18-447
Computer Architecture
Lecture 25: Main Memory Wrap-Up

Prof. Onur Mutlu
Carnegie Mellon University
Spring 2014, 4/2/2014

Upcoming Seminar on DRAM (April 3)

- April 3, Thursday, 4pm, this room (CIC Panther Hollow)
- Prof. Rajeev Balasubramonian, Univ. of Utah
- Memory Architectures for Emerging Technologies and Workloads
  - The memory system will be a growing bottleneck for many workloads running on high-end servers. Performance improvements from technology scaling are also expected to decline in the coming decade. Therefore, new capabilities will be required in memory devices and memory controllers to achieve the next big leaps in performance and energy efficiency. Some of these capabilities will be inspired by emerging workloads (e.g., in-memory big-data, approximate computing, co-scheduled VMs), some will be inspired by new memory technologies (e.g., 3D stacking). The talk will discuss multiple early-stage projects in the Utah Arch lab that focus on DRAM parameter variation, near-data processing, and memory security.

Cloud Workshop All Day on April 4

- http://www.industry-academia.org/event-carnegie-mellon-cloud-workshop.html
- You need to register to attend. Gates 6115. Many talks:
- Keynote: Prof. Onur Mutlu – Carnegie Mellon – "Rethinking Memory System Design for Data-Intensive Computing"
- Prof. Rajeev Balasubramonian – Utah – "Practical Approaches to Memory Security in the Cloud"
- Bryan Chin – Cavium – "Head in the Clouds - Building a Chip for Scale-out Computing"
- Dr. Joon Kim - SK Hynix – "The Future of NVM Memories"
- Prof. Andy Pavlo - Carnegie Mellon – "OLTP on NVM: YMMV"
- Dr. John Busch – SanDisk – "The Impact of Flash Memory on the Future of Cloud Computing"
- Keynote: Prof. Greg Ganger – Carnegie Mellon – "Scheduling Heterogeneous Resources in Cloud Datacenters"
- Paul Rad – Rackspace – "OpenStack-Based High Performance Cloud Architecture"
- Charles Butler – Ubuntu – "Cloud Service Orchestration with JuJu"
- Prof. Mor Harchol-Balter - Carnegie Mellon – "Dynamic Power Management in Data Centers"
- Prof. Eric Xing – Carnegie Mellon – "Petuum: A New Platform for Cloud-based Machine Learning to Efficiently Solve Big Data Problems"
- Majid Bemanian – Imagination Technologies – "Security in the Cloud and Virtualized Mobile Devices"
- Robert Broberg – Cisco – "Cloud Security Challenges and Solutions"

Cloud Career Fair on April 4

- http://www.industry-academia.org/event-carnegie-mellon-cloud-workshop.html
- Gates 6121, 11am-3pm
- Runs in Room 6121 in parallel to the Tech Forum, from 11am to 3PM. IAP members will have informational/recruiting tables on site. During the breaks in the technical presentations and lunch, the Tech Forum attendees can network on lining up an internship or that first full-time engineering job. Students who are only interested and/or able to attend the Career Fair are welcome to do so, but please indicate this specific interest on your registration application (see the "Register Here" button below).

Memory Interference and Scheduling in Multi-Core Systems

Review: PAR-BS Pros and Cons

- Upsides:
  - First scheduler to address bank parallelism destruction across multiple threads
  - Simple mechanism (vs. STFM)
  - Batching provides fairness
  - Ranking enables parallelism awareness

- Downsides:
  - Implementation in multiple controllers needs coordination for best performance → too frequent coordination since batching is done frequently
  - Does not always prioritize the latency-sensitive applications

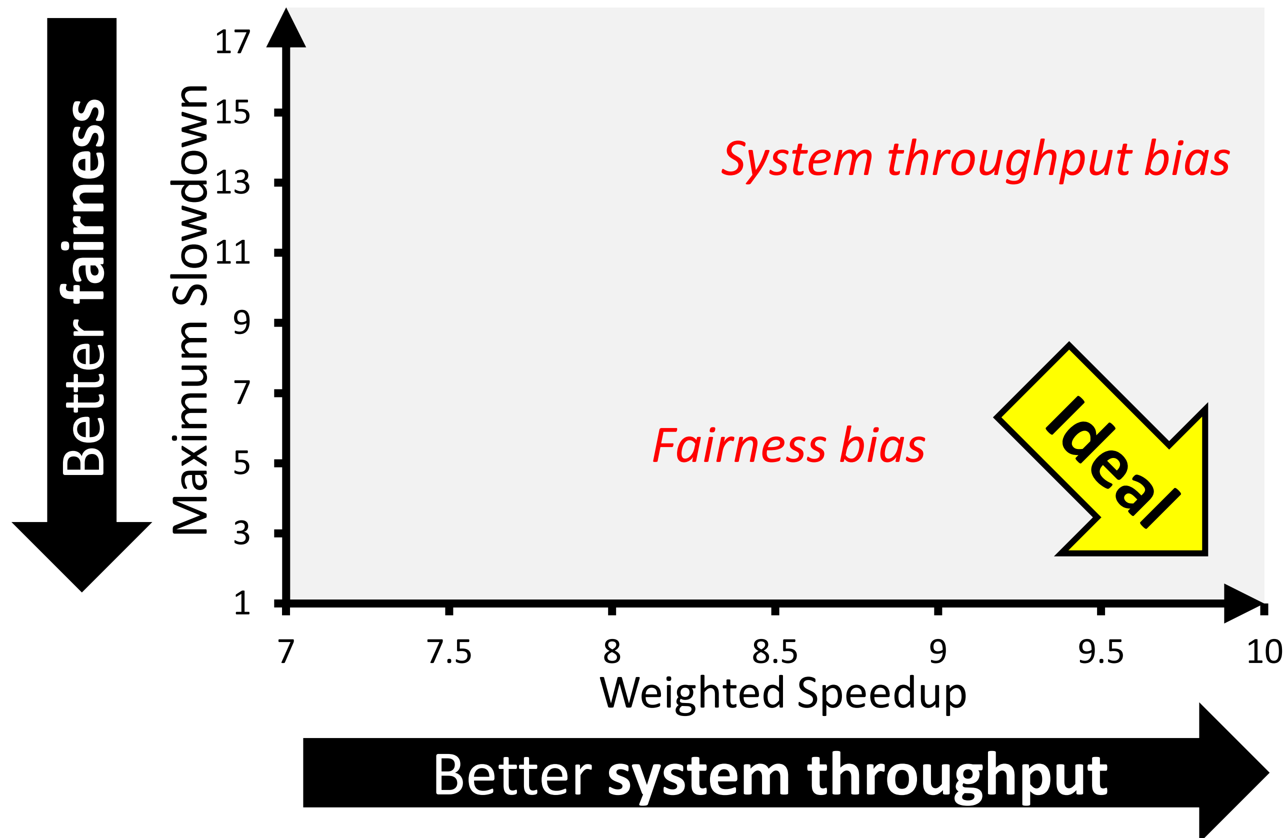
TCM: Thread Cluster Memory Scheduling

Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter,
"Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior"
43rd International Symposium on Microarchitecture (***MICRO***),
pages 65-76, Atlanta, GA, December 2010. Slides (pptx) (pdf)

TCM Micro 2010 Talk

Throughput vs. Fairness

24 cores, 4 memory controllers, 96 workloads



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

Throughput vs. Fairness

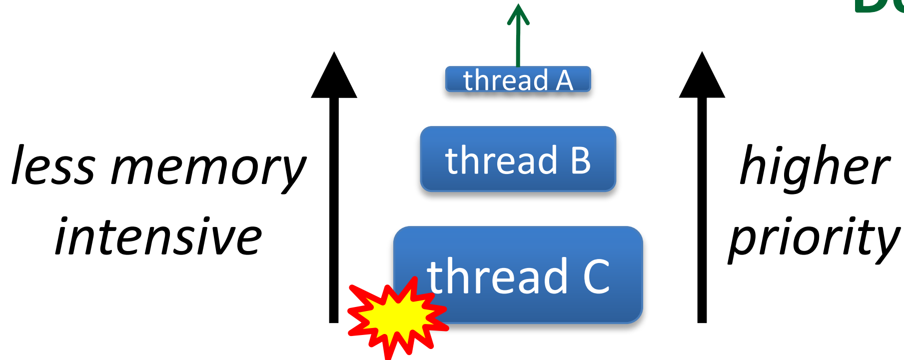
***Throughput biased** approach*

Prioritize less memory-intensive threads

***Fairness biased** approach*

Take turns accessing memory

Good for throughput



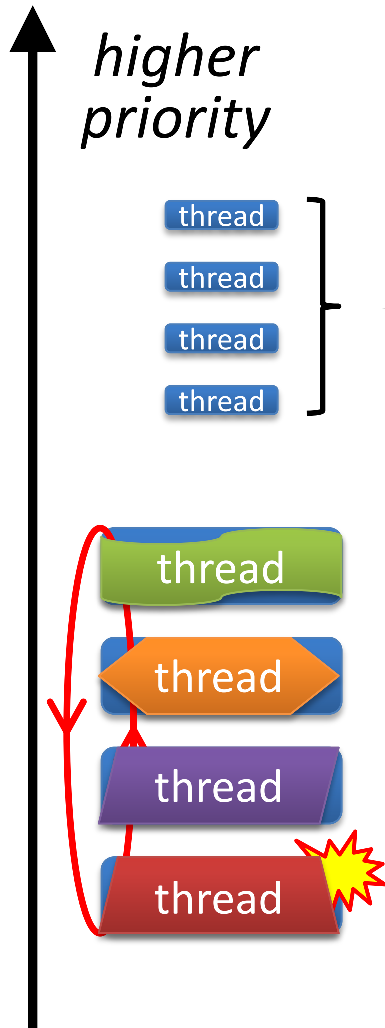
Does not starve

starvation → unfairness

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

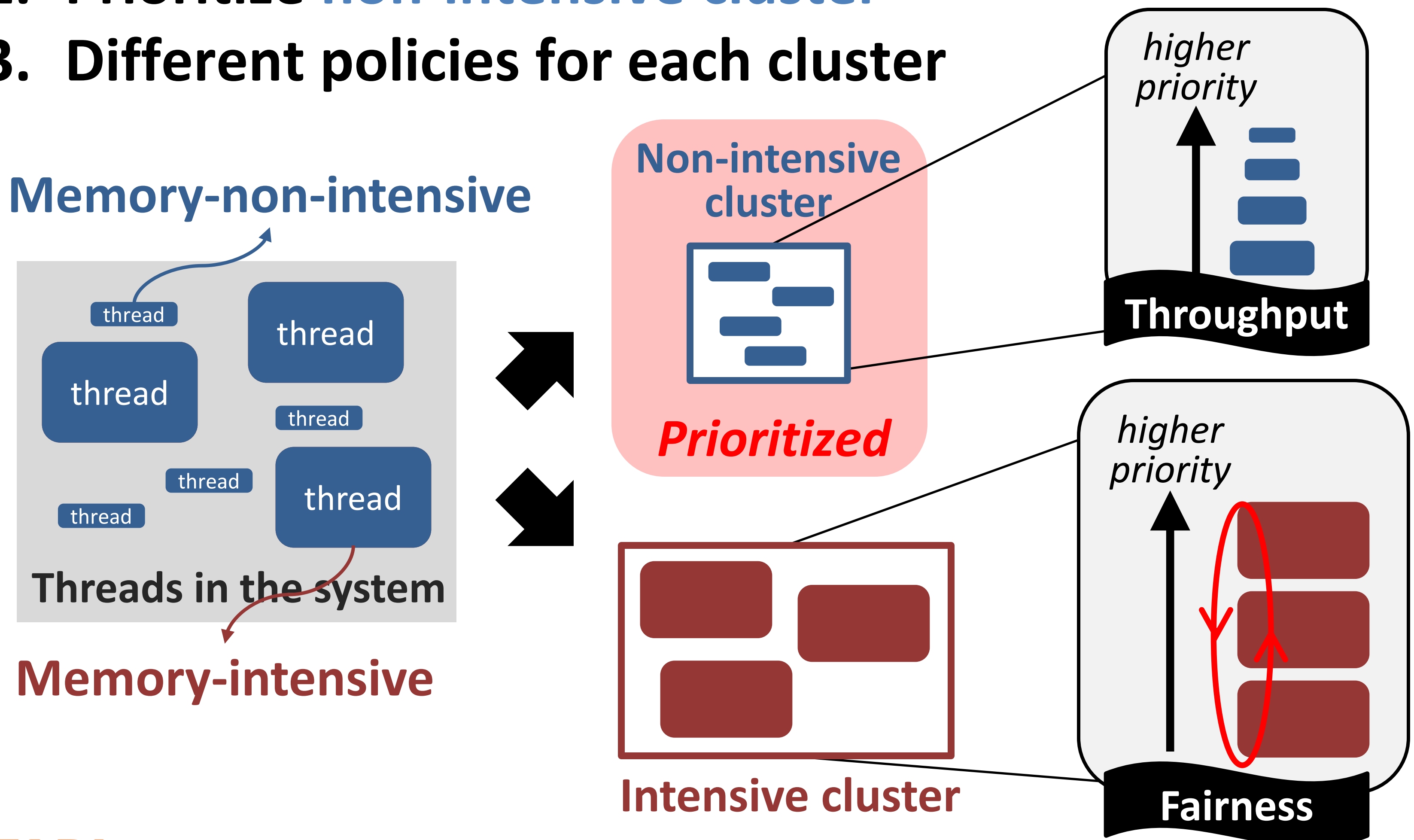
- Shuffle thread ranking

Memory-intensive threads have different vulnerability to interference

- Shuffle asymmetrically

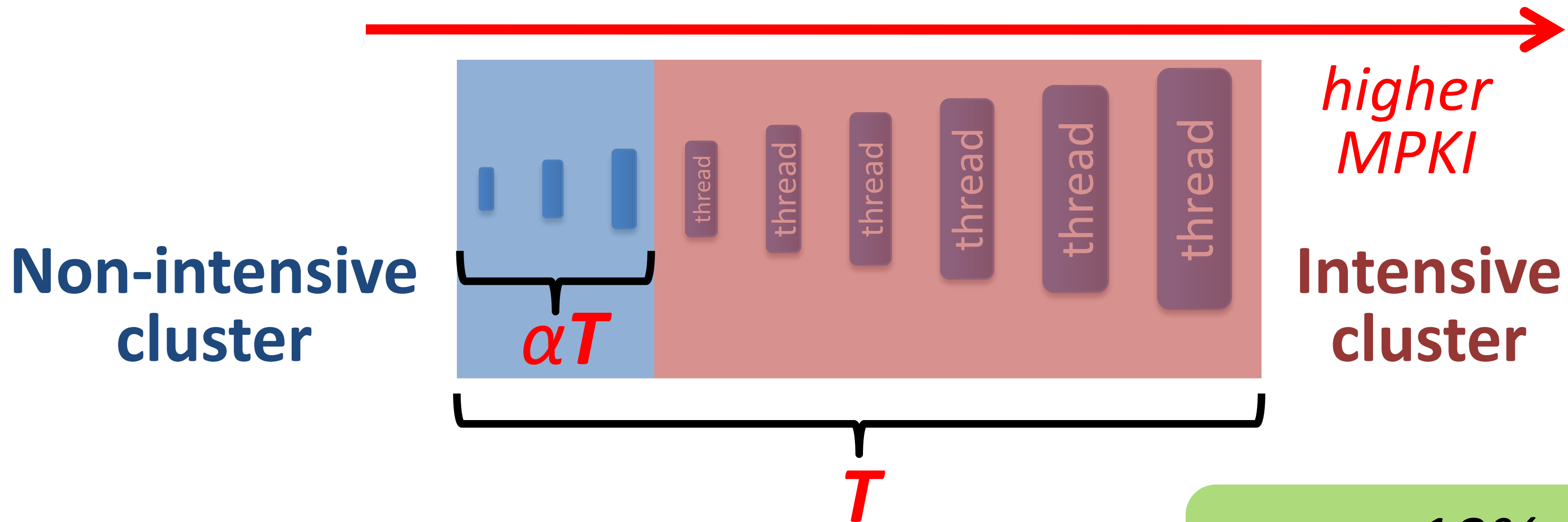
Thread Cluster Memory Scheduling [Kim+ MICRO'10]

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



Clustering Threads

Step1 Sort threads by **MPKI** (misses per kiloinstruction)

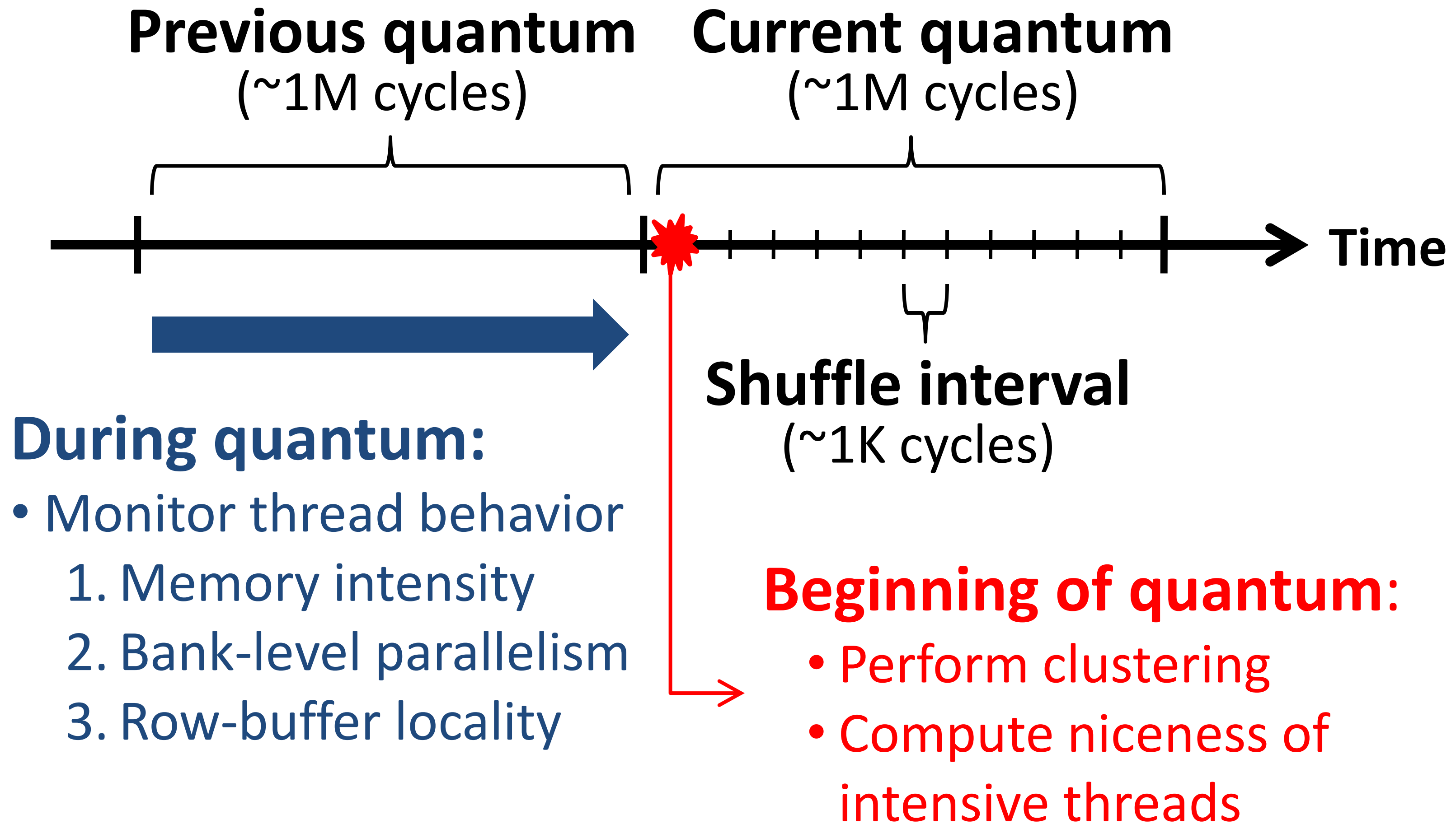


T **=** Total ***memory bandwidth usage***

***α** < 10%*
ClusterThreshold

Step2 Memory bandwidth usage ***αT*** divides clusters

TCM: Quantum-Based Operation



TCM: Scheduling Algorithm

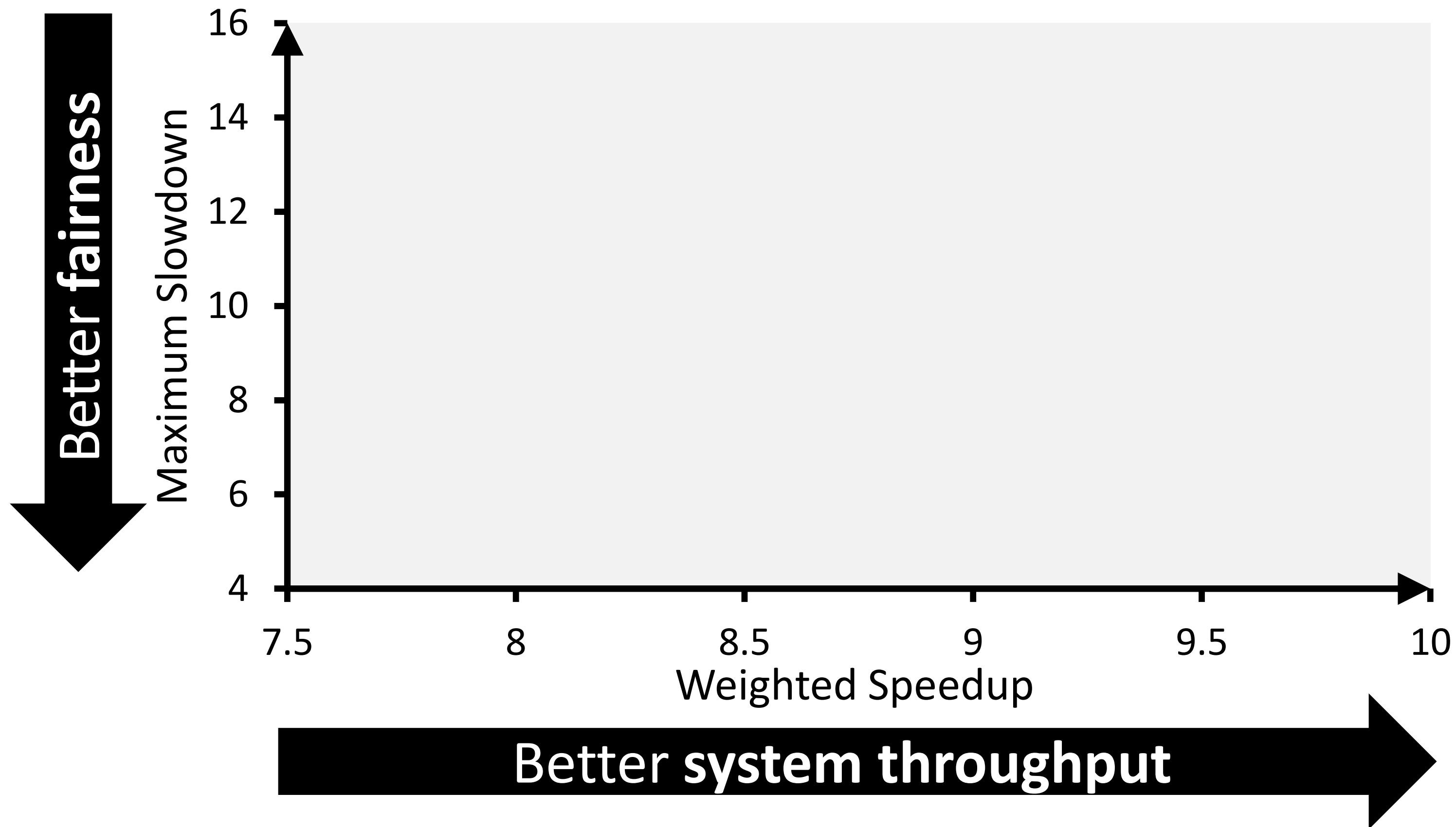
1. ***Highest-rank***: Requests from higher ranked threads prioritized
   - **Non-Intensive** cluster > **Intensive** cluster
   - **Non-Intensive** cluster: lower intensity → higher rank
   - **Intensive** cluster: rank shuffling

2. ***Row-hit***: Row-buffer hit requests are prioritized

3. ***Oldest***: Older requests are prioritized

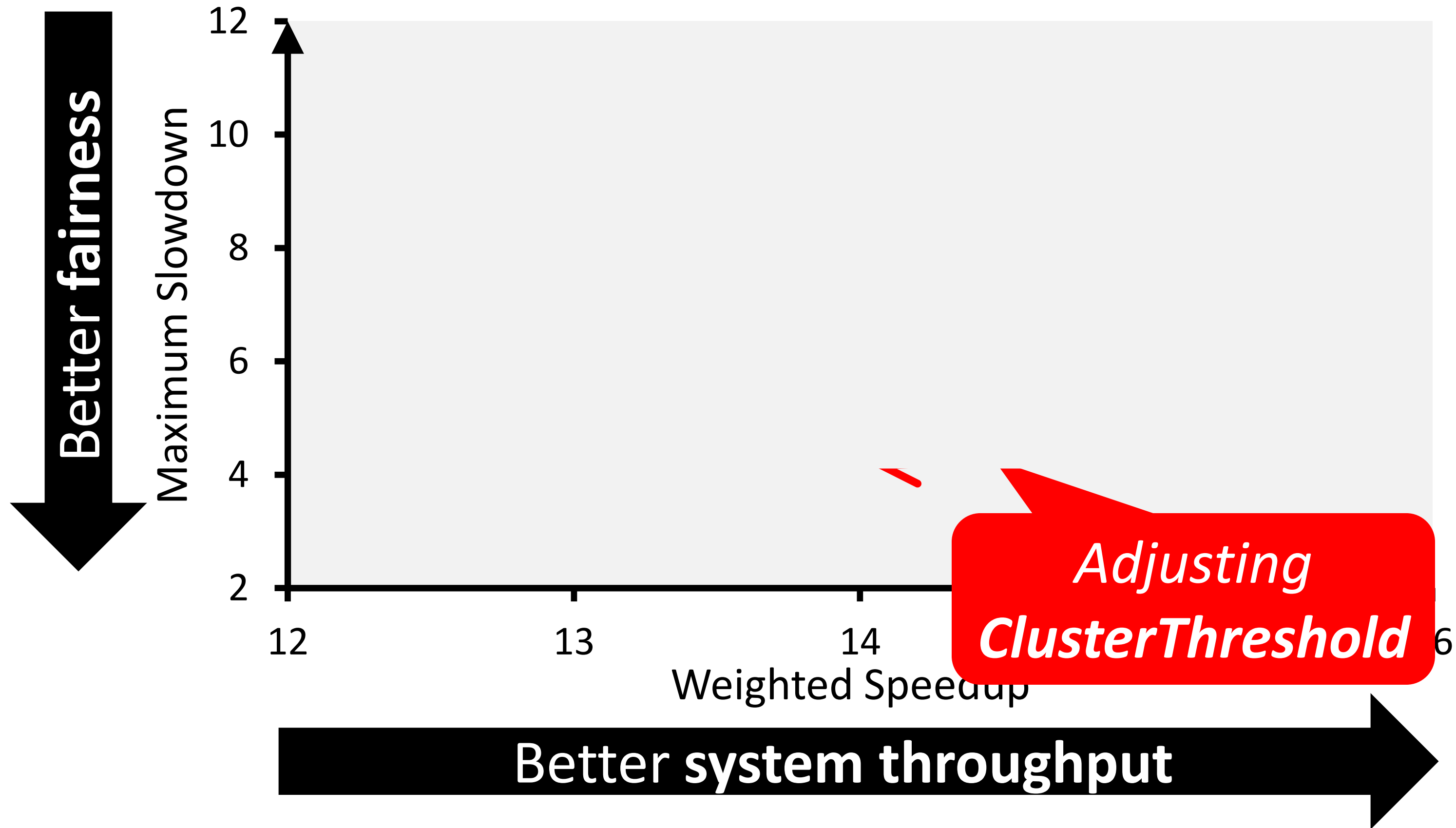
TCM: Throughput and Fairness

24 cores, 4 memory controllers, 96 workloads



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

TCM: Fairness-Throughput Tradeoff



TCM allows robust fairness-throughput tradeoff

TCM Pros and Cons

- Upsides:
  - Provides both high fairness and high performance
  - Caters to the needs for different types of threads (latency vs. bandwidth sensitive)
  - (Relatively) simple

- Downsides:
  - Scalability to large buffer sizes?
  - Robustness of clustering and shuffling algorithms?

Other Ways of Handling Memory Interference

Fundamental Interference Control Techniques

- Goal: to reduce/control interference

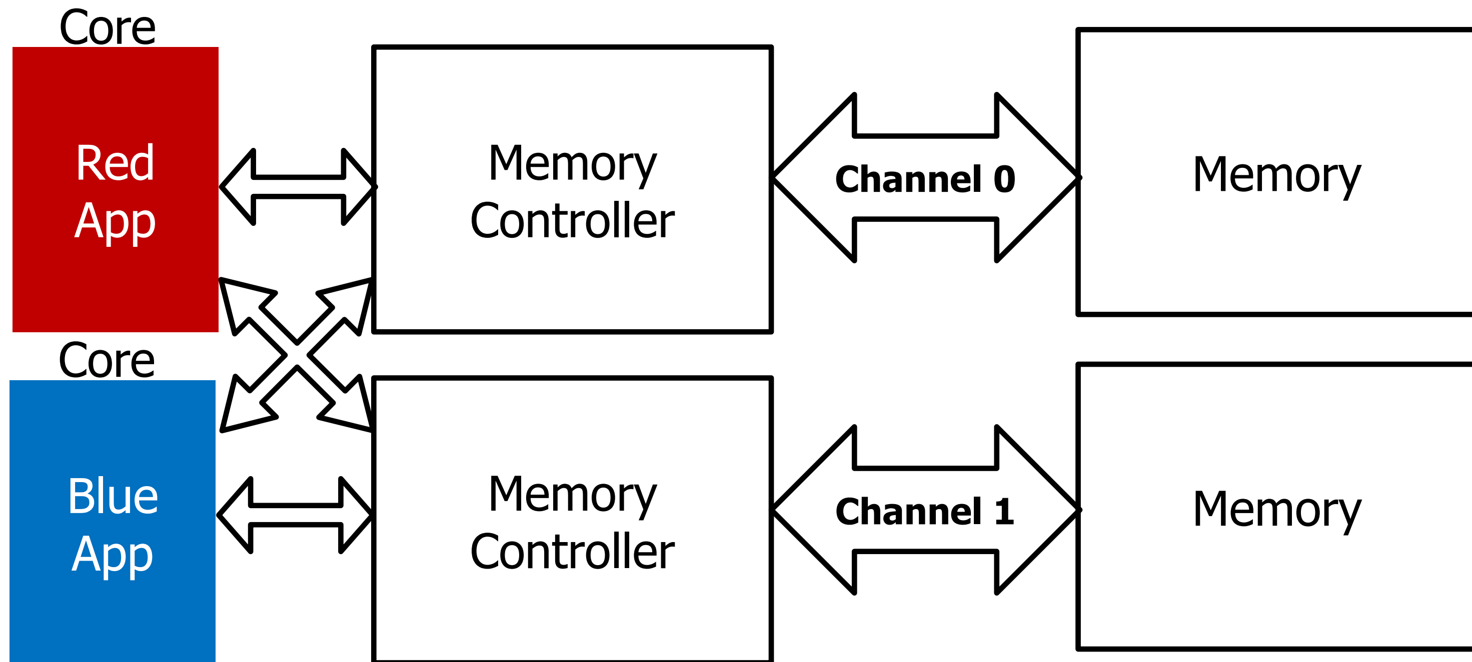
1. Prioritization or request scheduling

2. Data mapping to banks/channels/ranks

3. Core/source throttling

4. Application/thread scheduling

Observation: Modern Systems Have Multiple Channels

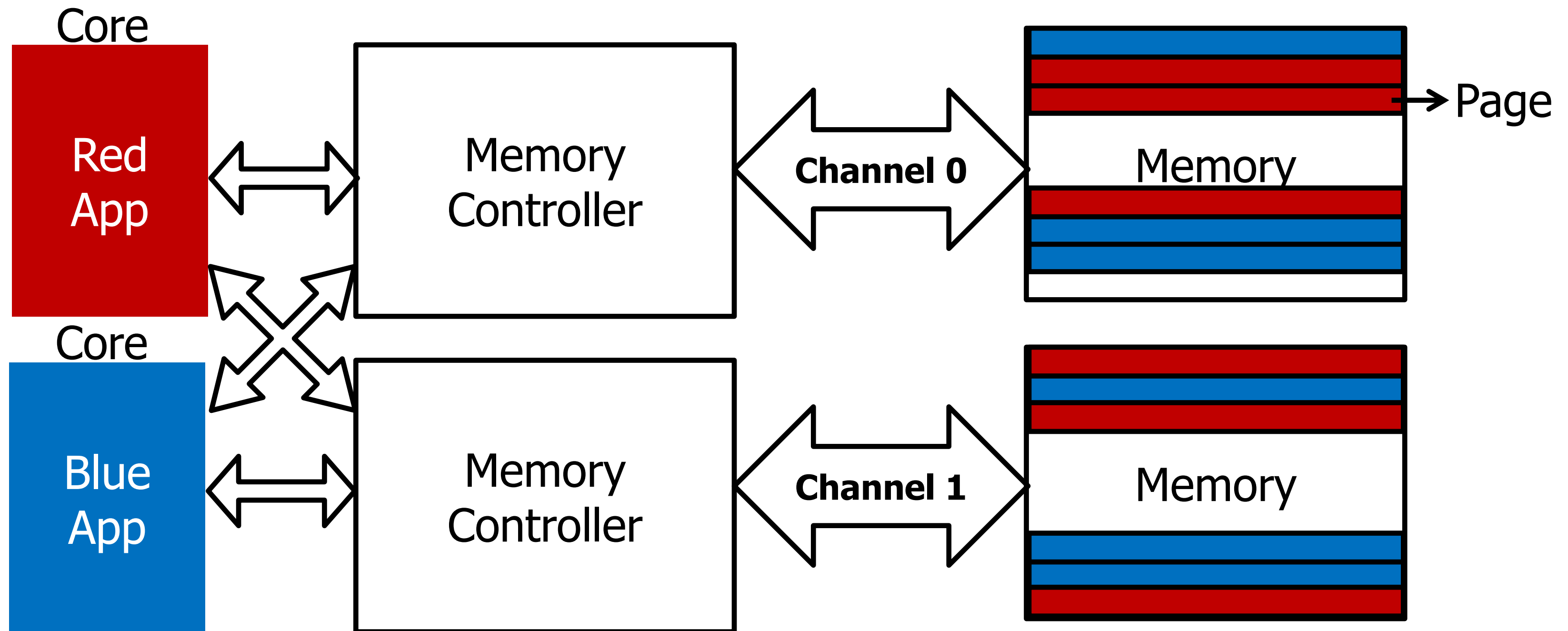


A new degree of freedom

Mapping data across multiple channels

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

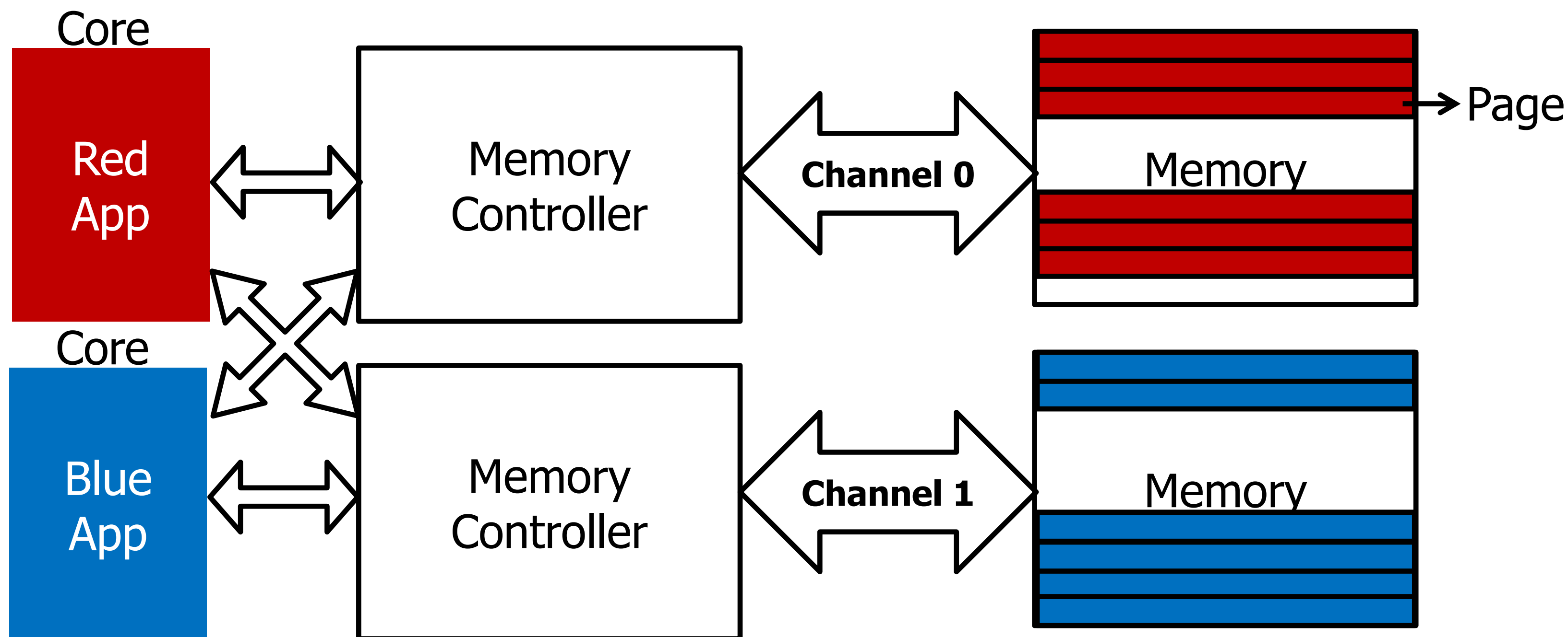
Data Mapping in Current Systems



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.

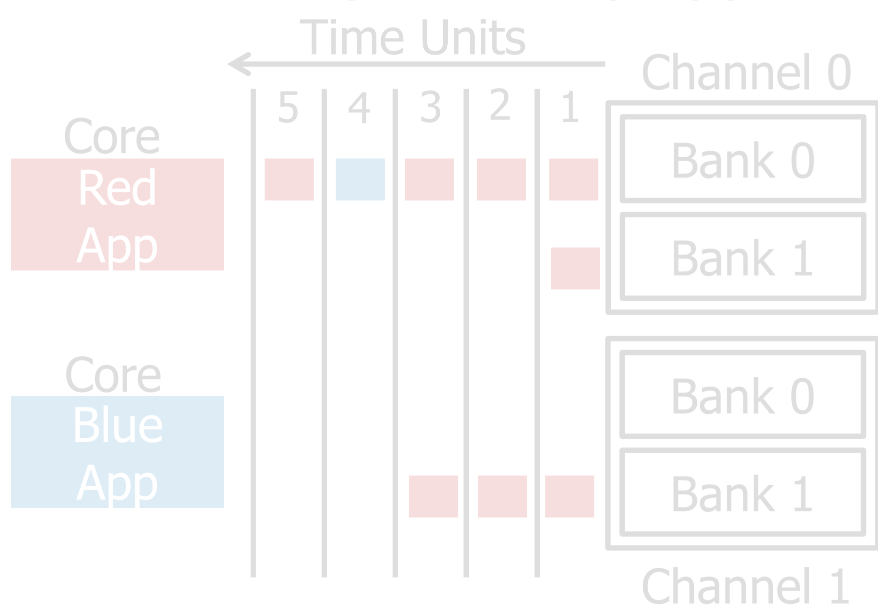
Overview: Memory Channel Partitioning (MCP)

- 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

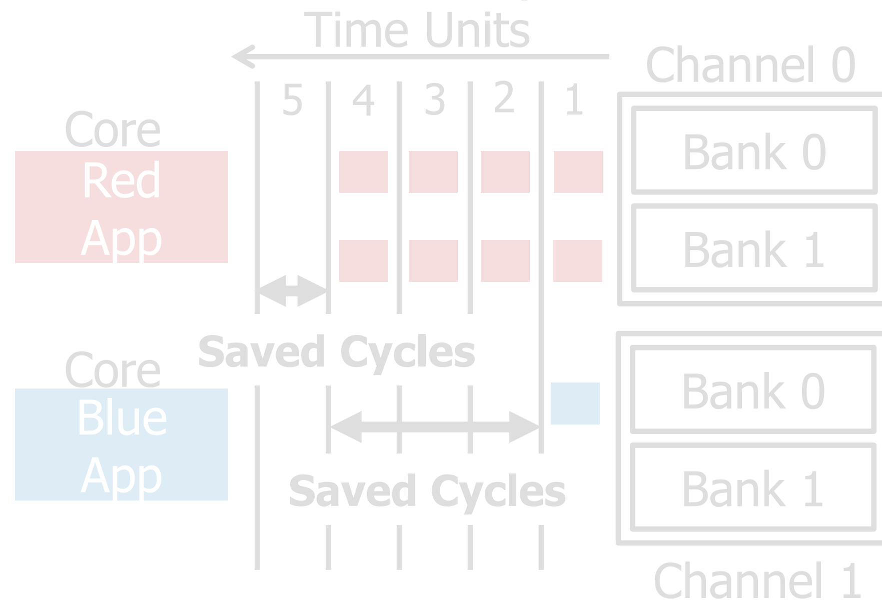
Muralidhara et al., "Memory Channel Partitioning," MICRO'11.

Key Insight 1: Separate by Memory Intensity

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



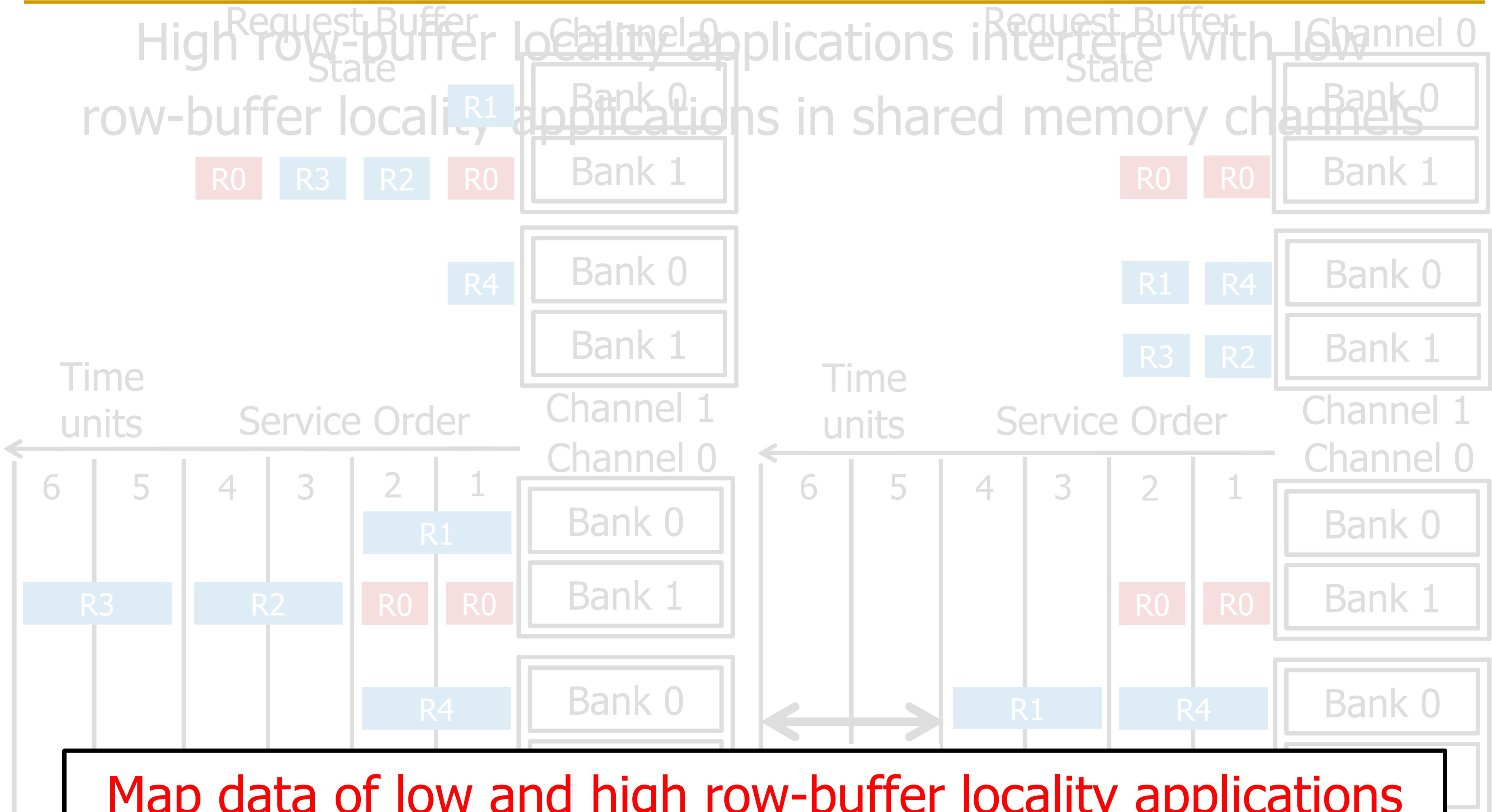
Conventional Page Mapping



Channel Partitioning

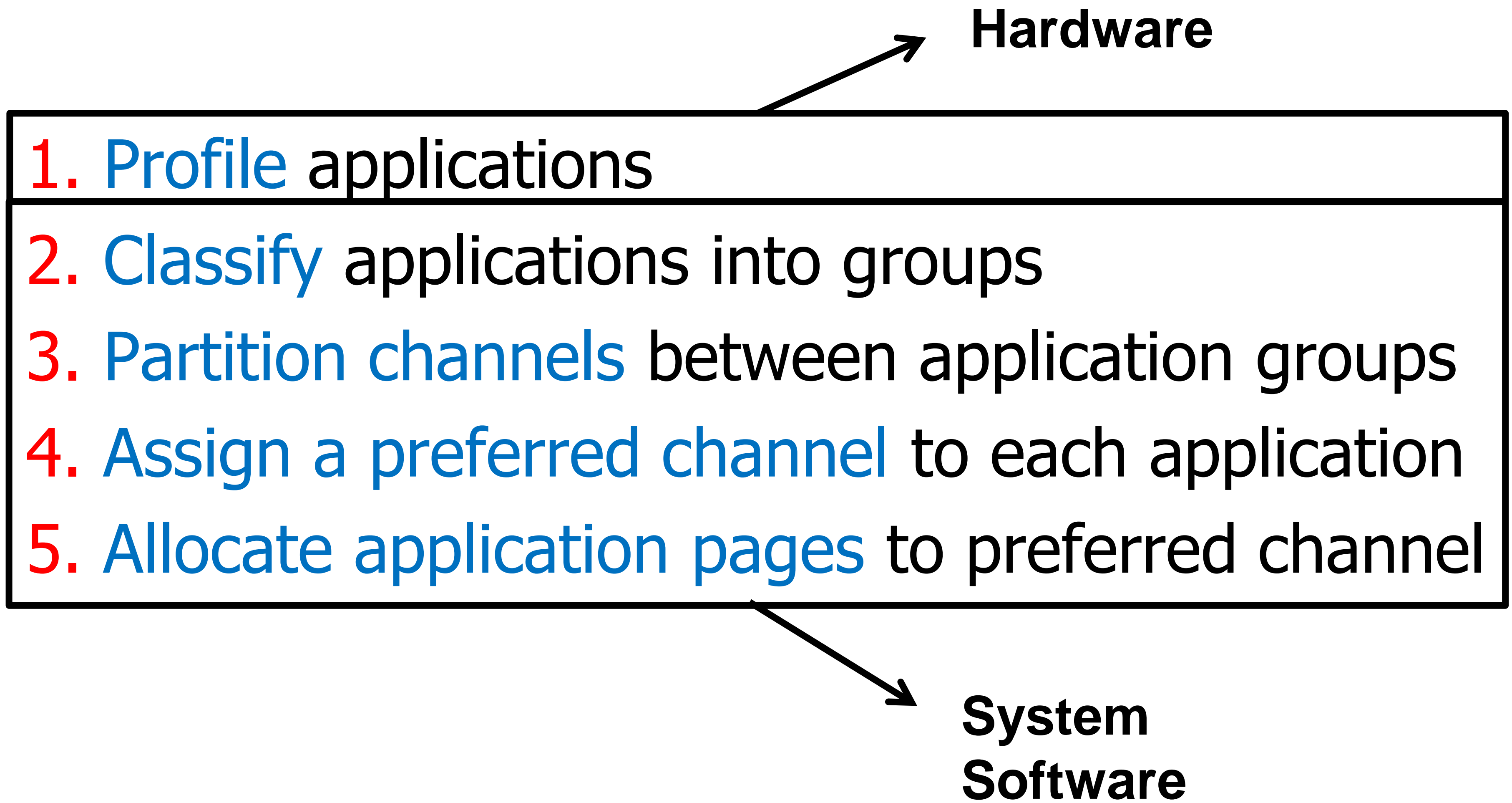
Map data of low and high memory-intensity applications to different channels

Key Insight 2: Separate by Row-Buffer Locality



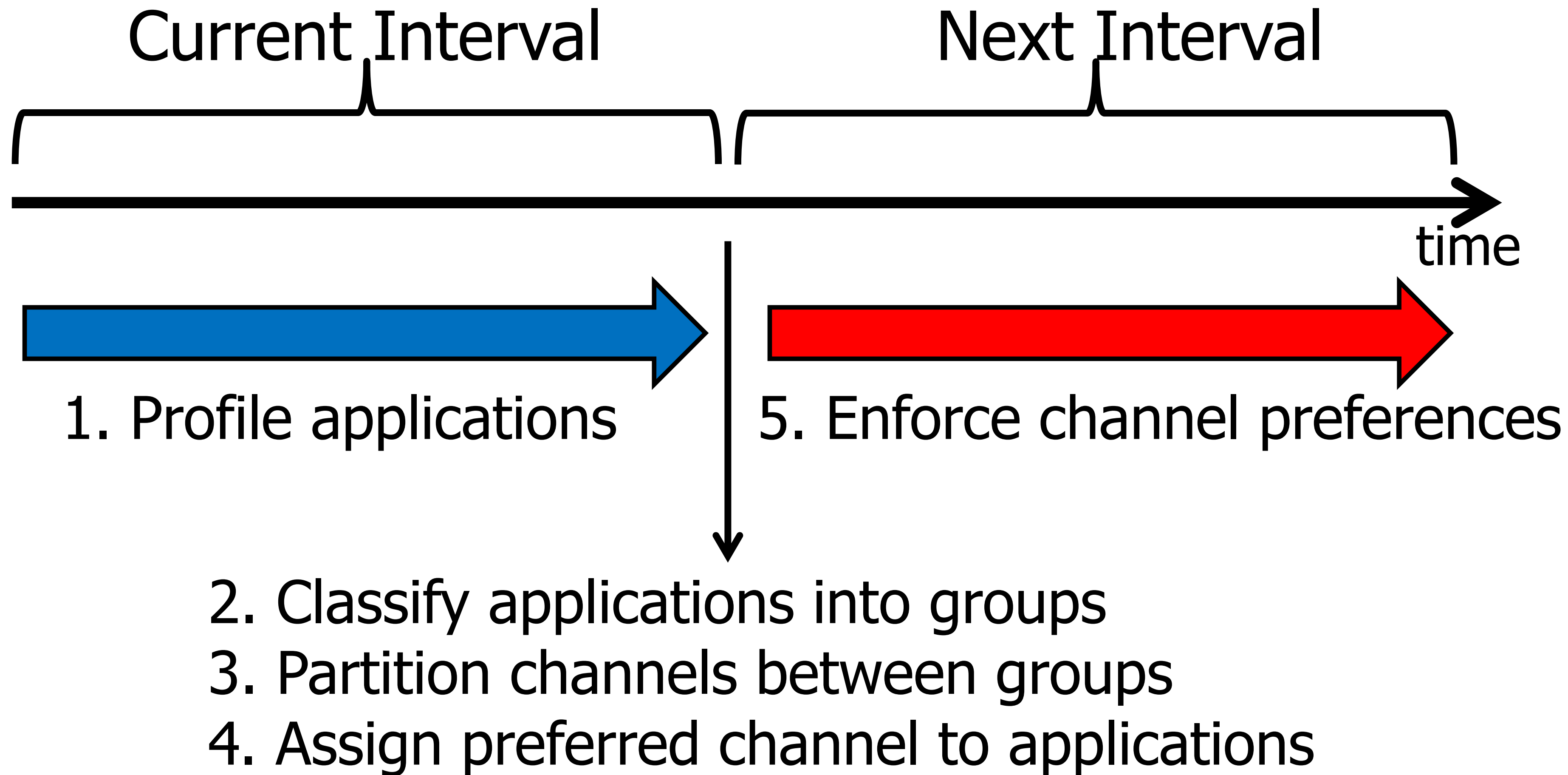
Map data of low and high row-buffer locality applications to different channels

Memory Channel Partitioning (MCP) Mechanism



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

Interval Based Operation



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.

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

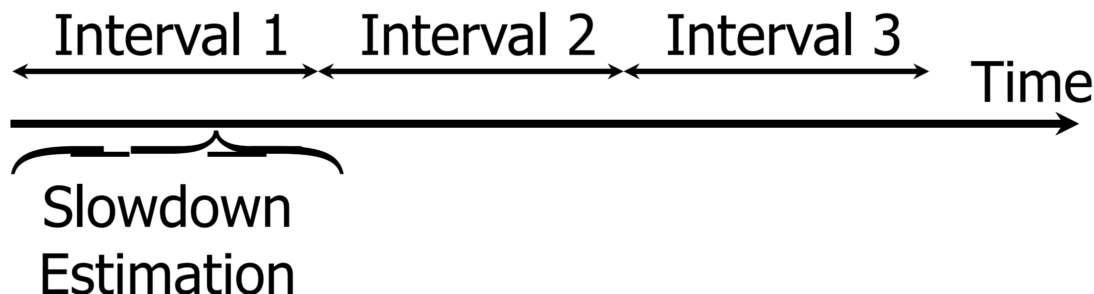
An Alternative Approach: Source Throttling

- 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) [ASPLOS'10]



FST

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
}
```

Core (Source) Throttling

- Idea: Estimate the slowdown due to (DRAM) interference and throttle down threads that slow down others
  - Ebrahimi et al., "Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems," ASPLOS 2010.

- 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

- Disadvantages

  - Requires interference/slowdown estimations

  - Thresholds can become difficult to optimize → throughput loss

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

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 that is falling behind the most in a *parallel for* loop

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?

More on DRAM Controllers

DRAM Power Management

- DRAM chips have power modes
- Idea: When not accessing a chip power it down

- Power states
  - Active (highest power)
  - All banks idle
  - Power-down
  - Self-refresh (lowest power)

- State transitions incur latency during which the chip cannot be accessed

Why are DRAM Controllers Difficult to Design?

- Need to obey DRAM timing constraints for correctness
  - There are many (50+) timing constraints in DRAM
  - tWTR: Minimum number of cycles to wait before issuing a read command after a write command is issued
  - tRC: Minimum number of cycles between the issuing of two consecutive activate commands to the same bank
  - ...
- Need to keep track of many resources to prevent conflicts
  - Channels, banks, ranks, data bus, address bus, row buffers
- Need to handle DRAM refresh
- Need to manage power consumption
- Need to optimize for performance (in the presence of constraints)
  - Reordering is not simple
  - Fairness and QoS needs complicates the scheduling problem

Many DRAM Timing Constraints

Latency	Symbol	DRAM cycles	Latency	Symbol	DRAM cycles
Precharge	^tRP	11	Activate to read/write	^tRCD	11
Read column address strobe	CL	11	Write column address strobe	CWL	8
Additive	AL	0	Activate to activate	^tRC	39
Activate to precharge	^tRAS	28	Read to precharge	^tRTP	6
Burst length	^tBL	4	Column address strobe to column address strobe	^tCCD	4
Activate to activate (different bank)	^tRRD	6	Four activate windows	^tFAW	24
Write to read	^tWTR	6	Write recovery	^tWR	12

Table 4. DDR3 1600 DRAM timing specifications

- From Lee et al., "DRAM-Aware Last-Level Cache Writeback: Reducing Write-Caused Interference in Memory Systems," HPS Technical Report, April 2010.

More on DRAM Operation

- Kim et al., "A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM," ISCA 2012.
- Lee et al., "Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture," HPCA 2013.

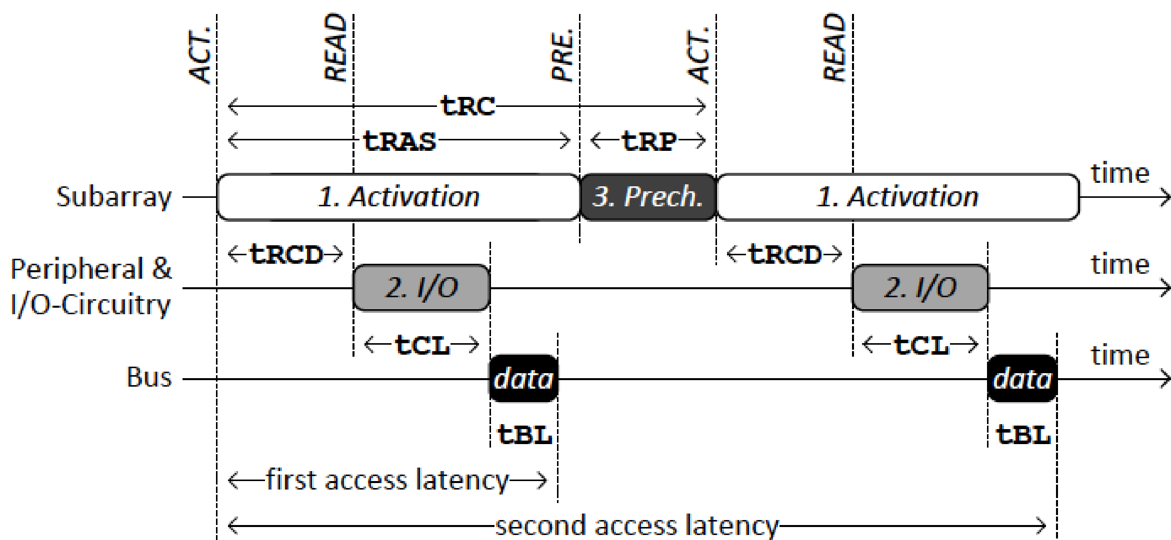


Figure 5. Three Phases of DRAM Access

Table 2. Timing Constraints (DDR3-1066) [43]

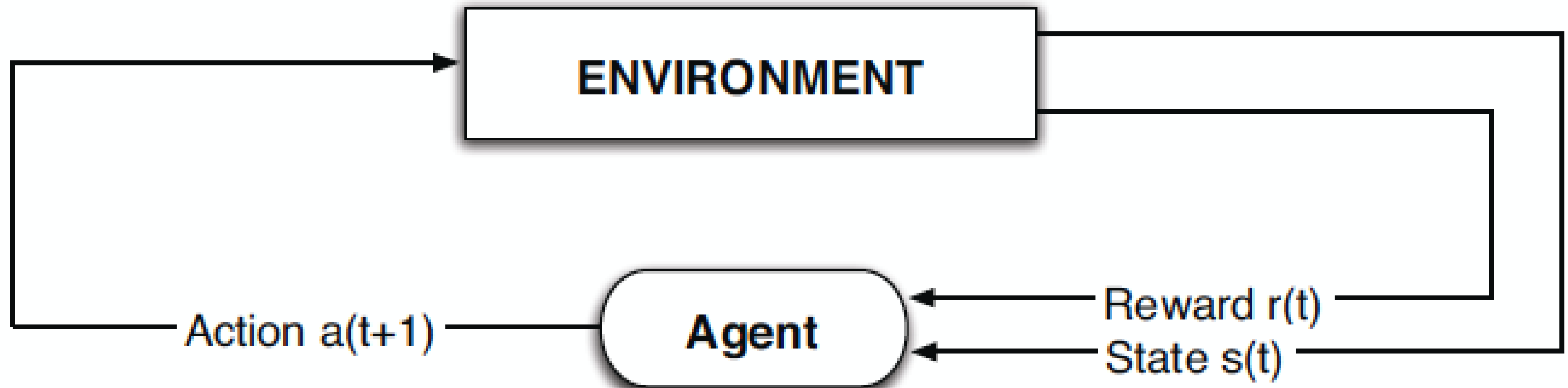
Phase	Commands	Name	Value
1	ACT → READ ACT → WRITE	tRCD	15ns
	ACT → PRE	tRAS	37.5ns
2	READ → *data* WRITE → *data*	tCL tCWL	15ns 11.25ns
	data burst	tBL	7.5ns
3	PRE → ACT	tRP	15ns
1 & 3	ACT → ACT	tRC (tRAS+tRP)	52.5ns

Self-Optimizing DRAM Controllers

- Problem: DRAM controllers difficult to design → It is difficult for human designers to design a policy that can adapt itself very well to different workloads and different system conditions
- Idea: Design a memory controller that adapts its scheduling policy decisions to workload behavior and system conditions using machine learning.
- Observation: Reinforcement learning maps nicely to memory control.
- Design: Memory controller is a reinforcement learning agent that dynamically and continuously learns and employs the best scheduling policy.

Ipek+, "Self Optimizing Memory Controllers: A Reinforcement Learning Approach," ISCA 2008.

Self-Optimizing DRAM Controllers



Goal: Learn to choose actions to maximize $r_0 + \gamma r_1 + \gamma^2 r_2 + \ldots \ (0 \leq \gamma < 1)$

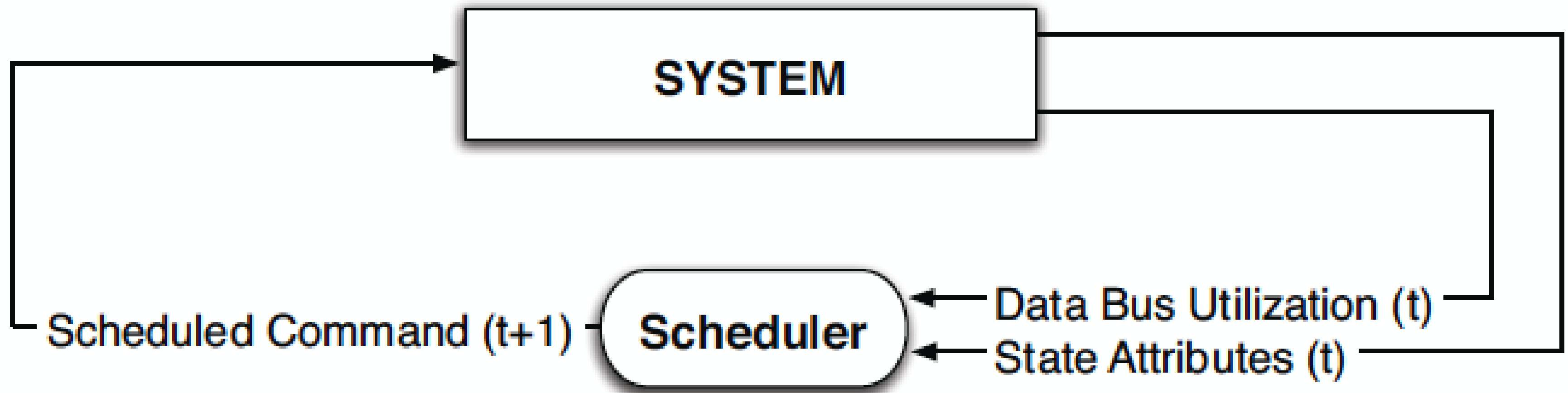
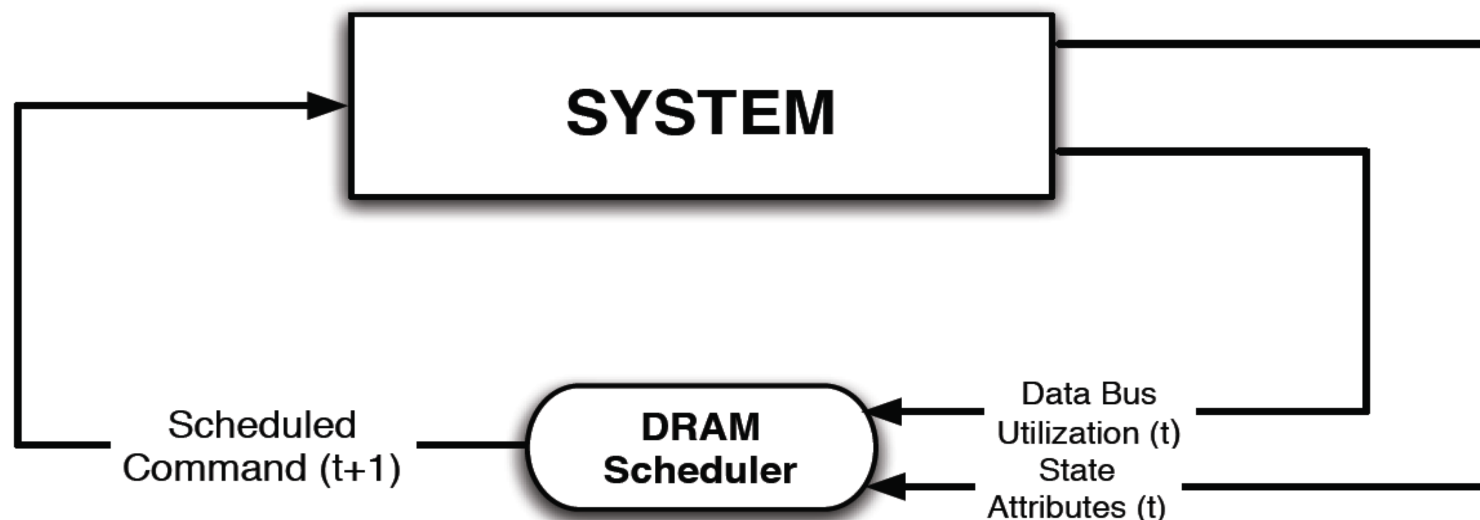


Figure 2: (a) Intelligent agent based on reinforcement learning principles; (b) DRAM scheduler as an RL-agent

Self-Optimizing DRAM Controllers

- Dynamically adapt the memory scheduling policy via interaction with the system at runtime
  - Associate system states and actions (commands) with long term reward values
  - Schedule command with highest estimated long-term value in each state
  - Continuously update state-action values based on feedback from system



Self-Optimizing DRAM Controllers

- Engin Ipek, Onur Mutlu, José F. Martínez, and Rich Caruana,
"Self Optimizing Memory Controllers: A Reinforcement Learning Approach"
*Proceedings of the 35th International Symposium on Computer Architecture (**ISCA**)*, pages 39-50, Beijing, China, June 2008.

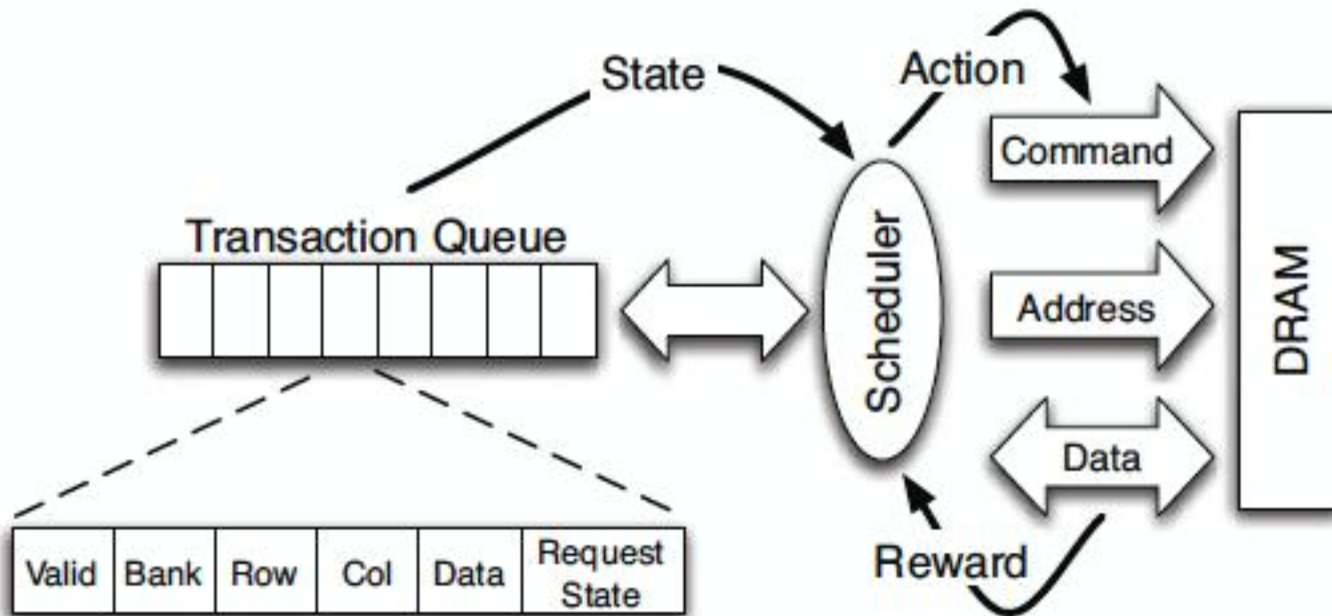


Figure 4: High-level overview of an RL-based scheduler.

States, Actions, Rewards

- Reward function
  - +1 for scheduling Read and Write commands
  - 0 at all other times

  Goal is to maximize data bus utilization

- State attributes
  - Number of reads, writes, and load misses in transaction queue
  - Number of pending writes and ROB heads waiting for referenced row
  - Request's relative ROB order

- Actions
  - Activate
  - Write
  - Read - load miss
  - Read - store miss
  - Precharge - pending
  - Precharge - preemptive
  - NOP

Performance Results

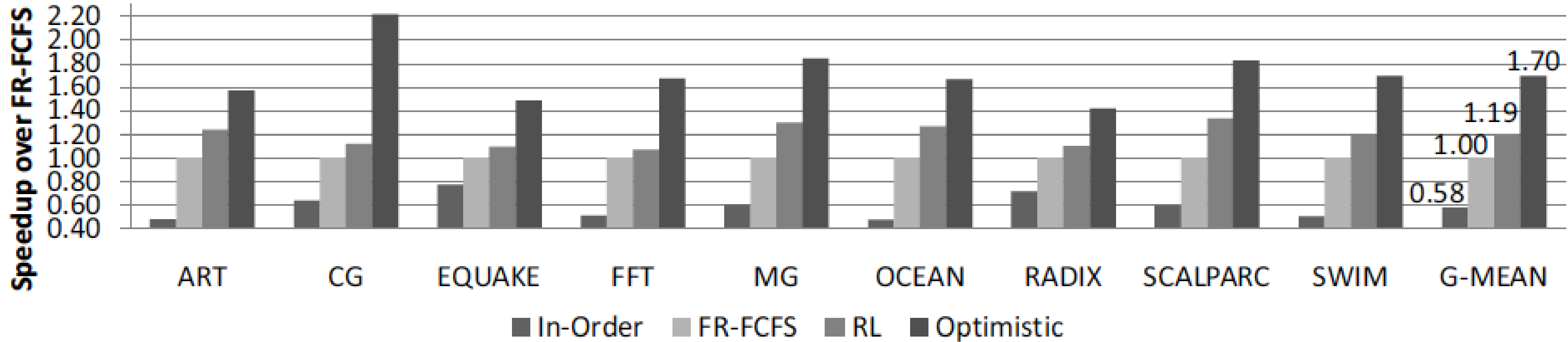


Figure 7: Performance comparison of in-order, FR-FCFS, RL-based, and optimistic memory controllers

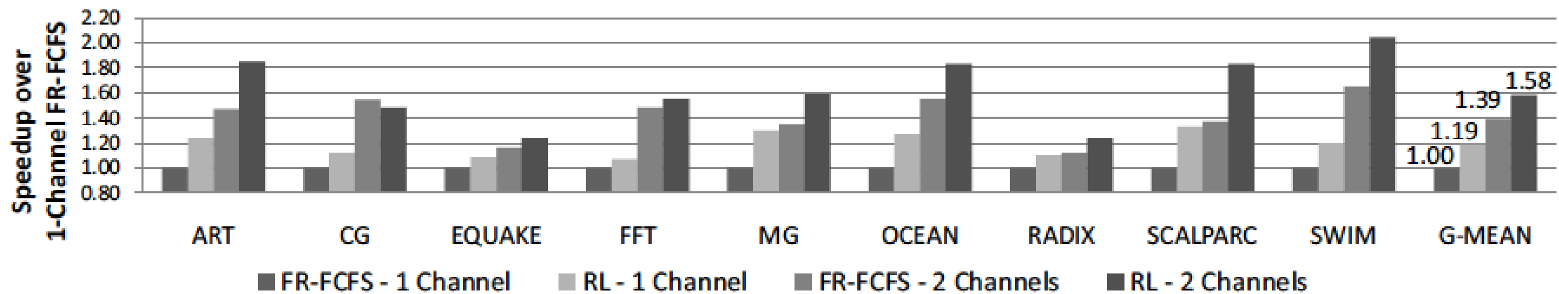


Figure 15: Performance comparison of FR-FCFS and RL-based memory controllers on systems with 6.4GB/s and 12.8GB/s peak DRAM bandwidth

Self Optimizing DRAM Controllers

- Advantages

  + Adapts the scheduling policy dynamically to changing workload behavior and to maximize a long-term target

  + Reduces the designer's burden in finding a good scheduling policy. Designer specifies:

    1) What system variables might be useful

    2) What target to optimize, but not how to optimize it

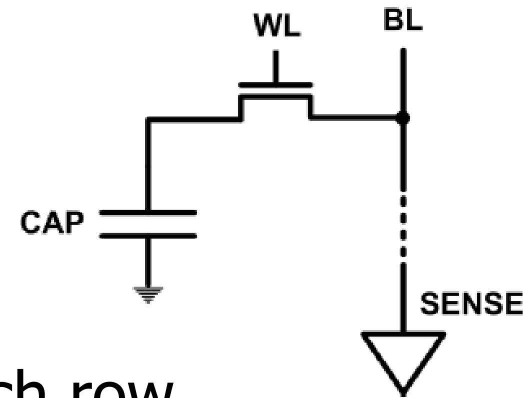
- Disadvantages

  -- Black box: designer much less likely to implement what she cannot easily reason about

  -- How to specify different reward functions that can achieve different objectives? (e.g., fairness, QoS)

DRAM Refresh

DRAM Refresh



- DRAM capacitor charge leaks over time
- The memory controller needs to refresh each row periodically to restore charge
  - Read and close each row every N ms
  - Typical N = 64 ms
- Downsides of refresh

  -- Energy consumption: Each refresh consumes energy

  -- Performance degradation: DRAM rank/bank unavailable while refreshed

  -- QoS/predictability impact: (Long) pause times during refresh

  -- Refresh rate limits DRAM capacity scaling

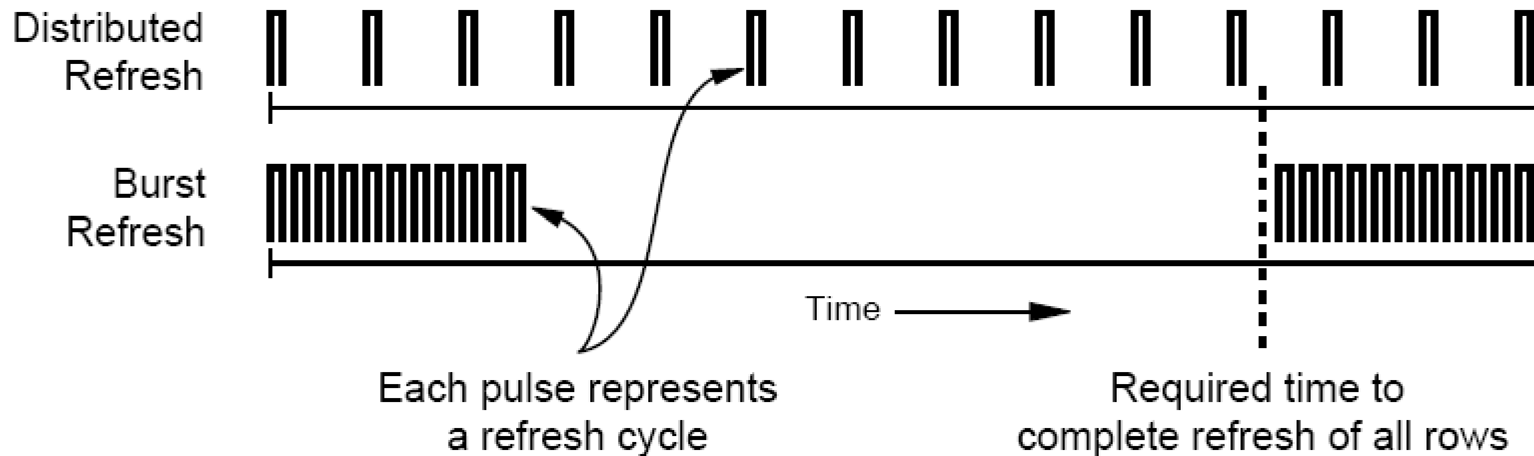
DRAM Refresh: Performance

- Implications of refresh on performance
  -- DRAM bank unavailable while refreshed
  -- Long pause times: If we refresh all rows in burst, every 64ms the DRAM will be unavailable until refresh ends

- Burst refresh: All rows refreshed immediately after one another

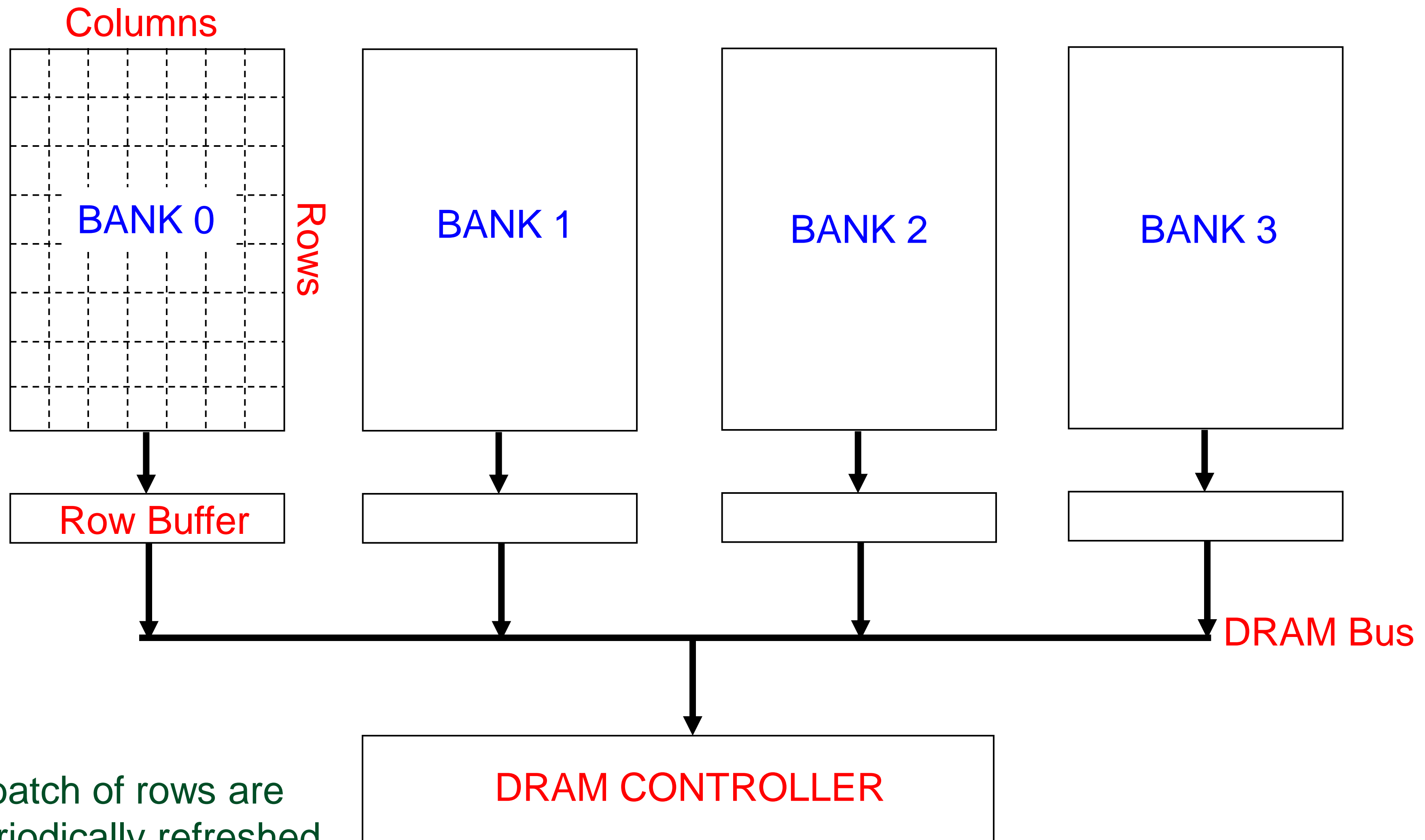
- Distributed refresh: Each row refreshed at a different time, at regular intervals

Distributed Refresh



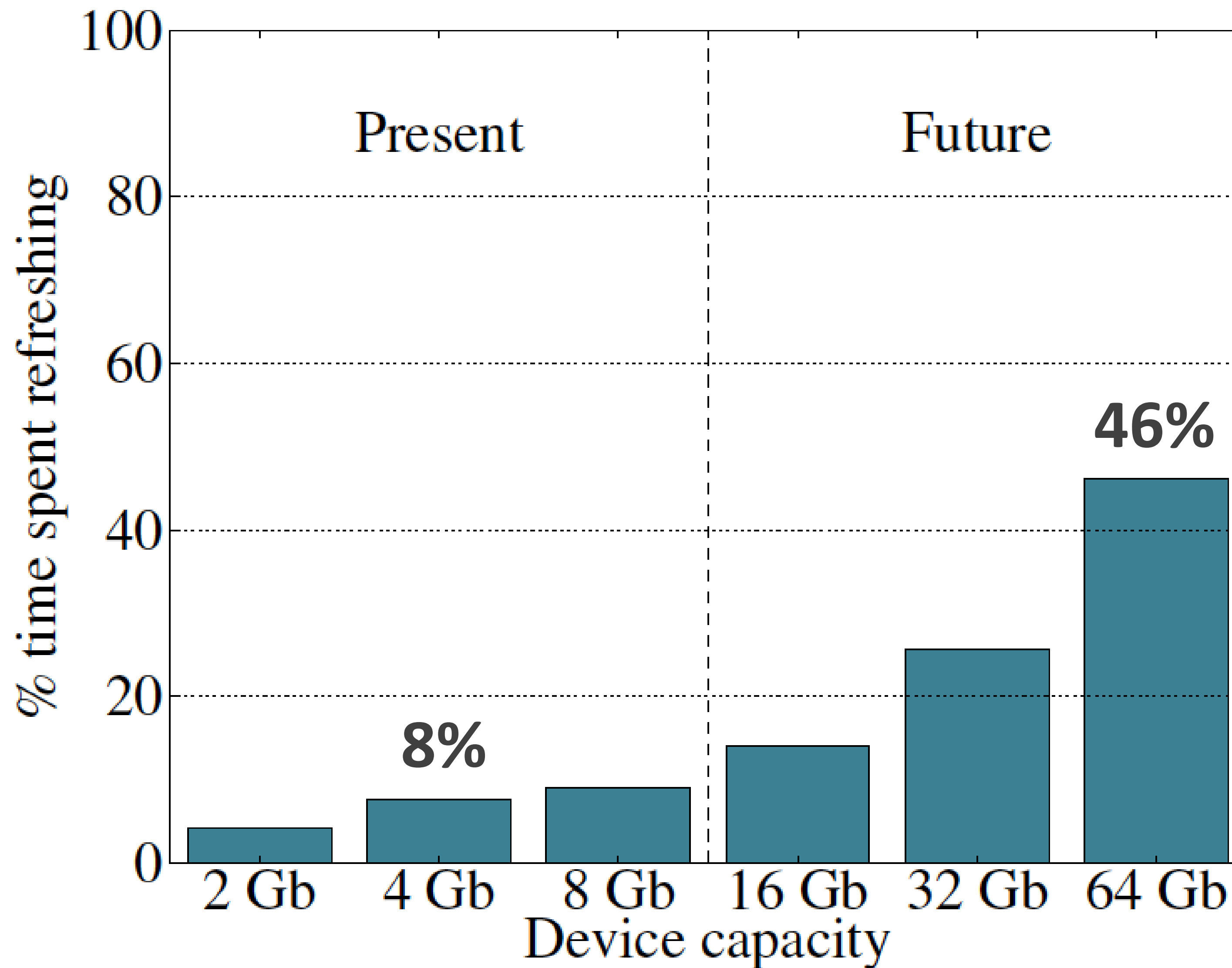
- Distributed refresh eliminates long pause times
- How else can we reduce the effect of refresh on performance/QoS?
- Does distributed refresh reduce refresh impact on energy?
- Can we reduce the number of refreshes?

Refresh Today: Auto Refresh



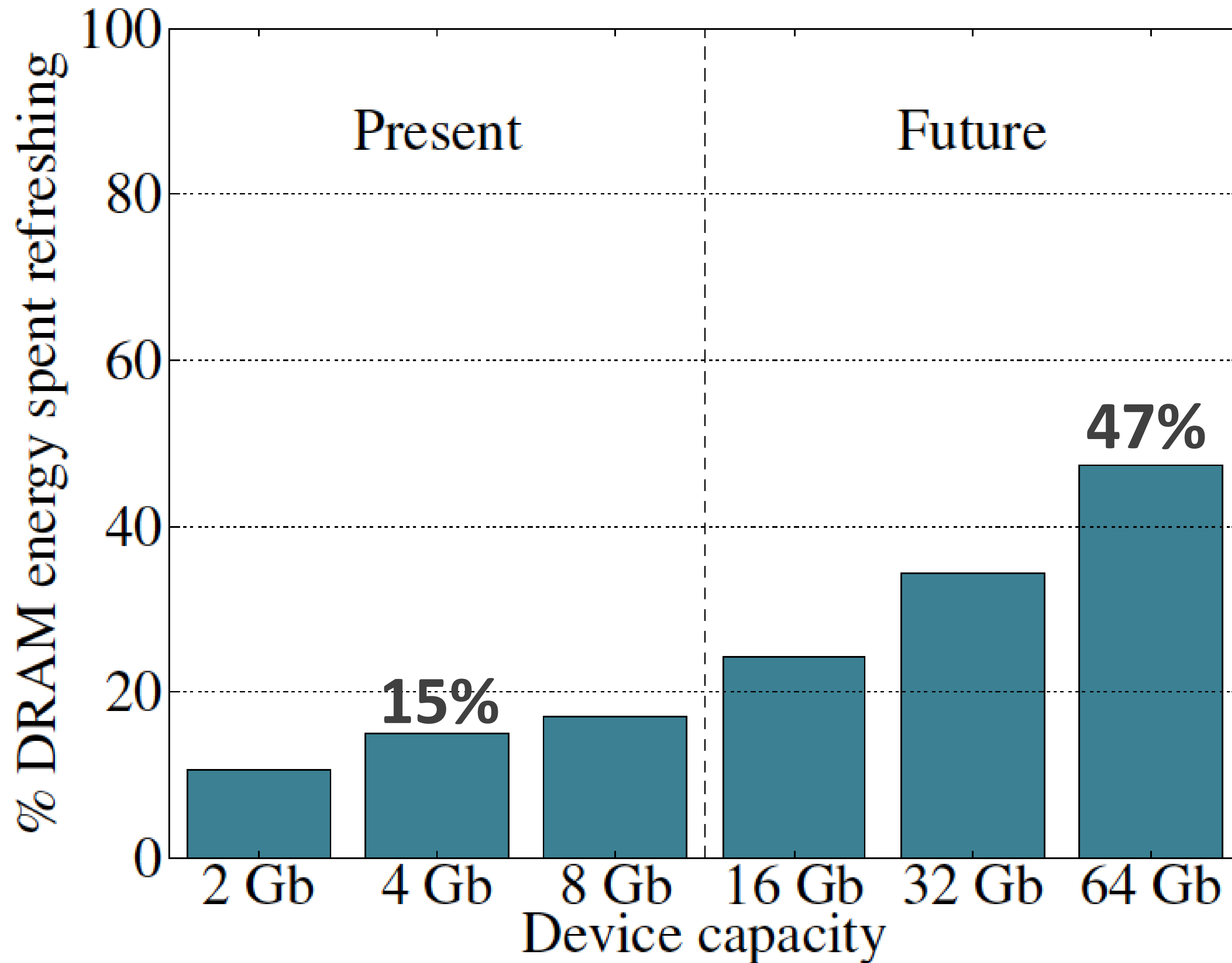
A batch of rows are periodically refreshed via the auto-refresh command

Refresh Overhead: Performance



Liu et al., "RAIDR: Retention-Aware Intelligent DRAM Refresh," ISCA 2012.

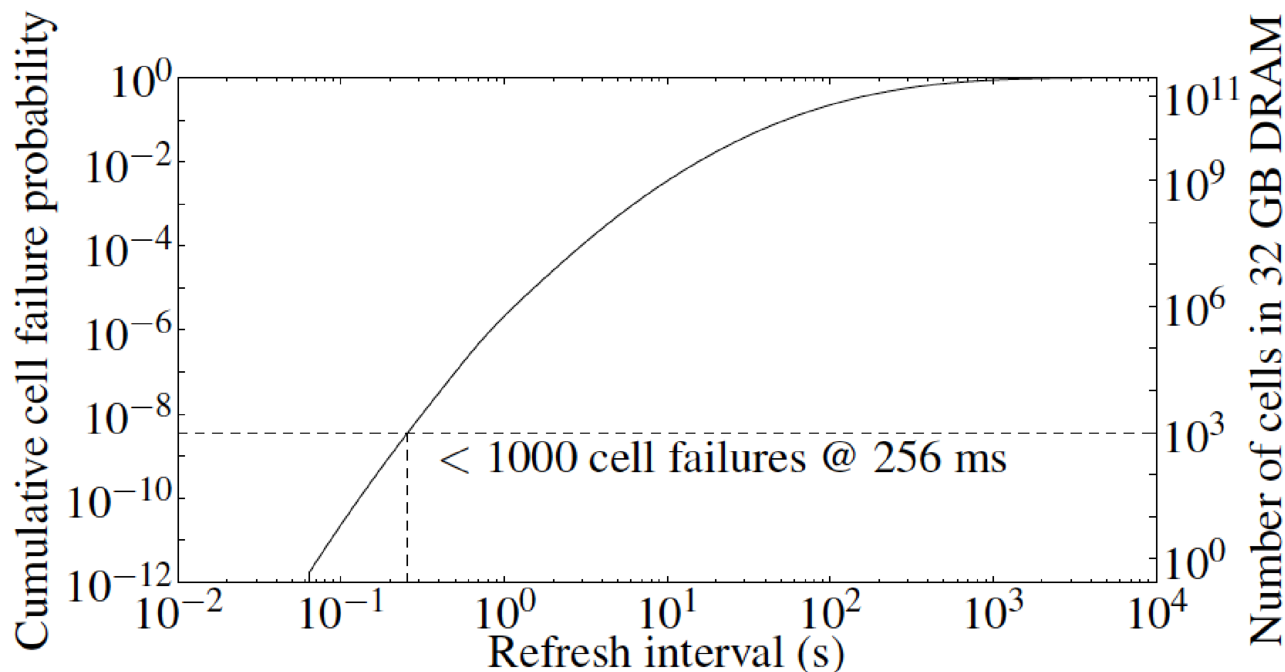
Refresh Overhead: Energy



Liu et al., "RAIDR: Retention-Aware Intelligent DRAM Refresh," ISCA 2012.

Problem with Conventional Refresh

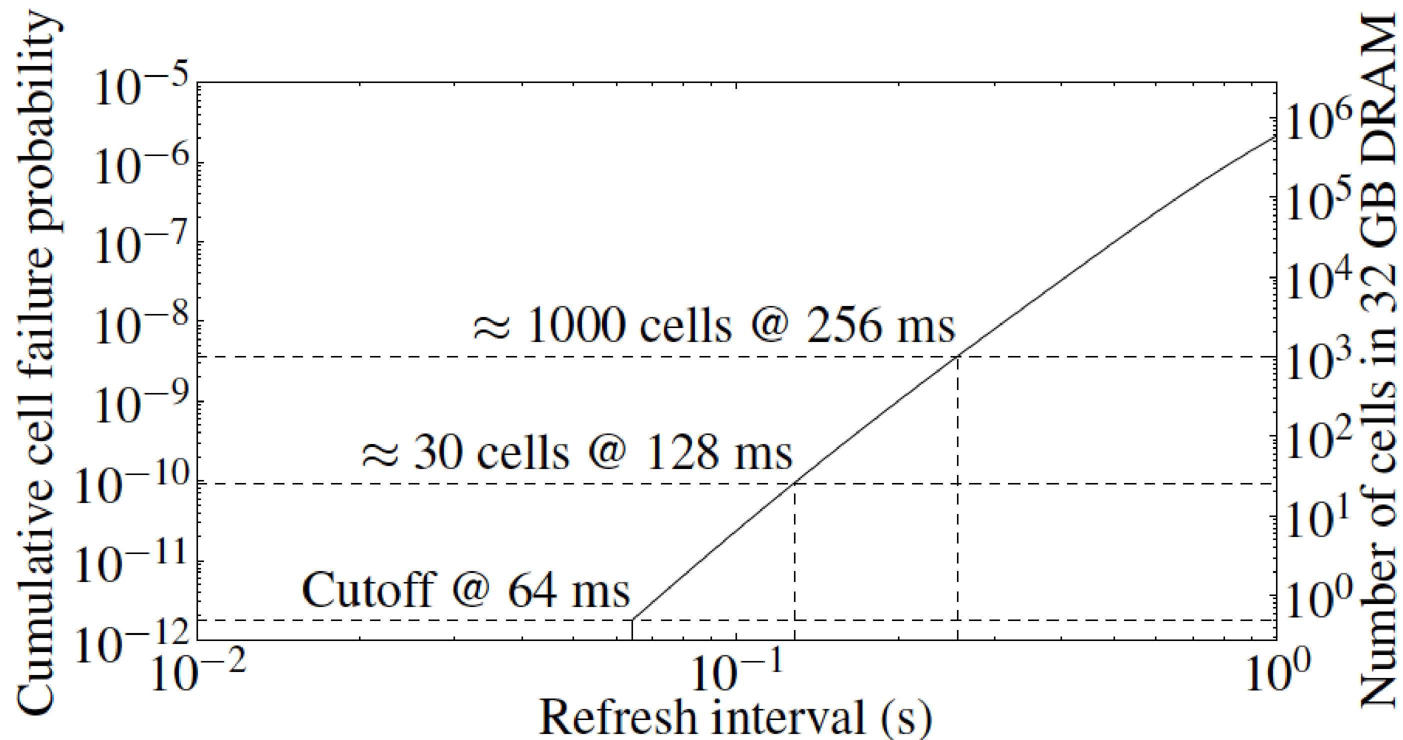
- Today: Every row is refreshed at the same rate



- Observation: Most rows can be refreshed much less often without losing data [Kim+, EDL'09]
- Problem: No support in DRAM for different refresh rates per row

Retention Time of DRAM Rows

- Observation: Only very few rows need to be refreshed at the worst-case rate



- Can we exploit this to reduce refresh operations at low cost?

Reducing DRAM Refresh Operations

- Idea: Identify the retention time of different rows and refresh each row at the frequency it needs to be refreshed

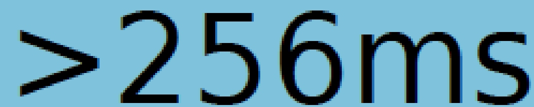
- (Cost-conscious) Idea: Bin the rows according to their minimum retention times and refresh rows in each bin at the refresh rate specified for the bin
  - e.g., a bin for 64-128ms, another for 128-256ms, …

- Observation: Only very few rows need to be refreshed very frequently [64-128ms] → Have only a few bins → Low HW overhead to achieve large reductions in refresh operations

- Liu et al., "RAIDR: Retention-Aware Intelligent DRAM Refresh," ISCA 2012.

RAIDR: Mechanism

64-128ms



1.25KB storage in controller for 32GB DRAM memory

128-256ms

bins at different rates

→ probe Bloom Filters to determine refresh rate of a row

1. Profiling

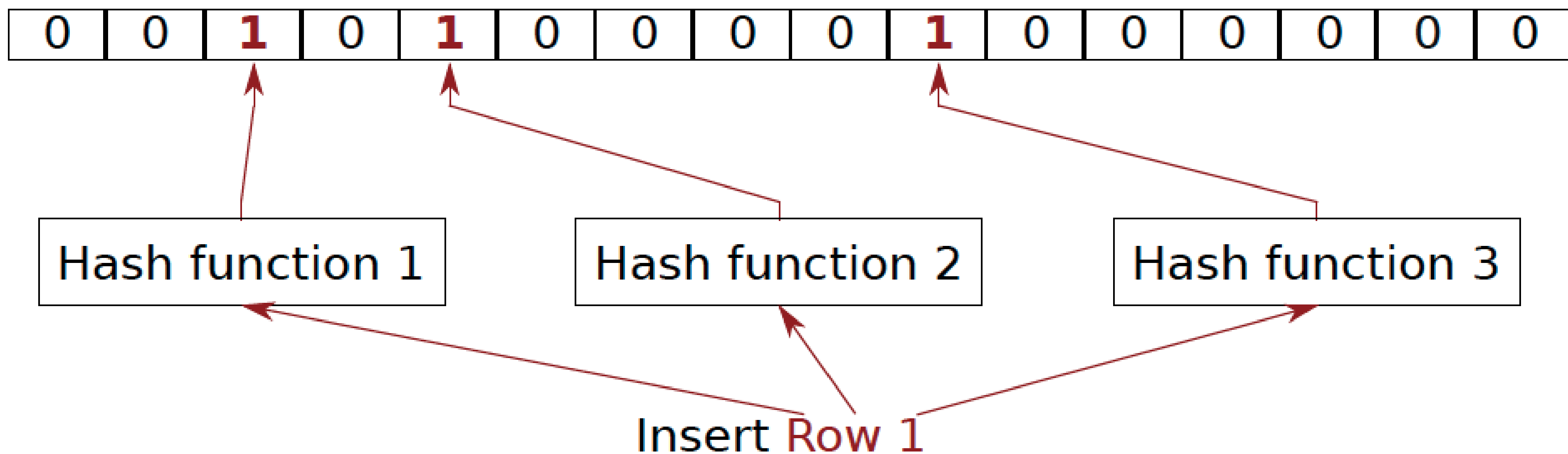
To profile a row:

1. Write data to the row
2. Prevent it from being refreshed
3. Measure time before data corruption

	Row 1	Row 2	Row 3
Initially	11111111...	11111111...	11111111...
After 64 ms	11111111...	11111111...	11111111...
After 128 ms	11011111... (64–128ms)	11111111...	11111111...
After 256 ms		11111011... (128–256ms)	11111111... (>256ms)

2. Binning

- How to efficiently and scalably store rows into retention time bins?
- Use Hardware Bloom Filters [Bloom, CACM 1970]



Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors", CACM 1970.

Bloom Filter

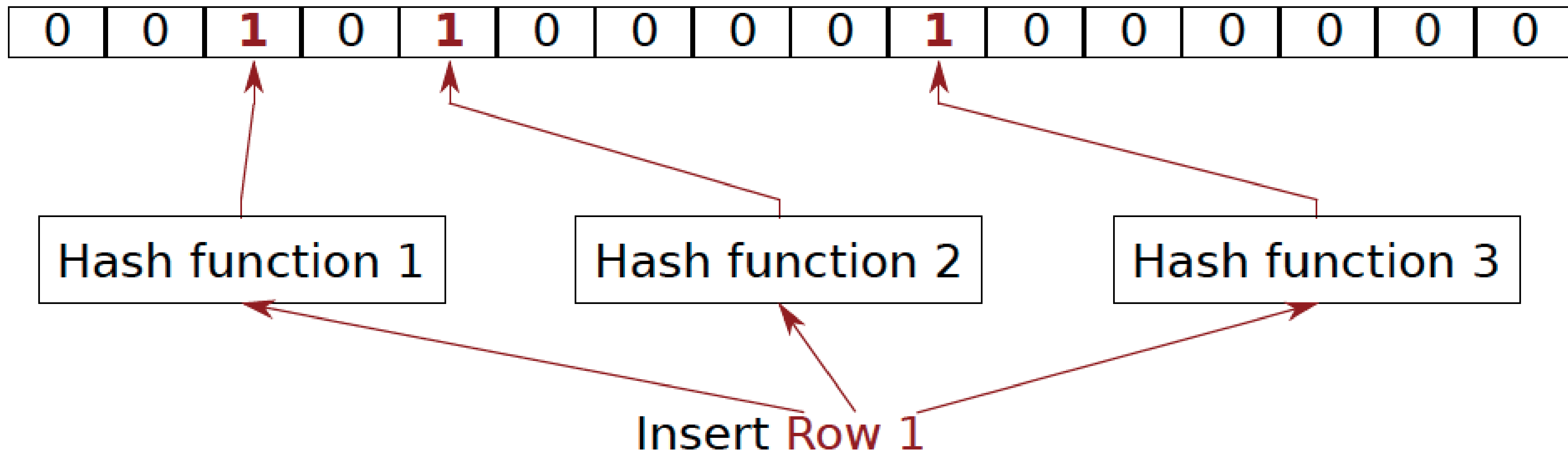
- [Bloom, CACM 1970]
- Probabilistic data structure that compactly represents set membership (presence or absence of element in a set)

- Non-approximate set membership: Use 1 bit per element to indicate absence/presence of each element from an element space of N elements
- Approximate set membership: use a much smaller number of bits and indicate each element's presence/absence with a subset of those bits
  - Some elements map to the bits other elements also map to

- Operations: 1) insert, 2) test, 3) remove all elements

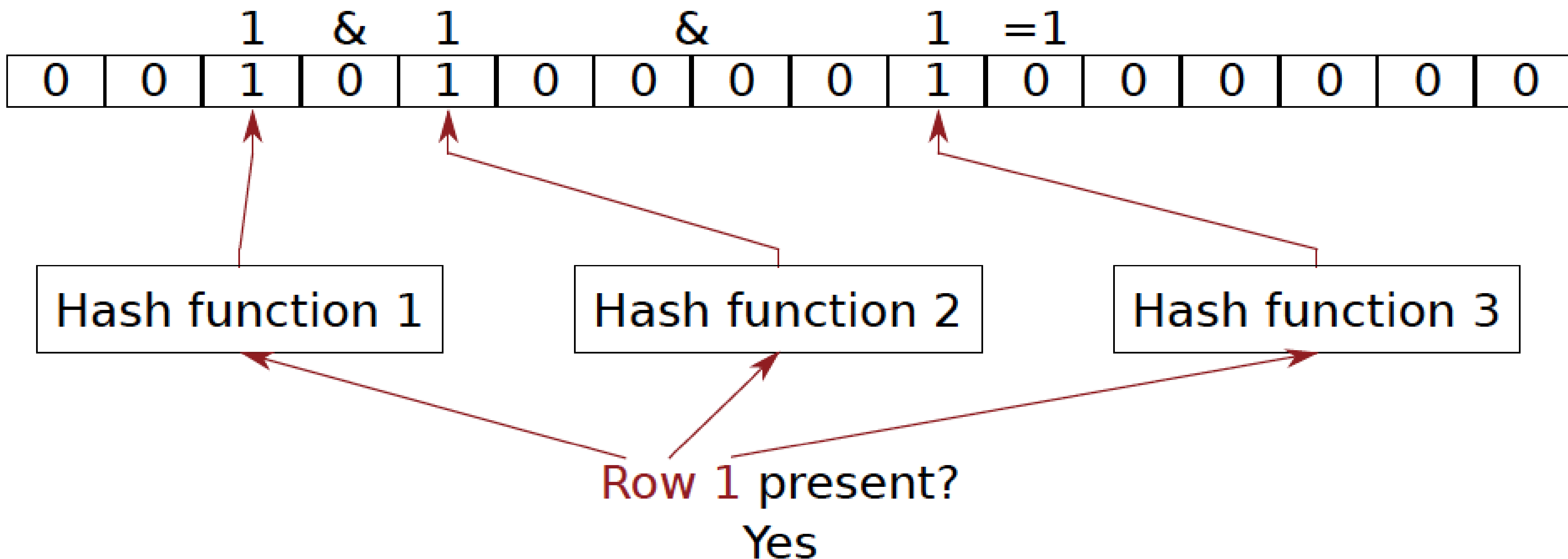
Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors", CACM 1970.

Bloom Filter Operation Example

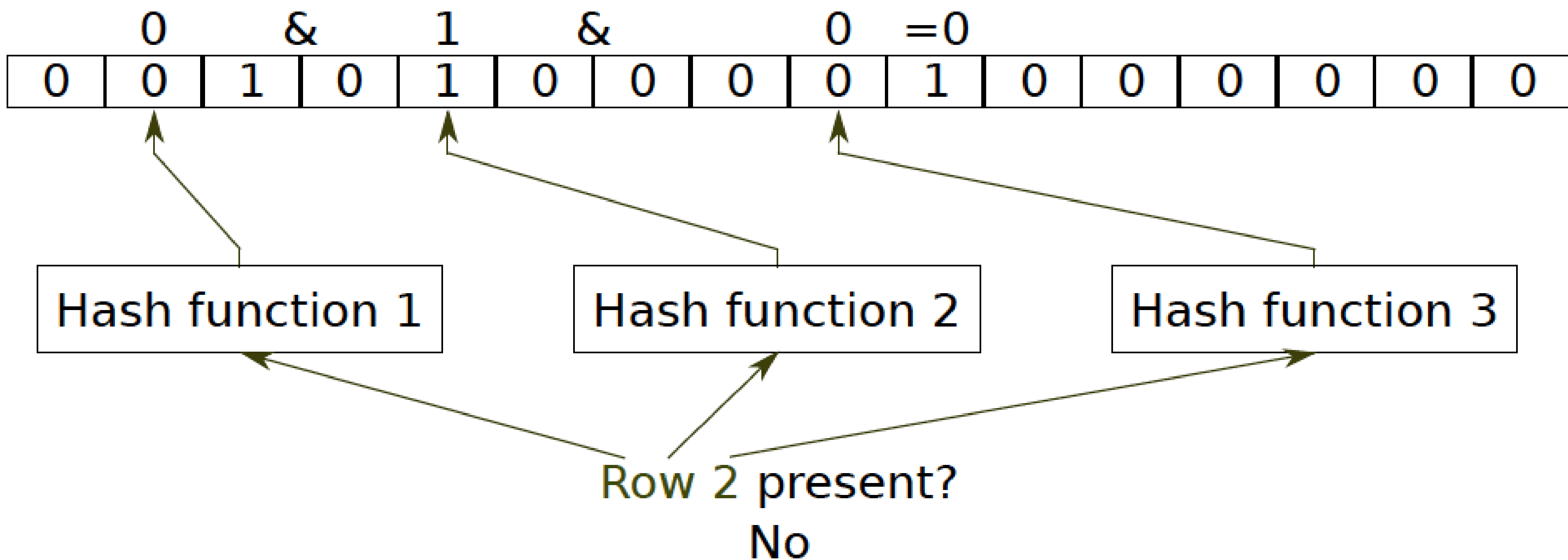


Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors", CACM 1970.

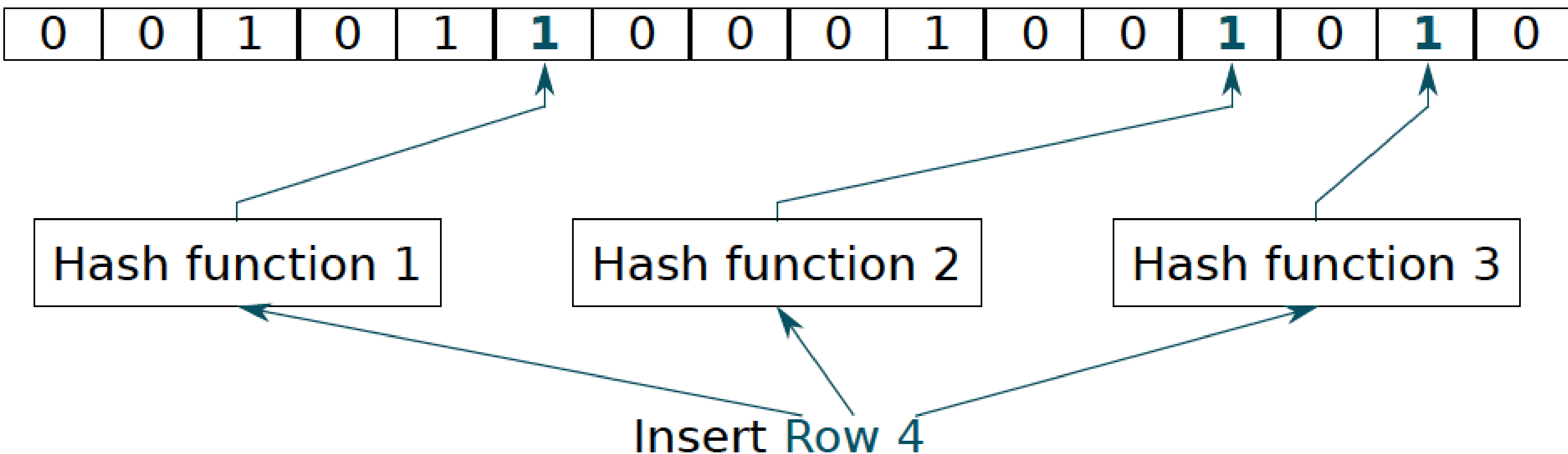
Bloom Filter Operation Example



Bloom Filter Operation Example

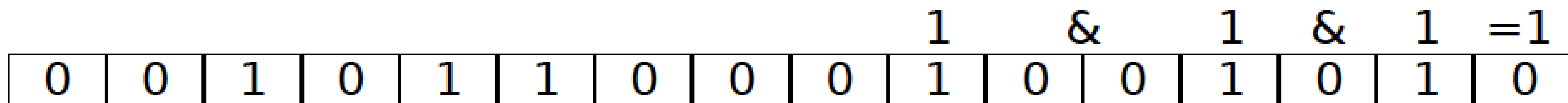


Bloom Filter Operation Example



Bloom Filter Operation Example

Example with 64–128ms bin:



Hash function 1

Hash function 2

Hash function 3

Row 5 present?

Yes (false positive)

Benefits of Bloom Filters as Bins

- False positives: a row may be declared present in the Bloom filter even if it was never inserted
  - Not a problem: Refresh some rows more frequently than needed
- No false negatives: rows are never refreshed less frequently than needed (no correctness problems)
- Scalable: a Bloom filter never overflows (unlike a fixed-size table)
- Efficient: No need to store info on a per-row basis; simple hardware → 1.25 KB for 2 filters for 32 GB DRAM system

Use of Bloom Filters in Hardware

- Useful when you can tolerate false positives in set membership tests

- See the following recent examples for clear descriptions of how Bloom Filters are used
  - Liu et al., "RAIDR: Retention-Aware Intelligent DRAM Refresh," ISCA 2012.
  - Seshadri et al., "The Evicted-Address Filter: A Unified Mechanism to Address Both Cache Pollution and Thrashing," PACT 2012.

3. Refreshing (RAIDR Refresh Controller)

Choose a refresh candidate row

Determine which bin the row is in

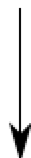
Determine if refreshing is needed

3. Refreshing (RAIDR Refresh Controller)

Memory controller
chooses each row
as a refresh candidate
every 64ms

Row in 64-128ms bin?
(First Bloom filter: 256B)

Row in 128-256ms bin?
(Second Bloom filter: 1KB)



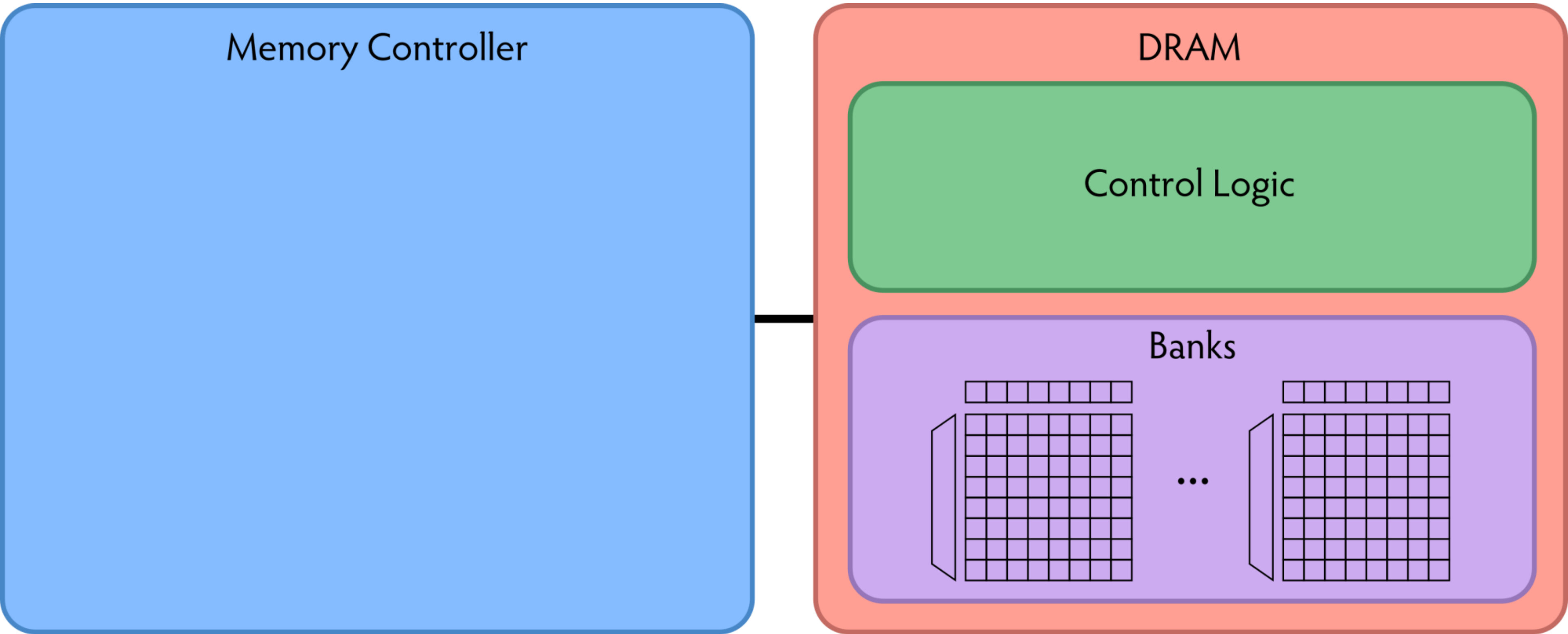
Refresh the row

Every other 64ms window,
refresh the row

Every 4th 64ms window,
refresh the row

Liu et al., "RAIDR: Retention-Aware Intelligent DRAM Refresh," ISCA 2012.

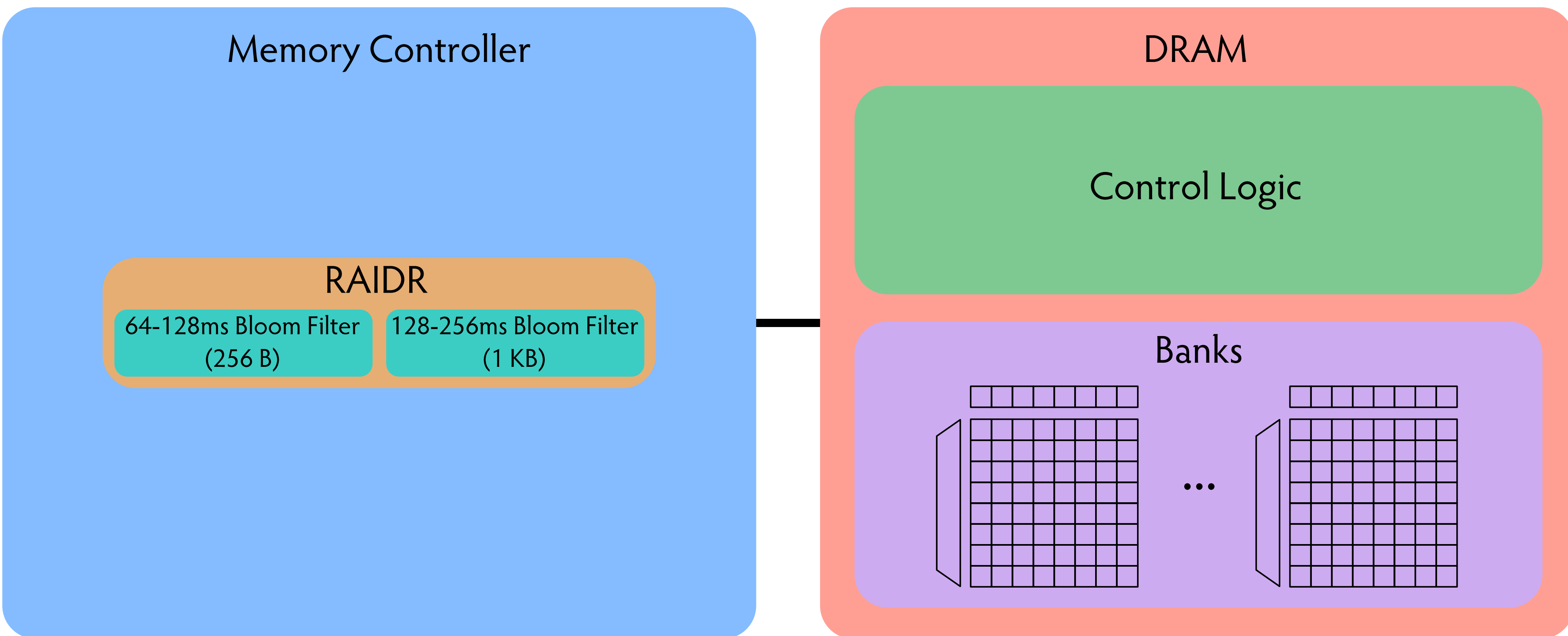
RAIDR: Baseline Design



Refresh control is in DRAM in today's auto-refresh systems

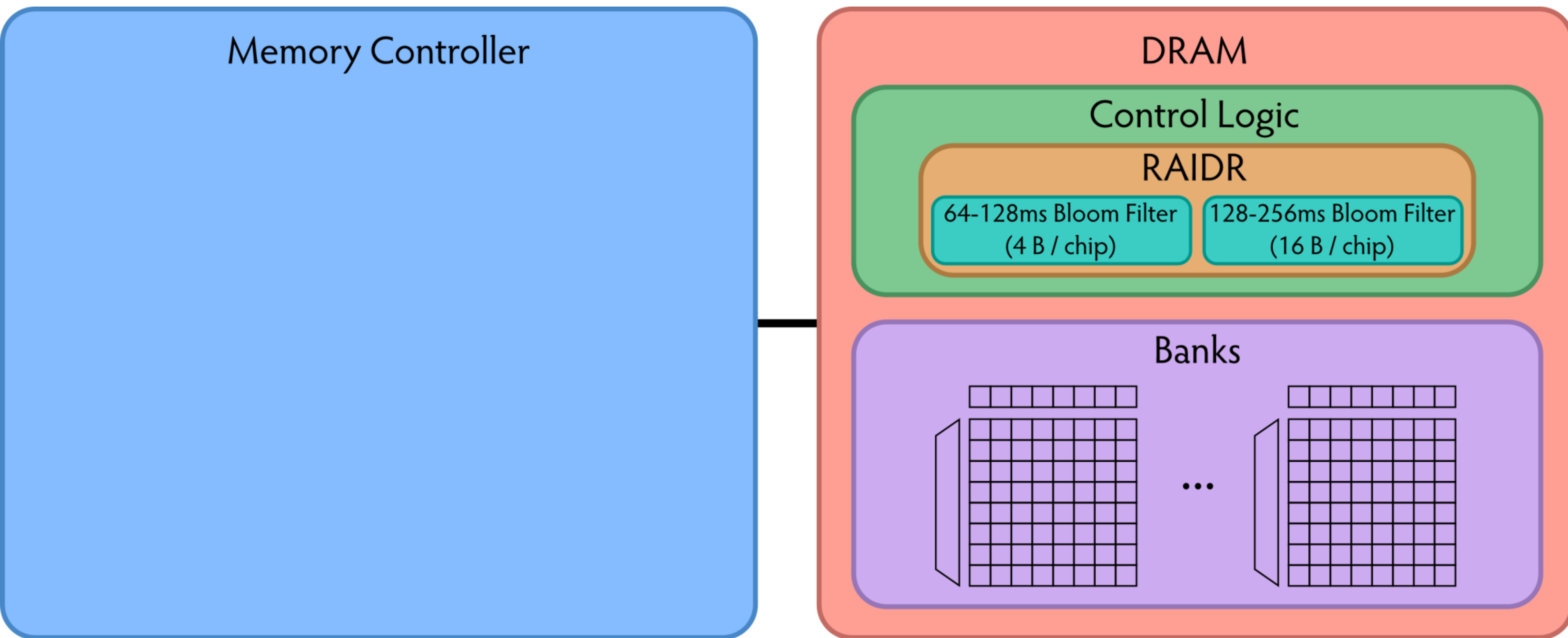
RAIDR can be implemented in either the controller or DRAM

RAIDR in Memory Controller: Option 1



Overhead of RAIDR in DRAM controller:
1.25 KB Bloom Filters, 3 counters, additional commands issued for per-row refresh (all accounted for in evaluations)

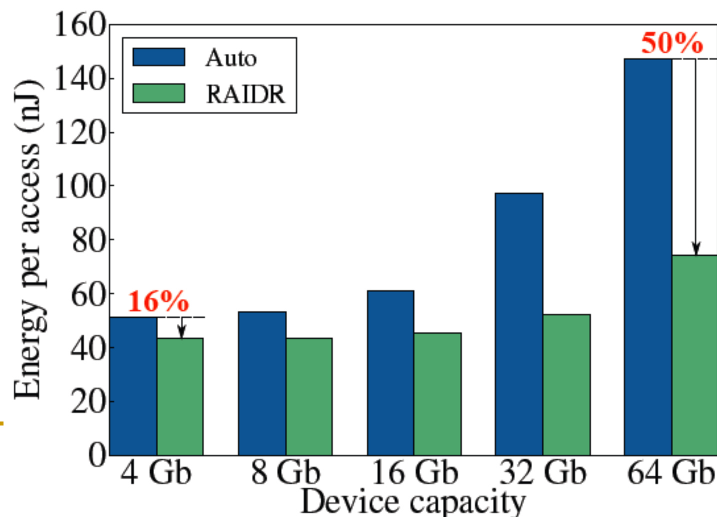
RAIDR in DRAM Chip: Option 2

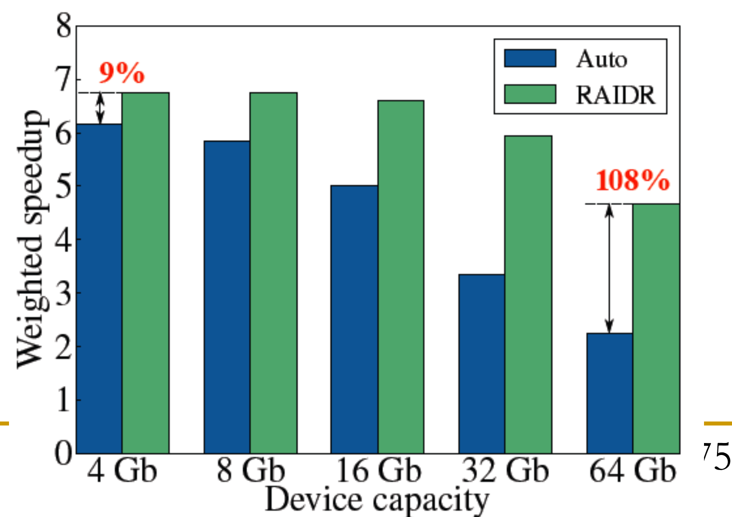


Overhead of RAIDR in DRAM chip:
Per-chip overhead: 20B Bloom Filters, 1 counter (4 Gbit chip)
Total overhead: 1.25KB Bloom Filters, 64 counters (32 GB DRAM)

RAIDR: Results and Takeaways

- System: 32GB DRAM, 8-core; SPEC, TPC-C, TPC-H workloads
- RAIDR hardware cost: 1.25 kB (2 Bloom filters)
- Refresh reduction: 74.6%
- Dynamic DRAM energy reduction: 16%
- Idle DRAM power reduction: 20%
- Performance improvement: 9%
- Benefits increase as DRAM scales in density





DRAM Refresh: More Questions

- What else can you do to reduce the impact of refresh?
- What else can you do if you know the retention times of rows?
- How can you accurately measure the retention time of DRAM rows?
- Recommended reading:
  - Liu et al., "An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms," ISCA 2013.