# Memory Systems in the Many-Core Era: Some Challenges and Solution Directions

Onur Mutlu

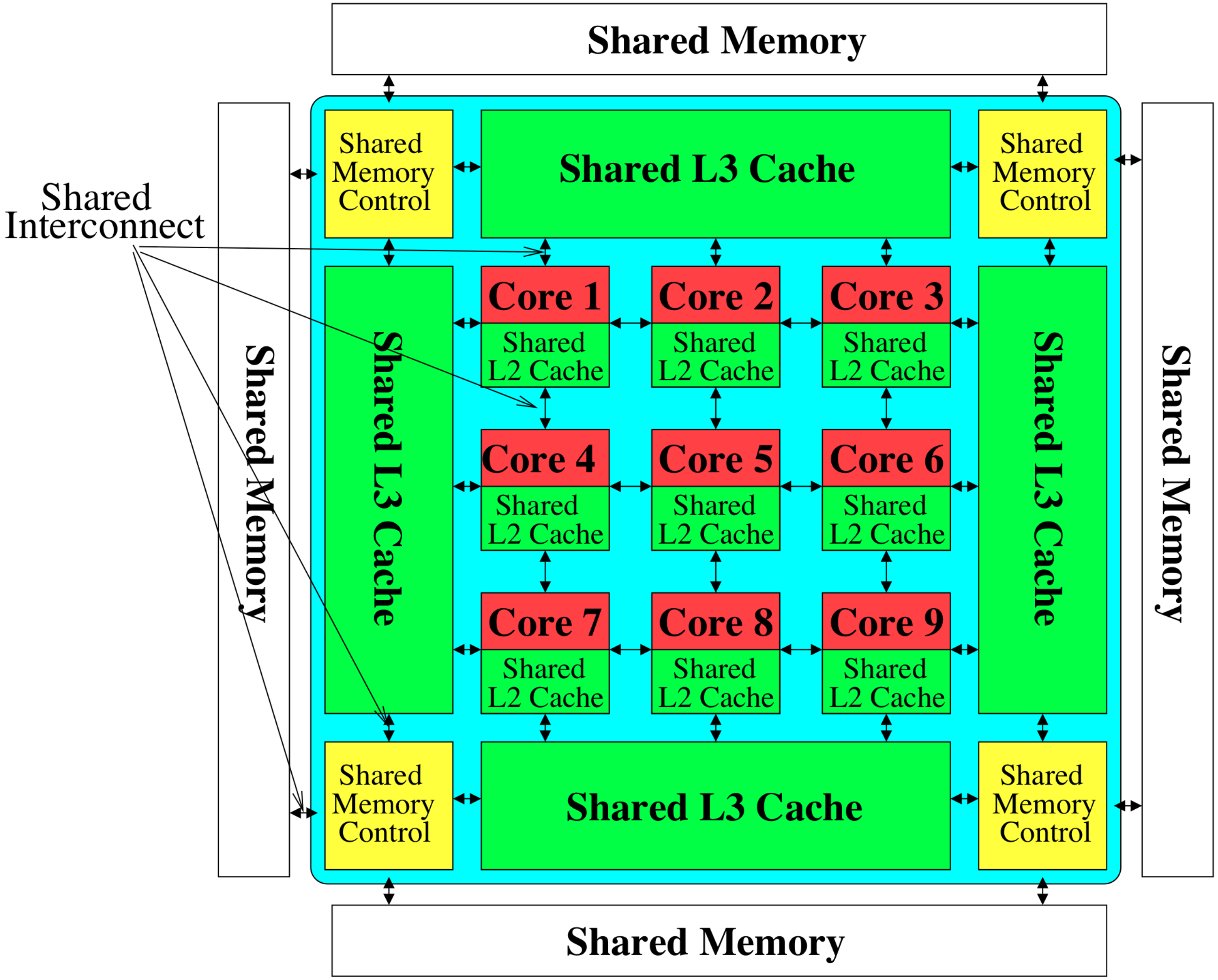
http://www.ece.cmu.edu/~omutlu

June 5, 2011

ISMM/MSPC

Carnegie Mellon

# Modern Memory System: A Shared Resource



# The Memory System

- The memory system is a fundamental performance and power bottleneck in almost all computing systems: server, mobile, embedded, desktop, sensor

- The memory system must scale (in size, performance, efficiency, cost) to maintain performance and technology scaling

- Recent technology, architecture, and application trends lead to new requirements from the memory system:
  - Scalability (technology and algorithm)
  - Fairness and QoS-awareness
  - Energy/power efficiency

# Agenda

- Technology, Application, Architecture Trends
- Requirements from the Memory Hierarchy
- Research Challenges and Solution Directions
  - Main Memory Scalability
  - QoS support: Inter-thread/application interference
- Summary

# Technology Trends

- DRAM does not scale well beyond N nm [ITRS 2009, 2010]
  - Memory scaling benefits: density, capacity, cost

- Energy/power already key design limiters
  - Memory hierarchy responsible for a large fraction of power
    - IBM servers: ~50% energy spent in off-chip memory hierarchy [Lefurgy+, IEEE Computer 2003]
    - DRAM consumes power when idle and needs periodic refresh

- More transistors (cores) on chip
- Pin bandwidth not increasing as fast as number of transistors
  - Memory is the major shared resource among cores
  - More pressure on the memory hierarchy

# Application Trends

- Many different threads/applications/virtual-machines (will) concurrently share the memory system
  - Cloud computing/servers: Many workloads consolidated on-chip to improve efficiency
  - GP-GPU, CPU+GPU, accelerators: Many threads from multiple applications
  - Mobile: Interactive + non-interactive consolidation

- Different applications with different requirements (SLAs)
  - Some applications/threads require performance guarantees
  - Modern hierarchies do not distinguish between applications

- Applications are increasingly data intensive
  - More demand for memory capacity and bandwidth

# Architecture/System Trends

- Sharing of memory hierarchy

- More cores and components
  - More pressure on the memory hierarchy
- Asymmetric cores: Performance asymmetry, CPU+GPUs, accelerators, …
  - Motivated by energy efficiency and Amdahl’s Law
- Different cores have different performance requirements
  - Memory hierarchies do not distinguish between cores

- Different goals for different systems/users
  - System throughput, fairness, per-application performance
  - Modern hierarchies are not flexible/configurable

# Summary: Major Trends Affecting Memory

- Need for main memory capacity and bandwidth increasing

- New need for handling inter-application interference; providing fairness, QoS

- Need for memory system flexibility increasing

- Main memory energy/power is a key system design concern

- DRAM is not scaling well

# Agenda

- Technology, Application, Architecture Trends
- Requirements from the Memory Hierarchy
- Research Challenges and Solution Directions
    - Main Memory Scalability
    - QoS support: Inter-thread/application interference
- Summary

# Requirements from an Ideal Memory System

- Traditional
  - High system performance
  - Enough capacity
  - Low cost

- New
  - Technology scalability
  - QoS support and configurability
  - Energy (and power, bandwidth) efficiency

# Requirements from an Ideal Memory System

- Traditional
  - High system performance: Need to reduce inter-thread interference
  - Enough capacity: Emerging tech. and waste management can help
  - Low cost: Other memory technologies can help

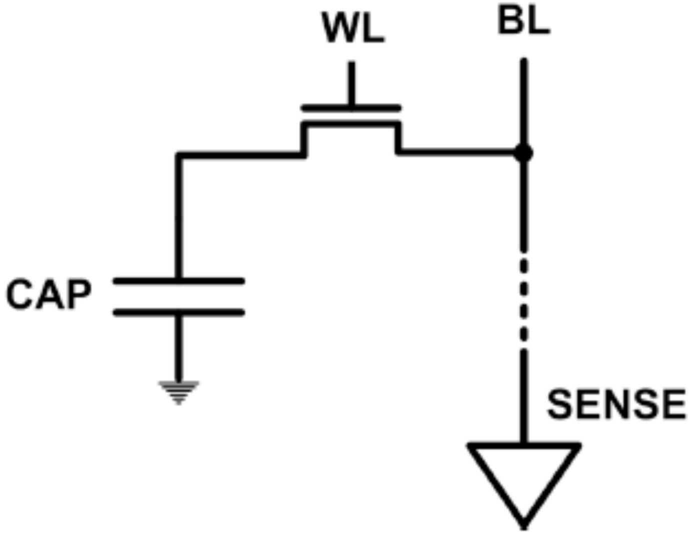
- New
  - Technology scalability
    - Emerging memory technologies (e.g., PCM) can help
  - QoS support and configurability
    - Need HW mechanisms to control interference and build QoS policies
  - Energy (and power, bandwidth) efficiency
    - One-size-fits-all design wastes energy; emerging tech. can help?

# Agenda

- Technology, Application, Architecture Trends
- Requirements from the Memory Hierarchy
- Research Challenges and Solution Directions
  - Main Memory Scalability
  - QoS support: Inter-thread/application interference
- Summary

# The DRAM Scaling Problem

- DRAM stores charge in a capacitor (charge-based memory)
  - Capacitor must be large enough for reliable sensing
  - Access transistor should be large enough for low leakage and high retention time
  - Scaling beyond 40-35nm (2013) is challenging [ITRS, 2009]



- DRAM capacity, cost, and energy/power hard to scale

# Concerns with DRAM as Main Memory

- Need for main memory capacity and bandwidth increasing
  - DRAM capacity hard to scale

- Main memory energy/power is a key system design concern
  - DRAM consumes high power due to leakage and refresh

- DRAM technology scaling is becoming difficult
  - DRAM capacity and cost may not continue to scale

# Possible Solution 1: Tolerate DRAM

- Overcome DRAM shortcomings with
  - System-level solutions
  - Changes to DRAM microarchitecture, interface, and functions

# Possible Solution 2: Emerging Technologies

- Some emerging resistive memory technologies are more scalable than DRAM (and they are non-volatile)

- Example: Phase Change Memory
  - Data stored by changing phase of special material
  - Data read by detecting material's resistance
  - Expected to scale to 9nm (2022 [ITRS])
  - Prototyped at 20nm (Raoux+, IBM JRD 2008)
  - Expected to be denser than DRAM: can store multiple bits/cell

- But, emerging technologies have shortcomings as well
  - Can they be enabled to replace/augment/surpass DRAM?

# Phase Change Memory: Pros and Cons

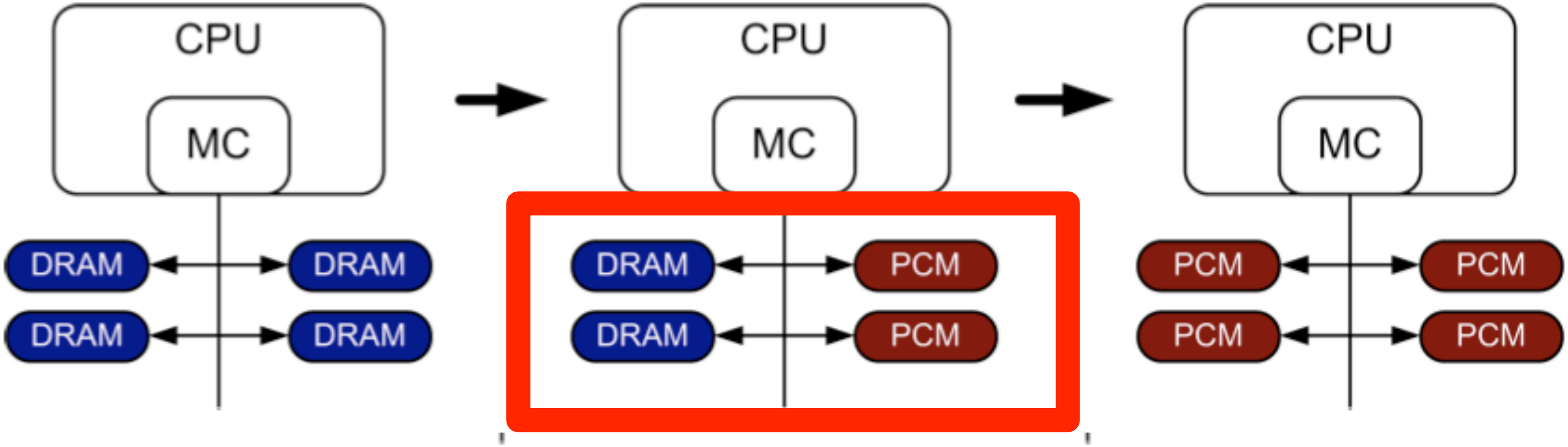
- Pros over DRAM
  - Better technology scaling (capacity and cost)
  - Non volatility
  - Low idle power (no refresh)

- Cons
  - Higher latencies: ~4-15x DRAM (especially write)
  - Higher active energy: ~2-50x DRAM (especially write)
  - Lower endurance (a cell dies after ~$10^8$ writes)

- Challenges in enabling PCM as DRAM replacement/helper:
  - Mitigate PCM shortcomings
  - Find the right way to place PCM in the system
  - Ensure secure and fault-tolerant PCM operation

# PCM-based Main Memory (I)

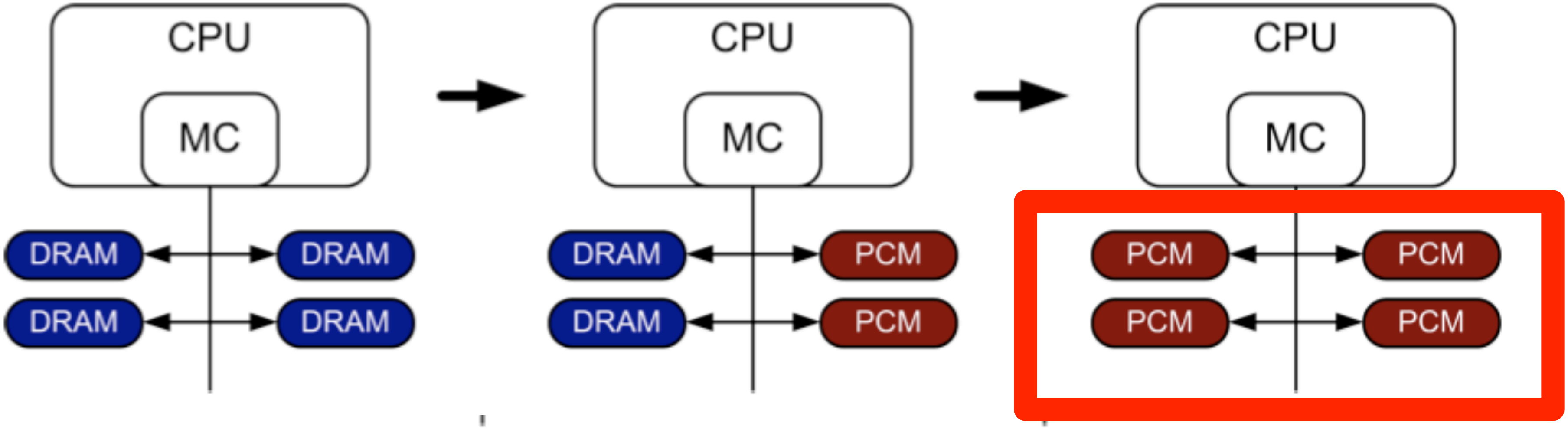
- How should PCM-based (main) memory be organized?



- Hybrid PCM+DRAM [Qureshi+ ISCA'09, Dhiman+ DAC'09]:
  - How to partition/migrate data between PCM and DRAM
    - Energy, performance, endurance
  - Is DRAM a cache for PCM or part of main memory?
  - How to design the hardware and software
    - Exploit advantages, minimize disadvantages of each technology

# PCM-based Main Memory (II)

- How should PCM-based (main) memory be organized?



- Pure PCM main memory [Lee et al., ISCA'09, Top Picks'10]:
  - How to redesign entire hierarchy (and cores) to overcome PCM shortcomings
    - Energy, performance, endurance

# PCM-Based Memory Systems: Research Challenges

- Partitioning
  - Should DRAM be a cache or main memory, or configurable?
  - What fraction? How many controllers?

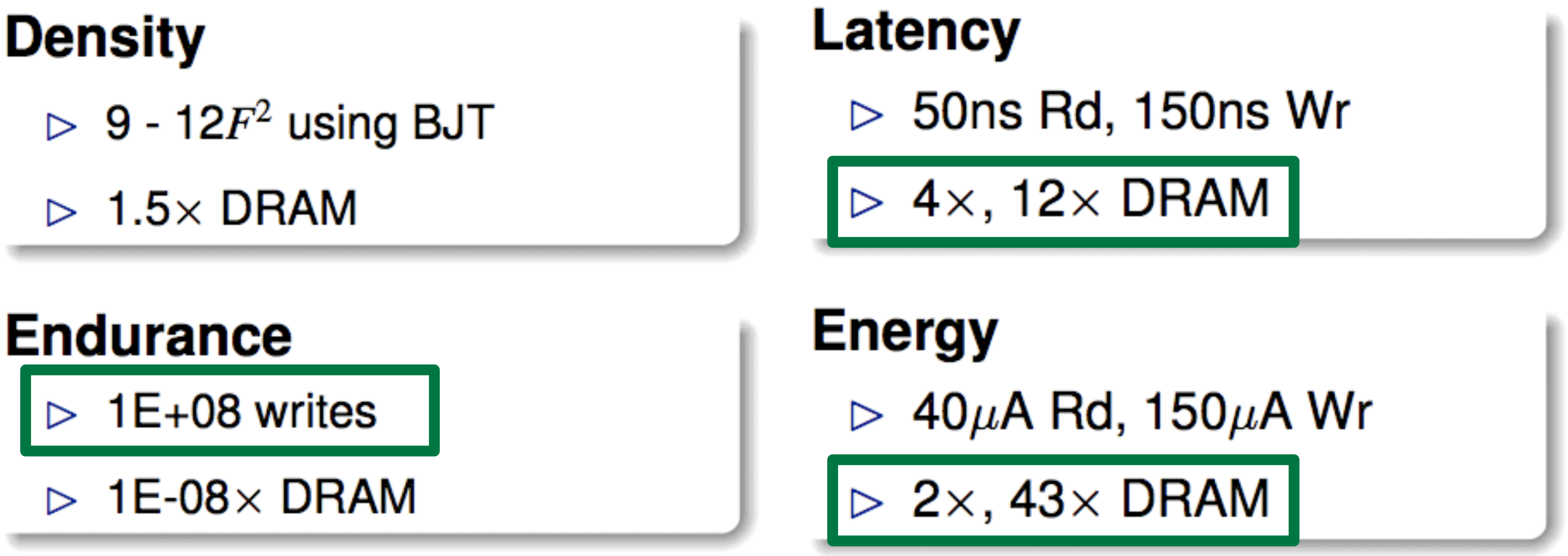
- Data allocation/movement (energy, performance, lifetime)
  - Who manages allocation/movement?
  - What are good control algorithms?
    - Latency-critical, heavily modified → DRAM, otherwise PCM?
    - Preventing denial/degradation of service

- Design of cache hierarchy, memory controllers, OS
  - Mitigate PCM shortcomings, exploit PCM advantages

- Design of PCM/DRAM chips and modules
  - Rethink the design of PCM/DRAM with new requirements

# An Initial Study: Replace DRAM with PCM

- Lee, Ipek, Mutlu, Burger, "Architecting Phase Change Memory as a Scalable DRAM Alternative," ISCA 2009.
  - Surveyed prototypes from 2003-2008 (e.g. IEDM, VLSI, ISSCC)
  - Derived "average" PCM parameters for F=90nm



**Density**

▷ 9 - 12$F^2$ using BJT

▷ 1.5× DRAM

**Latency**

▷ 50ns Rd, 150ns Wr

▷ 4×, 12× DRAM

**Endurance**

▷ 1E+08 writes

▷ 1E-08× DRAM

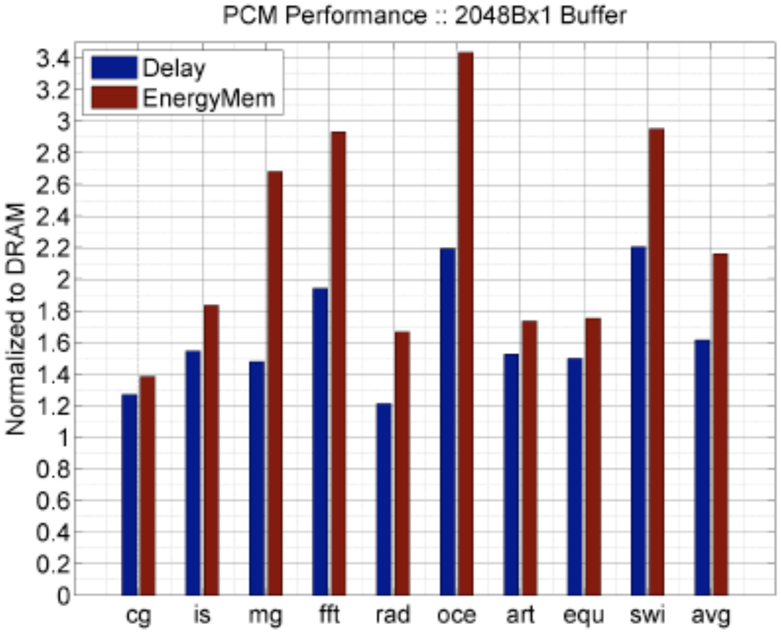
**Energy**

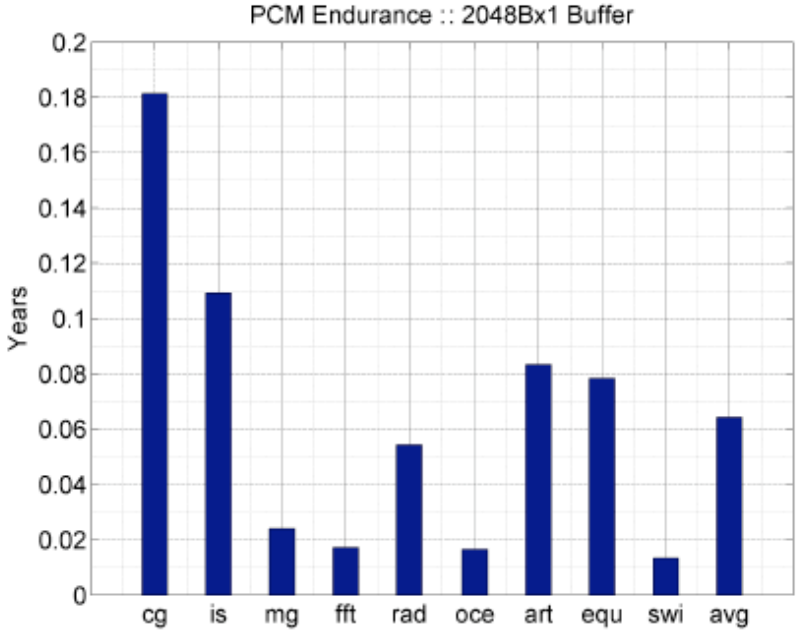
▷ 40$\mu$A Rd, 150$\mu$A Wr

▷ 2×, 43× DRAM

# Results: Naïve Replacement of DRAM with PCM

- Replace DRAM with PCM in a 4-core, 4MB L2 system
- PCM organized the same as DRAM: row buffers, banks, peripherals
- 1.6x delay, 2.2x energy, 500-hour average lifetime





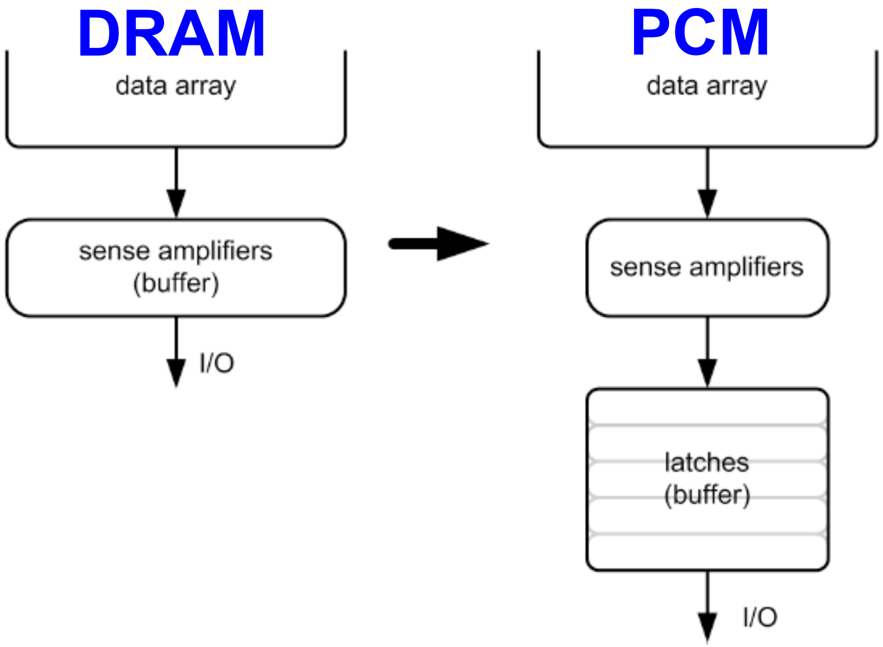
- Lee, Ipek, Mutlu, Burger, "Architecting Phase Change Memory as a Scalable DRAM Alternative," ISCA 2009.

# Architecting PCM to Mitigate Shortcomings

- Idea 1: Use narrow row buffers in each PCM chip
  → Reduces write energy, peripheral circuitry

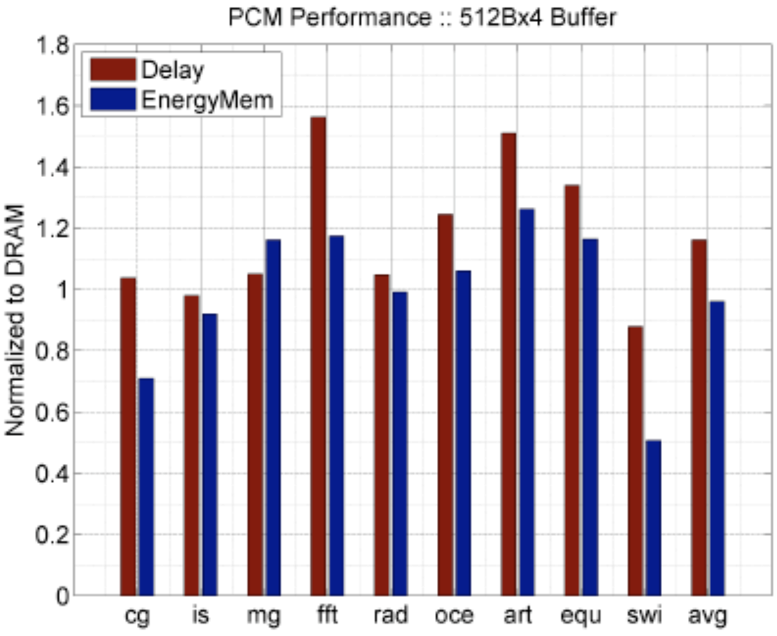
- Idea 2: Use multiple row buffers in each PCM chip
  → Reduces array reads/writes → better endurance, latency, energy

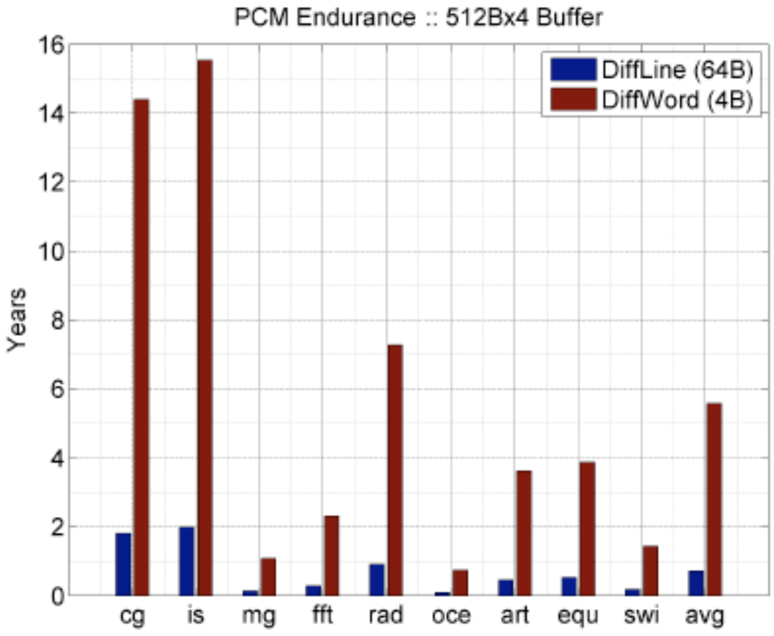
- Idea 3: Write into array at cache block or word granularity
  → Reduces unnecessary wear



# Results: Architected PCM as Main Memory

- 1.2x delay, 1.0x energy, 5.6-year average lifetime
- Scaling improves energy, endurance, density





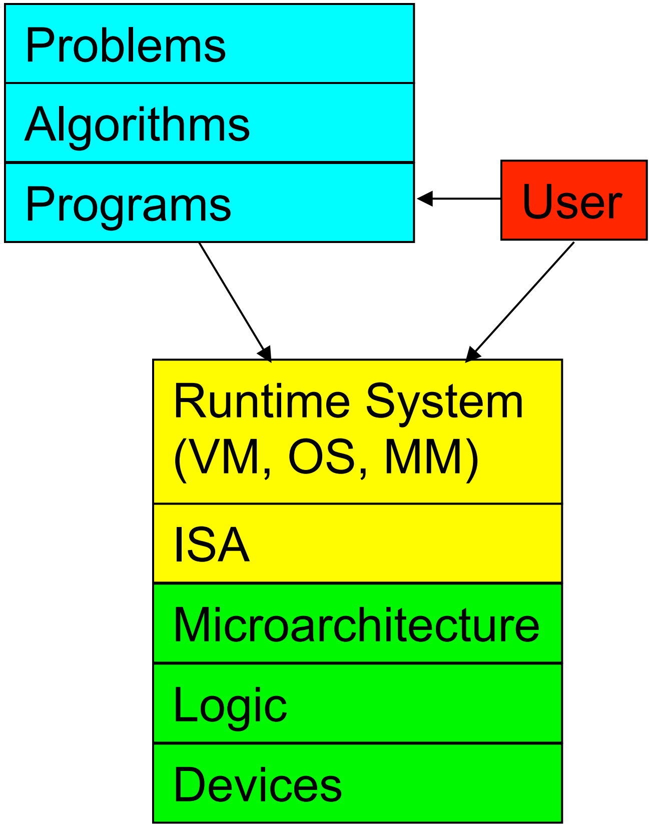
- Caveat 1: Worst-case lifetime is much shorter (no guarantees)
- Caveat 2: Intensive applications see large performance and energy hits
- Caveat 3: Optimistic PCM parameters?

# PCM as Main Memory: Research Challenges

- Many research opportunities from technology layer to algorithms layer

- Enabling PCM/NVM
  - How to maximize performance?
  - How to maximize lifetime?
  - How to prevent denial of service?

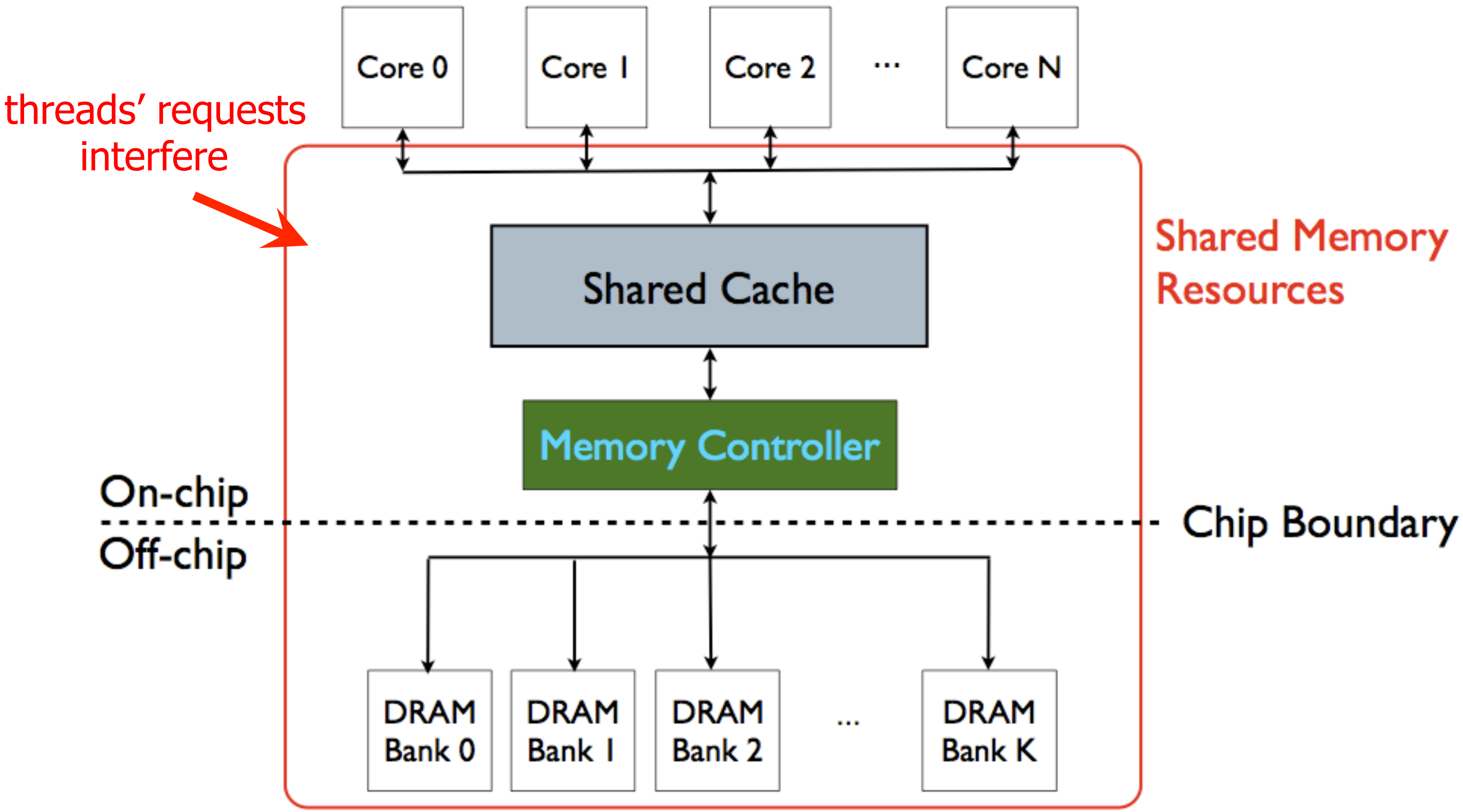
- Exploiting PCM/NVM
  - How to exploit non-volatility?
  - How to minimize energy consumption?
  - How to minimize cost?
  - How to exploit NVM on chip?



# Agenda

- Technology, Application, Architecture Trends
- Requirements from the Memory Hierarchy
- Research Challenges and Solution Directions
    - Main Memory Scalability
    - QoS support: Inter-thread/application interference
- Summary

# Memory System is the Major Shared Resource

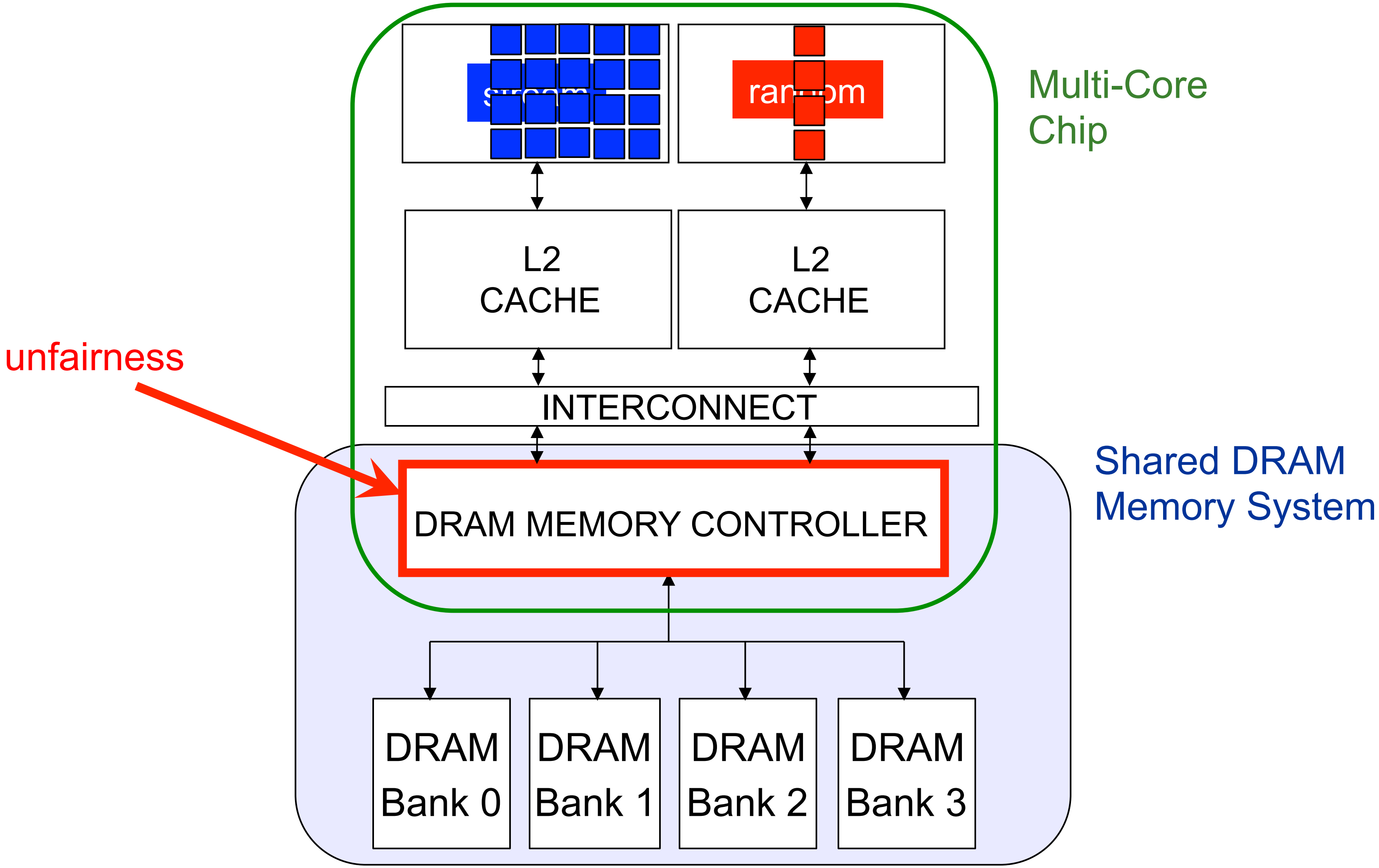


# Inter-Thread/Application Interference

- Problem: Threads share the memory system, but memory system does not distinguish between threads' requests

- Existing memory systems
  - Free-for-all, shared based on demand
  - Control algorithms thread-unaware and thread-unfair
  - Aggressive threads can deny service to others
  - Do not try to reduce or control inter-thread interference

# Uncontrolled Interference: An Example



# A Memory Performance Hog

```
// initialize large arrays A, B

for (j=0; j<N; j++) {
    index = j*linesize;
    A[index] = B[index];
    …
}
```

**streaming**

**STREAM**

- Sequential memory access
- Very high row buffer locality (96% hit rate)
- Memory intensive

```
// initialize large arrays A, B

for (j=0; j<N; j++) {
    index = rand();
    A[index] = B[index];
    …
}
```

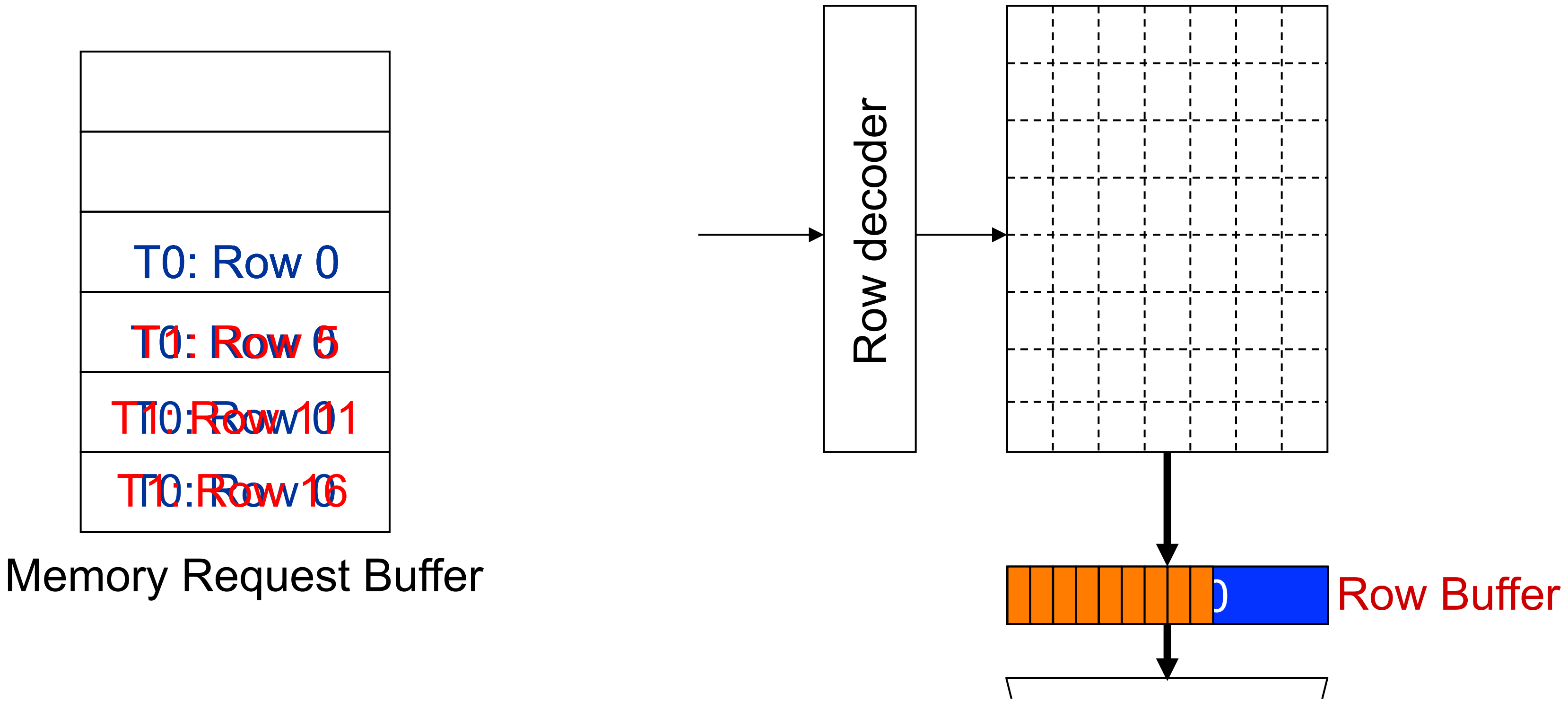
**random**

**RANDOM**

- Random memory access
- Very low row buffer locality (3% hit rate)
- Similarly memory intensive

Moscibroda and Mutlu, “Memory Performance Attacks,” USENIX Security 2007.

# What Does the Memory Hog Do?

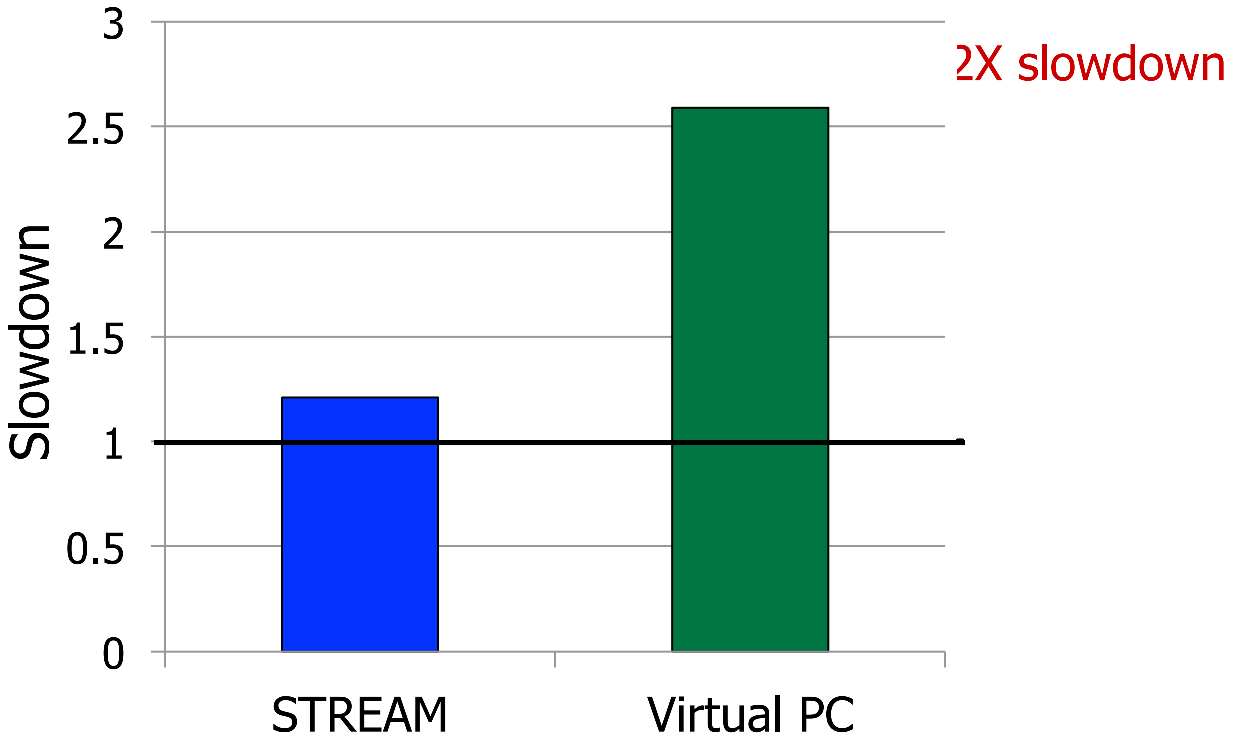


Row size: 8KB, cache block size: 64B

128 (8KB/64B) requests of T0 serviced before T1

Moscibroda and Mutlu, "Memory Performance Attacks," USENIX Security 2007.

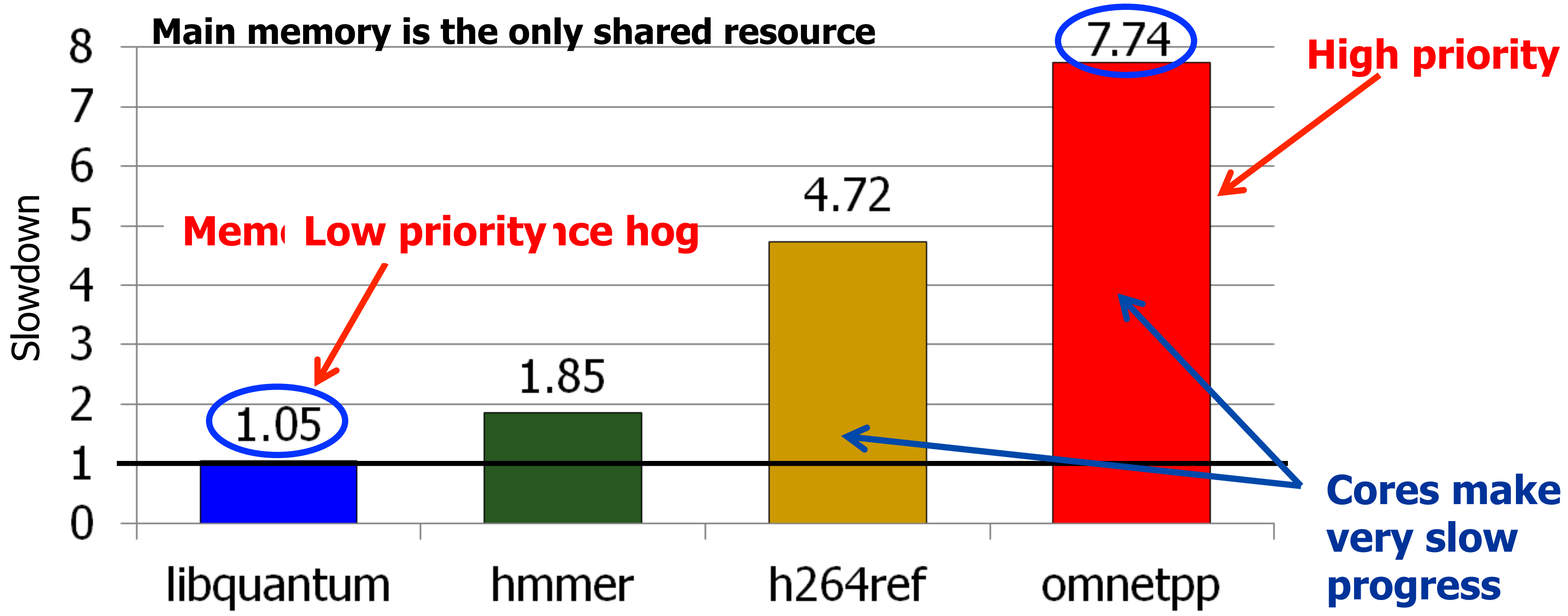
# Effect of the Memory Performance Hog



Results on Intel Pentium D running Windows XP
(Similar results for Intel Core Duo and AMD Turion, and on Fedora Linux)

Moscibroda and Mutlu, "Memory Performance Attacks," USENIX Security 2007.

# Problems due to Uncontrolled Interference



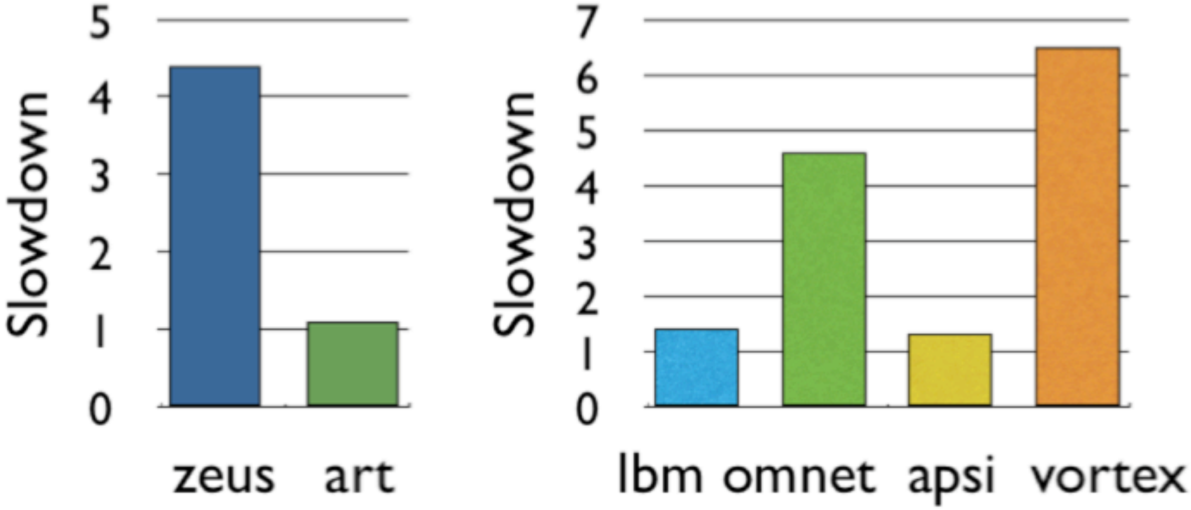
Mem Low priority nce hog

High priority

Cores make very slow progress

- Unfair slowdown of different threads [MICRO'07, ISCA'08, ASPLOS'10]
- Low system performance [MICRO'07, ISCA'08, HPCA'10, MICRO'10]
- Vulnerability to denial of service [USENIX Security'07]
- Priority inversion: unable to enforce priorities/SLAs [MICRO'07]
- Poor performance predictability (no performance isolation)

# Problems due to Uncontrolled Interference



- Unfair slowdown of different threads [MICRO'07, ISCA'08, ASPLOS'10]
- Low system performance [MICRO'07, ISCA'08, HPCA'10, MICRO'10]
- Vulnerability to denial of service [USENIX Security'07]
- Priority inversion: unable to enforce priorities/SLAs [MICRO'07]
- Poor performance predictability (no performance isolation)

# How Do We Solve The Problem?

- Inter-thread interference is uncontrolled in all memory resources
  - Memory controller
  - Interconnect
  - Caches
- We need to control it
  - i.e., design an interference-aware (QoS-aware) memory system

# QoS-Aware Memory Systems: Challenges

- How do we reduce inter-thread interference?
  - Improve system performance and core utilization
  - Reduce request serialization and core starvation

- How do we control inter-thread interference?
  - Provide mechanisms to enable system software to enforce QoS policies
  - While providing high system performance

- How do we make the memory system configurable/flexible?
  - Enable flexible mechanisms that can achieve many goals
    - Provide fairness or throughput when needed
    - Satisfy performance guarantees when needed

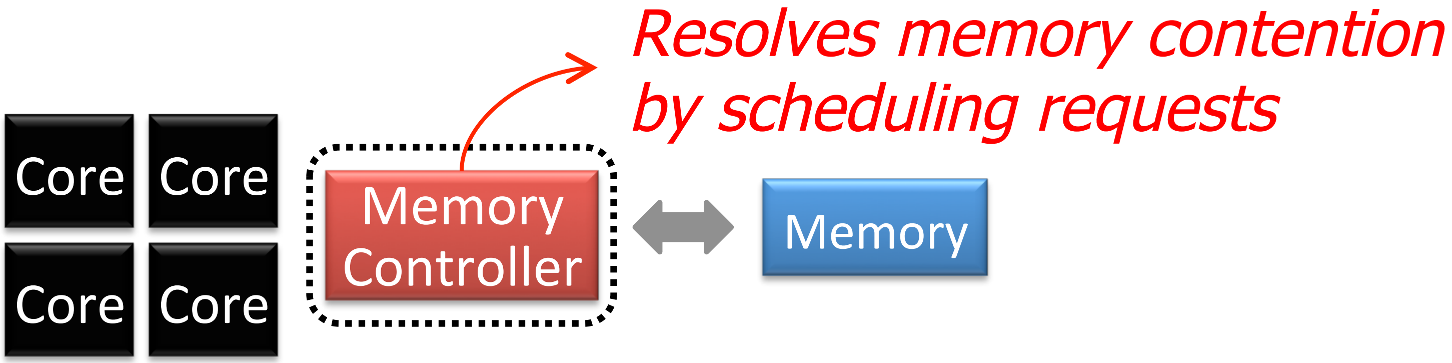
# Designing QoS-Aware Memory Systems: Approaches

- Smart resources: Design each shared resource to have a configurable interference control/reduction mechanism
  - QoS-aware memory controllers [Mutlu+ MICRO'07] [Moscibroda+, Usenix Security'07] [Mutlu+ ISCA'08, Top Picks'09] [Kim+ HPCA'10] [Kim+ MICRO'10, Top Picks'11]
  - QoS-aware interconnects [Das+ MICRO'09, ISCA'10, Top Picks '11] [Grot+ MICRO'09, ISCA'11]
  - 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 [Ebrahimi+ ASPLOS'10, ISCA'11] [Ebrahimi+ MICRO'09] [Nychis+ HotNets'10]
  - QoS-aware data mapping to memory controllers [Muralidhara+ CMU TR'11]
  - QoS-aware thread scheduling to cores

# Agenda

- Technology, Application, Architecture Trends
- Requirements from the Memory Hierarchy
- Research Challenges and Solution Directions
  - Main Memory Scalability
  - QoS support: Inter-thread/application interference
    - Smart Resources: Thread Cluster Memory Scheduling
    - Dumb Resources: Fairness via Source Throttling
- Summary

# QoS-Aware Memory Scheduling



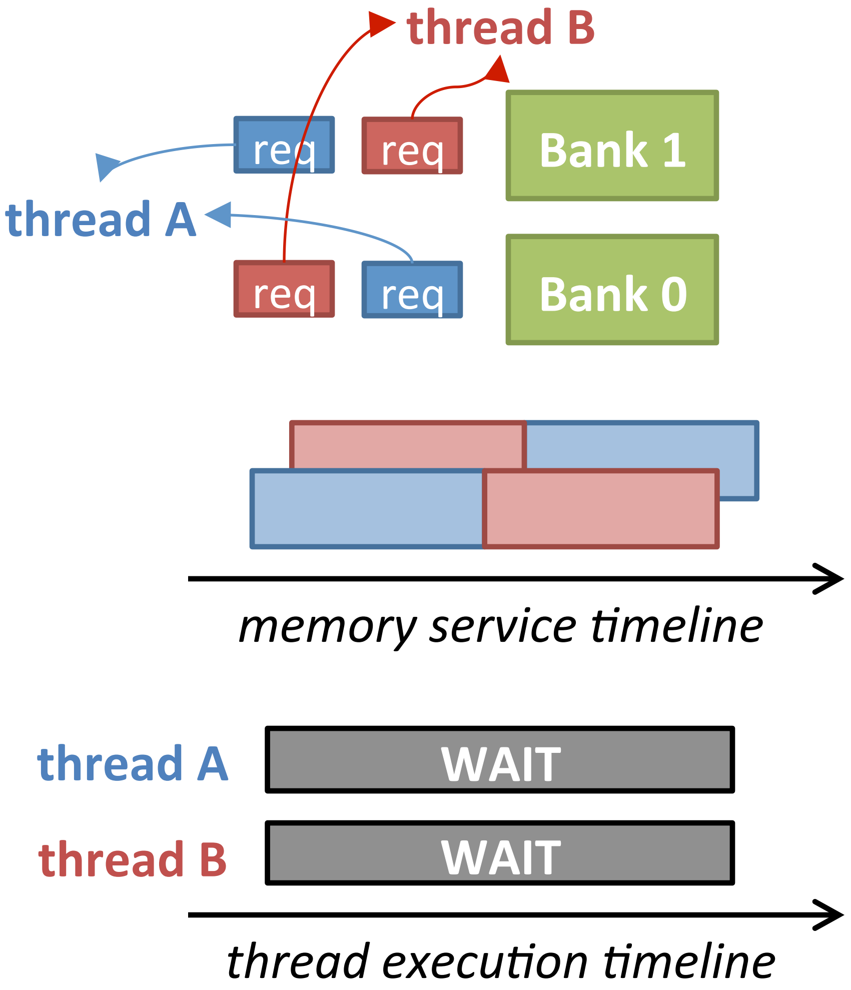
*Resolves memory contention by scheduling requests*

- How to schedule requests to provide
  - High system performance
  - High fairness to applications
  - Configurability to system software
- Memory controller needs to be aware of threads

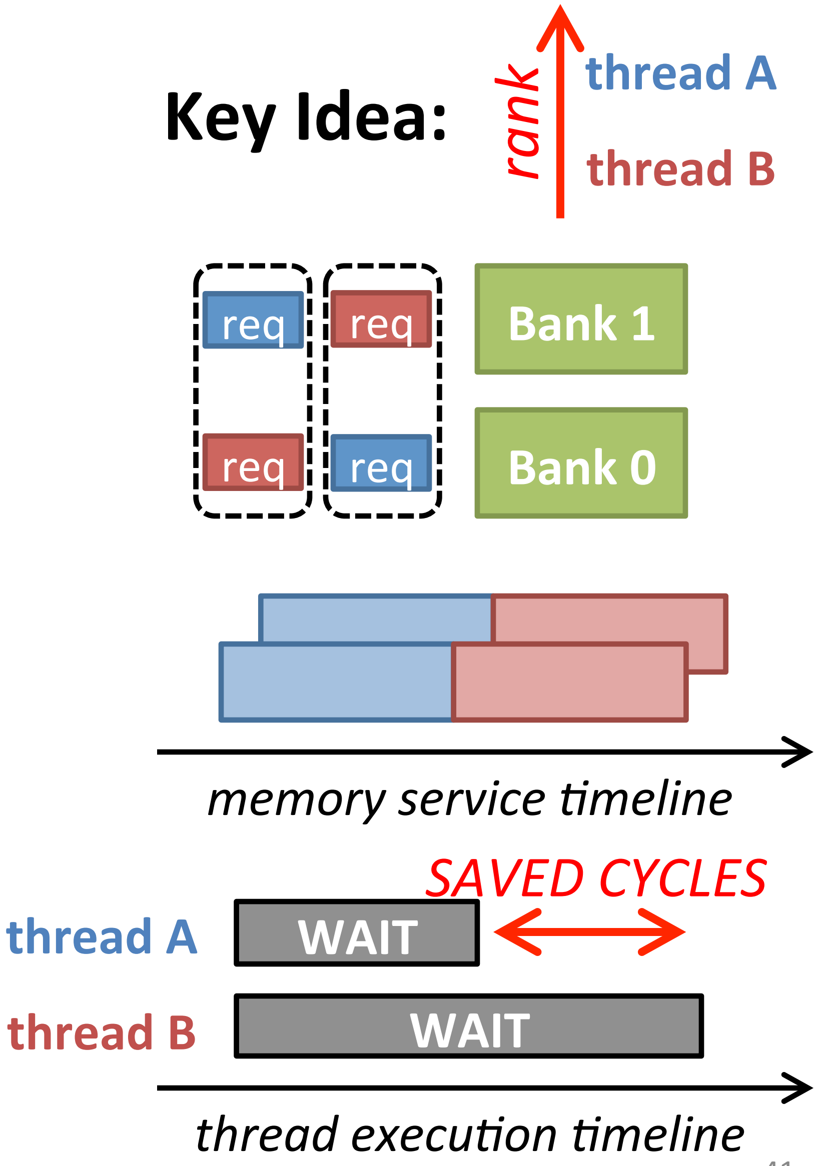
# QoS-Aware Memory Scheduling: Evolution

- Stall-time fair memory scheduling [Mutlu+ MICRO'07]
    - Idea: Estimate and balance thread slowdowns
    - Takeaway: Proportional thread progress improves performance, especially when threads are "heavy" (memory intensive)
- Parallelism-aware batch scheduling [Mutlu+ ISCA'08, Top Picks'09]
    - Idea: Rank threads and service in rank order (to preserve bank parallelism); batch requests to prevent starvation
- ATLAS memory scheduler [Kim+ HPCA'10]

# Within-Thread Bank Parallelism



**Key Idea:**



# QoS-Aware Memory Scheduling: Evolution

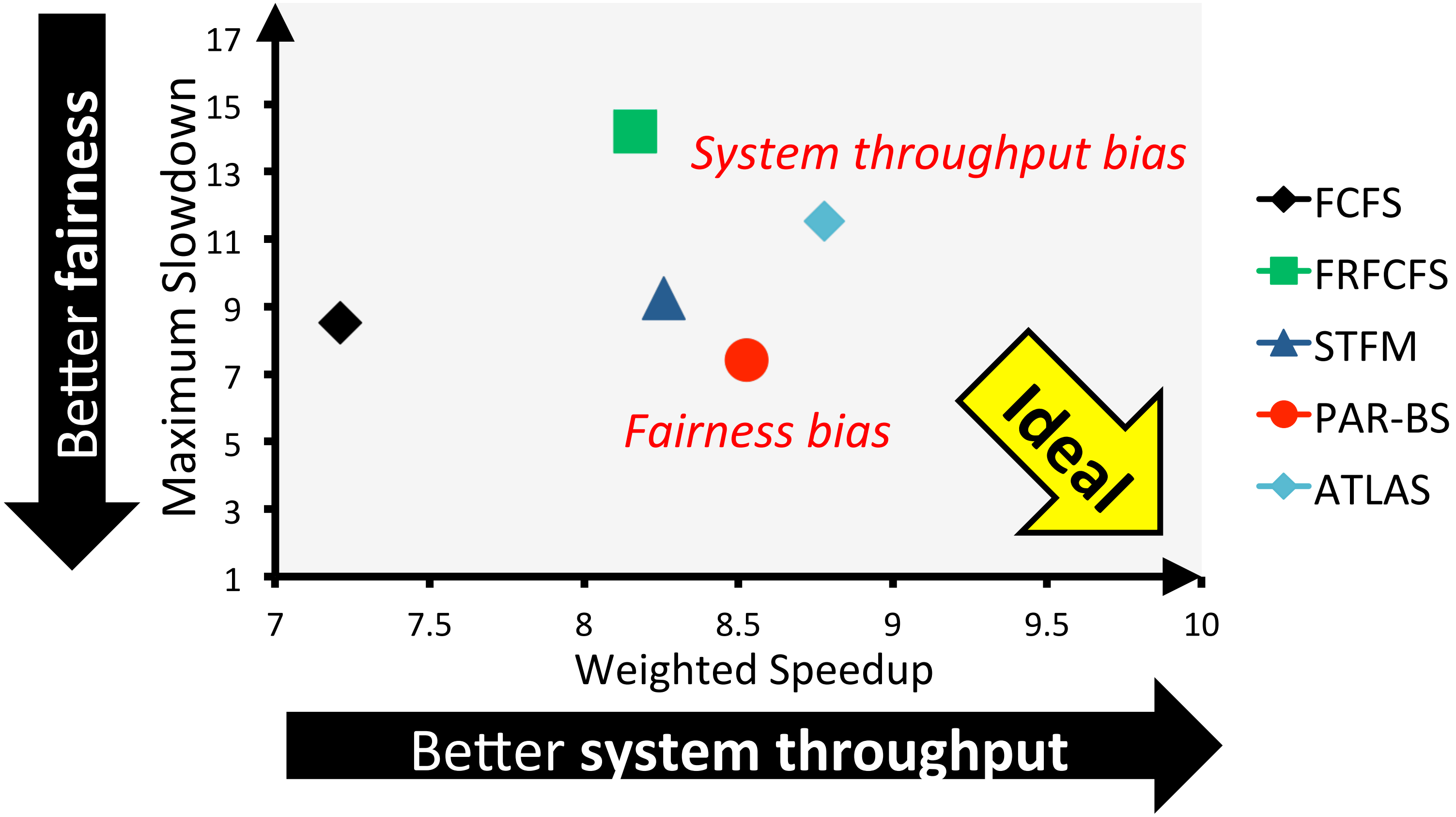
- Stall-time fair memory scheduling [Mutlu+ MICRO'07]
  - Idea: Estimate and balance thread slowdowns
  - Takeaway: Proportional thread progress improves performance, especially when threads are "heavy" (memory intensive)

- Parallelism-aware batch scheduling [Mutlu+ ISCA'08, Top Picks'09]
  - Idea: Rank threads and service in rank order (to preserve bank parallelism); batch requests to prevent starvation
  - Takeaway: Preserving within-thread bank-parallelism improves performance; request batching improves fairness

- ATLAS memory scheduler [Kim+ HPCA'10]
  - Idea: Prioritize threads that have attained the least service from the memory scheduler
  - Takeaway: Prioritizing "light" threads improves performance

# Previous Scheduling Algorithms are Biased

*24 cores, 4 memory controllers, 96 workloads*



*System throughput bias*

*Fairness bias*

Ideal

*No previous memory scheduling algorithm provides both the best fairness and system throughput*

# Throughput vs. Fairness

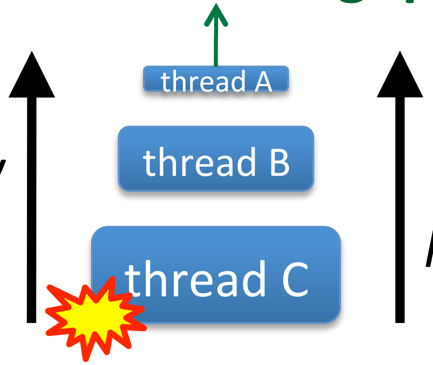
***Throughput biased** approach*

Prioritize less memory-intensive threads

***Fairness biased** approach*

Take turns accessing memory

**Good for throughput**

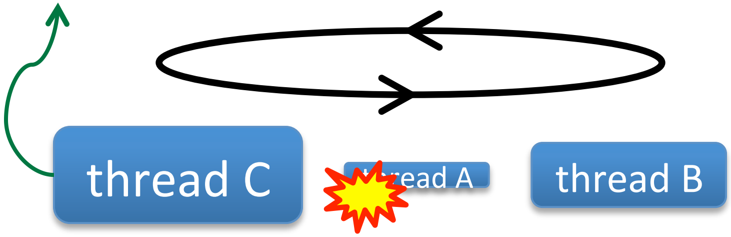


*less memory intensive*

*higher priority*

***starvation → unfairness***

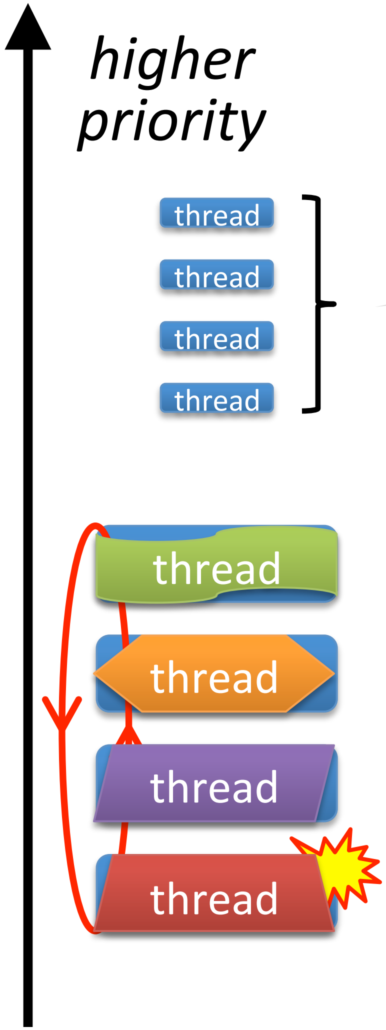
**Does not starve**



***not prioritized → reduced throughput***

**Single policy for all threads is insufficient**

# Achieving the Best of Both Worlds



**For Throughput**

**Prioritize memory-non-intensive threads**

**For Fairness**

**Unfairness caused by memory-intensive being prioritized over each other**

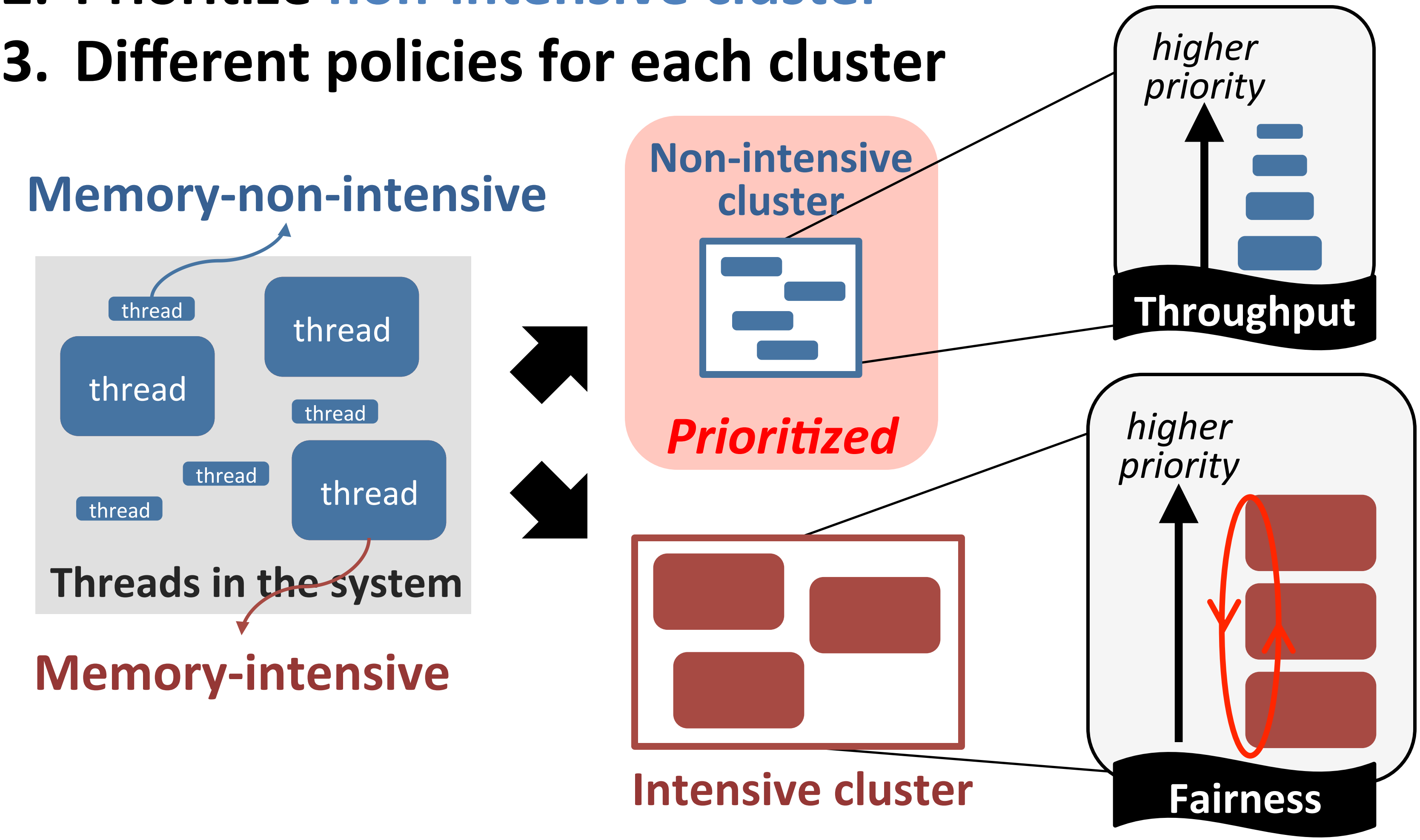
- Shuffle thread ranking

**Memory-intensive threads have different vulnerability to interference**

- Shuffle asymmetrically

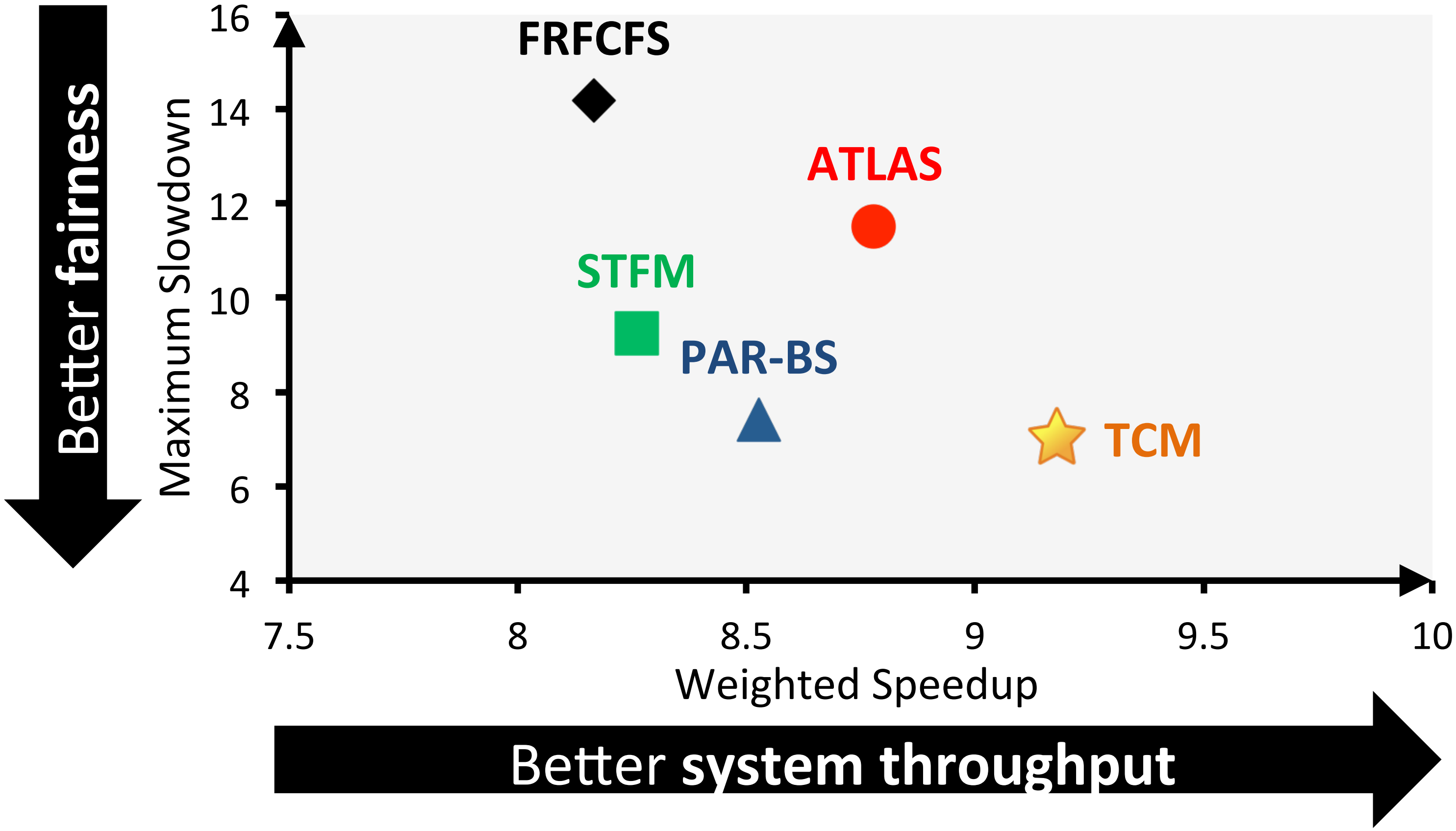
# Thread Cluster Memory Scheduling [Kim+ MICRO'10]

1. **Group threads into two *clusters***
2. **Prioritize non-intensive cluster**
3. **Different policies for each cluster**



# TCM: Throughput and Fairness

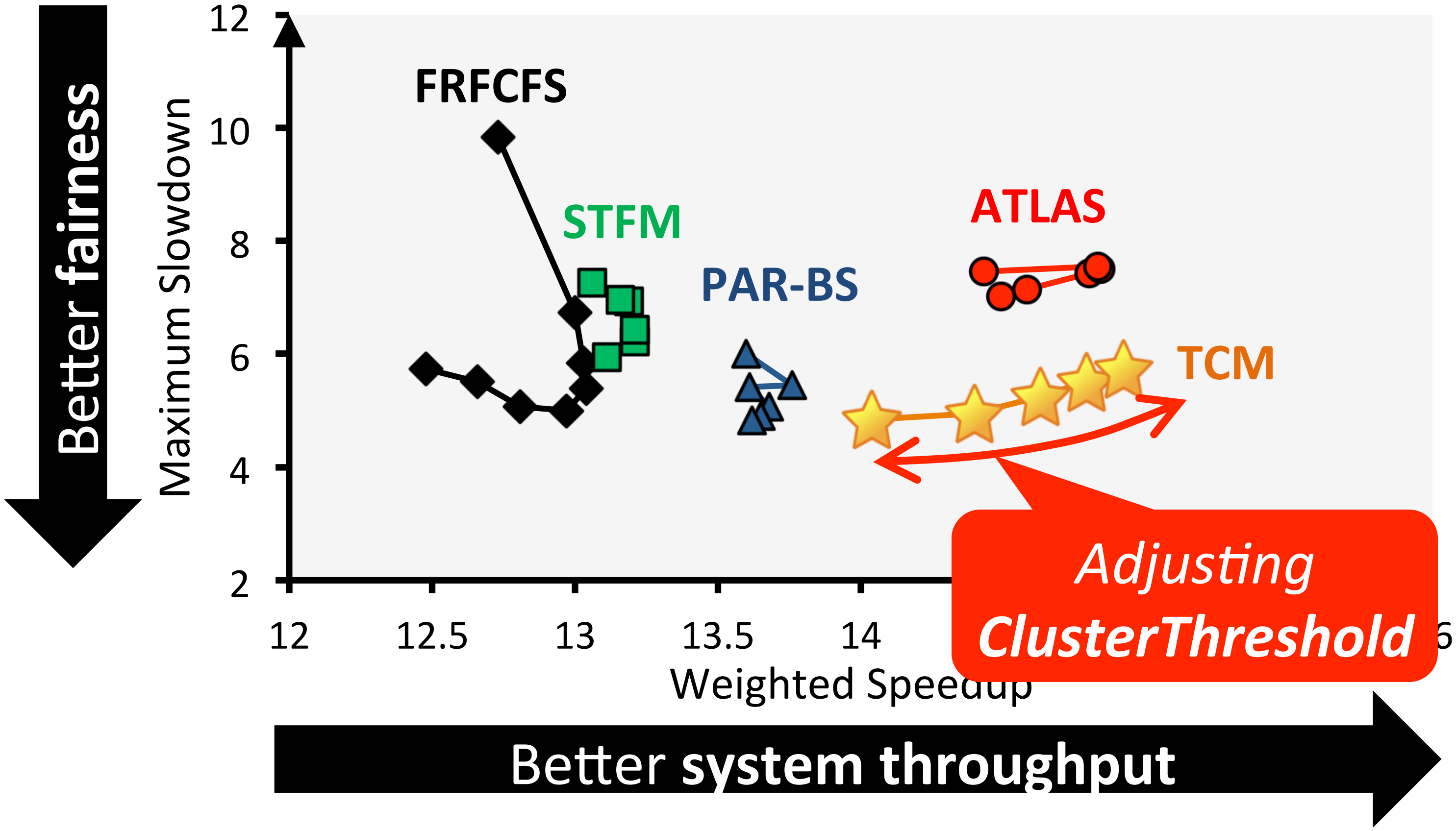
*24 cores, 4 memory controllers, 96 workloads*



*TCM, a heterogeneous scheduling policy, provides best fairness and system throughput*

# TCM: Fairness-Throughput Tradeoff

**When configuration parameter is varied…**



Better fairness

Better system throughput

*Adjusting ClusterThreshold*

*TCM allows robust fairness-throughput tradeoff*

# Agenda

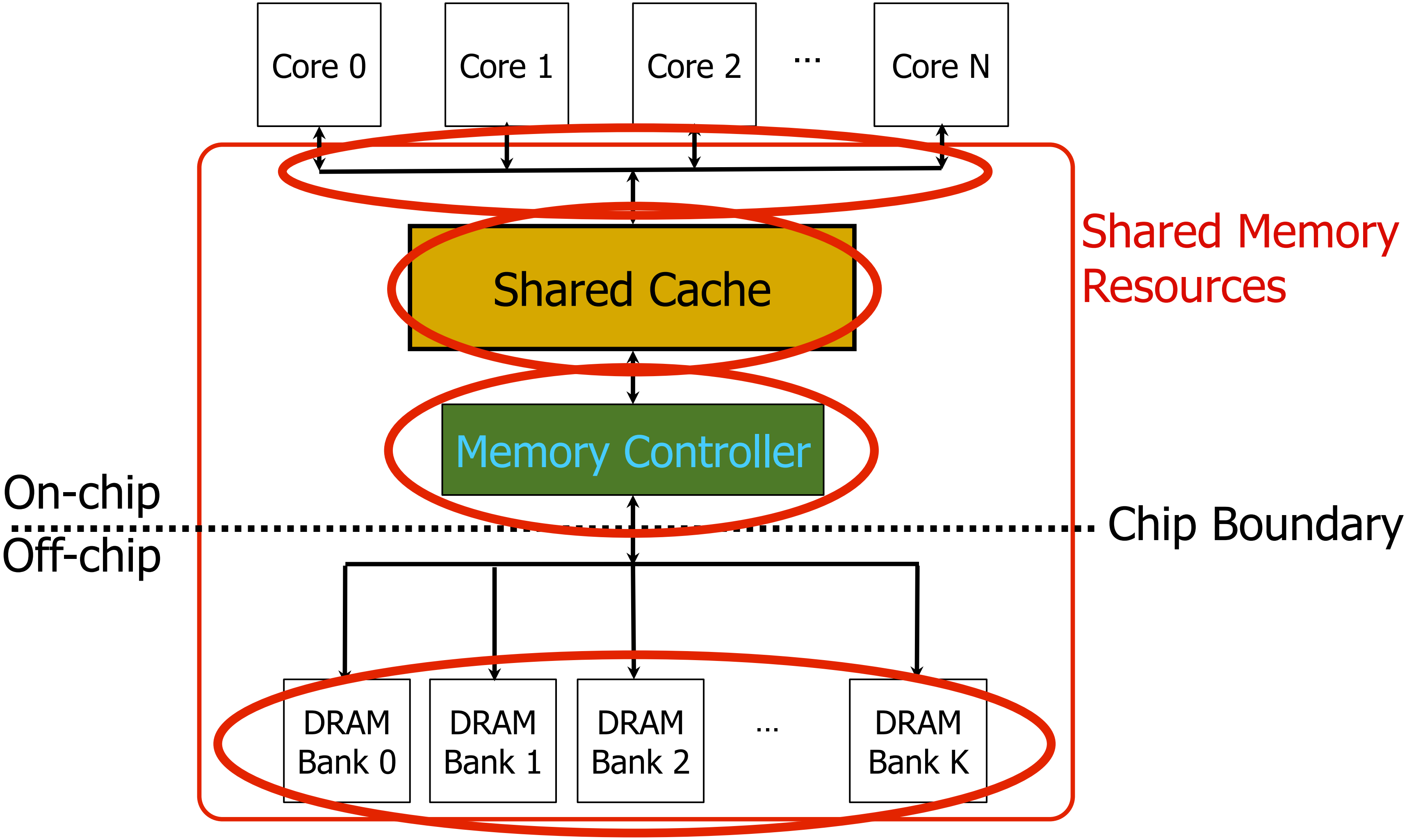
- Technology, Application, Architecture Trends
- Requirements from the Memory Hierarchy
- Research Challenges and Solution Directions
  - Main Memory Scalability
  - QoS support: Inter-thread/application interference
    - Smart Resources: Thread Cluster Memory Scheduling
    - Dumb Resources: Fairness via Source Throttling
- Summary

# Designing QoS-Aware Memory Systems: Approaches

- Smart resources: Design each shared resource to have a configurable interference control/reduction mechanism
  - QoS-aware memory controllers [Mutlu+ MICRO'07] [Moscibroda+, Usenix Security'07] [Mutlu+ ISCA'08, Top Picks'09] [Kim+ HPCA'10] [Kim+ MICRO'10, Top Picks'11]
  - QoS-aware interconnects [Das+ MICRO'09, ISCA'10, Top Picks '11] [Grot+ MICRO'09, ISCA'11]
  - 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 [Ebrahimi+ ASPLOS'10, ISCA'11] [Ebrahimi+ MICRO'09] [Nychis+ HotNets'10]
  - QoS-aware data mapping to memory controllers [Muralidhara+ CMU TR'11]
  - QoS-aware thread scheduling to cores

# 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?

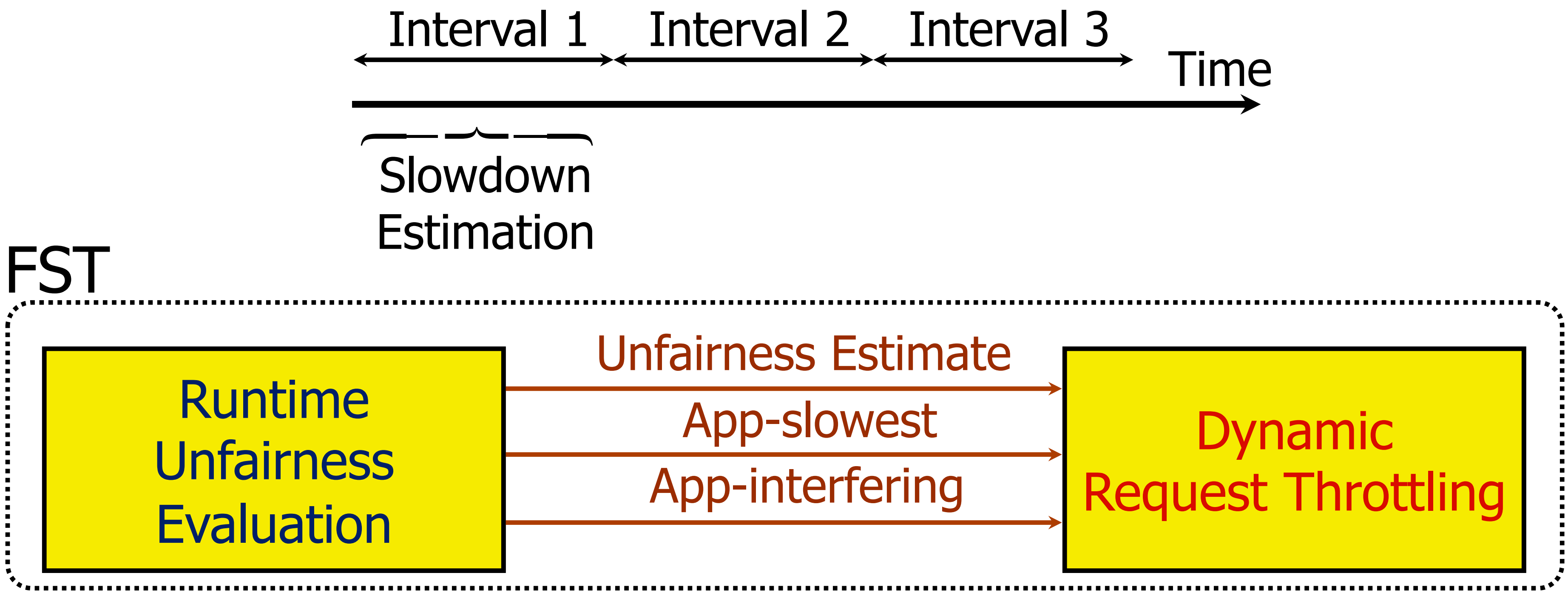
# An Alternative Approach: Source Throttling

- Manage inter-thread interference at the cores, 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.

# 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
}
```

# 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

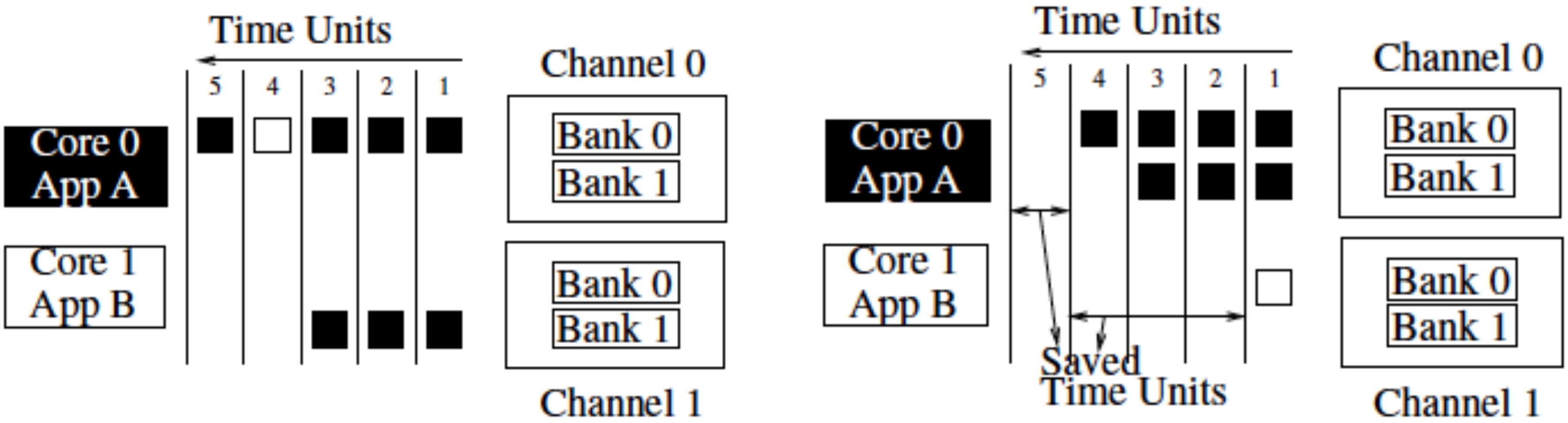
- Combined approaches are even more powerful
  - Source throttling and resource-based interference control

# Designing QoS-Aware Memory Systems: Approaches

- Smart resources: Design each shared resource to have a configurable interference control/reduction mechanism
  - QoS-aware memory controllers [Mutlu+ MICRO'07] [Moscibroda+, Usenix Security'07] [Mutlu+ ISCA'08, Top Picks'09] [Kim+ HPCA'10] [Kim+ MICRO'10, Top Picks'11]
  - QoS-aware interconnects [Das+ MICRO'09, ISCA'10, Top Picks '11] [Grot+ MICRO'09, ISCA'11]
  - 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 [Ebrahimi+ ASPLOS'10, ISCA'11] [Ebrahimi+ MICRO'09] [Nychis+ HotNets'10]
  - QoS-aware data mapping to memory controllers [Muralidhara+ CMU TR'11]
  - QoS-aware thread scheduling to cores

# Another Way of Reducing Interference

- Memory Channel Partitioning
  - Idea: Map badly-interfering applications' pages to different channels [Muralidhara+ CMU TR'11]



(a) Conventional Page Mapping.

(b) Channel Partitioning.

- Separate data of low/high intensity and low/high row-locality applications
- Especially effective in reducing interference of threads with "medium" and "heavy" memory intensity

# 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 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

# Two Related Talks at ISCA

- How to design QoS-aware memory systems (memory scheduling and source throttling) in the presence of prefetching
  - Ebrahimi et al., "Prefetch-Aware Shared Resource Management for Multi-Core Systems," ISCA'11.
  - Monday afternoon (Session 3B)

- How to design scalable QoS mechanisms in on-chip interconnects
  - Idea: Isolate shared resources in a region, provide QoS support only within the region, ensure interference-free access to the region
  - Grot et al., "Kilo-NOC: A Heterogeneous Network-on-Chip Architecture for Scalability and Service Guarantees," ISCA'11.
  - Wednesday morning (Session 8B)

# Agenda

- Technology, Application, Architecture Trends
- Requirements from the Memory Hierarchy
- Research Challenges and Solution Directions
    - Main Memory Scalability
    - QoS support: Inter-thread/application interference
        - Smart Resources: Thread Cluster Memory Scheduling
        - Dumb Resources: Fairness via Source Throttling
- Conclusions

# Conclusions

- Technology, application, architecture trends dictate new needs from memory system

- A fresh look at (re-designing) the memory hierarchy
  - Scalability: Enabling new memory technologies
  - QoS, fairness & performance: Reducing and controlling inter-application interference: QoS-aware memory system design
  - Efficiency: Customizability, minimal waste, new technologies

- Many exciting research topics in fundamental areas across the system stack
  - Hardware/software/device cooperation essential

# Thank you.

# Memory Systems in the Many-Core Era: Some Challenges and Solution Directions

Onur Mutlu

http://www.ece.cmu.edu/~omutlu

June 5, 2011

ISMM/MSPC

Carnegie Mellon