

18-447

Computer Architecture

Lecture 23: Memory Management

Prof. Onur Mutlu

Carnegie Mellon University

Spring 2015, 3/27/2015

Where We Are in Lecture Schedule

- The memory hierarchy
 - Caches, caches, more caches
 - Virtualizing the memory hierarchy: Virtual Memory
 - Main memory: DRAM
 - Main memory control, scheduling
 - Memory latency tolerance techniques
 - Non-volatile memory
-
- Multiprocessors
 - Coherence and consistency
 - Interconnection networks
 - Multi-core issues (e.g., heterogeneous multi-core)

Required Reading (for the Next Few Lectures)

- Onur Mutlu, Justin Meza, and Lavanya Subramanian, **"The Main Memory System: Challenges and Opportunities"**

*Invited Article in Communications of the Korean Institute of Information Scientists and Engineers (**KIISE**), 2015.*

http://users.ece.cmu.edu/~omutlu/pub/main-memory-system_kiise15.pdf

Required Readings on DRAM

■ DRAM Organization and Operation Basics

- Sections 1 and 2 of: Lee et al., “Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture,” HPCA 2013.

http://users.ece.cmu.edu/~omutlu/pub/tldram_hpca13.pdf

- Sections 1 and 2 of Kim et al., “A Case for Subarray-Level Parallelism (SALP) in DRAM,” ISCA 2012.

http://users.ece.cmu.edu/~omutlu/pub/salp-dram_isca12.pdf

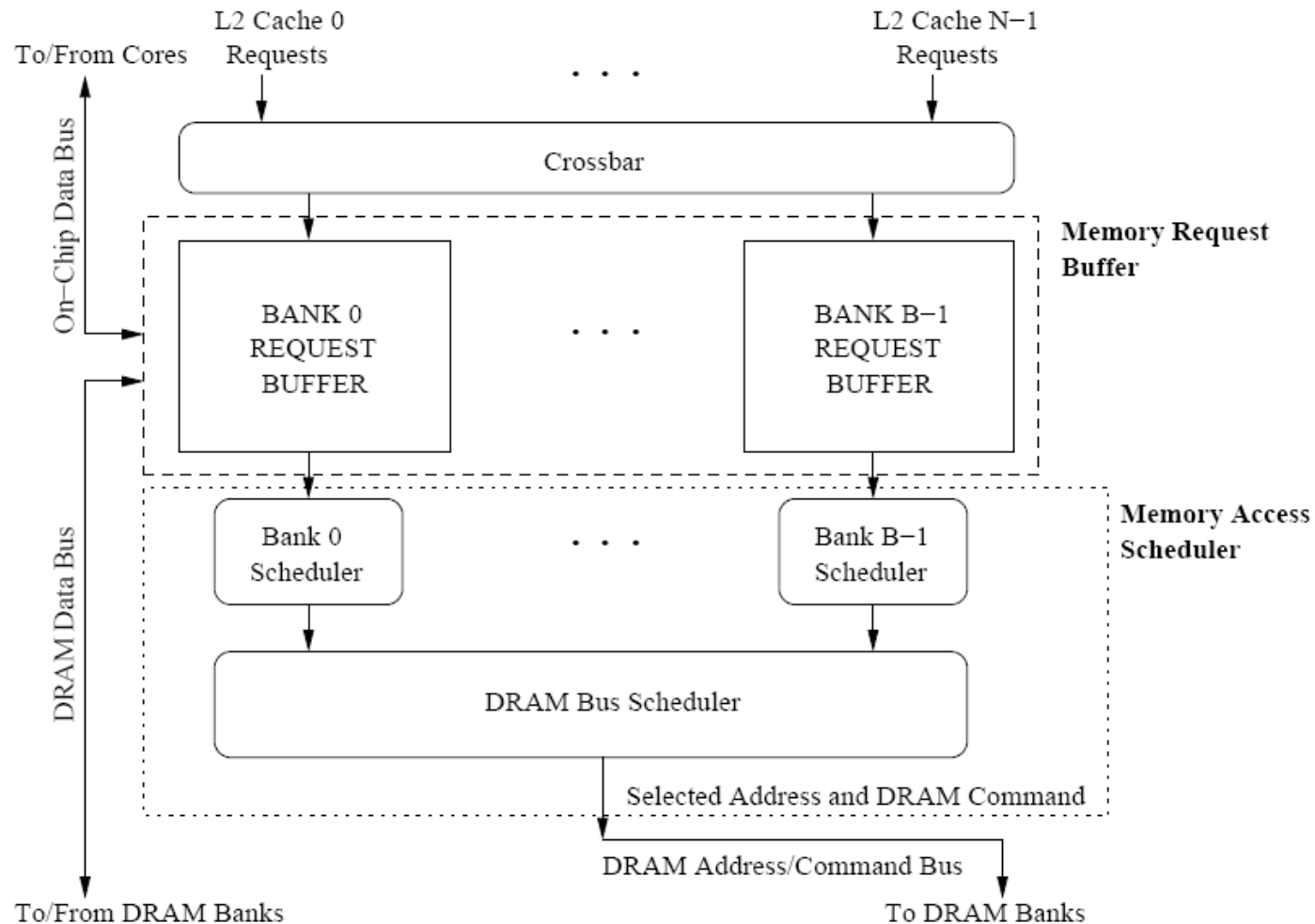
■ DRAM Refresh Basics

- Sections 1 and 2 of Liu et al., “RAIDR: Retention-Aware Intelligent DRAM Refresh,” ISCA 2012.

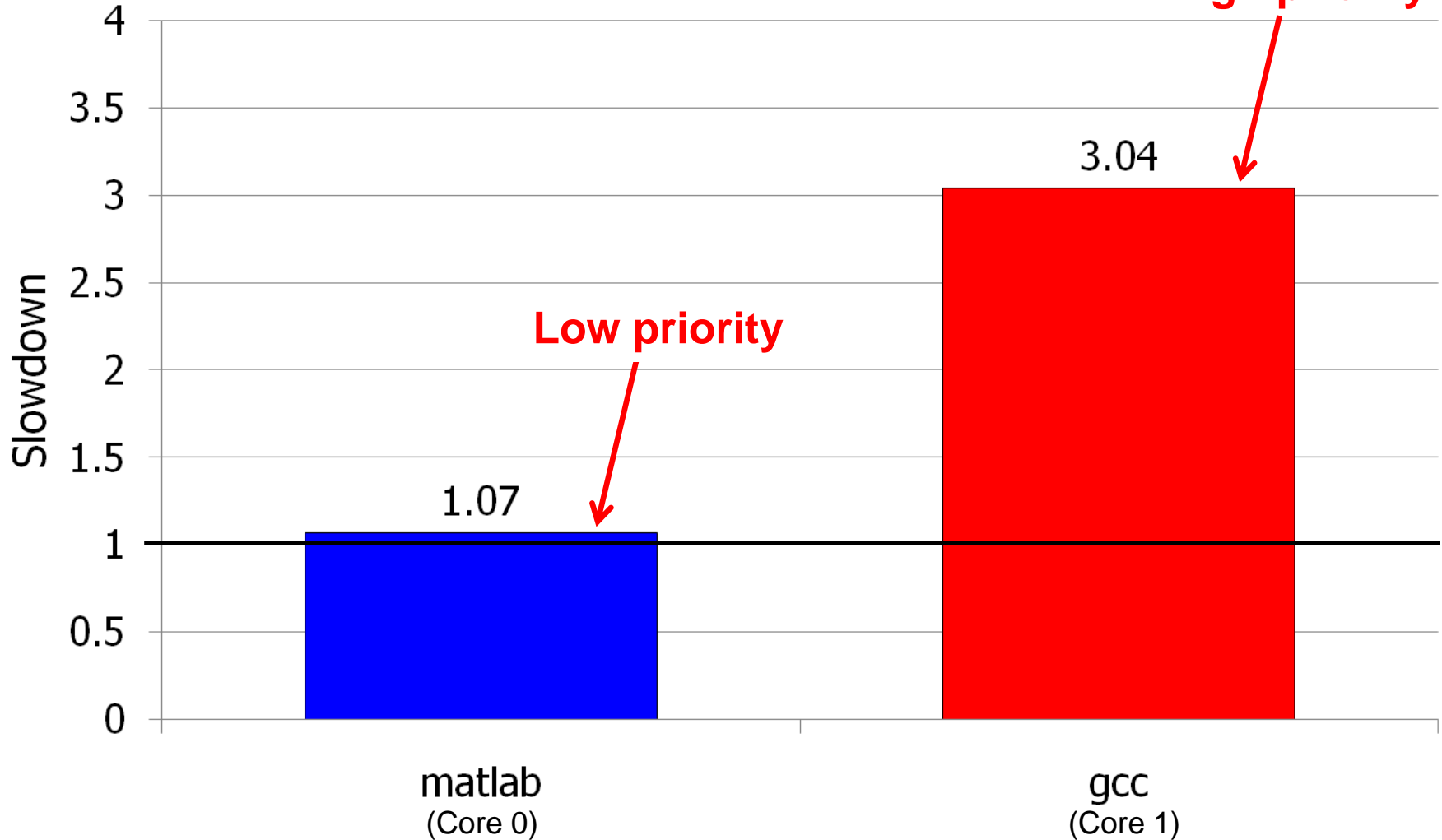
http://users.ece.cmu.edu/~omutlu/pub/raidr-dram-refresh_isca12.pdf

Memory Interference and Scheduling in Multi-Core Systems

Review: A Modern DRAM Controller



(Un)expected Slowdowns in Multi-Core



Moscibroda and Mutlu, “[Memory performance attacks: Denial of memory service in multi-core systems](#),” USENIX Security 2007.

Memory Scheduling Techniques

- We covered
 - FCFS
 - FR-FCFS
 - STFM (Stall-Time Fair Memory Access Scheduling)
 - PAR-BS (Parallelism-Aware Batch Scheduling)
 - ATLAS
 - TCM (Thread Cluster Memory Scheduling)
- There are many more ...
- See your required reading (Section 7):
 - Mutlu et al., “**The Main Memory System: Challenges and Opportunities**,” KIISE 2015.

Other Ways of Handling Memory Interference

Fundamental Interference Control Techniques

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

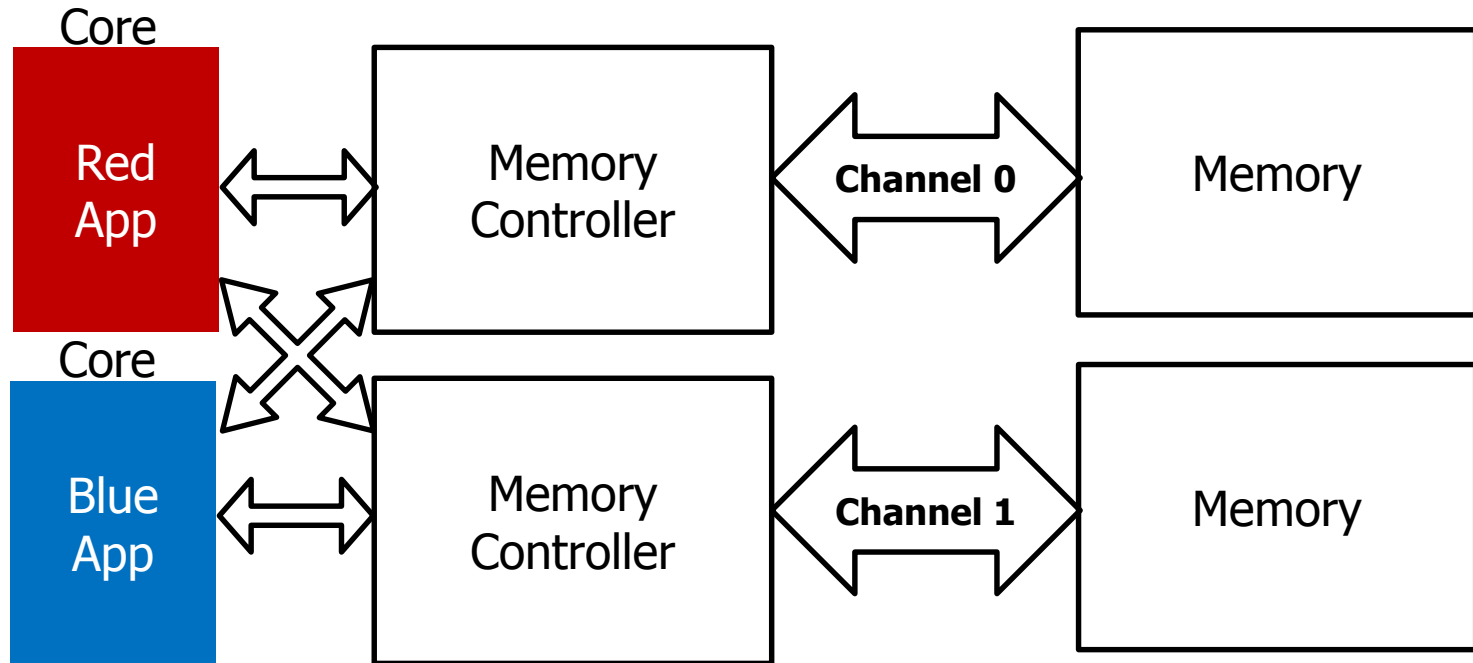
1. **Prioritization** or request scheduling

2. **Data mapping** to banks/channels/ranks

3. **Core/source throttling**

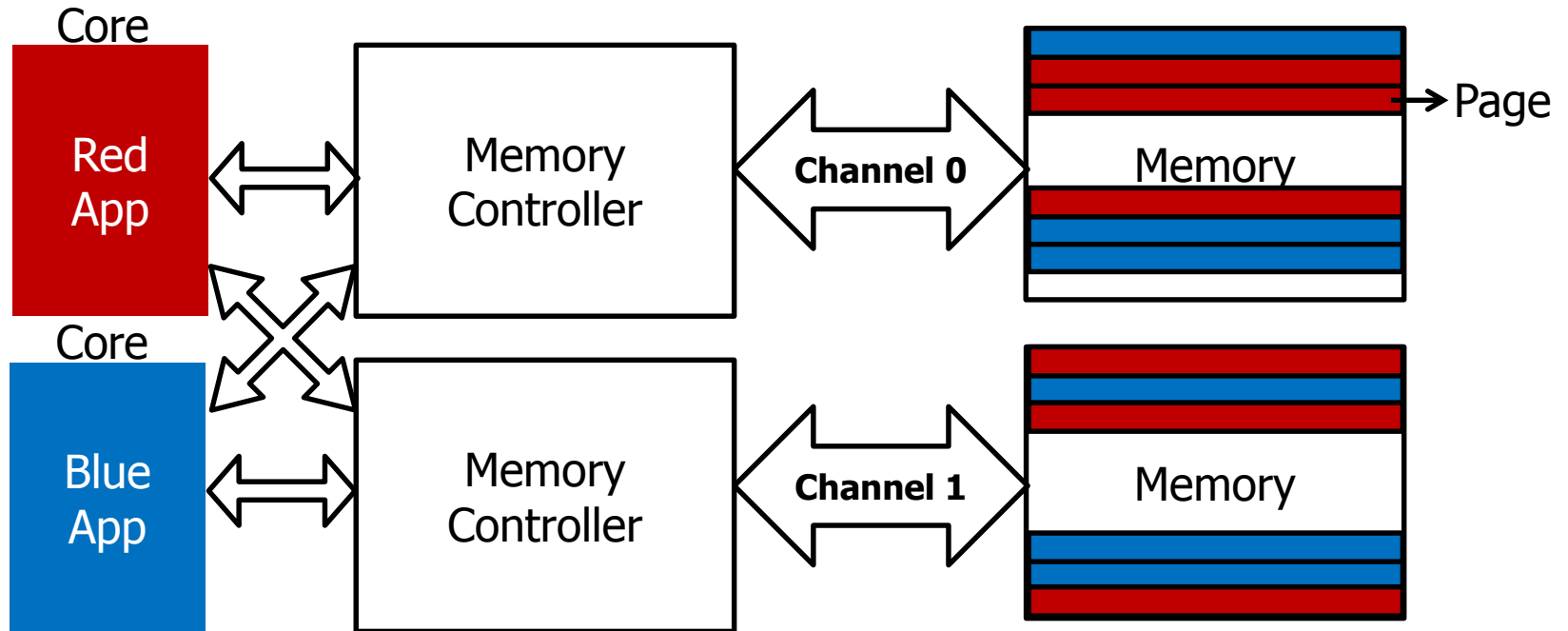
4. **Application/thread scheduling**

Observation: Modern Systems Have Multiple Channels



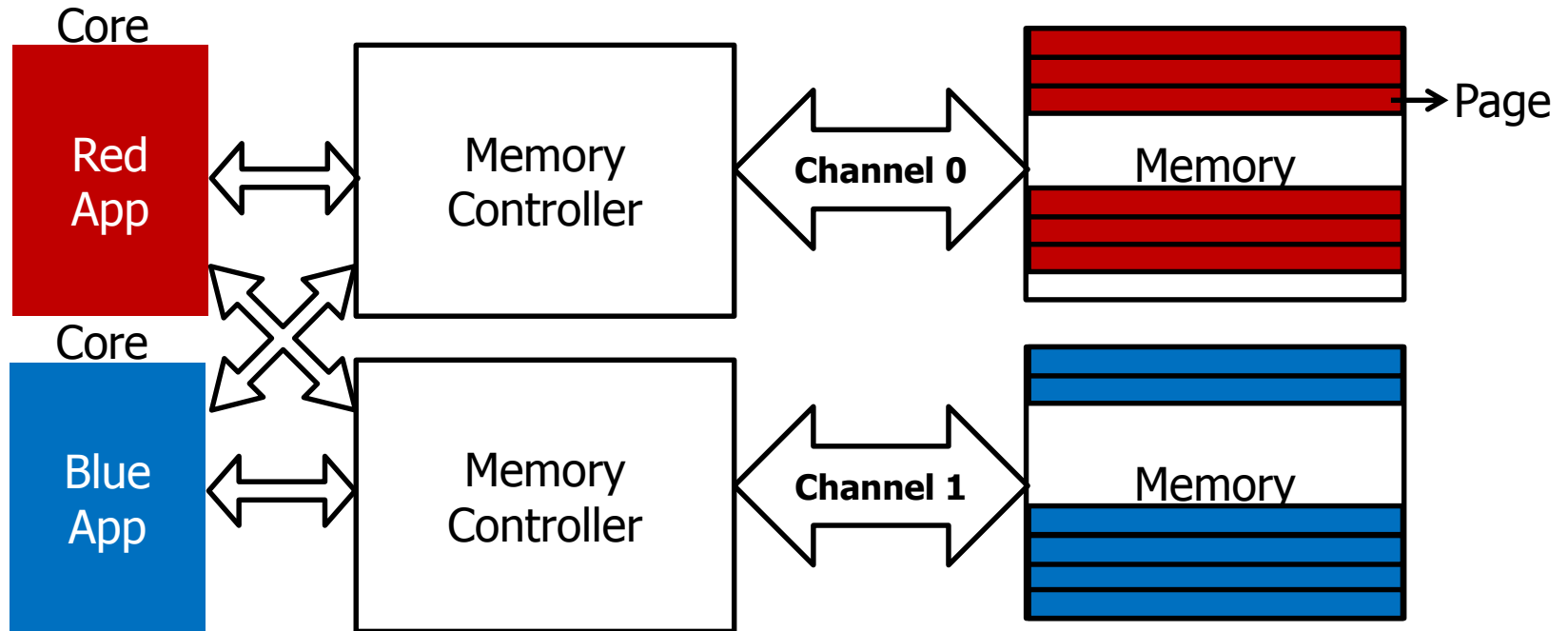
A new degree of freedom
Mapping data across multiple channels

Data Mapping in Current Systems



Causes interference between applications' requests

Partitioning Channels Between Applications



Eliminates interference between applications' requests

Overview: Memory Channel Partitioning (MCP)

■ Goal

- Eliminate harmful interference between applications

■ Basic Idea

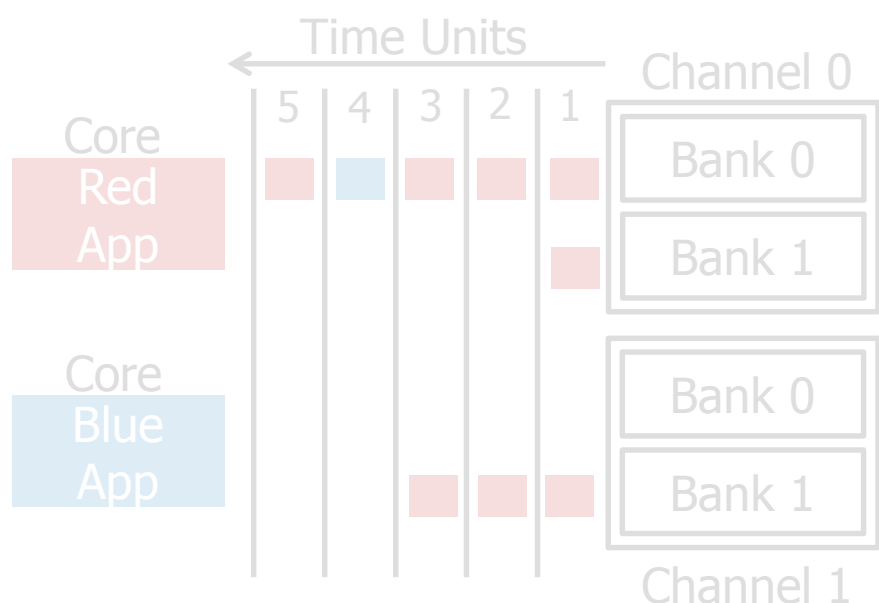
- Map the data of **badly-interfering applications** to different channels

■ Key Principles

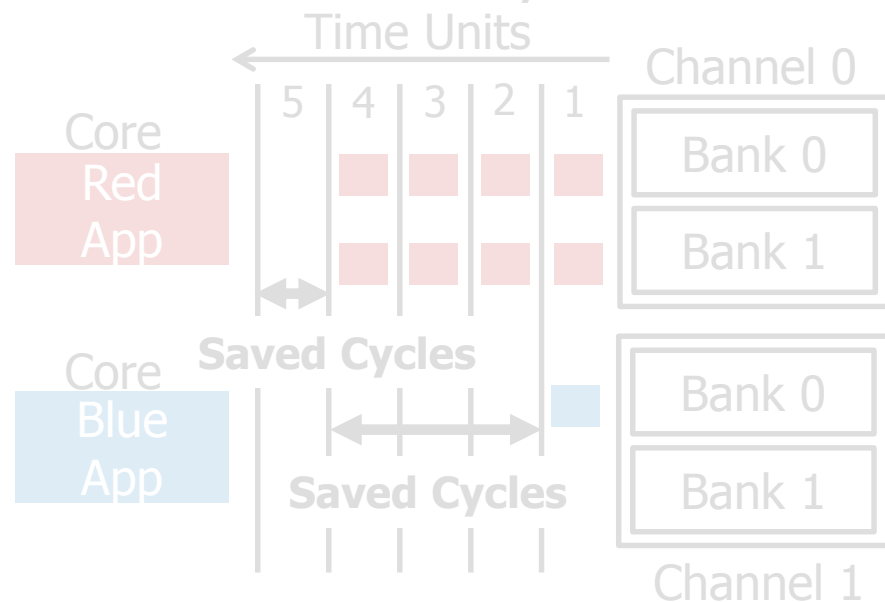
- Separate **low and high memory-intensity applications**
- Separate **low and high row-buffer locality applications**

Key Insight 1: Separate by Memory Intensity

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



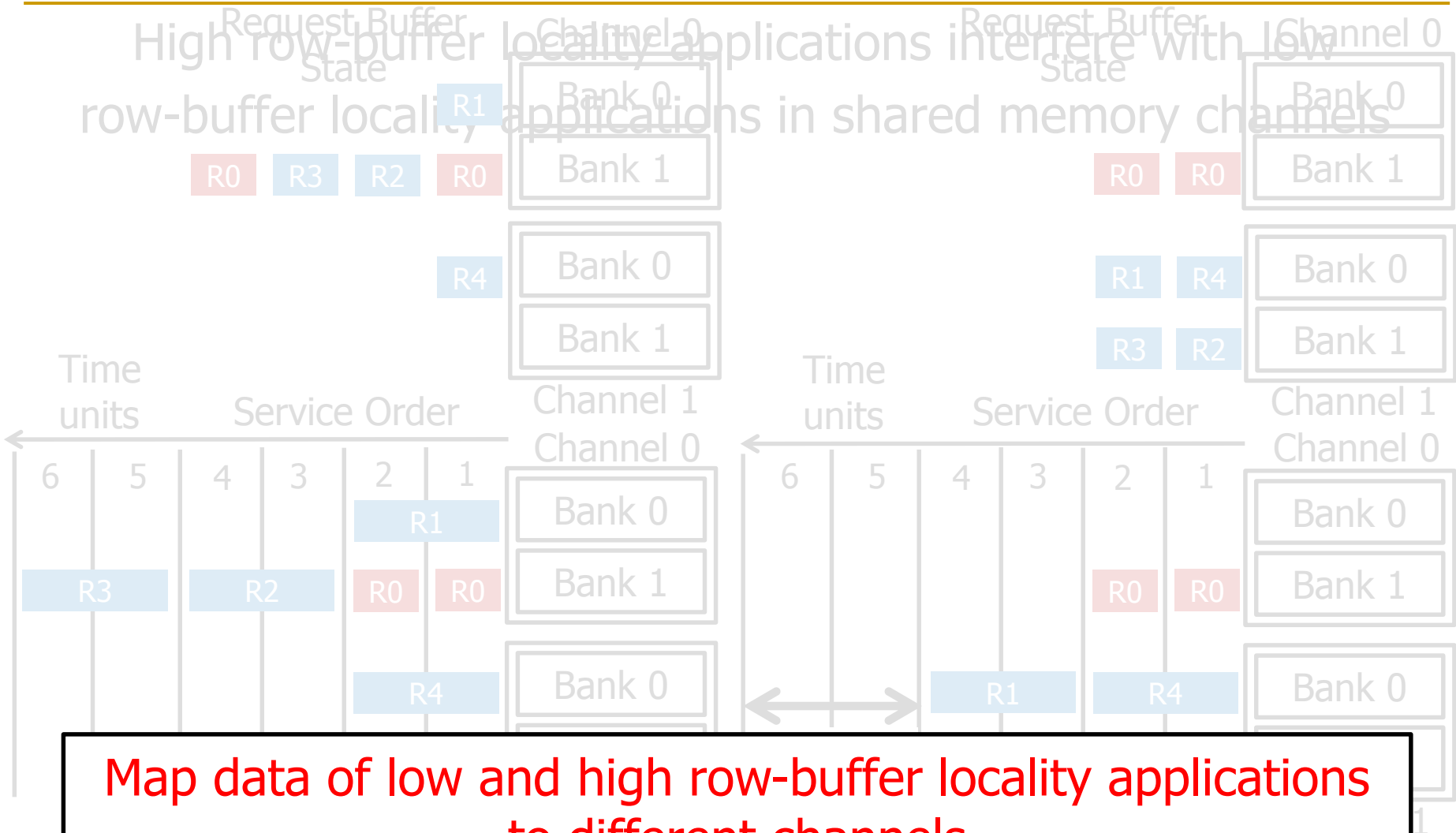
Conventional Page Mapping



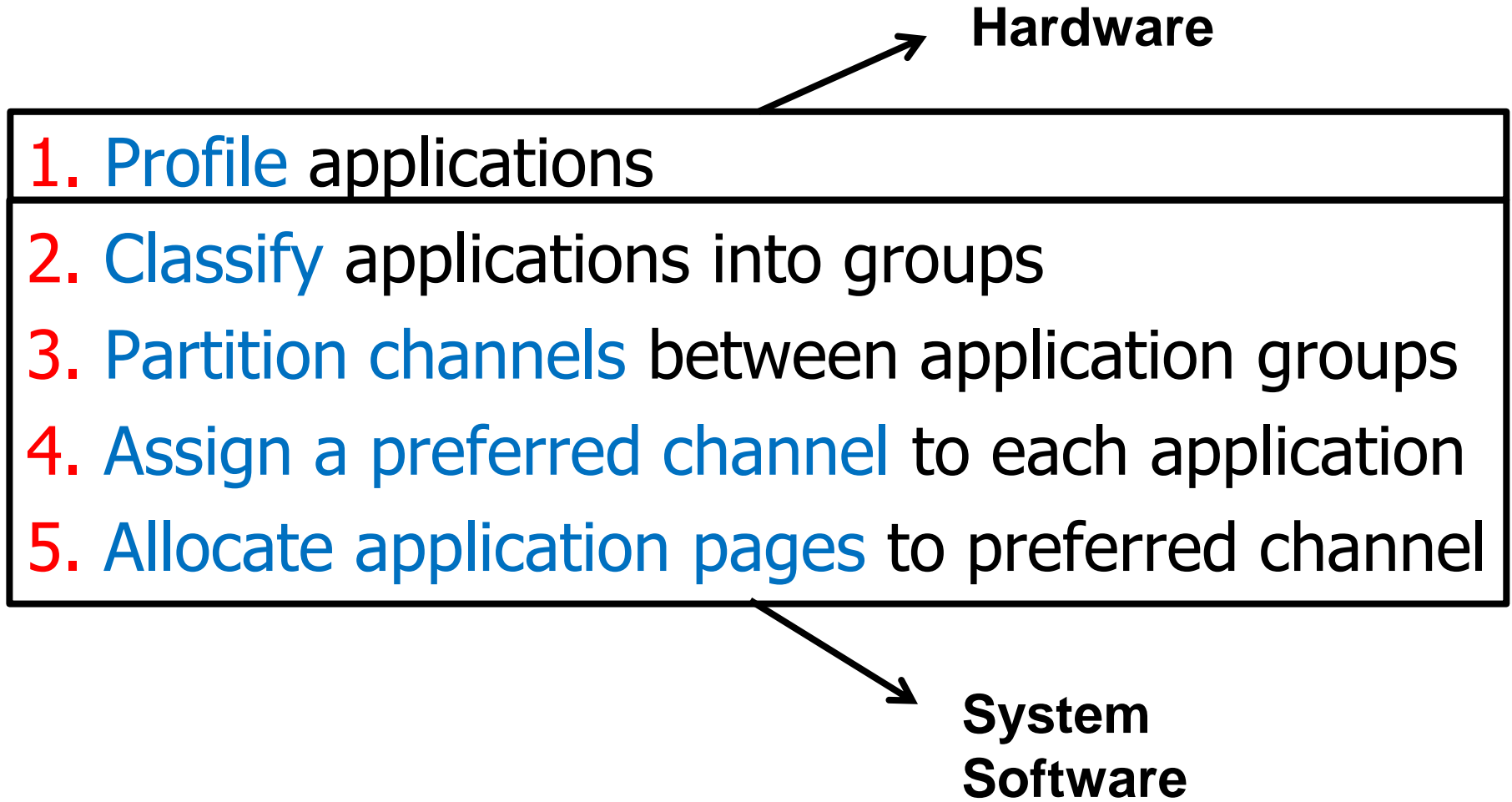
Channel Partitioning

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

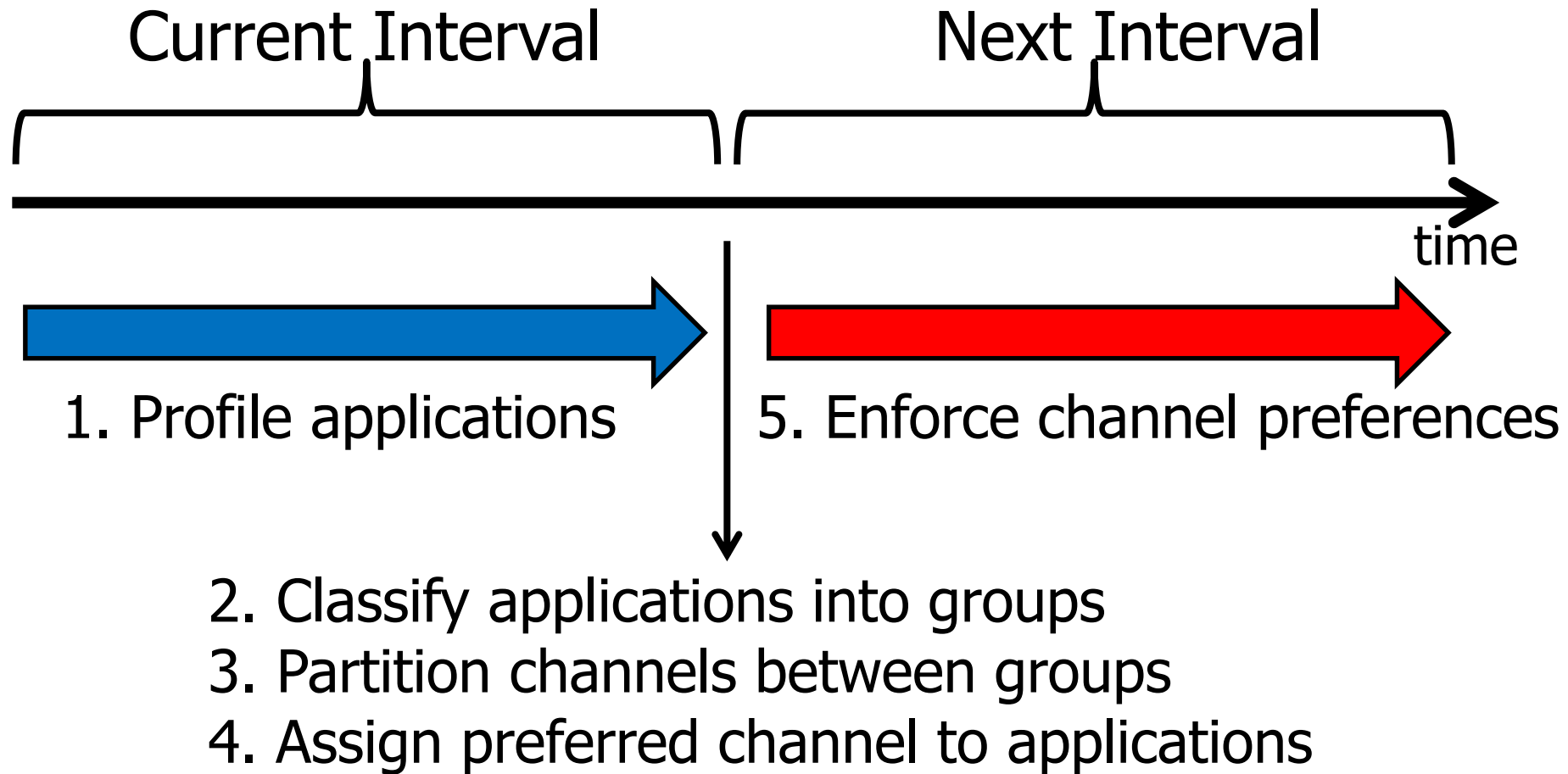
Key Insight 2: Separate by Row-Buffer Locality



Memory Channel Partitioning (MCP) Mechanism



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

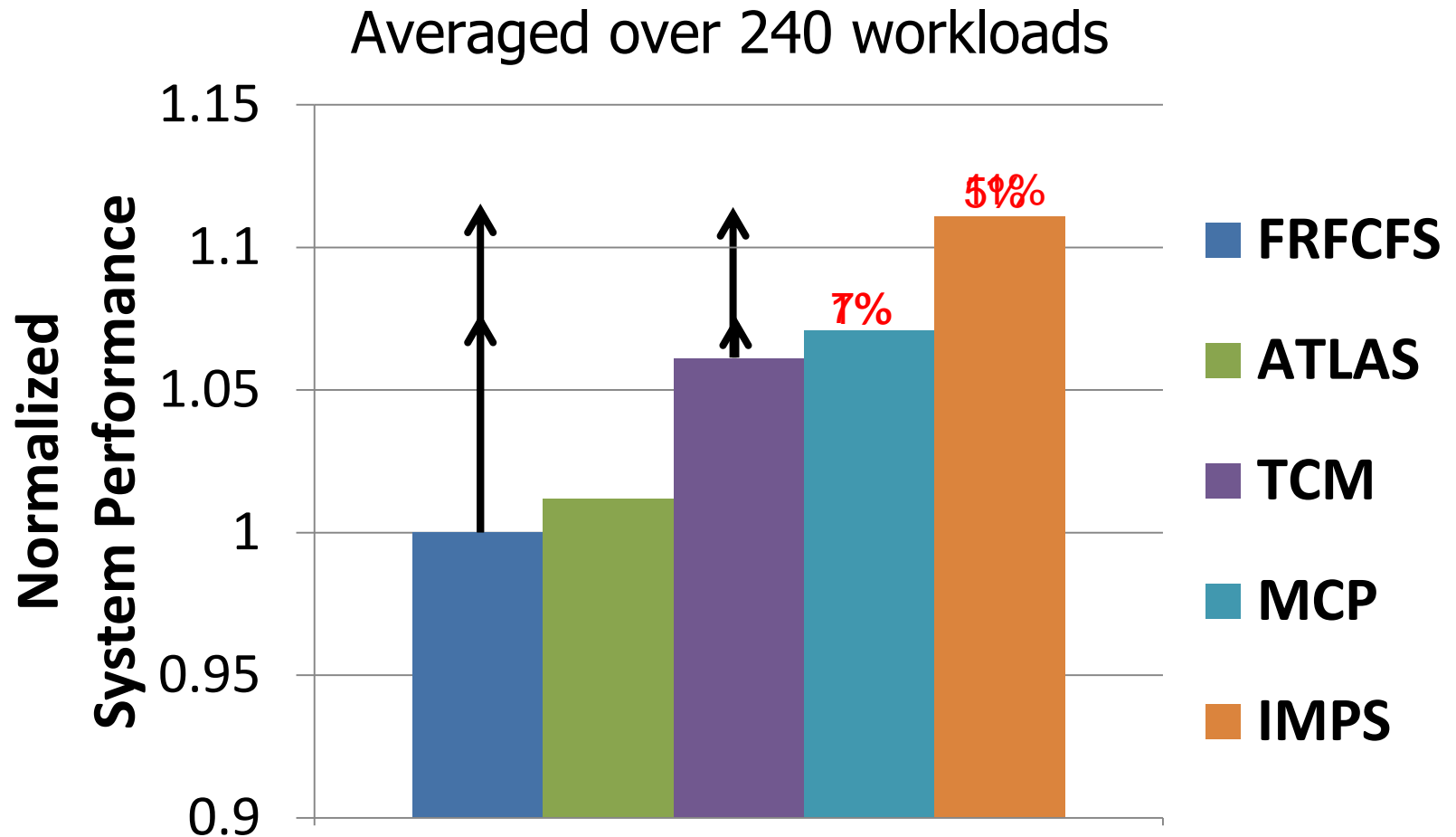
Integrated Memory Partitioning and Scheduling (IMPS)

- Always prioritize very low memory-intensity applications in the memory scheduler
- Use memory channel partitioning to mitigate interference between other applications

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

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

Fundamental Interference Control Techniques

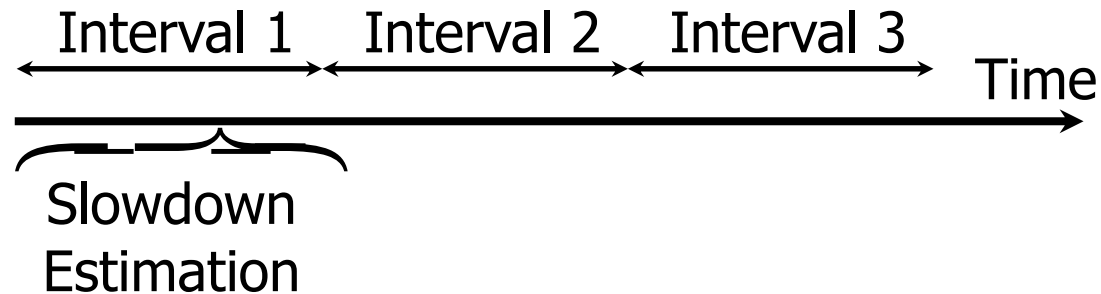
- **Goal:** to reduce/control inter-thread memory interference

1. **Prioritization** or request scheduling
2. **Data mapping** to banks/channels/ranks
3. **Core/source throttling**
4. **Application/thread scheduling**

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) [ASPLOS'10]



FST

Runtime
Unfairness
Evaluation

Unfairness Estimate

App-slowest

App-interfering

Dynamic
Request Throttling

- 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
 - + Can reduce overall load/contention in the memory system
- Disadvantages
 - Requires interference/slowdown estimations → difficult to estimate
 - 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**

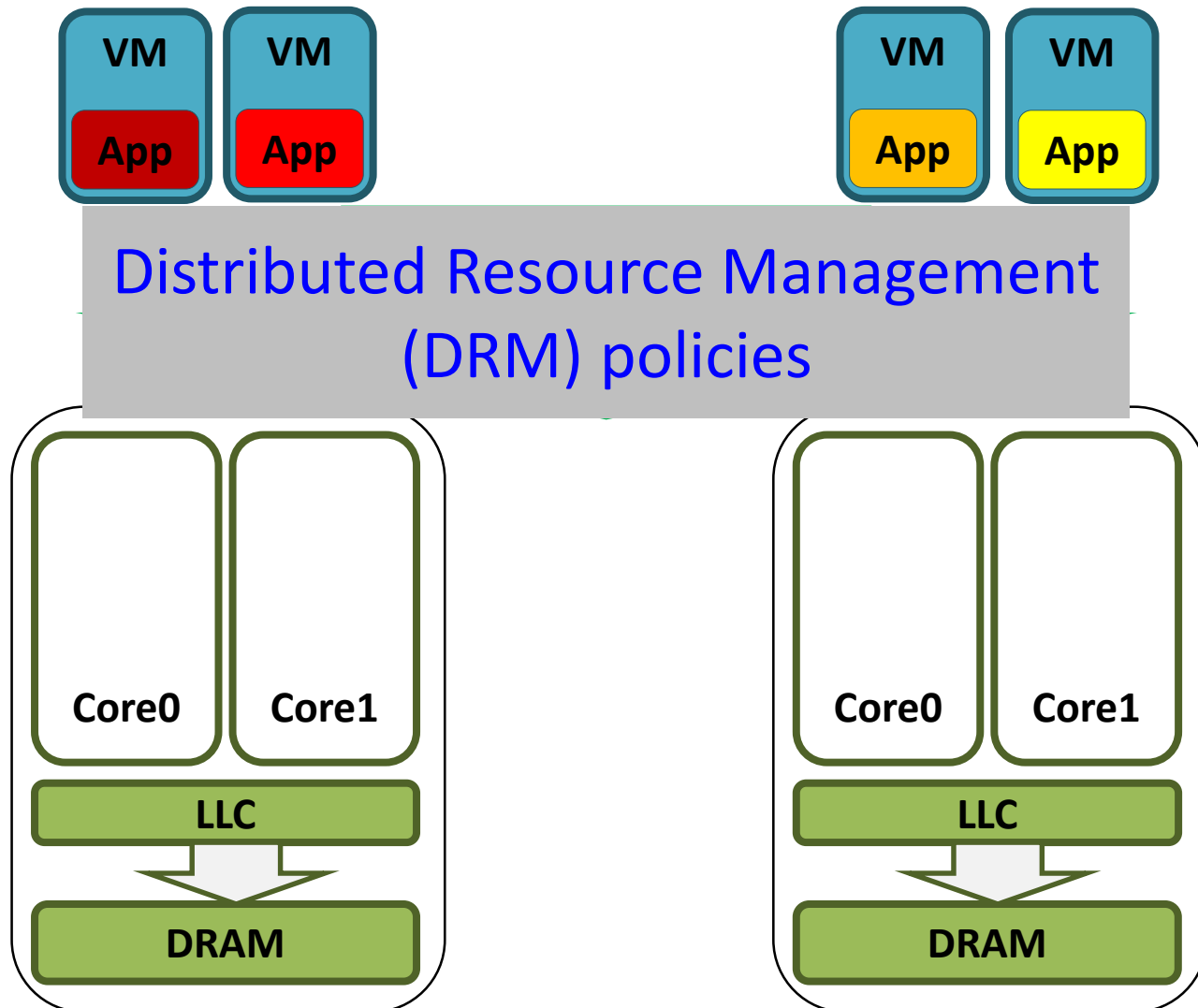
4. **Application/thread scheduling**

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

Interference-Aware Thread Scheduling

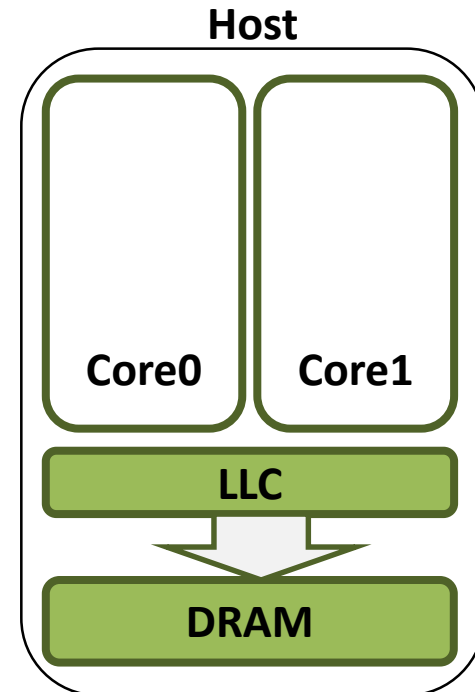
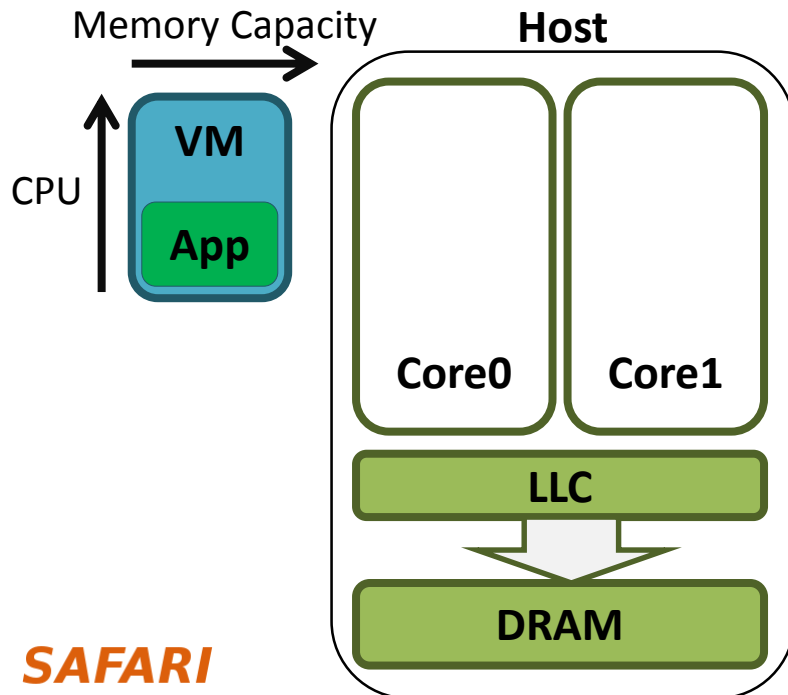
- An example from scheduling in clusters (data centers)
- Clusters can be running virtual machines

Virtualized Cluster



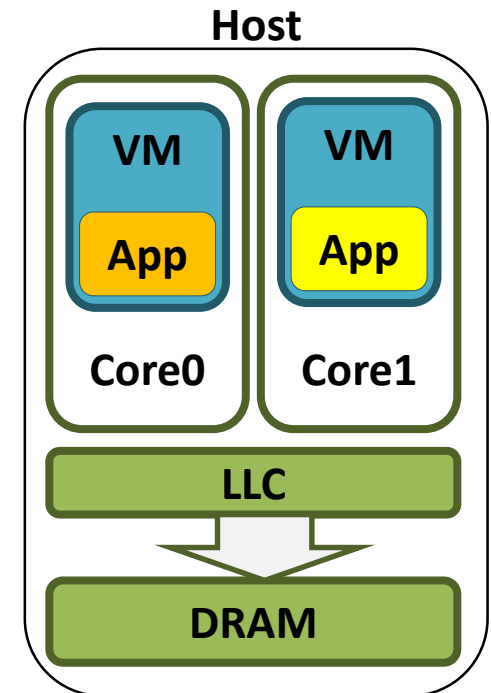
Conventional DRM Policies

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



Microarchitecture-level Interference

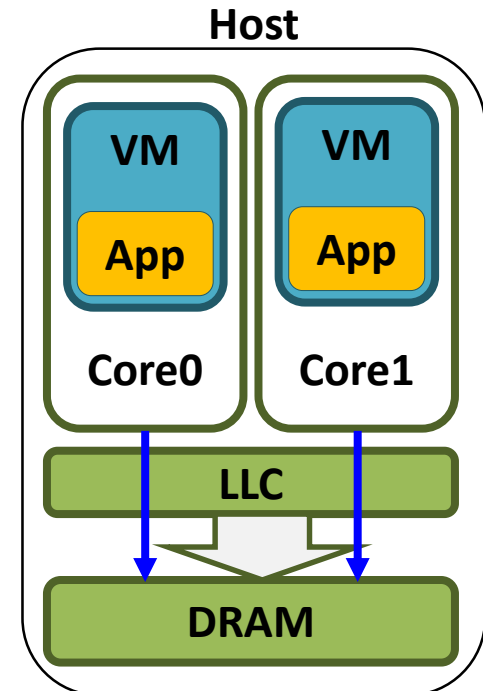
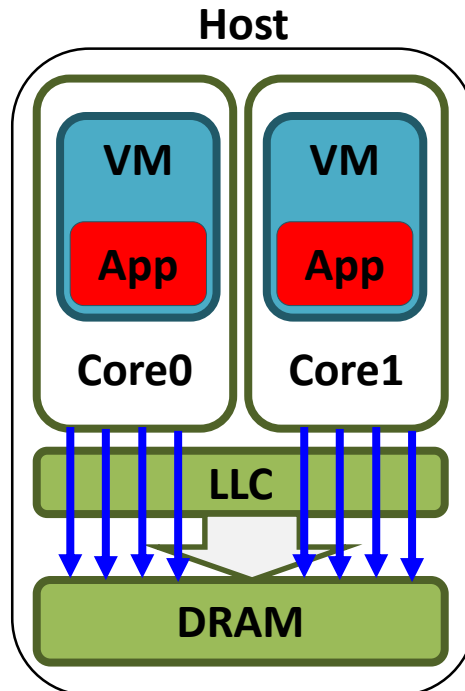
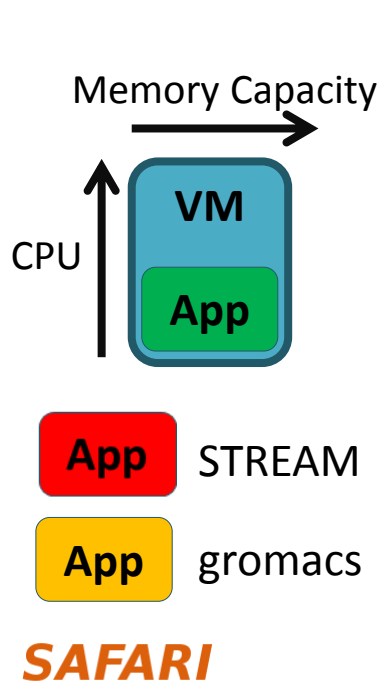
- VMs within a host compete for:
 - Shared cache capacity
 - Shared memory bandwidth



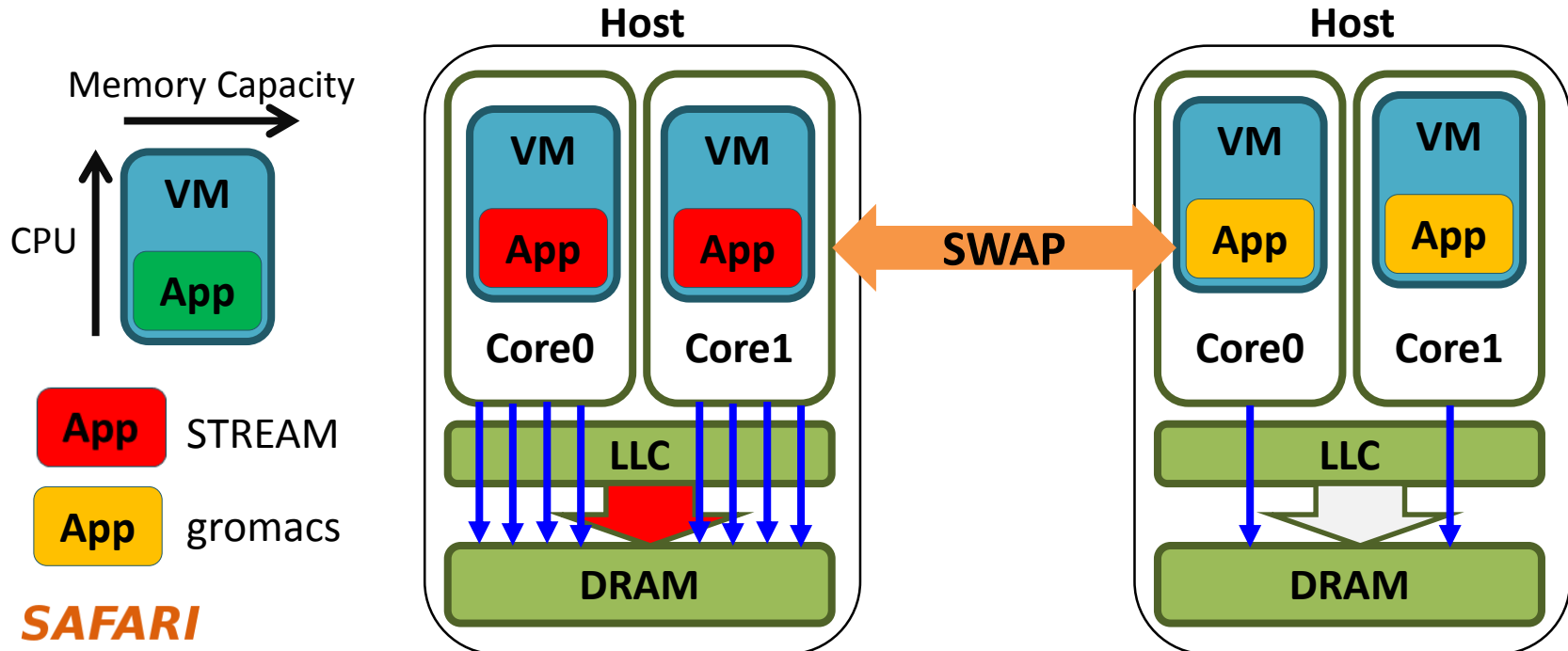
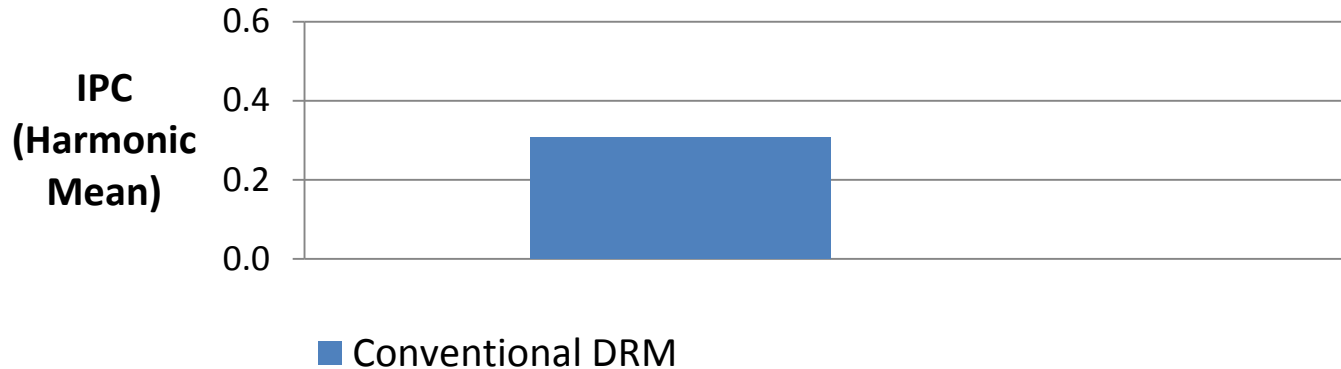
Can operating-system-level metrics capture the microarchitecture-level resource interference?

Microarchitecture Unawareness

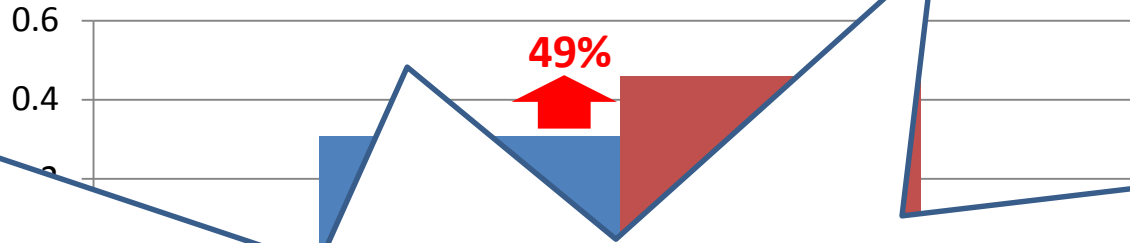
VM	Operating-system-level metrics		Microarchitecture-level metrics	
	CPU Utilization	Memory Capacity	LLC Hit Ratio	Memory Bandwidth
App	92%	369 MB	2%	2267 MB/s
App	93%	348 MB	98%	1 MB/s



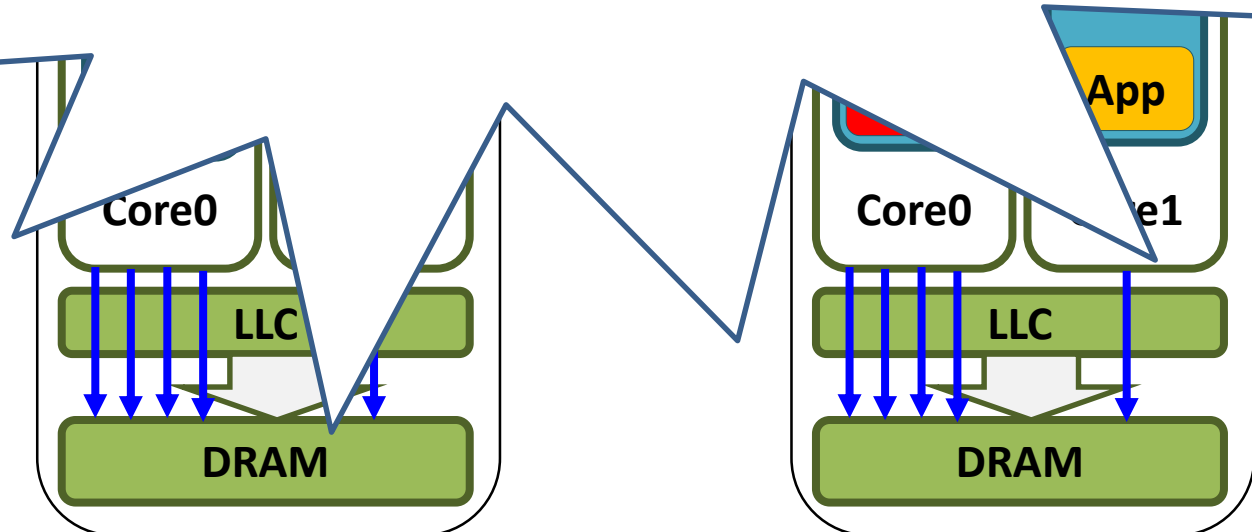
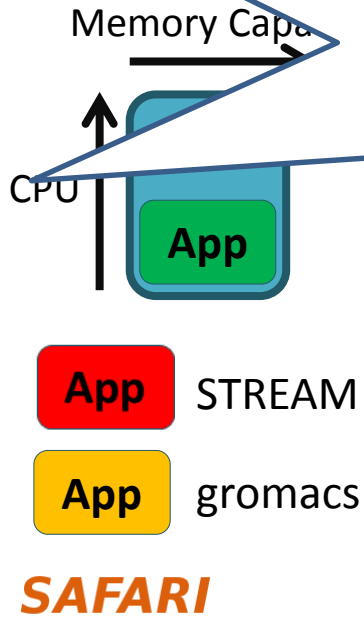
Impact on Performance



Impact on Performance



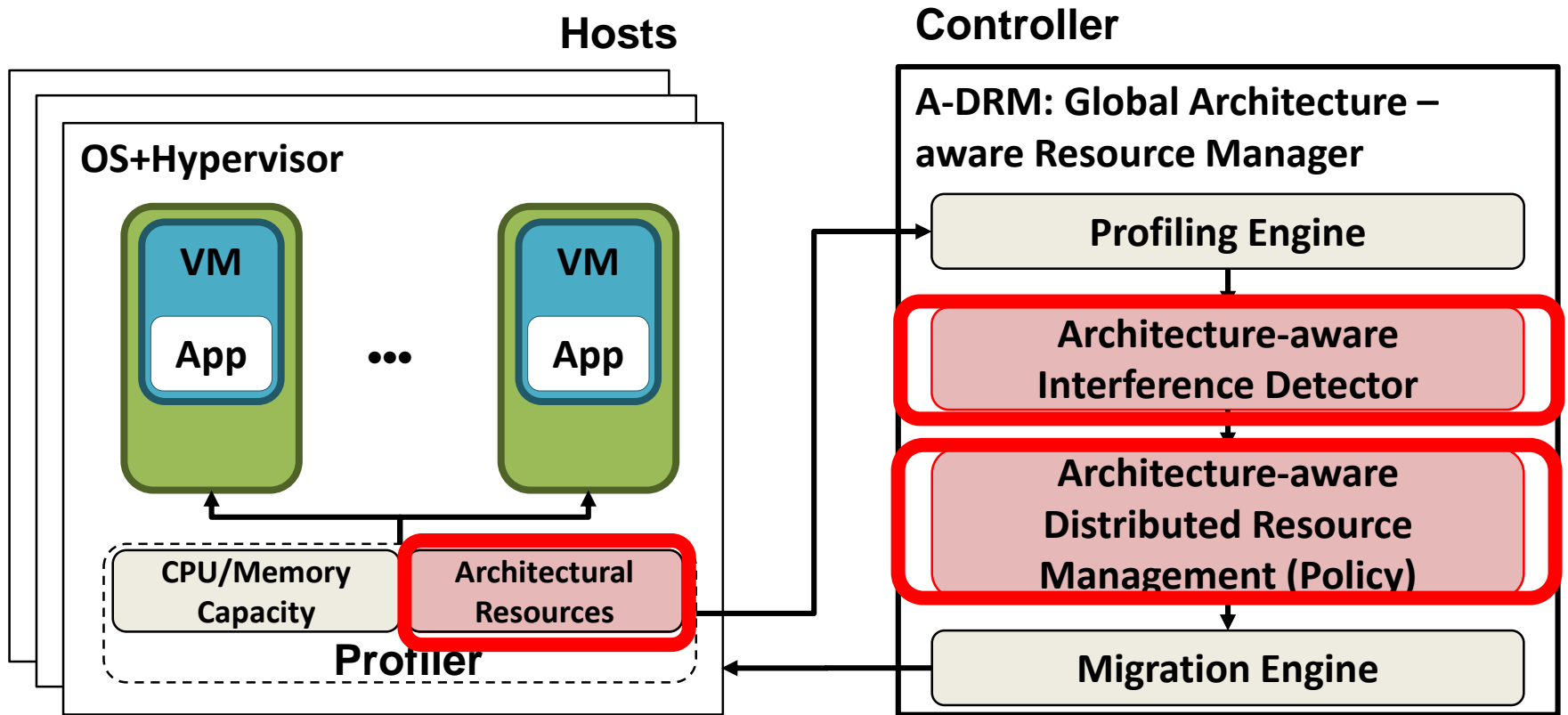
We need microarchitecture-level interference awareness in DRM!



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



More on Architecture-Aware DRM

- Optional Reading
- Wang et al., “A-DRM: Architecture-aware Distributed Resource Management of Virtualized Clusters,” VEE 2015.
 - http://users.ece.cmu.edu/~omutlu/pub/architecture-aware-distributed-resource-management_vee15.pdf

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 (no need to modify the hardware resources)

■ Disadvantages and Limitations

- High overhead to migrate threads 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?

Handling Memory Interference In Multithreaded Applications

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

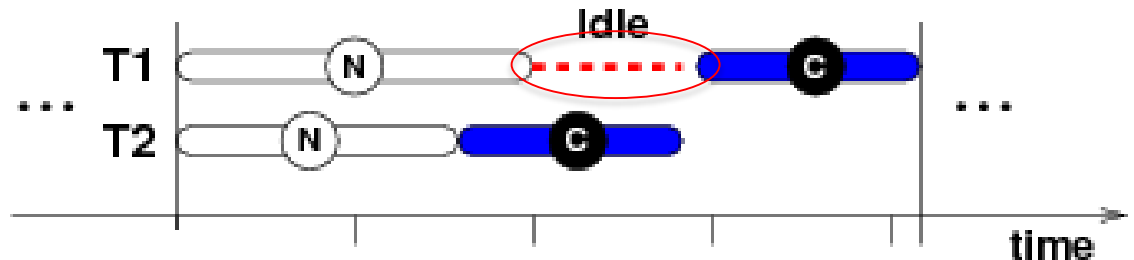
 lock(A)

 Update shared data

 unlock(A)

C

}



Barriers

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

Each thread:

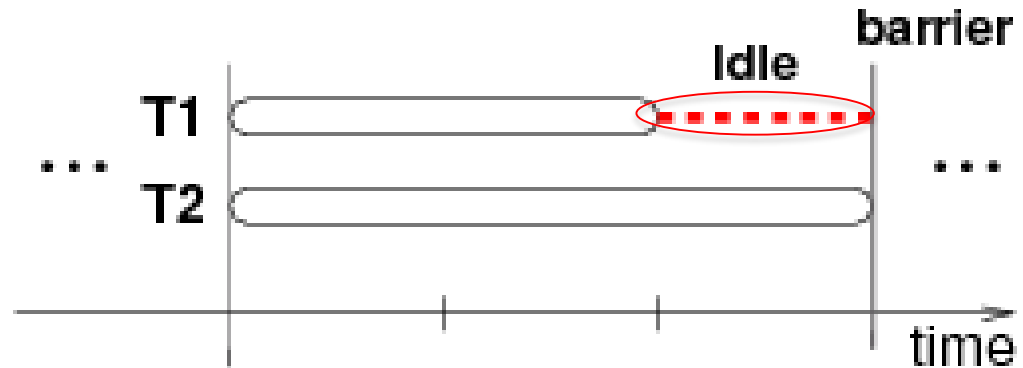
```
loop1 {  
    Compute
```

```
}
```

```
barrier
```

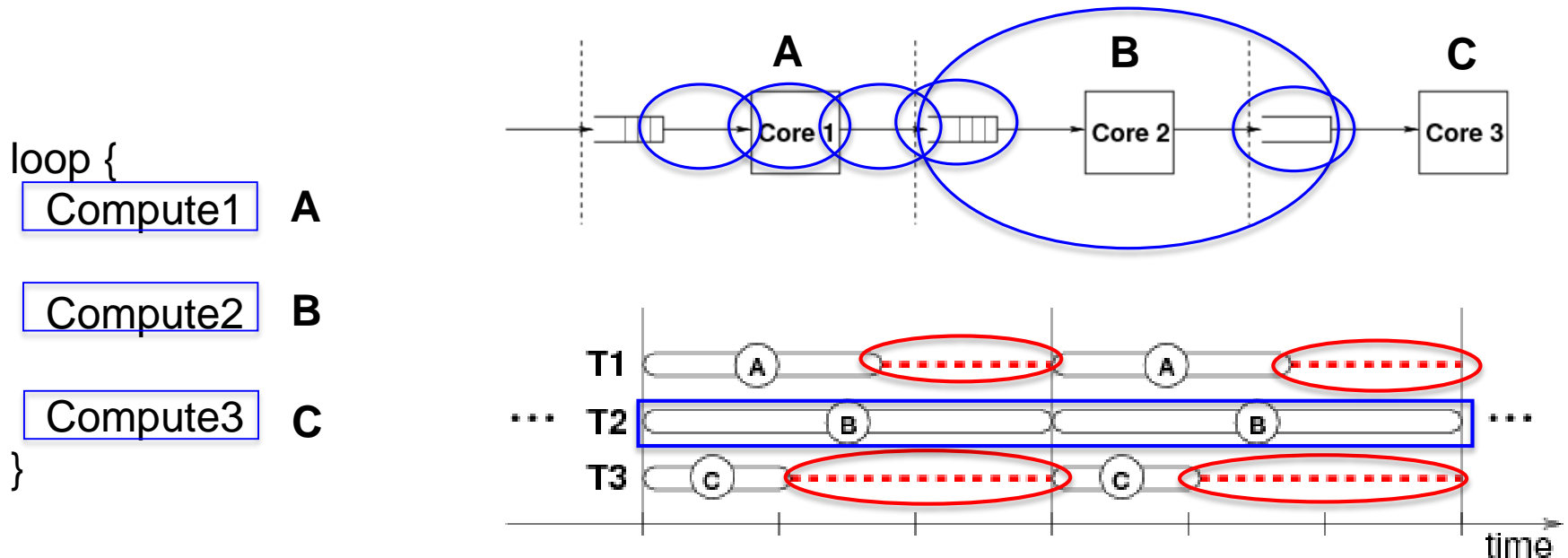
```
loop2 {  
    Compute
```

```
}
```



Stages of Pipelined Programs

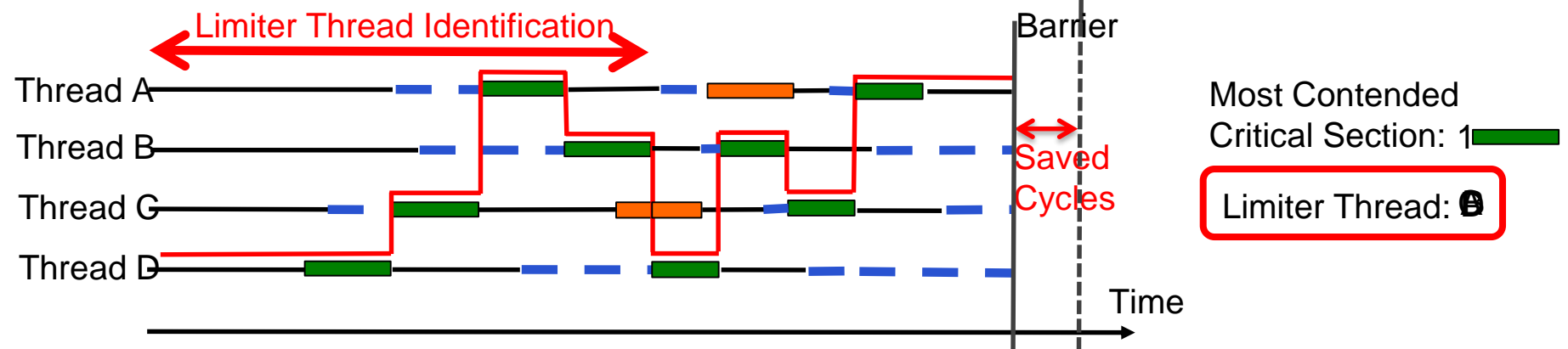
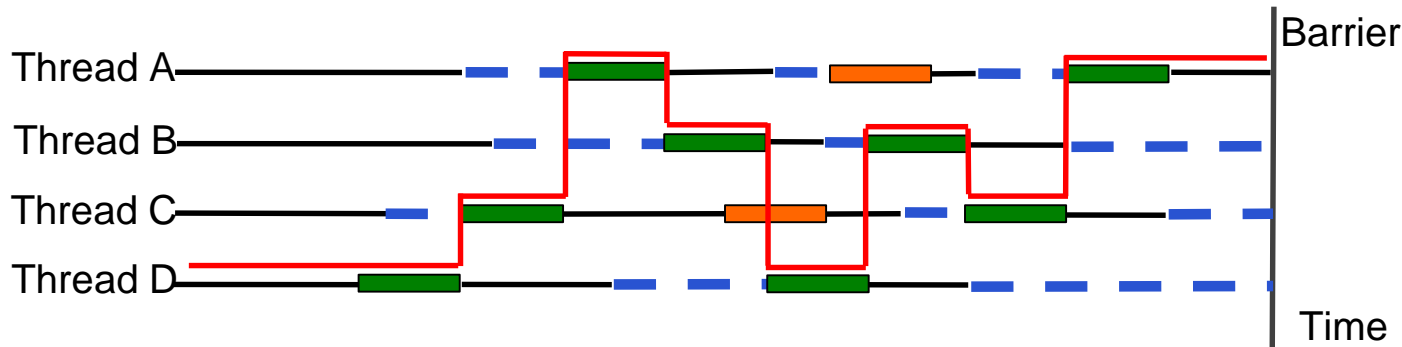
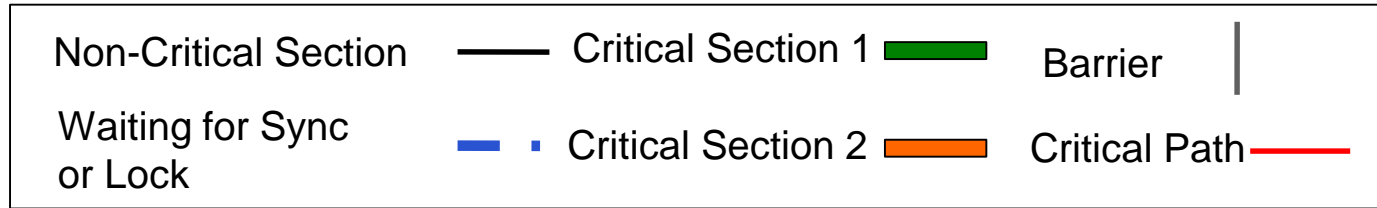
- Loop iterations are statically divided into code segments called *stages*
- Threads execute stages on different cores
- Thread executing the slowest stage is on the critical path



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

Prioritizing Requests from Limiter Threads



More on Parallel Application Memory Scheduling

- Optional reading
- Ebrahimi et al., “Parallel Application Memory Scheduling,” MICRO 2011.
 - http://users.ece.cmu.edu/~omutlu/pub/parallel-memory-scheduling_micro11.pdf

More on DRAM Management and DRAM Controllers

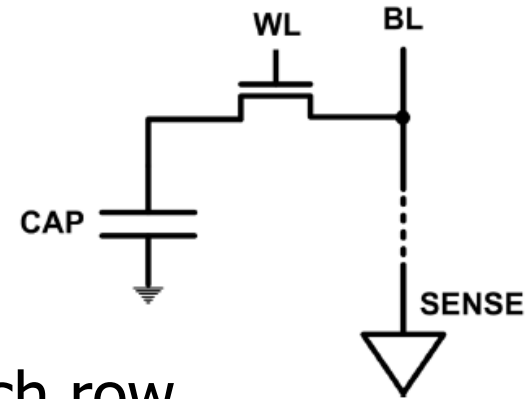
DRAM Power Management

- 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

DRAM Refresh

DRAM Refresh

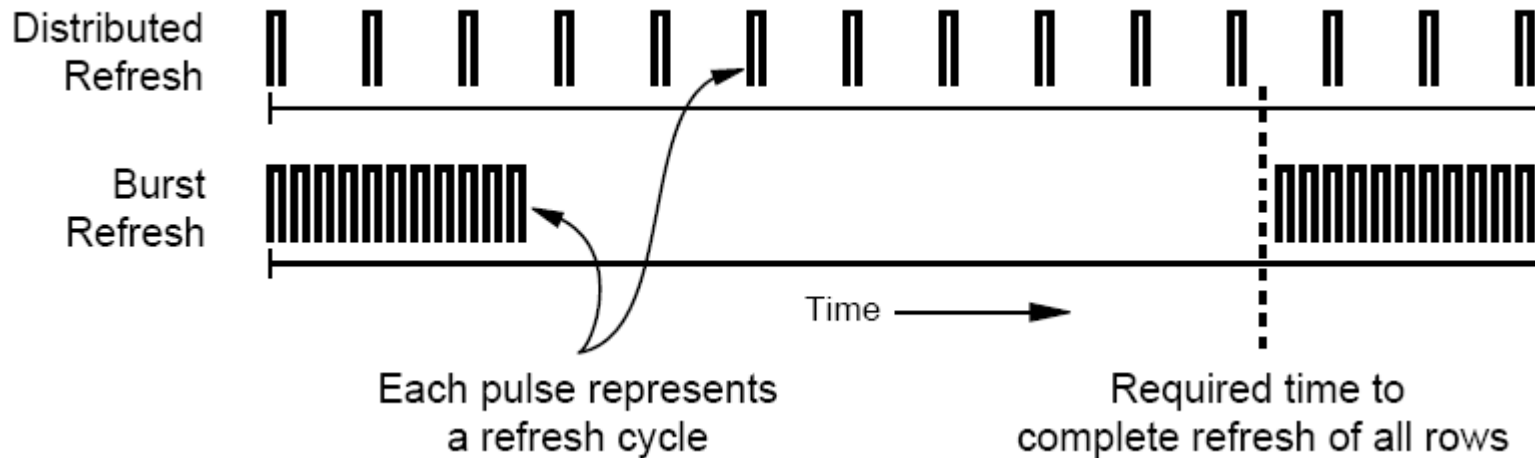
- DRAM capacitor charge leaks over time
- The memory controller needs to refresh each row periodically to restore charge
 - Read and close each row every N ms
 - Typical N = 64 ms
- Downsides of refresh
 - **Energy consumption**: Each refresh consumes energy
 - **Performance degradation**: DRAM rank/bank unavailable while refreshed
 - **QoS/predictability impact**: (Long) pause times during refresh
 - **Refresh rate limits DRAM capacity scaling**



DRAM Refresh: Performance

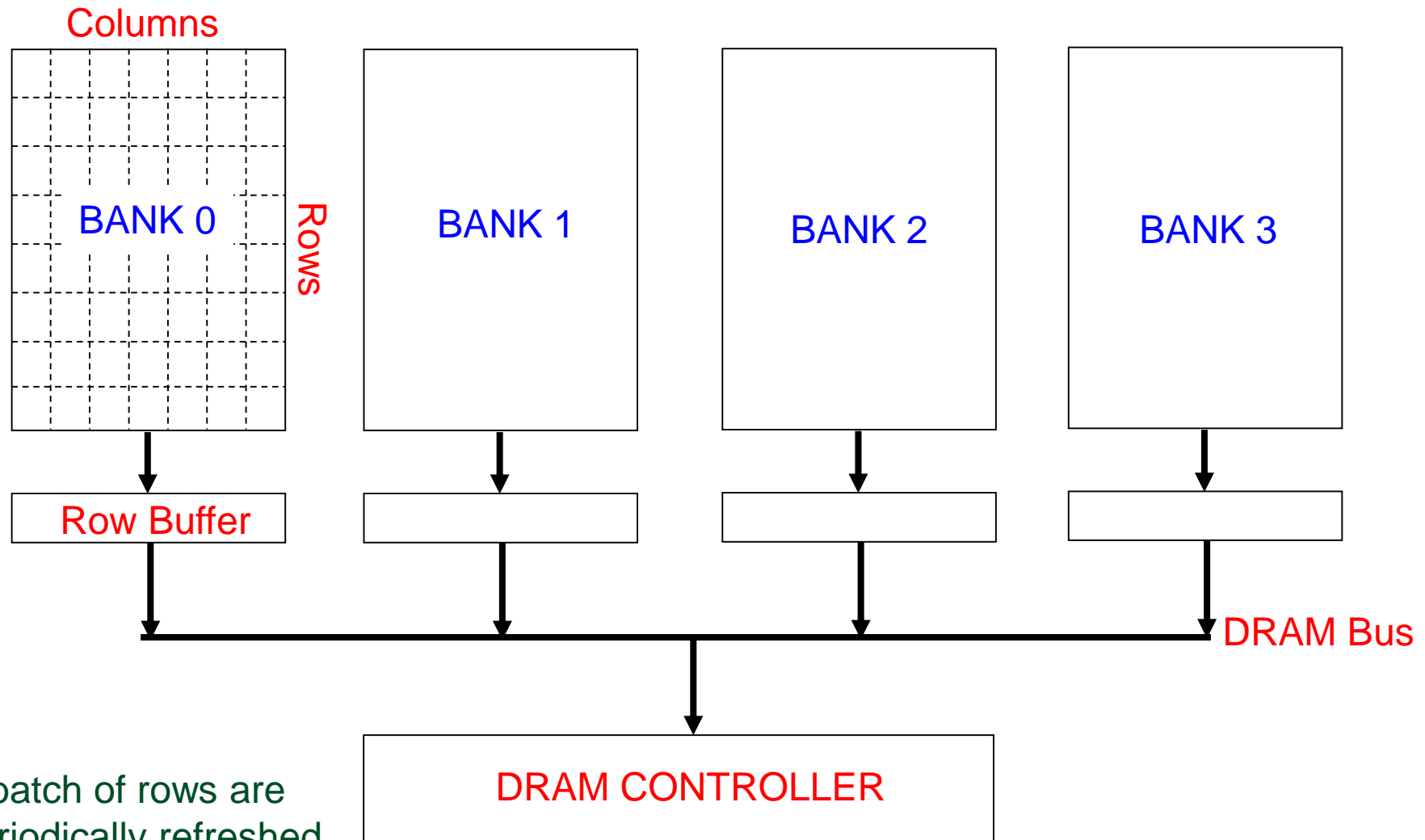
- Implications of refresh on performance
 - DRAM bank unavailable while refreshed
 - Long pause times: If we refresh all rows in burst, every 64ms the DRAM will be unavailable until refresh ends
- **Burst refresh**: All rows refreshed immediately after one another
- **Distributed refresh**: Each row refreshed at a different time, at regular intervals

Distributed Refresh



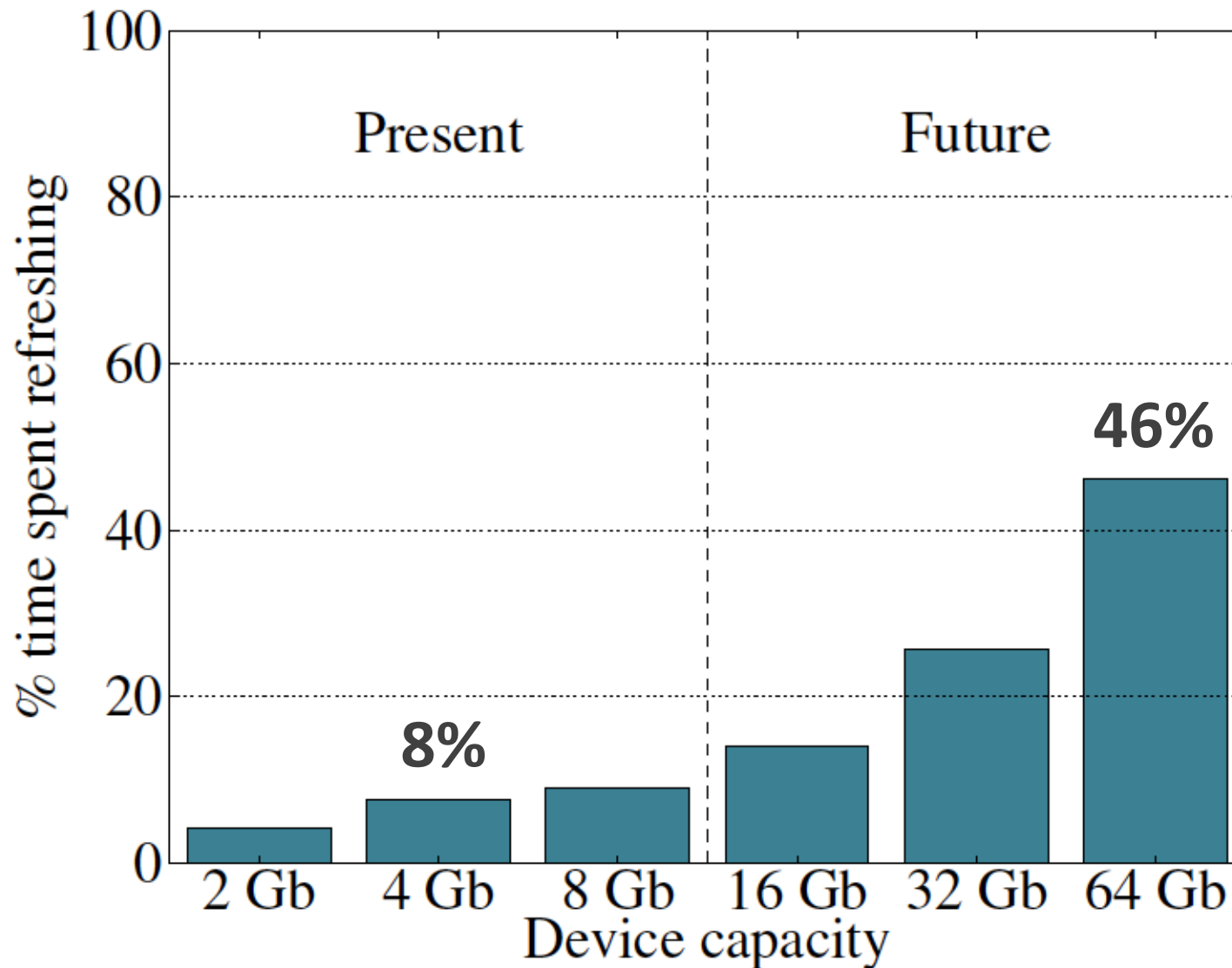
- Distributed refresh eliminates long pause times
- How else can we reduce the effect of refresh on performance/QoS?
- Does distributed refresh reduce refresh impact on energy?
- Can we reduce the number of refreshes?

Refresh Today: Auto Refresh

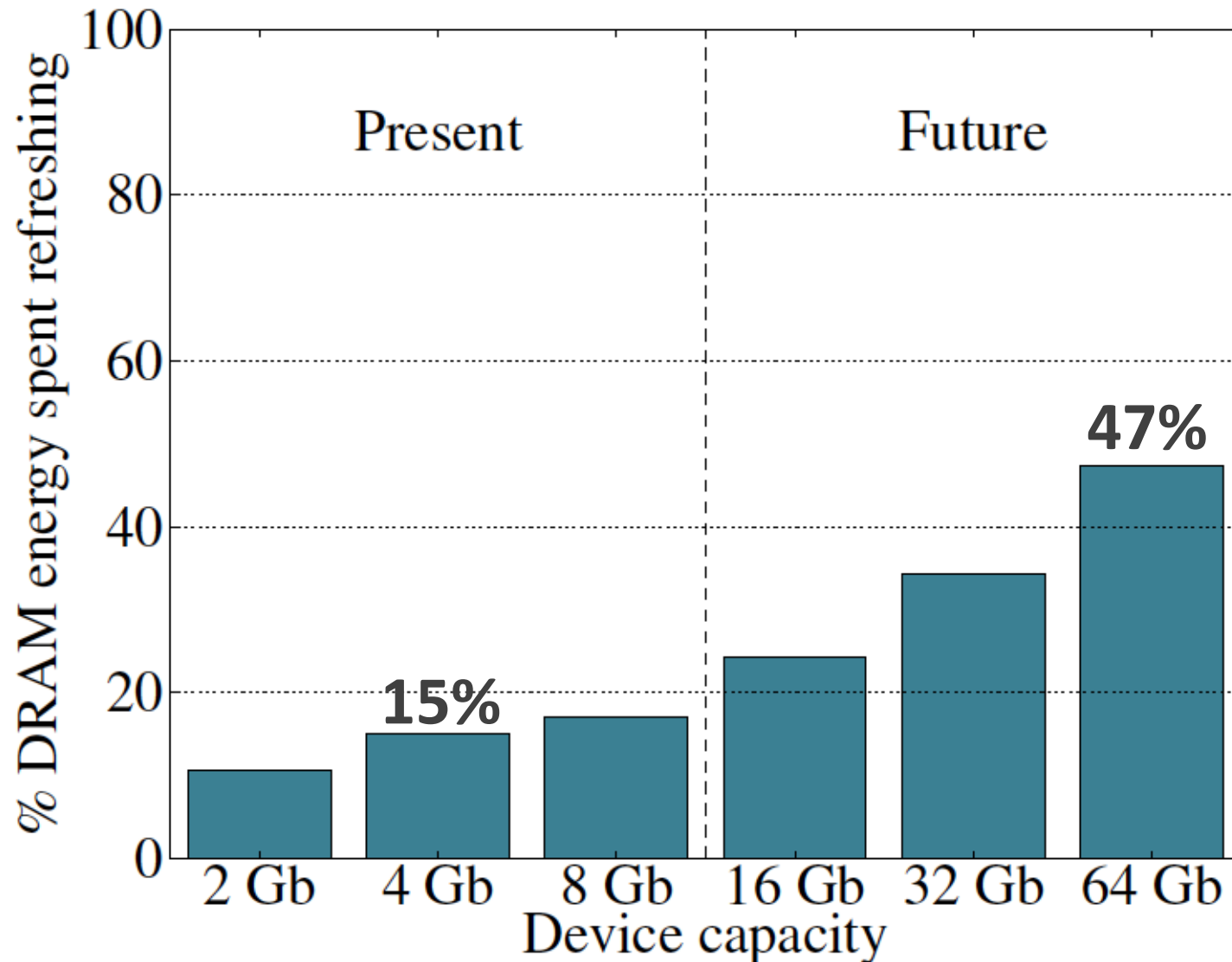


A batch of rows are periodically refreshed via the auto-refresh command

Refresh Overhead: Performance

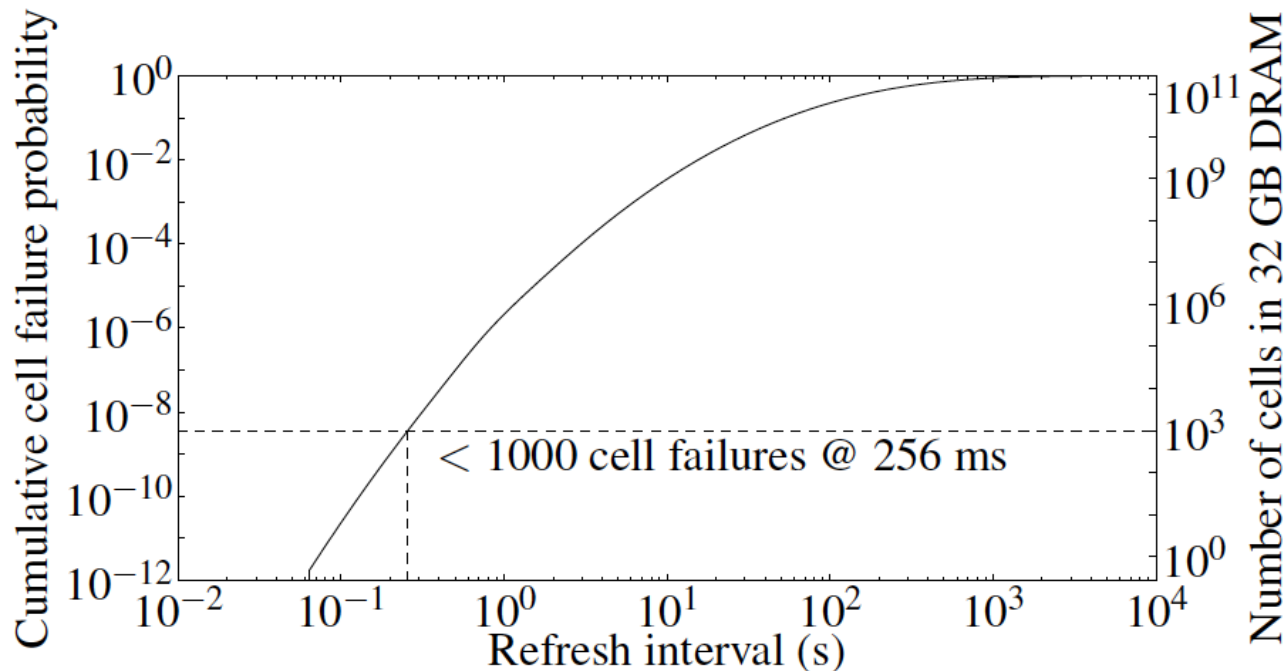


Refresh Overhead: Energy



Problem with Conventional Refresh

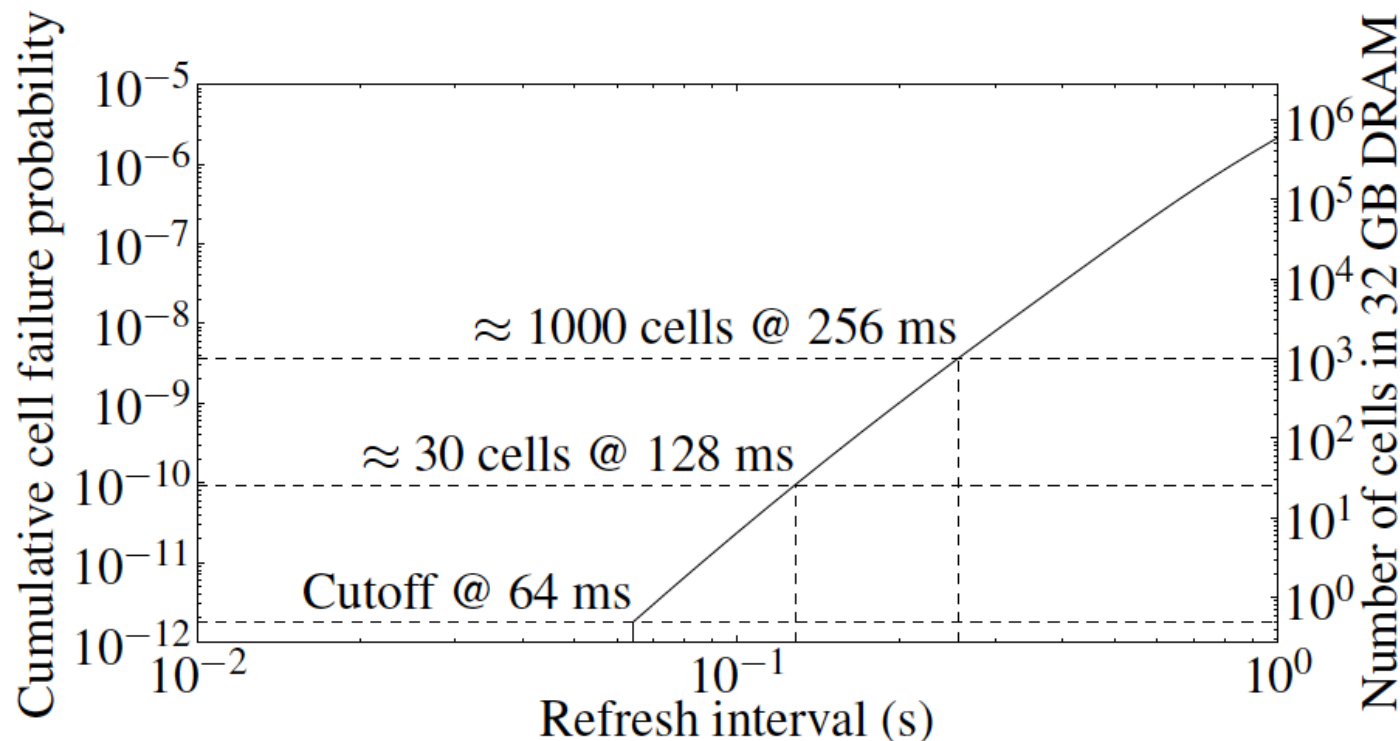
- Today: Every row is refreshed at the same rate



- 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



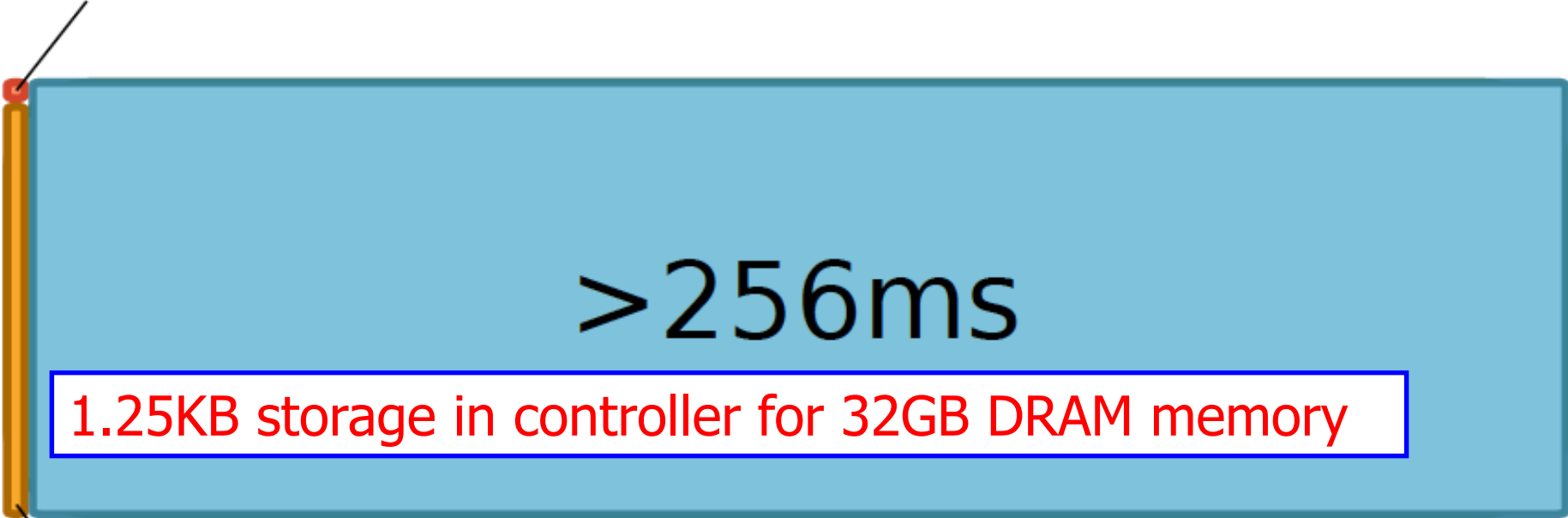
- Can we exploit this to reduce refresh operations at low cost?

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



>256ms

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

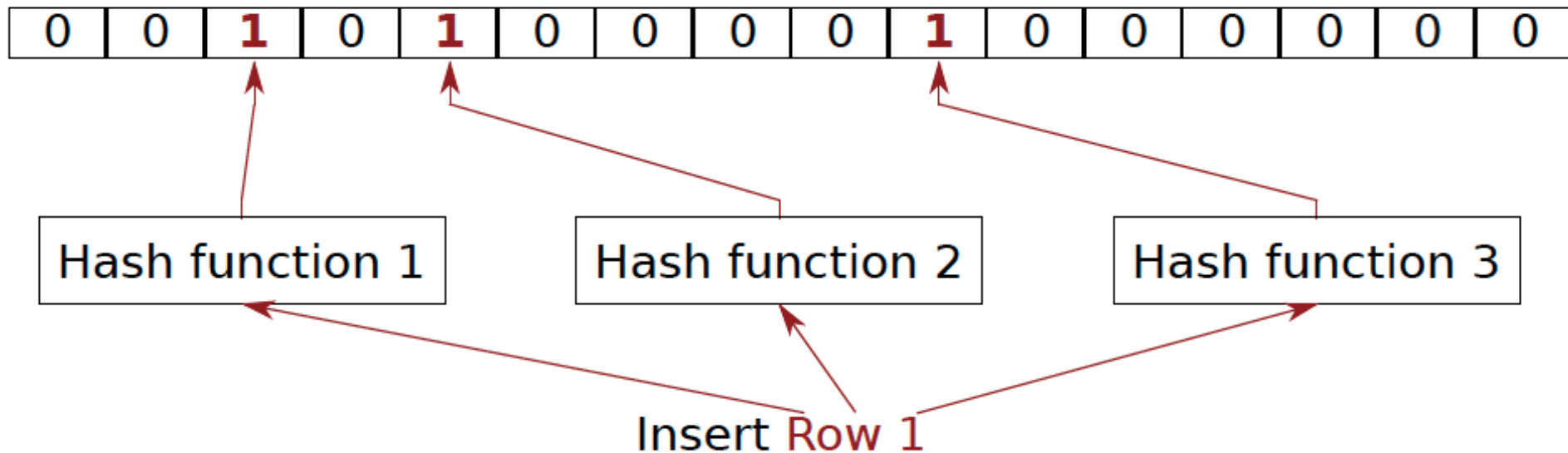
- How to efficiently and scalably store rows into retention time bins?
- Use Hardware Bloom Filters [Bloom, CACM 1970]

Bloom 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

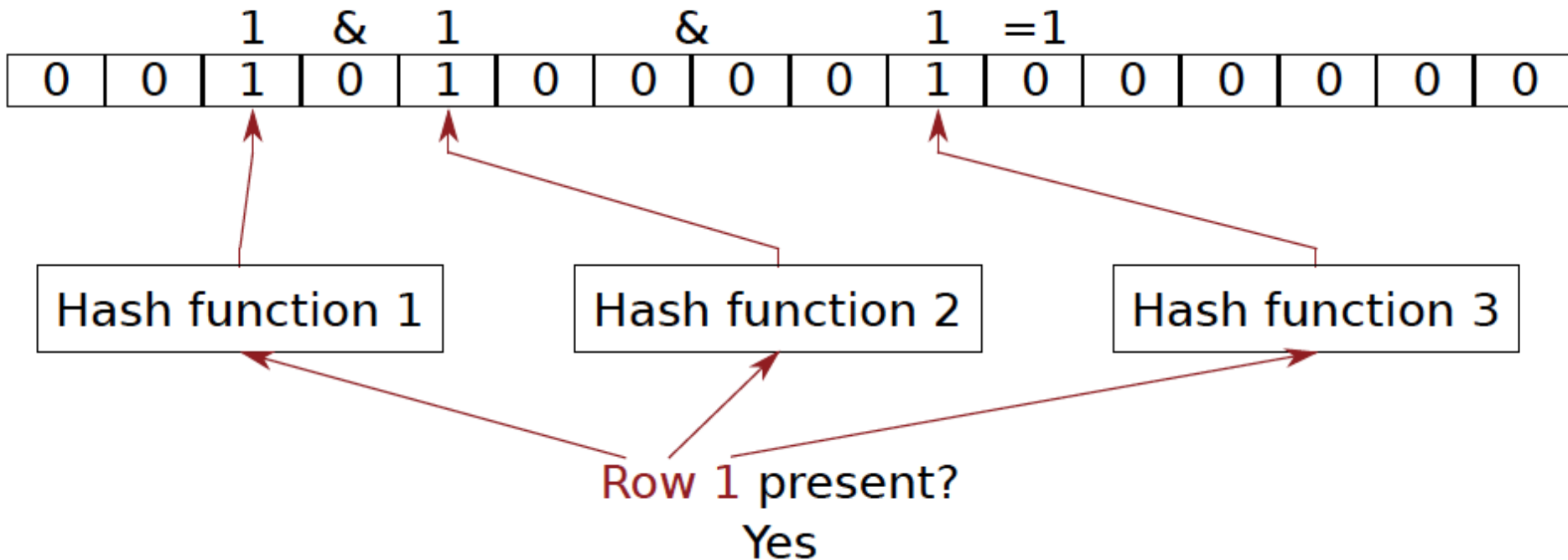
Bloom Filter Operation Example

Example with 64-128ms bin:



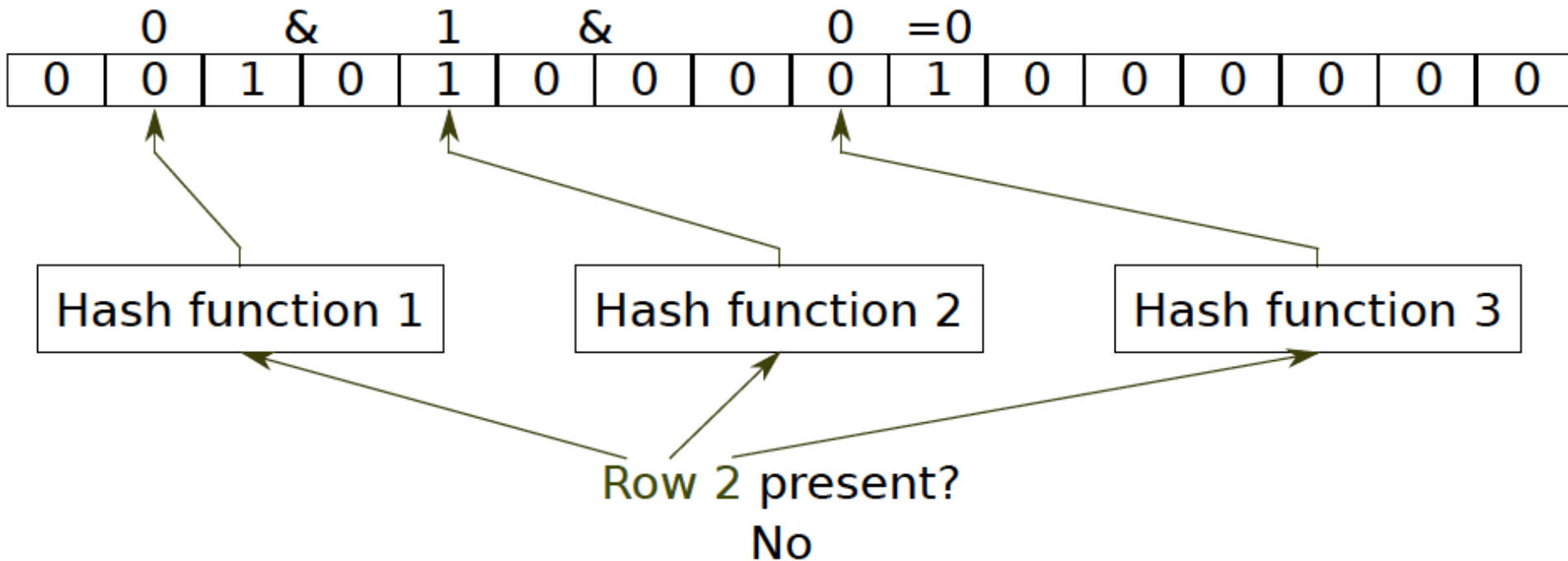
Bloom Filter Operation Example

Example with 64-128ms bin:



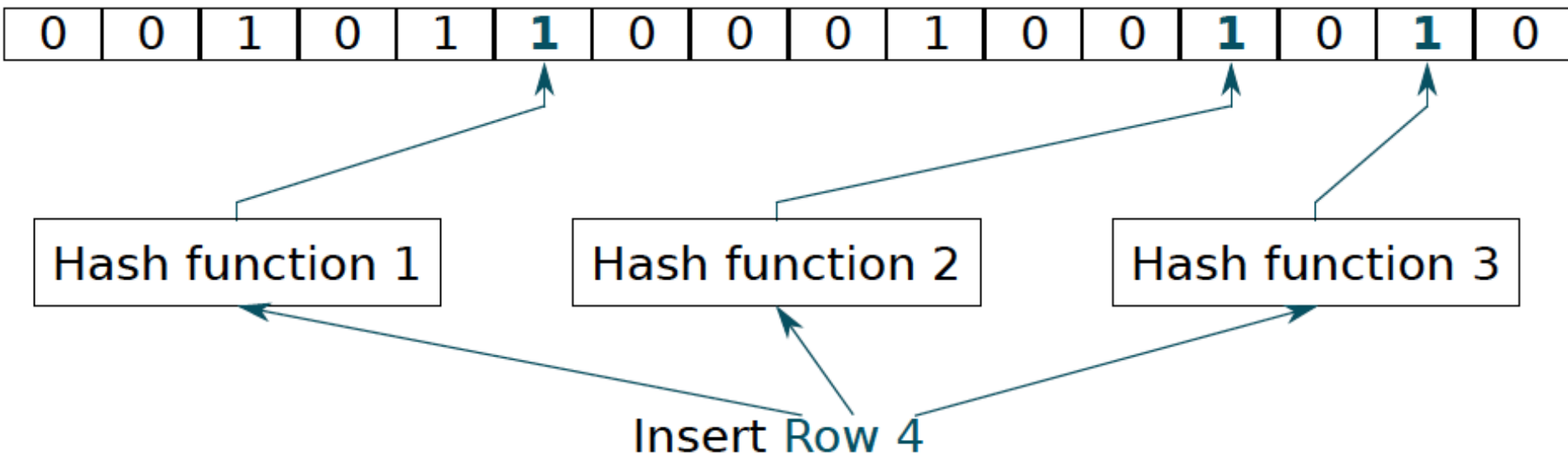
Bloom Filter Operation Example

Example with 64-128ms bin:



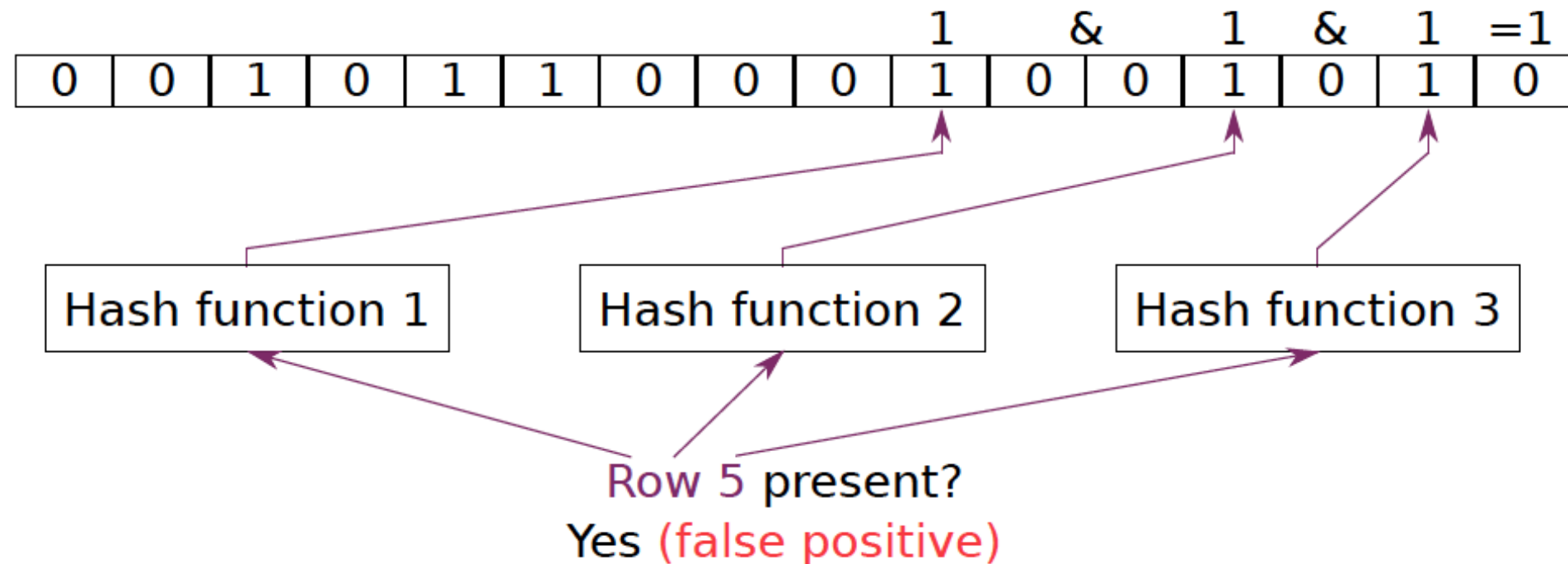
Bloom Filter Operation Example

Example with 64-128ms bin:



Bloom Filter Operation Example

Example with 64–128ms bin:



Bloom Filters

Space/Time Trade-offs in Hash Coding with Allowable Errors

BURTON H. BLOOM

Computer Usage Company, Newton Upper Falls, Mass.

In such applications, it is envisaged that overall performance could be improved by using a smaller core resident hash area in conjunction with the new methods and, when necessary, by using some secondary and perhaps time-consuming test to "catch" the small fraction of errors associated with the new methods. An example is discussed which illustrates possible areas of application for the new methods.

In this paper trade-offs among certain computational factors in hash coding are analyzed. The paradigm problem considered is that of testing a series of messages one-by-one for membership in a given set of messages. Two new hash-coding methods are examined and compared with a particular conventional hash-coding method. The computational factors considered are the size of the hash area (space), the time required to identify a message as a nonmember of the given set (reject time), and an allowable error frequency.

Bloom Filters: Pros and Cons

■ Advantages

- + Enables **storage-efficient** representation of set membership
- + Insertion and testing for set membership (presence) are **fast**
- + **No false negatives**: If Bloom Filter says an element is not present in the set, the element must not have been inserted
- + Enables **tradeoffs** between **time** & **storage efficiency** & **false positive rate** (via sizing and hashing)

■ Disadvantages

- **False positives**: An element may be deemed to be present in the set by the Bloom Filter but it may never have been inserted

Not the right data structure when you cannot tolerate false positives

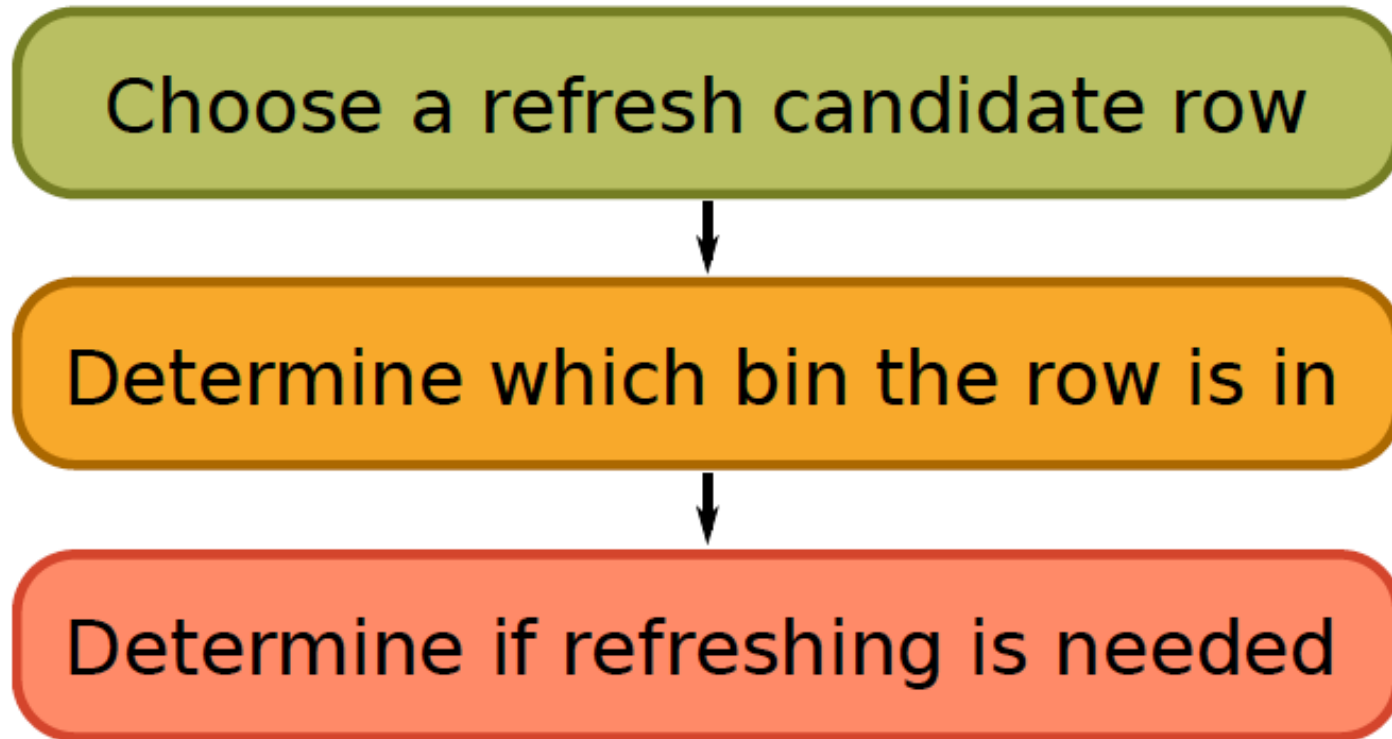
Benefits of Bloom Filters as Refresh Rate Bins

- **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

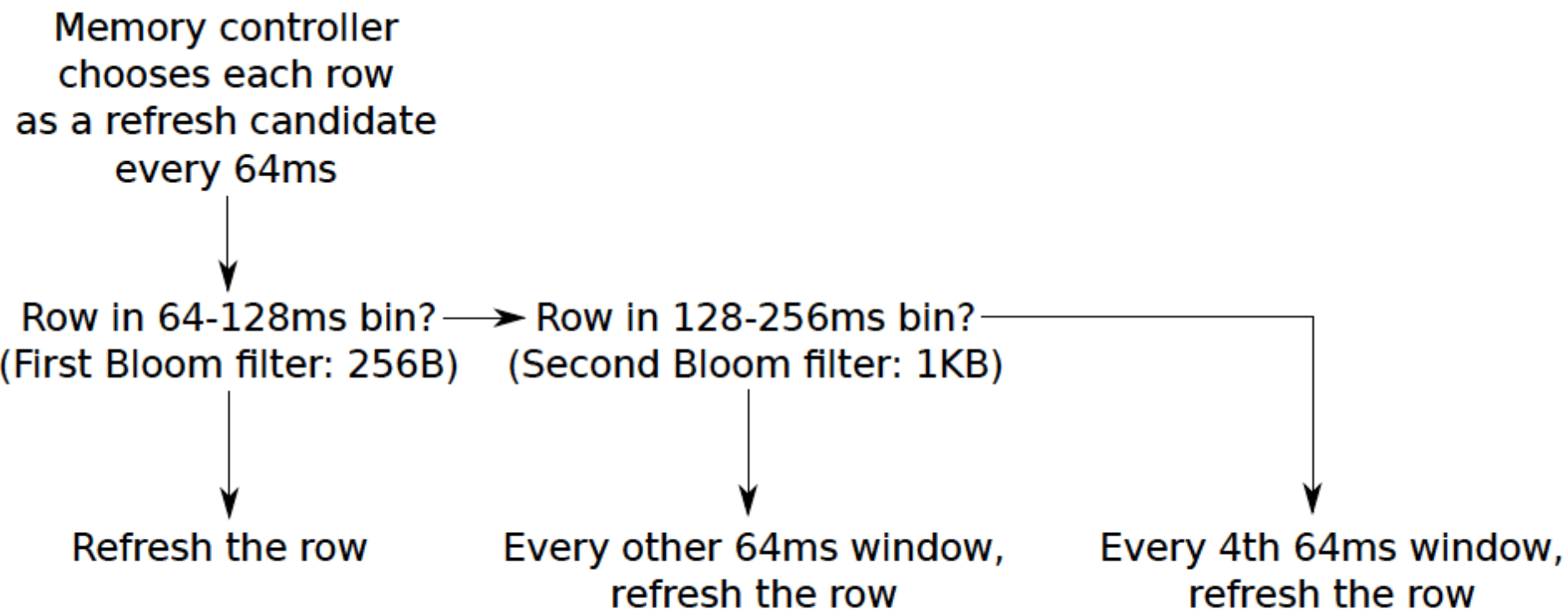
Use of Bloom Filters in Hardware

- 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)

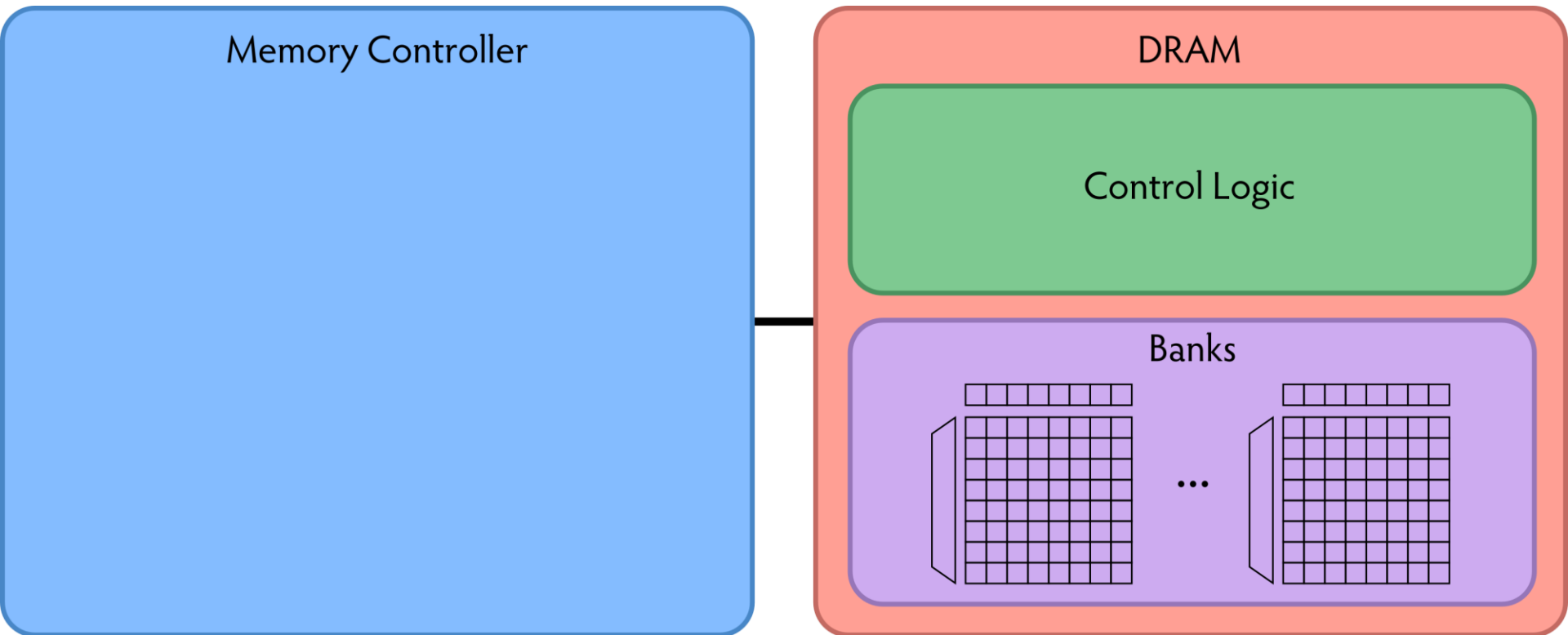


3. Refreshing (RAIDR Refresh Controller)



Liu et al., “[RAIDR: Retention-Aware Intelligent DRAM Refresh](#),” ISCA 2012.

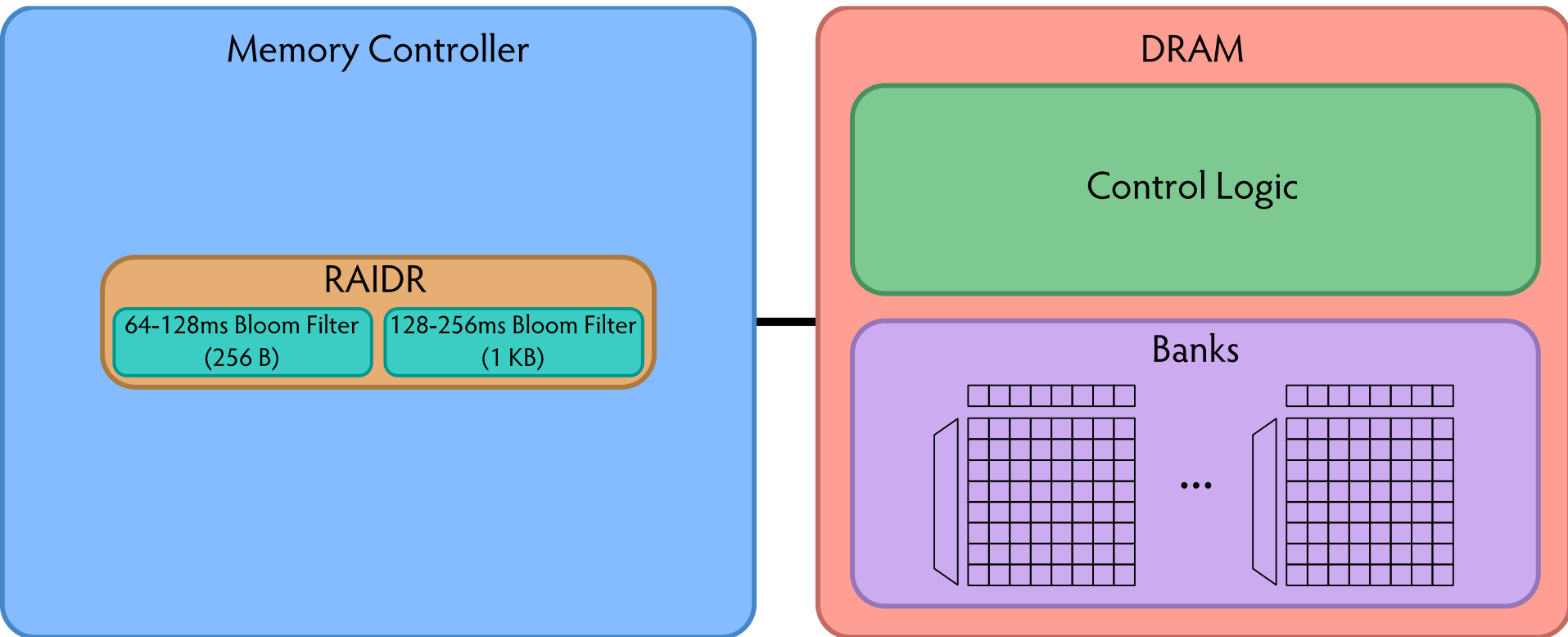
RAIDR: Baseline Design



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

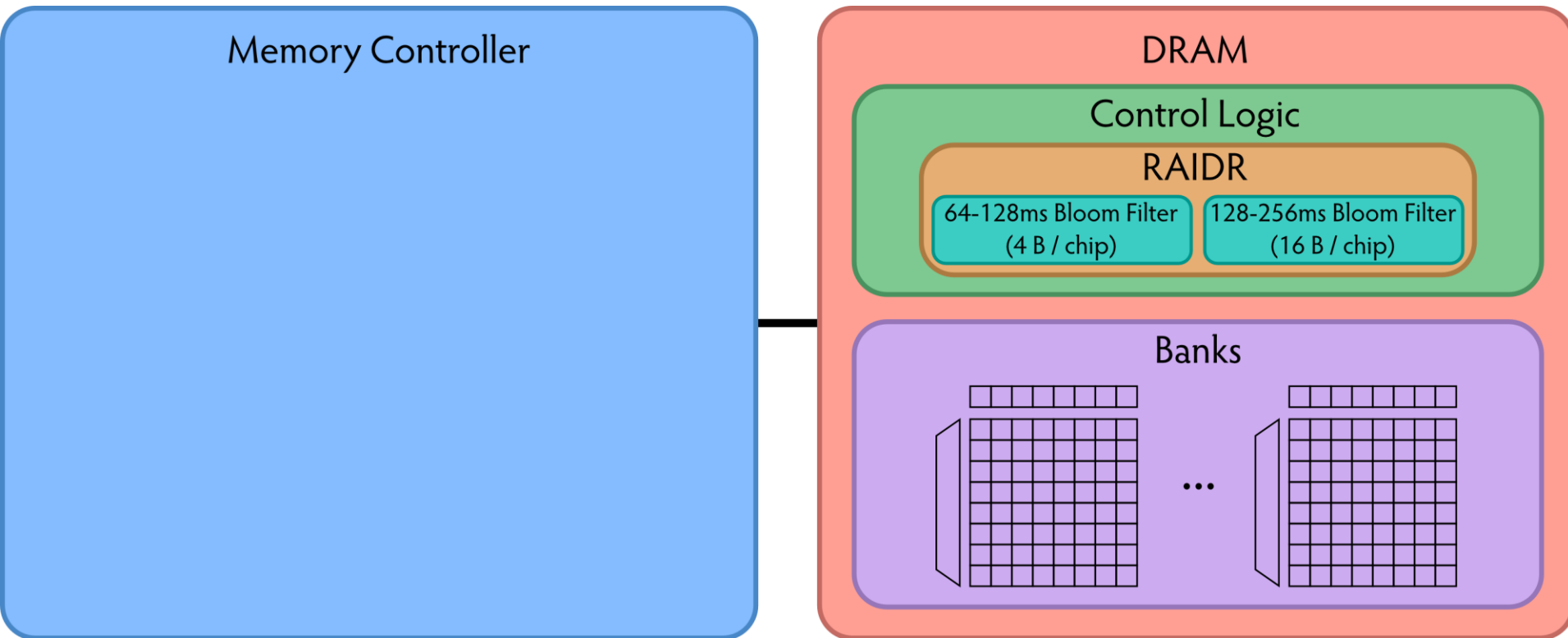
RAIDR can be implemented in either the controller or DRAM

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



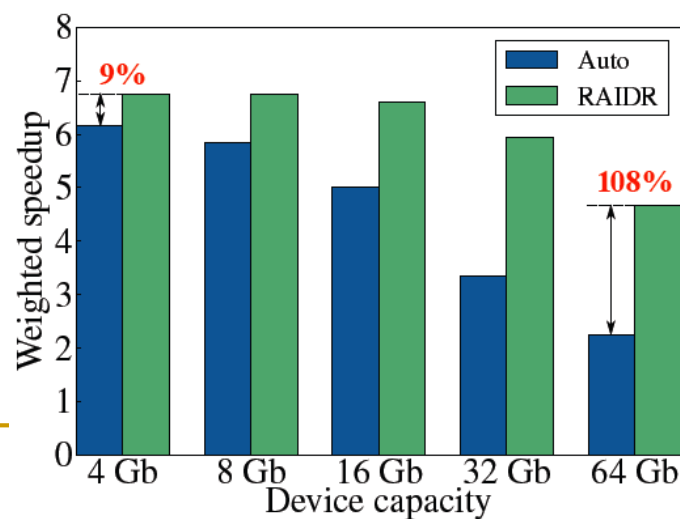
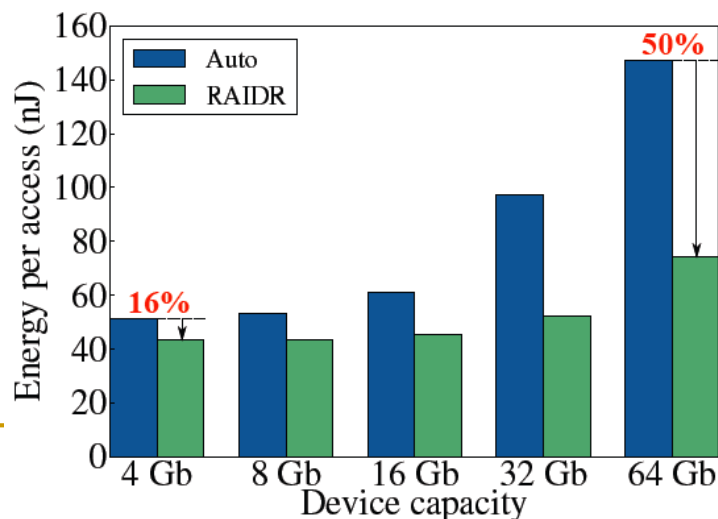
Overhead of RAIDR in DRAM chip:

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.

More Readings on DRAM Refresh

- Liu et al., “An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms,” ISCA 2013.
 - http://users.ece.cmu.edu/~omutlu/pub/dram-retention-time-characterization_isca13.pdf
- Chang+, “Improving DRAM Performance by Parallelizing Refreshes with Accesses,” HPCA 2014.
 - http://users.ece.cmu.edu/~omutlu/pub/dram-access-refresh-parallelization_hpca14.pdf