Tackling Memory Access Latency Through DRAM Row Management

Sriseshan Srikanth
Georgia Institute of Technology

Lavanya Subramanian
Intel Labs

Sreenivas Subramoney
Intel Labs

Thomas M. Conte
Georgia Institute of Technology

Hong Wang
Intel Labs

ABSTRACT

Memory latency is a critical bottleneck in today's systems. The organization of the DRAM main memory necessitates sensing and reading an entire row (around 4KB) of data in order to access a single cache block. The benefit of this organization is that subsequent accesses to the same row can be served faster (row hits). However, accesses to other rows incur high latency to prepare the DRAM bank for a subsequent access and read the contents of the new row (row conflicts). Therefore, the decision on how long a row is held open for is a key factor that determines the access latency incurred by requests to memory.

While prior work has tackled this problem, existing solutions are either complex or ineffective. Our goal, in this work, is to build a row management scheme that is simple yet effective. Towards this end, we first build a scoreboard scheme that determines how long to hold a row open, by i) predicting the number of row hits and row conflicts for different lengths of time rows are held open and ii) picking the time that maximizes row hits without increasing row conflicts significantly. We then observe that a small set of rows tend to experience a large number of back-to-back accesses. We build a row exclusion scheme that identifies such rows and prevents them from being closed until the next access to a different row arrives. Our evaluations show that our scoreboard and row exclusion policies together incur less than 0.4% of the additional storage cost of the most effective prior mechanism, while surpassing it in terms of performance.

CCS CONCEPTS

• **Hardware → Dynamic memory**;

ACM Reference Format:
Sriseshan Srikanth, Lavanya Subramanian, Sreenivas Subramoney, Thomas M. Conte, and Hong Wang. 2018. Tackling Memory Access Latency Through DRAM Row Management. In *The International Symposium on Memory Systems (MEMSYS), October 1–4, 2018, Old Town Alexandria, VA, USA.* The International Symposium on Memory Systems (MEMSYS), 2018, 11 pages. https://doi.org/10.1145/3240302.3240314

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MEMSYS, October 1–4, 2018, Old Town Alexandria, VA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6475-1/18/10…$15.00
https://doi.org/10.1145/3240302.3240314

1 INTRODUCTION

Memory latency is a critical performance bottleneck in today's systems. Requests that miss in the on-chip caches and need to access the main memory experience significant delays and cause high performance degradation. The key reason for such high latencies is the organization of the DRAM main memory system. While a DRAM memory system offers parallelism at multiple levels of the hierarchy, requests to the same memory bank cannot benefit from bank-level parallelism, causing serialization. Furthermore, there are many key timing constraints that limit how quickly back-to-back requests to a bank can be served. When a cache block worth of data needs to be read from a bank, an entire row (around 4KB) worth of data needs to be sensed and read into an internal row buffer/sense amplifier (activation). Subsequent accesses to data in the same row can be served from the row buffer (called a row hit). However, before the next access to a different row can be served at the same bank (called a row conflict), the bitlines that enable sensing and reading data from the DRAM cells need to be precharged to a certain voltage level. An access to a bank whose bitlines have already been precharged incurs only the activation latency (called a row miss). The activation and precharge latencies are significant (around 12-18 ns) in typical DDR memories.

Holding a row open until the next request to a different row arrives (known as open row policy) maximizes the number of possible row hits, but incurs the precharge latency every time a request to a different row than the currently open one is served. On the other hand, closing the row after each request (closed row policy) incurs the activation latency even for consecutive requests to the same row. Prior work has sought to find a middle ground between these two extremes by managing when to hold a row open vs. when to close a row. Some prior works [13, 26, 28, 29] employ low complexity schemes that predict whether or not to hold a row open based on recent past behavior or probabilistic estimates. Other prior works [1, 17, 34, 37] have explored schemes that track row hit/miss behavior at a per-row/address granularity and employ such per-row/address information to determine when to close a row, incurring high complexity. **We observe that these schemes are either simple, but ineffective or effective, but complex.**

Our goal in this work is to achieve the best of both performance and complexity. We seek to build a DRAM row management scheme that is simple yet effective.

Towards this end, we make two key observations. First, we observe that we can determine an optimal length of time (timeout window) for which rows should be held open globally, (i.e., irrespective of row address) if we knew the row hit and conflict counts

for different timeout windows. Such knowledge of row hit and conflict counts across different timeout windows would enable us to determine the length of the timeout window that would maximize row hits, while not increasing row conflicts significantly. We build a scoreboard mechanism to project whether each request would result in a row hit, miss or conflict for different timeout windows. Specifically, a row conflict could become a row miss if a row is closed earlier or a row miss could become a row hit if a row is held open longer. Our scoreboard mechanism projects such conversions for different timeout windows for each request. The row hit and row conflict counts computed from such projections for different timeouts are then employed to determine the best timeout window periodically, to adapt to various application phases.

Second, we observe that although the scoreboard mechanism is effective in capturing global row hit and conflict trends to determine a timeout window that is effective across rows, there is a small set of rows that experience a large number of back to back accesses. We build a row exclusion mechanism that identifies such rows and holds them open even after the timeout window has expired and closes such rows only after the next access to a different row arrives at the same bank. This enables us to capture the inherent row locality that is available for such rows.

1.1 Contributions

Managing the row buffer effectively is an important aspect of DRAM access latency mitigation. Prior works have explored various DRAM row management policies, however, they are either simple but ineffective or effective but complex. As such, we make the following contributions:

- We develop the key insight that the knowledge of row hit and conflict counts for different timeout windows enables determination of the best timeout window to maximize row hits, without increasing row conflicts significantly. Towards this end, we build a **scoreboard** mechanism to project the row hit and conflict counts for different timeout windows. As a result, our cycle-accurate simulations indicate a performance improvement of 6.3%, on average for memory intensive workloads, over a static timeout window.
- We design a **row exclusion** mechanism that identifies a small set of rows with inherently high row buffer locality and subsequently holds such rows open beyond the global timeout window. When used in conjunction with our scoreboard mechanism, an average performance improvement of 6.8% is seen over a static timeout window, for memory intensive workloads.
- The scoreboard and row exclusion mechanisms together incur an additional **storage overhead of less than 0.4% of that of the most effective previous mechanism**, while, in fact, surpassing it in terms of performance improvement.

2 BACKGROUND AND MOTIVATION

In this section, we first describe the organization of a typical DRAM main memory, its operation and the different timing constraints that contribute to the high DRAM access latency. We then focus on the row management policy and its impact on performance, motivating why we seek to specifically tackle the DRAM row management policy.

2.1 DRAM Organization and Operation

The DRAM main memory is organized hierarchically as channels, ranks and banks, as shown in Figure 1a. Banks represent the smallest exposed unit of parallel access in a DRAM memory system. Banks that are part of the same rank share some peripheral access circuitry, whereas banks in different ranks are decoupled, providing more potential for parallelism than banks in the same rank. In some cases (such as DDR4), banks are clustered into bankgroups, such that accessing them in a time division multiplexed manner hides the speed difference between the faster interface and slower DRAM core. The ranks (and banks) on a channel share the address and data buses. The DRAM memory system is designed to support different degrees of parallelism at multiple levels of the hierarchy. However, requests to the same bank are serialized and experience delays in access, due to the internal organization of a DRAM bank.

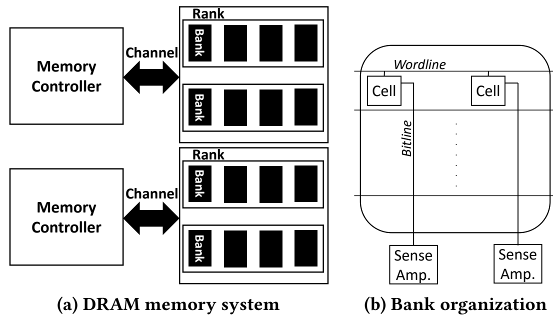
A DRAM bank is a 2 dimensional array of capacitive cells that store data in the form of charge. Cells are connected to wordlines in the horizontal direction and bitlines in the vertical direction, to enable access, as shown in Figure 1b. Each bitline is connected to a sense amplifier that senses and amplifies the charge stored in the cell. A bank is in turn implemented as a collection of subarrays; the collective array of sense-amplifiers associated with all the subarrays of a bank is commonly referred to as a row buffer. Before the data in a cell can be accessed, the bitline needs to be precharged to a $V_{dd}/2$ voltage level (precharge or PRE operation). When the data in a cell needs to be accessed, the corresponding wordline is activated, connecting the cell to a bitline. The charge in the cell perturbs the bitline from its $V_{dd}/2$ voltage level. This perturbation is amplified by the sense amplifier and the bitline voltage is raised to 0 or V_{dd} depending on the data value (charge) in the cell. This constitutes an activate operation. Only after a row has been activated and read into the sense amplifier/row buffer, can the data in the row be read through a Column Access Strobe (CAS) operation.

In order to save command bandwidth, modern memory systems support additional commands such as auto-precharge (RDA/WRA), which would automatically close a row after a column access. However, when not command bandwidth limited, the difference between RDA and RD+PRE is insignificant. The larger question is to determine when such row-closing commands (auto-precharge or otherwise) have to be issued, and this is indeed the focus of this paper. For example, always using auto-precharge commands effectively emulates a closed row policy, which is evaluated in this paper.

2.2 Impact of the precharge and activation latencies

These precharge and activate operations are time consuming (12-18 ns in typical DDR DRAM memories) and contribute significantly to overall memory access latency. Any memory access to a different row than the one currently open incurs the precharge and activation latencies (called a row conflict). A memory access to the same row as the one that is currently open does not incur the precharge and activation latencies (row hit) whereas a memory access to a bank that has no row currently open incurs only the activation latency (row miss).

Figure 2 shows the results of a limit study, in which the ACT latency is avoided whenever possible for a request, and the PRE



(a) DRAM memory system (b) Bank organization

Figure 1: DRAM organization

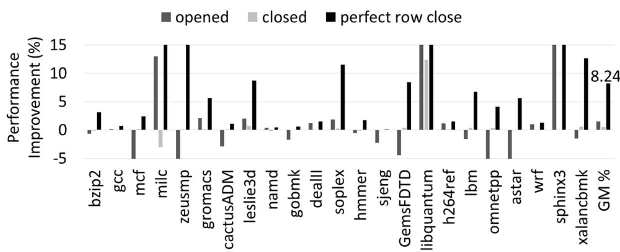


Figure 2: Performance benefits from perfect row close.

latency is hidden altogether, for a set of representative workloads. Specifically, we assume that the row buffer can be kept open for all subsequent requests to an already open row (resulting in row hits), while closing it just in time to not incur the precharge latency for an access to a new row (preventing row conflicts). The results are normalized to a scheme where a row is kept open for 50 cycles after the last CAS, if no other request arrives to a different row within the same bank. As can be seen, the performance gains from avoiding the precharge and activation latencies are 8.2% across all these workloads. Therefore, a row buffer management policy that keeps a row open just long enough to capture the most row hits, while closing the row just in time to minimize row conflicts and transforming them into row misses instead, could achieve significant performance benefits.

In some cases, for more targeted domains, the application can be adapted to reduce the number of redundant row activations and precharges by co-designing the layout of data in DRAM and its access pattern [6, 33]. However, it is highly improbable that this approach would scale to general purpose workloads, and a runtime mechanism in some form becomes necessary.

2.3 Balancing Performance and Complexity

Several prior works have explored policies that seek to manage the row buffer open/close policy to maximize row hits and minimize row conflicts. On the one hand, there are schemes [13, 26, 28, 29] that seek to employ low complexity schemes to maximize row hits and minimize row conflicts towards improving performance. For instance, Jagtap et al. [13] adapt the basic open/close policy to make the idle row closure decision depending on which row of the given bank would be accessed by the pending request queue. With slightly more overhead, Park and Park [26] employ a two-bit saturating counter to track row hit and conflict outcomes and depending on the prediction from the saturating counter, either keep a row open until the next activate closes it, or precharge immediately after a read from a row. A patent by Rokicki [28] presents the high level idea of probabilistically keeping rows in only a subset of banks open at any time. These schemes are very rudimentary and do not capture applications' memory access characteristics effectively, thereby not resulting in significant performance gains (as we show in Sections 7.2).

On the other hand, other prior works [1, 17, 34, 37] have explored schemes that track row hit/miss behavior at a per-row granularity, incurring high complexity. Specifically, Xu et al. [37] propose to employ a two-level access based predictor (similar to branch prediction). Khurshid et al. [17] propose to employ a global history buffer like structure to store sequences of previous accesses and predict row open/close behavior based on tracking such previous access sequences. Awasthi et al. [1] propose to track the number of reads to a row the last time it was open and hold the row open until the same number of reads have occurred when it is accessed in the future. Stankovic et al. [34] propose a two-pronged approach, the first of which is a liveness detector using a 2-bit counter per row of memory, and a global dead time predictor that keeps track of per bank access intervals. These schemes incur high cost and complexity to track access characteristics for each row/address.

In summary, prior works in idle row closure are either ineffective or very expensive. As a result, industry-strength memory controllers have resorted to the simplest implementation – that does not suffer from the polarizing drawbacks of the open/close policy – a fixed (static) timeout policy [2], which forms the baseline used in this paper.

Our goal in this work is to achieve the best of both performance and complexity. Specifically, we seek to manage the row buffer to maximize row hits and minimize row conflicts towards effectively reducing memory access latencies and improving performance, while incurring low hardware cost.

3 KEY OBSERVATIONS

Having described the need for a DRAM row management scheme that is effective in reducing latency/achieving high performance at low cost, we describe our key observations that enable us to build such a mechanism. We seek to achieve an effective balance between performance and complexity by employing a combination of two schemes that i) effectively capture and employ global DRAM row access behavior *and* ii) augment this global row access information with per-row access characteristics, for a small set of rows.

3.1 Effectively Capturing Global Behavior

The ideal length of time to hold a row open after an access, can be effectively predicted by determining the row hit and conflict counts for different timeout windows.

As we describe in Section 2.2, after a read access to an open DRAM row, if the DRAM row is held open, subsequent accesses to the same row would hit in the row buffer and incur low access latencies. Whereas, if the row were closed immediately after the

first access, subsequent accesses to the same row would miss in the row buffer and incur the activation latency. However, if the row is held open indefinitely, the next access to a different row would result in a row conflict and incur the precharge latency. An ideal row management policy would hold the row open just long enough to capture all the subsequent row hits, and would then close it so the next request to a different row results in a row miss, rather than a row conflict.

If we knew the number of row hits and conflicts we would incur if we held DRAM rows open for different lengths of time or timeout windows, as we call them through the rest of the paper, we could pick the timeout window that maximizes row hits without increasing row conflicts significantly. Table 1a shows row hit and conflict counts from a representative phase of execution of length 10000 memory requests, for one of our workloads. The current timeout window of 50 cycles results in 7263 row hits and 684 row conflicts. Increasing the timeout window to 100, 150 and 200 cycles increases the number of row hits steadily. However, it is for a timeout window of 150 cycles that the relative increase in hits vs. conflicts – calculated as difference between the increase in hits and the increase in conflicts with respect to the current timeout – the highest.

Table 1: Row hits, conflicts and speedup for diff. timeouts

(a) Row hit and conflict counts / **(b) Speedup**

Timeout Window	Row hits	Row conflicts	HitIncr-ConflictIncr	Speedup (%)
50-current	7263	684	0	0%
100	8223	662	982	7.6%
150	**8933**	**717**	**1637**	**19.5%**
200	8959	762	1618	17.35%

Table 1b shows the speedup (with respect to the current timeout window) for these different timeout windows, for the same representative workload phase. As can be seen, there is a clear correlation between the relative increase in row hits vs. conflicts and the speedup. This is because the activation and precharge penalties are similar to each other in most typical DDR technologies. Hence, the latency benefit of converting a potential row miss to a row hit and the latency increase from converting a row miss to a row conflict are similar. Therefore, measuring/estimating the row hit and conflict counts for different timeout windows serves as an effective mechanism to capture the latency impact of different timeout windows.

We observe that the row hit and conflict counts for different timeout values can be determined by projecting each request to be a row hit, miss or a conflict for each of the different timeout values. Figure 3 shows an example command sequence for three different timeout windows, where 100 is the current timeout window. The first CAS is treated similarly for all timeout windows. The second CAS, though, would become a miss with a shorter timeout window of 50 cycles, since it arrives more than 50 cycles after the first CAS. It would still remain a hit though for larger timeout windows. The third CAS, on the other hand, would hit in the row buffer with a larger timeout window of 200 cycles, since it arrives within 200 cycles of the second CAS. We propose to project such hit, miss and conflict

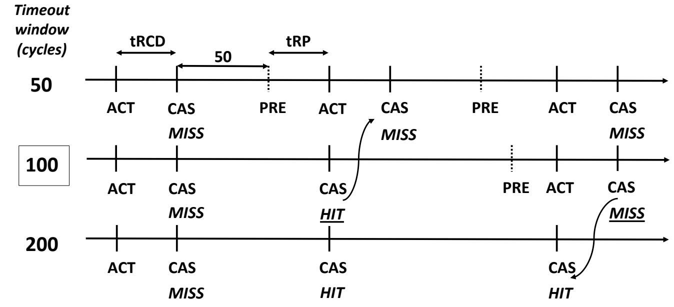


Figure 3: Hit conversions for requests to the same bank, for different timeout windows.

outcomes for each request, for different timeout windows. These projections enable us to determine the best timeout window for different program phases, as we describe in detail in Section 4.1.

3.2 Accounting for Local Variations

We observe that a small set of DRAM rows experience a large number of consecutive accesses. In such cases, holding the row open until a read to the next row arrives can enable these accesses to hit in the row buffer.

While the scoreboard mechanism described in Section 3.1 captures global DRAM row access characteristics, some rows might experience a very different access pattern and the globally determined timeout window might not be effective in capturing the inherently high row locality available for such rows. We seek to identify such rows and employ a different, localized timeout window for such rows alone, that is better tuned to capture row hits while not increasing row conflicts significantly for such rows.

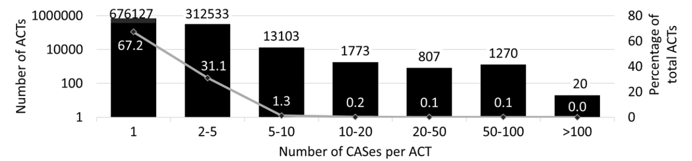


Figure 4: Histogram of number of CASes per ACT, averaged over the benchmark suite. Several instances of row activation are associated with a high number of subsequent consecutive accesses. A closed page policy is equivalent to setting the number of CASes per ACT to unity.

Specifically, we observe that some rows experience a large number of consecutive accesses, which could be row hits if the row were held open long enough, but would result in row misses if it were closed early. Figure 4 shows a histogram of the number of CASes to an open row, across all our workloads, when the row is kept open until an access to a different row arrives. As can be seen, a small set of rows experience significantly higher number of CASes as compared to other rows. The global scoreboard mechanism would pick a timeout window that would maximize the number of row hits overall without increasing the number of row conflicts significantly. However, this globally determined timeout window might

not capture row hits effectively for rows that tend to experience a larger than average number of consecutive accesses.

We propose to identify such rows that experience a large number of consecutive accesses and prevent such rows from being closed even after the timeout window expires. We detect such rows by tracking instances when a row is closed upon expiration of the timeout window and the same row is opened again. Once we detect such a row, we place it in a row history structure and prevent the row from being closed upon expiration of the timeout window, thereby enabling more row hits to such rows. This selective row exclusion mechanism enables us to capture differences in local row-level access behavior from the global row access characteristics and effectively augment the scoreboard scheme. Furthermore, since we track row-level behavior only selectively for a small number of rows (a 64 entry row tracking structure is effective in our evaluations), the additional hardware cost we incur is minimal.

4 MECHANISM

In this section, we describe the details of our proposed row management mechanisms. First, we describe our global scoreboard scheme that predicts how long DRAM rows should remain open after an access, in order to capture possible accesses to the same row (row hits), while not increasing the number of row conflicts (Section 4.1). Next, we describe a local scheme that predicts which rows would benefit from staying open for longer, beyond what the global scoreboard scheme dictates (Section 4.2).

4.1 Global Scoreboarding

Overview. Figure 5 shows a high level depiction of our scoreboarding mechanism to determine the timeout window. The scoreboard is maintained at the memory controller at a per-bank level and determines the timeout window for all rows in that bank. For every incoming request, the hit, miss or conflict status is determined for each of the different timeout windows being tracked and evaluated in the scoreboard, as we describe below. The scoreboard is then updated accordingly. These row hit and conflict counts are then used to periodically determine the timeout window, as we describe below.

Scoreboard update. Upon each request, the possible row hit, miss, conflict status of the request is determined for each of the timeout windows tracked in the scoreboard. We seek to maximize row hits without increasing row conflicts significantly, since doing so effectively reduces/hides the activate and precharge latencies. Hence, we track only the row hit and conflict counts in the scoreboard. The hit, conflict status is projected for each request, for different timeout windows, using the basic idea illustrated in Figure 3. Specifically, a row miss request could be converted to a row hit request for larger timeout windows than the currently employed timeout window. On the other hand, a row conflict request could be converted to a row miss when the projected timeout window is shorter than the current timeout window. These conversions are projected for different timeout windows, with respect to the time of the previous CAS (as shown in Figure 3).

The algorithm in the flow chart in Figure 6 shows the details of how these conversions are projected for a request, for all timeout windows. Upon an activate command, if the activate is to a different row than the previously open row, it would result in a row conflict for a specific timeout window if there would not have been enough time to safely (timing constraint wise) issue a precharge after the given timeout window expires. This is evaluated for each timeout window, with lower values of the timeout window more likely to result in a row miss, while larger values of the timeout window are more likely to result in a row conflict. On the other hand, if the activate command is to the same row as previously open, it would have resulted in a row hit for a specific timeout window had the time between the current request and the last CAS been smaller than that timeout window. Similarly, for a CAS command, the time between the current request and the last CAS would determine if the command would result in a row hit or a row miss.

Scoreboard use and reset. The row hit and conflict counts that are tracked in the scoreboard are used to periodically evaluate the timeout window, as shown in the procedure below. The timeout window with the largest difference between the hit count increase and the conflict count increase reflects the timeout window that maximizes row hits, without increasing row conflicts significantly. Hence, we pick this timeout window for the next N requests. We further define a parameter, *variation threshold*, to only update the timeout if there was a substantial benefit projected.

```
Repeat Every N Requests:
  T = currTimeout
  # Compute hit and conflict increments from scoreboard
  for t in {timeout windows in scoreboard}:
     hitsIncr [t] = hits[t] − hits[T]
     conflictsIncr [t] = conflicts [t] − conflicts [T]
     hits[t] = conflicts [t] = 0

  # Pick the timeout that maximizes hits,
  # and minimizes conflicts
  nextT = argmax (hitsIncr [] − conflictsIncr [])

  # If variation is not substantial, do not change timeout
  if max(hitsIncr [] − conflictsIncr []) <
     (1 + variationThreshold) * min(hitsIncr [] −
          conflictsIncr []):
     nextT = T
  # nextT represents the timeout for the next N requests
```

4.2 Local Row Exclusion

Some rows tend to be closed upon expiration of the timeout window only to be opened immediately after several times over consecutively, as shown in Section 3.2. The timeout window determined by the global scoreboard mechanism would not be able to capture the potential row hits effectively for such rows. Our row exclusion policy strives to identify such rows and prevent such rows from being closed upon expiration of the timeout window. It consists of two key components - detection of such rows and exemption of such rows from closure upon the timeout window expiration, in the future.

Detection. We detect rows that tend to be closed due to timeout expiration and opened again immediately, by tracking the last open row and if it was closed due to timeout expiration. If an activated row is the same as the previous row and was closed due to the

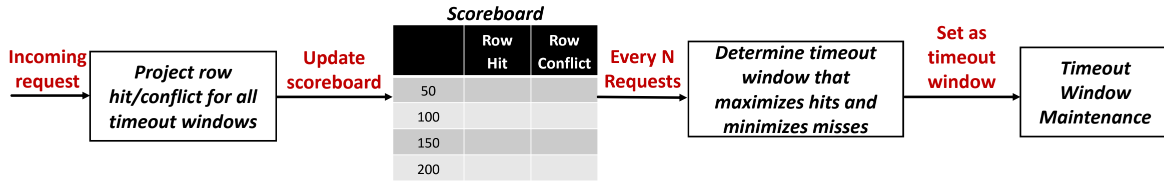


Figure 5: Overview of the scoreboard mechanism

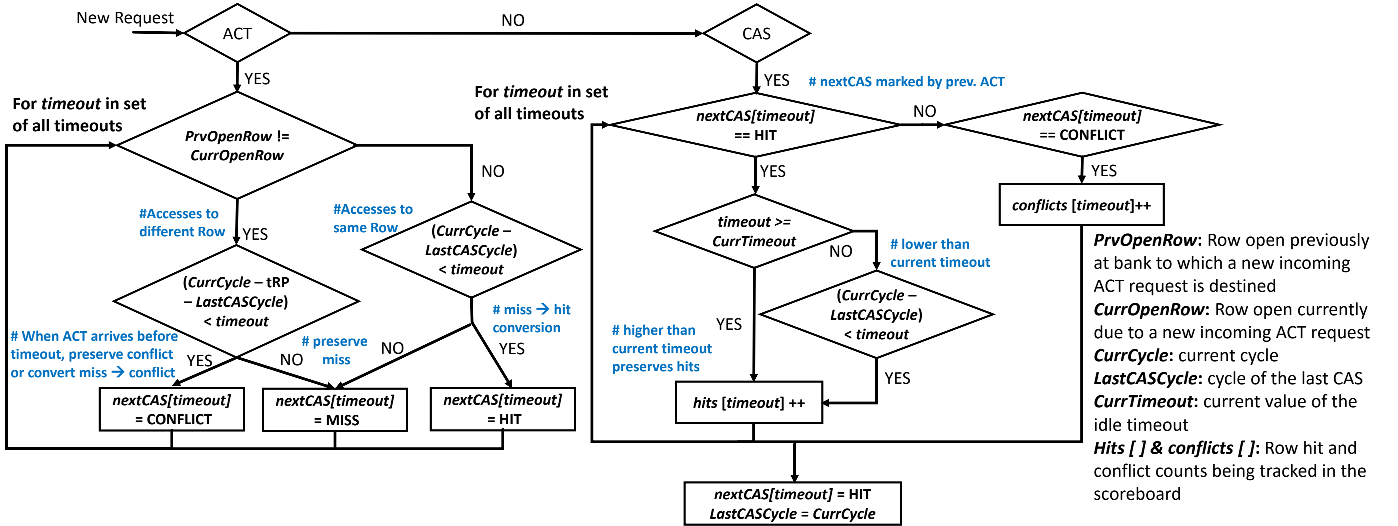


Figure 6: Scoreboard update

expiration of the timeout window the previous time it was open, it is placed in a row exclusion store. We explore two variants of tracking rows in the row exclusion store – i) track the full address of the row, including the channel, rank, bank information, ii) track only the row address, since a physical page could be distributed across the same row in multiple channels, banks due to the interleaving policy and rows containing the same physical page could behave similarly (we call this row aliasing). We observe that tracking the entire address provides better performance (Section 7.4) without increasing cost significantly, since even a small row exclusion store (64 entries) is effective.

Exemption from timeout based closure. The rows tracked in the row exclusion store are exempt from closure upon expiration of the timeout window. Specifically, when the timeout window for a row has expired and the row is being considered for closure, the row exclusion store is looked up. If the row is present in the row exclusion store, it is not closed right away. Instead, the row is only closed when a request to another row in the same bank arrives.

Managing the row exclusion store. We explored multiple replacement policies for when the row exclusion store is full and a new row needs to be placed in it. We observe that a policy that picks a row that caused the most recent row conflict as a result of its row exclusion to provide the best performance.

5 HARDWARE OVERHEAD

In this section, we provide details of the implementation overhead of our proposed mechanisms. Both of our mechanisms are intended to be integrated into the memory controller, with no modification required to commodity DRAM itself.

5.1 Scoreboard

The scoreboard itself consists of N entries per bank, with each entry storing three values:

- Timeout value for which the scoreboard entry is maintained. [8 bits].
- Number of (projected/measured) row hits during a window of operation, for this timeout. [16 bits].
- Number of (projected/measured) row conflicts during a window of operation, for this timeout. [16 bits].

This amounts to a total of 5 bytes per scoreboard entry, per bank. Our evaluated configuration (Section 6) implements 2 channels with 8 banks each, thereby rendering a storage overhead of 560 bytes for $N = 7$ entries.

In updating the scoreboard entries (Figure 6), the memory controller needs to store the following, for each bank:

- Previous open row address. [16 bits].
- Last CAS cycle. [32 bits].
- Next CAS type. [(N =)7 bits].

Table 2: Baseline Configuration

Parameter	Dimensions
OoO Fetch/Retire width/ROB	4/4/128
L1 size/associativity	32kB/8-way
L2 size/associativity	256kB/8-way
L3 size/associativity	2MB/8-way
Load to use latency L1/L2/L3	4/4+12/4+12+31 cycles
MSHR per cache	16
Caching policy	Non-inclusive/LRU
Core-Memory frequency ratio	8:3
DRAM JEDEC Standard	LPDDR4 (2ch: 1ra/8ba each)
Address interleaving	Channel (RoBaRaCoCh)
Scheduling policy	FRFCFS_prioritizeHit
Scoreboard entries	7 per bank
Timeouts	50/100/150/200/300/400/800
Variation threshold	3%
Update window	30000 requests
Row exclusion store	64 entries
Replacement policy	Most recent to cause conflict

This amounts to an additional overhead of 110 bytes, for a total footprint of 670 bytes for the scoreboard.

5.2 Row exclusion

We now quantify the storage overhead of a 64-entry row exclusion store, with and without row aliasing (Section 4.2):

- Previous open row address, already available due to scoreboard.
- Row closure type. [1 bit].
- Counter to assist replacement policy. [6 bits per entry].
- Excluded row address. [16 bits (with row aliasing) or 20 bits (without) per entry].

Maintained at a channel granularity, this amounts to a total overhead of at most 432 bytes. *Together with a scoreboard, the total overhead is a little over a kilobyte.*

The logic to implement either of our mechanisms is achievable with a small set of comparators and other simple logic gates.

6 METHODOLOGY

We extend ramulator [20] to simulate our scoreboarding and row exclusion mechanisms at the memory controller, in addition to other row closure mechanisms that we compare against. Ramulator [20] models the DRAM main memory system in detail. It employs a simple out of order core frontend that is driven by a Pin [23] tool. The pin traces collected store both, the memory instructions as well as the number of non-memory-ops between successive memory ops. The front-end does not stall due to data dependencies originating from non-memory-ops and prefetches are turned off. The cache hierarchy consists of non-inclusive L1, L2, L3 caches. We summarize our baseline system and mechanism parameters in Table 2. We evaluate single-core traces from the SPEC 2006 benchmark suite [36] and use multi-programmed traces from the same suite for evaluating two-core systems in Section 7.3.

7 EVALUATION

7.1 Performance Results

Figure 7 shows the performance of our combined scoreboard and row exclusion mechanisms against several variants of open, closed and fixed timeout policies. We draw two observations. First, we increase the idle row closure timeout from 0 (*closed*), to 100 (*timeout_100*), to 200 (*timeout_200*) and then all the way to infinity (*opened*). We observe that increasing the timeout yields gains initially owing to increased row hits rather than row misses. However, with an open row policy, row conflicts begin to negate the benefits of row hits. The mechanisms proposed in this paper strive to minimize the relative number of row conflicts while also trying to achieve as many conversions to row hits as possible. Second, our proposed mechanisms not only provide improved performance over canonical idle row closure mechanisms on average, we provide the added benefit of not causing slowdowns when the applications do not benefit from such optimizations. This is possible because of the dynamic, adaptive nature of our mechanisms, both at the global and local levels.

7.2 Comparison to Prior Work

Static timeouts are most ubiquitous although there have been proposals that aim to update the timeout dynamically to a certain extent (Sections 2.3 and 8). This is mainly because such proposals provide little benefit or are expensive to implement.

Instead, more successful proposals try and capture global behavior or local behavior of idle row closure. In this section, we present a more detailed look into examples that capture global behavior – *smith* [26], local behavior – *abp* [1] as well as the scheme that is the most performant among the prior works evaluated – *ldp* [34].

Recall of prior works. We first present a brief recall of these prior mechanisms before we present quantitative comparisons to them.

Smith employs a 2-bit saturating counter per bank, that is incremented/decremented respectively on row hits/row conflicts. The value of the saturating counter is used to predict if a row hit or a row conflict would occur and the DRAM row is correspondingly held open or closed following a memory request.

Abp is a row management scheme that tracks the number of potential row hits *at a per-row level.* ABP employs a tracking structure with a large number of entries (8192) to track the number of row hits seen to each of these rows the previous time the row was accessed. The row is held open until the same number of requests is seen. This tracking structure is updated when the number of requests to a row is different from the predicted value.

Ldp proposes to use i) *per-row saturating counters* to predict if a row is likely to see a hit or a conflict upon the next access (similar to *smith, but* at a per-row level, for *every* row of memory), ii) per-bank counters that track, on average, when a row is not likely to see any more accesses. These two predictors are employed together to make row management decisions.

Performance vs. area overhead. Figure 9 summarizes the performance and hardware cost of these prior works and our proposal, in a pareto-style chart. Four key conclusions are in order.

First, while the implementation overhead of *smith* is low, it's performance is also low. This is because of its inability to capture local behavior.

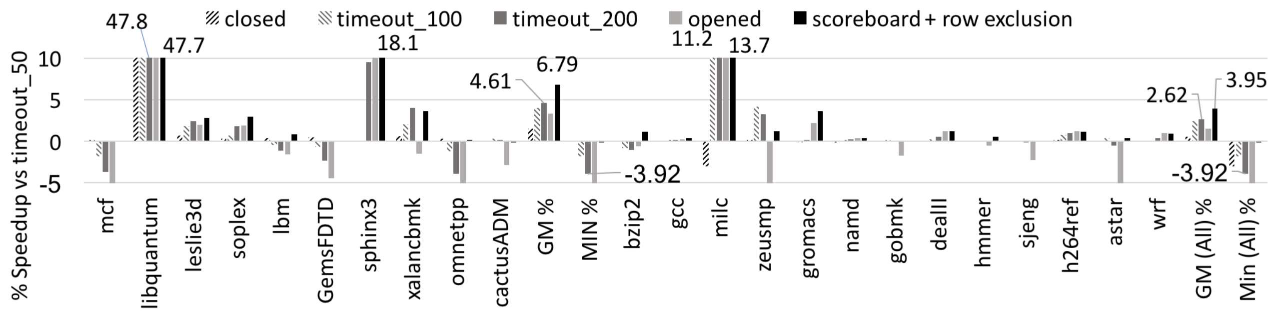


Figure 7: Our mechanisms outperform static idle row closure timeouts and do so without negatively impacting the performance of any single workload. Workloads on the left are memory intensive, while workloads on the right are memory-non-intensive.

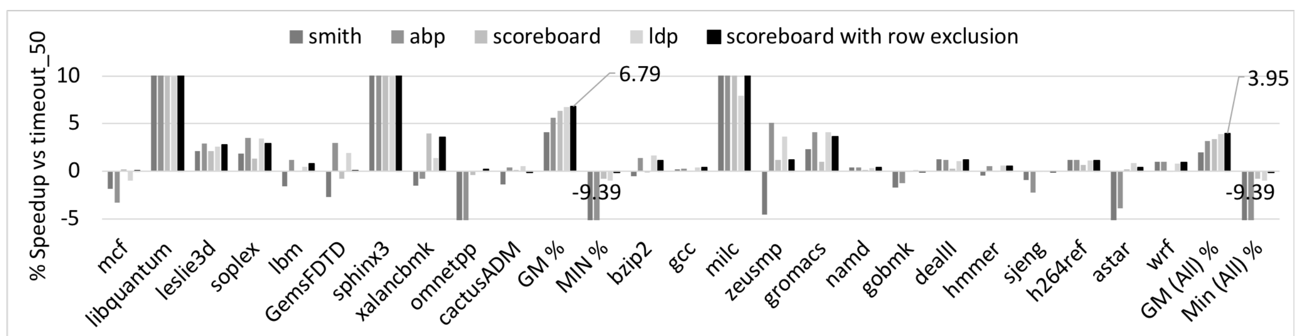


Figure 8: Our mechanisms outperform prior work that adapt to global or local behavior individually. Workloads on the left are memory intensive, while workloads on the right are memory-non-intensive.

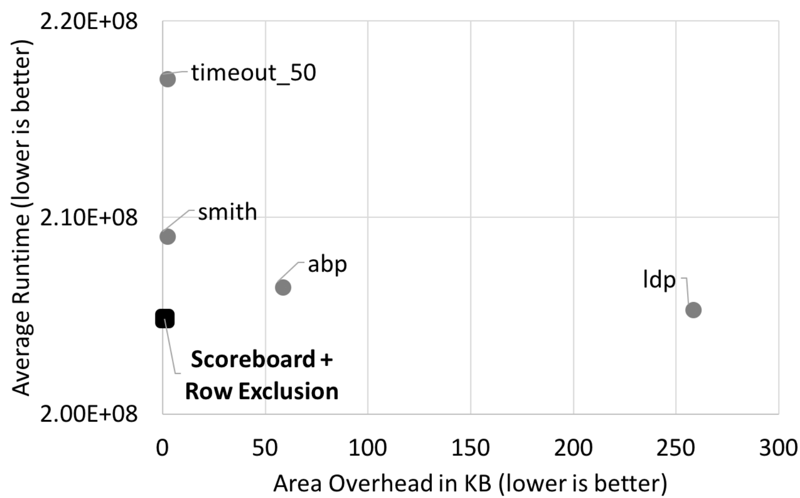


Figure 9: Not only do our mechanisms outperform prior work that adapt to global or local behavior, but also do so at an overhead that is insignificant.

Second, *abp* provides improved performance, however, it comes at an increased overhead - 8K entries per channel (as published) requiring over 56 kilobytes of total storage (20 bits for row address,

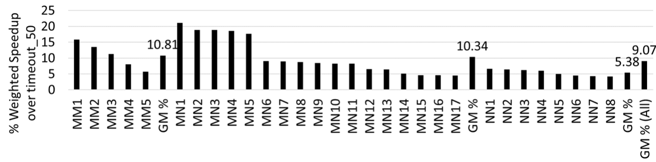


Figure 10: Weighted speedups are calculated for the workload mixes when used with scoreboarding and row inclusion, with timeout_50 as the baseline. Workload mixes are separated by memory intensity, where *M* stands for memory intensive and *N* stands for non-memory-intensive workload.

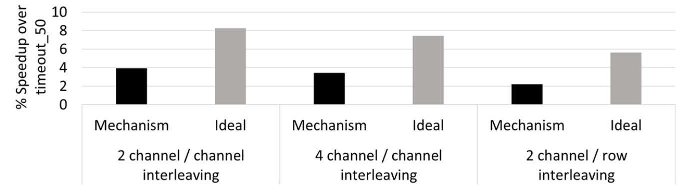


Figure 11: Impact of memory system configuration on our proposed mechanisms. Scoreboard and row exclusion provide application speedup, but their extent depends upon how sensitive the configuration is to optimal idle row closure.

8 bits for threshold, and ignoring the overhead due to LRU counters). Note that this is over 50× the storage overhead than our proposal. Furthermore, *abp* causes significant performance degradation in certain workloads, as shown in Figure 8.

Third, *ldp* extends this tradeoff to achieve more improved performance, again, at an even higher overhead. Recall that the liveness predictor alone requires a 2-bit counter per row, therefore causing the overhead to be over 256 kilobytes of storage.

Finally, our scoreboarding and history buffer proposals are not only effective in improving performance, but do so at a very low hardware cost. In fact, we achieve performance that is on par or better than the fastest prior work, at an overhead that is less than $\frac{1}{250}^{th}$ of its cost.

7.3 Sensitivity to System Parameters

We simulate multi-programmed workloads on a 2 core system and find that our proposed mechanisms are capable of yielding significant benefits under a wide variety of workload mixes, as shown in Figure 10. The workload mixes are arbitrarily chosen, but are classified based on their memory intensity to include all combinations of memory-intensive and memory-non-intensive applications.

Figure 11 depicts the impact of changing the memory configuration. As our mechanisms target DRAM latency, we see similar performance improvements upon increasing the number of channels. Using a suboptimal address interleaving policy (such as row interleaving - ChRaBaRoCo) that does not exploit sufficient memory level parallelism causes a significant performance slowdown of the entire system, thereby also lowering the impact idle row closure mechanisms have on system performance. In conclusion, our proposed mechanisms provide for application speedup irrespective of memory configuration, although the relative impact depends upon the sensitivity of the configuration to idle row closure.

7.4 Sensitivity to Mechanism Parameters

Table 3 summarizes the relative impact that row exclusion as well as row aliasing (if row exclusion is used) has on performance. Using scoreboard alone (without row exclusion) renders superior performance when compared to both *smith* and *abp*. However, further capturing local behavior via the proposed row exclusion mechanism yields an added performance boost that is sufficient to outperform the fastest prior scheme. We observe that storing the entire address, including channel, bank information avoids aliasing and provides better performance.

Table 3: Impact of row exclusion and row aliasing

% Speedup over timout_50	No row exclusion	Row Aliasing	No Row Aliasing
Memory Intensive	6.34%	6.41%	6.79%
All Benchmarks	3.35%	3.47%	3.95%

While we evaluate several replacement policies, we find that a policy that eliminates the most recent entry that excluded a row from idle row closure only to cause a row conflict after the fact, to be best performing. For scoreboard parameters, we find that varying the variation threshold, update window and number of scoreboard entries still render superior performance to other schemes. We omit detailed results for these as they do not provide any further insight than what has already been discussed in this paper thus far, to the community.

However, we wish to point out the following methodology in terms of selecting scoreboard entries. With the ACT and PRE latencies in the order of 25-30 cycles, it is only logical to space the timeout windows by at least 50 cycles or so. Our simulations indicated that this was indeed beneficial. Later, to explore the impact of larger jumps, scoreboard entries with much higher timeout values were also included, for an added performance boost with little-to-no additional cost.

8 RELATED WORK

DRAM row management policies. We describe the closest prior DRAM row management policies qualitatively in Section 2.3 and compared to them quantitatively in Section 7.2. Patents on DRAM row management have proposed the high level idea of a hill climbing scheme [16, 29] that increments or decrements the timeout window based on row hit and conflict counts. However, these, being patents lack concrete mechanisms and descriptions. In principle, the notion of hill climbing suffers from the fundamental challenge of being stuck in local minima. Our scoreboard mechanism has a much more holistic view of the benefits/pitfalls of potentially employing each timeout window when it makes the decision of picking the timeout window.

HAPPY [9] builds on the concepts introduced by other mechanisms; their focus being on reducing implementation overhead by aggregating performance counters at the granularity of address bit positions instead of maintaining counters for each DRAM row. While HAPPY is complementary to our scoreboard mechanism and could be augmented with it, it would not be useful to combine it with our row exclusion mechanism as the fine grained behavior leveraged by row exclusion would not be captured by HAPPY encoding. Loss of fine grained sensitivity is indeed acknowledged in their paper.

Relaxing DRAM timing parameters. Prior work [3, 21] observe that DRAM timing parameters are derived by building large margins that account for temperature and process variations. Hence, they propose to relax timing parameters when the operating conditions permit to do so. Hassan et al. [11] observe that recently accessed cells store higher charge due to their capacitive nature and relax timing constraints for accesses to recently accessed rows. Scoreboard and row exclusion are complementary to these techniques and can be used along with them to better tackle DRAM latency.

Prefetching to tackle latency. A large body of prior work [7, 8, 12, 14, 15, 25, 32] has explored various different kinds of prefetchers that understand access patterns and exploit knowledge of access patterns to prefetch data from memory into the caches, ahead of when the data is accessed by a demand request. Our proposed mechanisms are complementary to such prefetching. Specifically, our proposed DRAM row management mechanisms can mitigate the DRAM precharge and activation latency latencies for prefetch and demand requests alike, thereby enabling more timely prefetches.

Changes to DRAM internals. Kim et al. propose sub-array level parallelism [19] that enables access to subarrays in parallel, resulting in fewer bank conflicts and lower latencies. Lee et al. propose tiered-latency DRAM (TL-DRAM) [22], a scheme that partitions a sub-array into two regions using an isolation transistor and enables faster access to the closer region. Several other previous works [4, 5, 10, 30, 31] also propose changes to DRAM architecture/organization to enable latency reduction. All of these techniques, unlike our scoreboard and row exclusion mechanisms, require changes to DRAM internals, which makes them hard to adopt, given the cost conscious nature of DRAM manufacturing. Furthermore, our proposals can be employed in conjunction with these schemes to enable even better latency reduction and performance enhancement.

Managing contention at the memory controller. Prior works [18, 24, 27, 35, 38] have tackled the problem of contention between multiple applications' requests at the memory controller. Our proposals, on the other hand, tackle the inherent DRAM precharge and activation latencies, rather than queueing from contention and hence, can be effectively combined with these techniques.

In summary, tackling the fundamental DRAM precharge and activation latencies is orthogonal to the multitude of DRAM improvements and memory controller optimizations that have been proposed over the past couple of decades. Furthermore, because the effects of queuing delay are often exacerbated by poor service delays, lowering the fundamental latency-bound inefficiencies can help compound the benefits due to bandwidth-centric improvements.

9 CONCLUSION

We tackle the problem of DRAM access latency, which is a critical performance bottleneck. We observe that row management, specifically, decisions on how long a row is held open play a key role in hiding/avoiding the activation and precharge latencies, which are key components of DRAM access. We propose two schemes that tackle the problem of DRAM row management both globally across rows in a bank and locally at the individual row level for a small set of rows that require different treatment. Our proposed schemes are effective in tackling the DRAM access latency and can act as effective substrates for current and future memory systems.

REFERENCES

[1] Manu Awasthi, David Nellans, Rajeev Balasubramonian, and Al Davis. 2011. Prediction Based DRAM Row-Buffer Management in the Many-Core Era. In *PACT*.

[2] Matthew Blackmore. 2013. A quantitative analysis of memory controller page policies. (2013).

[3] Karthik Chandrasekar, Sven Goossens, Christian Weis, Martijn Koedam, Benny Akesson, Norbert Wehn, and Kees Goossens. 2014. Exploiting Expendable Process-margins in DRAMs for Run-time Performance Optimization. In *DATE*.

[4] K. K. Chang, P. J. Nair, D. Lee, S. Ghose, M. K. Qureshi, and O. Mutlu. 2016. Low-Cost Inter-Linked Subarrays (LISA): Enabling fast inter-subarray data movement in DRAM. In *HPCA*.

[5] Jungwhan Choi, Wongyu Shin, Jaemin Jang, Jinwoong Suh, Yongkee Kwon, Youngsuk Moon, and Lee-Sup Kim. 2015. Multiple Clone Row DRAM: A Low Latency and Area Optimized DRAM. In *ISCA*.

[6] Erik P DeBenedictis, Jeanine Cook, Sriseshan Srikanth, and Thomas M Conte. 2017. Superstrider associative array architecture: Approved for unlimited unclassified release: SAND2017-7089 C. In *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*. IEEE, 1–7.

[7] E. Ebrahimi, O. Mutlu, and Y. N. Patt. 2009. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In *HPCA*.

[8] John W. C. Fu, Janak H. Patel, and Bob L. Janssens. 1992. Stride Directed Prefetching in Scalar Processors. In *MICRO*.

[9] Mohsen Ghasempour, Aamer Jaleel, Jim D. Garside, and Mikel Luján. 2016. HAPPY: Hybrid Address-based Page Policy in DRAMs. In *MEMSYS*.

[10] Nagendra Dwarakanath Gulur, R. Manikantan, Mahesh Mehendale, and R. Govindarajan. 2012. Multiple Sub-row Buffers in DRAM: Unlocking Performance and Energy Improvement Opportunities. In *ICS*.

[11] Hasan Hassan, Gennady Pekhimenko, Nandita Vijaykumar, Vivek Seshadri, Donghyuk Lee, Oguz Ergin, and Onur Mutlu. 2016. ChargeCache: Reducing DRAM latency by exploiting row access locality. In *HPCA*.

[12] Ibrahim Hur and Calvin Lin. 2006. Memory Prefetching Using Adaptive Stream Detection. In *MICRO*.

[13] Radhika Jagtap, Matthias Jung, Wendy Elsasser, Christian Weis, Andreas Hansson, and Norbert Wehn. 2017. Integrating DRAM power-down modes in gem5 and quantifying their impact. In *Proceedings of the International Symposium on Memory Systems*. ACM, 86–95.

[14] Victor Jiménez, Roberto Gioiosa, Francisco J. Cazorla, Alper Buyuktosunoglu, Pradip Bose, and Francis P. O'Connell. 2012. Making Data Prefetch Smarter: Adaptive Prefetching on POWER7. In *PACT*.

[15] Norman P. Jouppi. 1990. Improving Direct-mapped Cache Performance by the Addition of a Small Fully-associative Cache and Prefetch Buffers. In *ISCA*.

[16] O. Kahn and J. Wilcox. 2004. Method for Dynamically Adjusting a Memory Page Closing Policy. U.S. Patent Number 6799241-B2.

[17] Mushfique Khurshid, Mohit Chainani, Alekhya Perugupalli, and Rahul Srikumar. 2012. Stride and Global History Based DRAM Page Management. In *JWAC*.

[18] Yoongu Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. 2010. Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior. In *MICRO*.

[19] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu. 2012. A Case for Subarray-Level Parallelism (SALP) in DRAM. In *ISCA*.

[20] Yoongu Kim, Weikun Yang, and Onur Mutlu. 2016. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE CAL* 15, 1 (Jan. 2016).

[21] D. Lee, Y. Kim, G. Pekhimenko, S. Khan, V. Seshadri, K. Chang, and O. Mutlu. 2015. Adaptive-Latency DRAM: Optimizing DRAM Timing for the Common-Case. In *HPCA*.

[22] D. Lee, Y. Kim, V. Seshadri, J. Liu, L. Subramanian, and O. Mutlu. 2013. Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture. In *HPCA*.

[23] C. K. Luk. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*.

[24] S. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda. 2011. Reducing Memory Interference in Multi-Core Systems via Application-Aware Memory Channel Partitioning. In *MICRO*.

[25] S. Palacharla and R. E. Kessler. 1994. Evaluating Stream Buffers As a Secondary Cache Replacement. In *ISCA*.

[26] Seong-Il Park and In-Cheol Park. 2003. History-based memory mode prediction for improving memory performance. In *ISCAS*.

[27] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. 2000. Memory Access Scheduling. In *ISCA*.

[28] T. Rokicki. 2002. Method and Computer System for Speculatively Closing Pages in Memory. U.S. Patent Number 6389514-B1.

[29] B. Sander, P. Madrid, and G. Samus. 2005. Dynamic Idle Counter Threshold Value for Use in Memory Paging Policy. U.S. Patent Number 6976122-B1.

[30] Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2013. RowClone: Fast and Efficient In-DRAM Copy and Initialization of Bulk Data. In *MICRO*.

[31] Vivek Seshadri, Thomas Mullins, Amirali Boroumand, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2015. Gather-scatter DRAM: In-DRAM Address Translation to Improve the Spatial Locality of Non-unit Strided Accesses. In *MICRO*.

[32] Manjunath Shevgoor, Sahil Koladiya, Rajeev Balasubramonian, Chris Wilkerson, Seth H. Pugsley, and Zeshan Chishti. 2015. Efficiently Prefetching Complex Address Patterns. In *MICRO*.

[33] Sriseshan Srikanth, Thomas M Conte, Erik P DeBenedictis, and Jeanine Cook. 2017. The Superstrider Architecture: Integrating Logic and Memory Towards Non-Von Neumann Computing. In *Rebooting Computing (ICRC), 2017 IEEE International Conference on*. IEEE, 1–8.

[34] Vladimir V Stankovic and Nebojsa Z Milenkovic. 2005. Dram controller with a close-page predictor. In *Computer as a Tool, 2005. EUROCON 2005. The International Conference on*, Vol. 1. IEEE, 693–696.

[35] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu. 2013. MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems. In *HPCA*.

[36] The Standard Performance Evaluation Corporation [n. d.]. *Welcome to SPEC*. The Standard Performance Evaluation Corporation. http://www.specbench.org/.

[37] Ying Xu, Aabhas S. Agarwal, and Brian T. Davis. 2009. Prediction in Dynamic SDRAM Controller Policies. In *SAMOS*.

[38] Y. Zhou and D. Wentzlaff. 2016. MITTS: Memory Inter-arrival Time Traffic Shaping. In *ISCA*.