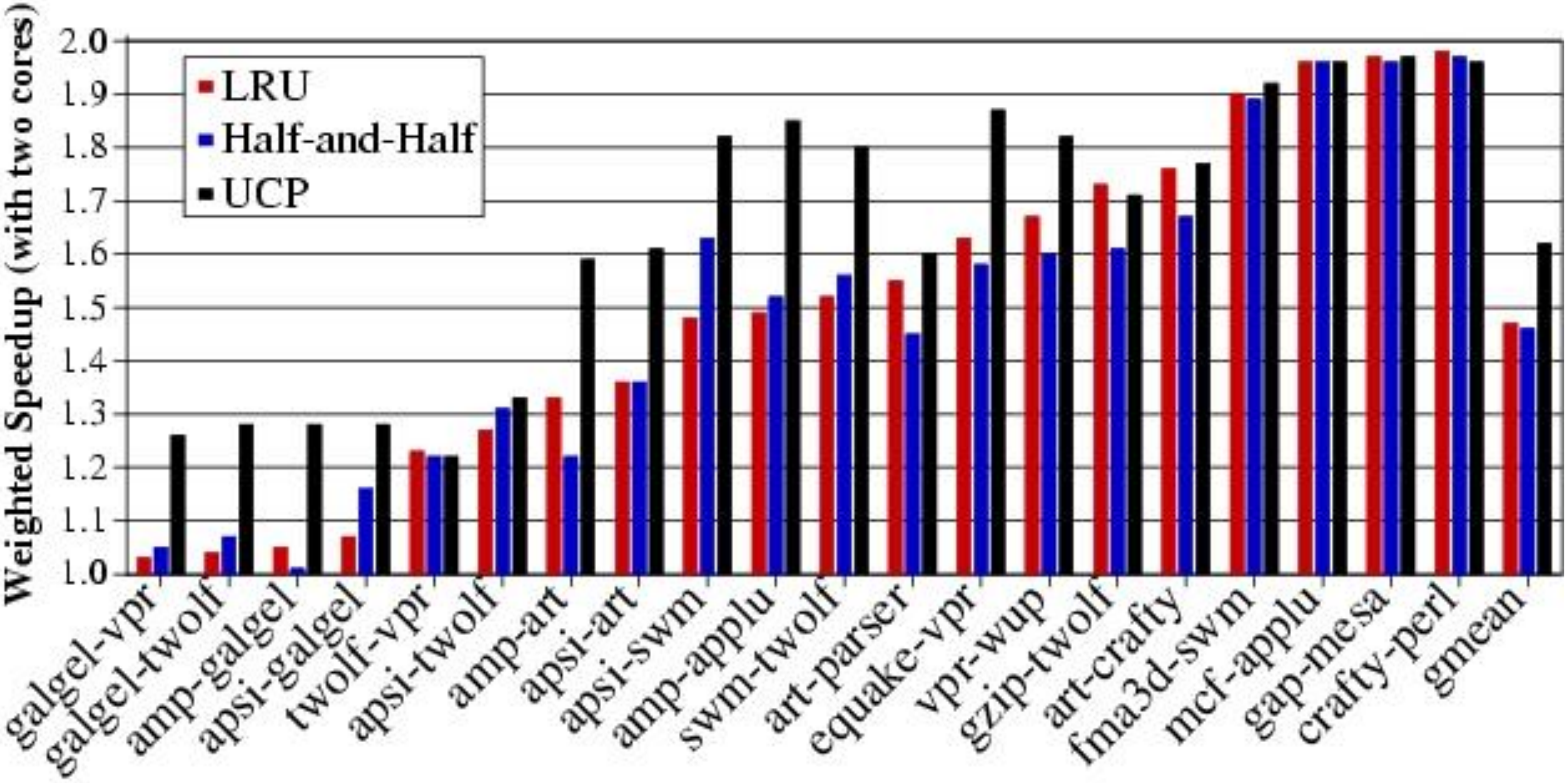
2. Throughput

   ➔ perf = $IPC_1 + IPC_2$
   ➔ can be unfair to low-IPC application

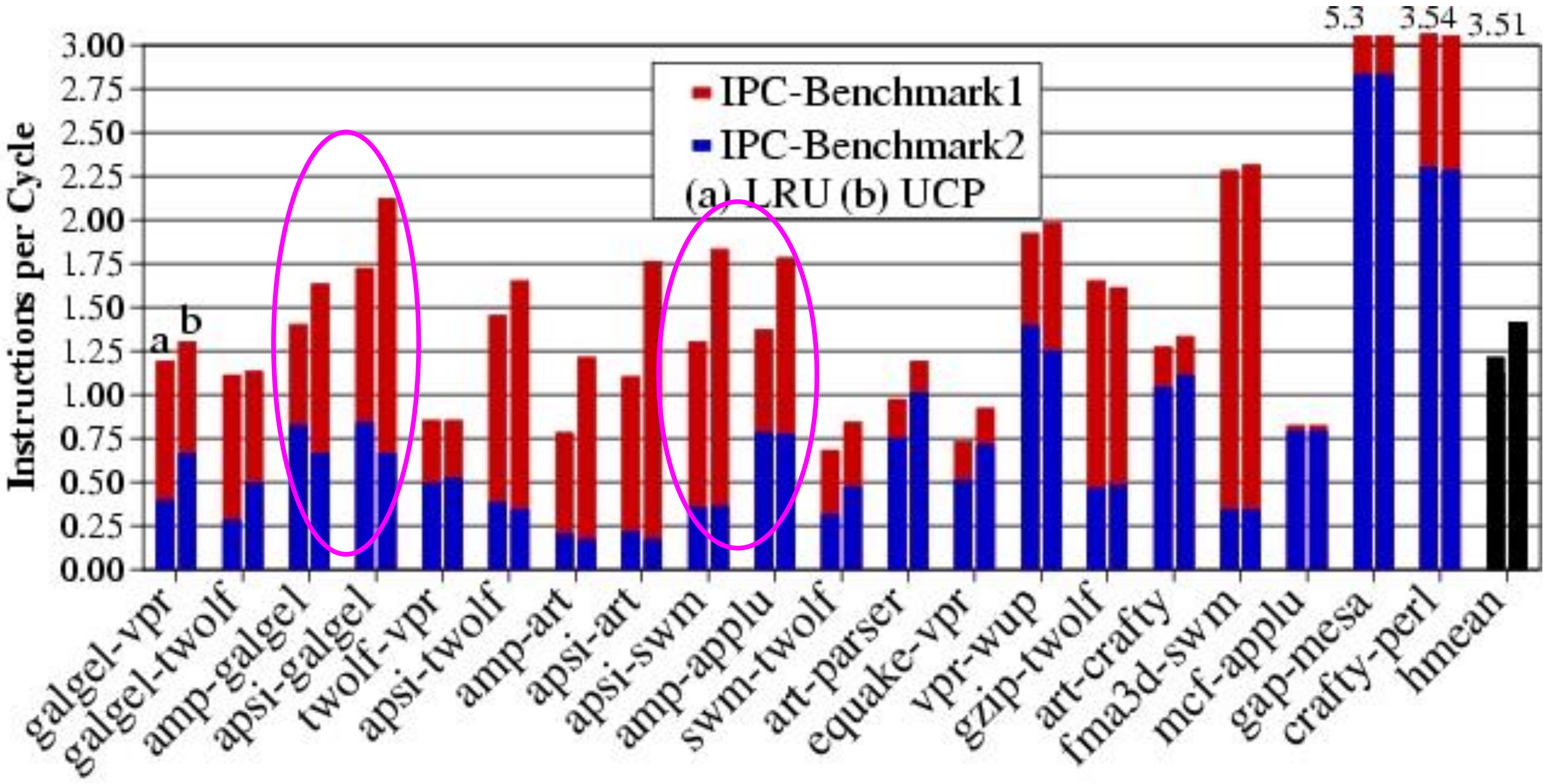
3. Hmean-fairness

   ➔ 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

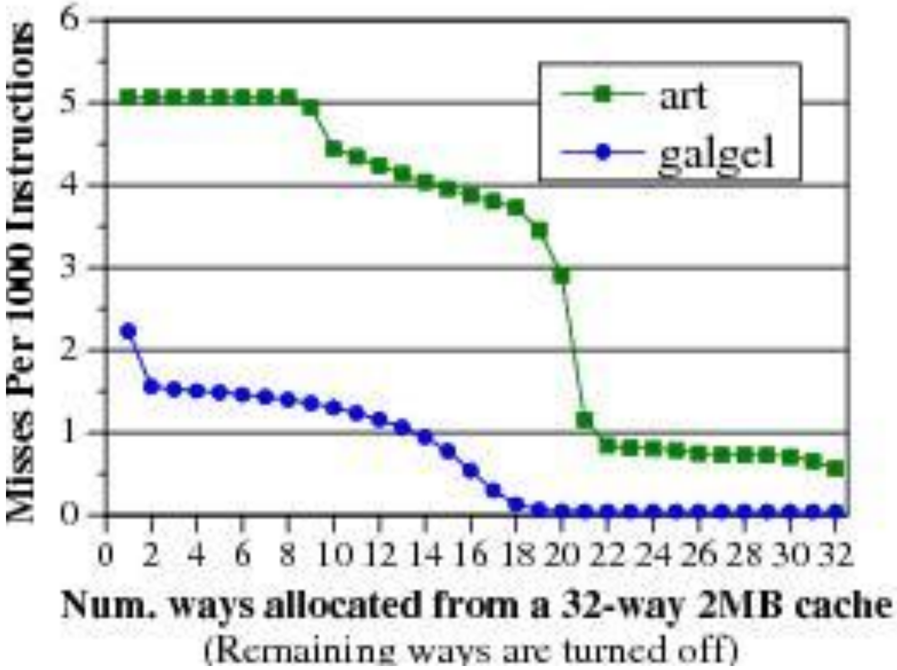
- Problem NP hard → need scalable partitioning algorithm

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

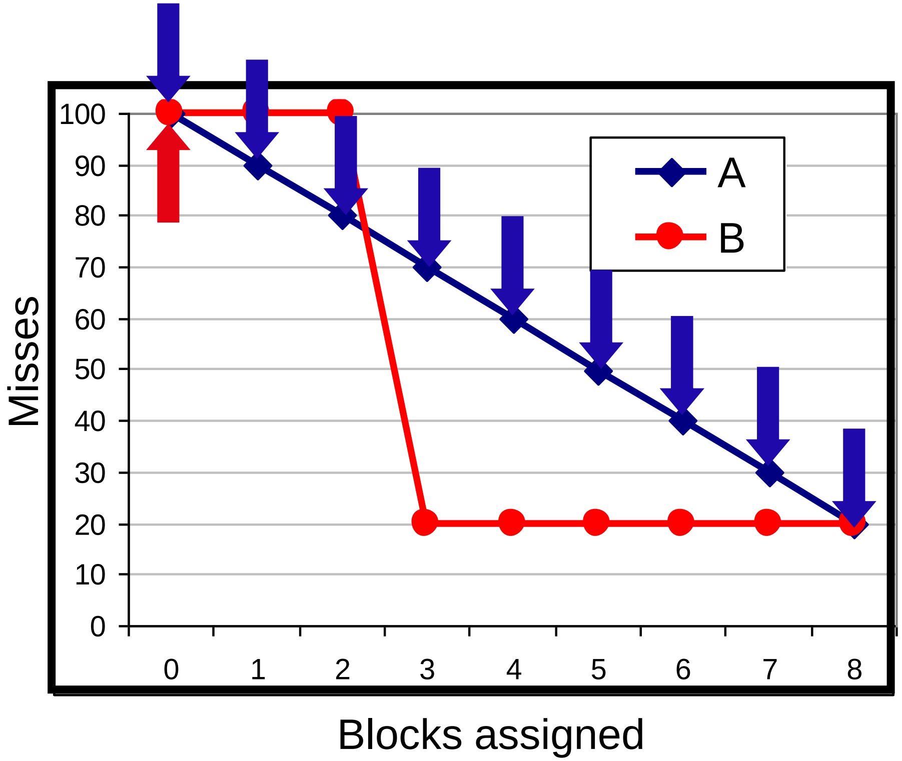
- Optimal partitioning when utility curves are convex

- 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

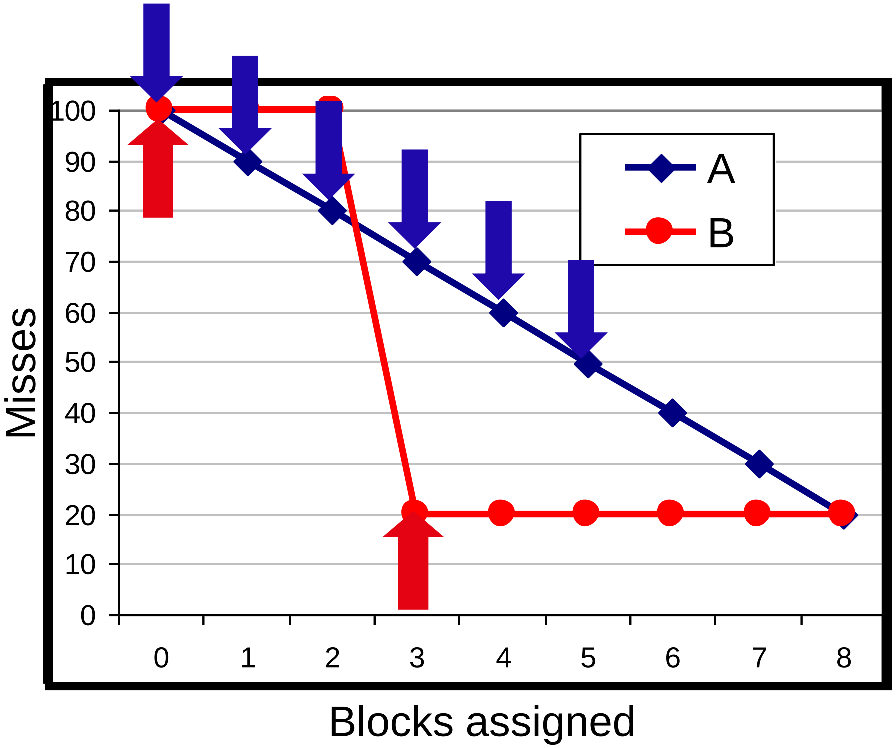
- Marginal Utility (MU) = Utility per cache resource
  - $MU_a^b = U_a^b/(b-a)$

- 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

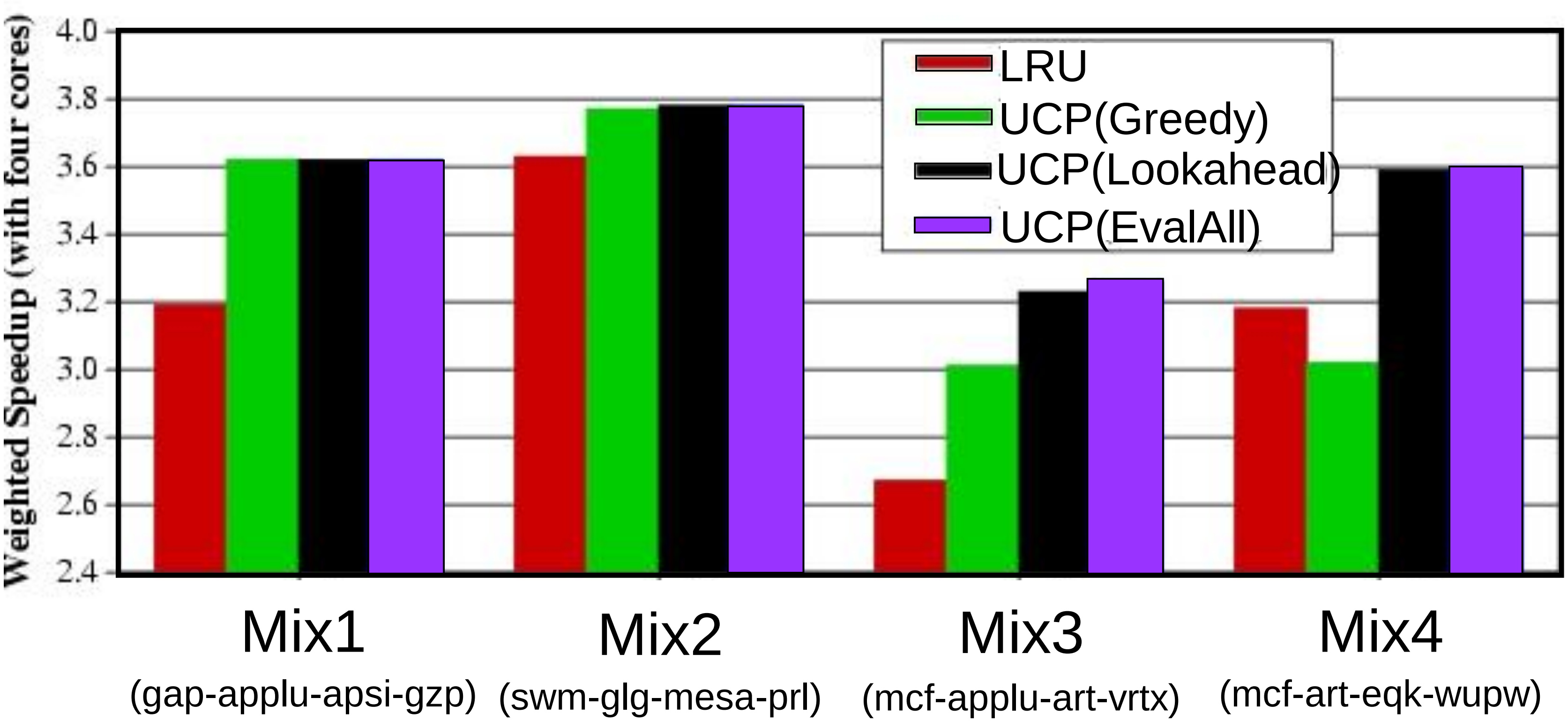
A gets 1 block

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



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?

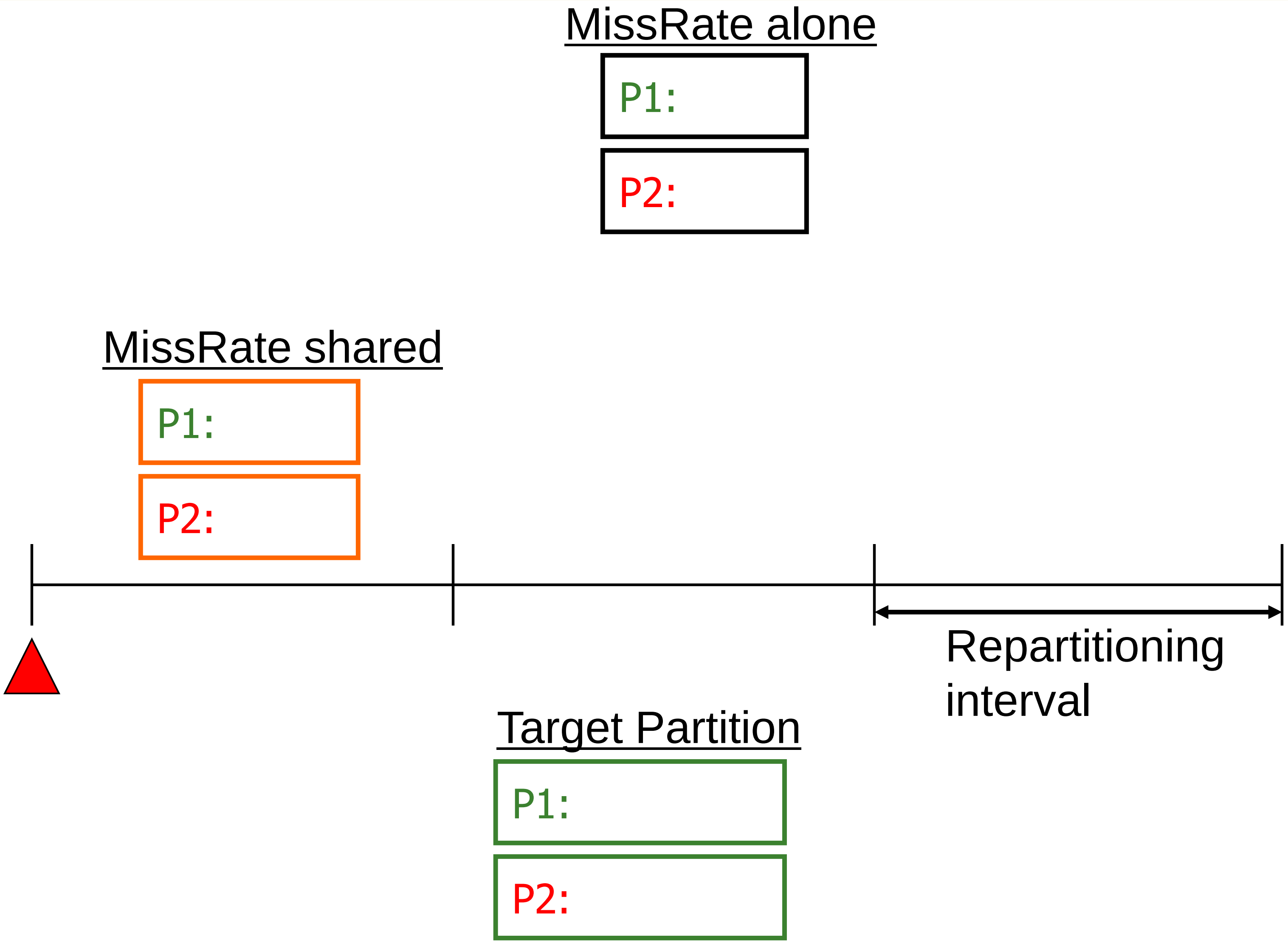
  - What if past behavior is not a good predictor of utility?

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

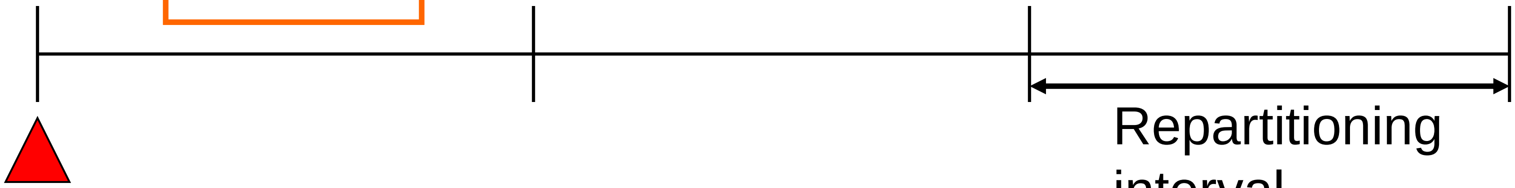
P1:20%

P2: 5%

MissRate shared

P1:20%

P2:15%



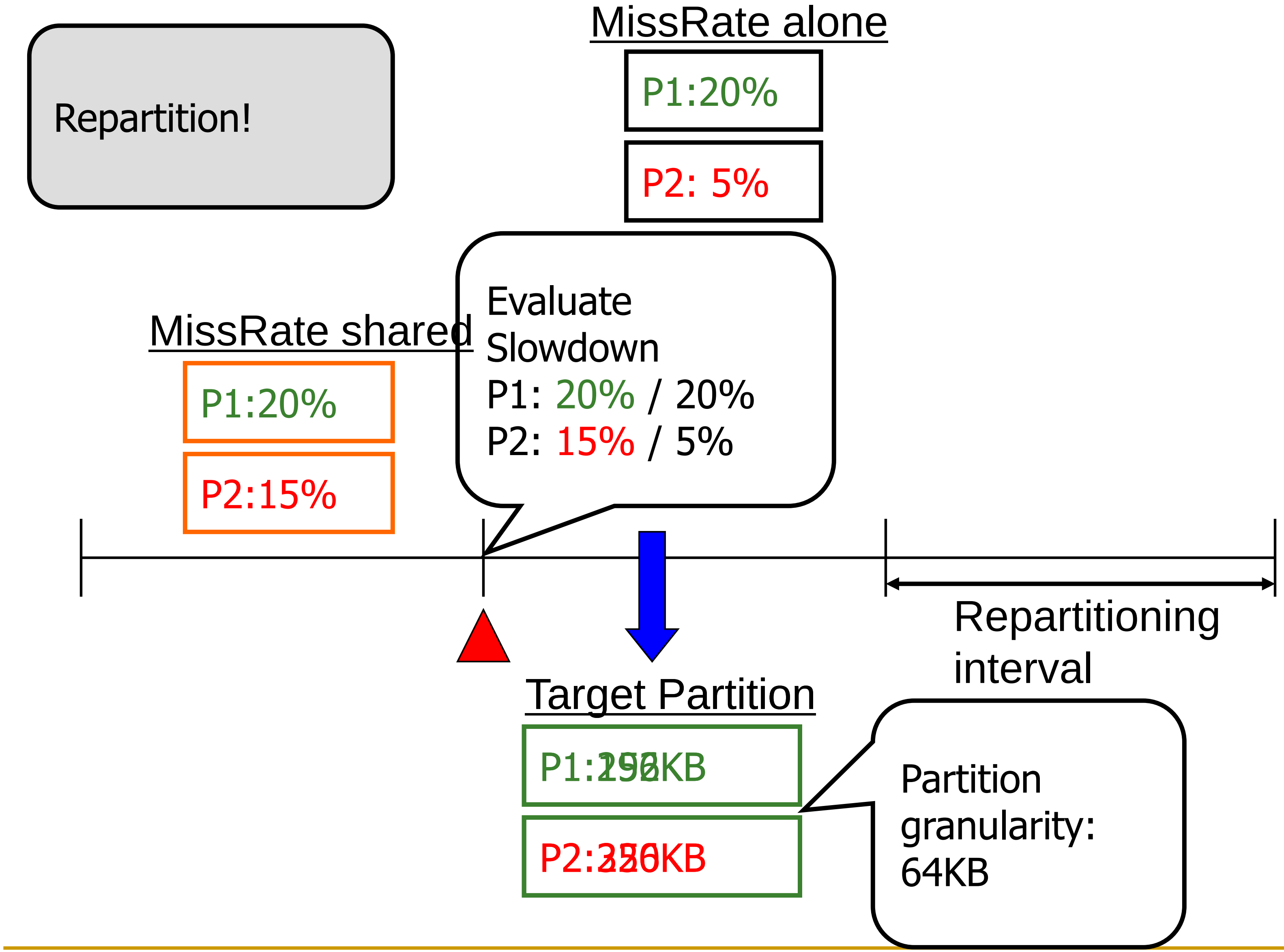
Repartitioning interval

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

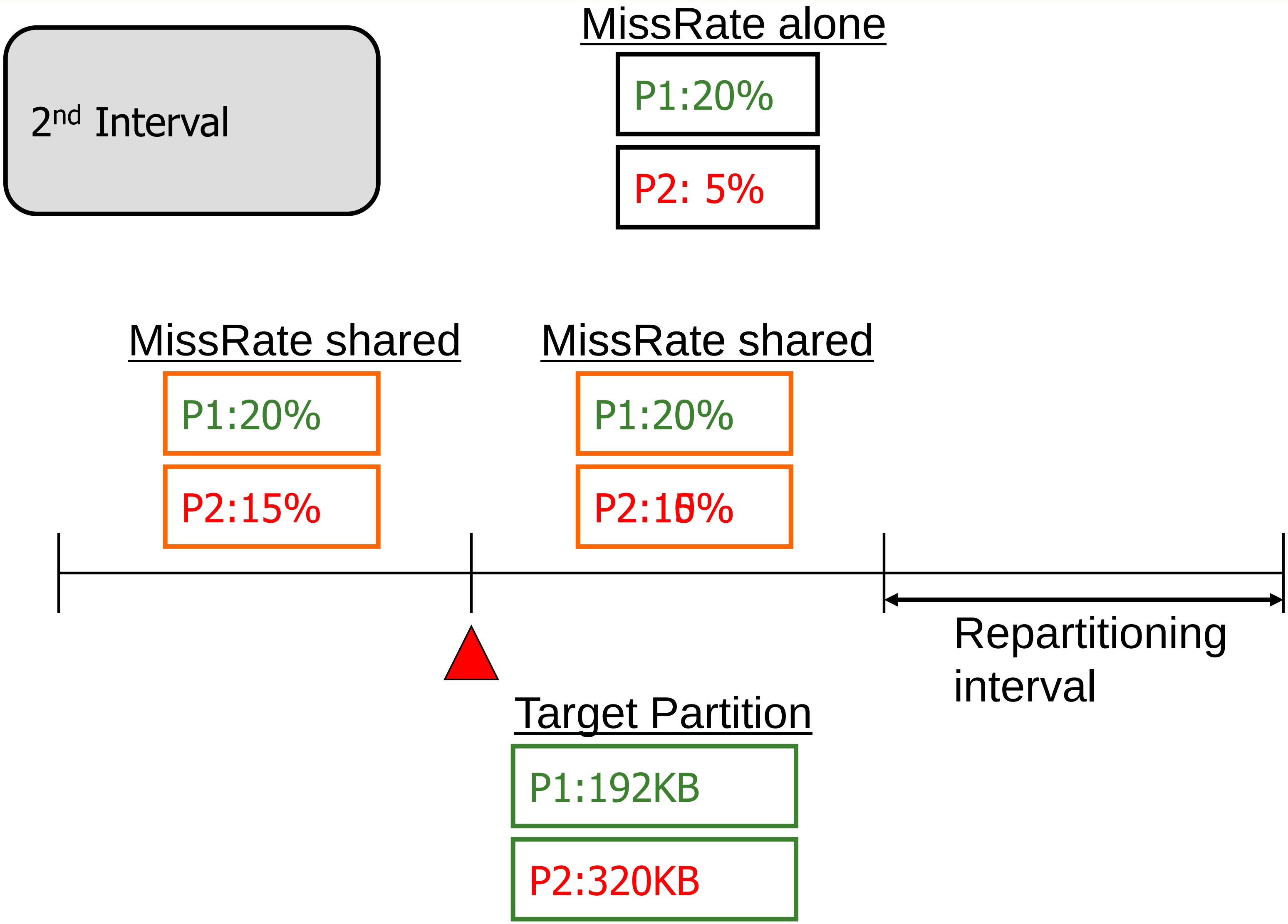
Target Partition

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%

P2:15%
P2:10%

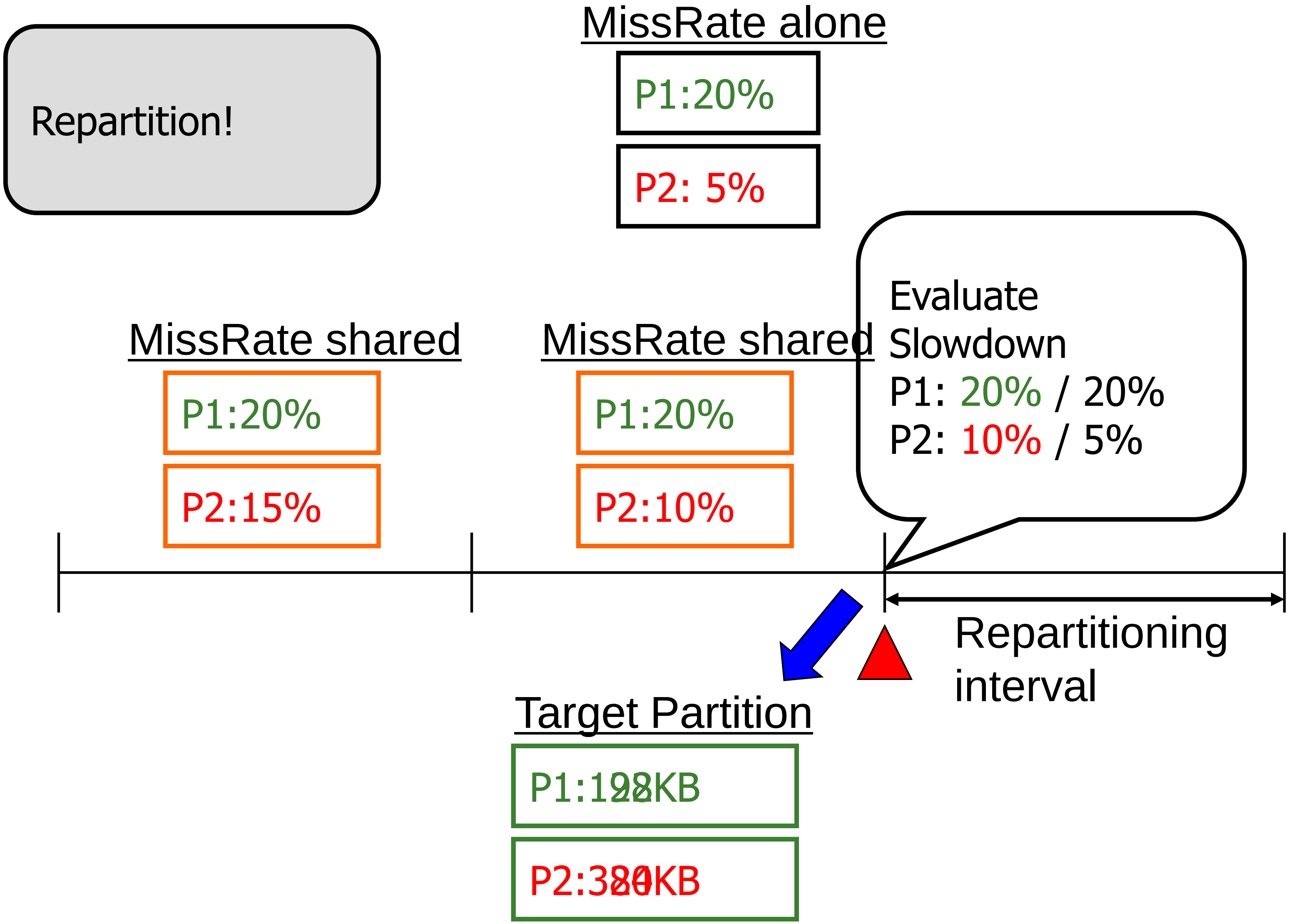
Repartitioning interval

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

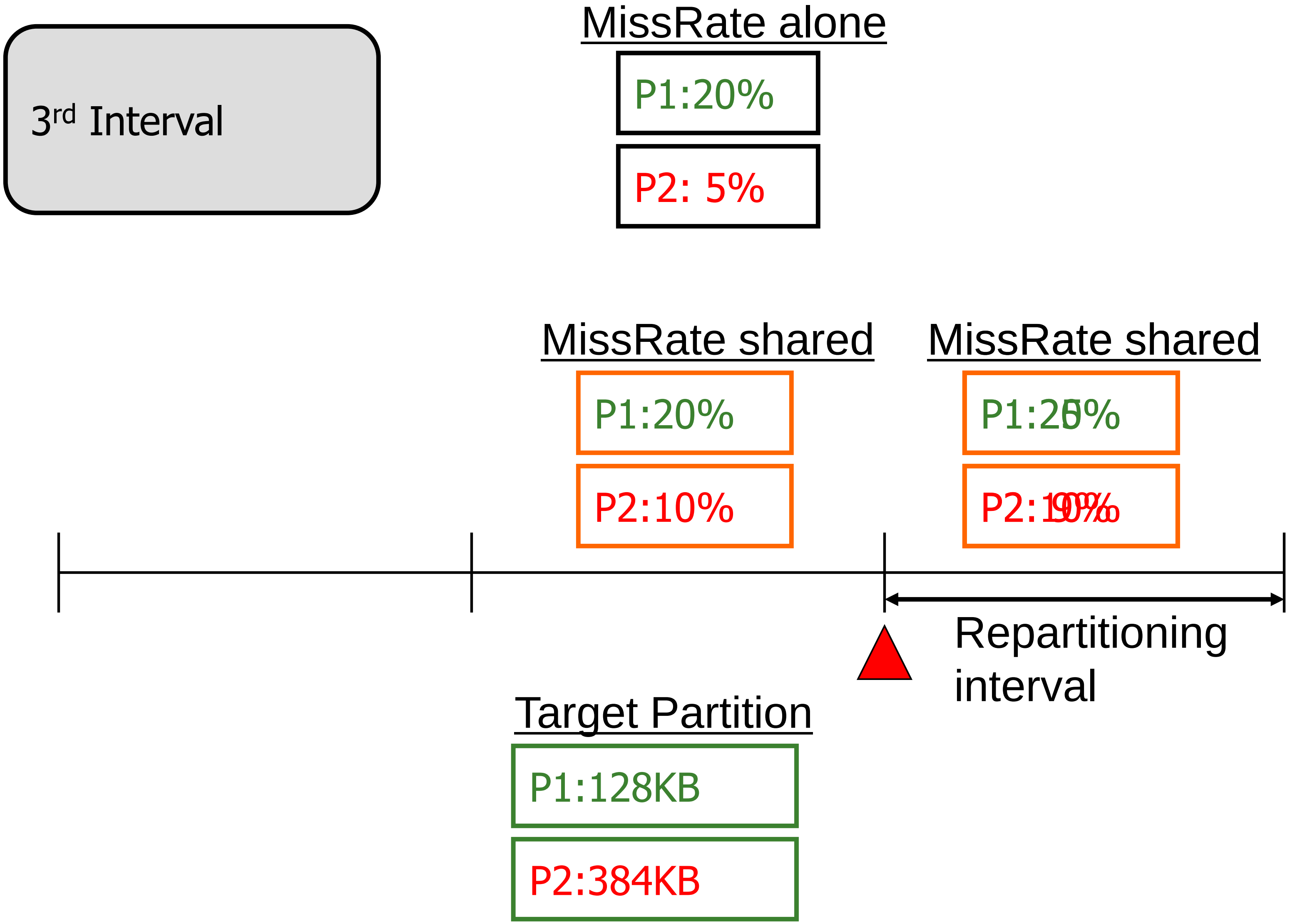
Repartitioning interval

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%

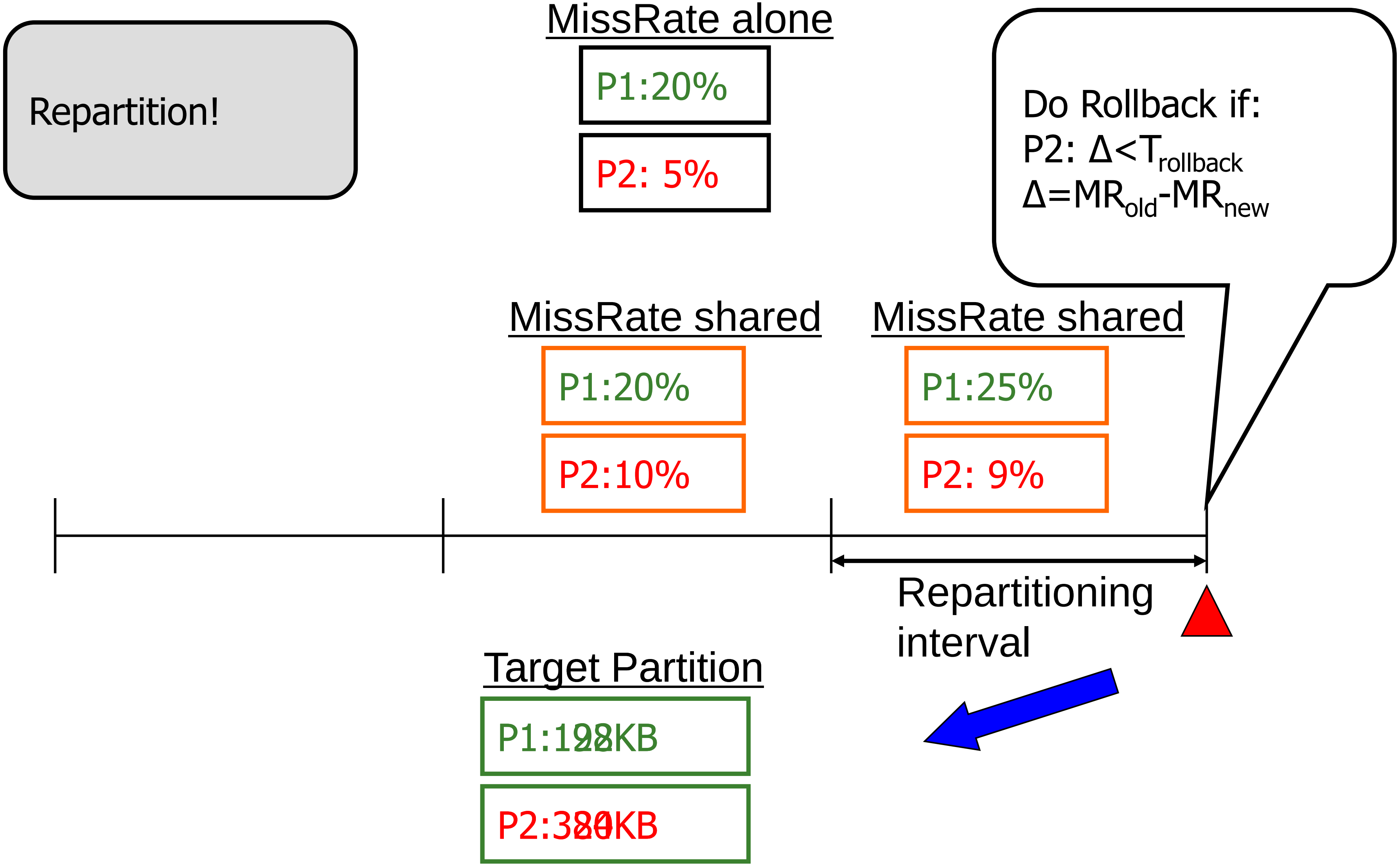
Repartitioning interval

Target Partition

P1:128KB

P2:384KB

Dynamic Fair Caching Algorithm



Repartition!

MissRate alone

P1:20%

P2: 5%

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

MissRate shared

P1:20%

P2:10%

MissRate shared

P1:25%

P2: 9%

Repartitioning interval

Target Partition

P1:192KB
P1:128KB

P2:320KB
P2:384KB

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

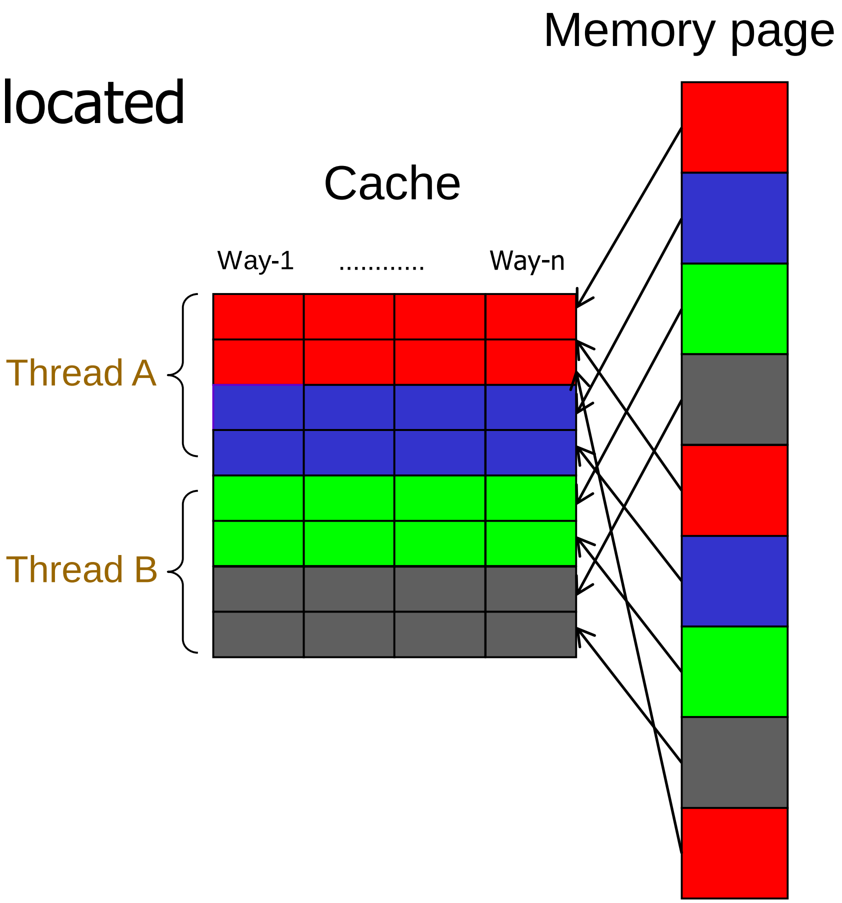
OS Based Cache Partitioning

- 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

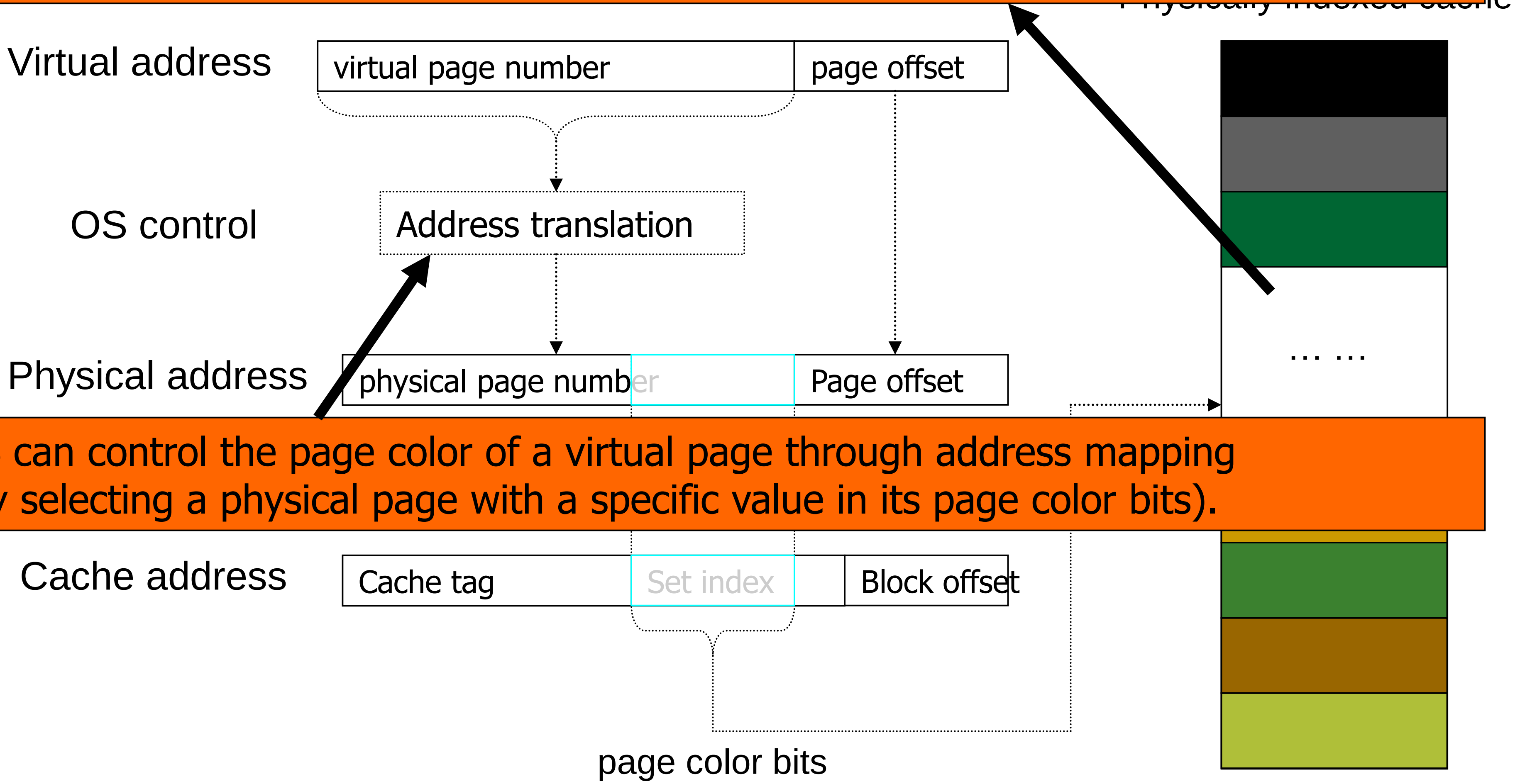
Page Coloring

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



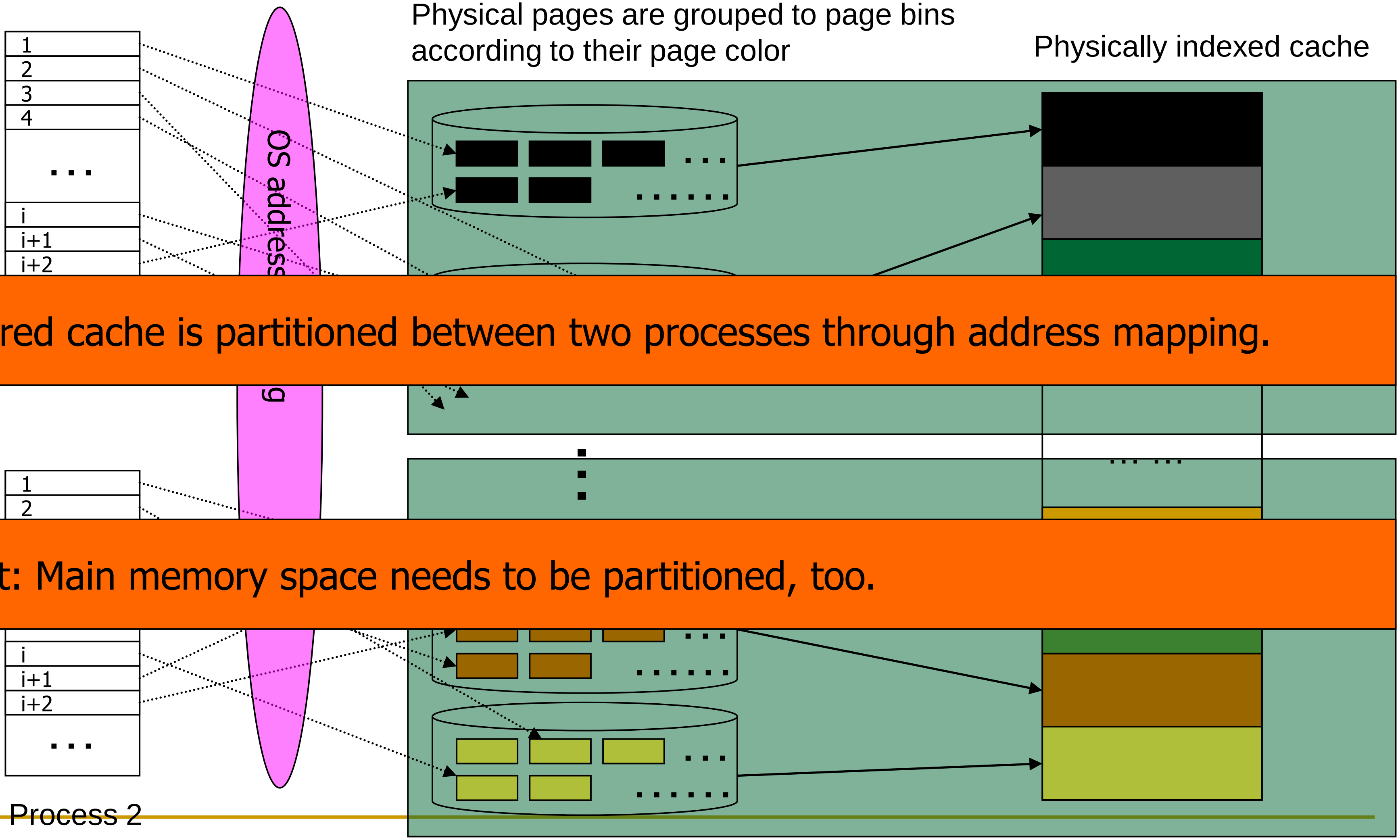
Page Coloring

- 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



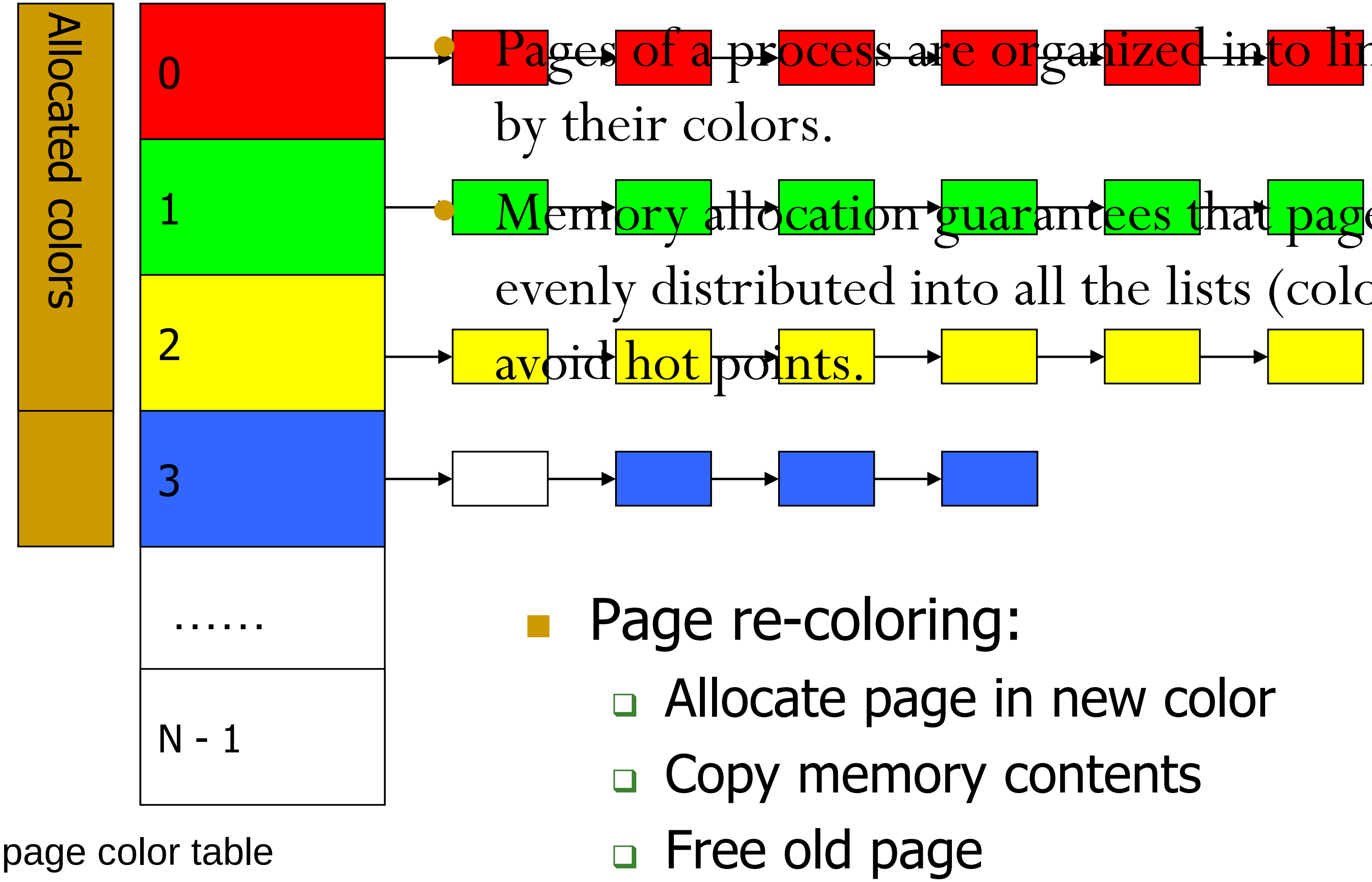
Physical pages are grouped to page bins according to their page color

Physically indexed cache

Shared cache is partitioned between two processes through address mapping.

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

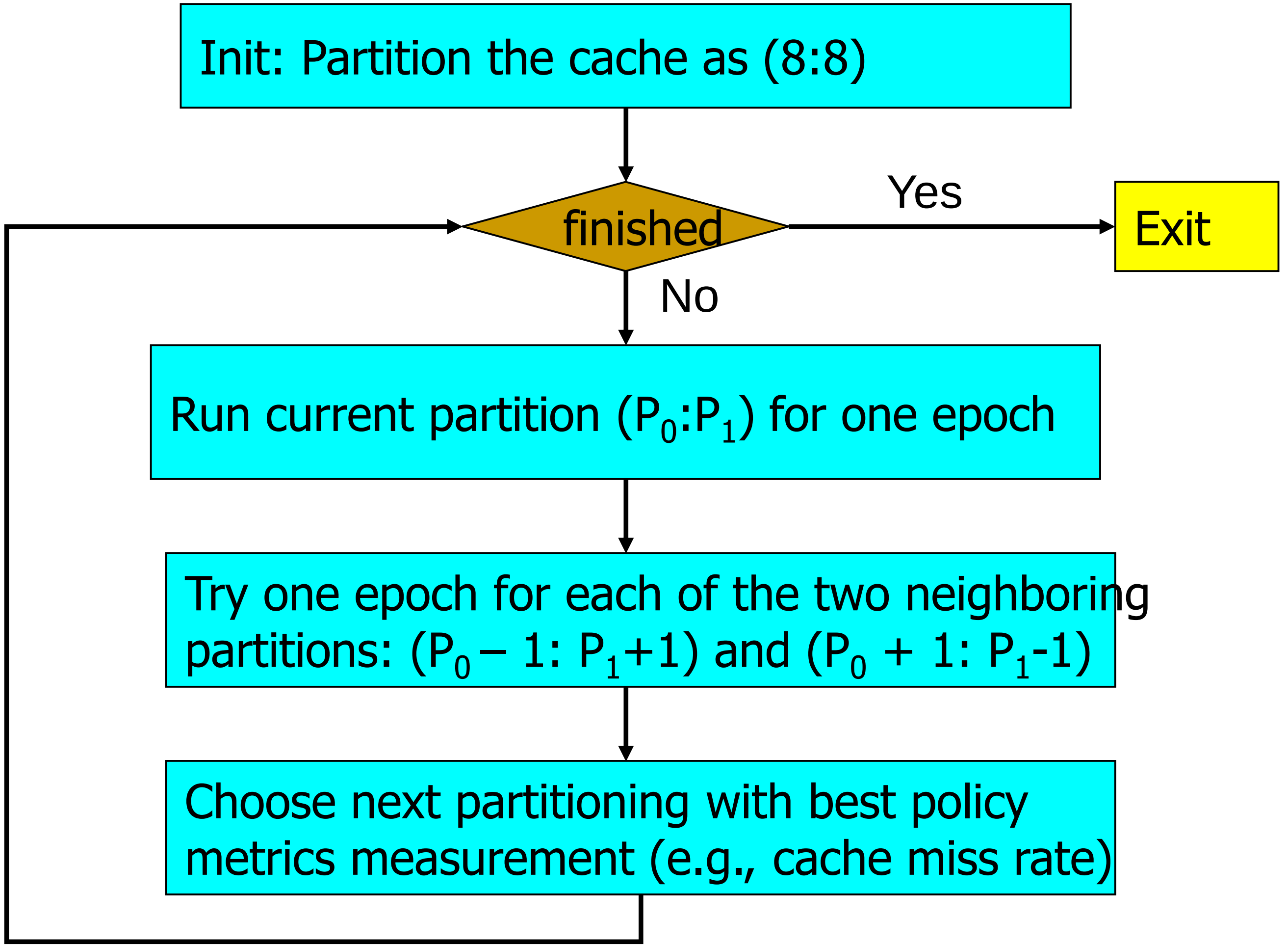
Dynamic Cache Partitioning via Page Re-Coloring



- 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)

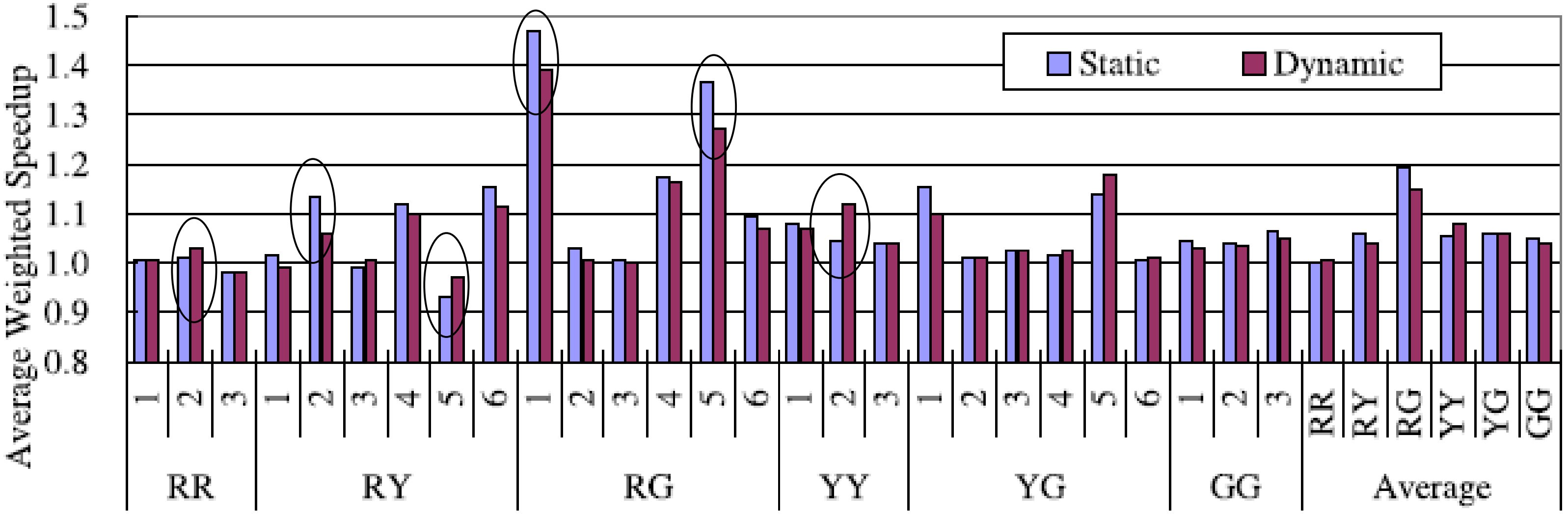
Choose next partitioning with best policy metrics measurement (e.g., cache miss rate)

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

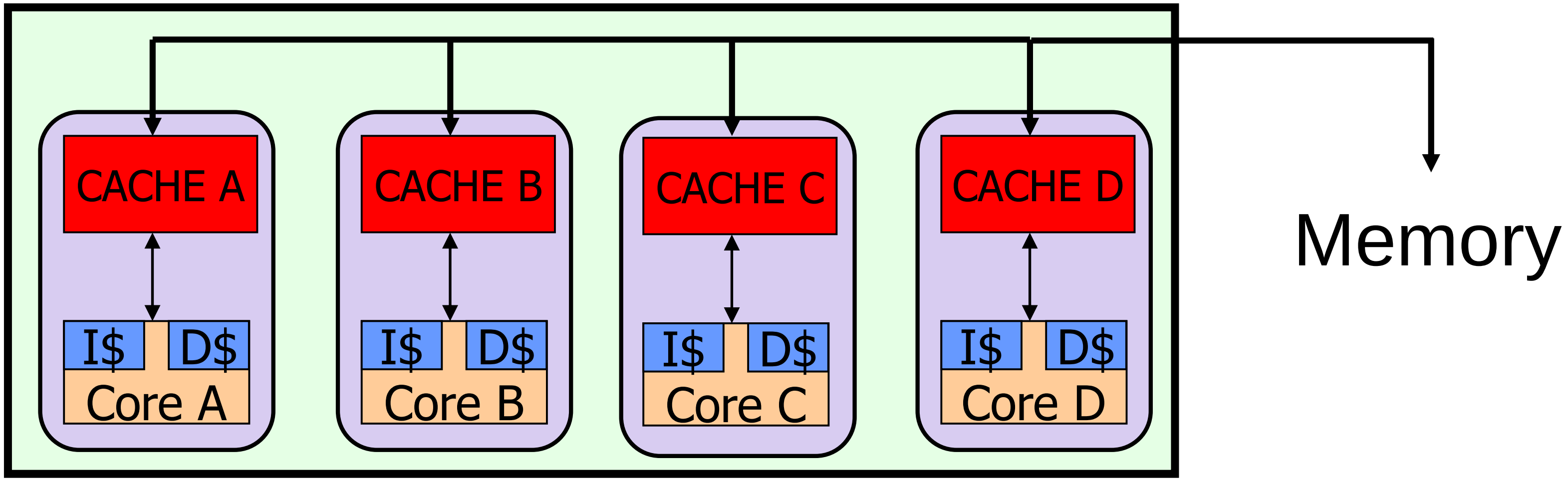
- Goal: Achieve the benefits of private caches (low latency, performance isolation) while sharing cache capacity across cores

- 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

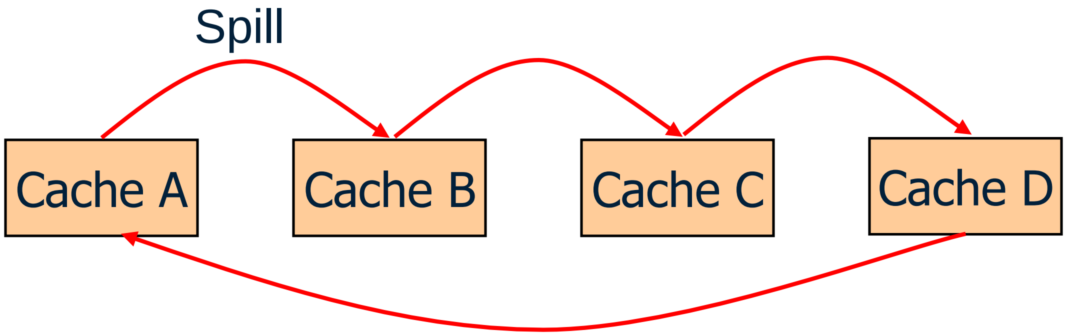


Problem: When one core needs more cache and other core has spare cache, private-cache CMPs cannot share capacity

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

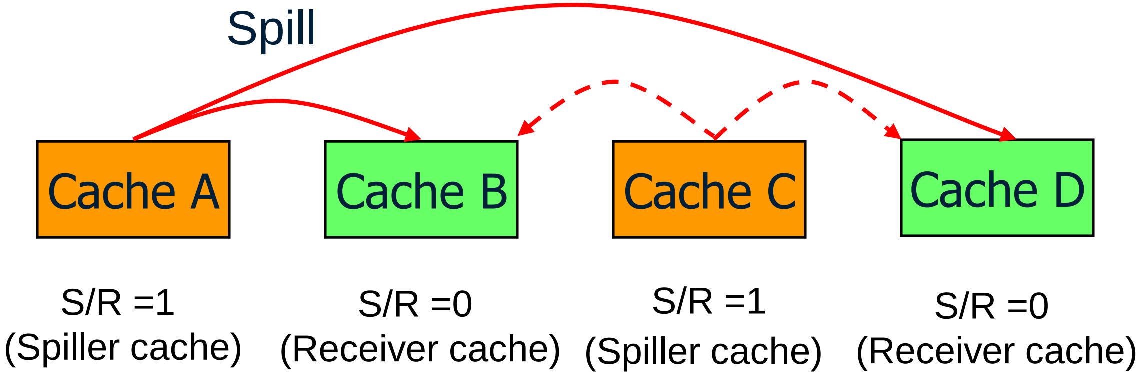
Goal: Robust High-Performance Capacity Sharing with Negligible Overhead

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.

Dynamic Spill-Receive via "Set Dueling"

Divide the cache in three:

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

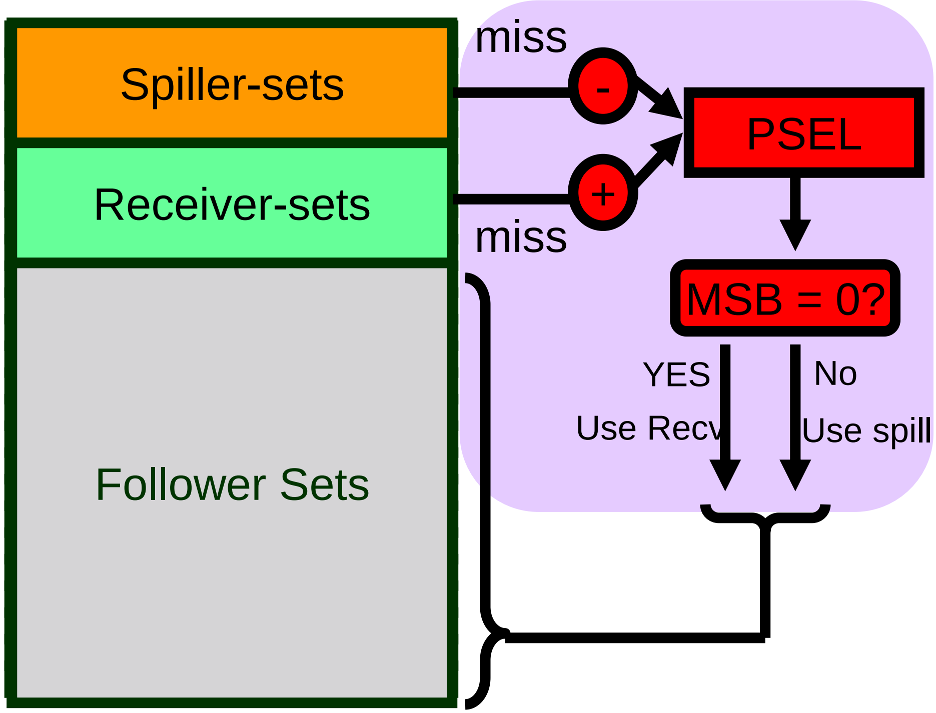
n-bit PSEL counter

misses to spiller-sets: PSEL--

misses to receiver-set: PSEL++

MSB of PSEL decides policy for Follower sets:

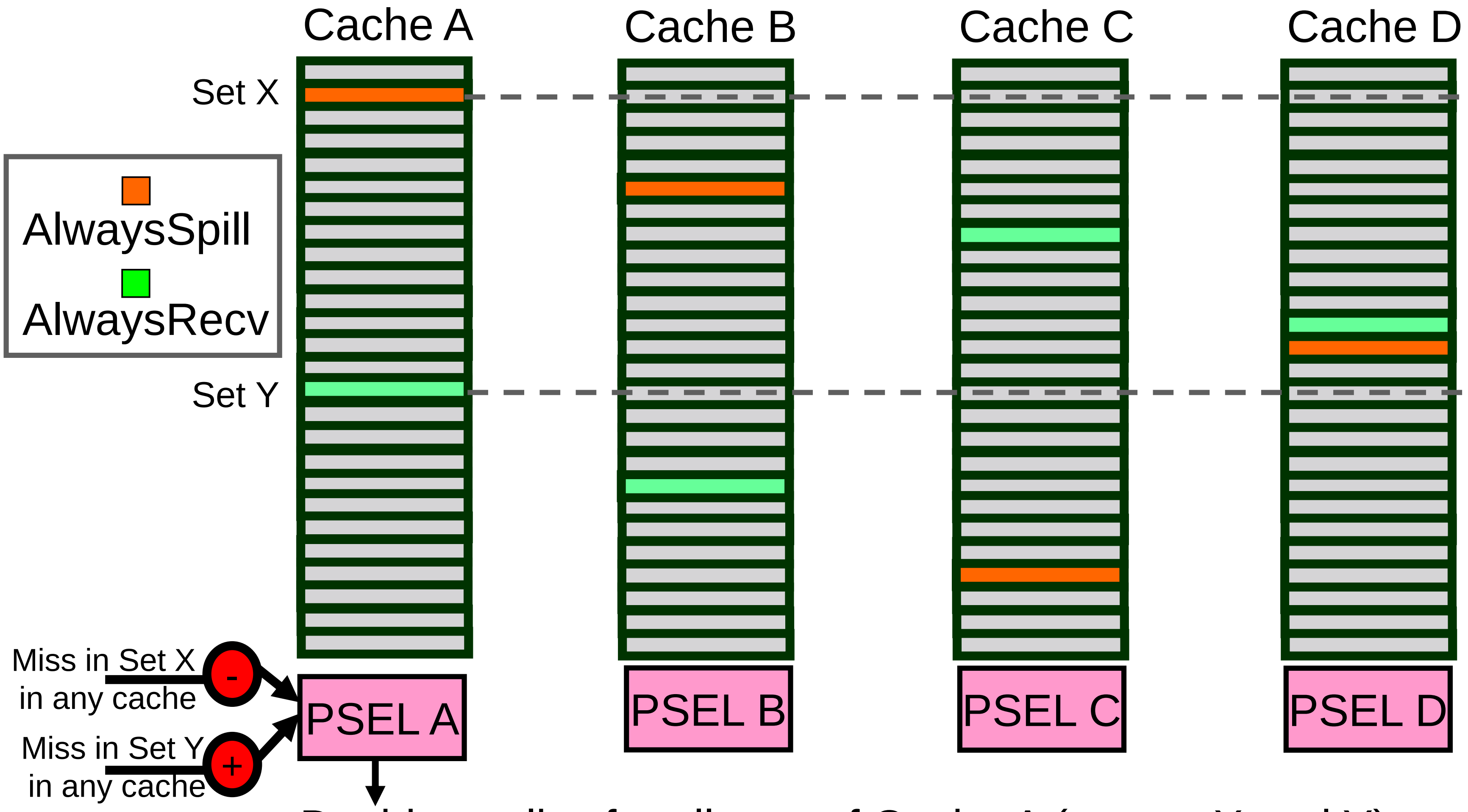
- MSB = 0, Use spill
- MSB = 1, Use receive



monitor → choose → apply
(using a single counter)

Dynamic Spill-Receive Architecture

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



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

Experimental Setup

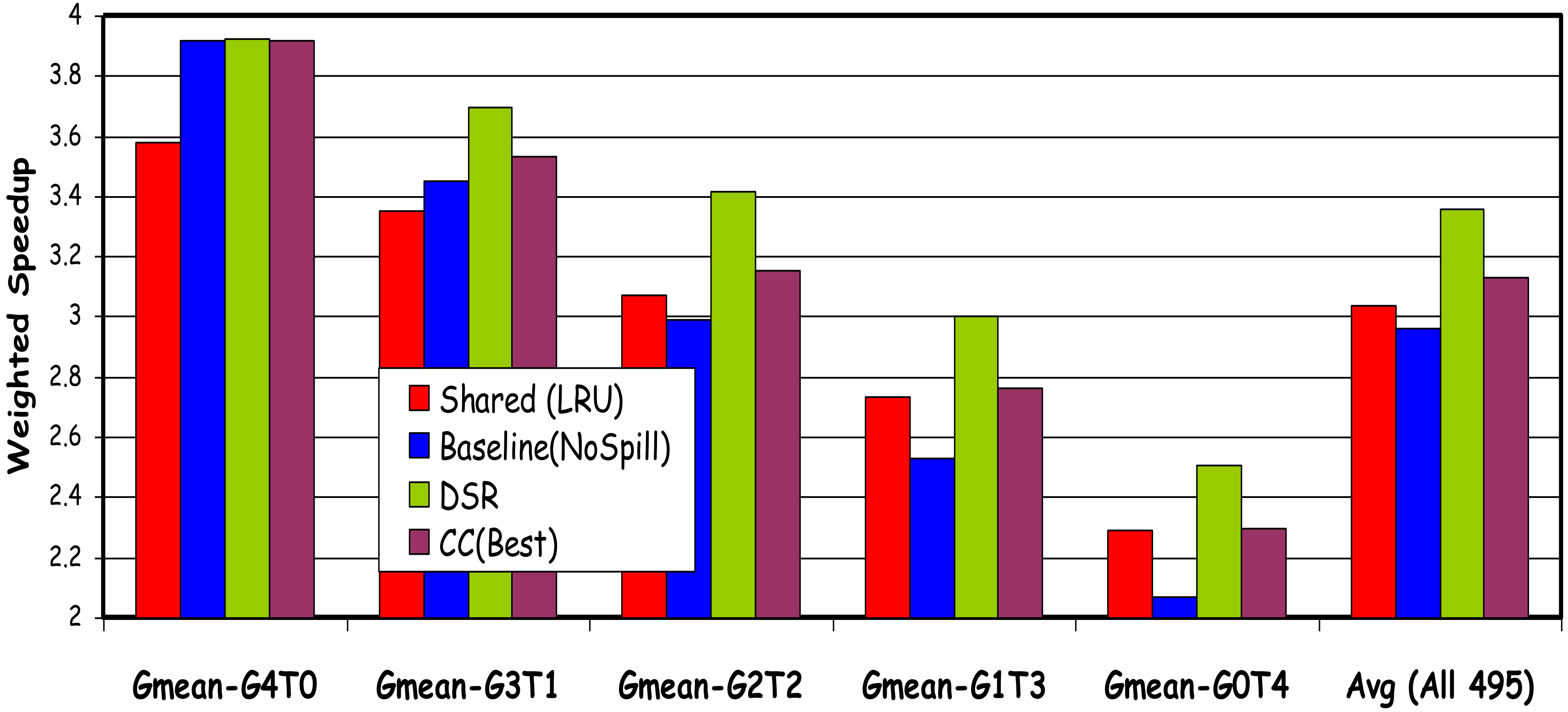
- Baseline Study:
  - 4-core CMP with in-order cores
  - Private Cache Hierarchy: 16KB L1, 1MB L2
  - 10 cycle latency for local hits, 40 cycles for remote hits

- 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 Weighted Speedup



On average, DSR improves weighted speedup by 13%

Distributed Caches

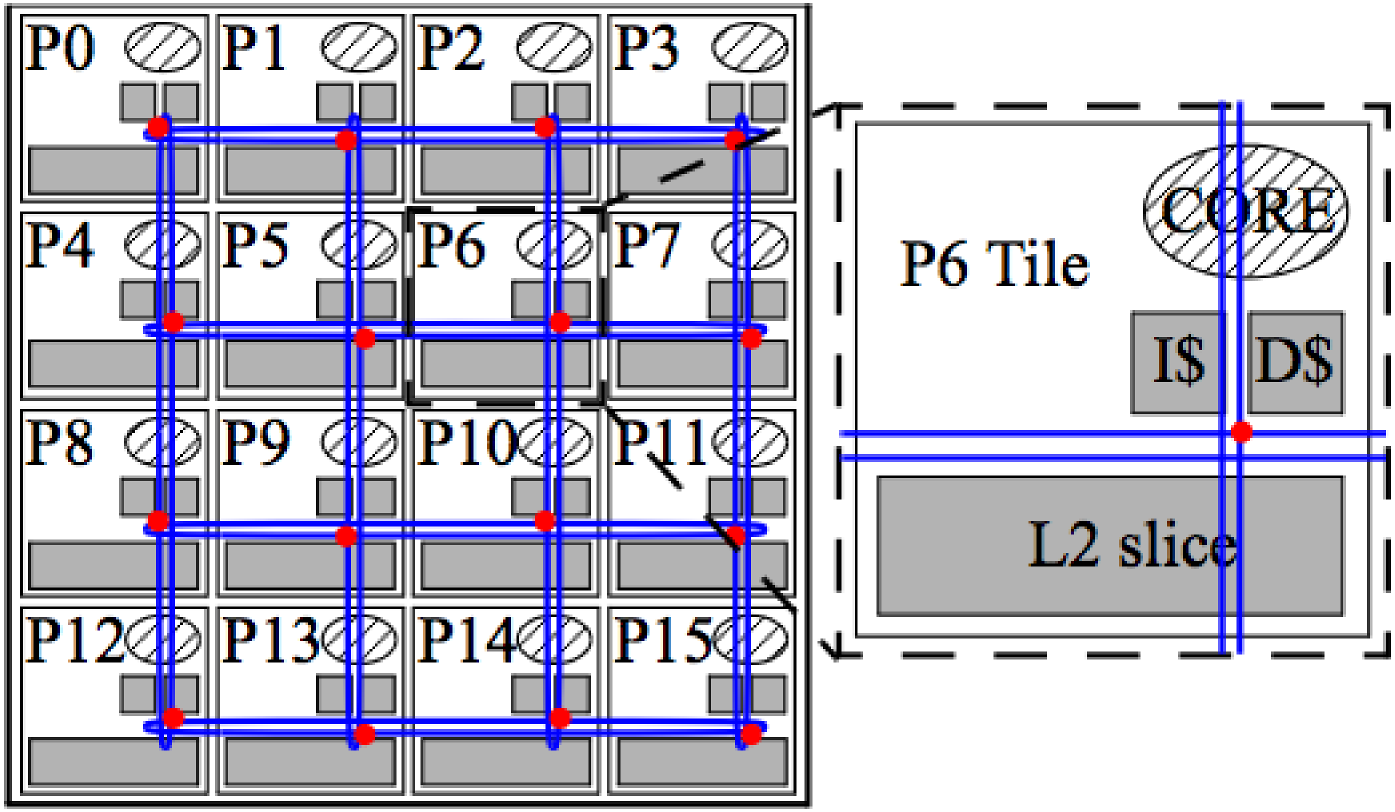
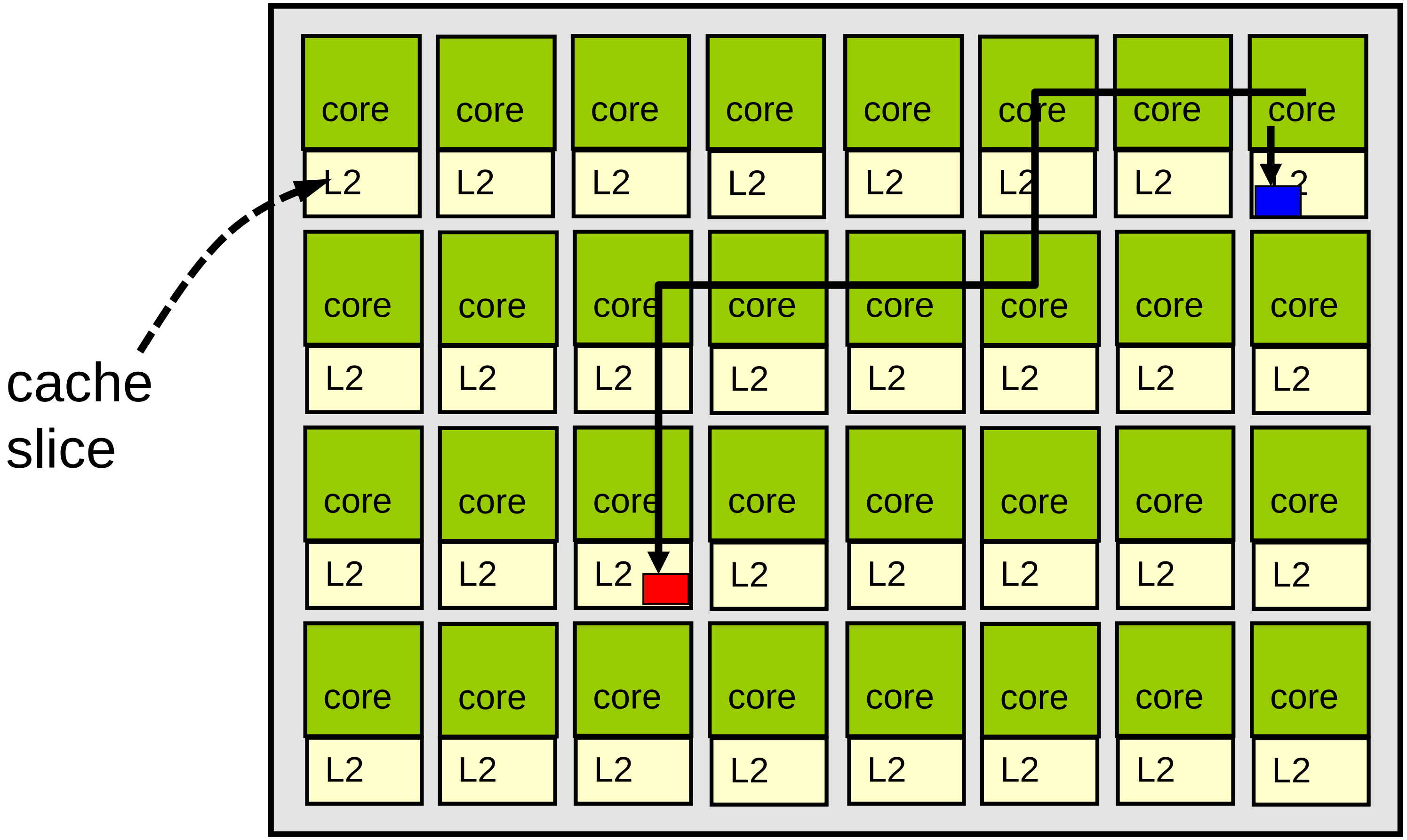


FIGURE 1. Typical tiled architecture. Tiles are interconnected into a 2-D folded torus. Each tile contains a core, L1 instruction and data caches, a shared-L2 cache slice, and a router/switch.

Caching for Parallel Applications



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

Efficient Cache Utilization

Efficient Cache Utilization: Examples

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

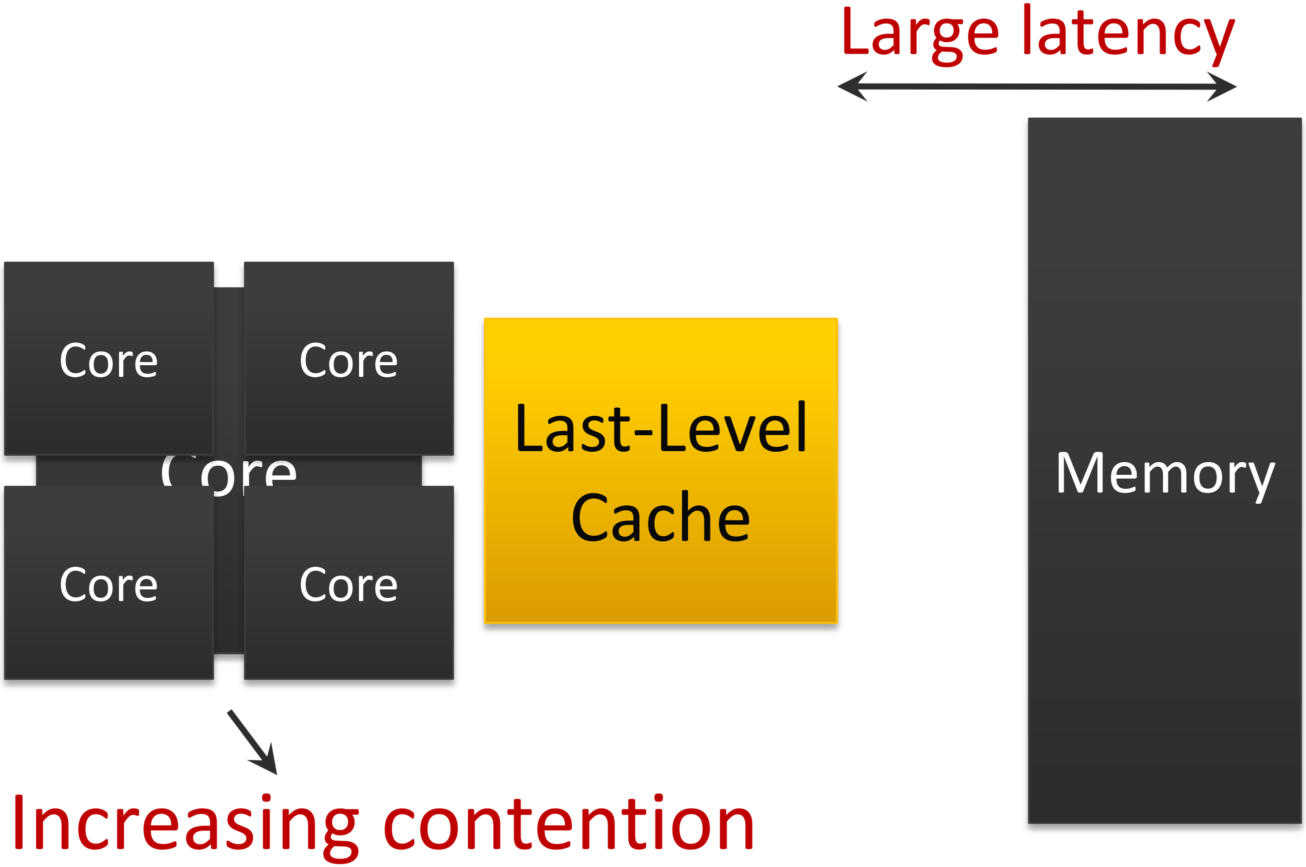
- Seshadri et al., "The Evicted-Address Filter: A Unified Mechanism to Address both Cache Pollution and Thrashing," PACT 2012.

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

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)

Cache Utilization is Important



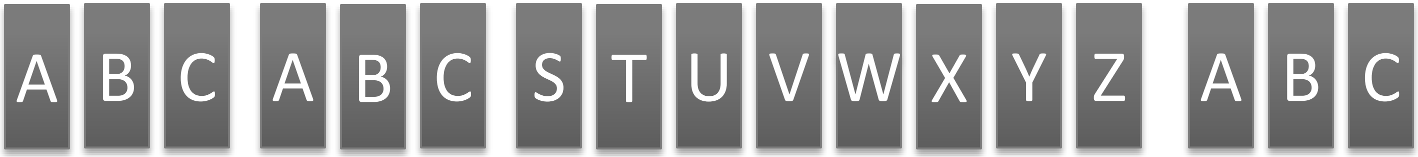
Increasing contention

Effective cache utilization is important

Reuse Behavior of Cache Blocks

Different blocks have different reuse behavior

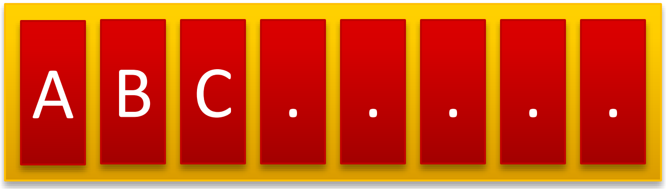
Access Sequence:



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

🟥 High-reuse block 🟦 Low-reuse block

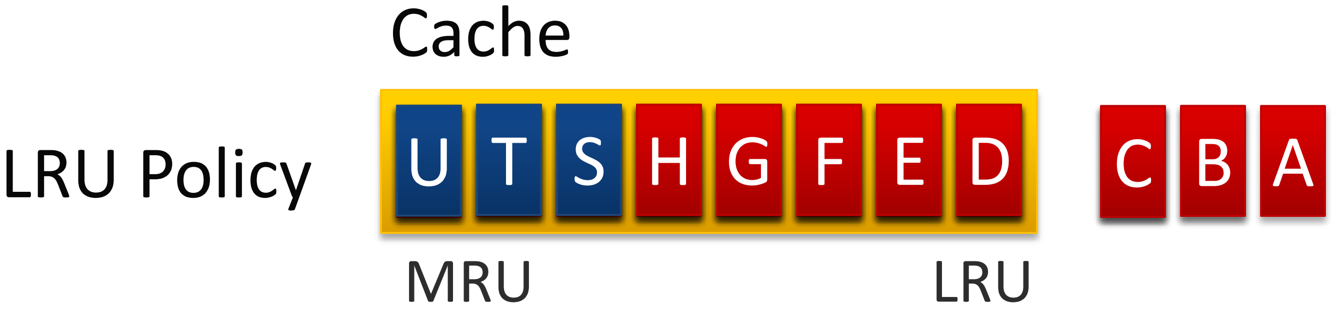
Ideal Cache



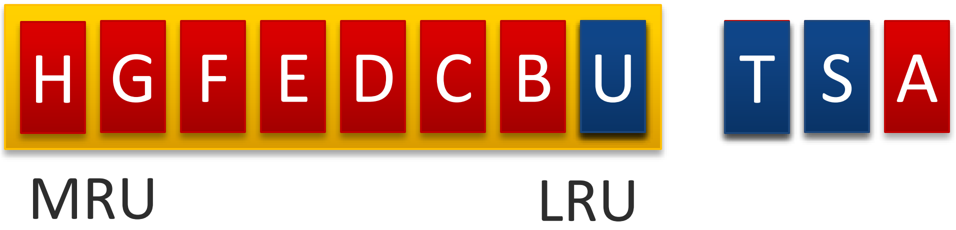
A B C . . . . .

Cache Pollution

Problem: Low-reuse blocks evict high-reuse blocks

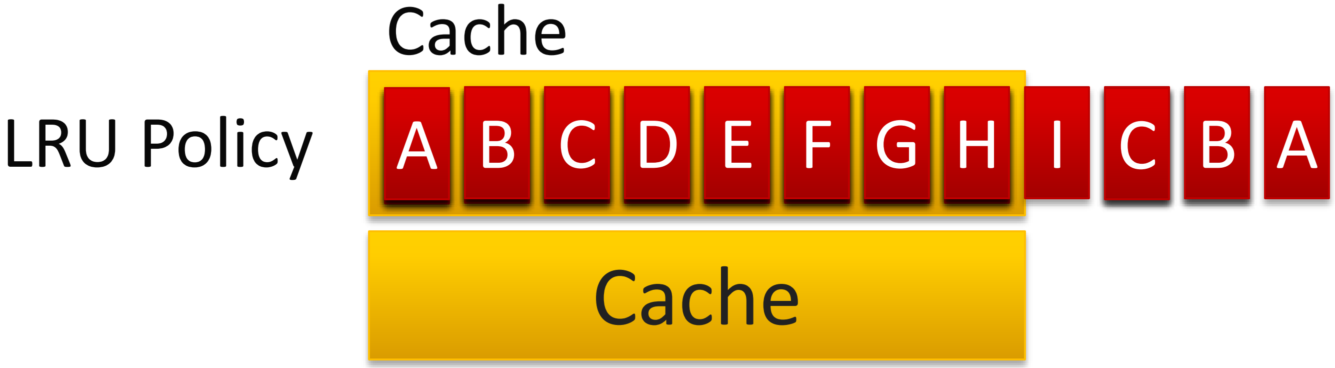


Idea: Predict reuse behavior of missed blocks. Insert low-reuse blocks at LRU position.

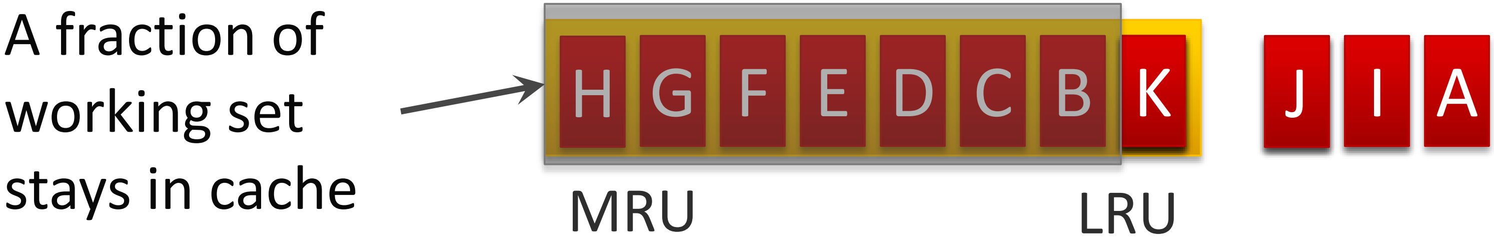


Cache Thrashing

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

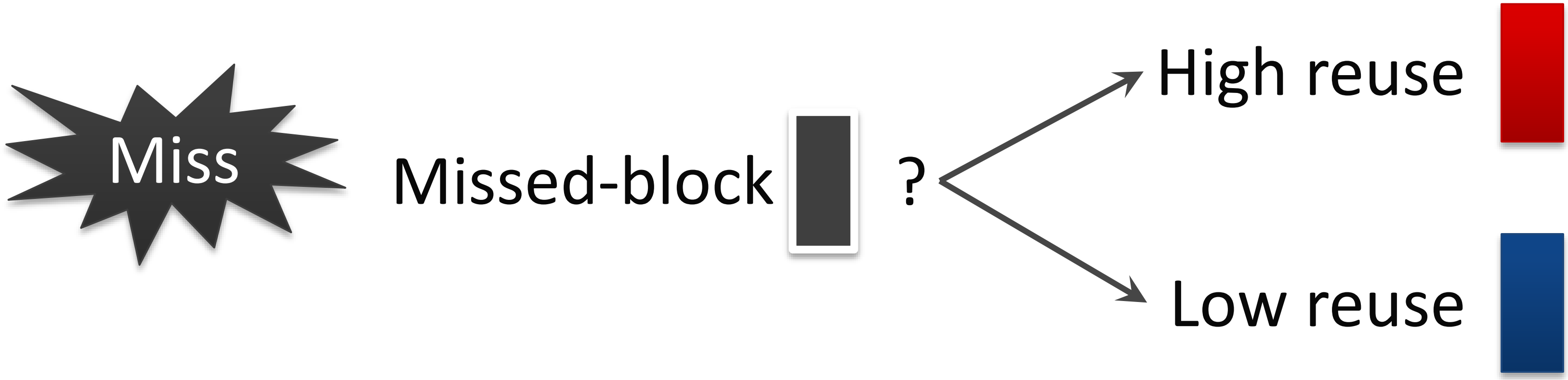
Cache Pollution

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

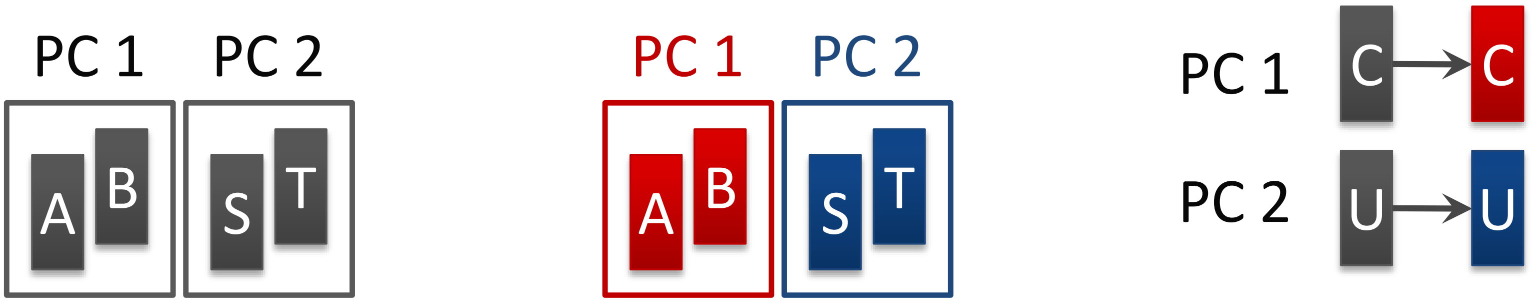
Approaches to Reuse Prediction

Use program counter or memory region information.

1. Group Blocks

2. Learn group behavior

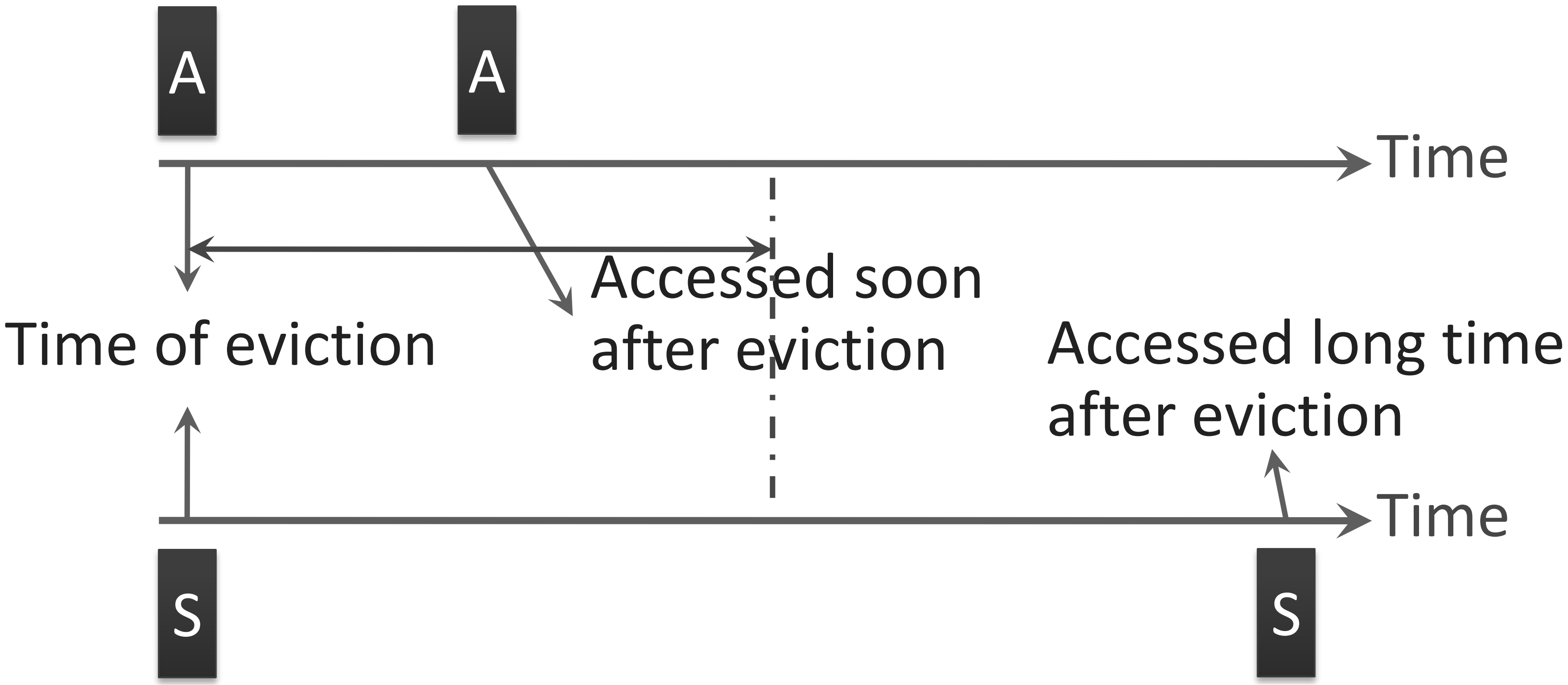
3. Predict reuse



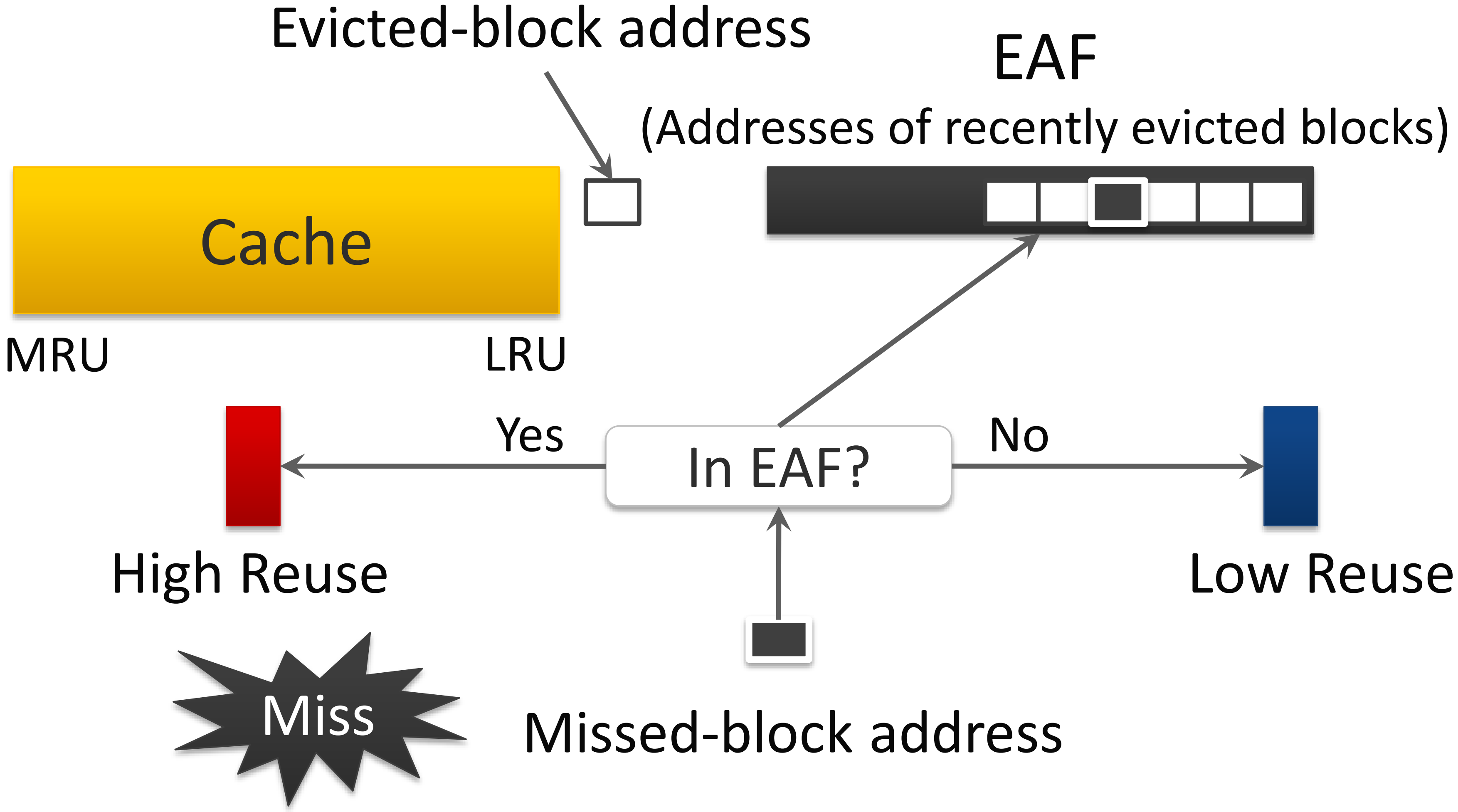
1. Same group ↛ same reuse behavior
2. No control over number of high-reuse blocks

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)



Evicted-block address

EAF

(Addresses of recently evicted blocks)

Cache

MRU

LRU

Yes

In EAF?

No

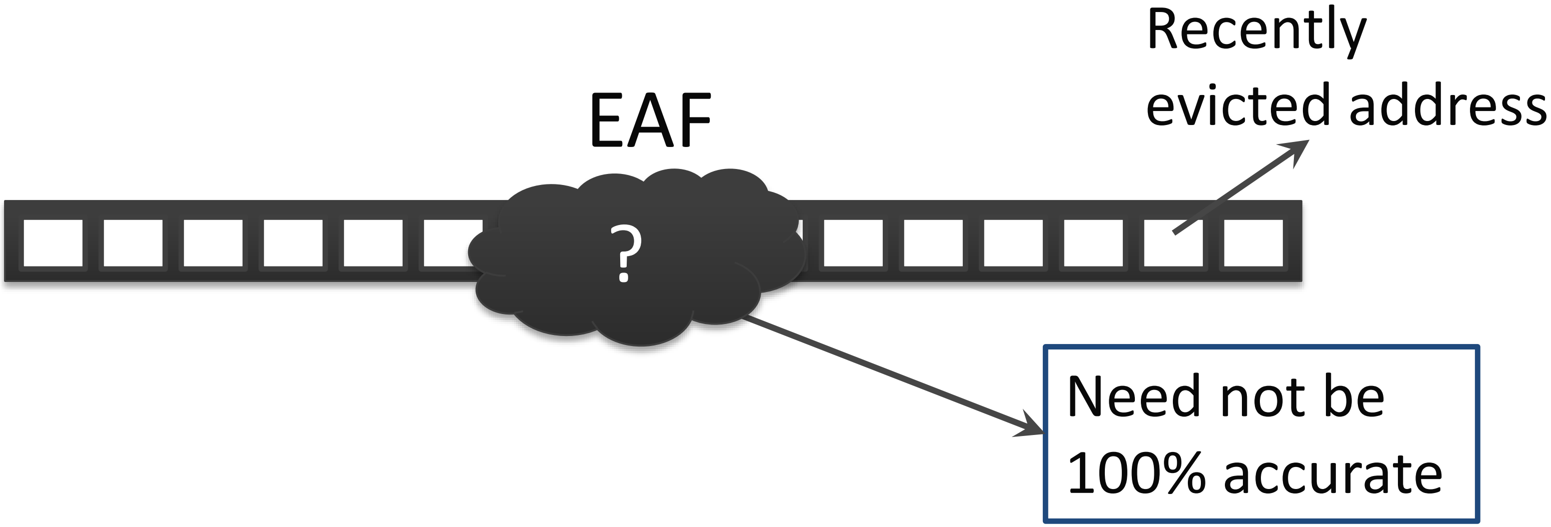
High Reuse

Low Reuse

Miss

Missed-block address

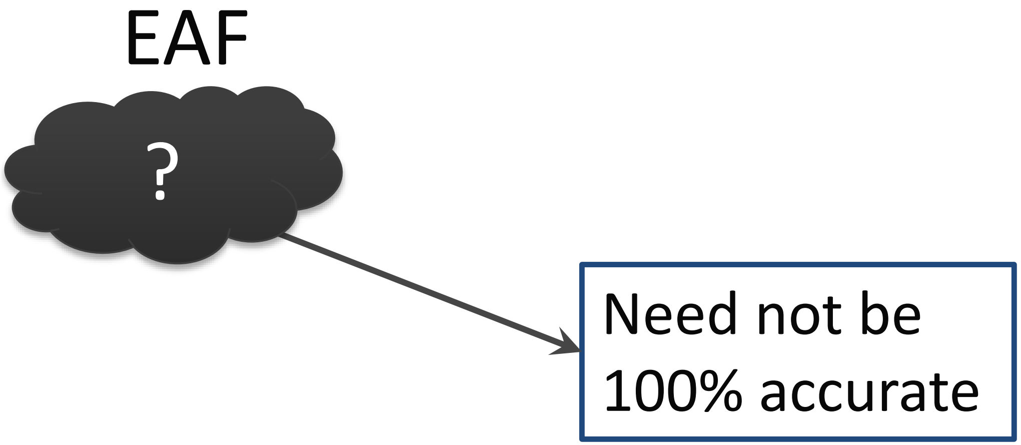
Naïve Implementation: Full Address Tags



Need not be 100% accurate

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

Low-Cost Implementation: Bloom Filter



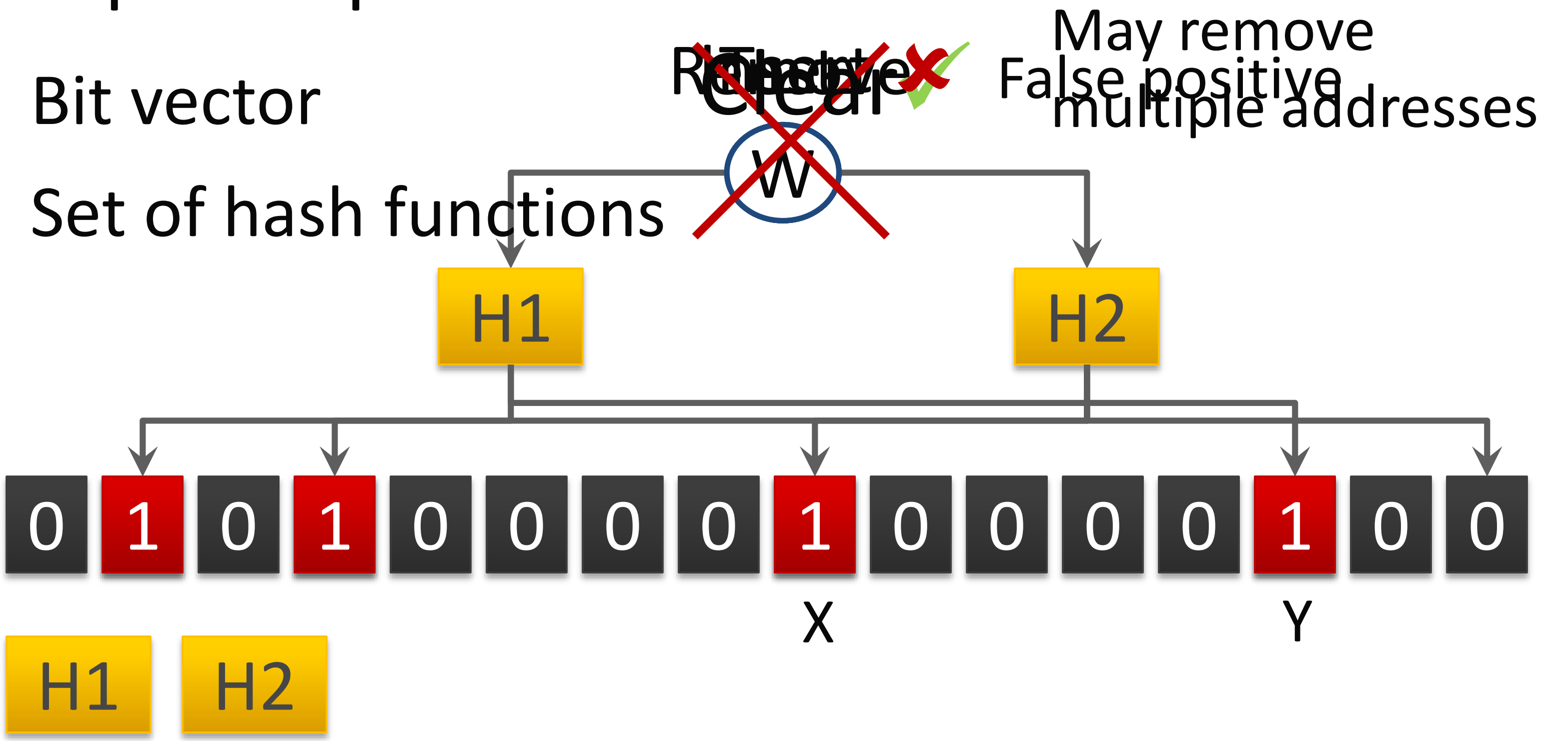
Need not be 100% accurate

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

Bloom Filter

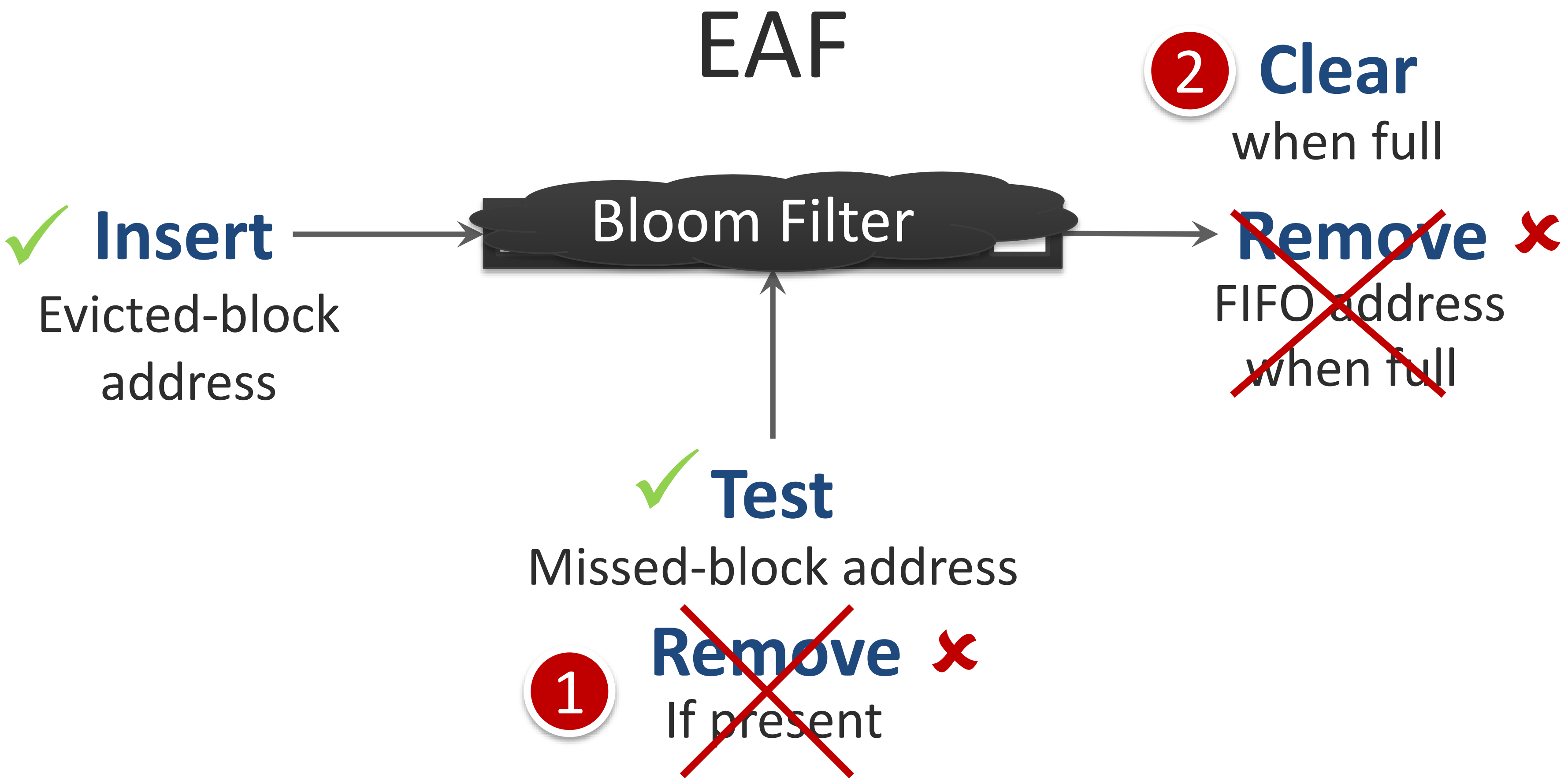
Compact representation of a set

1. Bit vector
2. Set of hash functions



Inserted Elements: X Y

EAF using a Bloom Filter



EAF

✓ **Insert**
Evicted-block address

Bloom Filter

2 **Clear**
when full

~~**Remove**~~ ✗
~~FIFO address when full~~

✓ **Test**
Missed-block address

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**

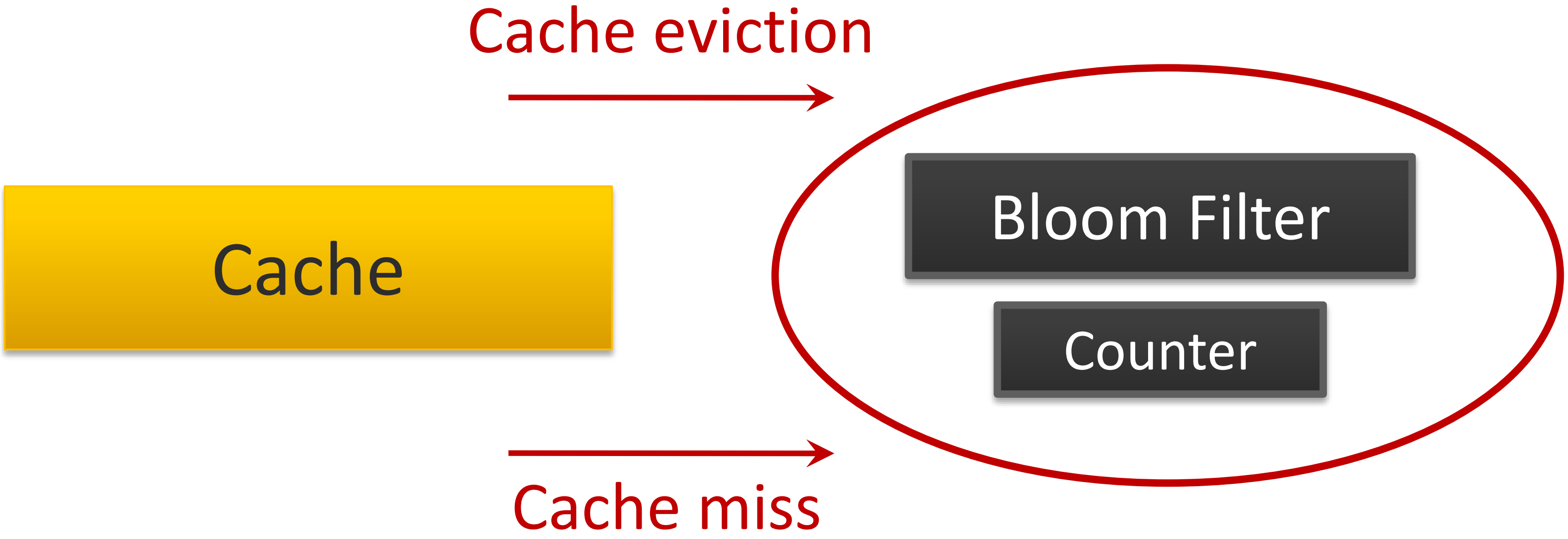
Clear filter and counter

2 **Cache miss**

Test if address is present in filter

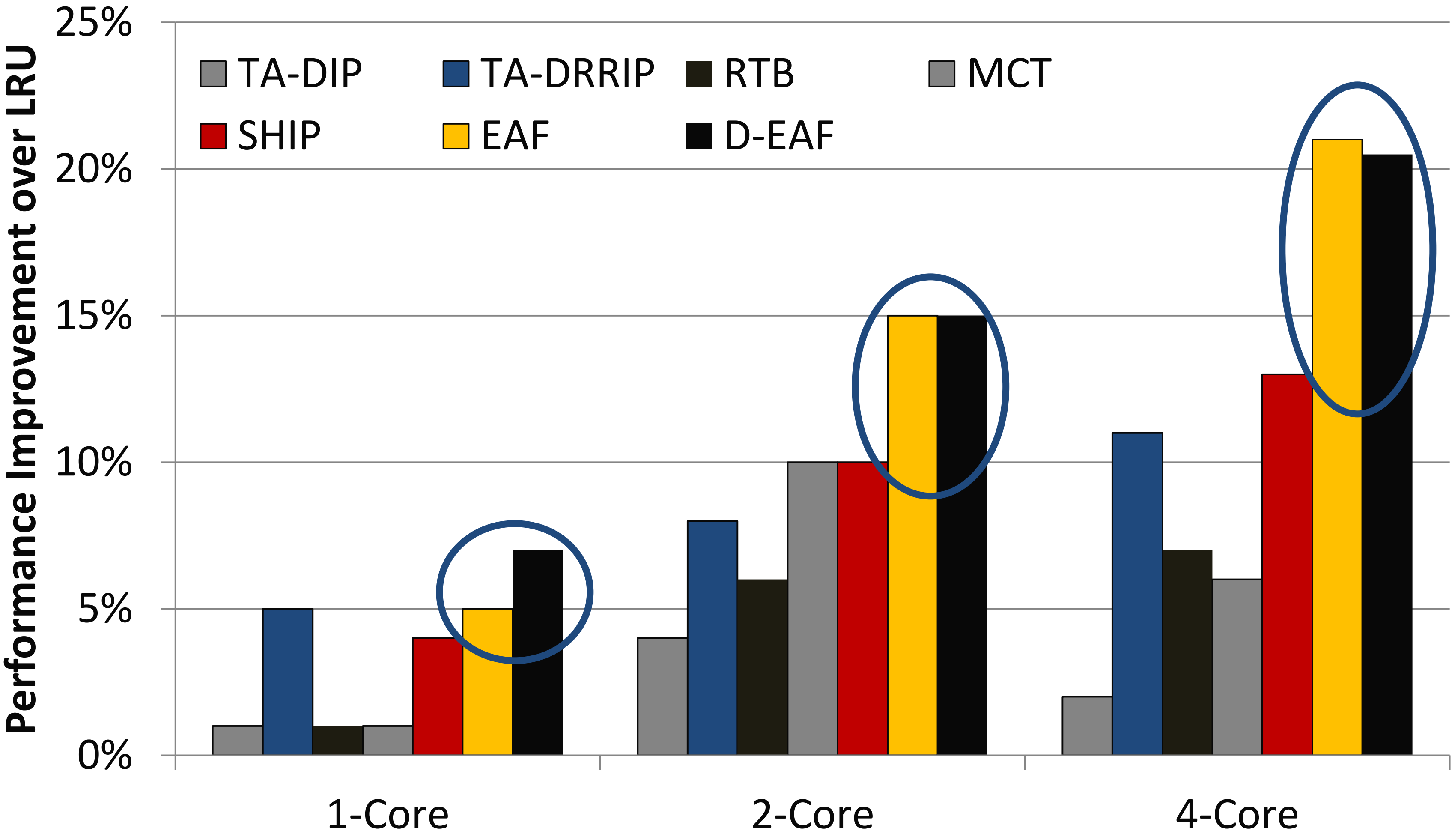
Yes, insert at MRU. No, insert with BIP

EAF: Advantages



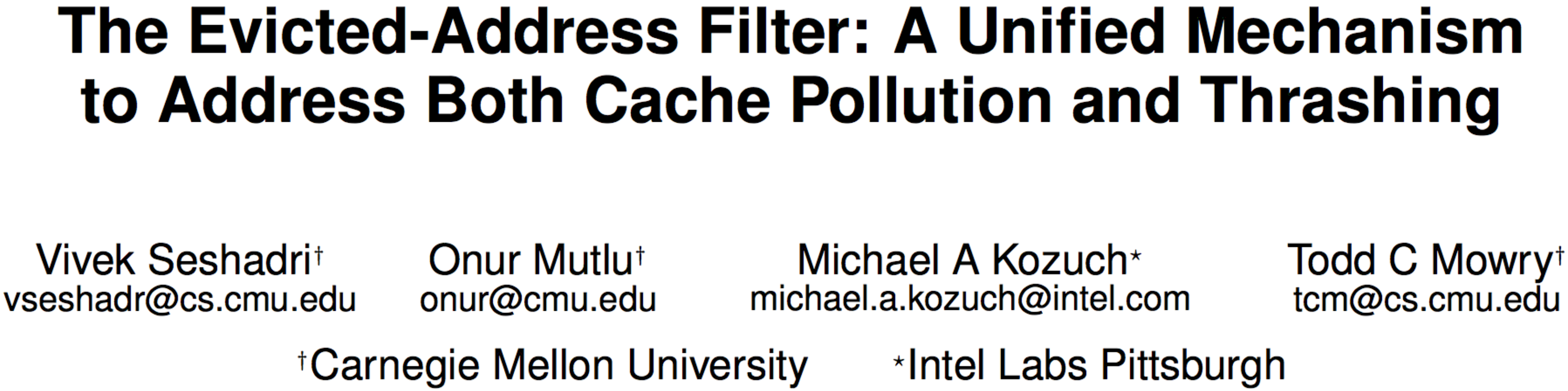
1. Simple to implement
2. Easy to design and verify
3. Works with other techniques (replacement policy)

EAF Performance – Summary



More on Evicted Address Filter Cache

- 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 International Conference on Parallel Architectures and Compilation Techniques (**PACT**)*, Minneapolis, MN, September 2012. Slides (pptx) Source Code



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

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

[†]Carnegie Mellon University [⋆]Intel Labs Pittsburgh

Cache Compression

Motivation for Cache Compression

Significant redundancy in data:

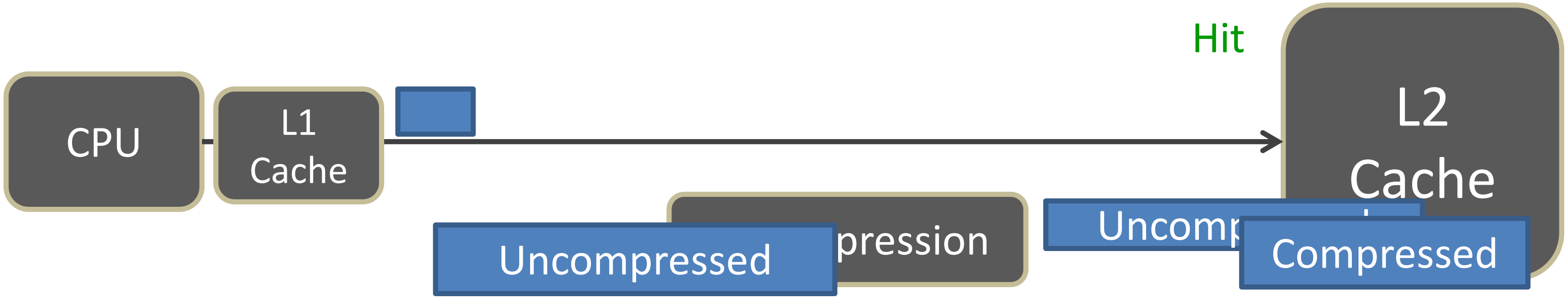
0x**000000**00	0x**000000**0B	0x**000000**03	0x**000000**04	…

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	✓	✓	✓

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

Key Data Patterns in Real Applications

Zero Values: initialization, sparse matrices, NULL pointers

0x00000000	0x00000000	0x00000000	0x00000000	…

Repeated Values: common initial values, adjacent pixels

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

Narrow Values: small values stored in a big data type

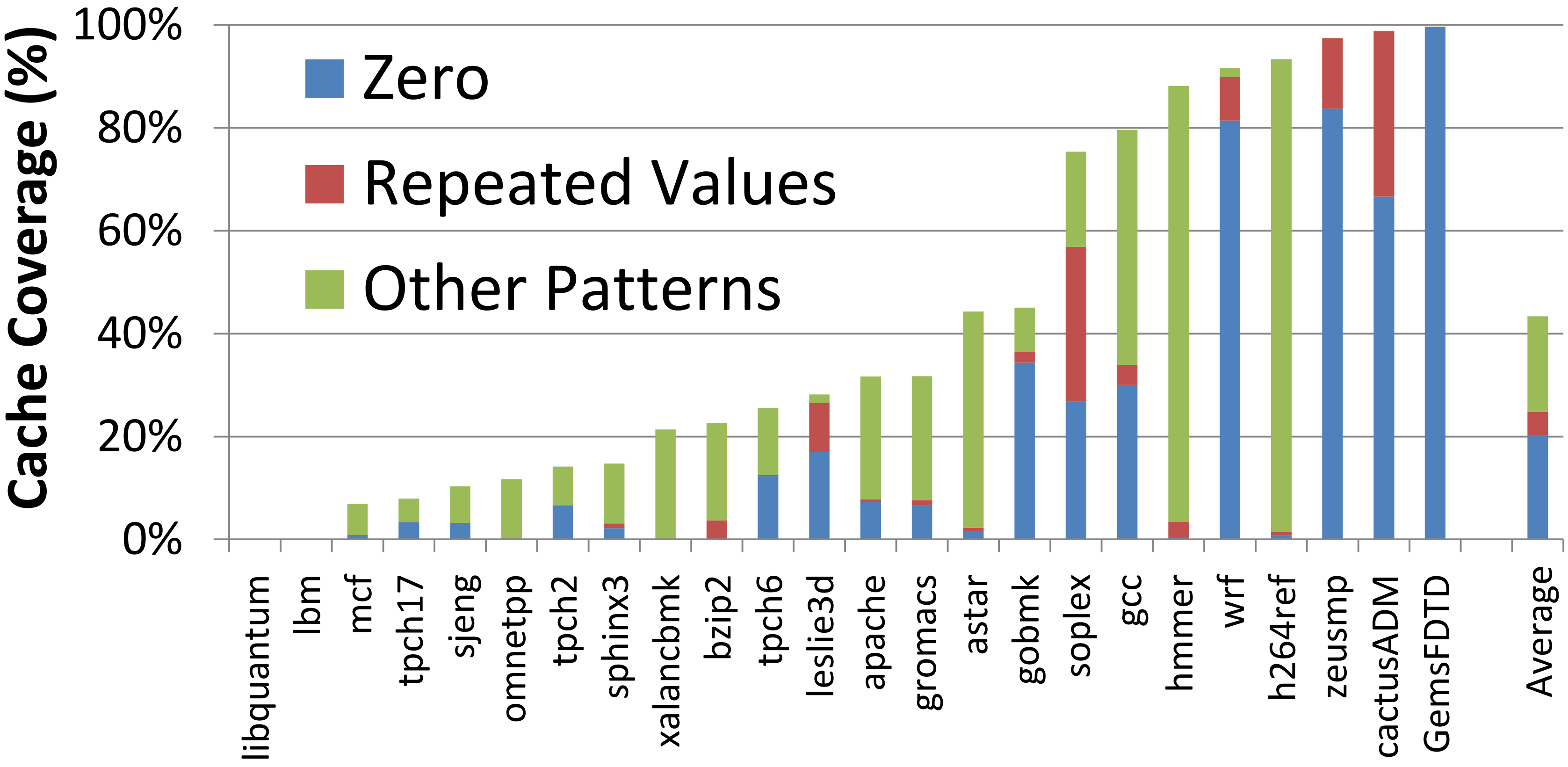
0x*000000***00**	0x*000000***0B**	0x*000000***03**	0x*000000***04**	…

Other Patterns: pointers to the same memory region

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

How Common Are These Patterns?

SPEC2006, databases, web workloads, 2MB L2 cache
"Other Patterns" include Narrow Values



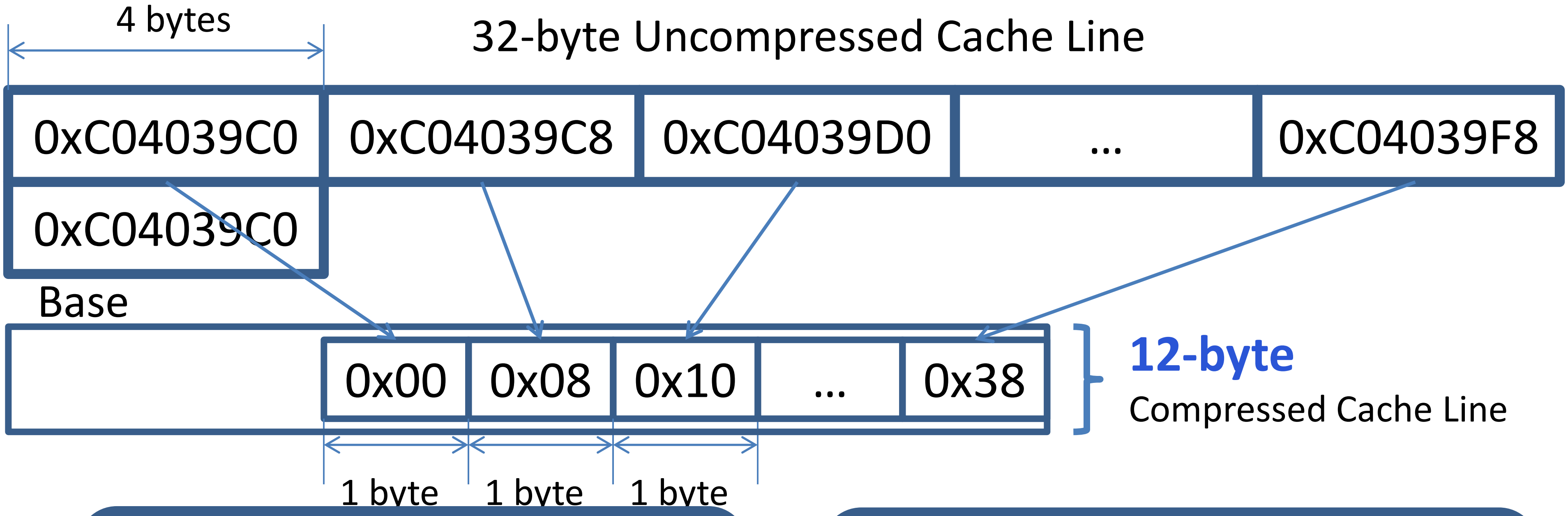
43% of the cache lines belong to key patterns

Key Data Patterns in Real Applications

Low Dynamic Range:

Differences between values are significantly smaller than the values themselves

Key Idea: Base+Delta (B+Δ) Encoding



✓ **Fast Decompression:** vector addition

✓ **Simple Hardware:** arithmetic and comparison

✓ **Effective:** good compression ratio

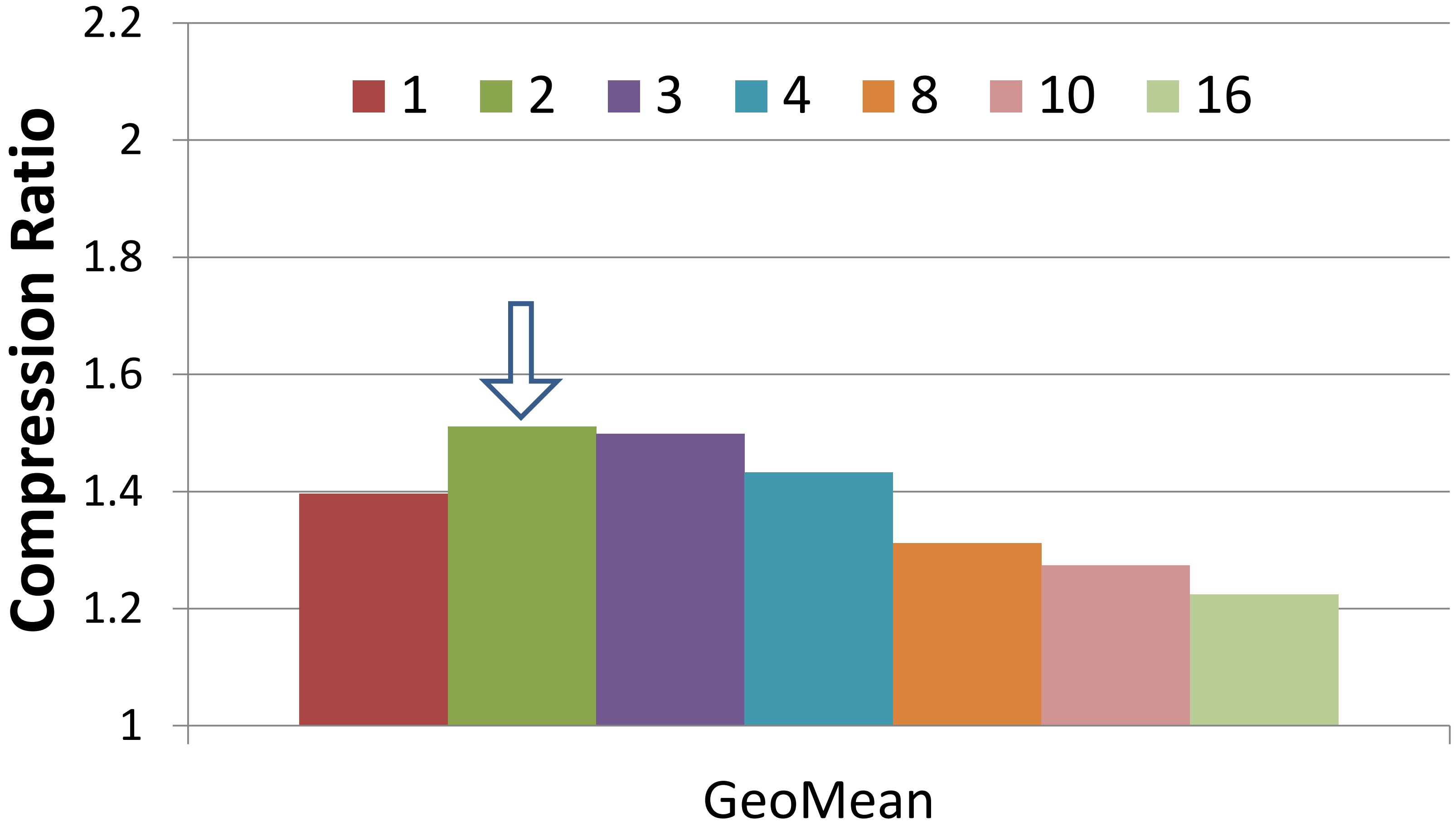
Can We Do Better?

- Uncompressible cache line (with a single base):

0x00000000	**0x09A4**0178	**0x000000**0B	**0x09A4**A838	…

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

B+Δ with Multiple Arbitrary Bases



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

How to Find Two Bases Efficiently?

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

✓ **Base+Delta part**

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

✓ **Immediate part**

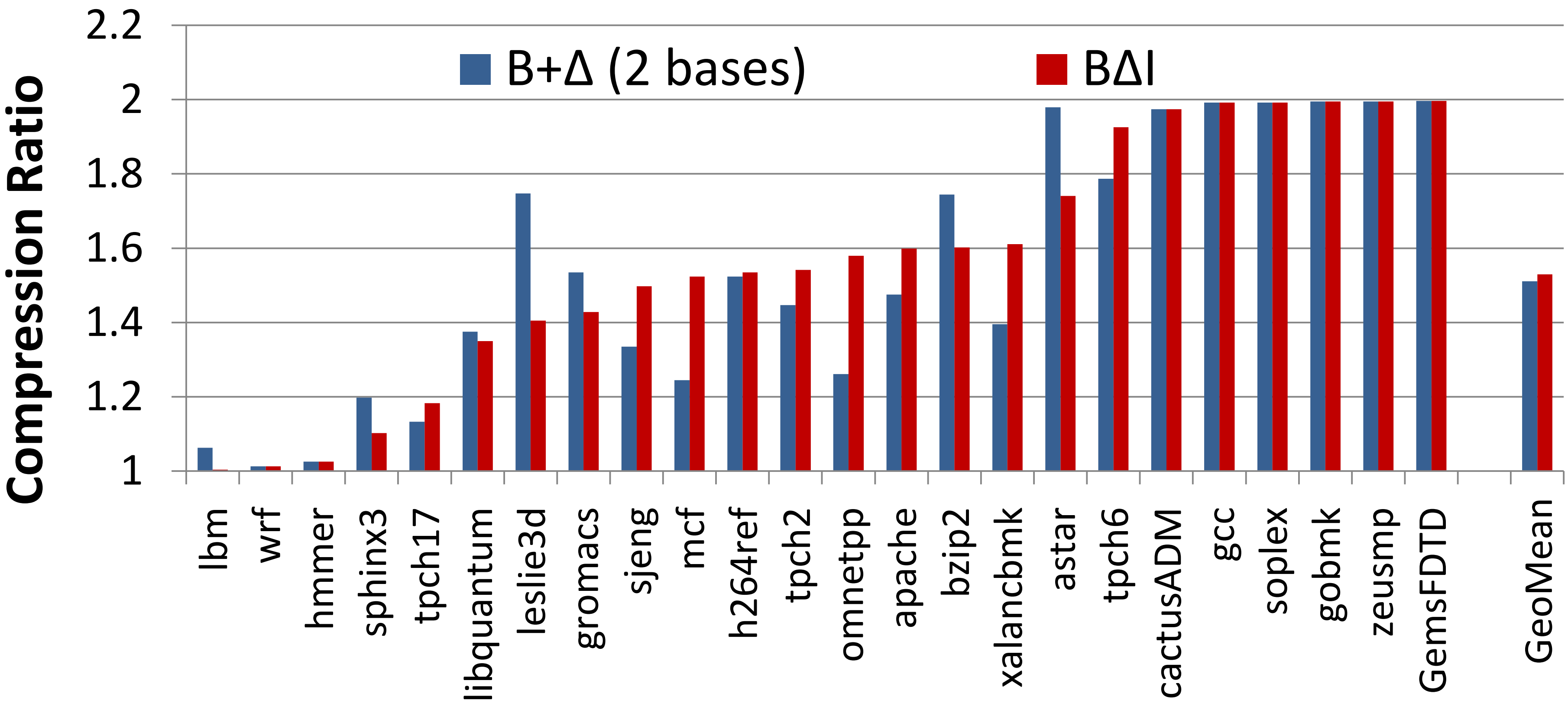
Advantages over 2 arbitrary bases:

– Better compression ratio

– Simpler compression logic

Base-Delta-Immediate (BΔI) Compression

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



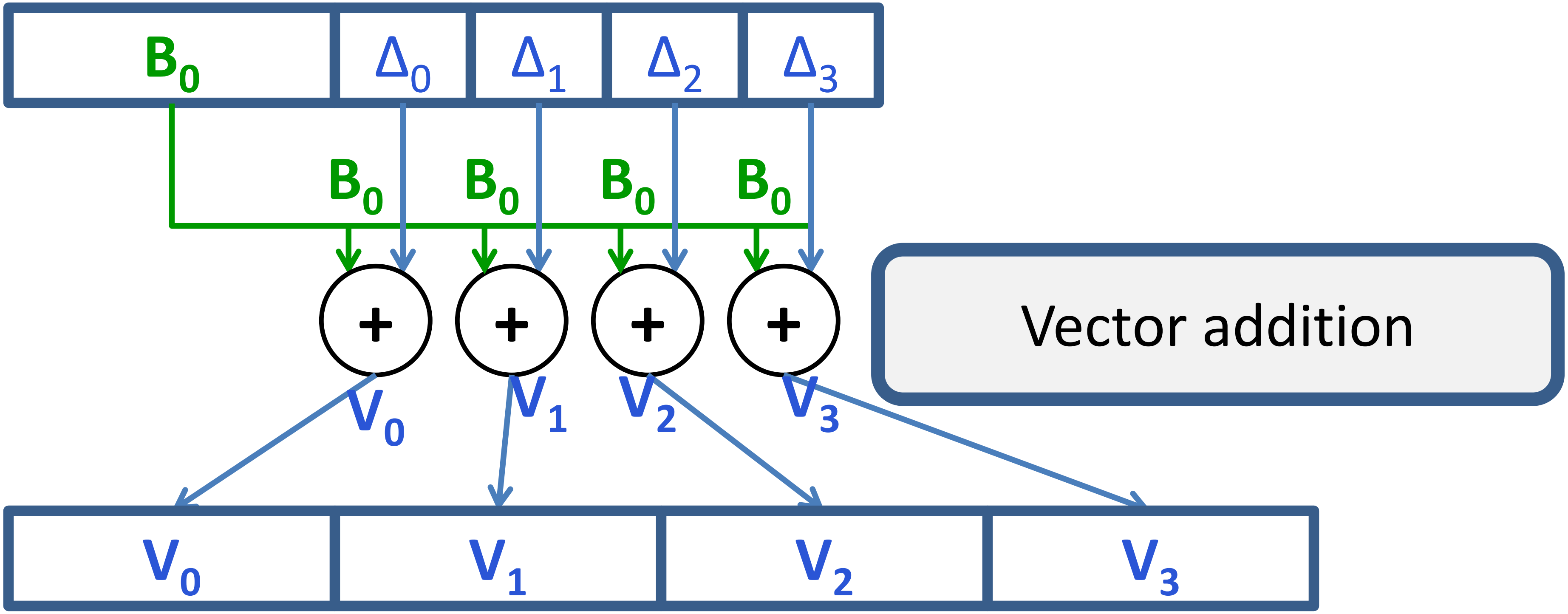
Average compression ratio is close, but **BΔI** is **simpler**

BΔI Cache Compression Implementation

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

BΔI Decompressor Design

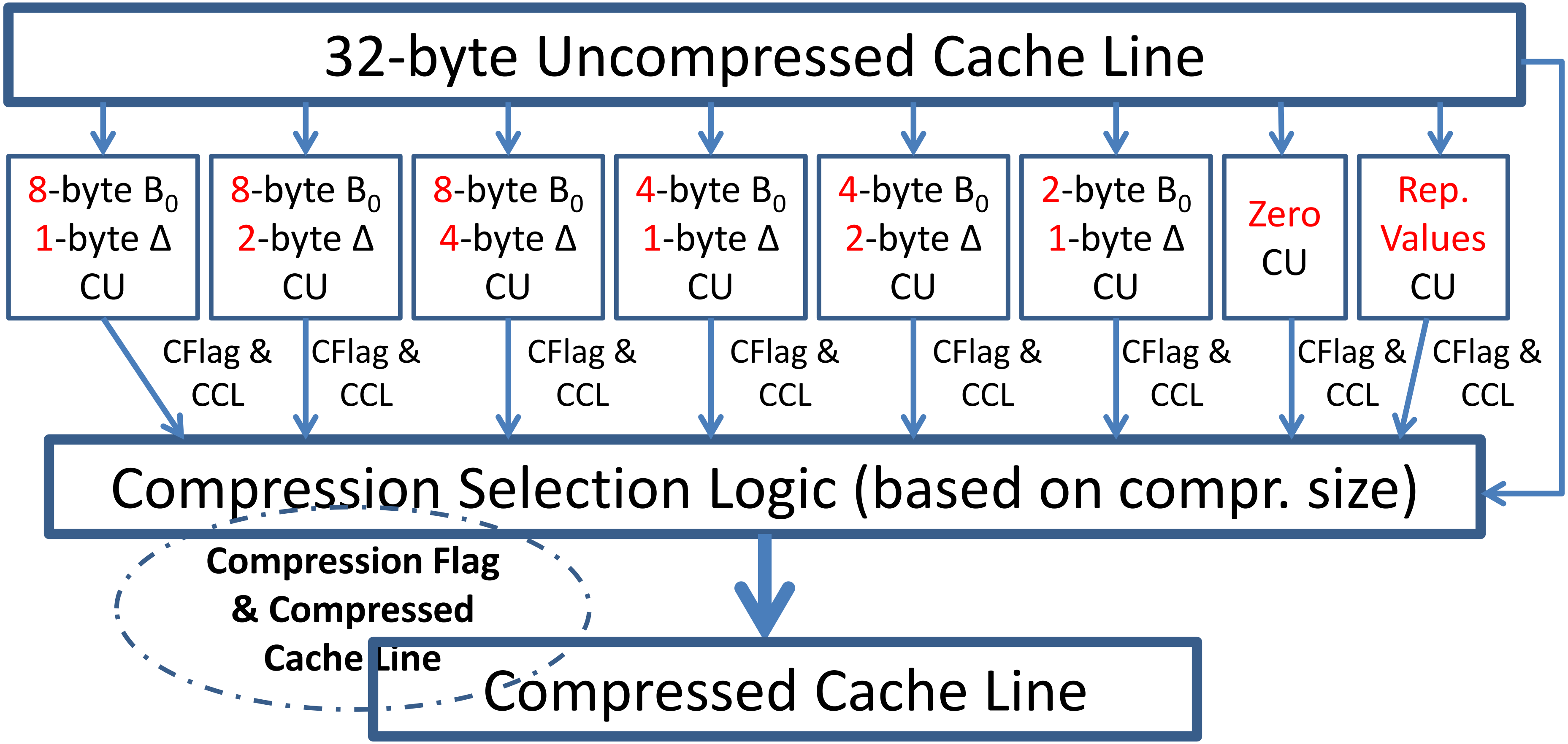
Compressed Cache Line



Vector addition

Uncompressed Cache Line

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

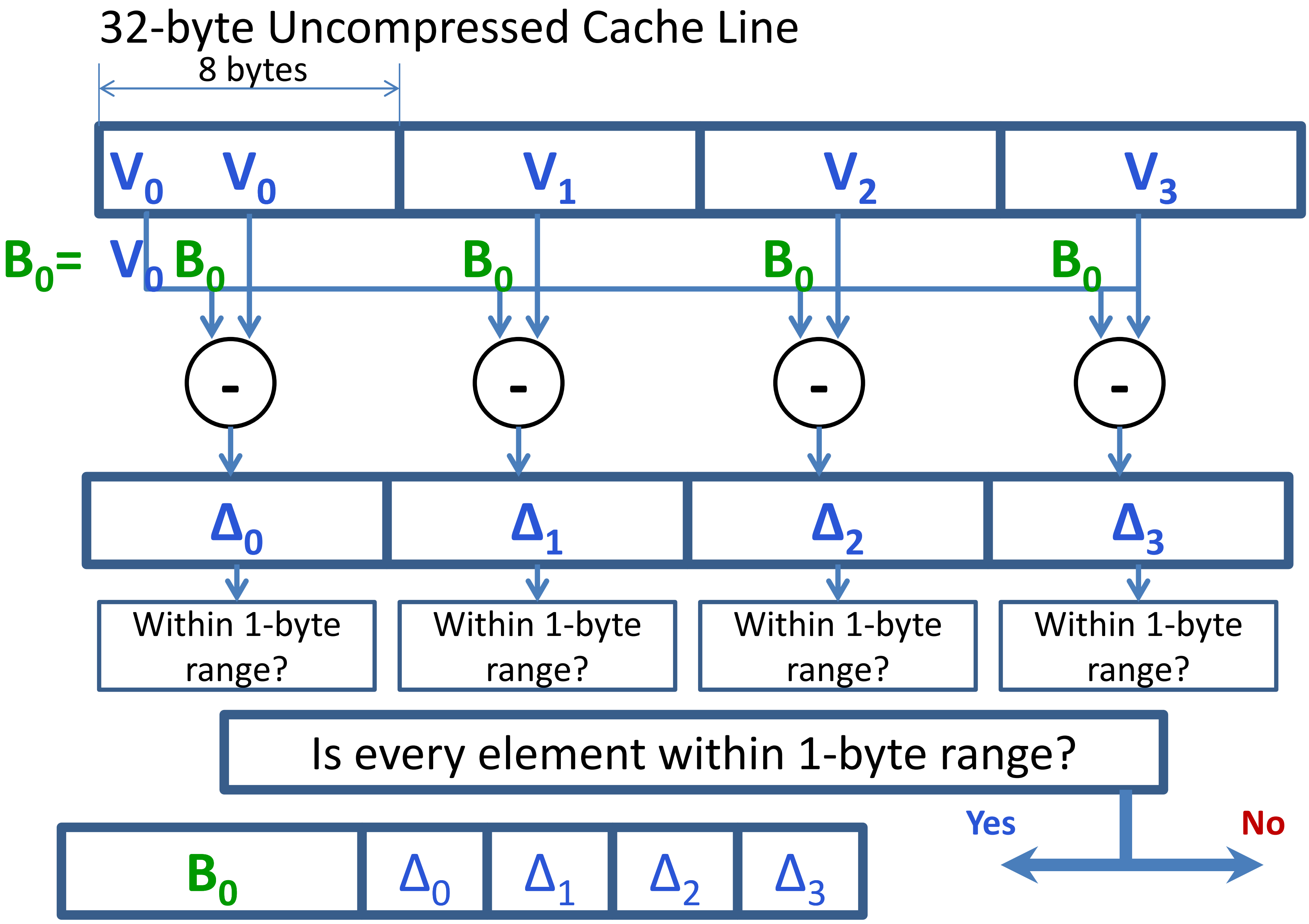
CFlag & CCL

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

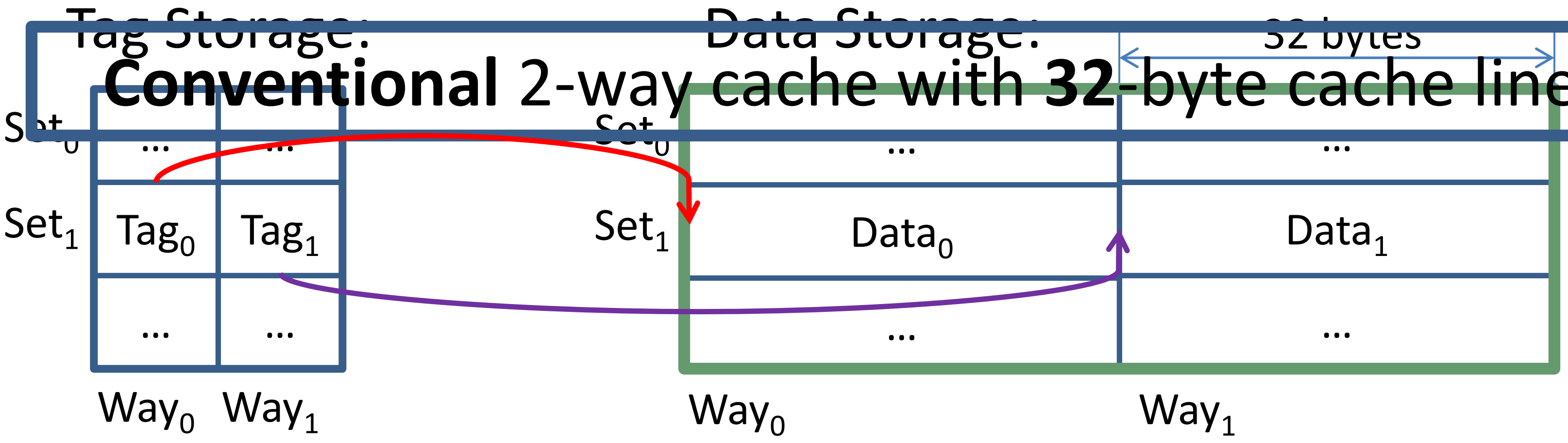


32-byte Uncompressed Cache Line

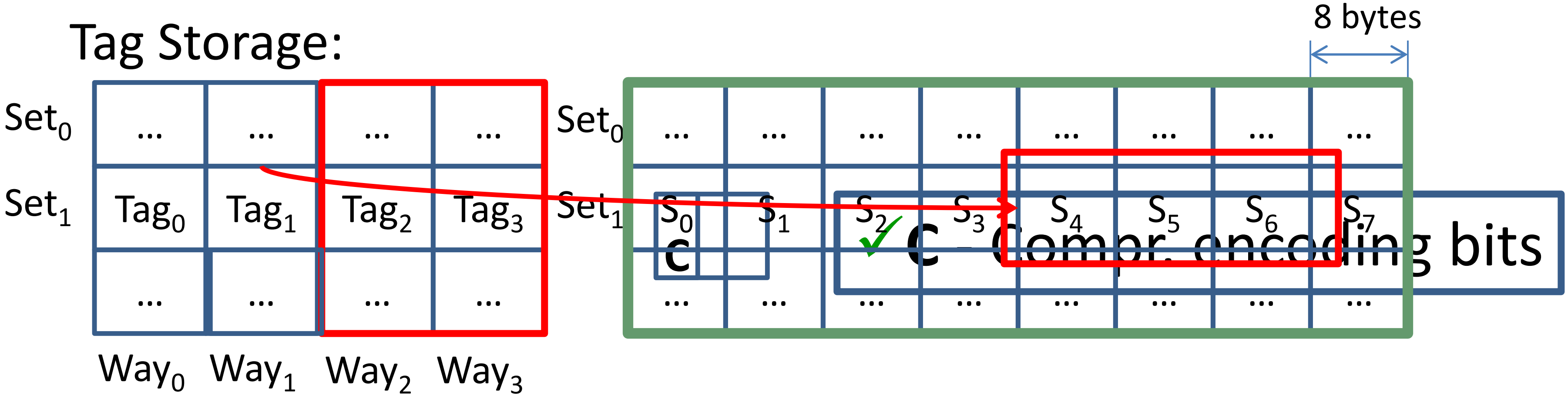
Is every element within 1-byte range?

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

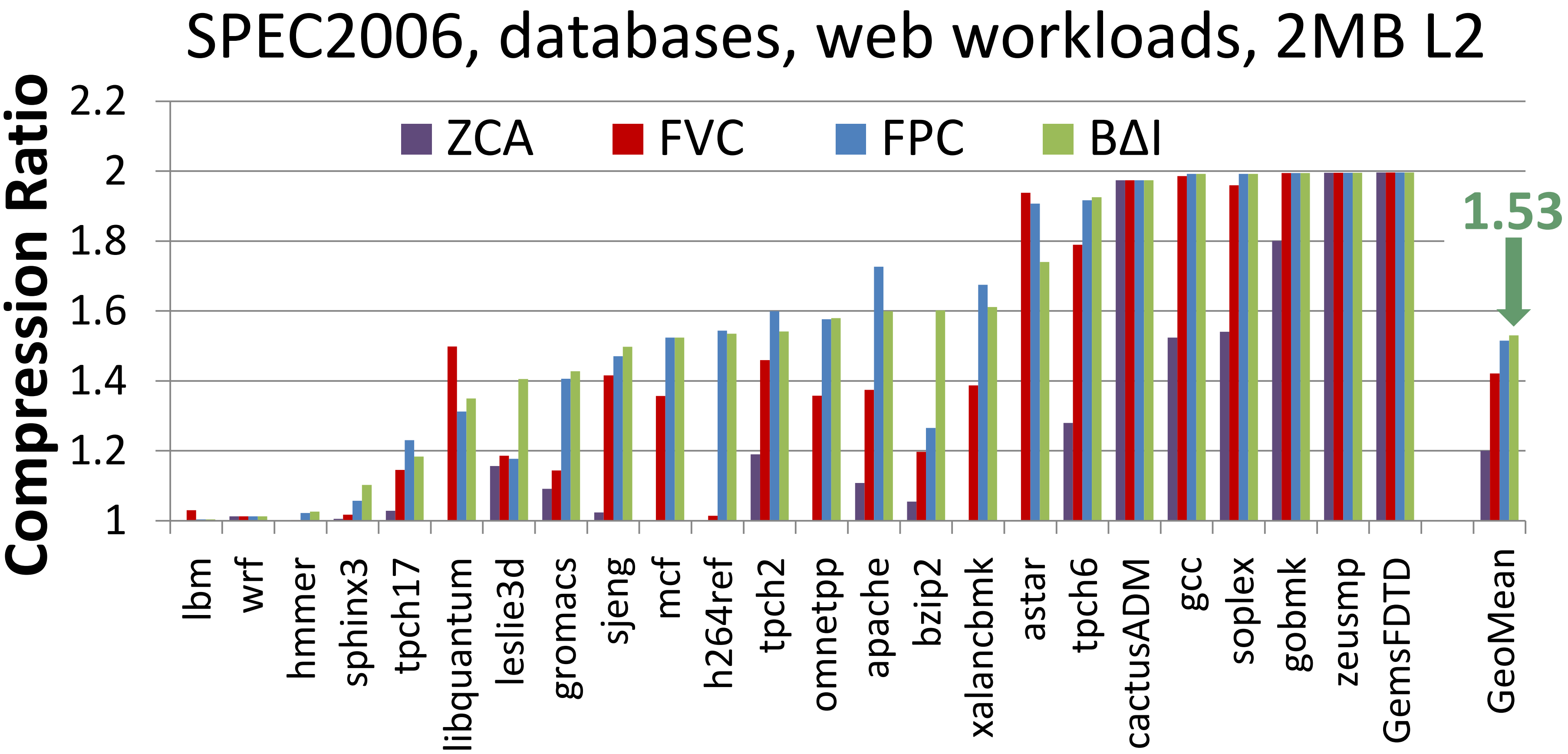
✓Tags map to multiple adjacent segments

2.3% overhead for 2 MB cache

Qualitative Comparison with Prior Work

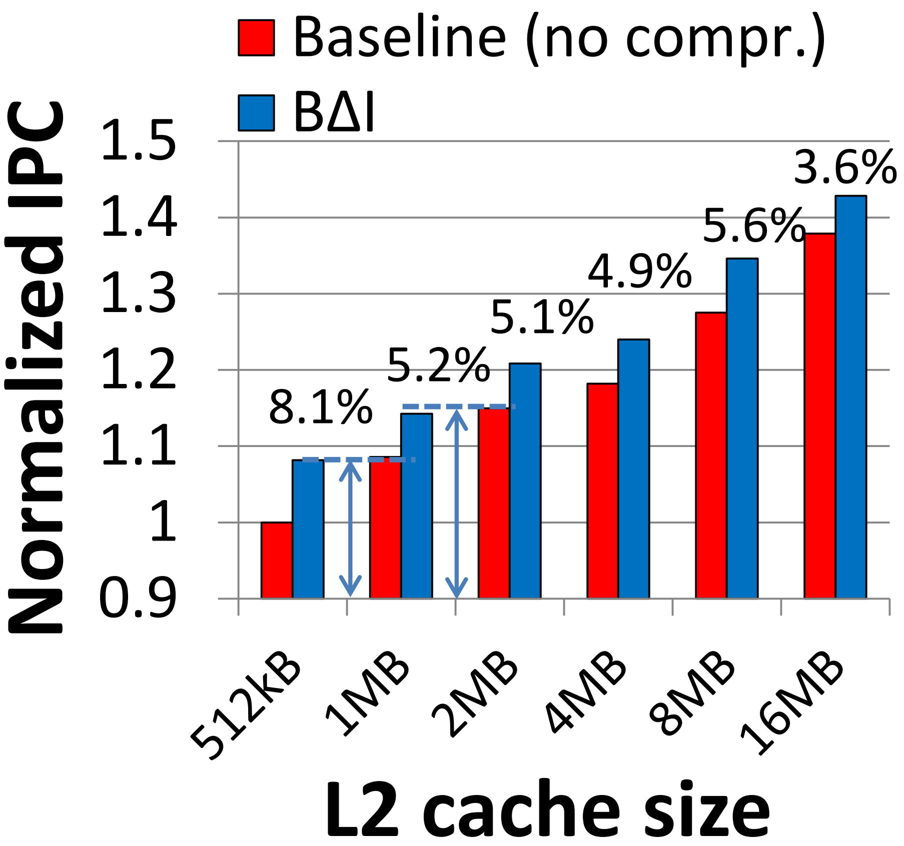
- **Zero-based designs**
  - ZCA *[Dusser+, ICS'09]*: zero-content augmented cache
  - ZVC *[Islam+, PACT'09]*: zero-value cancelling
  - Limited applicability (only zero values)
- **FVC** *[Yang+, MICRO'00]*: frequent value compression
  - High decompression latency and complexity
- **Pattern-based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

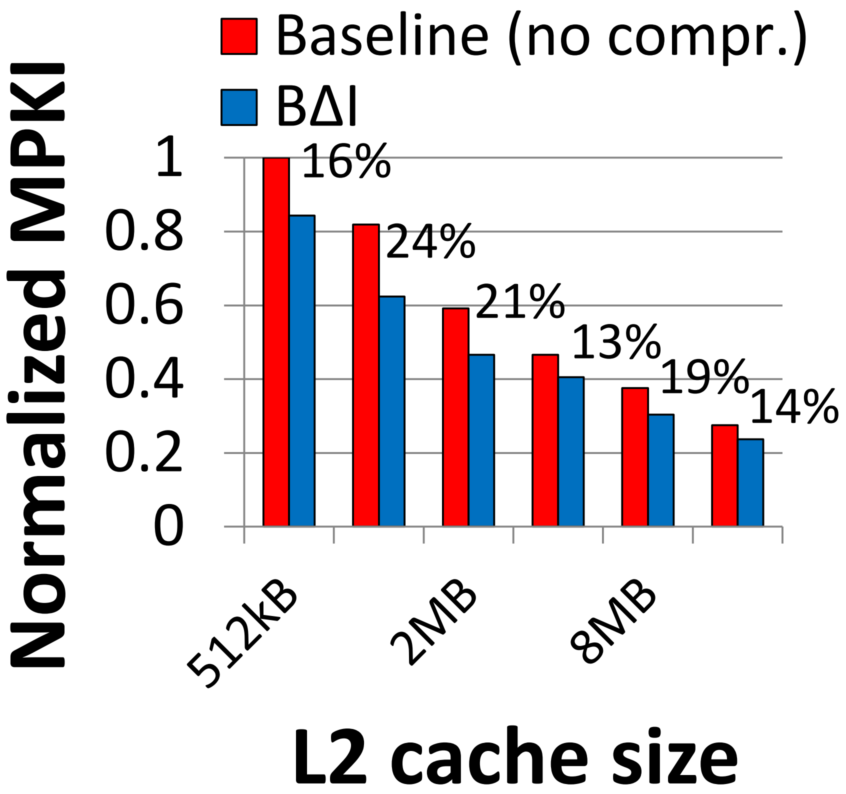
Cache Compression Ratios



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)

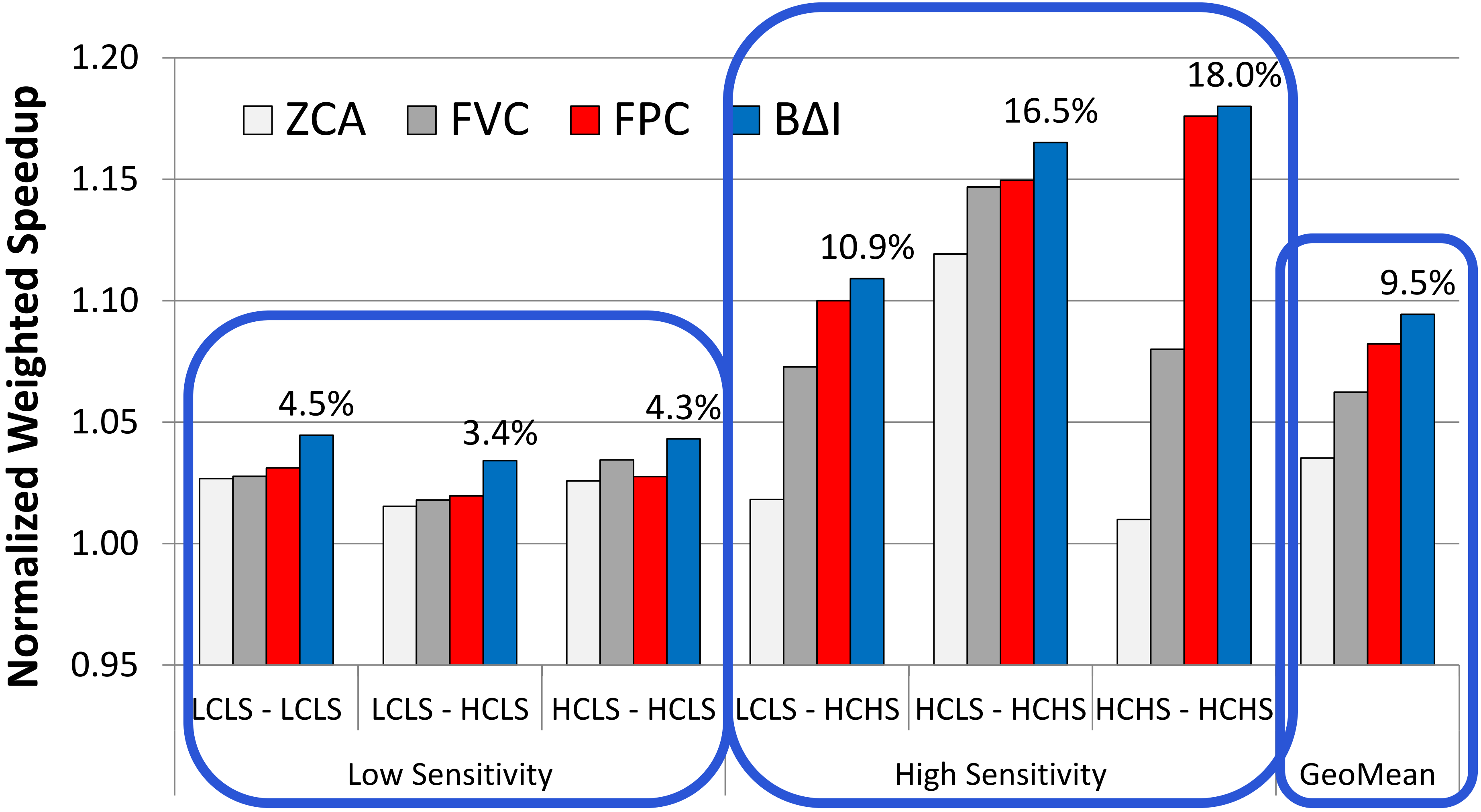
  Sensitivity: performance gain with more cache

  (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%**)

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

Readings on Memory Compression (I)

- 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 International Conference on Parallel Architectures and Compilation Techniques (**PACT**)*, Minneapolis, MN, September 2012. Slides (pptx) Source Code

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

Gennady Pekhimenko[†]
gpekhime@cs.cmu.edu

Vivek Seshadri[†]
vseshadr@cs.cmu.edu

Onur Mutlu[†]
onur@cmu.edu

Michael A. Kozuch[*]
michael.a.kozuch@intel.com

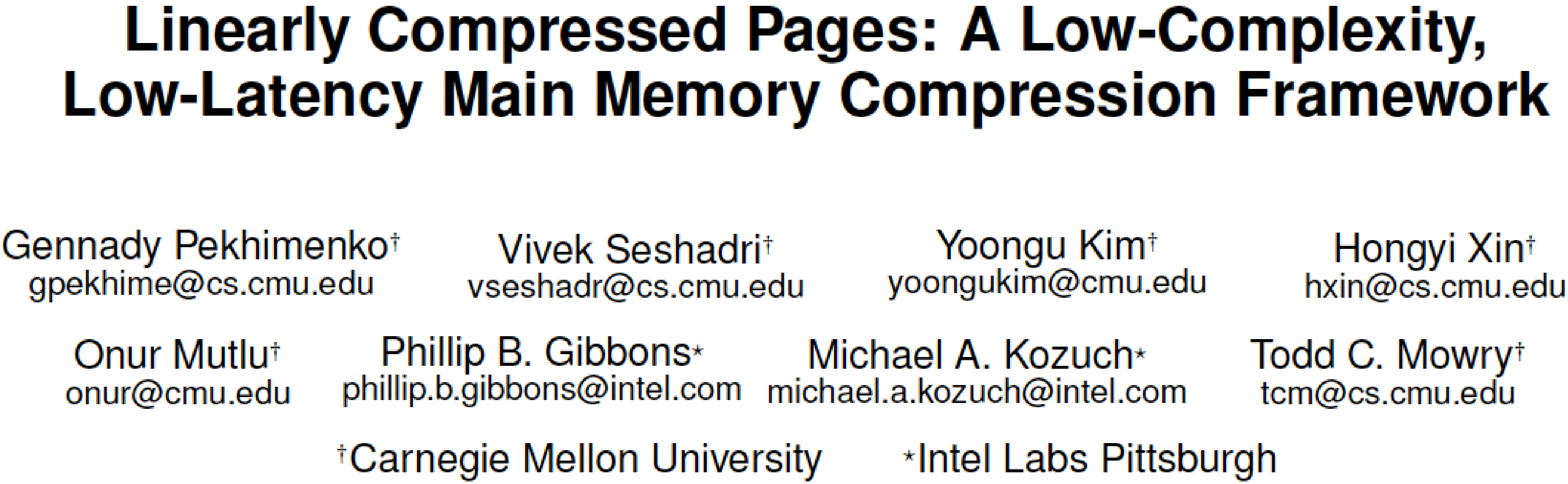
Phillip B. Gibbons[*]
phillip.b.gibbons@intel.com

Todd C. Mowry[†]
tcm@cs.cmu.edu

[†]Carnegie Mellon University [*]Intel Labs Pittsburgh

Readings on Memory Compression (II)

- Gennady Pekhimenko, Vivek Seshadri, Yoongu Kim, Hongyi Xin, Onur Mutlu, Michael A. Kozuch, Phillip B. Gibbons, and Todd C. Mowry, **"Linearly Compressed Pages: A Low-Complexity, Low-Latency Main Memory Compression Framework"** *Proceedings of the 46th International Symposium on Microarchitecture (**MICRO**)*, Davis, CA, December 2013. [Slides (pptx) (pdf)] [Lightning Session Slides (pptx) (pdf)] Poster (pptx) (pdf)]



Linearly Compressed Pages: A Low-Complexity, Low-Latency Main Memory Compression Framework

Gennady Pekhimenko[†]
gpekhime@cs.cmu.edu

Vivek Seshadri[†]
vseshadr@cs.cmu.edu

Yoongu Kim[†]
yoongukim@cmu.edu

Hongyi Xin[†]
hxin@cs.cmu.edu

Onur Mutlu[†]
onur@cmu.edu

Phillip B. Gibbons[*]
phillip.b.gibbons@intel.com

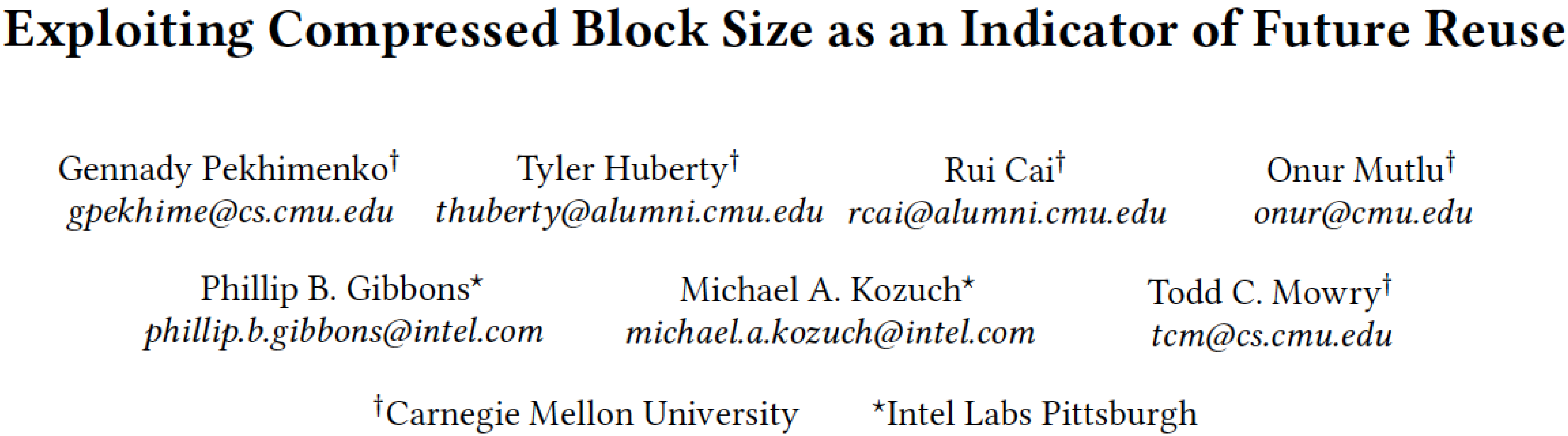
Michael A. Kozuch[*]
michael.a.kozuch@intel.com

Todd C. Mowry[†]
tcm@cs.cmu.edu

[†]Carnegie Mellon University [*]Intel Labs Pittsburgh

Readings on Memory Compression (III)

- Gennady Pekhimenko, Tyler Huberty, Rui Cai, Onur Mutlu, Phillip P. Gibbons, Michael A. Kozuch, and Todd C. Mowry,
"Exploiting Compressed Block Size as an Indicator of Future Reuse"
*Proceedings of the 21st International Symposium on High-Performance Computer Architecture (**HPCA**)*, Bay Area, CA, February 2015.
[Slides (pptx) (pdf)]



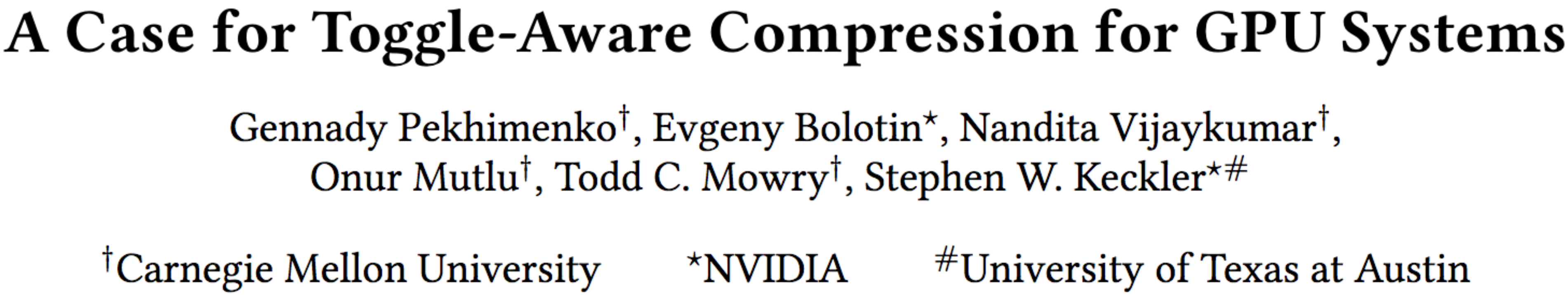
Exploiting Compressed Block Size as an Indicator of Future Reuse

Gennady Pekhimenko[†] gpekhime@cs.cmu.edu
Tyler Huberty[†] thuberty@alumni.cmu.edu
Rui Cai[†] rcai@alumni.cmu.edu
Onur Mutlu[†] onur@cmu.edu
Phillip B. Gibbons[*] phillip.b.gibbons@intel.com
Michael A. Kozuch[*] michael.a.kozuch@intel.com
Todd C. Mowry[†] tcm@cs.cmu.edu

[†]Carnegie Mellon University [*]Intel Labs Pittsburgh

Readings on Memory Compression (IV)

- Gennady Pekhimenko, Evgeny Bolotin, Nandita Vijaykumar, Onur Mutlu, Todd C. Mowry, and Stephen W. Keckler, **"A Case for Toggle-Aware Compression for GPU Systems"** *Proceedings of the 22nd International Symposium on High-Performance Computer Architecture (**HPCA**)*, Barcelona, Spain, March 2016. [Slides (pptx) (pdf)]



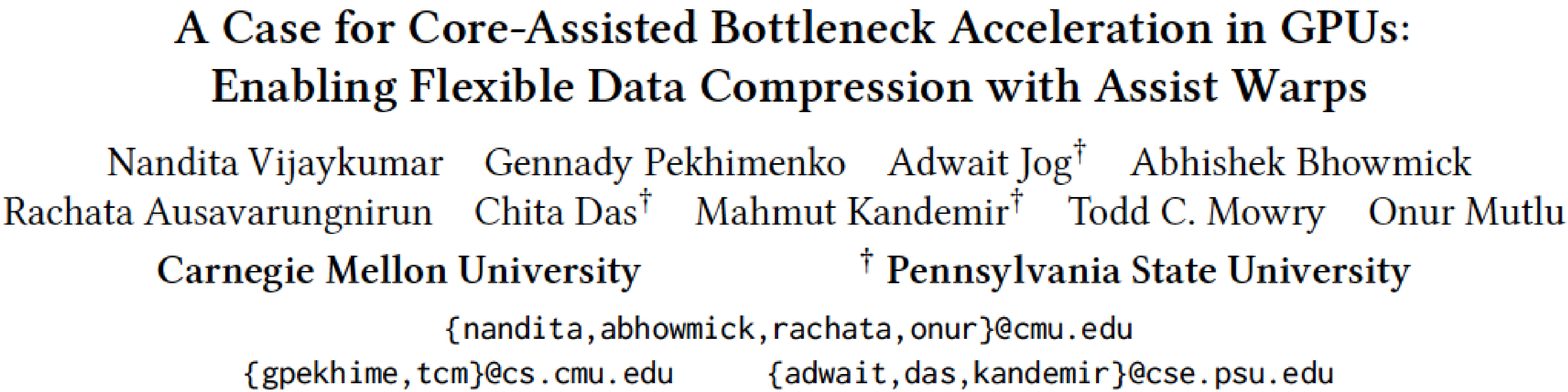
A Case for Toggle-Aware Compression for GPU Systems

Gennady Pekhimenko[†], Evgeny Bolotin[*], Nandita Vijaykumar[†],
Onur Mutlu[†], Todd C. Mowry[†], Stephen W. Keckler[*#]

[†]Carnegie Mellon University [*]NVIDIA [#]University of Texas at Austin

Readings on Memory Compression (V)

- Nandita Vijaykumar, Gennady Pekhimenko, Adwait Jog, Abhishek Bhowmick, Rachata Ausavarungnirun, Chita Das, Mahmut Kandemir, Todd C. Mowry, and Onur Mutlu,
"A Case for Core-Assisted Bottleneck Acceleration in GPUs: Enabling Flexible Data Compression with Assist Warps"
Proceedings of the 42nd International Symposium on Computer Architecture (***ISCA***), Portland, OR, June 2015.
[Slides (pptx) (pdf)] [Lightning Session Slides (pptx) (pdf)]



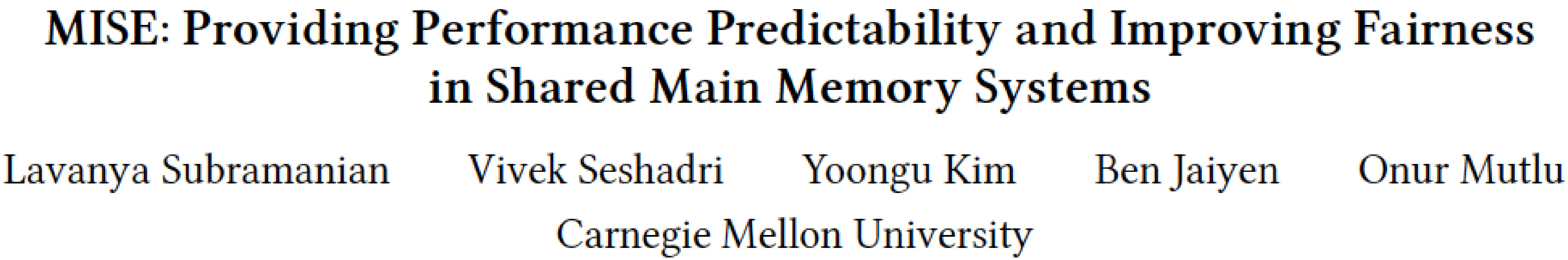
A Case for Core-Assisted Bottleneck Acceleration in GPUs:
Enabling Flexible Data Compression with Assist Warps

Nandita Vijaykumar Gennady Pekhimenko Adwait Jog[†] Abhishek Bhowmick
Rachata Ausavarungnirun Chita Das[†] Mahmut Kandemir[†] Todd C. Mowry Onur Mutlu
Carnegie Mellon University [†] Pennsylvania State University
{nandita,abhowmick,rachata,onur}@cmu.edu
{gpekhime,tcm}@cs.cmu.edu {adwait,das,kandemir}@cse.psu.edu

Predictable Performance Again: Strong Memory Service Guarantees

Remember MISE?

- Lavanya Subramanian, Vivek Seshadri, Yoongu Kim, Ben Jaiyen, and Onur Mutlu,
"MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems"
*Proceedings of the 19th International Symposium on High-Performance Computer Architecture (**HPCA**)*, Shenzhen, China, February 2013. Slides (pptx)



MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems

Lavanya Subramanian Vivek Seshadri Yoongu Kim Ben Jaiyen Onur Mutlu

Carnegie Mellon University

Extending Slowdown Estimation to Caches

- How do we extend the MISE model to include shared cache interference?

- Answer: Application Slowdown Model

- Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, and Onur Mutlu,
"The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-Application Interference at Shared Caches and Main Memory"
*Proceedings of the 48th International Symposium on Microarchitecture (**MICRO**)*, Waikiki, Hawaii, USA, December 2015.
[Slides (pptx) (pdf)] [Lightning Session Slides (pptx) (pdf)] [Poster (pptx) (pdf)]
[Source Code]

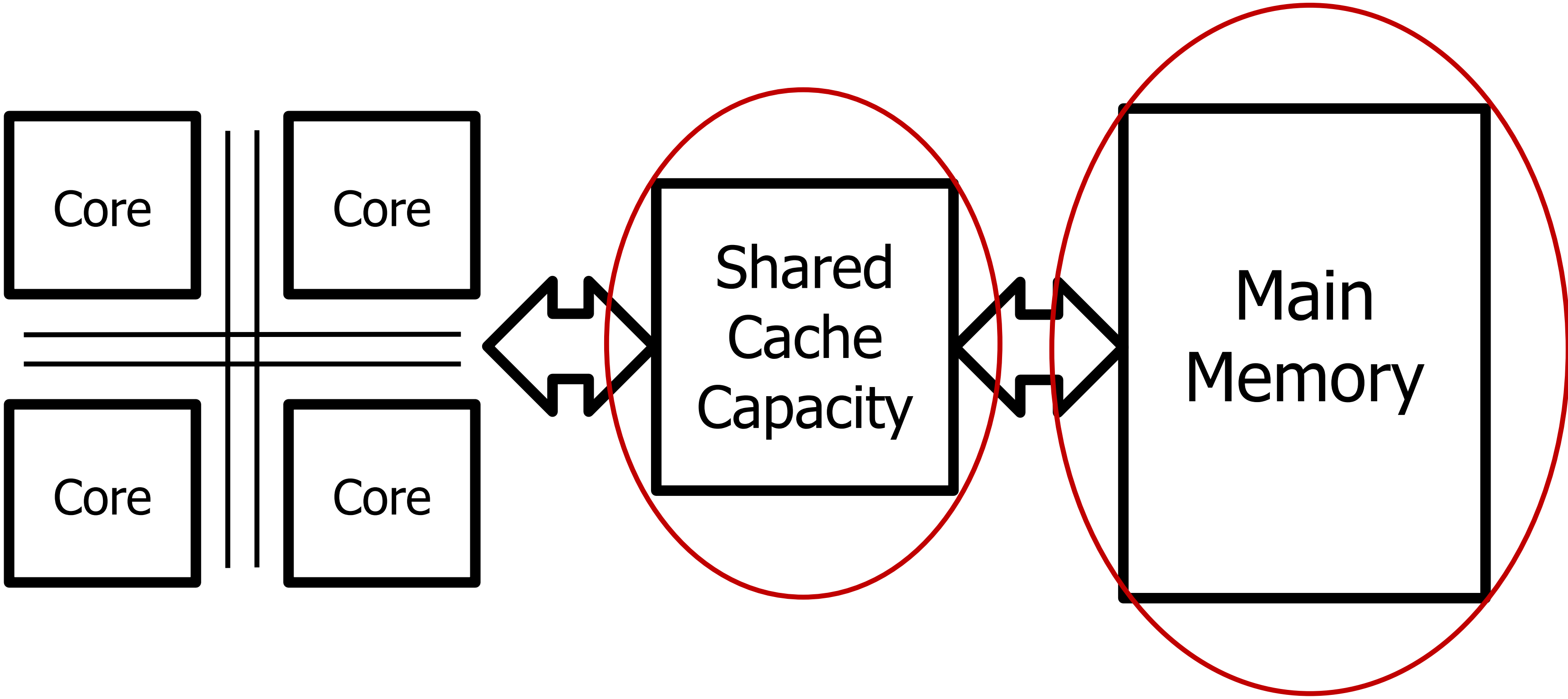
Application Slowdown Model

Quantifying and Controlling Impact of Interference at Shared Caches and Main Memory

Lavanya Subramanian, Vivek Seshadri,
Arnab Ghosh, Samira Khan, Onur Mutlu

SAFARI Carnegie Mellon intel University of Virginia 1819

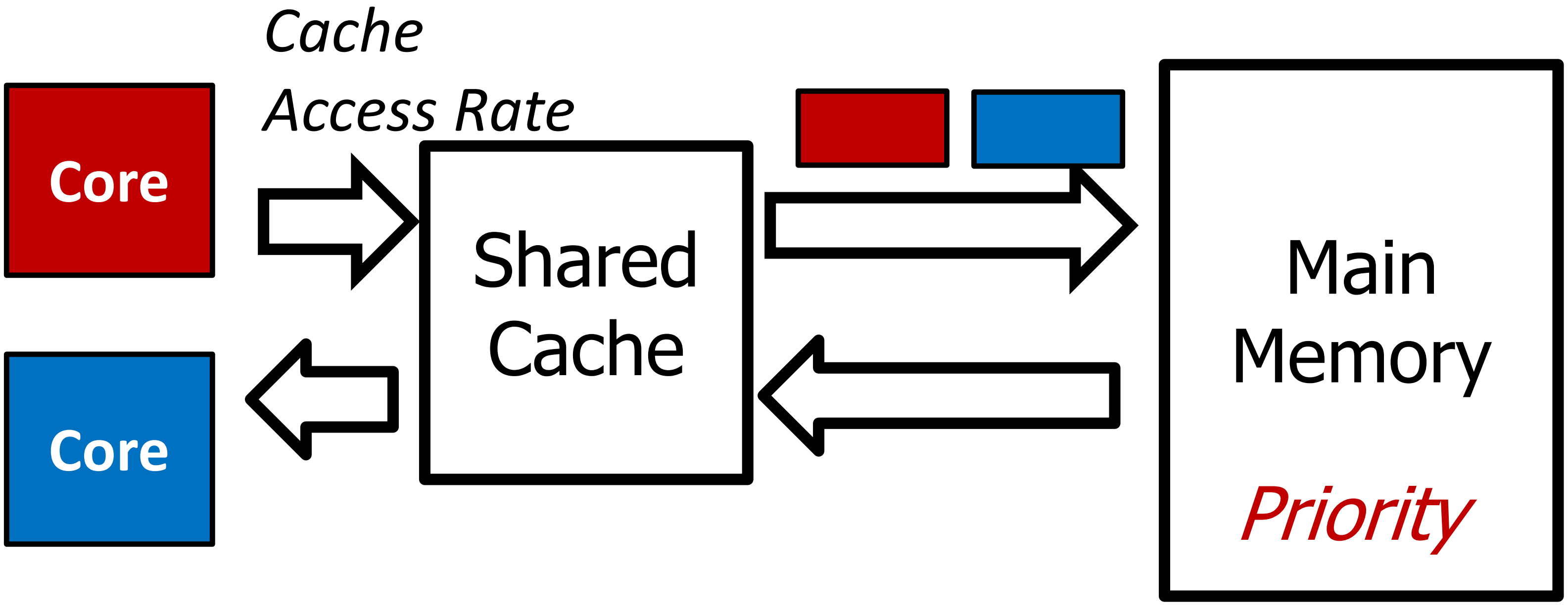
Shared Cache and Memory Contention



$$\text{Slowdown} = \frac{\text{Request Service Rate}_{\text{Alone}}}{\text{Request Service Rate}_{\text{Shared}}}$$

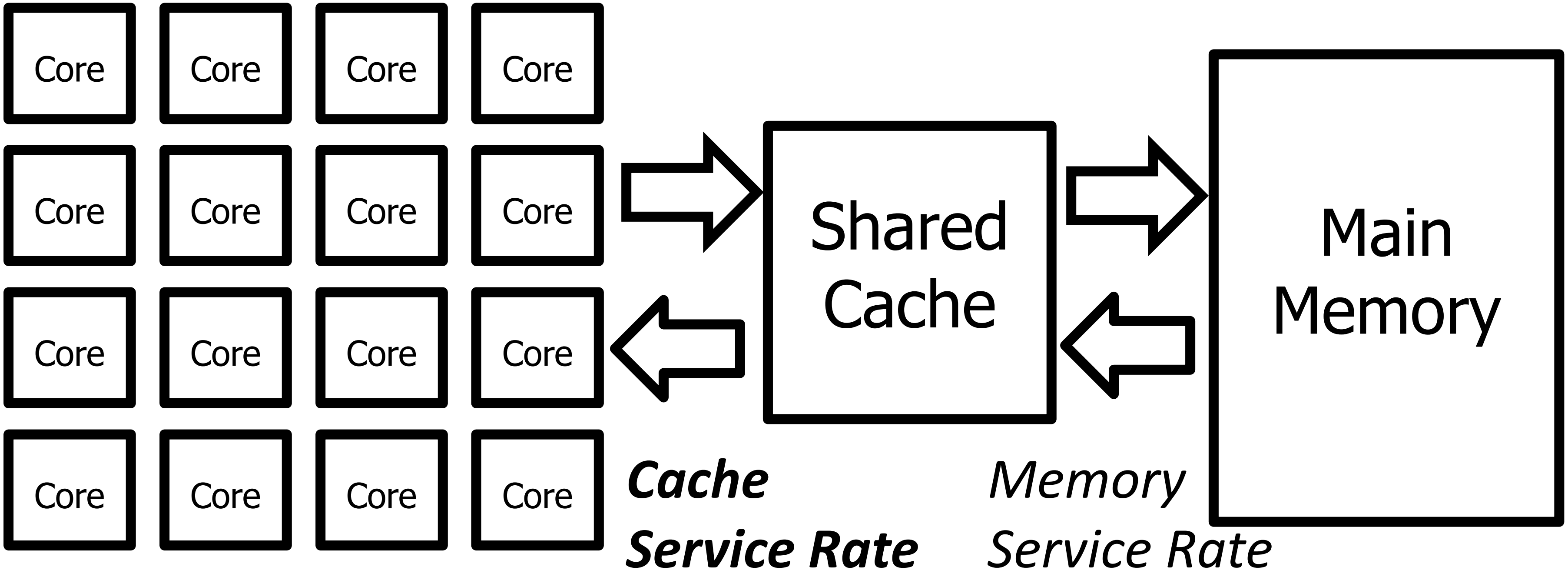
MISE [HPCA'13]

Cache Capacity Contention



Applications evict each other's blocks from the shared cache

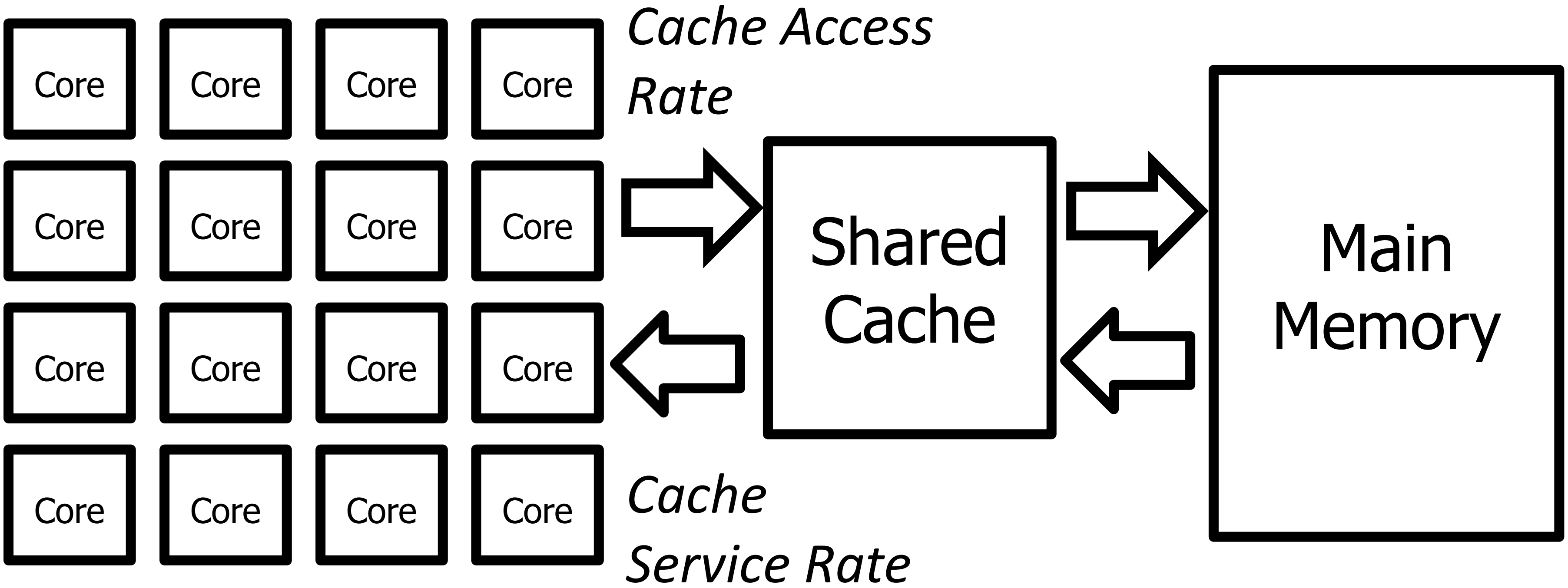
Estimating Cache and Memory Slowdowns



Cache Service Rate

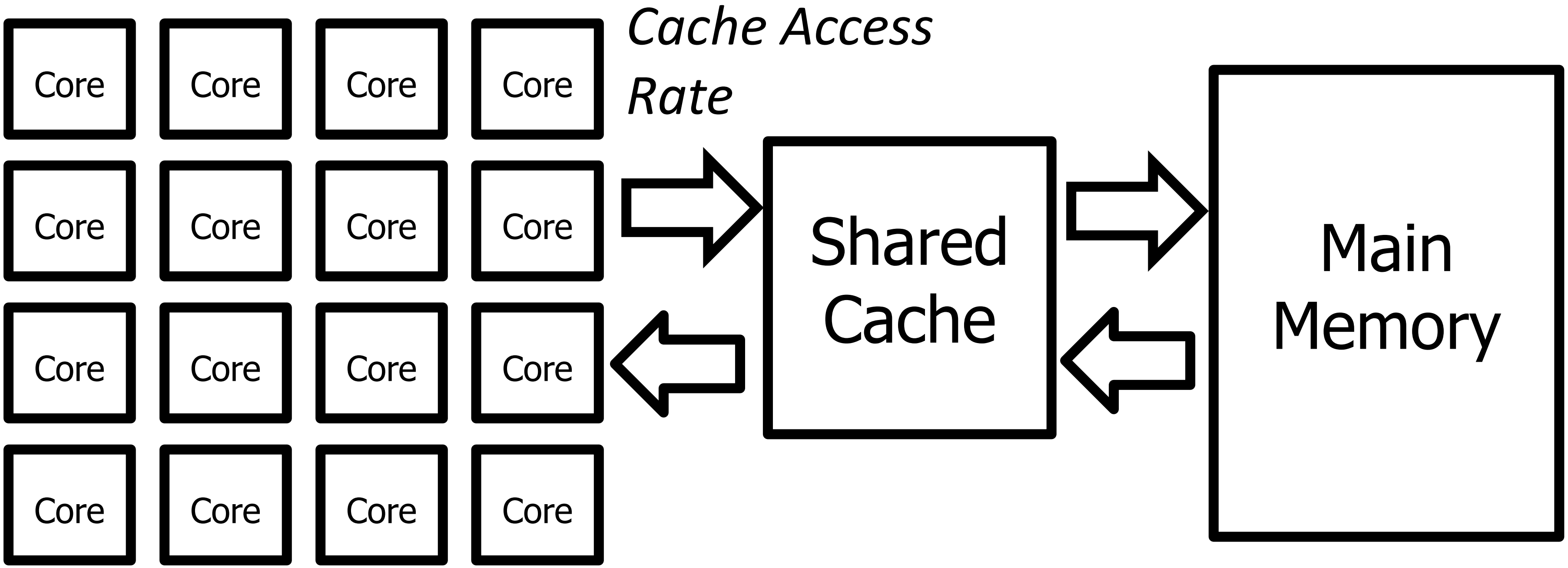
Memory Service Rate

Service Rates vs. Access Rates



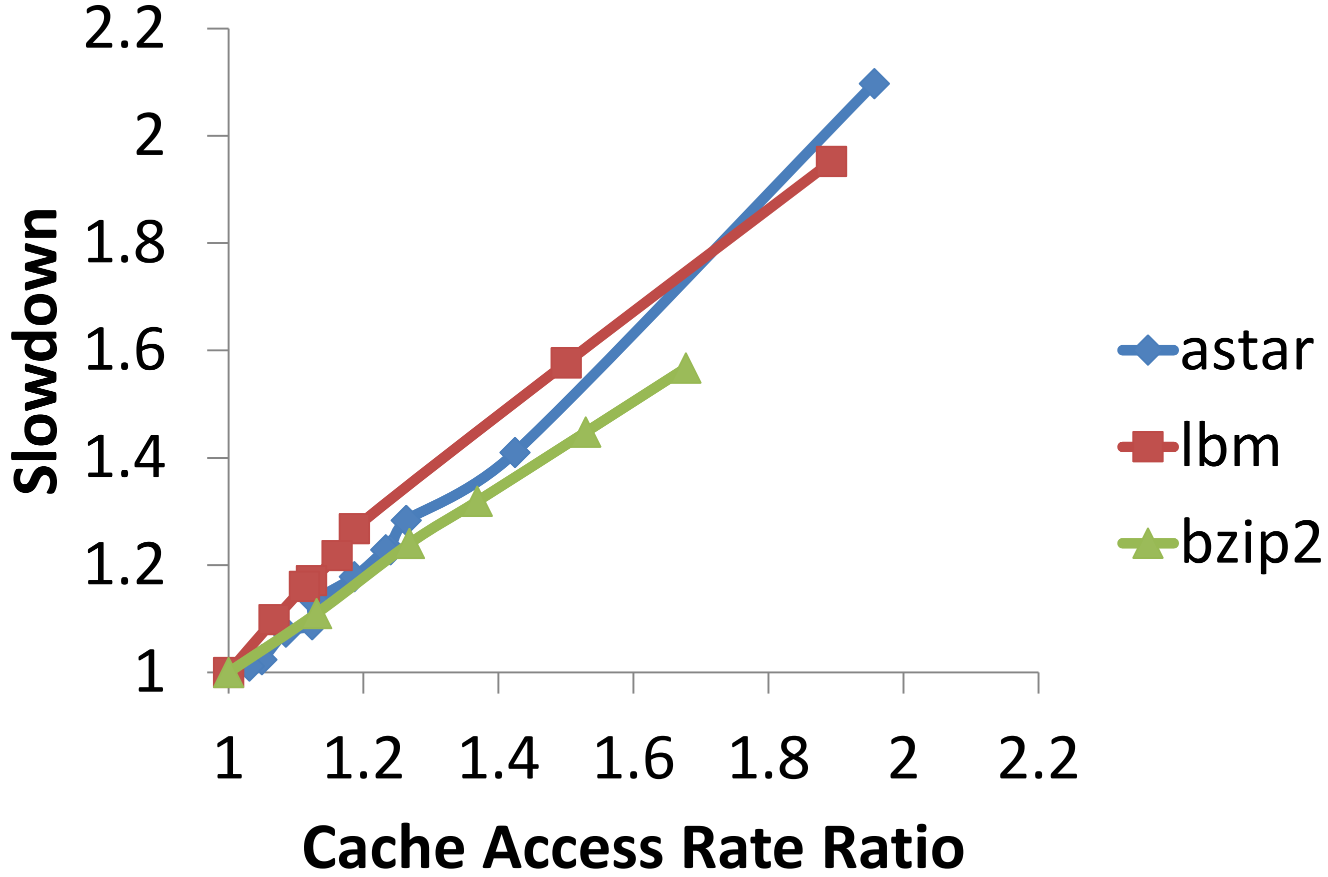
Request service and access rates are tightly coupled

The Application Slowdown Model



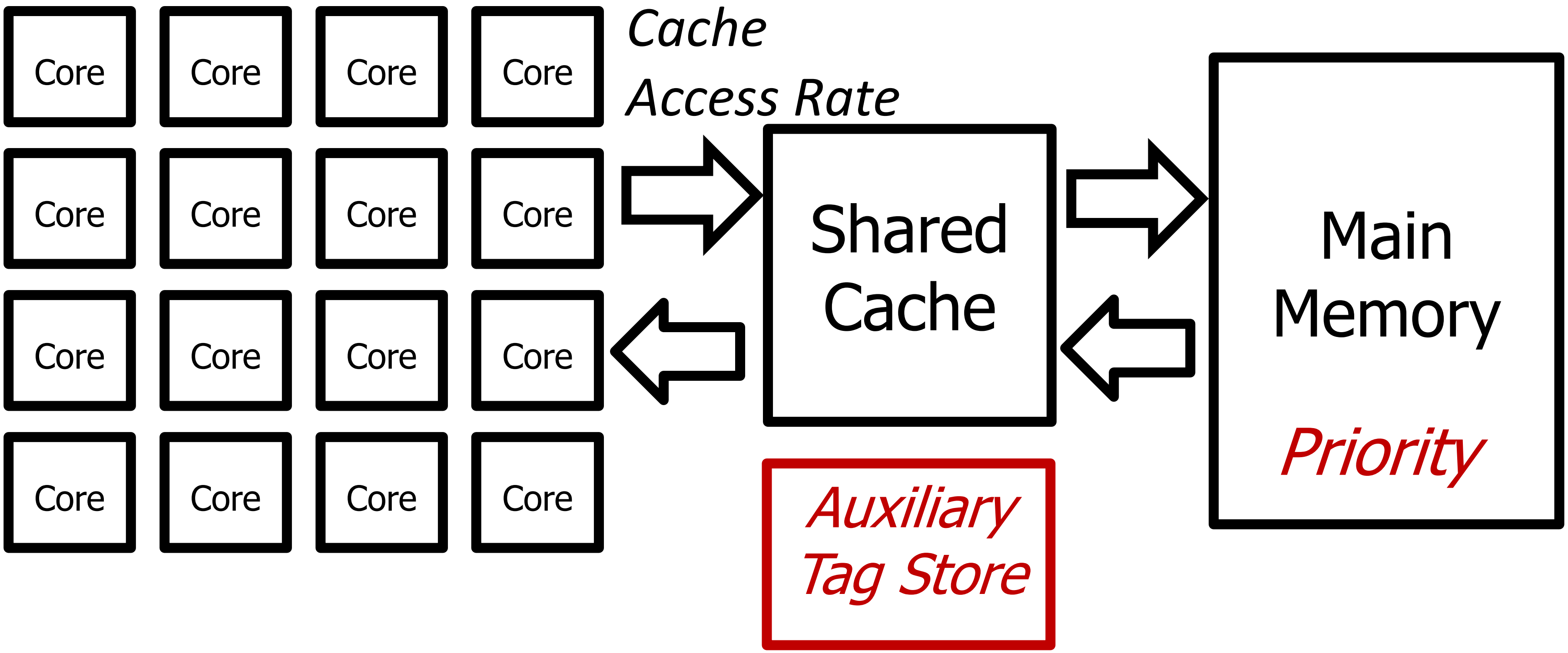
$$\text{Slowdown} = \frac{\text{Cache Access Rate}_{\text{Alone}}}{\text{Cache Access Rate}_{\text{Shared}}}$$

Real System Studies: Cache Access Rate vs. Slowdown

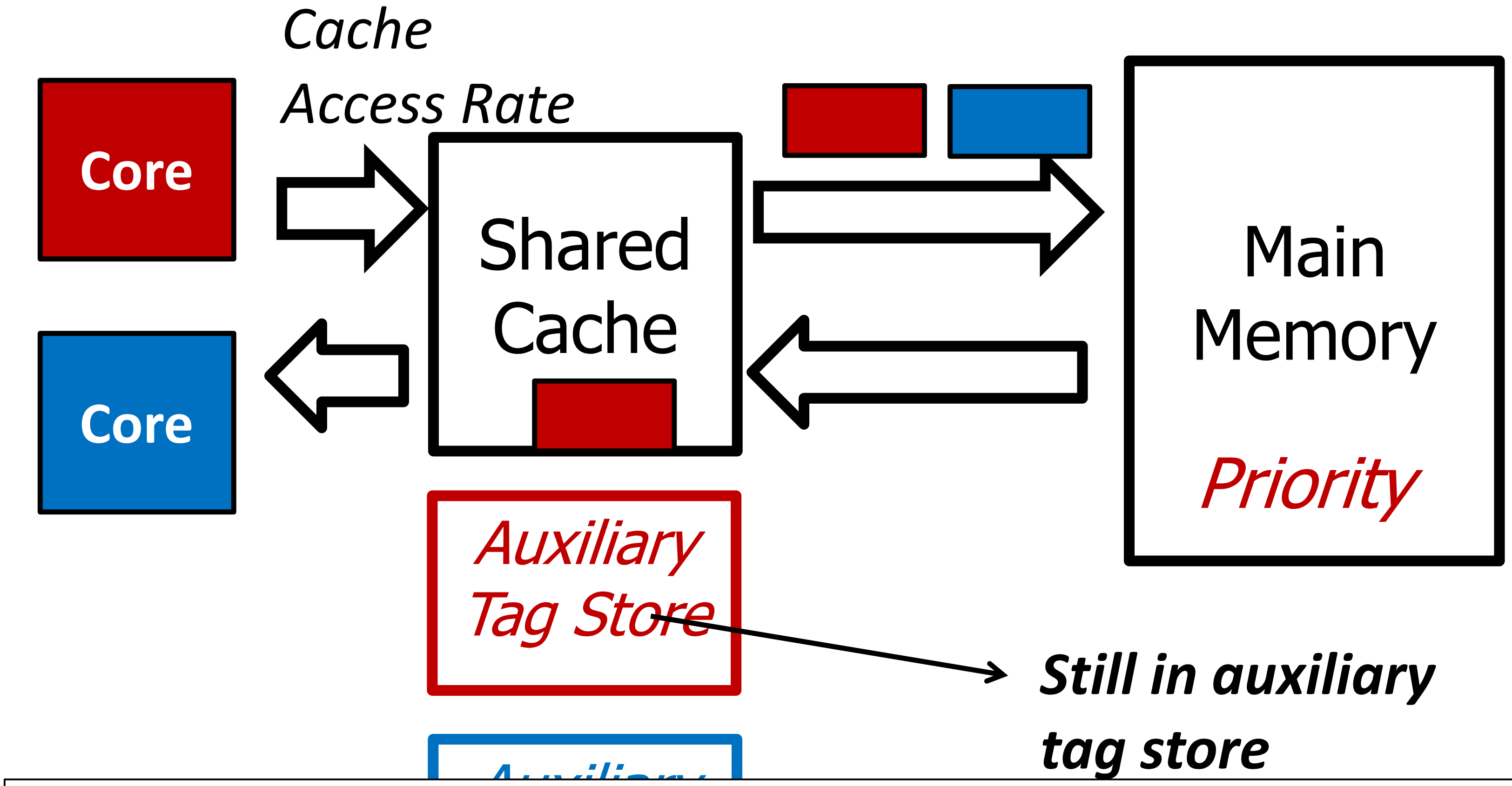


Challenge

How to estimate alone cache access rate?



Auxiliary Tag Store



Auxiliary tag store tracks such ***contention misses***

Accounting for Contention Misses

- Revisiting alone memory request service rate

$$\text{Alone Request Service Rate of an Applicatio n} = \frac{\text{\# Requests During High Priority Epochs}}{\text{\# High Priority Cycles}}$$

Cycles serving contention misses should not count as high priority cycles

Alone Cache Access Rate Estimation

$$\text{Cache Access Rate}_{\text{Alone}} \text{ of an Application} = \frac{\#\text{ Requests During High Priority Epochs}}{\#\text{ High Priority Cycles} - \boxed{\#\text{ Cache Contention Cycles}}}$$

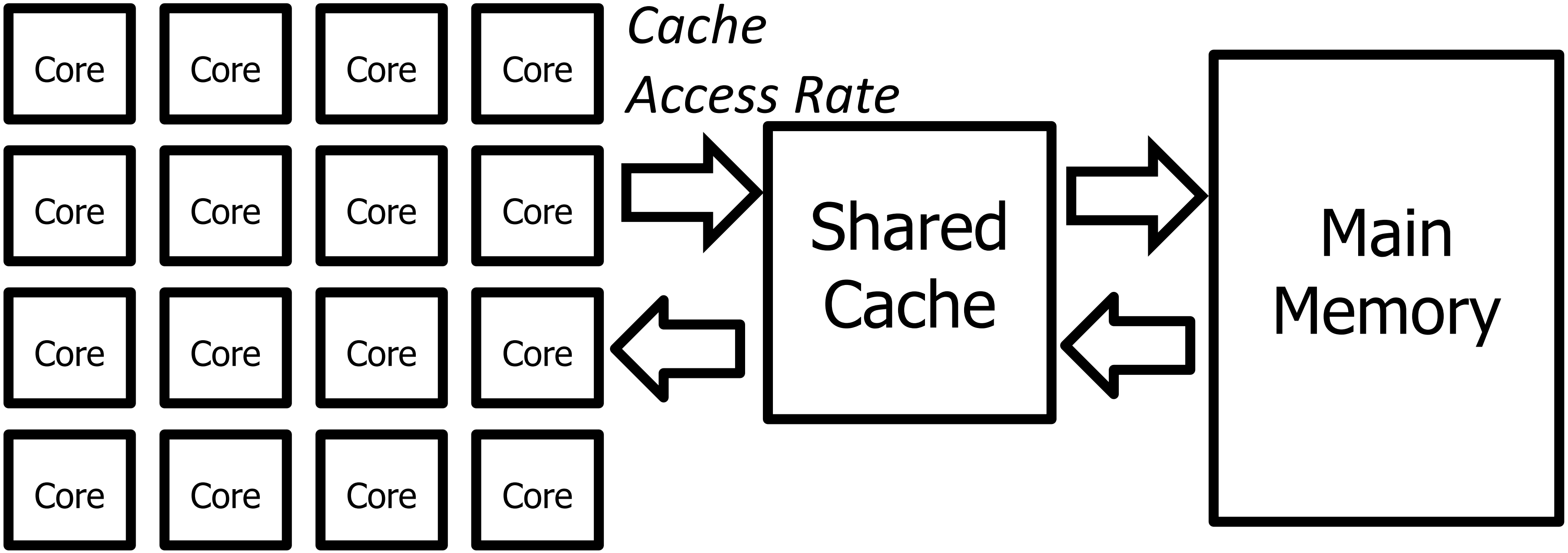
Cache Contention Cycles: Cycles spent serving contention misses

$$\text{Cache Contention Cycles} = \underbrace{\#\text{ Contention Misses}}_{\text{From auxiliary tag store when given high priority}} \text{ x } \underbrace{\text{Average Memory Service Time}}_{\text{Measured when given high priority}}$$

From auxiliary tag store when given high priority

Measured when given high priority

Application Slowdown Model (ASM)



$$\text{Slowdown} = \frac{\text{Cache Access Rate}_{\text{Alone}}}{\text{Cache Access Rate}_{\text{Shared}}}$$

Previous Work on Slowdown Estimation

- Previous work on slowdown estimation
  - **STFM** (Stall Time Fair Memory) Scheduling [Mutlu et al., MICRO '07]
  - **FST** (Fairness via Source Throttling) [Ebrahimi et al., ASPLOS '10]
  - **Per-thread Cycle Accounting** [Du Bois et al., HiPEAC '13]

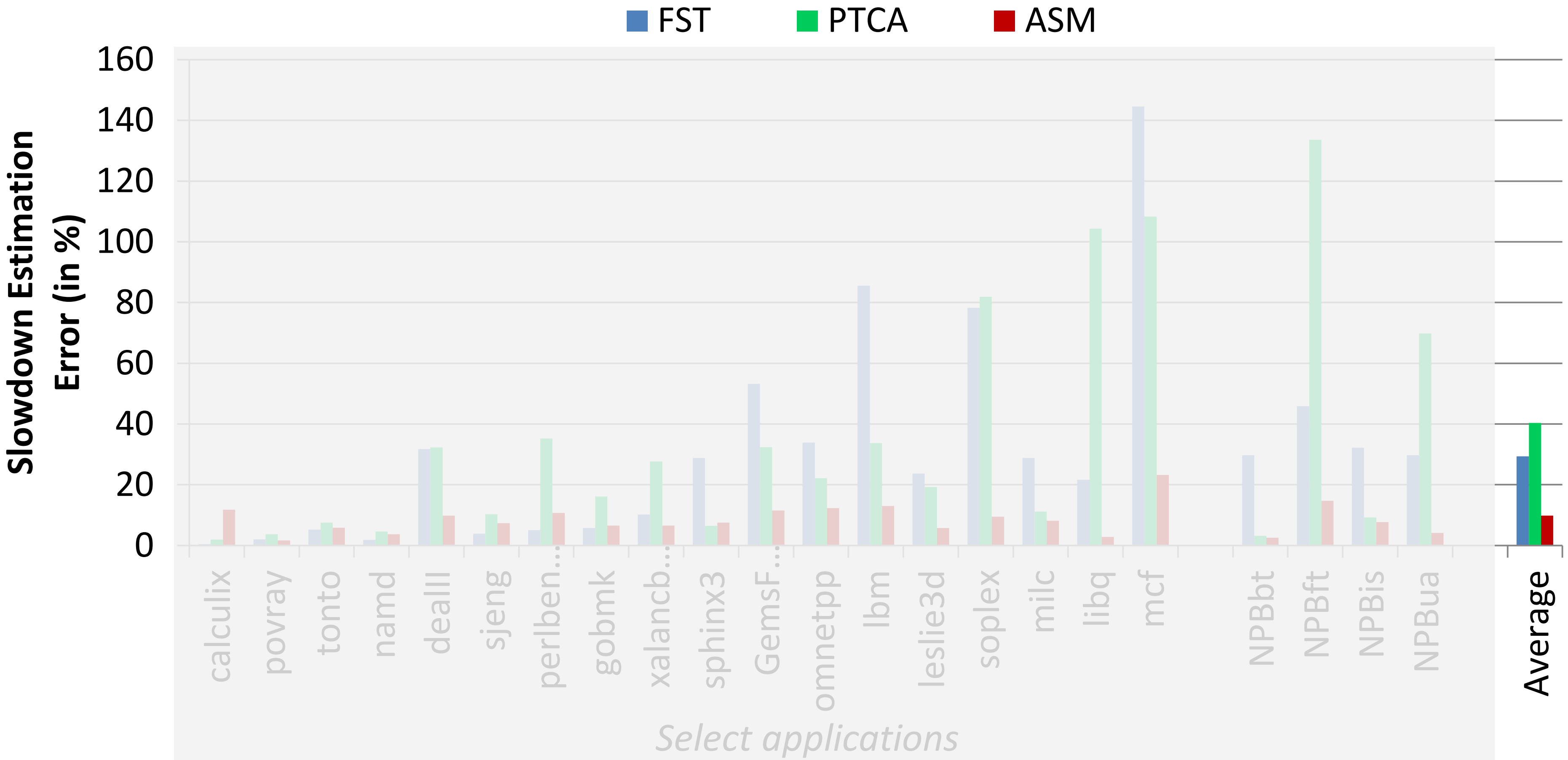
- Basic Idea:

$$\text{Slowdown} = \frac{\text{Execution Time}_{\text{Alone}}}{\text{Execution Time}_{\text{Shared}}}$$

Count interference experienced by each request → Difficult

ASM's estimates are much more coarse grained → Easier

Model Accuracy Results

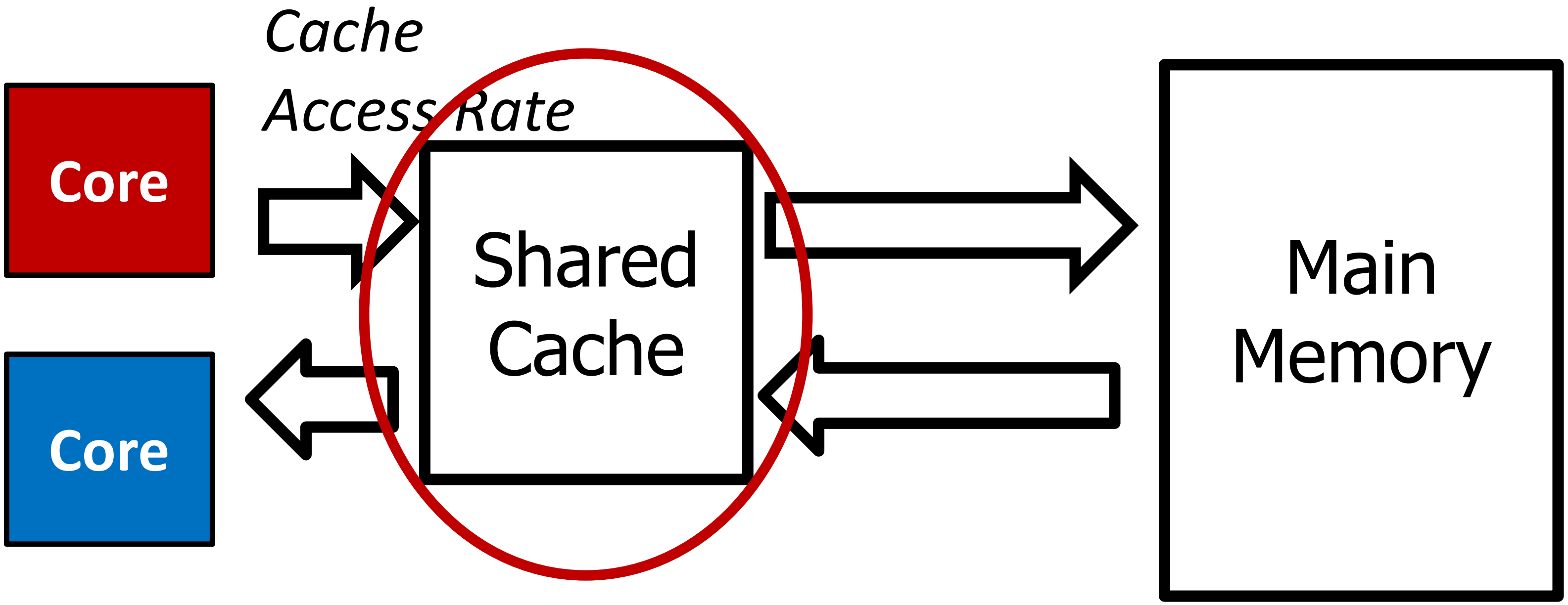


Average error of ASM's slowdown estimates: 10%

Leveraging ASM’s Slowdown Estimates

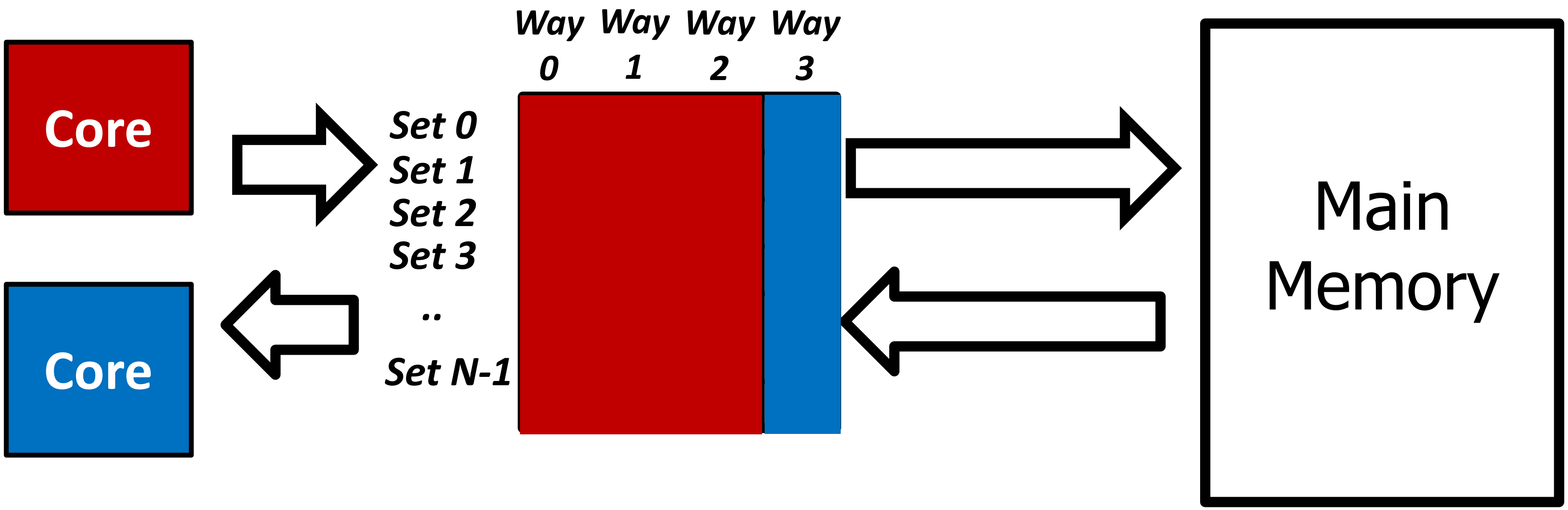
- *Slowdown-aware resource allocation for high performance and fairness*
- *Slowdown-aware resource allocation to bound application slowdowns*
- *VM migration and admission control schemes* ***[VEE ’15]***
- *Fair billing schemes in a commodity cloud*

Cache Capacity Partitioning



Goal: Partition the shared cache among applications to mitigate contention

Cache Capacity Partitioning



Previous partitioning schemes optimize for miss count

Problem: Not aware of performance and slowdowns

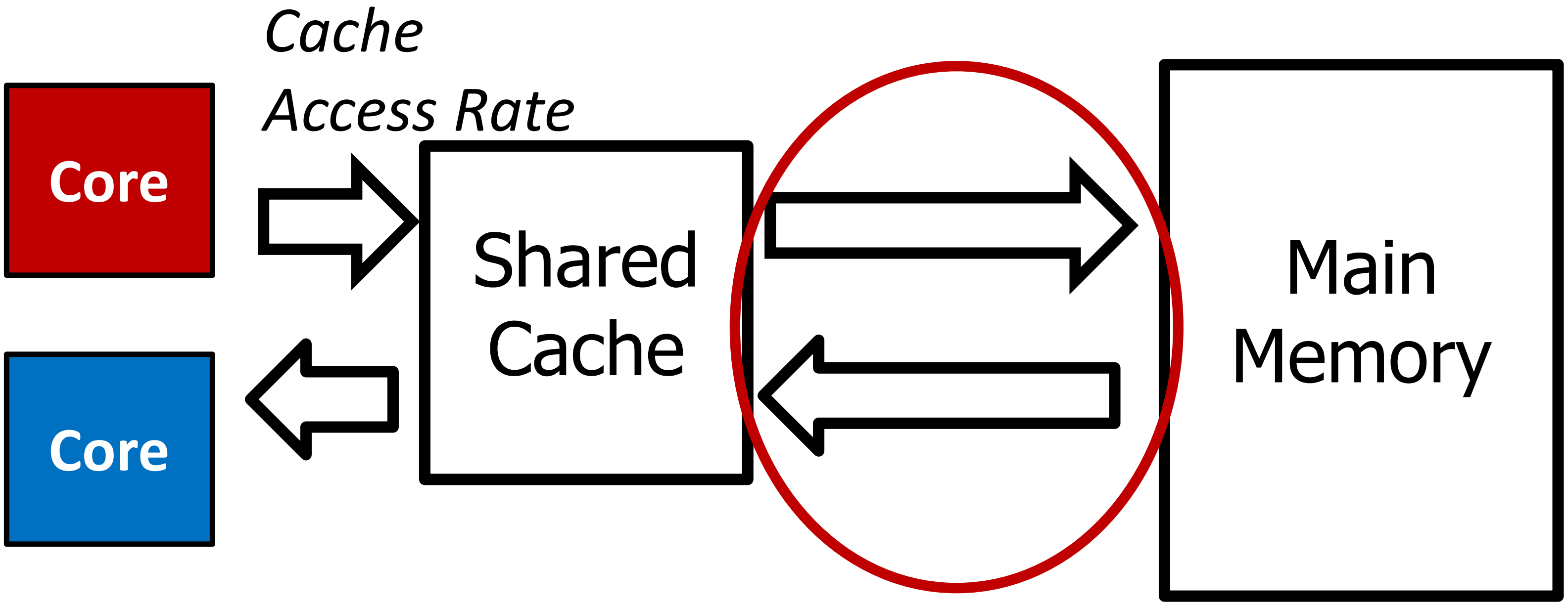
ASM-Cache: Slowdown-aware Cache Way Partitioning

- *Key Requirement: Slowdown estimates for all possible way partitions*

- *Extend ASM to estimate slowdown for all possible cache way allocations*

- *Key Idea: Allocate each way to the application whose slowdown reduces the most*

Memory Bandwidth Partitioning



Goal: Partition the main memory bandwidth among applications to mitigate contention

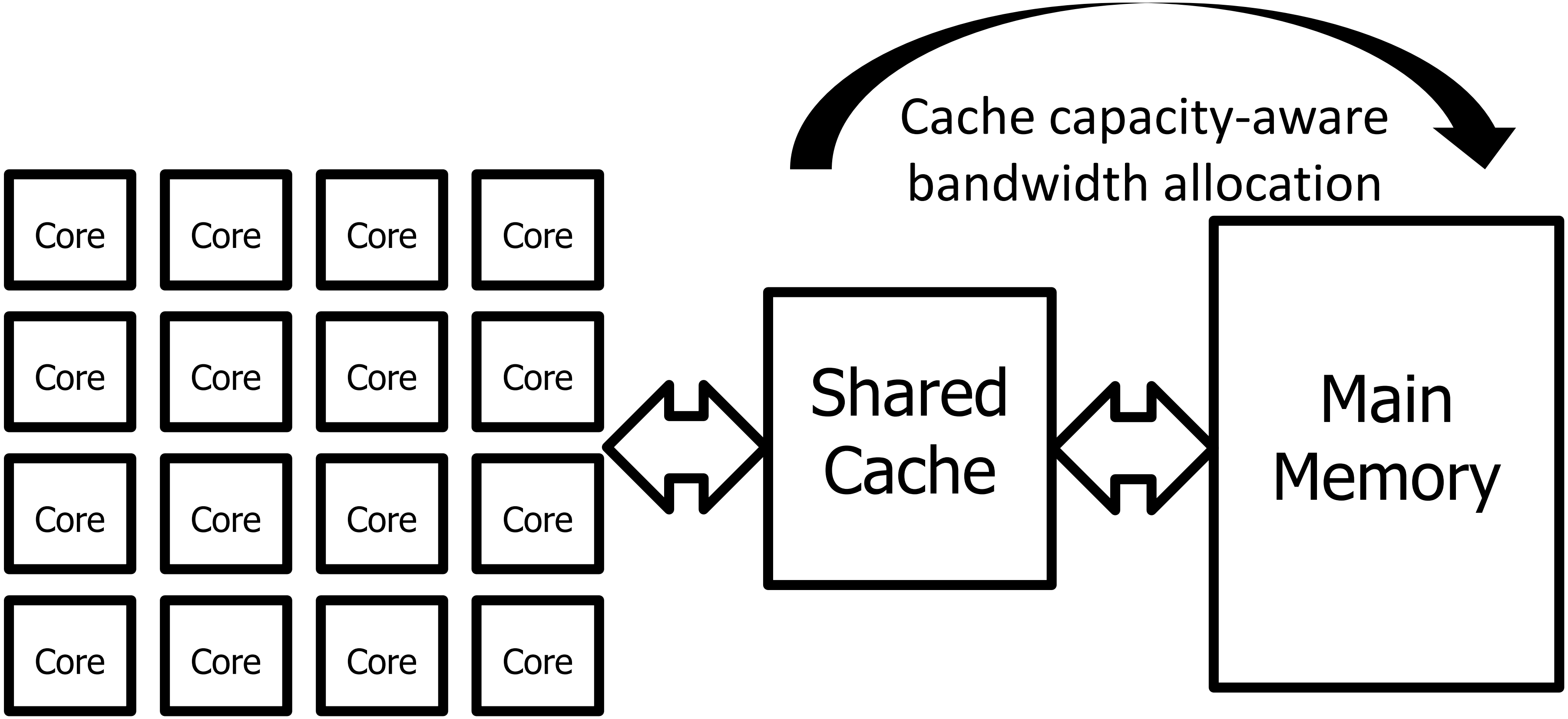
ASM-Mem: Slowdown-aware Memory Bandwidth Partitioning

- *Key Idea: Allocate high priority proportional to an application's slowdown*

$$\text{High Priority Fraction}_i = \frac{\text{Slowdown}_i}{\sum_j \text{Slowdown}_j}$$

- *Application i's requests given highest priority at the memory controller for its fraction*

Coordinated Resource Allocation Schemes

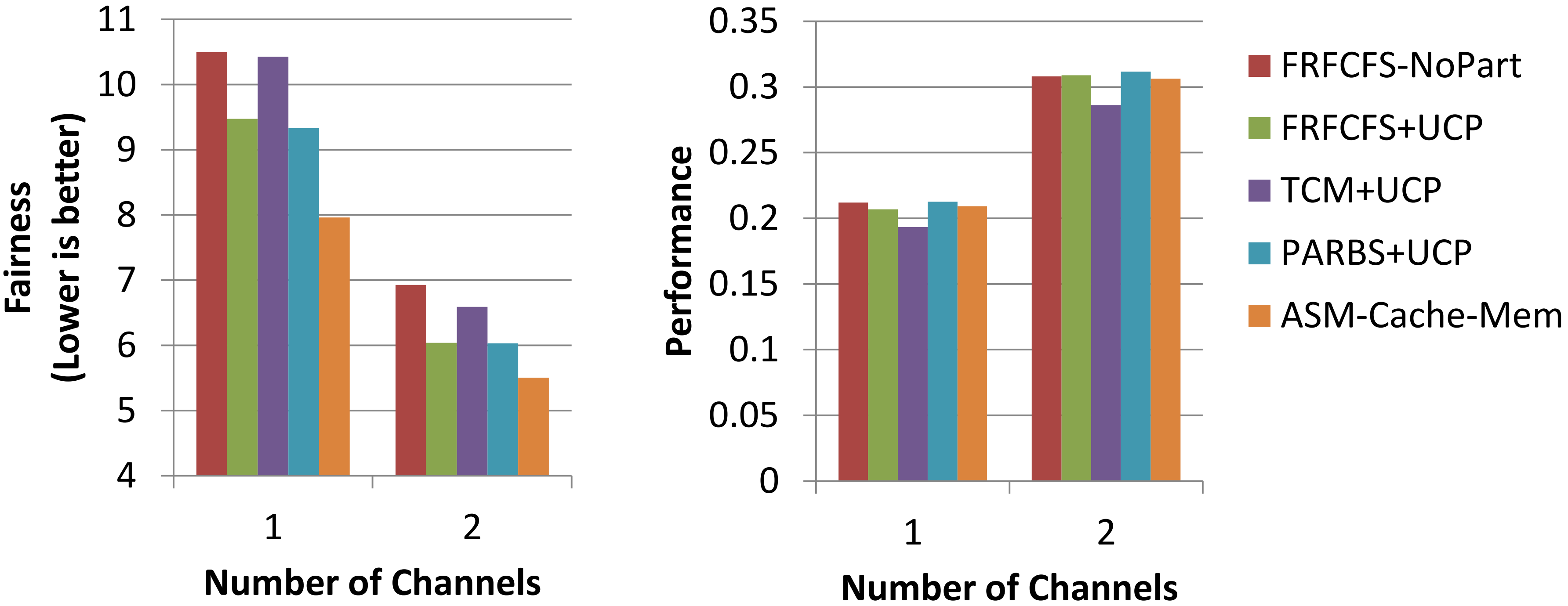


1. Employ ASM-Cache to partition cache capacity

2. Drive ASM-Mem with slowdowns from ASM-Cache

Fairness and Performance Results

16-core system
100 workloads



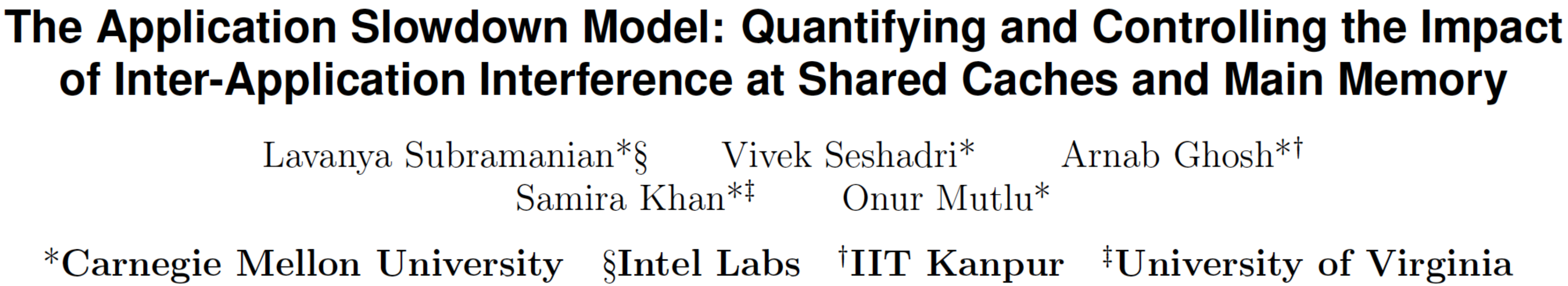
Significant fairness benefits across different channel counts

Summary

- Problem: Uncontrolled memory interference cause high and unpredictable application slowdowns
- Goal: Quantify and control slowdowns
- Key Contribution:
  - ASM: An accurate slowdown estimation model
  - Average error of ASM: 10%
- Key Ideas:
  - Shared cache access rate is a proxy for performance
  - Cache Access Rate $_{\text{Alone}}$ can be estimated by minimizing memory interference and quantifying cache interference
- Applications of Our Model
  - Slowdown-aware cache and memory management to achieve high performance, fairness and performance guarantees
- *Source Code Released in January 2016*

More on Application Slowdown Model

- Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, and Onur Mutlu,
"The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-Application Interference at Shared Caches and Main Memory"
*Proceedings of the 48th International Symposium on Microarchitecture (**MICRO**)*, Waikiki, Hawaii, USA, December 2015.
[Slides (pptx) (pdf)] [Lightning Session Slides (pptx) (pdf)] [Poster (pptx) (pdf)]
[Source Code]



The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-Application Interference at Shared Caches and Main Memory

Lavanya Subramanian*§ Vivek Seshadri* Arnab Ghosh*†
Samira Khan*‡ Onur Mutlu*

*Carnegie Mellon University §Intel Labs †IIT Kanpur ‡University of Virginia