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

ASPLOS 2010

HPS Research Group
The University of Texas at Austin

Computer Architecture Laboratory
Carnegie Mellon University

Executive Summary

- Motivation: Cores in a chip-multiprocessor system share multiple hardware resources in the memory subsystem
  - Interference in the shared resources can lead to unfair slowdown for some applications
- Problem: Existing fairness mechanisms focus on a single resource
  - Multiple independently implemented mechanisms can make contradictory decisions, leading to low fairness and loss of performance
- Goal: provides fairness in the entire shared memory system without degrading performance
- Key Contributions: Fairness via Source Throttling(FST) provides two major mechanisms
  - 1) Runtime fairness evaluation
  - 2) Dynamic request throttling
- Major Results: improve performance by 25.6%/14.5% and reduce unfairness by 44%/36%

Outline

- Background, Problem & Goal
- Novelty
- Key Approach & Ideas
- Mechanism
- Methodology & Evaluation
- Conclusion

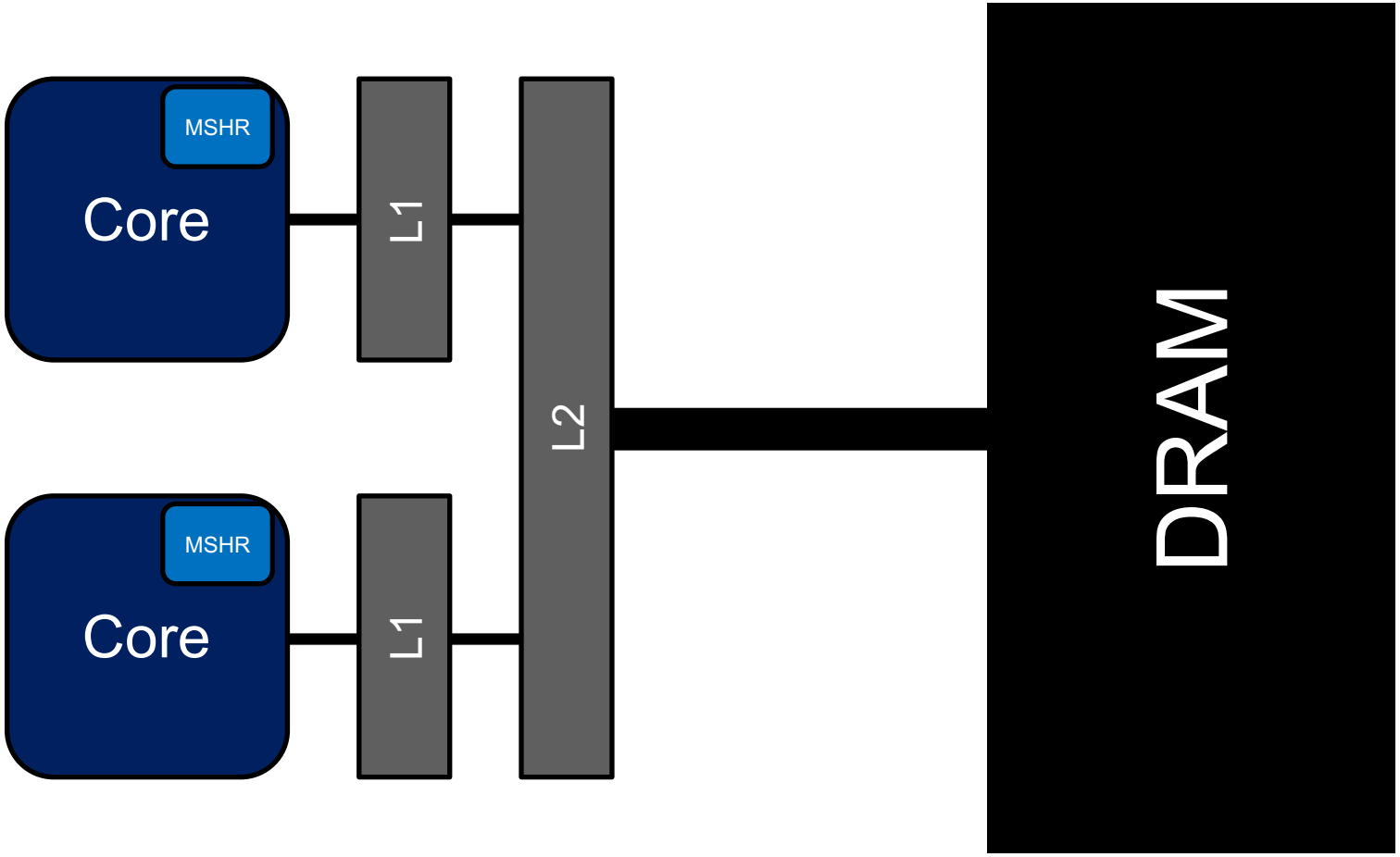
- Takeaways
- Strengths and Weaknesses
- Thoughts & Ideas
- Open Discussion

Outline

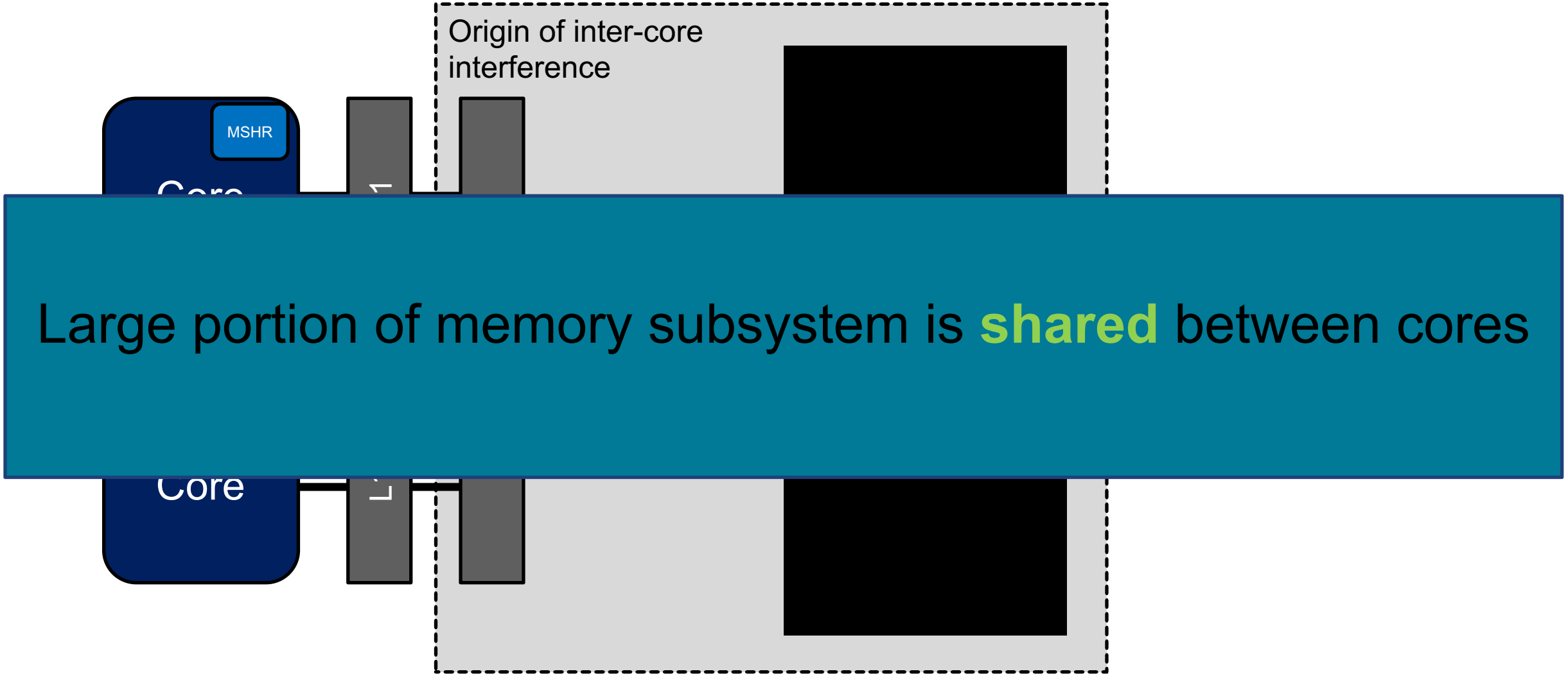
- Background, Problem & Goal
- Novelty
- Key Approach & Ideas
- Mechanism
- Methodology & Evaluation
- Conclusion

- Takeaways
- Strengths and Weaknesses
- Thoughts & Ideas
- Open Discussion

System Layout

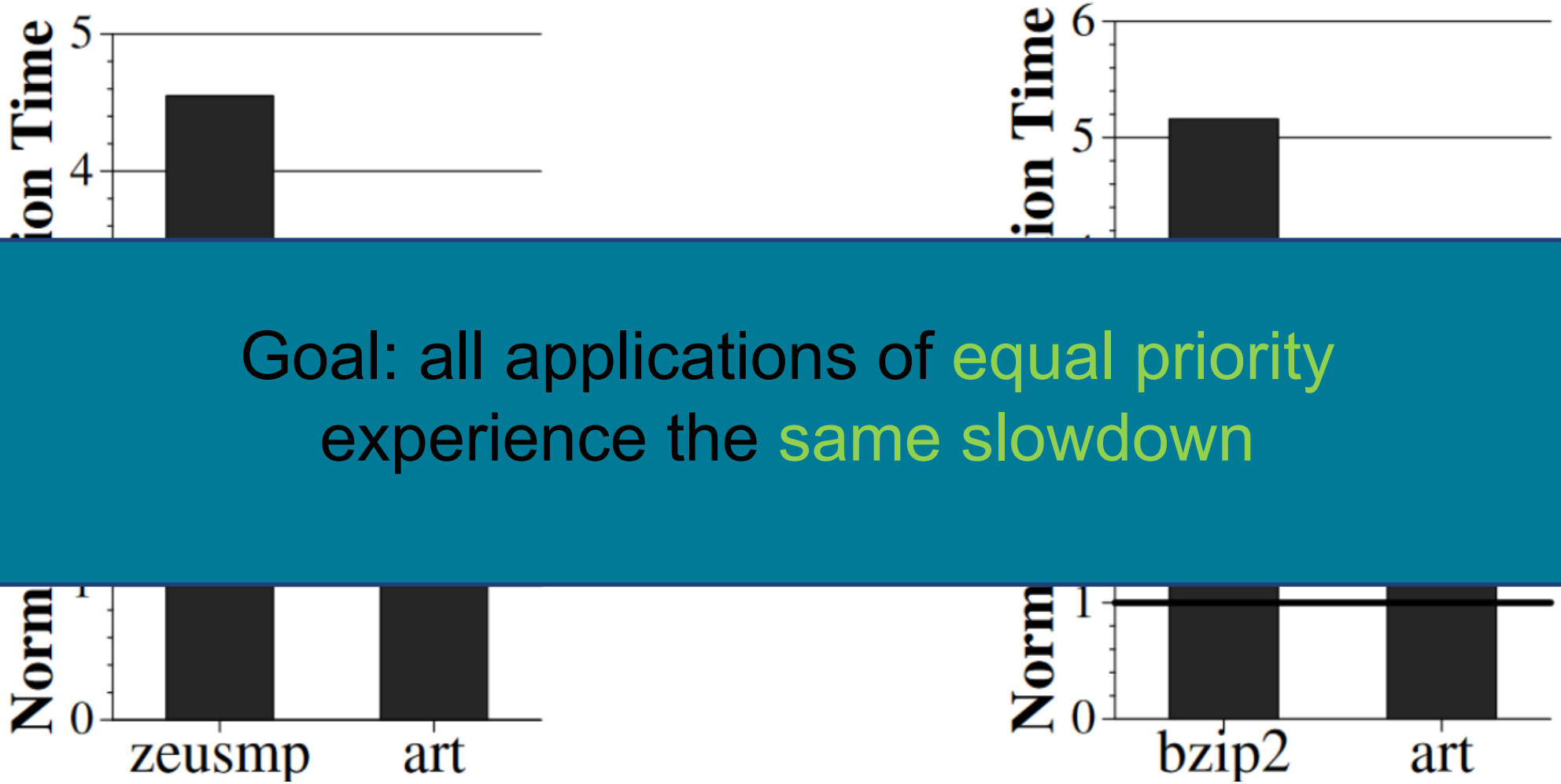


System Layout



Large portion of memory subsystem is **shared** between cores

Interference and delays lead to slowdown

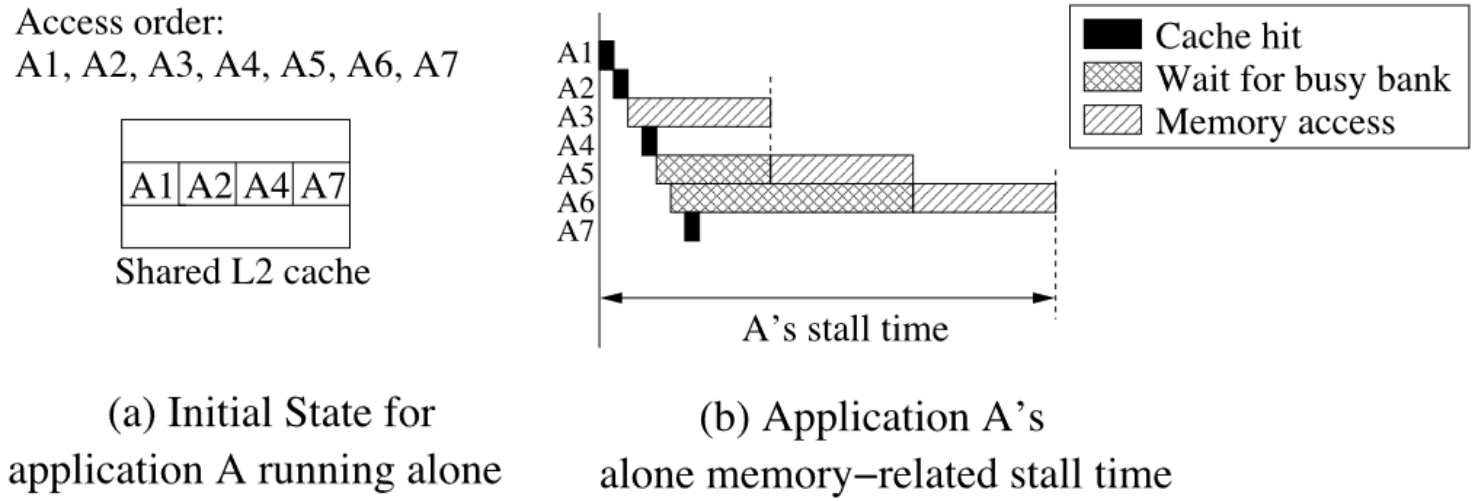


Goal: all applications of equal priority experience the same slowdown

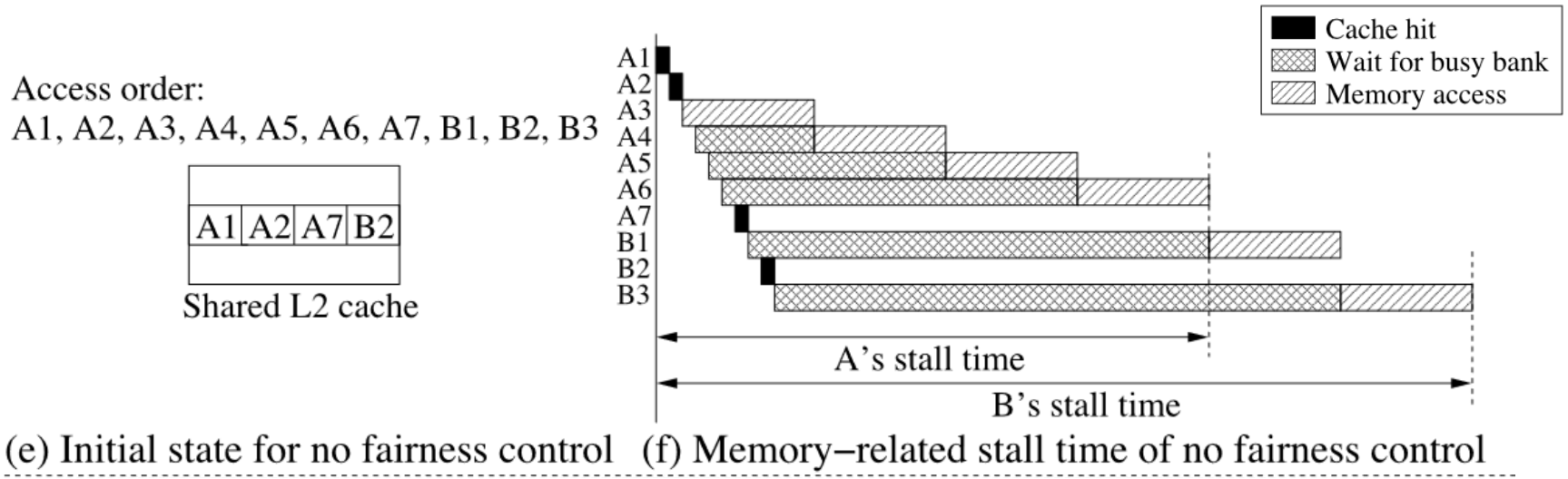
Previous Approach

- Previous research focused on individual resources

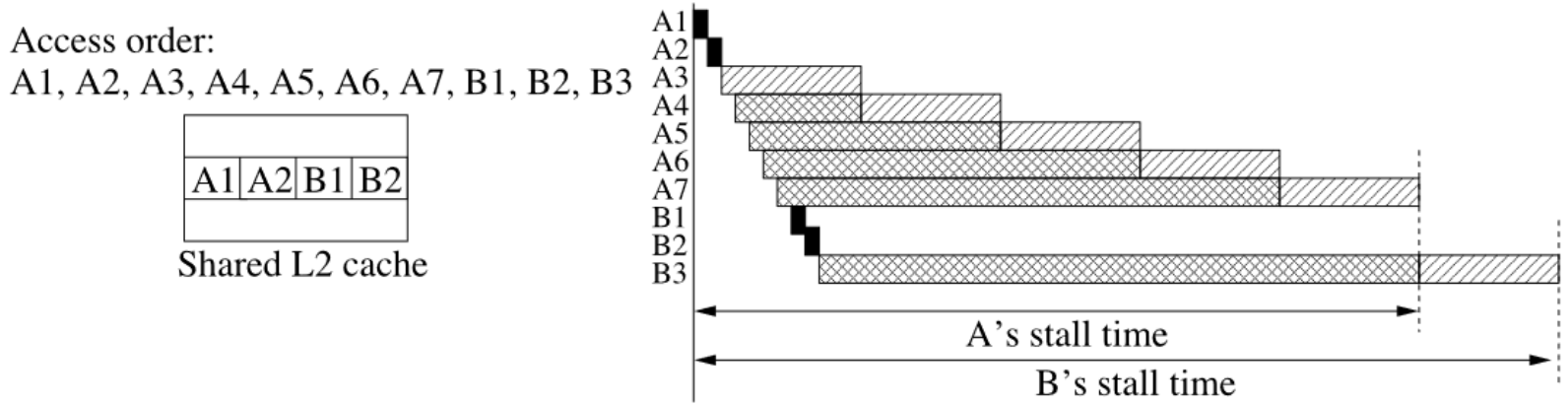
Previous Approach



Previous Approach



(e) Initial state for no fairness control (f) Memory–related stall time of no fairness control



(g) Initial state for fair cache (h) Memory–related time of fair cache

Previous Approach

- Previous research focused on **individual** resources
- It is challenging to properly **coordinate multiple** fairness mechanisms
  - Partitioning one resource may change demands on another shared resource

Outline

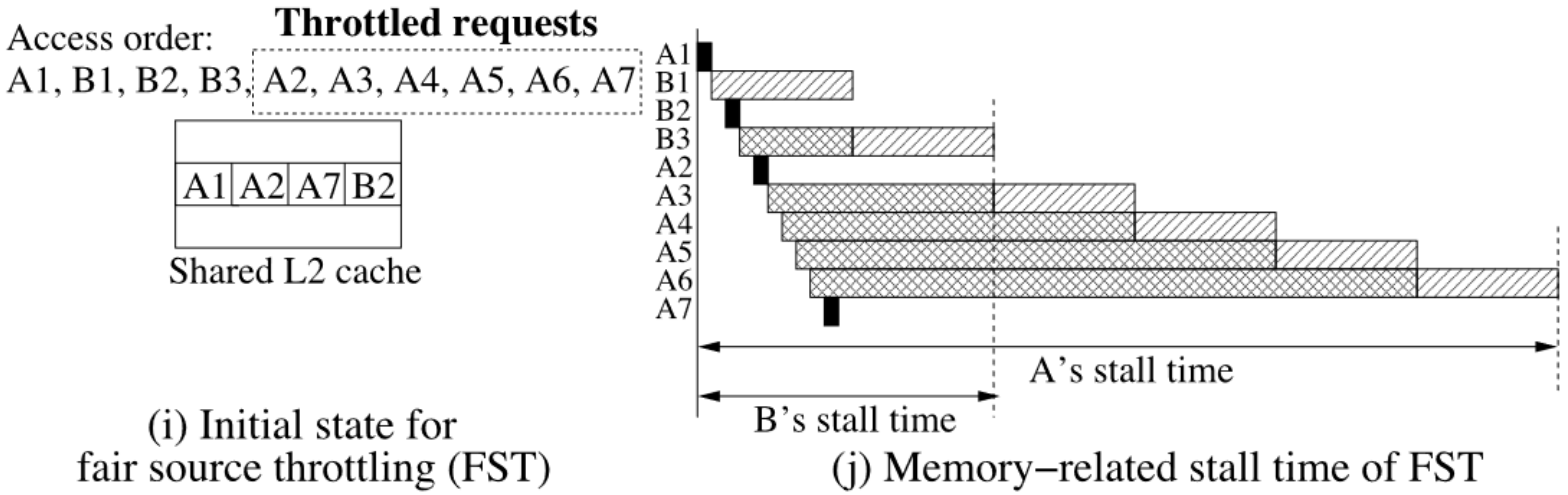
- Background, Problem & Goal
- Novelty
- Key Approach & Ideas
- Mechanism
- Methodology & Evaluation
- Conclusion

- Takeaways
- Strengths and Weaknesses
- Thoughts & Ideas
- Open Discussion

System-Wide Fairness

- Tackle unfairness in the **entire shared memory system**
  - Eliminate the need for multiple fairness mechanisms

- Control fairness by **orchestrating memory requests**
  - Rate of memory request injections
  - Number of memory request injections



(i) Initial state for fair source throttling (FST)

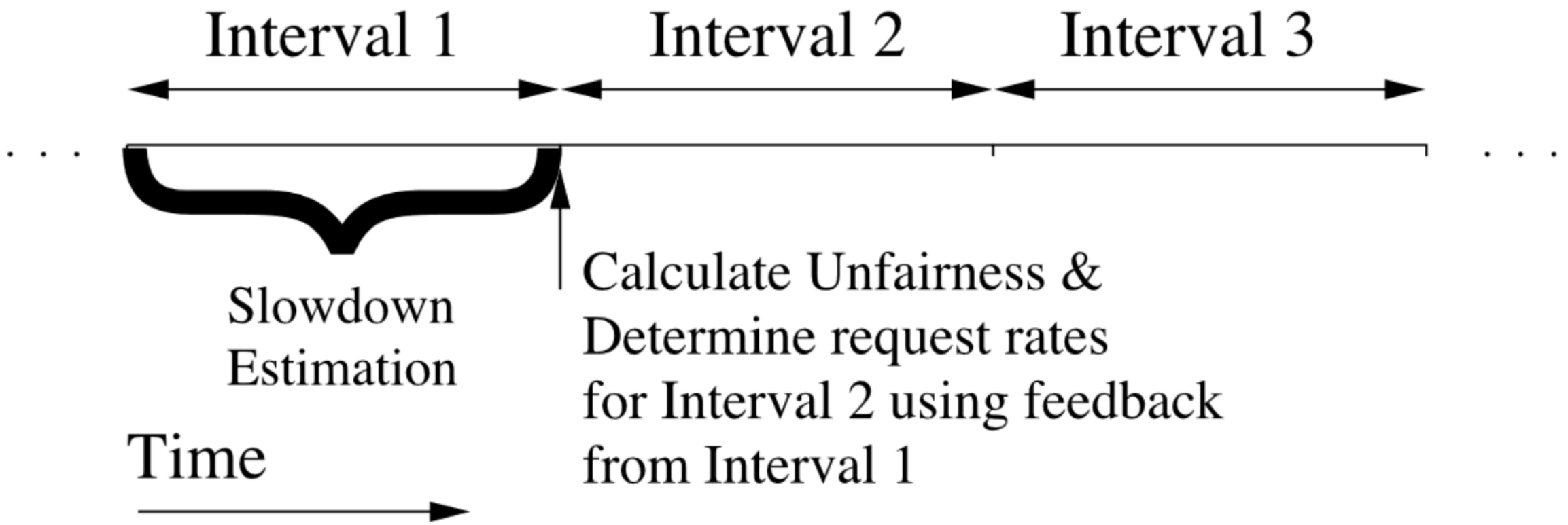
(j) Memory–related stall time of FST

Outline

- Background, Problem & Goal
- Novelty
- Key Approach & Ideas
- Mechanism
- Methodology & Evaluation
- Conclusion

- Takeaways
- Strengths and Weaknesses
- Thoughts & Ideas
- Open Discussion

Interval based Estimation and Throttling



Outline

- Background, Problem & Goal
- Novelty
- Key Approach & Ideas
- Mechanism
- Methodology & Evaluation
- Conclusion

- Takeaways
- Strengths and Weaknesses
- Thoughts & Ideas
- Open Discussion

Runtime Fairness Evaluation

- Goal: **dynamically estimate** system unfairness
- Slowdown: T_{shared}/T_{alone}
  - T_{shared}: number of cycles to execute **simultaneously** with other applications
  - T_{alone}: number of cycles to execute **alone**
- Estimating T_{alone} while running multiple applications
  - T_{excess}: number of excess execution cycles induced by **inter-core interference**
  - $T_{alone} = T_{shared} - T_{excess}$

$$IS_i = \frac{T_i^{shared}}{T_i^{alone}}, \quad Unfairness = \frac{MAX\{IS_0, IS_1, ..., IS_{N-1}\}}{MIN\{IS_0, IS_1, ..., IS_{N-1}\}}$$

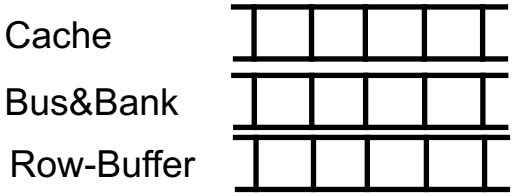
Tracking Inter-Core Interference

- Three sources of inter-core interference:
  - Shared cache
  - DRAM bus and bank
  - DRAM row-buffer

- InterferencePerCore bit-vector

  - Indicate whether a core is **delayed** due to inter-core interference

- Bit-vector for each source
  - Update main copy by taking union of the source bit-vectors



Cache Interference

- Goal: Estimating inter-core interference on the cache by tracking **cache misses** caused by **another core**

- Pollution filter for each core
  - Bit-vector is indexed by the lower order bits of the accessed cache line address
  - A set entry in the bit-vector indicates that a cache line belonging to this core was **evicted** by **another core**

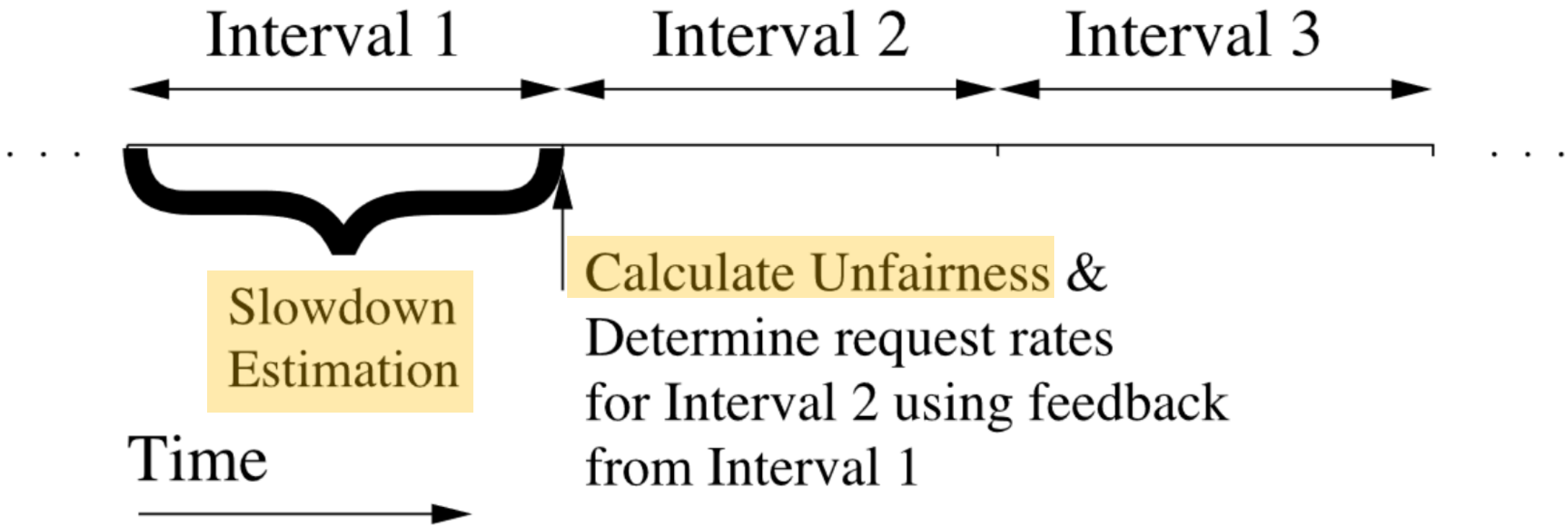
- Three steps in case of cache miss:
  - 1) on cache miss access pollution filter with the missing address and check wether **bit is set**
  - 2) set the bit in the **InterferencePerCore** vector and reset the bit in the **pollution filter**
  - 3) when the interfered-with memory **request is serviced** reset the InterferencePerCore bit

DRAM Bus & Bank Interference

- Goal: Estimate inter-core interference caused by an **inability** to access DRAM due to another core using the bus or requesting service from the bank
- This situation is easily detectable
  - If detected the corresponding InterferencePerCore bit is set
- The InterferencePerCore bit is reset when **no request** from this core is being **prevented** access to DRAM by another cores requests

DRAM Row-Buffer Interference

- Goal: Estimate interference caused by the **conversion** of row-buffer hits to a **miss/conflict** due to another cores memory request
- Shadow Row-Buffer Address Register for each core and for each bank
  - Whenever memory request accesses some row X, the SRAR is updated to X
- Three Steps in case of Row-Buffer miss:
  - 1) on row-buffer miss consult SRAR
  - 2) if the **SRAR** bit is set, interference is present, hence InterferencePerCore bit is set
  - 3) once the memory **request is serviced** the InterferencePerCore bit is reset

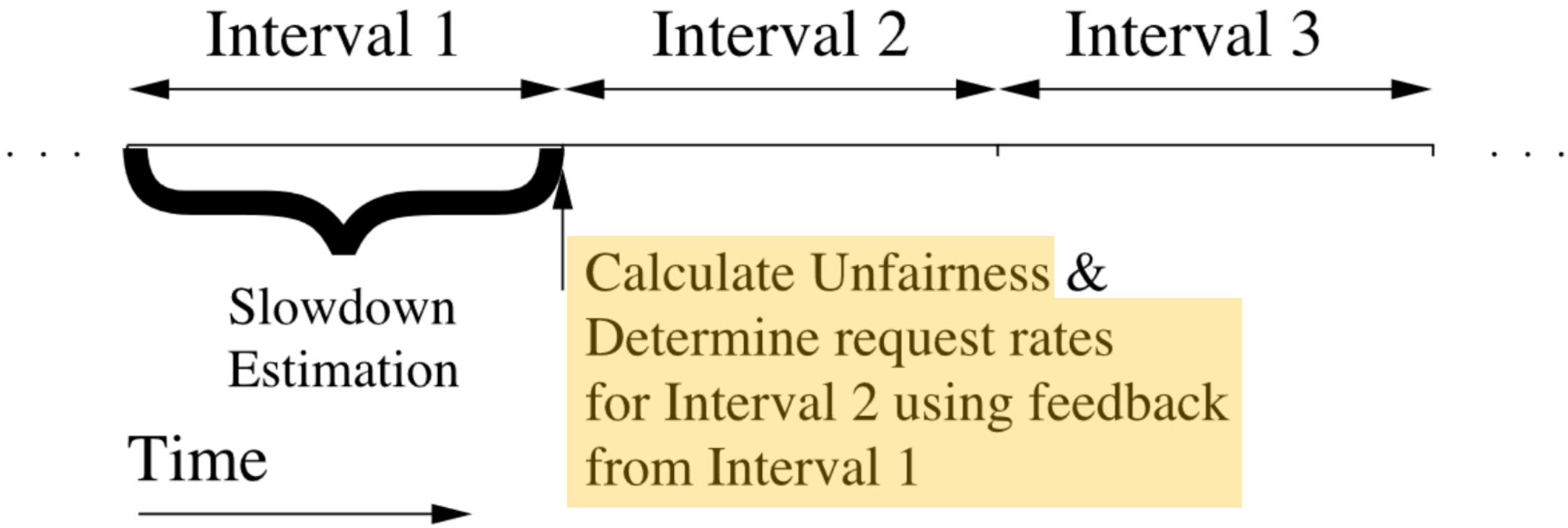

Interval 1
Interval 2
Interval 3
Slowdown Estimation
Calculate Unfairness &
Determine request rates for Interval 2 using feedback from Interval 1
Time

Estimation of T_{excess} for Core *i*

- Every cycle:
  - Check whether core *i* experiences interference
  - Increment T_{excess} by 1

- $T_{alone} = T_{shared} - T_{excess}$

$$IS_i = \frac{T_i^{shared}}{T_i^{alone}}, \quad Unfairness = \frac{MAX\{IS_0, IS_1, ..., IS_{N-1}\}}{MIN\{IS_0, IS_1, ..., IS_{N-1}\}}$$


Interval 1
Interval 2
Interval 3
Slowdown
Estimation
Time
Calculate Unfairness &
Determine request rates
for Interval 2 using feedback
from Interval 1

Dynamic Request Throttling

- Check whether the estimated unfairness is **bigger** than a certain unfairness threshold
  - Throttle **down** application with the **smallest** slowdown
  - Throttle **up** application with the **largest** slowdown
- After fairness was achieved for a certain number of successive intervals:
  - Throttle up **all** applications

Throttling Mechanisms

- 1) Adjust MSHR quota
  - **MSHR quota** determines the **max. number of outstanding misses** for each core
  - Reduce the pressure by decreasing the number of concurrent request contending for service
- 2) Adjust the rate of issuing requests to the shared cache
  - Reduce number of memory **requests per unit time**
  - This allows requests from other applications to be prioritized

System Software Support

- Different Fairness Objectives:
  - The goal to be achieved by FST can be **configured** by system software (trigger condition)
- Thread Weights:
  - **Adjust priority** of different applications by applying weights
- Thread Migration and Context Switches:
  - On context switch or thread migration the corresponding interference state is cleared
  - On restart of execution, the thread starts with max. throttle and then FST dynamically adapts

Scalability to more Cores

- Each core maintains a set of N-1 counters, with N being the number of cores, which keep track of the inter-core interference caused by each other core
- This can be used to **identify** which core experiences the most slowdown *(App$_{slow}$)* and who of the other cores is the main contributor (*App$_{interfering}$*)
- Once identified, the **main contributor** will be throttled down and *App$_{slow}$* will be throttled up
- Cores other than the *App$_{slow}$* and *App$_{interfering}$* are throttled up every *threshold* intervals to optimize performance

Preventing Bank Service Denial due to FR-FCFS

- FR-FCFS has the potential to **starve** application with no row-buffer locality
  - Even if the interfering application gets throttled down the problem can still exist
  - This denial of service can happen continuously
- Stop prioritizing row-buffer hits

Outline

- Background, Problem & Goal
- Novelty
- Key Approach & Ideas
- Mechanism
- Methodology & Evaluation
- Conclusion

- Takeaways
- Strengths and Weaknesses
- Thoughts & Ideas
- Open Discussion

System Specification

- In-house cycle-accurate x86 CMP simulator
- Faithfully model all port contention, queuing effects, bank conflicts, and other major DDR3 DRAM system constraints

Execution Core	6.6 GHz out of order processor, 15 stages, Decode/retire up to 4 instructions Issue/execute up to 8 micro instructions 256-entry reorder buffer
Front End	Fetch up to 2 branches; 4K-entry BTB 64K-entry Hybrid branch predictor
On-chip Caches	L1 I-cache: 32KB, 4-way, 2-cycle, 64B line L1 D-cache: 32KB, 4-way, 2-cycle, 64B line Shared unified L2: 1MB (2MB for 4-core), 8-way (16-way for 4-core), 16-bank, 15-cycle (20-cycle for 4-core), 1 port, 64B line size
DRAM Controller	On-chip, FR-FCFS scheduling policy [27] 128-entry MSHR and memory request buffer
DRAM and Bus	667MHz bus cycle, DDR3 1333MHz [20] 8B-wide data bus Latency: 15-15-15ns (^{t}RP-^{t}RCD-CL) 8 DRAM banks, 16KB row buffer per bank Round-trip L2 miss latency: Row-buffer hit: 36ns, conflict: 66ns

Workloads

- 18 two-application workloads from the SPEC CPU 2000/2006 benchmark
  - Two-application workloads were chosen such that **at least one** of them is highly memory-intensive

- 10 four-application workloads from the SPEC CPU 2000/2006 benchmark
  - Four-applications workloads were chosen such that **at least one** of them has high intensity and **one** has at least medium or high intensity

Metrics

- Weighted Speedup(Wspeedup):
  - IPC^{alone} is the IPC(instructions per cycle) measured when running **alone**
  - IPC^{shared} is measured while running in **tandem** with other applications

- Harmonic mean of Speedups(Hspeedup):
  - Balanced measure between fairness and system throughput

$$Wspeedup = \sum_{i=0}^{N-1} \frac{IPC_i^{shared}}{IPC_i^{alone}}$$

$$Hspeedup = \frac{N}{\sum_{i=0}^{N-1} \frac{IPC_i^{alone}}{IPC_i^{shared}}}$$

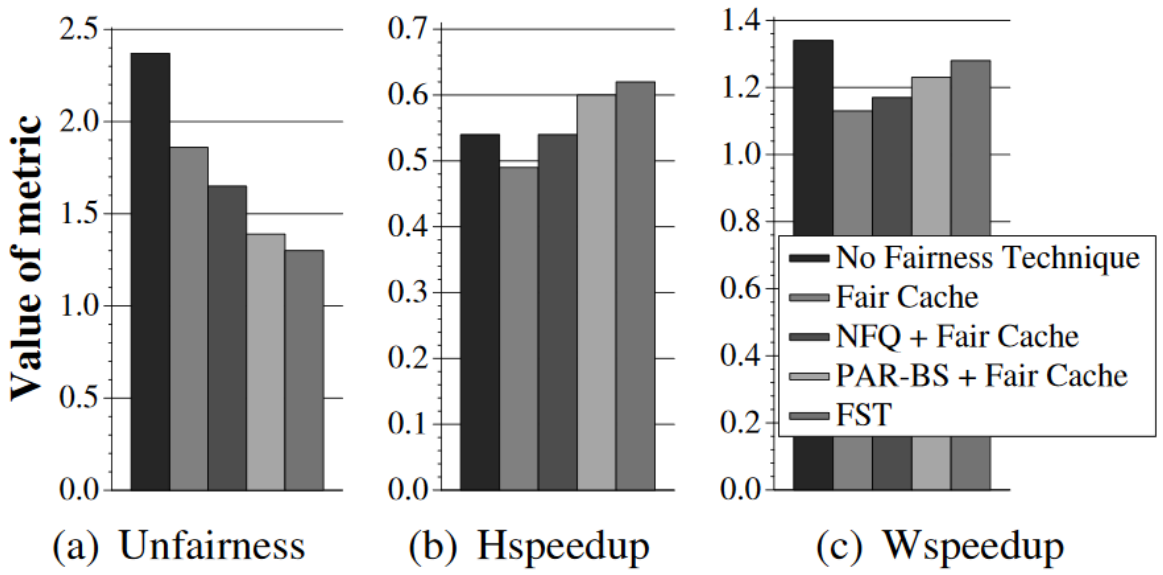
Methodology

- NoFairness:
  - Employs **no fairness techniques** in the shared memory subsystem
  - Uses LRU cache replacement and FR-FCFS
- FairCache:
  - Uses Virtual private caches technique for **fair capacity management**
- NFQ+FairCache:
  - Uses a **network fair queuing**(NFQ) fair memory scheduler combined with FairCache
- PAR-BS+FairCache:
  - Use **parallelism-aware batch scheduling** fair memory scheduler and FairCache

2-Core System Results

- All Fairness techniques degrade Wspeedup to some extent
  - Unsophisticated fairness mechanisms can have a **negative** effect on system performance
  - FST provides a significantly better **balance** between system fairness and performance

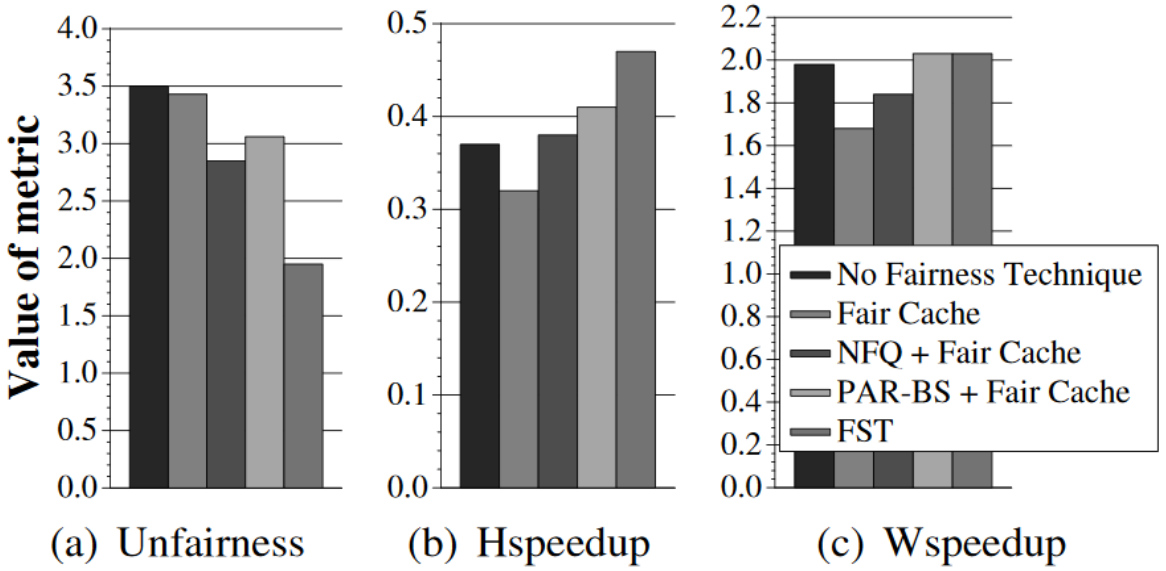
Average performance on the 2 core system



4-Core System Results

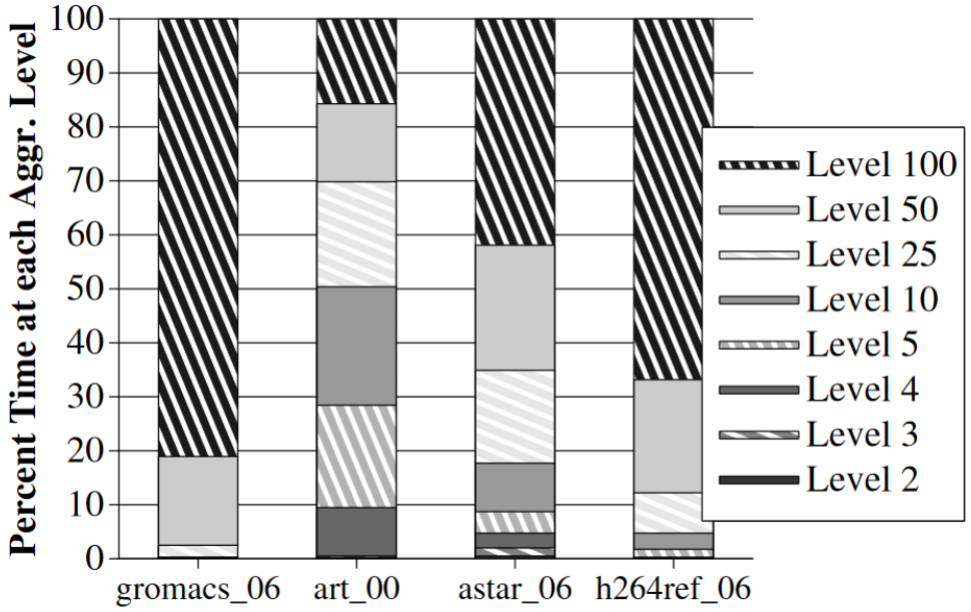
- Previous fairness mechanisms fail to improve system fairness significantly
  - Prioritize nonintensive applications regardless of whether or not those experience slowdown
- FST is the best technique for system fairness and Hspeedup, while not falling behind in Wspeedup

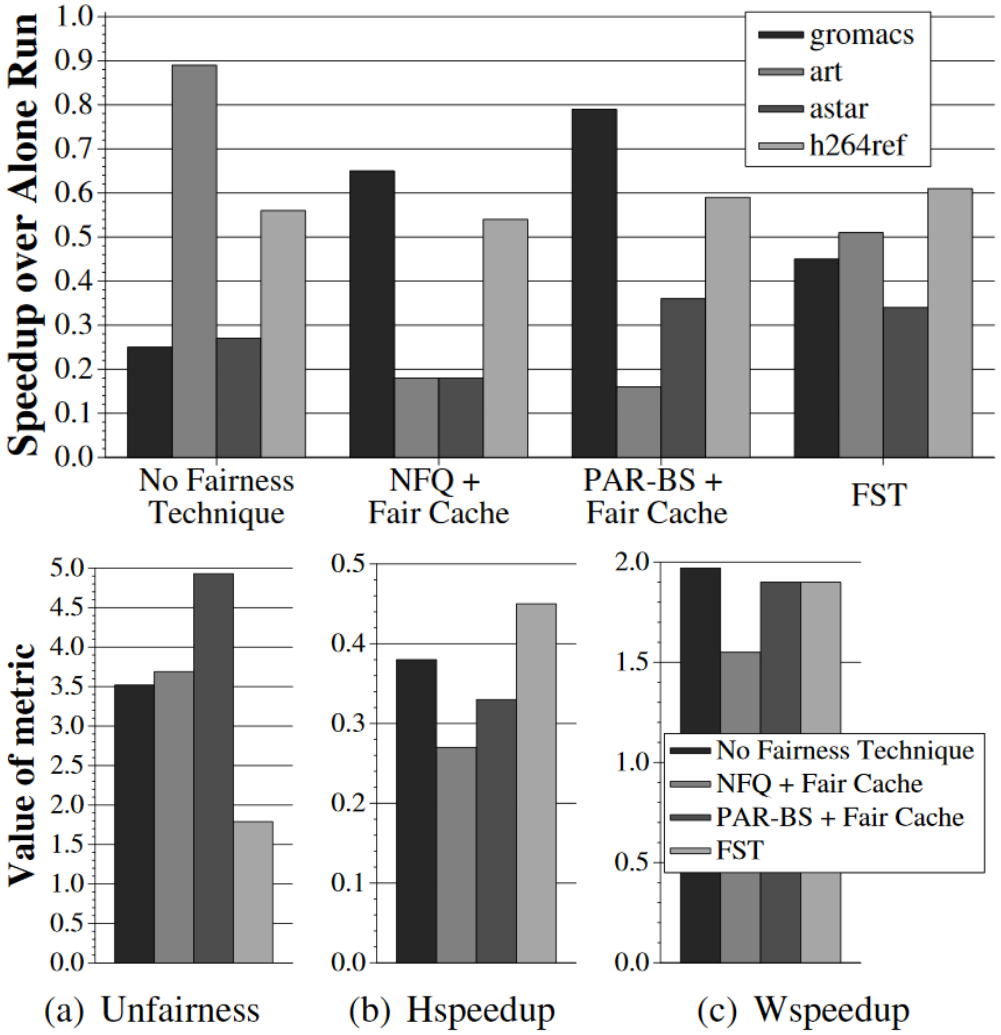
Average performance on the 4 core system



Case Study

- Art and Astar are memory intensive:
  - These are slowed down **too much** by NFQ+FairCache and PAR-BS+FairCache, causing high unfairness
  - Inability to detect when slowdown is caused





Hardware Cost

	Cost for N cores	Cost for N = 4
$ExcessCycles$ counters	N × N × 16 bits/counter	256 bits
Interference pollution filter per core	2048 entries × N × (1 pollution bit + (log_2 N) bit processor id)/entry	24,576 bits
$InterferingCoreId$ per MSHR entry	32 entries/core × N × 2 interference sources × (log_2 N) bits/entry	512 bits
$InterferencePerCore$ bit-vector	3 interference sources × N × N × 1 bit	48 bits
Shadow row-buffer address register	N × # of DRAM banks (B) × 32 bits/address	1024 bits
$Successive\ Fairness\ Achieved\ Intervals$ counter $Intervals\ To\ Wait\ To\ Throttle\ Up$ counter per core $Inst\ Count\ Each\ Interval$ per core	(2 × N + 1) × 16 bits/counter	144 bits
Core id per tag store entry in K MB L2 cache	16384 blocks/Megabyte × K × (log_2 N) bit/block	65,536 bits
Total hardware cost for N-core system	Sum of the above	92092 = **11.24 KB**
Percentage area overhead (as fraction of the baseline K MB L2 cache)	Sum (KB) × 100 / (K × 1024)	11.24KB/2048KB = **0.55%**

Hardware cost of FST on a 4-core CMP system

Outline

- Background, Problem & Goal
- Novelty
- Key Approach & Ideas
- Mechanism
- Methodology & Evaluation
- Conclusion

- Takeaways
- Strengths and Weaknesses
- Thoughts & Ideas
- Open Discussion

Executive Summary & Conclusion

- Motivation: Cores in a chip-multiprocessor system share multiple hardware resources in the memory subsystem
  - Interference in the shared resources can lead to unfair slowdown for some applications
- Problem: Existing fairness mechanisms focus on a single resource
  - Multiple independently implemented mechanisms can make contradictory decisions, leading to low fairness and loss of performance
- Goal: provides fairness in the entire shared memory system without degrading performance
- Key Contributions: Fairness via Source Throttling(FST) provides two major mechanisms
  - 1) Runtime fairness evaluation
  - 2) Dynamic request throttling
- Major Results: improve performance by 25.6%/14.5% and reduce unfairness by 44%/36%

Outline

- Background, Problem & Goal
- Novelty
- Key Approach & Ideas
- Mechanism
- Methodology & Evaluation
- Conclusion

- Takeaways
- Strengths and Weaknesses
- Thoughts & Ideas
- Open Discussion

Takeaways

- In order to ensure good performance for multiple applications in a shared system, controlling system-wide fairness is necessary
- By implementing FST one can decrease system complexity, due to the fact that no more coordination between multiple fairness techniques is needed

Outline

- Background, Problem & Goal
- Novelty
- Key Approach & Ideas
- Mechanism
- Methodology & Evaluation
- Conclusion

- Takeaways
- Strengths and Weaknesses
- Thoughts & Ideas
- Open Discussion

Strengths

- A new approach to an old problem, which will only **get worse** with rising core counts
- In addition to improving **system-wide fairness** it also provides comparable or superior **performance** compared to prior fairness mechanisms
- Reduce system **complexity** by replacing multiple resource-based mechanisms with FST
- FST can accomplish multiple **different** fairness objectives
- The evaluation provides a good overview, while the case study provides more insight
- It is well written

Weaknesses/Limitations

- **False** positive and negative in the pollution filter
- Implementation cost of FST may **scale** poorly since the number of cores directly determines the cost
- Diminishing returns on a system with a lot of **thread migration** and **context switches**
- The optimal unfairness threshold mentioned in the paper might be hard to find

Outline

- Background, Problem & Goal
- Novelty
- Key Approach & Ideas
- Mechanism
- Methodology & Evaluation
- Conclusion

- Takeaways
- Strengths and Weaknesses
- Thoughts & Ideas
- Open Discussion

Thoughts and Ideas

- Interval-based estimation and throttling
  - What impact will an application with rapidly and randomly changing memory intensity have?
- Aggressiveness levels
  - Would it make sense to have throttle function based on slowdown instead of fixed levels?
- Security aspects are not evaluated
  - Could a single or a group of bad actors attack FST?

Outline

- Background, Problem & Goal
- Novelty
- Key Approach & Ideas
- Mechanism
- Methodology & Evaluation
- Conclusion

- Takeaways
- Strengths and Weaknesses
- Thoughts & Ideas
- Open Discussion

Discussion starters

- Will the problem become more important over time?
- Are there situations where FST might not work?
- Do you think the increase in cost due to higher bank and core counts will be overshadowed by the increase in performance?
- Can you think of some disadvantages that I missed or even some way of improving FST?

Backup Slides

Algorithm 2 Estimation of T_{excess} for core i

Every cycle
  if *inter-core cache or DRAM bus or DRAM bank or DRAM row-buffer interference* **then**
    set $InterferencePerCore$ bit i
    set $InterferingCoreId$ in delayed memory request
  end if
  if $InterferencePerCore$ bit i is set **then**
    Increment $ExcessCycles$ for core i
  end if

Every L2 cache fill for a miss due to interference OR
Every time a memory request which is a row-buffer miss due to interference is serviced
  reset $InterferencePerCore$ bit of core i
  $InterferingCoreId$ of core i = i (no interference)

Every time a memory request is scheduled to DRAM
  if Core i has no requests waiting on any bank which is busy servicing another core j (j != i) **then**
    reset $InterferencePerCore$ bit of core i
  end if

1: check wether some sort of inter-core interference is present
2: if increment the ExcessCycles counter

1: whenever a interfered with memory request is serviced reset InterferencePerCore bit and set InterferingCoreId of core i to i
2: whenever a memory request is scheduled and also has no other request waiting on any bank busy servicing another core

Dynamic request throttling

Algorithm 1 Dynamic Request Throttling

```
if Estimated Unfairness > Unfairness Threshold then
    Throttle down application with the smallest slowdown
    Throttle up application with the largest slowdown
    Reset Successive Fairness Achieved Intervals

else
    if Successive Fairness Achieved Intervals = threshold then
        Throttle all applications up
        Reset Successive Fairness Achieved Intervals
    else
        Increment Successive Fairness Achieved Intervals
    end if
end if
```

- This is a simplified version for dual cores
- After a certain number of fair intervals both cores are allowed to throttle up

Algorithm 3 Dynamic Request Throttling - General Form

```
if Estimated Unfairness > Unfairness Threshold then
    Throttle down application that causes most interference (App_interfering) for application with largest slowdown
    Throttle up application with the largest slowdown (App_slow)
    Reset Successive Fairness Achieved Intervals
    Reset Intervals To Wait To Throttle Up for App_interfering.

    // Preventing bank service denial
    if App_interfering throttled lower than Switch_thr AND causes greater than Interference_thr amount of App_slow's total interference then
        Temporarily stop prioritizing App_interfering due to row hits in memory controller
    end if
    if App_RowHitNotPrioritized has not been App_interfering for SwitchBack_thr intervals then
        Allow it to be prioritized in memory controller based on row-buffer hit status of its requests
    end if

    for all applications except App_interfering and App_slow do
        if Intervals To Wait To Throttle Up = threshold1 then
            throttle up
            Reset Intervals To Wait To Throttle Up for this app.
        else
            Increment Intervals To Wait To Throttle Up for this app.
        end if
    end for

else
    if Successive Fairness Achieved Intervals = threshold2 then
        Throttle up application with the smallest slowdown
        Reset Successive Fairness Achieved Intervals
    else
        Increment Successive Fairness Achieved Intervals
    end if
end if
```

1) Responsible for throttling down the most interfering application

2) Solving bank service denial due to FR-FCFS

3) Throttling up all applications that are neither App_{slow} nor $App_{interfering}$ every *threshold1* intervals

4) Throttling up ??? application after number of threshold2 intervals

FST Parameter used in the evaluation

- We use 8 different aggressiveness levels:
  - 2%, 3%, 4%, 5%, 10%, 25%, 50% and 100%

$Fairness\ Threshold$	$Successive\ Fairness\ Achieved\ Intervals\ Threshold$	$Intervals\ Wait\ To\ Throttle\ Up$	$Interval\ Length$
1.4	4	2	25Kinsts

$Switch_{thr}$	$Interference_{thr}$	$SwitchBack_{thr}$
5%	70%	3 intervals