Paper Discussion for 18-742

Kevin Hsieh

Sep 9, 2014

Carnegie Mellon

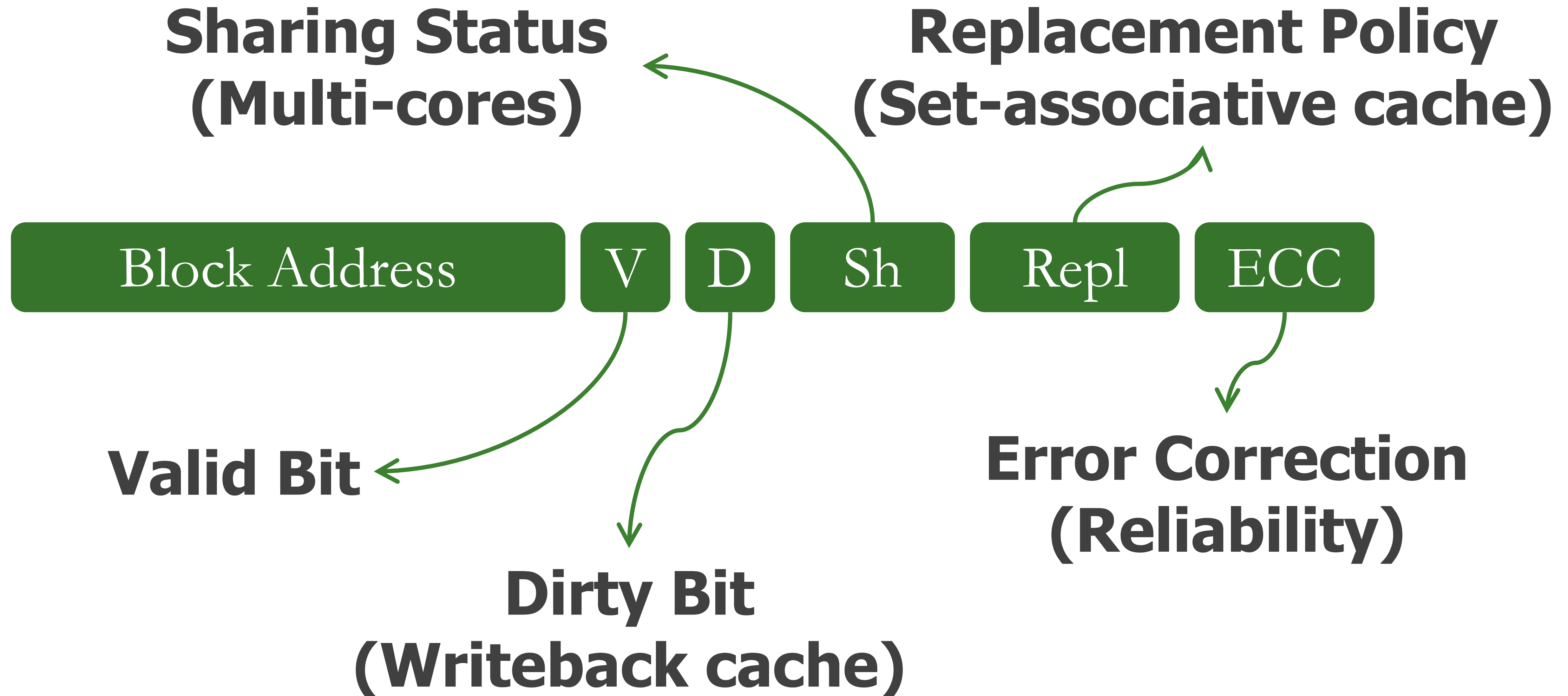
Paper to Discuss (1/3)

- Seshadri et al., "The Dirty-Block Index", ISCA 2014.
  - Introducing a cache organization to achieve better performance and cost.
- Ausavarungnirun et al., "Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems", ISCA 2012.
  - Introducing a memory controller which is simpler and works better in heterogeneous system.
- Chang et al., "Improving DRAM Performance by Parallelizing Refreshes with Accesses", HPCA 2014.
  - Introducing a few parallelism schemes for refresh commands.

Background and Problem

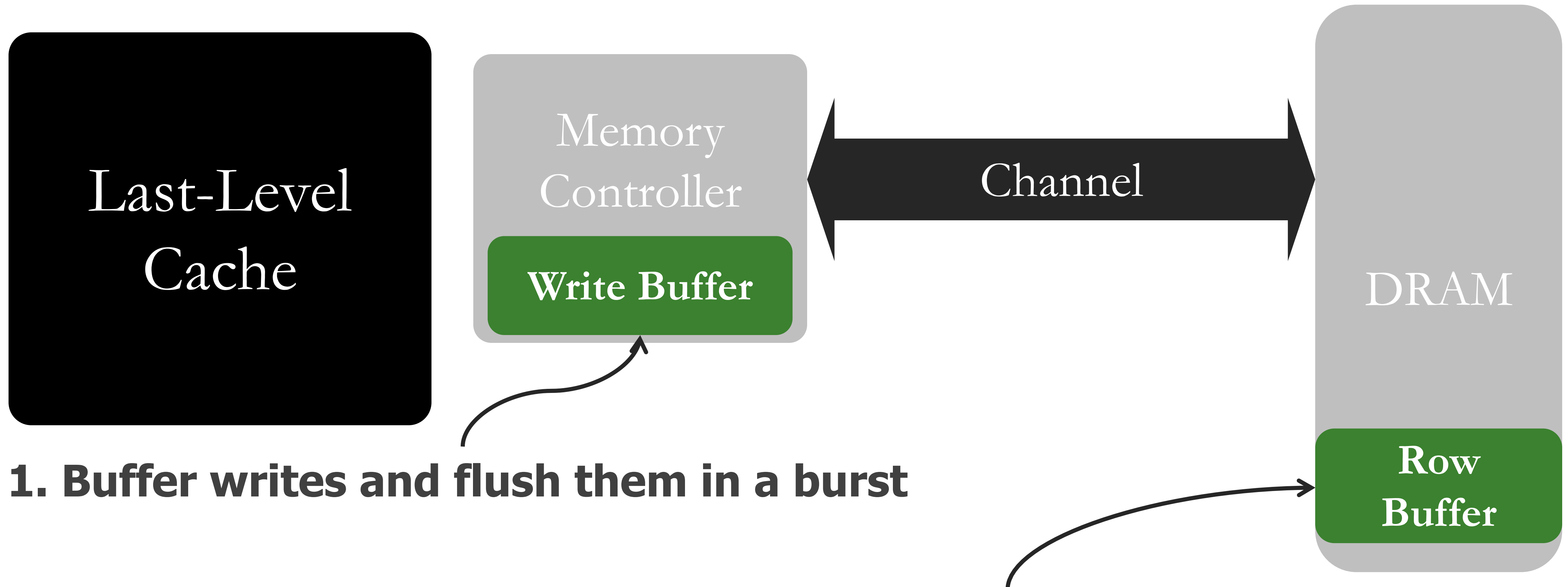
- Traditionally, the metadata of the cache is organized according to cache block (or cache line)
  - Each cache block has a corresponding cache metadata, which maintains all the attributes for this cache block (valid, dirty, tag address, etc.)
  - It's intuitive, simple, and scalable.
- However, there are shortcomings
  - All metadata query is relatively expensive
  - It makes some cache improvement difficult to implement
    - DRAM-Aware Writeback [TR-HPS-2010-2]
    - Bypassing Cache Lookups [HPCA 2003, PACT 2012]

Block-Oriented Metadata Organization (Vivek's Slide)



DRAM-Aware Writeback (Vivek's slide)

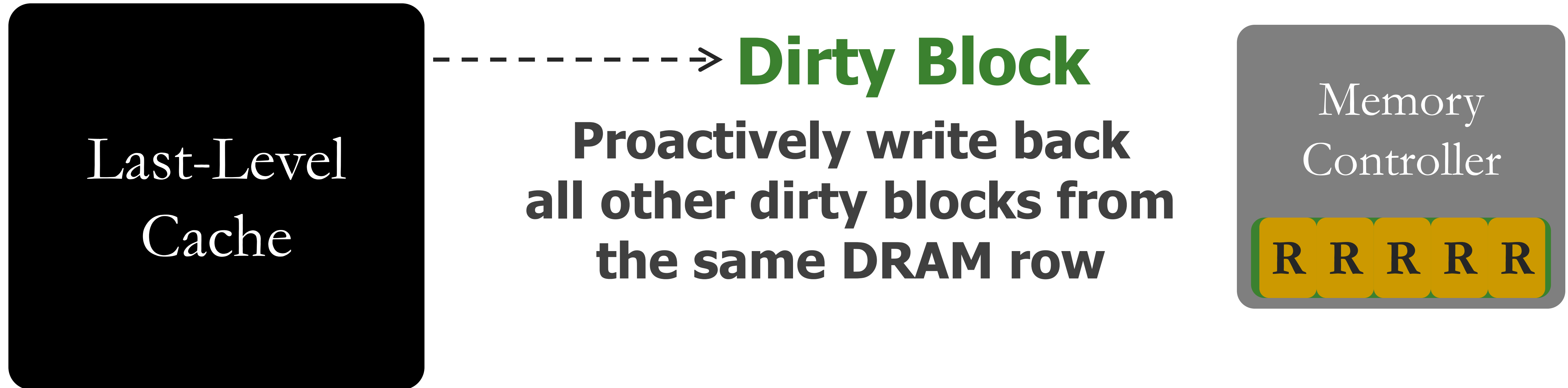
Virtual Write Queue [ISCA 2010], DRAM-Aware Writeback [TR-HPS-2010-2]



2. Row buffer hits are faster and more efficient than row misses

DRAM-Aware Writeback (Vivek's slide)

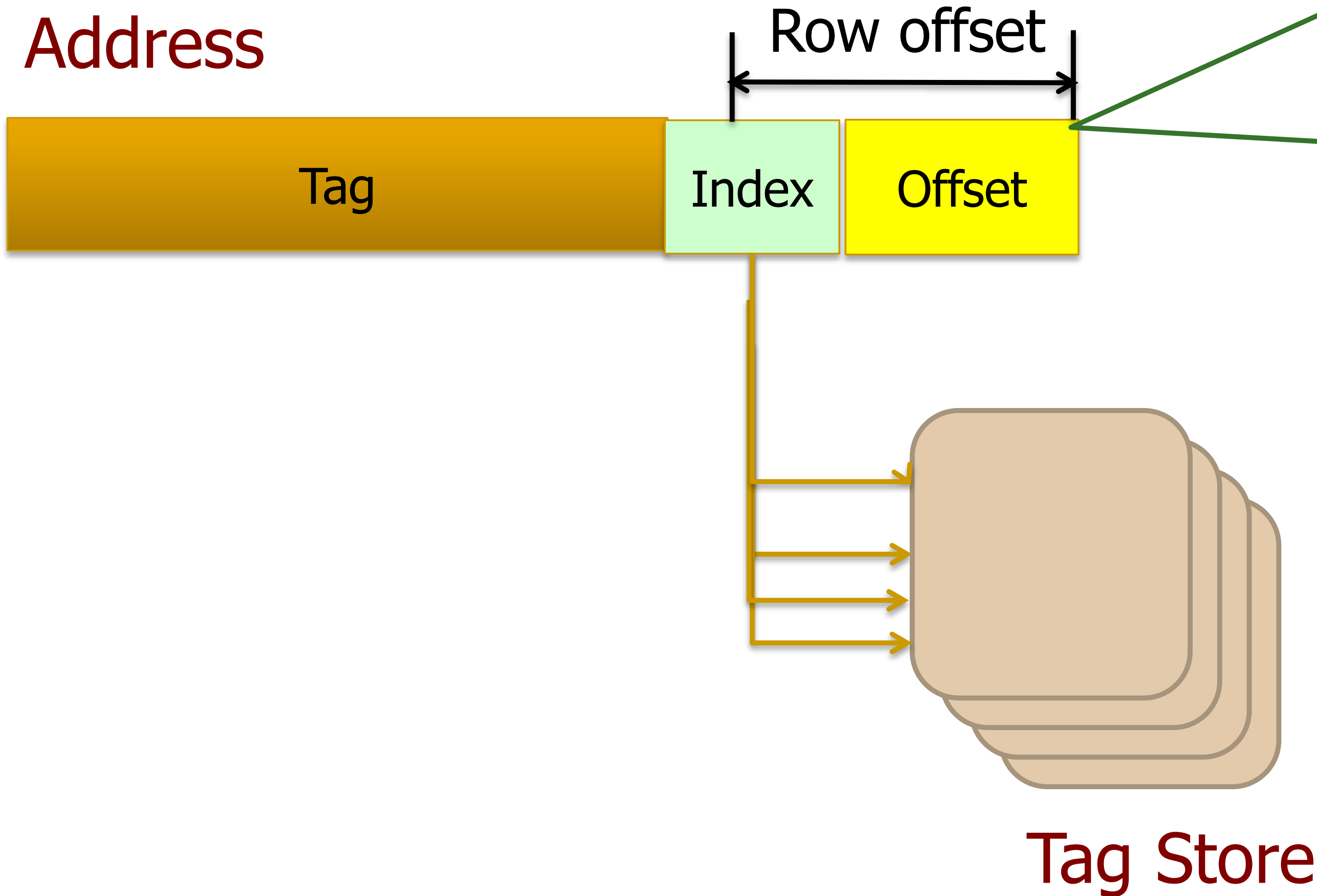
Virtual Write Queue [ISCA 2010], DRAM-Aware Writeback [TR-HPS-2010-2]



Significantly increases the DRAM write row hit rate

Get all dirty blocks of DRAM row 'R'

Query to Tag Store for a DRAM row

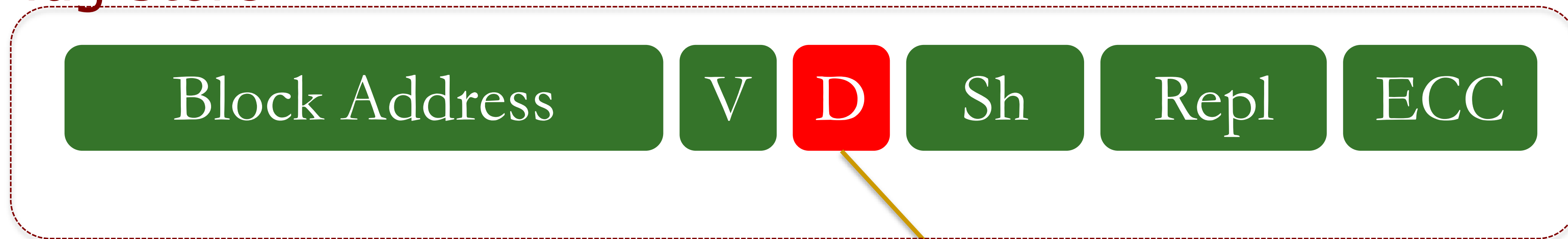


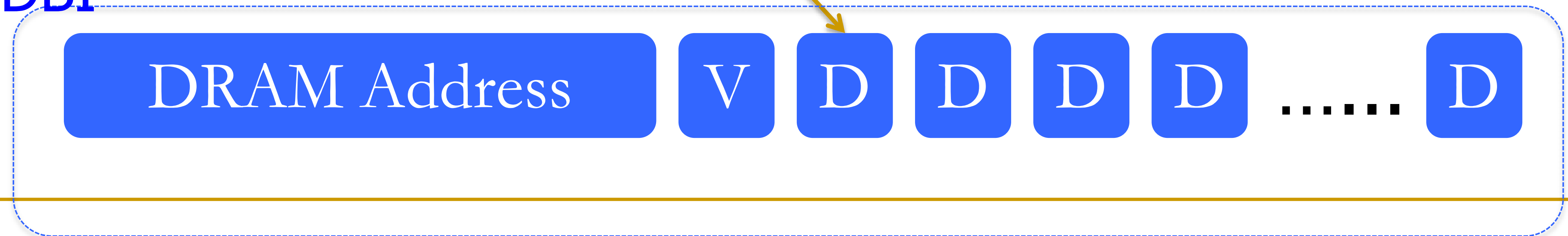
A DRAM row can be distributed into a lot of sets.

It is very inefficient if we want to query all cache blocks for a certain DRAM row

The Dirty-Block Index

- Key Idea
  - Decouple dirty bits from main tag store and indexed them by DRAM row. This separated structure makes query for dirty bit (especially in terms of DRAM row) much more efficient.





Benefit of DBI

- **DRAM-aware writeback**
  - With DBI, a single query can know all the dirty block in a DRAM row
  - No more tag store contention

- **Bypassing cache lookups**
  - The idea was to bypass cache lookup if it's very likely to miss. However it must not bypass dirty cache block.
  - With DBI, it can check the dirty status much faster to seamlessly enable this optimization.

- **Reducing ECC overhead**
  - The idea was only dirty block requires error correction, others only requires detection.
  - With DBI, it's much easier to track the error correction codes

Operation of DBI

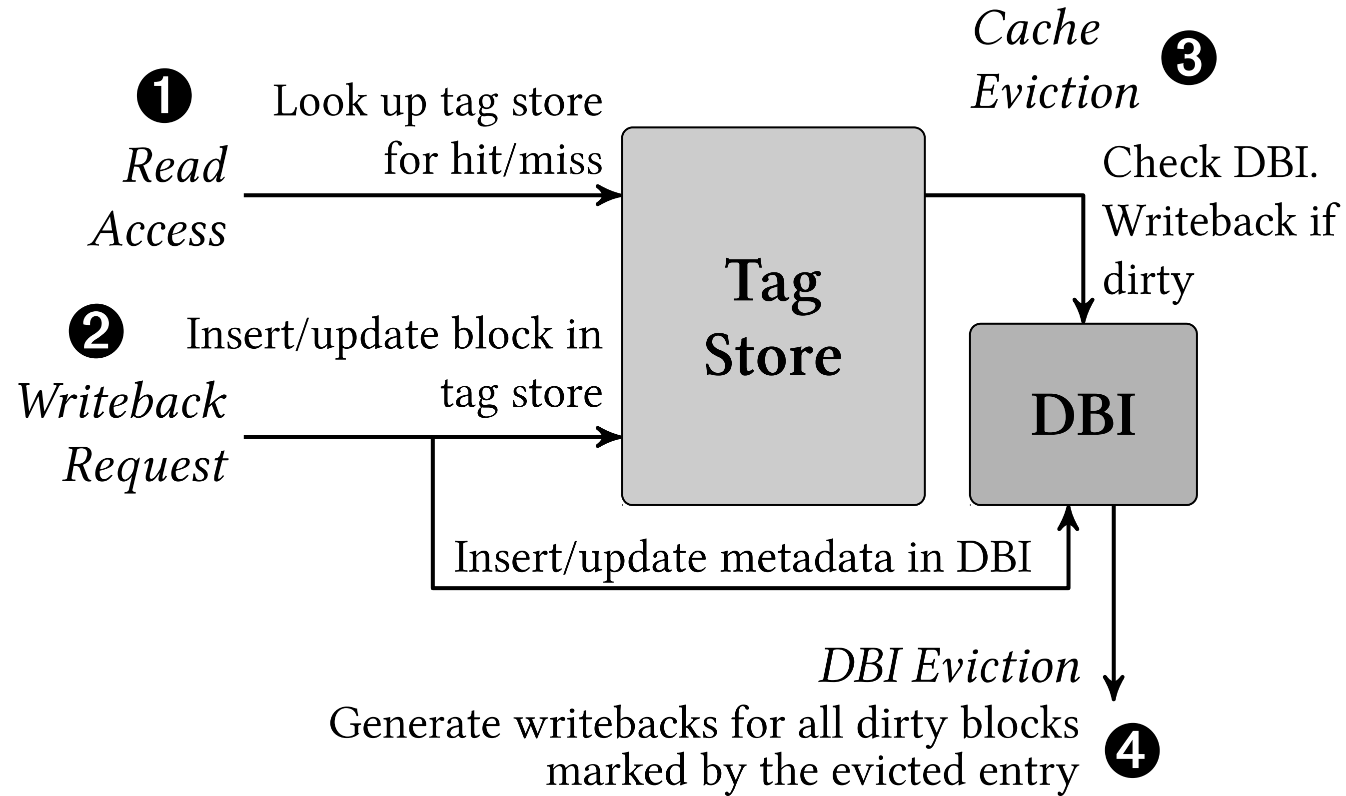


Figure 2: Operation of a cache with DBI

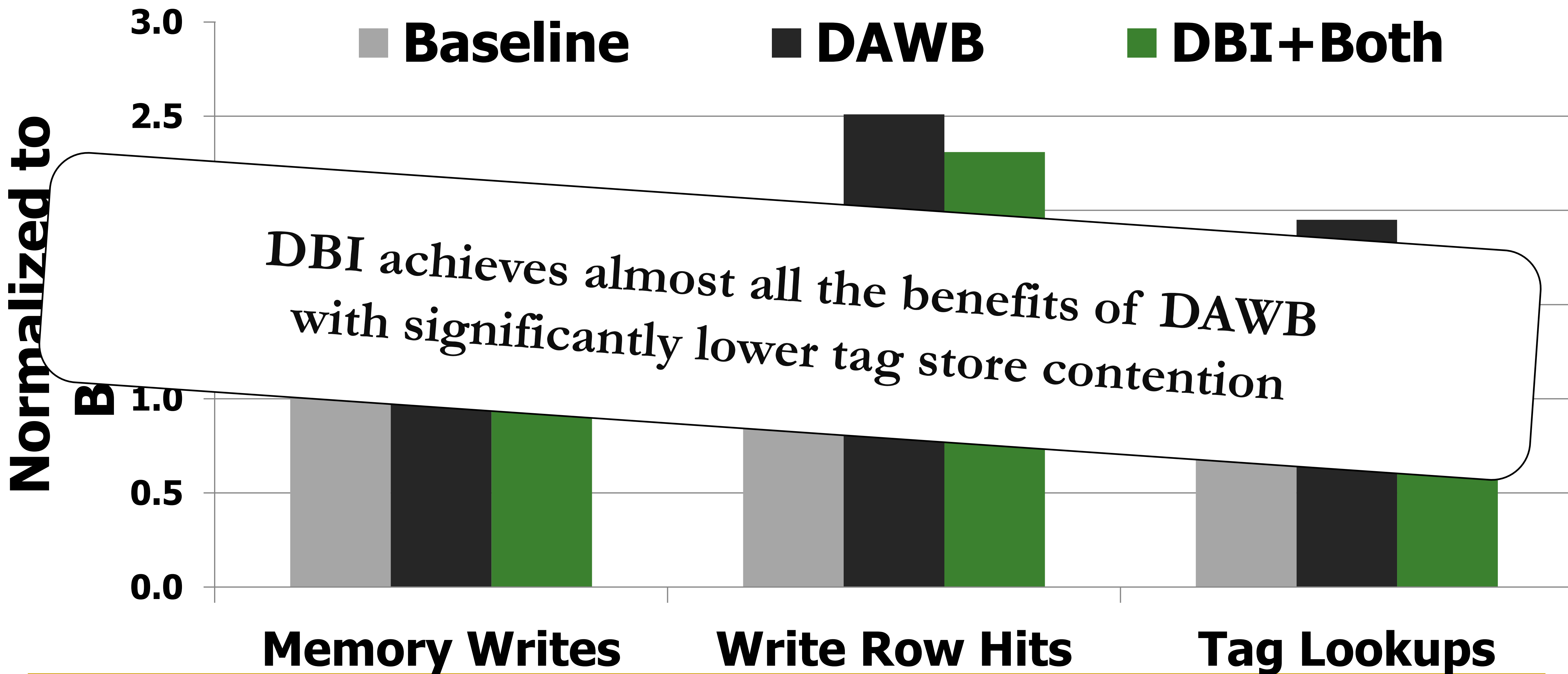
Evaluated Mechanism

- Baseline (LRU, Least Recently Used)
- TA-DIP (Thread-aware dynamic insertion policy)
- DAWB (DRAM aware writeback)
- VWQ (Virtual write queue)
- CLB (cache lookup bypass)
- DBI
  - No optimization
  - +DAWB
  - +CLB
  - +AWB+DBI

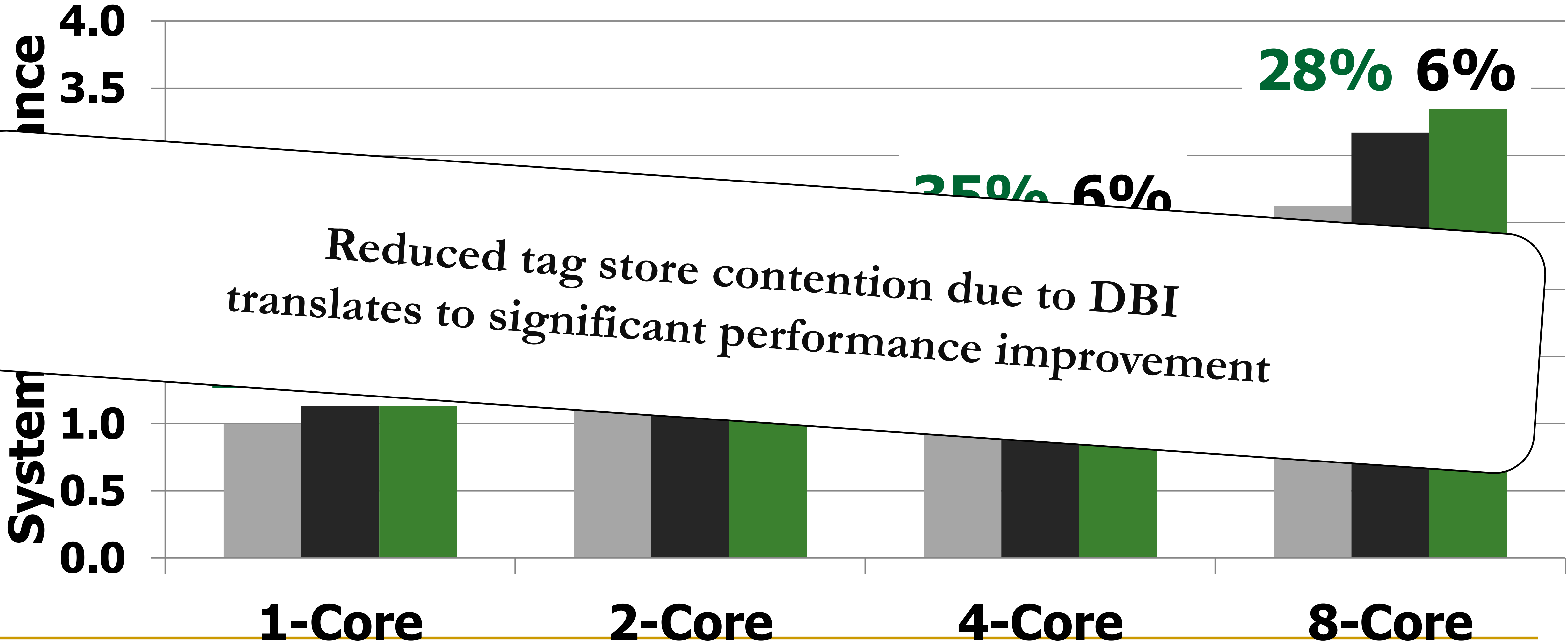
System Configuration

Processor	1-8 cores, 2.67 GHz, Single issue, Out-of-order, 128 entry instruction window
L1 Cache	Private, 32KB, 2-way set-associative, tag store latency = 2 cycles, data store latency = 2 cycles, parallel tag and data lookup, LRU replacement policy, number of MSHRs = 32
L2 Cache	Private, 256KB, 8-way set-associative, tag store latency = 12 cycles, data store latency = 14 cycles, parallel tag and data lookup, LRU replacement policy
L3 Cache	Shared, 2MB/core. 1/2/4/8-core, 16/32/32/32-way set-associative, tag store latency = 10/12/13/14 cycles, data store latency = 24/29/31/33 cycles, serial tag and data lookup, LRU replacement policy
DBI	Size (α) = $^1/_4$, granularity = 64, associativity = 16, latency = 4 cycles, LRW replacement policy (Section 4.3)
DRAM Controller	Open row, row interleaving, FR-FCFS scheduling policy [45, 60], 64-entry write buffer, drain when full policy [27]
DRAM and Bus	DDR3-1066 MHz [20], 1 channel, 1 rank, 8 banks, 8B-wide data bus, burst length = 8, 8KB row buffer

Effect on Writes and Tag Lookups (Vivek's slide)



System Performance (Vivek's slide)



Area and Power

Bit cost reduction of cache

DBI Size (α)	Without ECC		With ECC	
	Tag Store	Cache	Tag Store	Cache
$^1/_4$	2%	0.1%	**44%***	**7%**
$^1/_2$	1%	0.0%	26%*	4%

Power increase in cache (not including memory)

Cache size	2 MB	4 MB	8 MB	16MB
Static	0.12%	0.21%	0.21%	0.22%
Dynamic	4%	1%	1%	2%

DBI design consideration

- Major design considerations
  - DBI size (number of total blocks tracked by DBI, α, in ratio of total cache blocks)
  - DBI granularity (number of blocks tracked by a single DBI entry)
  - DBI replacement policy
    - 5 policies evaluated, better policy gives better performance

Granularity		16	32	64	128
Size	$\alpha = 1/4$	10%	12%	12%	13%
	$\alpha = 1/2$	10%	12%	13%	14%

Conclusion

- Dirty-Block Index is a new cache organization, which decouples dirty bit information from main tag store. All dirty bits are indexed by DRAM row at a separate, much smaller store.

- By doing so, it's much faster to
  - Query all dirty bits for a DRAM row (makes AWB much easier)
  - Query whether certain cache line is dirty (makes CLB much easier)
  - Organize correction bits for dirty cache line (hybrid ECC)

- Evaluation results showed
  - 6% performance improvement over best previous mechanism
  - 8% overall cache area reduction (with ECC)

Open Discussion

- What are the major strengths?
- What are the major weakness?
- Any other ideas from this paper?

My 2 cents - Strength

- This paper proposed a novel cache organization which can improve both performance and cost of cache at the same time.

- The analysis of design consideration was comprehensive and solid. The authors not only mentioned most of major considerations, but also rigorously evaluated the sensitivity of them (granularity, size, replacement policy)

My 2 cents - Weakness

- The DBI eviction induces some unnecessary write back traffics when handling write request. The worst case is all DBI-associated cache lines have to be written back.
- The evaluation was done on a system with L1/L2/L3 caches and specific 2-core, 4-core, 8-core workloads. Some analysis should be done to prove it works on different memory hierarchy and different workload/core combinations.
- One of the major weakness of DBI is that its structure is limited to DRAM row (though it can be adjusted by granularity). If the dirty lines are sparsely distributed in different DRAM row, this structure will be inefficient and keep thrashing DBI cache.

My 2 cents - Ideas

- Investigate the possibility of moving other bits out of tag store, such as cache coherency bits or even ECC bits.
- Try if DRAM-oriented cache structure works better, not just dirty bits. We can have smaller α and a vector to manage cache lines by DRAM row tag address.
- Evaluate the density of DRAM rows in cache for various workloads. Also evaluate DBI with different memory hierarchies to show whether it can be applied to different systems.

Paper to Discuss (2/3)

- Seshadri et al., "The Dirty-Block Index", ISCA 2014.
  - Introducing a cache organization to achieve better performance and cost.
- Ausavarungnirun et al., "Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems", ISCA 2012.
  - Introducing a memory controller which is simpler and works better in heterogeneous system.
- Chang et al., "Improving DRAM Performance by Parallelizing Refreshes with Accesses", HPCA 2014.
  - Introducing a few parallelism schemes for refresh commands.

Background and Problem

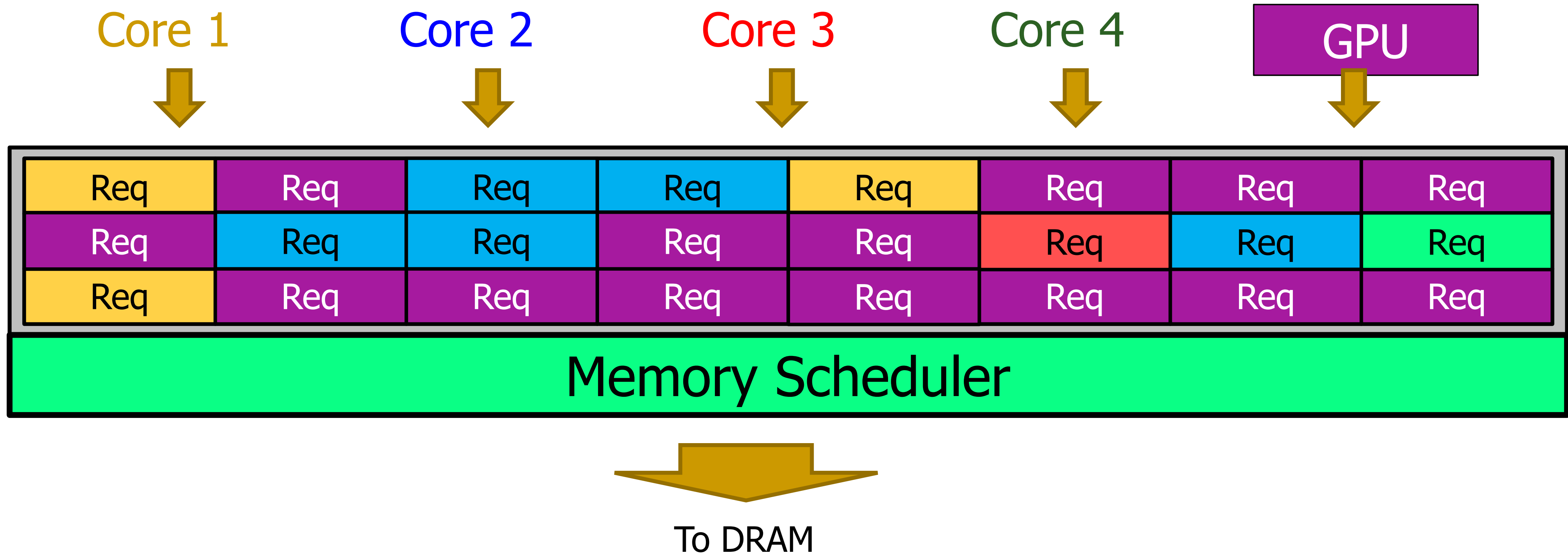
- In multi-core CPU-GPU systems, the memory requests from GPUs can overwhelm that from CPUs.
  - The nature of GPU makes it be able to issue much more outstanding requests than CPU
  - GPU is much more (4x-20x) memory-intensive than CPU
- The memory controller with centralized request buffer will become very complex and costly for such system.
  - Unless it has a lot of buffer, it can't see enough pending requests from CPUs as buffers are overwhelmed by those from GPU
  - But a lot of buffer can make the design very complicated

Prior Memory Scheduling Schemes

- FR-FCFS (First ready, First Come First Serve) [Rixner+, ISCA'00]
  - Totally memory throughput driven
  - Can lead to serious fairness problem
- PAR-BS (Parallelism-aware Batch Scheduling) broda, MICRO'07]
  - Batches request based on arrival time
  - Minimize fairness issue
- ATLAS (Adaptive per-Th d-Service memory scheduling) [Kim+, HPCA'10]
  - Prioritize memory service. But high memory intensity ap d down significantly.
- TC ster Memory Scheduling) [Kim+, MICRO'10]
  - Cl reads into high or low memory-intensity buckets and apply different approaches

Most schemes need certain visibility to requests from all applications to be effective

Introducing the GPU into the System (Rachata's slide)



- GPU occupies a significant portion of the request buffers
  - Limits the MC's visibility of the CPU applications' differing memory behavior → can lead to a poor scheduling decision

The performance of memory schedulers in CPU-GPU systems

- Results showed it's highly dependent on buffer size

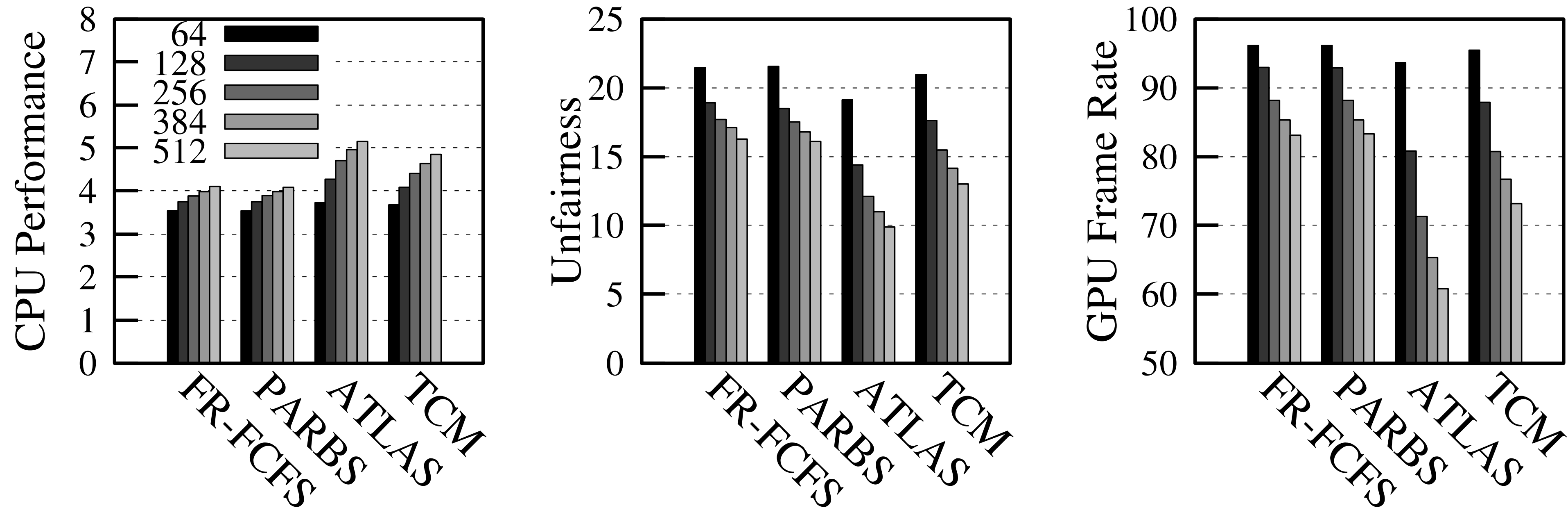


Figure 3. Performance and unfairness of different memory scheduling algorithms with varying request buffer sizes, averaged over 35 workloads.

Problems with Large Monolithic Buffer (Rachata's slide)

Req	Req	Req	Req	Req	Req	Req	Req
Req	Req	Req	Req	Req	Req	Req	Req
Req	Req	Req	Req	Req	Req	Req	Req
Req	Req	Req	Req	Req	Req	Req	Req
Req	Req	Req	Req	Req	Req	Req	Req
Req	Req						

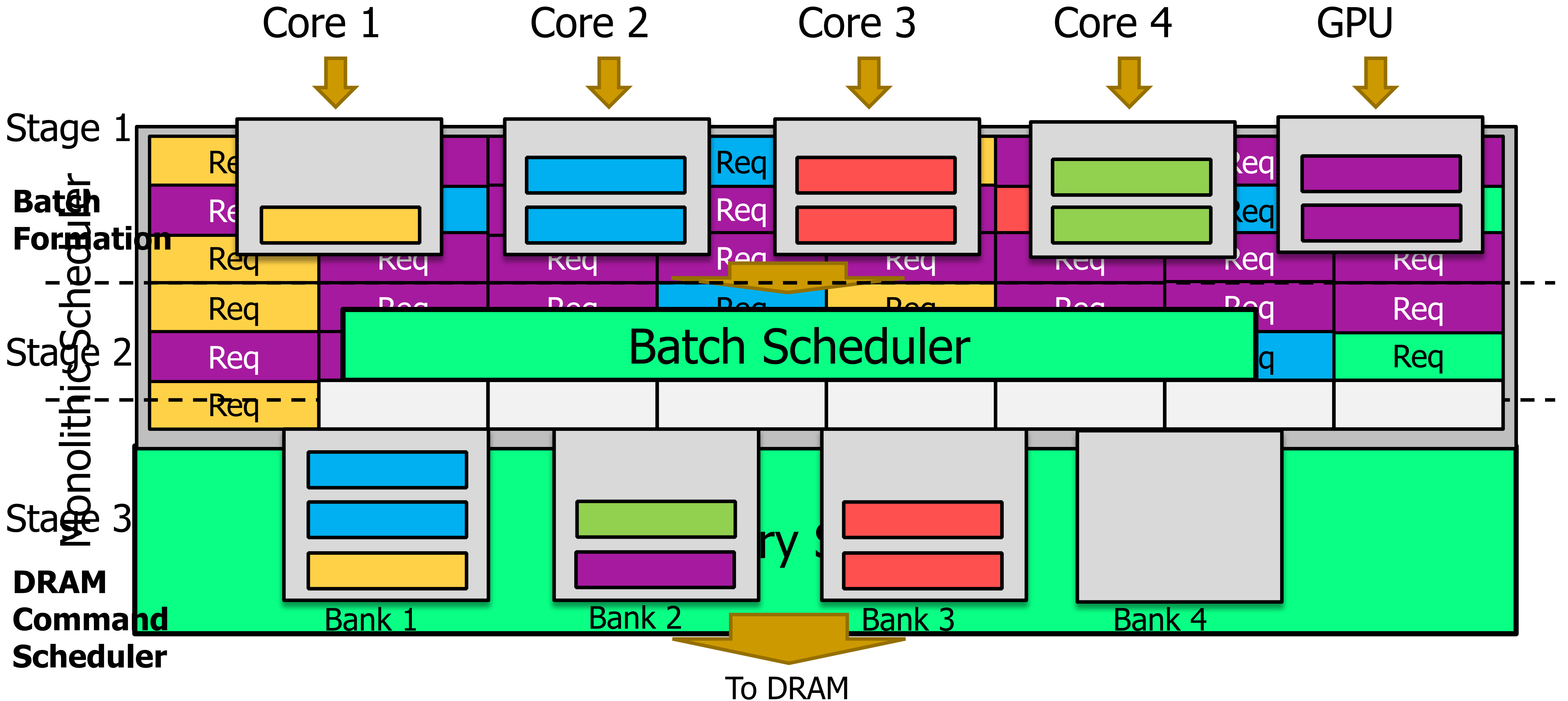
More Complex Memory Scheduler

- A
  -
  -
  - Assign and enforce priorities
- This leads to high complexity, high power, large die area

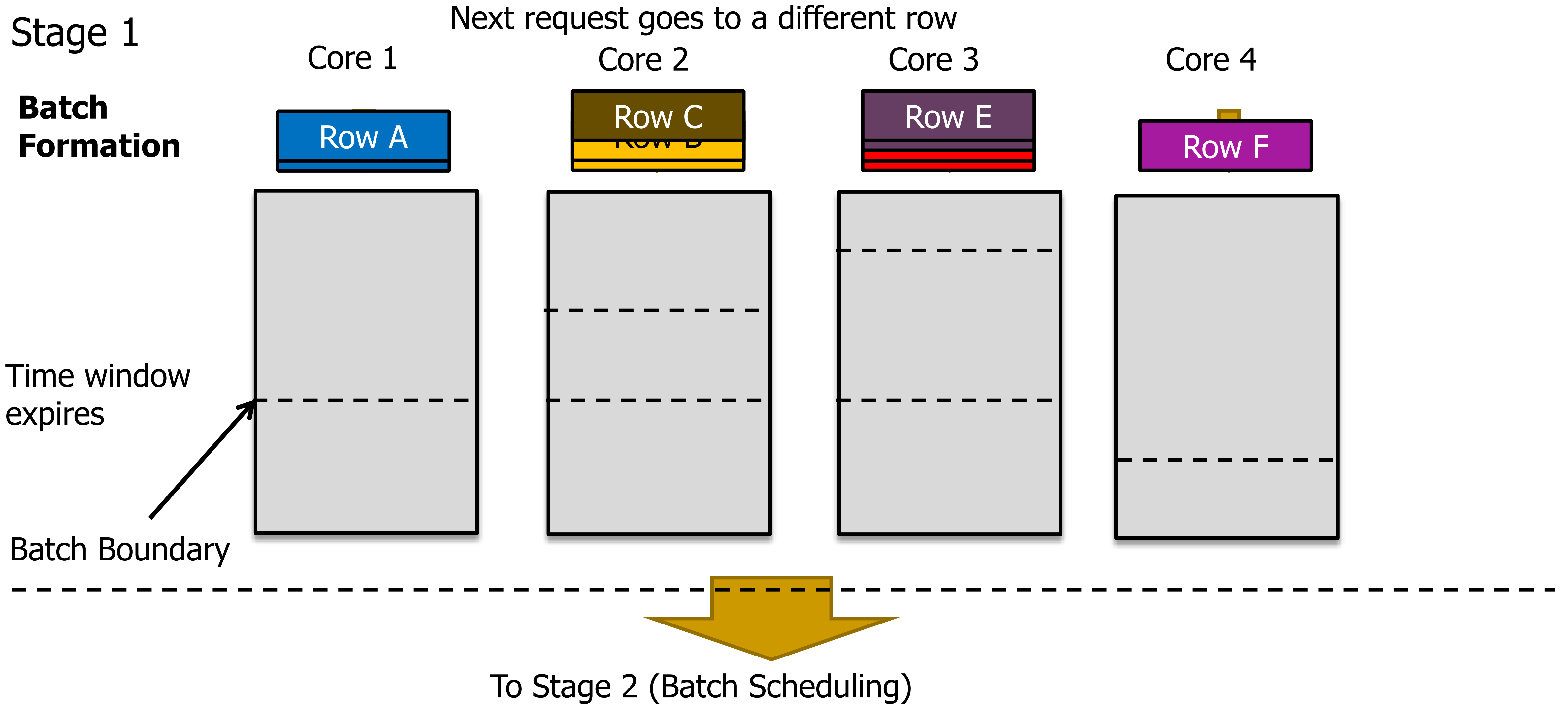
Key Idea: Staged Memory Scheduler

- Decouple the memory controller's three major tasks into three significantly simpler structures
- Stage 1: Group requests by locality
  - The requests to the same row from the same source are grouped as a batch
  - No out-of-order batch
- Stage 2: Prioritize inter-application requests
  - Schedule batches based on SJF (shortest job first) or RR (round-robin)
  - The probability of applying SJF is based on a configurable parameter p
  - Always pick from the head of FIFO from each source
- Stage 3: Schedule low-level DRAM commands
  - Issue batches to DRAM, no reorder

SMS: Staged Memory Scheduling (Rachata's slide)



Stage 1: Batch Formation Example (Rachata's slide)



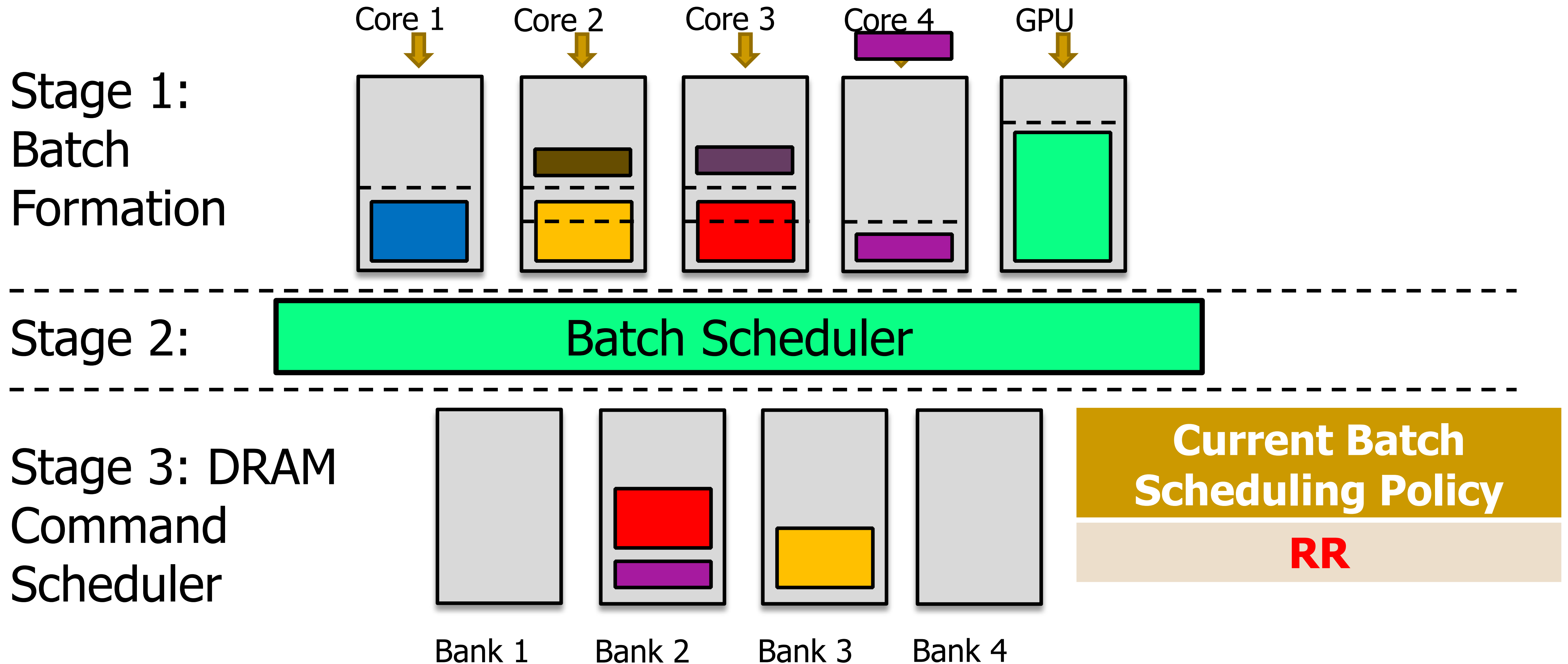
Stage 2: SJF vs RR

- Aside from simplicity, SMS provided another major advantage: Configurable probability p for SJF

- SJF (Shortest Job First)
  - Good for low memory intensive applications (mostly from CPUs)
  - The price is overall system bandwidth
- RR (Round Robin)
  - Good for high memory intensive applications (mostly from GPUs)
  - The price is the fairness of low memory intensive applications

- Make it as a configurable parameter provides flexibility. The system can adjust p to reach best tradeoff between GPU performance and CPU fairness.

Putting Everything Together (Rachata's slide)



Evaluation

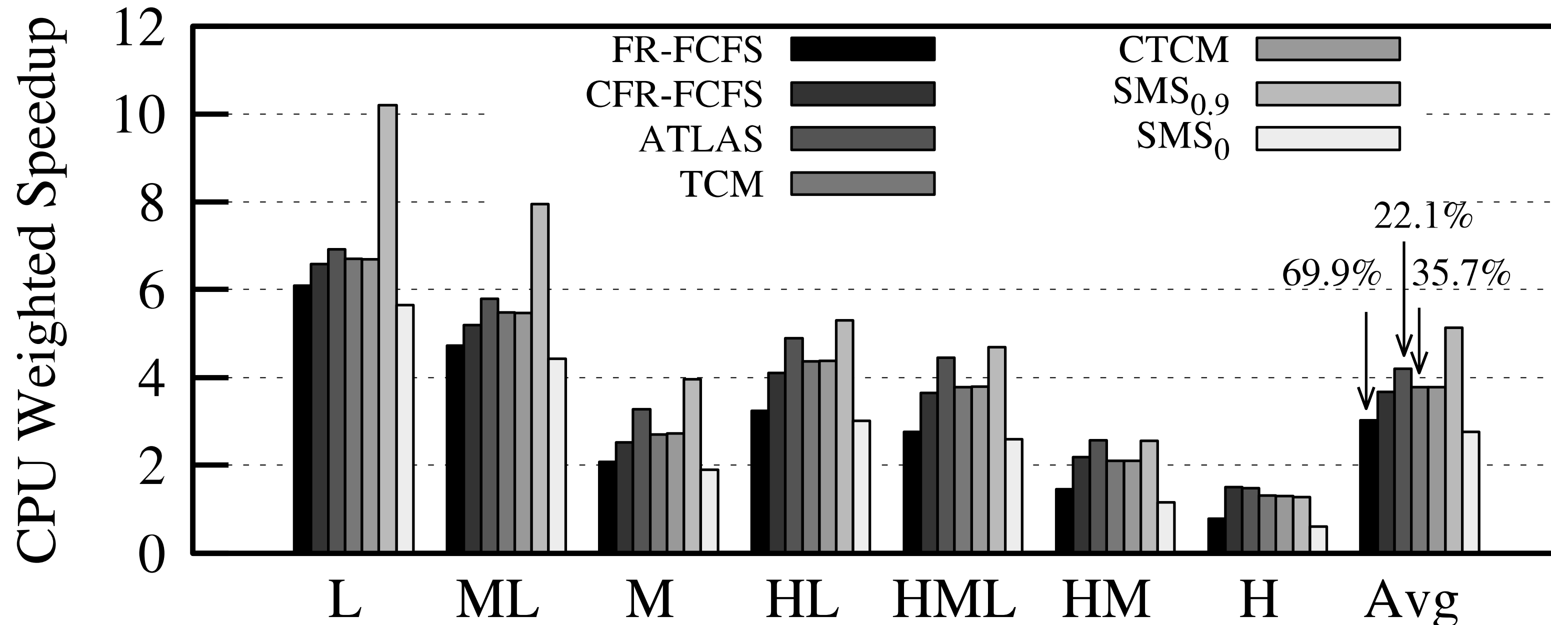
- Power and Area (in 110nm, compared to FR-FCFS)
  - 66% less area
  - 46% less static power

- Performance Metrics
  - GPU Weight means how important GPU performance is for a system
  - If it's important, SMS can use smaller p value to reach best overall performance

$$CGWS = CPU_{WS} + GPU_{Speedup} * GPU_{Weight}$$

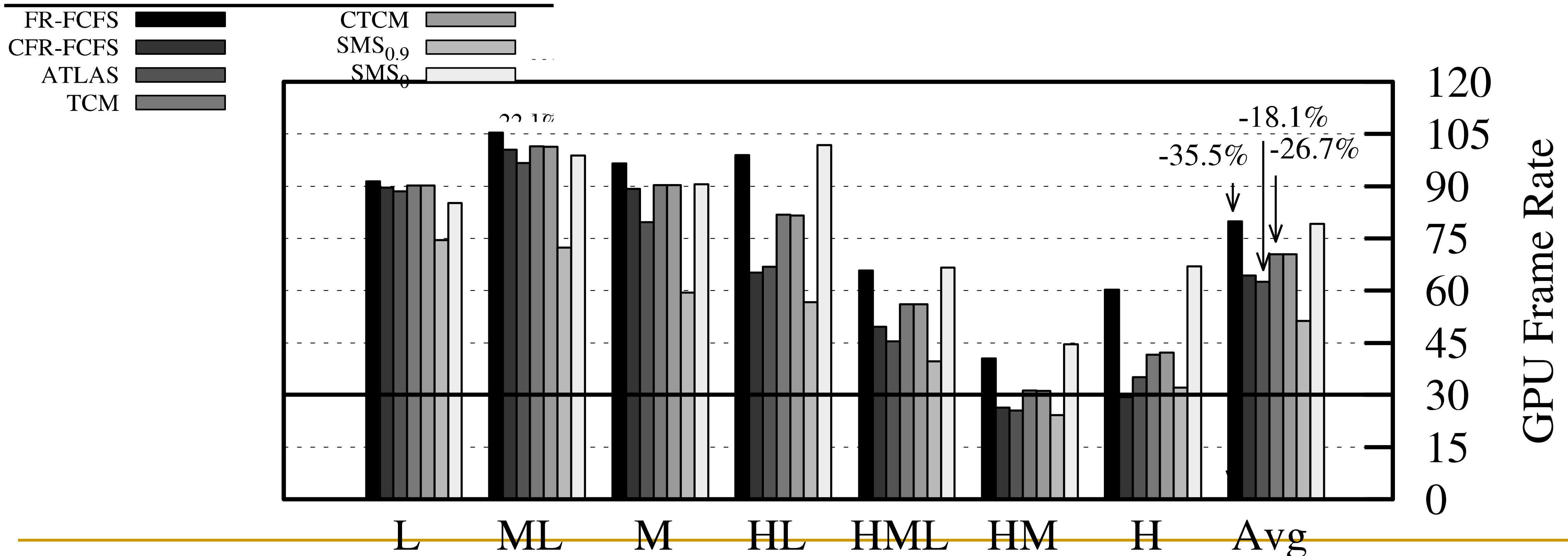
Results – CPU Speedup

- $SMS_{0.9}$ worked very well for low memory intensity applications, but not so well for high ones.
- SMS_0 performed inversely.



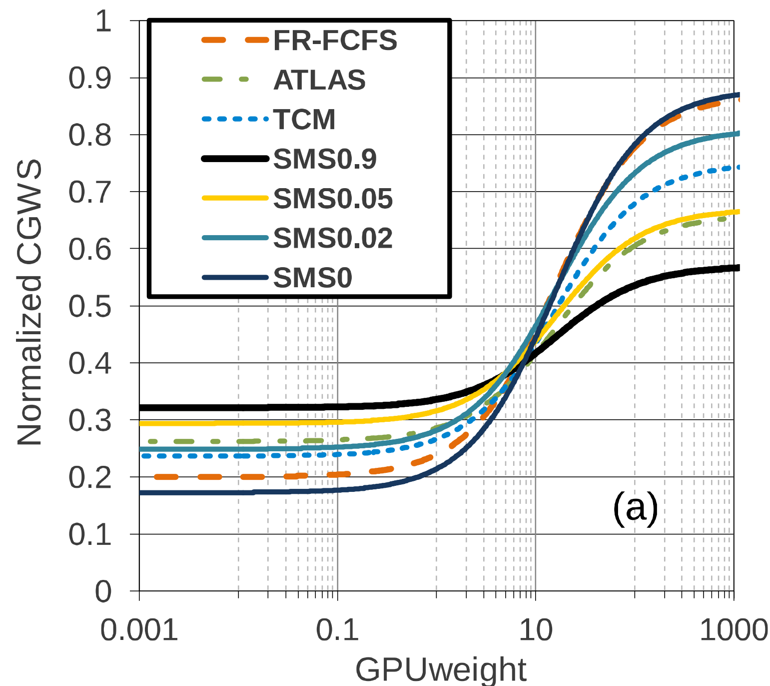
Results – GPU frame rate

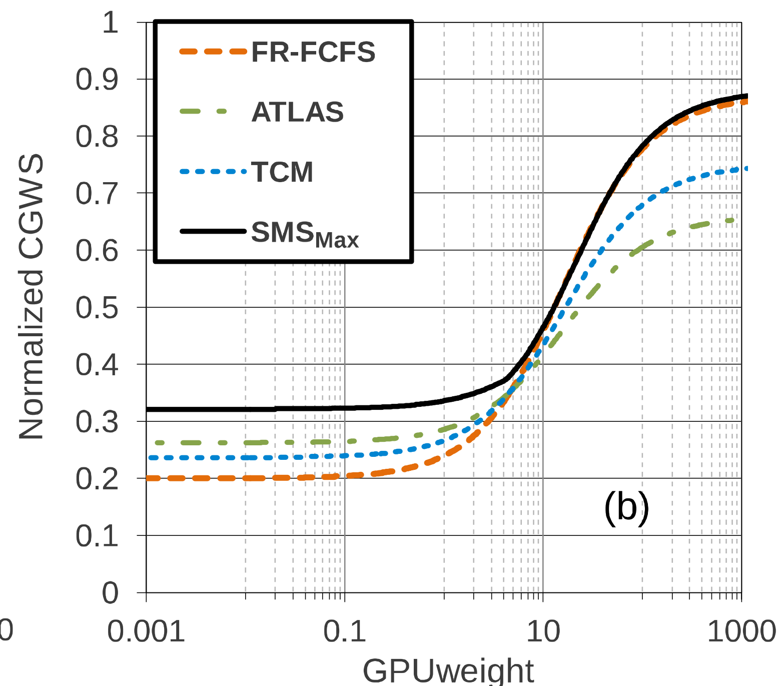
- $SMS_{0.9}$ was the worst in terms of GPU frame rate.
- But SMS_0 gave a comparable GPU frame rate with FR-FCFS.

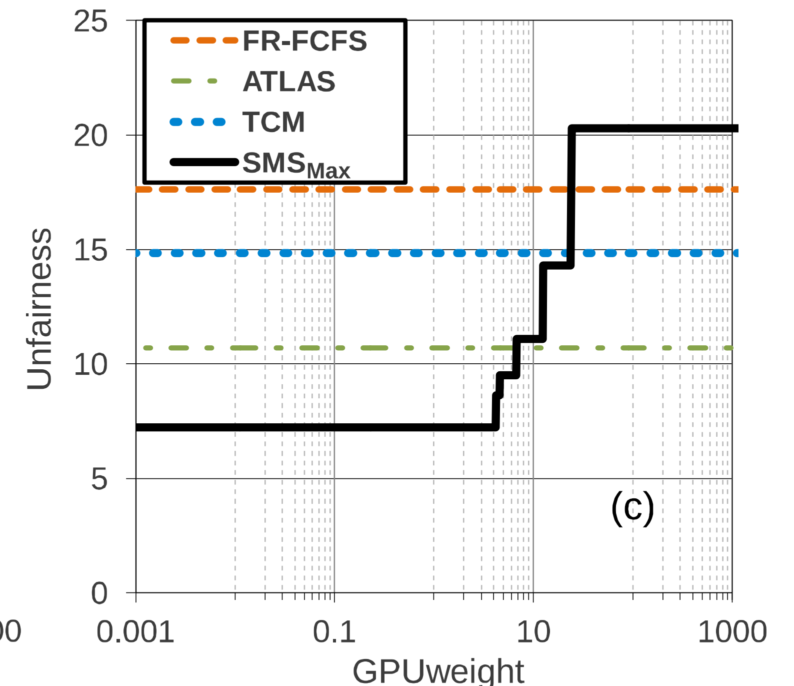


Results – Combined CPU and GPU speed up

- If sticking with a constant p, SMS may be good at some GPU weight but very bad at the others
- If choosing p wisely, SMS can reach best combined CPU+GPU speedup







Conclusion

- Prior memory scheduling scheme can't handle CPU-GPU systems effectively because the memory intensity from GPU is much higher.
- The proposed SMS (Stage Memory Scheduling) decouples the memory scheduler into 3 stages
  - Batch formation, batch scheduling, DRAM command scheduling
- Evaluation result
  - 66% less area
  - 46% less static power
  - Flexible parameter p to reach optimal tradeoff between fairness and system throughput (or best combined CPU+GPU performance, according to system goal)

Open Discussion

- What are the major strengths?
- What are the major weakness?
- Any other ideas from this paper?

My 2 cents - Strength

- Presented an important challenge to memory controller scheduling with very good data analysis.

- The discussion of SMS rationale was pretty thorough. It's the key to make the mechanism much simpler than previous schedulers.

- The experimental evaluation was pretty solid. It included a lot of metrics and also evaluated on the sensitivity of parameters of SMS.

- The proposed scheduler was simpler and more flexible than state-of-art ones. The rigorous evaluation also showed that SMS had better potential to fulfill different needs

My 2 cents - Weakness

- The tunable parameter p provided a flexibility to fulfill different needs (either fairness or performance). However that can be a problem as an incorrect parameter can cause severe performance or fairness degradation.

- This memory scheduler was very simple but gave up some important optimization opportunity. For example, it gave up cross-source row buffer locality. It didn't try to prevent row conflict either.

- The workload used unrelated workloads between CPU and GPU for evaluation, which may not represent the system performance goal very well.

My 2 cents - Ideas

- Research on inter-dependent, state-of-art benchmark for CPU-GPU multicore system. Use it to validate state-of-art memory controllers and maybe come out a more efficient scheduling mechanism.

- Research other mechanism which can do better tradeoff between performance and fairness by considering more optimization opportunities. It shouldn't limit to CPU-GPU only and should take more system configurations into account.

Paper to Discuss (3/3)

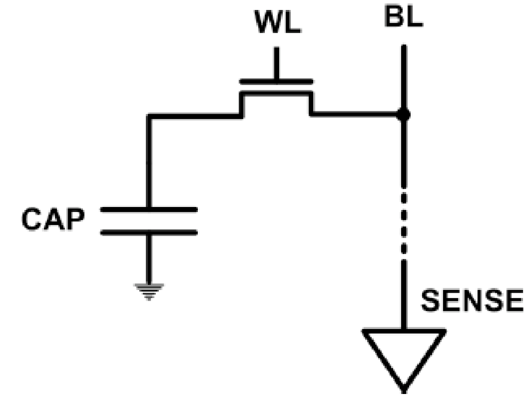
- Seshadri et al., "The Dirty-Block Index", ISCA 2014.
  - Introducing a cache organization to achieve better performance and cost.

- Ausavarungnirun et al., "Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems", ISCA 2012.
  - Introducing a memory controller which is simpler and works better in heterogeneous system.

- Chang et al., "Improving DRAM Performance by Parallelizing Refreshes with Accesses", HPCA 2014.
  - Introducing a few parallelism schemes for refresh commands.

Background – DRAM Refresh

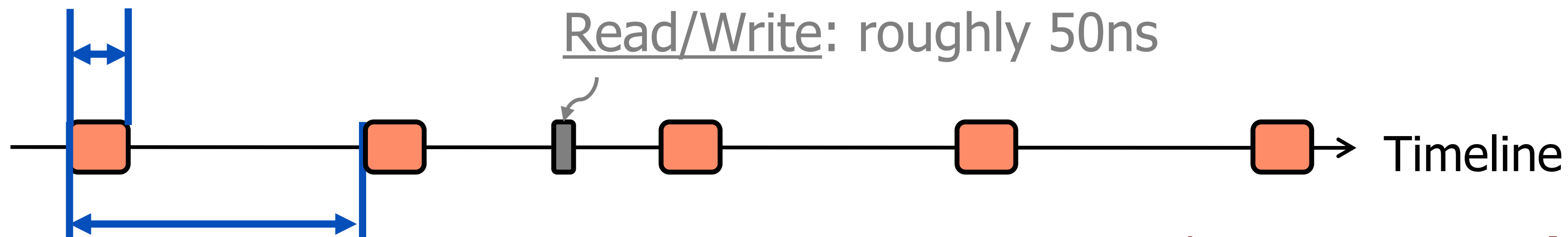
- DRAM stores data in capacitors, which leaks charge over time
  - So all DRAM cells need to be refreshed for a certain period of time (tREFI)



- Because all cells need to be refreshed, the time to do refresh (tRFC) depends on the capacity of DRAM cells

tRefLatency (tRFC): Varies based on DRAM chip density (e.g., 350ns)

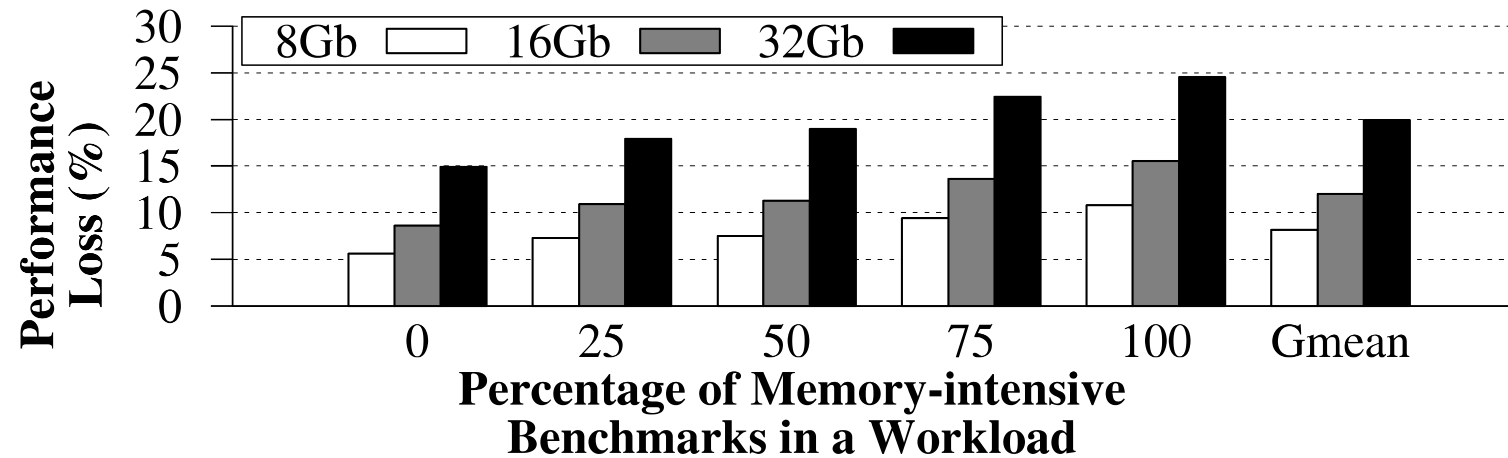
Read/Write: roughly 50ns



tRefPeriod (tREFI): Remains **constant**

*From Kevin's slide

Refresh Overhead: Performance



Refresh Problem

- During refresh, DRAM is not accessible
  - All bank refresh (REF_{ab}): all banks can't be accessed
  - Per bank refresh (REF_{pb}): the refreshing bank can't be accessed
- As DRAM capacity grows, the refresh overhead will take too much time and makes performance unacceptable
  - Even with per bank refresh, it still takes too much time
  - Besides, the total time spent on REF_{pb} is more than REF_{ab}
    - $tRFC_{pb}$*banknum > $tRFC_{ab}$
    - That is because all bank refresh are done on multiple banks simultaneously.

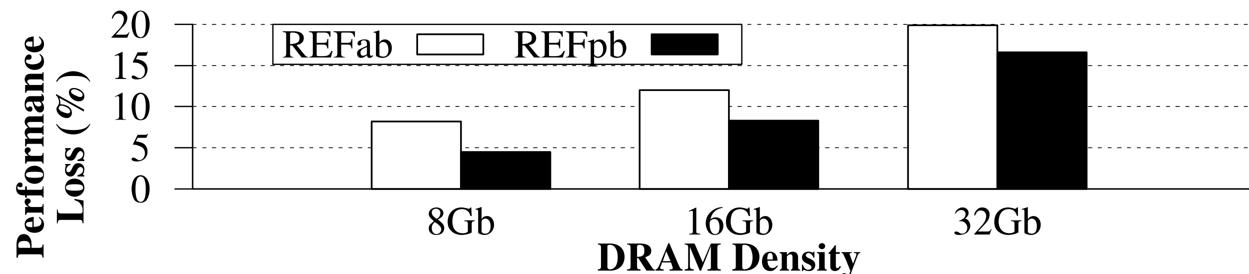


Figure 7: Performance loss due to *REF_{ab}* and *REF_{pb}*.

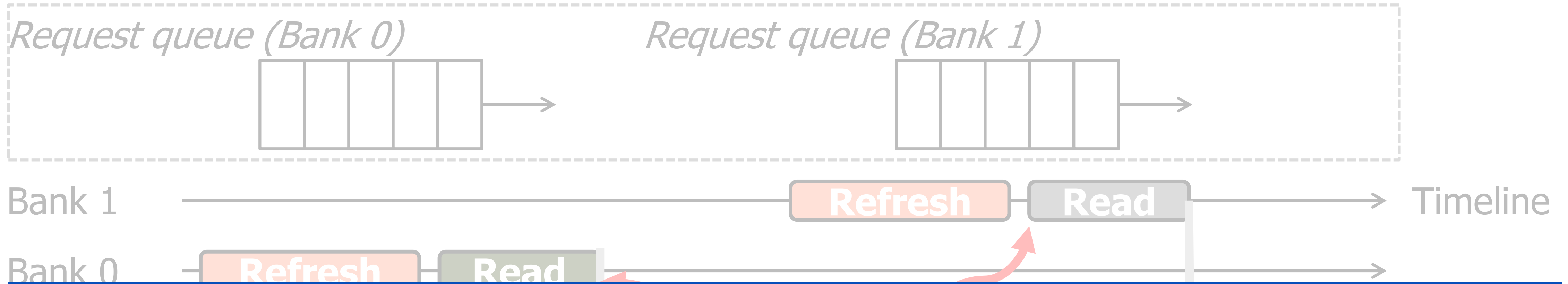
Idea one - DARP

- DARP - Dynamic Access Refresh Parallelization
  - Currently, the order of per bank refresh is round-robin and controlled by DRAM chip.
  - The key idea of DARP is to let memory controller fully control which bank to refresh

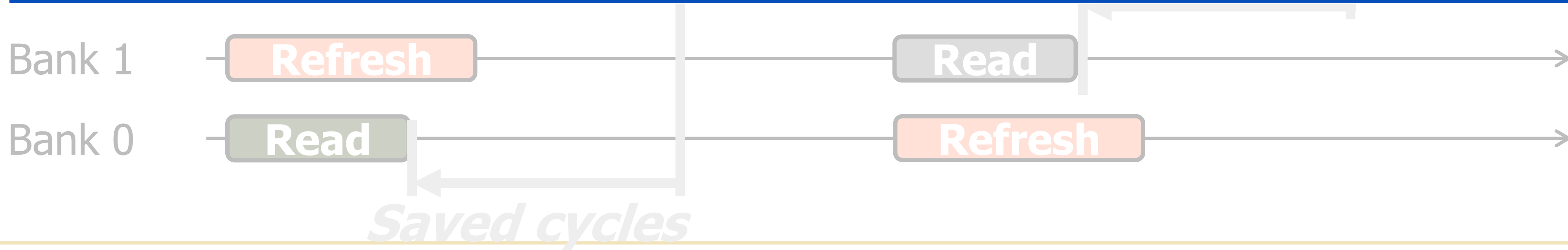
- Out-of-Order Per-Bank Refresh
  - Memory controller can do optimal scheduling based on the command queue
  - It can schedule the refresh to the banks with least pending commands (within timing constraint)
- Write-refresh Parallelism
  - Most memory controller will have burst write mode to save read-write turnaround overhead
  - Memory controller can schedule more refresh during burst write mode, as write latency generally won't be at critical path of system performance.

1) Out-of-Order Per-Bank Refresh (Kevin's slide)

Baseline: Round robin

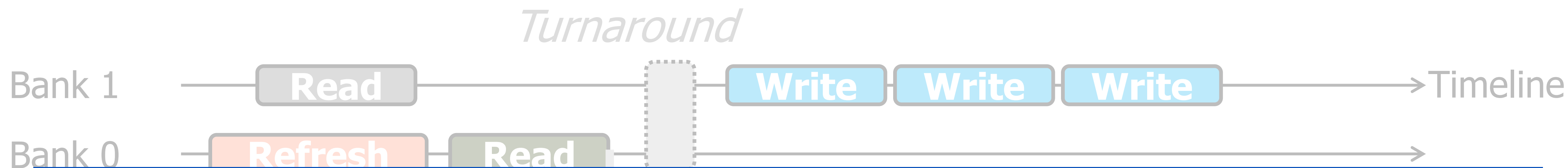


Reduces refresh penalty on demand requests by refreshing idle banks first in a flexible order



2) Write-Refresh Parallelization (Kevin's slide)

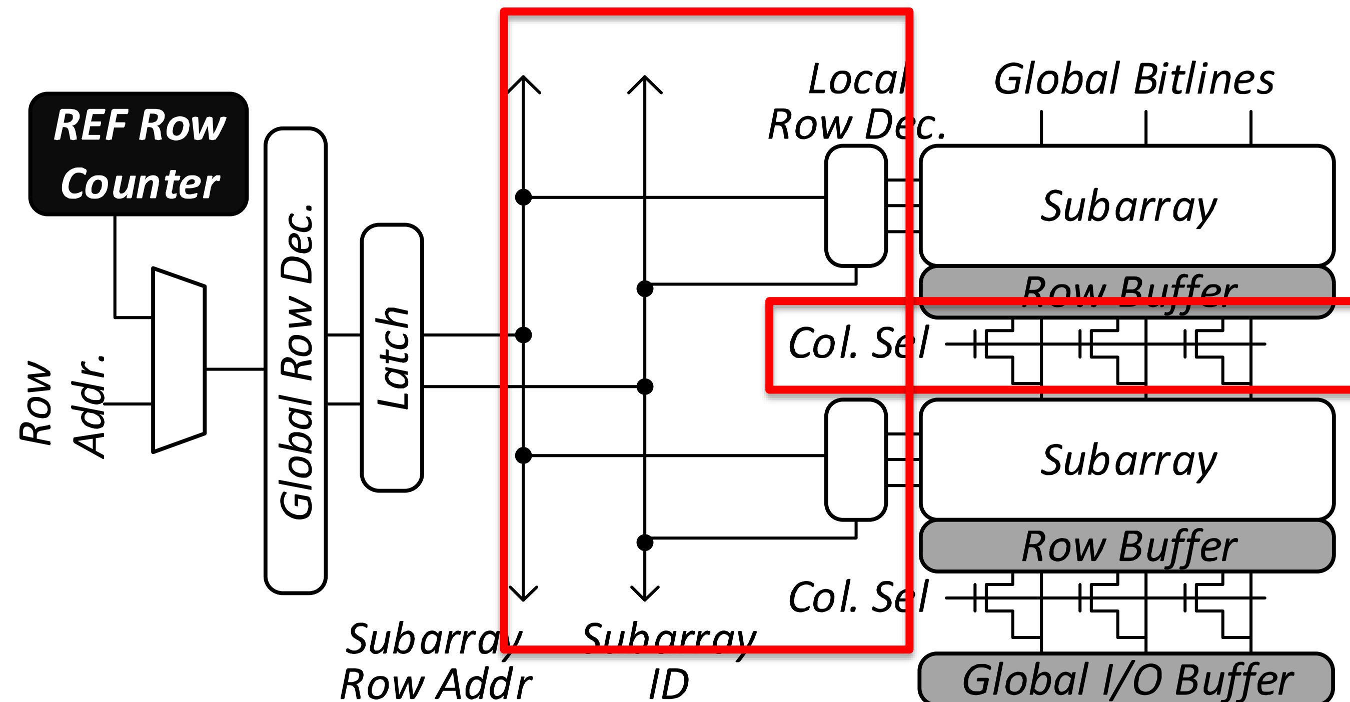
- Proactively schedules refreshes when banks are serving **write batches**



Avoids stalling latency-critical read requests by refreshing with non-latency-critical writes

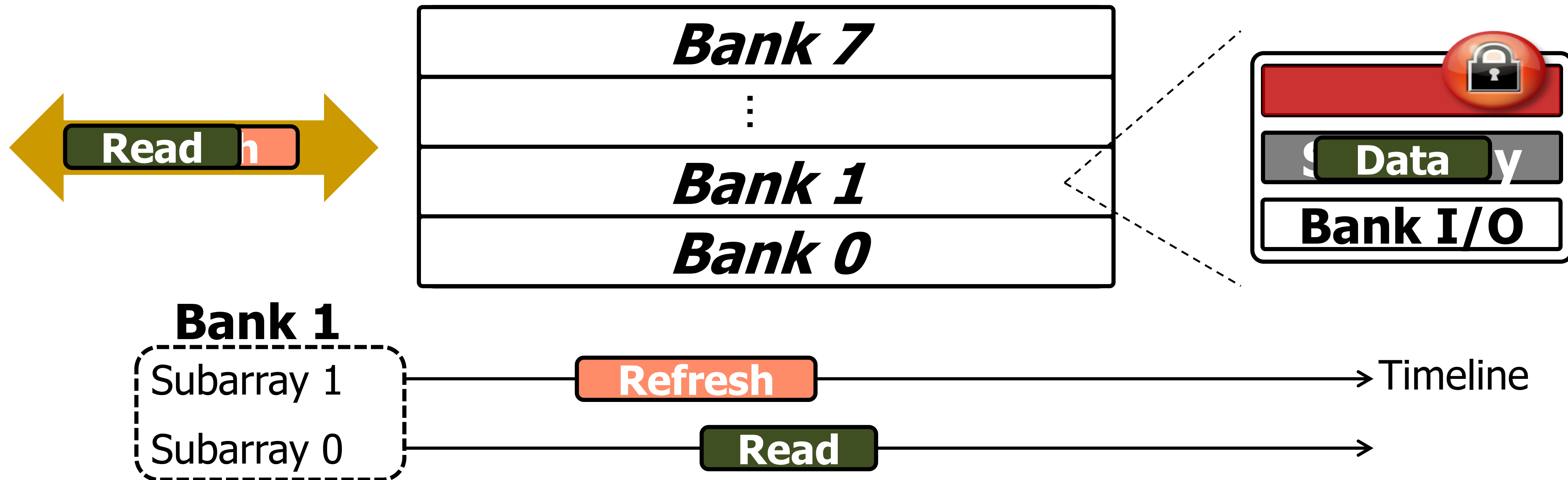
Idea two - SARP

- SARP – Subarray Access Refresh Parallelization
  - Each subarray has its own row buffer.
  - With some modification, we can parallelize refresh and access of different subarrays in the same bank



Our Second Approach: SARP (Kevin's slide)

- **Subarray Access-Refresh Parallelization (SARP):**
  - Parallelizes refreshes and accesses **within a bank**



Very modest DRAM modifications: 0.71% die area overhead

Evaluation

- Evaluated schemes
  - REF_{ab} : All bank refresh (baseline)
  - REF_{pb} : Per-bank refresh
  - Elastic : Try to schedule refresh when memory idle
  - DARP : The first idea, out-of-order per-bank refresh
  - $SARP_{pb}$: The second idea, subarray level parallism and works on REF_{pb}
  - DSARP : The combination of DARP and SARP
  - No REF : Ideal case, no refresh required

Results (1/3)

- When memory capacity grows to 32Gb, the benefit of DARP is decreasing
  - That is because the refresh time is too long and no way to hide it any more
- But with SARP, the performance gain is increasing with memory capacity
  - That is because SARP parallelize access and refresh to subarray level.

Density	Mechanism	Max (%) REF_{pb}	Max (%) REF_{ab}	Gmean (%) REF_{pb}	Gmean (%) REF_{ab}
8Gb	DARP	6.5	17.1	2.8	7.4
	$SARP_{pb}$	7.4	17.3	3.3	7.9
	DSARP	7.1	16.7	3.3	7.9
16Gb	DARP	11.0	23.1	4.9	9.8
	$SARP_{pb}$	11.0	23.3	6.7	11.7
	DSARP	14.5	24.8	7.2	12.3
32Gb	DARP	10.7	20.5	3.8	8.3
	$SARP_{pb}$	21.5	28.0	13.7	18.6
	DSARP	27.0	36.6	15.2	20.2

Table 2: Maximum and average WS improvement due to our mechanisms over REF_{pb} and REF_{ab}.

Results (2/3)

- REFpb can't improve too much when memory capacity grows
- Surprisingly, DSARP can be almost as good as ideal no-refresh memory

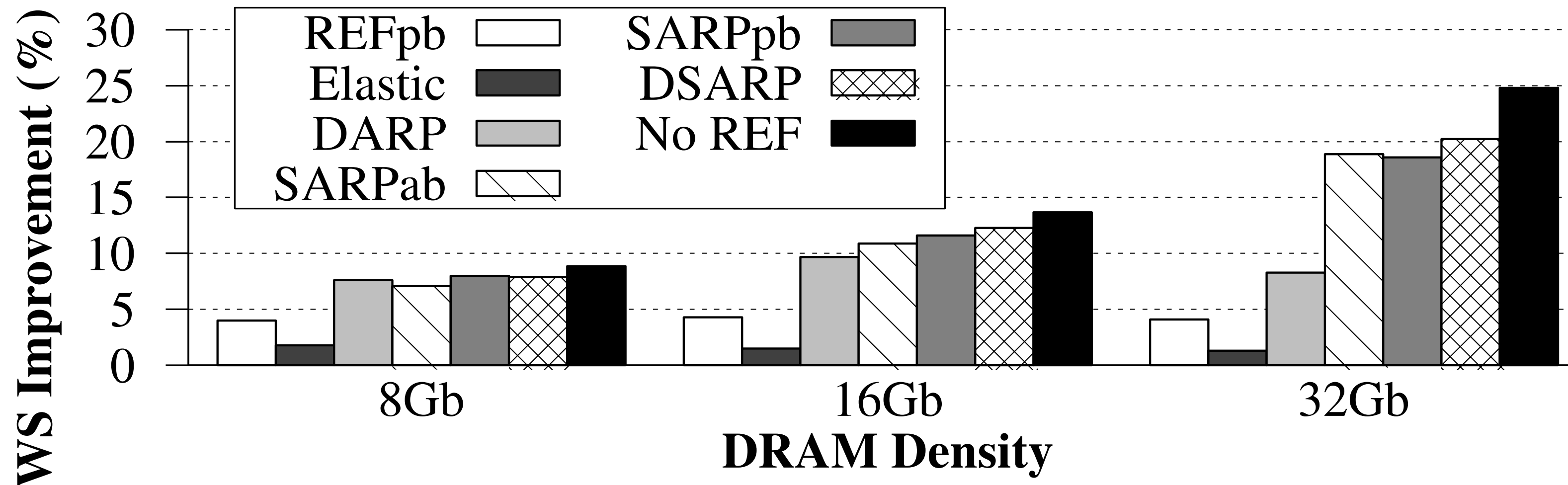


Figure 13: Average system performance improvement over *REF_{ab}*.

Results (3/3)

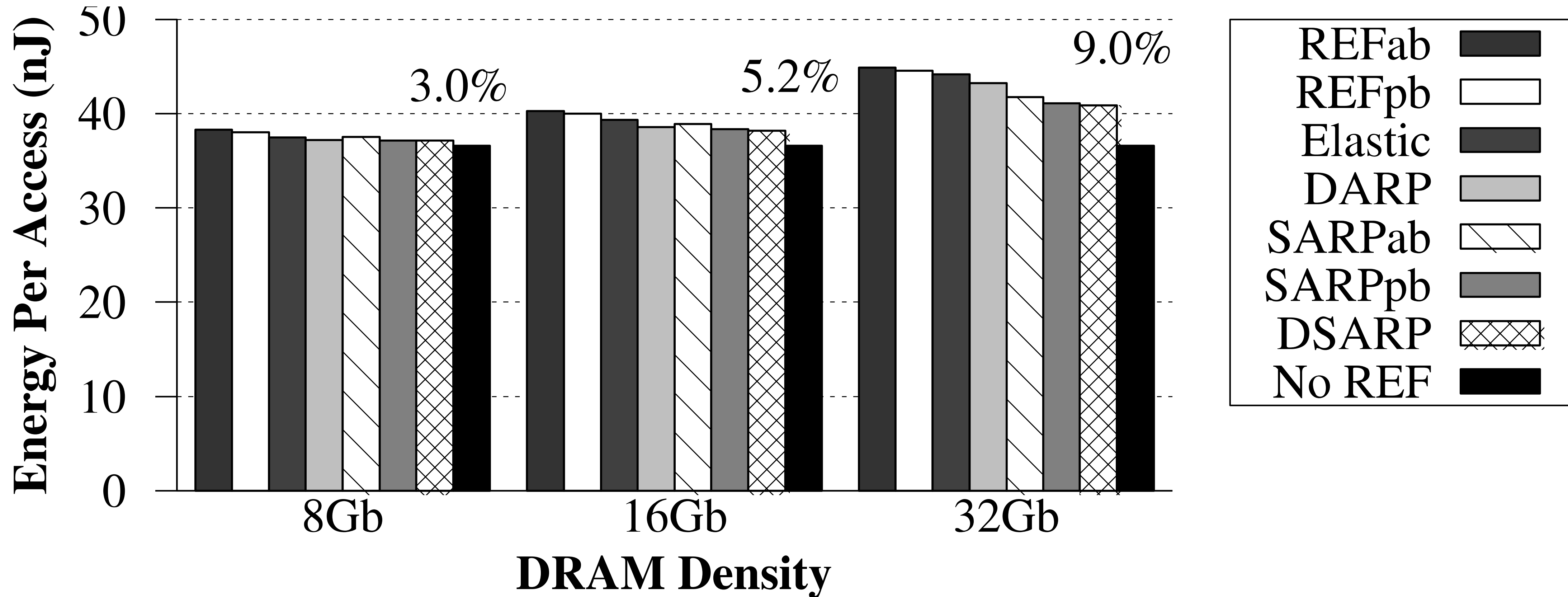


Figure 14: Energy consumption. Value on top indicates percentage reduction of DSARP compared to REF_{ab}.

Conclusion

- To mitigate the deteriorating DRAM refresh overhead, this paper proposed two mechanisms to parallelize refresh with access
- DARP enables memory controller to fully control the order and timing of per-bank refresh
- SARP enables memory controller to issue access and refresh to different subarrays in the same bank
- The performance improvement is significant and consistent, and close to ideal no-refresh memory
  - 7.9% for 8Gb DRAM
  - 20.2% for 32Gb DRAM
  - With only 0.71% DRAM die area cost

Open Discussion

- What are the major strengths?
- What are the major weakness?
- Any other ideas from this paper?

My 2 cents - Strength

- The statement of the refresh problem was very clear with a good background introduction. The explanation of all-bank refresh and per-bank refresh gave enough details to understand the optimizations.

- The proposed mechanism tackled the refresh problem with several novel ideas on DRAM architecture modification. The idea of parallelizing and fined-grained control on refreshes provided promising direction to mitigate refresh overhead.

- The evaluation mechanisms and results were solid. The best among them is the comparison to ideal non-refresh case, which showed how good DARP and SARP are. The sensitivity analysis on workloads, core count, timing, and DRAM architecture was comprehensive, too.

My 2 cents - Weakness

- SARP can make the scheduling of refresh very complicated. This paper didn't discuss how complicated it can be or how to do it efficiently.

- This paper didn't discuss in detail about the scheduling policy of REF_{ab} and REF_{pb}. Elastic refresh should be applied to REF_{pb}, too. By considering this, the performance improvement may not be that significant

- The energy data didn't include memory controller, where significant complexity was added by DARP and SARP. The algorithm of out-of-order per-bank refresh may cost too much energy, as it has to make a complex decision on every cycle

My 2 cents - Ideas

- As an extensive work, the energy of memory controller could be a concern. If it is, it will be interesting to find a better algorithm to exploit DARP/SARP with competitive power consumption.
    - Or better refresh scheduling to achieve better performance.

- Another idea is to research more aggressive parallelism.
    - Is it possible to parallel REF_{pb} with another REF_{pb}?
    - Is it possible to optimize the DRAM timing with more understand on its limitation?