Improving DRAM Performance by Parallelizing Refreshes with Accesses

Kevin Kai-Wei Chang kevincha@cmu.edu
Donghyuk Lee donghyu1@cmu.edu
Zeshan Chishti† zeshan.a.chishti@intel.com
Alaa R. Alameldeen† alaa.r.alameldeen@intel.com
Chris Wilkerson† chris.wilkerson@intel.com
Yoongu Kim yoongukim@cmu.edu
Onur Mutlu onur@cmu.edu

Carnegie Mellon University †Intel Labs

Abstract

Modern DRAM cells are periodically refreshed to prevent data loss due to leakage. Commodity DDR (double data rate) DRAM refreshes cells at the rank level. This degrades performance significantly because it prevents an entire DRAM rank from serving memory requests while being refreshed. DRAM designed for mobile platforms, LPDDR (low power DDR) DRAM, supports an enhanced mode, called per-bank refresh, that refreshes cells at the bank level. This enables a bank to be accessed while another in the same rank is being refreshed, alleviating part of the negative performance impact of refreshes. Unfortunately, there are two shortcomings of per-bank refresh employed in today's systems. First, we observe that the per-bank refresh scheduling scheme does not exploit the full potential of overlapping refreshes with accesses across banks because it restricts the banks to be refreshed in a sequential round-robin order. Second, accesses to a bank that is being refreshed have to wait.

To mitigate the negative performance impact of DRAM refresh, we propose two complementary mechanisms, DARP (Dynamic Access Refresh Parallelization) and SARP (Subarray Access Refresh Parallelization). The goal is to address the drawbacks of per-bank refresh by building more efficient techniques to parallelize refreshes and accesses within DRAM. First, instead of issuing per-bank refreshes in a round-robin order, as it is done today, DARP issues per-bank refreshes to idle banks in an out-of-order manner. Furthermore, DARP proactively schedules refreshes during intervals when a batch of writes are draining to DRAM. Second, SARP exploits the existence of mostly-independent subarrays *within a bank. With minor modifications to DRAM organization, it allows a bank to serve memory accesses to an idle subarray while another subarray is being refreshed. Extensive evaluations on a wide variety of workloads and systems show that our mechanisms improve system performance (and energy efficiency) compared to three state-of-the-art refresh policies and the performance benefit increases as DRAM density increases.*

1. Introduction

Modern main memory is predominantly built using *dynamic random access memory* (DRAM) cells. A DRAM cell consists of a capacitor to store one bit of data as electrical charge. The capacitor leaks charge over time, causing stored data to change. As a result, DRAM requires an operation called *refresh* that periodically restores electrical charge in DRAM cells to maintain data integrity.

Each DRAM cell must be refreshed periodically every *refresh interval* as specified by the DRAM standards [11, 14]. The exact refresh interval time depends on the DRAM type (e.g., DDR or LPDDR) and the operating temperature. While DRAM is being refreshed, it becomes unavailable to serve memory requests. As a result, refresh latency significantly degrades system performance [24, 31, 33, 41] by delaying in-flight memory requests. This problem will become more prevalent as DRAM density increases, leading to more DRAM rows to be refreshed within the same refresh interval. DRAM chip density is expected to increase from 8Gb to 32Gb by 2020 as it doubles every two to three years [10]. Our evaluations show that DRAM refresh, as it is performed today, causes an average performance degradation of 8.2% and 19.9% for 8Gb and 32Gb DRAM chips, respectively, on a variety of memory-intensive workloads running on an 8-core system. Hence, it is important to develop practical mechanisms to mitigate the performance penalty of DRAM refresh.

There are two major ways refresh operations are performed in modern DRAM systems: *all-bank refresh (or, rank-level refresh)* and *per-bank refresh*. These methods differ in what levels of the DRAM hierarchy refresh operations tie up to. A modern DRAM system is organized as a hierarchy of ranks and banks. Each rank is composed of multiple banks. Different ranks and banks can be accessed independently. Each bank contains a number of rows (e.g., 16-32K in modern chips). Because successively refreshing *all* rows in a DRAM chip would cause very high delay by tying up the entire DRAM device, modern memory controllers issue a number of refresh commands that are evenly distributed throughout the refresh interval [11, 14, 23, 24, 33]. Each refresh command refreshes a small number of rows.[1] The two common refresh methods of today differ in where in the DRAM hierarchy the rows refreshed by a refresh command reside.

In *all-bank refresh*, employed by both commodity DDR and LPDDR DRAM chips, a refresh command operates at the rank level: it refreshes a number of rows in *all* banks of a rank concurrently. This causes every bank within a rank to be unavailable to serve memory requests until the refresh command is complete. Therefore, it degrades performance significantly, as we demonstrate in Section 3 and as others have demonstrated [24, 31, 33, 41]. In *per-bank refresh*, employed by LPDDR DRAM [14, 28] as an alternative refresh mode, a refresh command operates at the bank level: it refreshes a

[1] The time between two refresh commands is fixed to an amount that is dependent on the DRAM type and temperature.

978-1-4799-3097-5/14/$31.00 ©2014 IEEE

number of rows in only a single bank of a rank at a time.[2] This enables a bank to be accessed while another in the same rank is being refreshed, alleviating part of the negative performance impact of refreshes (as we show in Section 3). Unfortunately, per-bank refresh suffers from two shortcomings that limit the ability of DRAM to serve requests while refresh operations are being performed. First, we observe that the per-bank refresh scheduling scheme does not exploit the full potential of overlapping refreshes with accesses across banks because it restricts the banks to be refreshed in a strict sequential *round-robin order* [14]. Second, accesses to a bank that is being refreshed still have to wait as there is no way today to perform a refresh and an access to the same bank concurrently. We find that, due to these shortcomings, a significant performance degradation still exists with per-bank refresh (Section 3).

Our goal is to alleviate the shortcomings of per-bank refresh by enabling more efficient parallelization of refreshes and accesses within DRAM. The major ideas are to (1) provide a more flexible scheduling policy for traditional per-bank refresh and (2) exploit the internal *subarray* structure of a bank to enable parallelization of refreshes and accesses within a bank. To this end, we propose two complementary techniques.

The first technique, *Dynamic Access Refresh Parallelization (DARP)*, is a new refresh scheduling policy based on two key ideas: *out-of-order per-bank refresh* and *write-refresh parallelization*. The first idea enables the memory controller to specify an idle bank to be refreshed as opposed to the state-of-the-art per-bank refresh policy that refreshes banks in a strict round-robin order. By monitoring the bank request queues' occupancies, DARP avoids refreshing a bank that has pending memory requests and instead refreshes an idle bank to maximize parallelization of refreshes and accesses. The second idea hides refresh latency with write accesses by proactively scheduling per-bank refreshes during the time period when banks are serving write requests. There are two reasons why we attempt to parallelize refresh operations with write accesses. First, modern systems typically buffer write requests in memory controllers and drain them to DRAM in a batch to mitigate the *bus turnaround* penalty [4, 20, 42]. Write batching allows our mechanism to proactively schedule a per-bank refresh to a bank while other banks are serving the accumulated write requests, thus enabling more parallelization of refreshes and writes to hide refresh latency. Second, although DARP can potentially delay write requests to some of the banks, this does not significantly affect system performance. The reason is that DRAM writes (which are writebacks from the last-level cache [20, 42]) are not latency-critical as processors do not stall to wait for them to finish.

The second technique, *Subarray Access Refresh Parallelization (SARP)*, takes advantage of the fact that a DRAM bank is composed of multiple *subarrays* and each subarray has its own local *sense amplifiers* [9, 18, 22, 30, 44]. We observe that only a few subarrays are refreshed within a bank when the other subarrays and the DRAM I/O bus remain completely idle. Based on this observation, *SARP* enables a bank to be accessible while being refreshed: it can serve read or write requests to idle subarrays while other subarrays in the bank are being refreshed. Therefore, SARP reduces the interference of refreshes on demand requests at the cost of very modest modifications to DRAM devices.

We make the following major **contributions**:

- We propose a new per-bank refresh scheduling policy, DARP (Dynamic Access Refresh Parallelization), to proactively schedule refreshes to banks that are idle or that are draining writes.
- We propose a new DRAM refresh mechanism, SARP (Subarray Access Refresh Parallelization), to enable a bank to serve memory requests in idle subarrays while other subarrays are being refreshed.
- We comprehensively evaluate the performance and energy benefits of DARP and SARP, and their combination, *DSARP*, compared to three state-of-the-art refresh policies across a wide variety of workloads and system configurations. One particular evaluation shows that DSARP improves system performance by 3.3%/7.2%/15.2% on average (and up to 7.1%/14.5%/27.0%) across 100 workloads over the best previous mechanism (per-bank refresh) for 8/16/32Gb DRAM devices. DSARP's performance gain increases as workload memory intensity and core count increase.

2. Background

2.1. DRAM System Organization

At a high level, a DRAM system is organized as a hierarchy of ranks and banks as shown in Figure 1. Each rank consists of multiple banks that share an internal bus for reading/writing data.[3] Because each bank acts as an independent entity, banks can serve multiple memory requests in parallel, offering *bank-level parallelism* [17, 21, 32].

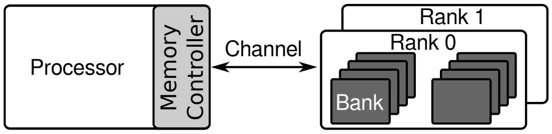


Figure 1: DRAM system organization.

A DRAM bank is further sub-divided into multiple *subarrays* [18, 37, 44] as shown in Figure 2. A subarray consists of a 2-D array of cells organized in rows and columns.[4] Each DRAM cell has two components: 1) a *capacitor* that stores one bit of data as electrical charge and 2) an *access transistor* that connects the capacitor to a wire called *bitline* that is shared by a column of cells. The access transistor is controlled by a wire called *wordline* that is shared by a row of cells. When a wordline is raised to V_{DD}, a row of cells becomes connected to the bitlines, allowing reading or writing data to the connected row of cells. The component that reads or writes a bit of data on a bitline is called a *sense amplifier*, shared by an entire column of

[2] One can think of per-bank refresh as splitting up a single large all-bank refresh operation performed on an entire rank into smaller groups of refresh operations performed on each bank.

[3] A typical DRAM system has 2 ranks connected to each channel and 8 banks per rank.

[4] Physically, DRAM has 32 to 64 subarrays, which varies depending on the number of rows (typically 16-32K) within a bank. We divide them into 8 subarray groups and refer to a subarray group as a subarray [18].

cells. A row of sense amplifiers is also called a *row buffer*. All subarrays' row buffers are connected to an I/O buffer [15, 30] that reads and writes data from/to the bank's I/O bus.

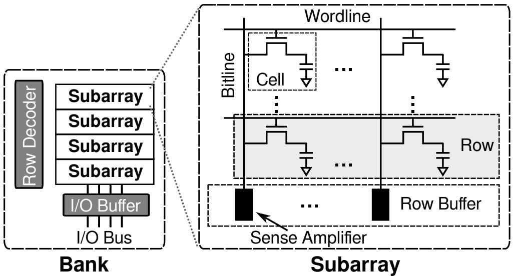


Figure 2: DRAM bank and subarray organization.

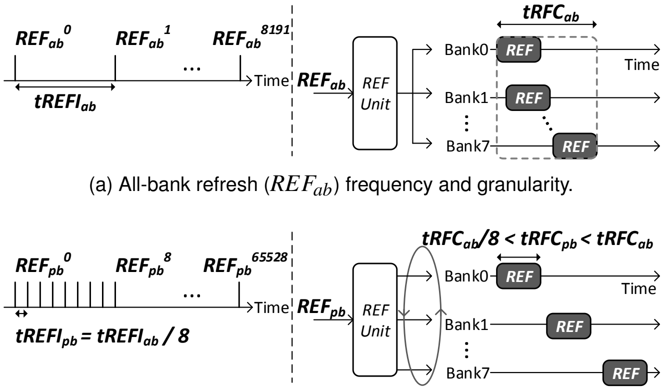
2.2. DRAM Refresh

2.2.1. All-Bank Refresh (REF_{ab}). The minimum time interval during which any cell can retain its electrical charge without being refreshed is called the *minimum retention time*, which depends on the operating temperature and DRAM type. Because there are tens of thousands of rows in DRAM, refreshing all of them in bulk incurs high latency. Instead, memory controllers send a number of refresh commands that are evenly distributed throughout the retention time to trigger refresh operations, as shown in Figure 3a. Because a typical refresh command in a commodity DDR DRAM chip operates at an entire rank level, it is also called an *all-bank refresh* or REF_{ab} for short [11, 14, 28]. The timeline shows that the time between two REF_{ab} commands is specified by $tREFI_{ab}$ (e.g., 7.8μs for 64ms retention time). Therefore, refreshing a rank requires $^{64ms}/_{7.8\mu s} \approx 8192$ refreshes and each operation refreshes exactly $^{1}/_{8192}$ of the rank's rows.

When a rank receives a refresh command, it sends the command to a DRAM-internal refresh unit that selects which specific rows or banks to refresh. A REF_{ab} command triggers the refresh unit to refresh a number of rows in every bank for a period of time called $tRFC_{ab}$ (Figure 3a). During $tRFC_{ab}$, banks are not refreshed simultaneously. Instead, refresh operations are staggered (pipelined) across banks [31]. The main reason is that refreshing every bank simultaneously would draw more current than what the power delivery network can sustain, leading to potentially incorrect DRAM operation [31, 38]. Because a REF_{ab} command triggers refreshes on all the banks within a rank, the rank cannot process any memory requests during $tRFC_{ab}$, The length of $tRFC_{ab}$ is a function of the number of rows to be refreshed.

2.2.2. Per-Bank Refresh (REF_{pb}). To allow partial access to DRAM during refresh, LPDDR DRAM (which is designed for mobile platforms), supports an additional finer-granularity refresh scheme, called *per-bank refresh* (REF_{pb} for short) [14, 28]. It splits up a REF_{ab} operation into eight separate operations scattered across eight banks (Figure 3b). Therefore, a REF_{pb} command is issued eight times more frequently than a REF_{ab} command (i.e., $tREFI_{pb} = tREFI_{ab}/\ 8$).

Similar to issuing a REF_{ab}, a controller simply sends a REF_{pb} command to DRAM every $tREFI_{pb}$ without specifying which particular bank to refresh. Instead, when a rank's internal refresh unit receives a REF_{pb} command, it refreshes only one bank for each command following a *sequential round-robin order* as shown in Figure 3b. The refresh unit uses an internal counter to keep track of which bank to refresh next.



(a) All-bank refresh (REF_{ab}) frequency and granularity.

(b) Per-bank refresh (REF_{pb}) frequency and granularity.

Figure 3: Refresh command service timelines.

By scattering refresh operations from REF_{ab} into multiple and non-overlapping per-bank refresh operations, the refresh latency of REF_{pb} ($tRFC_{pb}$) becomes shorter than $tRFC_{ab}$. Disallowing REF_{pb} operations from overlapping with each other is a design decision made by the LPDDR DRAM standard committee [14]. The reason is simplicity: to avoid the need to introduce new timing constraints, such as the timing between two overlapped refresh operations.[5]

With the support of REF_{pb}, LPDDR DRAM can serve memory requests to non-refreshing banks in parallel with a refresh operation in a single bank. Figure 4 shows pictorially how REF_{pb} provides performance benefits over REF_{ab} from parallelization of refreshes and reads. REF_{pb} reduces refresh interference on reads by issuing a refresh to Bank 0 while Bank 1 is serving reads. Subsequently, it refreshes Bank 1 to allow Bank 0 to serve a read. As a result, REF_{pb} alleviates part of the performance loss due to refreshes by enabling parallelization of refreshes and accesses across banks.

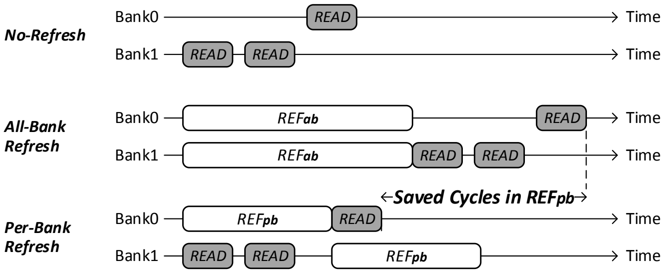


Figure 4: Service timelines of all-bank and per-bank refresh.

3. Motivation

In this section, we first describe the scaling trend of commonly used all-bank refresh in both LPDDR and DDR DRAM as chip density increases in the future. We then provide a quantitative analysis of all-bank refresh to show its performance impact on multi-core systems followed by performance comparisons to per-bank refresh that is only supported in LPDDR.

[5] At slightly increased complexity, one can potentially propose a modified standard that allows overlapped refresh of a subset of banks within a rank.

3.1. Increasing Performance Impact of Refresh

During the $tRFC_{ab}$ time period, the entire memory rank is locked up, preventing the memory controller from sending any memory request. As a result, refresh operations degrade system performance by increasing the latency of memory accesses. The negative impact on system performance is expected to be exacerbated as $tRFC_{ab}$ increases with higher DRAM density. The value of $tRFC_{ab}$ is currently 350ns for an 8Gb memory device [11]. Figure 5 shows our estimated trend of $tRFC_{ab}$ for future DRAM generations using linear extrapolation on the currently available and previous DRAM devices. The same methodology is used in prior works [24, 41]. *Projection 1* is an extrapolation based on 1, 2, and 4Gb devices; *Projection 2* is based on 4 and 8Gb devices. We use the more optimistic *Projection 2* for our evaluations. As it shows, $tRFC_{ab}$ may reach up to 1.6μs for future 64Gb DRAM devices. This long period of unavailability to process memory accesses is detrimental to system performance.

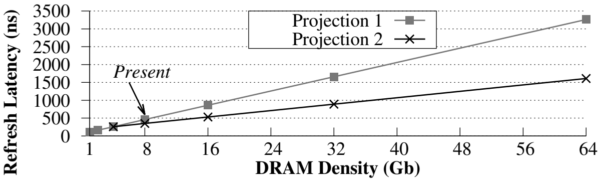


Figure 5: Refresh latency (*$tRFC_{ab}$*) trend.

To demonstrate the negative system performance impact of DRAM refresh, we evaluate 100 randomly mixed workloads categorized to five different groups based on memory intensity on an 8-core system using various DRAM densities.[6] We use up to 32Gb DRAM density that the ITRS predicts to be manufactured by 2020 [10]. Figure 6 shows the average performance loss due to all-bank refresh compared to an ideal baseline without any refreshes for each memory-intensity category. The performance degradation due to refresh becomes more severe as either DRAM chip density (i.e., $tRFC_{ab}$) or workload memory intensity increases (both of which are trends in systems), demonstrating that it is increasingly important to address the problem of DRAM refresh.

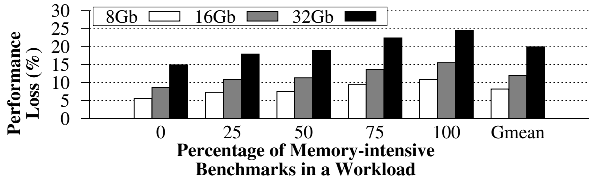


Figure 6: Performance degradation due to refresh.

Even though the current DDR3 standard does not support REF_{pb}, we believe that it is important to evaluate the performance impact of REF_{pb} on DDR3 DRAM because DDR3 DRAM chips are widely deployed in desktops and servers. Furthermore, adding per-bank refresh support to a DDR3 DRAM chip should be non-intrusive because it does not change the internal bank organization. We estimate the refresh latency of REF_{pb} in a DDR3 chip based on the values used in an LPDDR2 chip. In a 2Gb LPDDR2 chip, the per-bank refresh latency ($tRFC_{pb}$) is 90ns and the all-bank refresh latency ($tRFC_{ab}$) is 210ns, which takes *2.3x* longer than $tRFC_{pb}$ [28].[7] We apply this multiplicative factor to $tRFC_{ab}$ to calculate $tRFC_{pb}$.

Based on the estimated $tRFC_{pb}$ values, we evaluate the performance impact of REF_{pb} on the same 8-core system and workloads.[6] Figure 7 shows the average performance degradation of REF_{ab} and REF_{pb} compared to an ideal baseline without any refreshes. Even though REF_{pb} provides performance gains over REF_{ab} by allowing DRAM accesses to non-refreshing banks, its performance degradation becomes exacerbated as $tRFC_{pb}$ increases with higher DRAM density. With 32Gb DRAM chips using REF_{pb}, the performance loss due to DRAM refresh is still a significant 16.6% on average, which motivates us to address issues related to REF_{pb}.

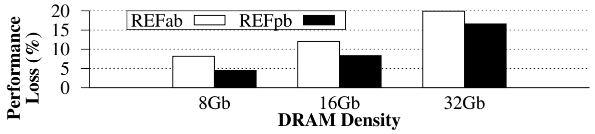


Figure 7: Performance loss due to *REF_{ab}* and *REF_{pb}*.

3.2. Our Goal

We identify two main problems that REF_{pb} faces. First, REF_{pb} commands are scheduled in a very restrictive manner in today's systems. Memory controllers have to send REF_{pb} commands in a sequential round-robin order without any flexibility. Therefore, the current implementation does not exploit the full benefit from overlapping refreshes with accesses across banks. Second, REF_{pb} cannot serve accesses to a refreshing bank until the refresh of that bank is complete. Our goal is to provide practical mechanisms to address these two problems so that we can minimize the performance overhead of DRAM refresh.

4. Mechanisms

4.1. Overview

We propose two mechanisms, *Dynamic Access Refresh Parallelization (DARP)* and *Subarray Access Refresh Parallelization (SARP)*, that hide refresh latency by parallelizing refreshes with memory accesses across *banks* and *subarrays*, respectively. DARP is a new refresh scheduling policy that consists of two components. The first component is *out-of-order per-bank refresh* that enables the memory controller to specify a particular (idle) bank to be refreshed as opposed to the standard per-bank refresh policy that refreshes banks in a strict round-robin order. With out-of-order refresh scheduling, DARP can avoid refreshing (non-idle) banks with pending memory requests, thereby avoiding the refresh latency for those requests. The second component is *write-refresh parallelization* that proactively issues per-bank refresh to a bank while DRAM is draining write batches to other banks, thereby overlapping refresh latency with write latency. The second mechanism, SARP, allows a bank to serve memory accesses in idle subarrays while other subarrays within the same bank are being refreshed. SARP exploits the

[6] Detailed methodology is described in Section 5, including workloads, simulation methodology, and performance metrics.

[7] LPDDR2 has a shorter $tRFC_{ab}$ than DDR3 because LPDDR2 1) has a retention time of 32ms instead of 64ms in DDR3 under normal operating temperature and 2) each operation refreshes fewer rows.

fact that refreshing a row is contained within a subarray, without affecting the other subarrays' components and the I/O bus used for transferring data. We now describe each mechanism in detail.

4.2. Dynamic Access Refresh Parallelization

4.2.1. Out-of-order Per-bank Refresh. The limitation of the current per-bank refresh mechanism is that it disallows a memory controller from specifying which bank to refresh. Instead, a DRAM chip has internal logic that strictly refreshes banks in a *sequential round-robin order*. Because DRAM lacks visibility into a memory controller's state (e.g., request queues' occupancy), simply using an in-order REF_{pb} policy can unnecessarily refresh a bank that has multiple pending memory requests to be served when other banks may be free to serve a refresh command. To address this problem, we propose the first component of DARP, *out-of-order per-bank refresh*. The idea is to remove the bank selection logic from DRAM and make it the memory controller's responsibility to determine which bank to refresh. As a result, the memory controller can refresh an idle bank to enhance parallelization of refreshes and accesses, avoiding refreshing a bank that has pending memory requests as much as possible.

Due to REF_{pb} reordering, the memory controller needs to guarantee that deviating from the original in-order schedule still preserves data integrity. To achieve this, we take advantage of the fact that the contemporary DDR JEDEC standard [11, 13] actually provides some refresh scheduling flexibility. The standard allows up to *eight* all-bank refresh commands to be issued late (postponed) or early (pulled-in). This implies that each bank can tolerate up to eight REF_{pb} to be postponed or pulled-in. Therefore, the memory controller ensures that reordering REF_{pb} preserves data integrity by limiting the number of postponed or pulled-in commands.

Figure 8 shows the algorithm of our mechanism. The out-of-order per-bank refresh scheduler makes a refresh decision every DRAM cycle. There are three key steps. First, when the memory controller hits a per-bank refresh schedule time (every $tREFI_{pb}$), it postpones the scheduled REF_{pb} if the to-be-refreshed bank (R) has pending demand requests (read or write) *and* it has postponed fewer refreshes than the limit of eight (❶). The hardware counter that is used to keep track of whether or not a refresh can be postponed for each bank is called the *refresh credit (ref_credit)*. The counter decrements on a postponed refresh and increments on a pulled-in refresh for each bank. Therefore, a REF_{pb} command can be postponed if the bank's ref_credit stays above -8. Otherwise the memory controller is required to send a REF_{pb} command to comply with the standard. Second, the memory controller prioritizes issuing commands for a demand request if a refresh is not sent at any given time (❷). Third, if the memory controller cannot issue any commands for demand requests due to the timing constraints, it instead randomly selects one bank (B) from a list of banks that have no pending demand requests to refresh. Such a refresh command is either a previously postponed REF_{pb} or a new pulled-in REF_{pb} (❸).

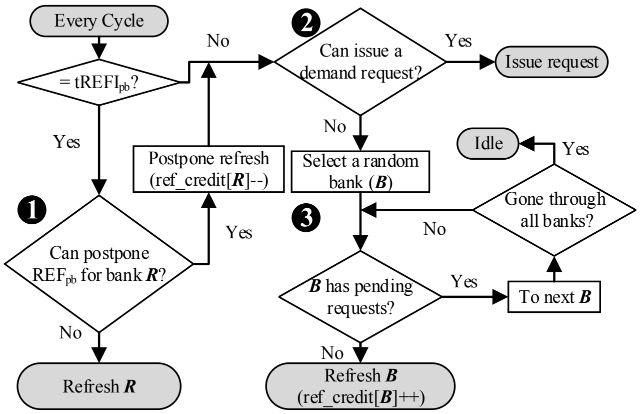
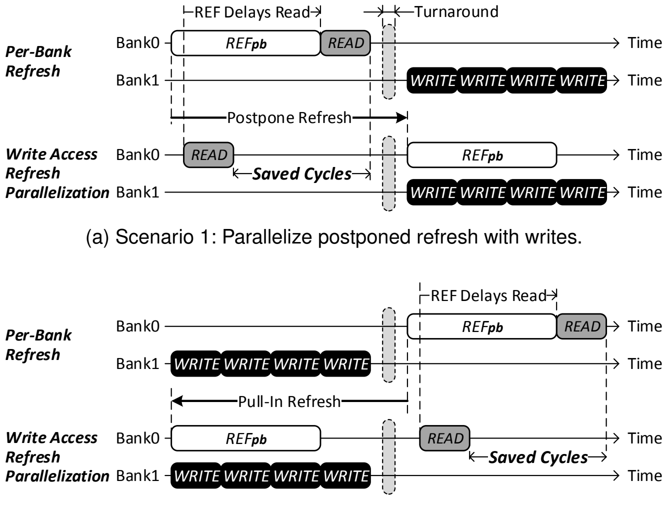


Figure 8: Algorithm of out-of-order per-bank refresh.

4.2.2. Write-refresh Parallelization. The key idea of the second component of DARP is to actively avoid refresh interference on read requests and instead enable more parallelization of refreshes with *write requests*. We make two observations that lead to our idea. First, *write batching* in DRAM creates an opportunity to overlap a refresh operation with a sequence of writes, without interfering with reads. A modern memory controller typically buffers DRAM writes and drains them to DRAM in a batch to amortize the *bus turnaround latency*, also called *tWTR* or *tRTW* [11, 18, 20], which is the additional latency incurred from switching between serving writes to reads because DRAM I/O bus is half-duplex. Typical systems start draining writes when the write buffer occupancy exceeds a certain threshold until the buffer reaches a low watermark. This draining time period is called the *writeback mode*, during which no rank within the draining channel can serve read requests [4, 20, 42]. Second, DRAM writes are not latency-critical because processors do not stall to wait for them: DRAM writes are due to dirty cache line evictions from the last-level cache [20, 42].

Given that writes are not latency-critical and are drained in a batch for some time interval, we propose the second component of DARP, *write-refresh parallelization*, that attempts to maximize parallelization of refreshes and writes. Write-refresh parallelization selects the bank with the minimum number of pending demand requests (both read and write) and preempts the bank's writes with a per-bank refresh. As a result, the bank's refresh operation is hidden by the writes in other banks.

The reasons why we select the bank with the lowest number of demand requests as a refresh candidate during writeback mode are two-fold. First, the goal of the writeback mode is to drain writes as fast as possible to reach a low watermark that determines the end of the writeback mode [4, 20, 42]. Extra time delay on writes can potentially elongate the writeback mode by increasing queueing delay and reducing the number of writes served in parallel across banks. Refreshing the bank with the lowest write request count (zero or more) has the smallest impact on the writeback mode length because other banks can continue serving their writes to reach to the low watermark. Second, if the refresh scheduled to a bank during the writeback mode happens to extend beyond writeback mode, it is likely that the refresh 1) does not delay immediate reads within the same bank because the selected bank has no reads or 2) delays



(a) Scenario 1: Parallelize postponed refresh with writes.

(b) Scenario 2: Parallelize pulled-in refresh with writes.

Figure 9: Service timeline of a per-bank refresh operation along with read and write requests using different scheduling policies.

reads in a bank that has less contention. Note that we only preempt one bank for refresh because the JEDEC standard [14] disallows overlapping per-bank refresh operations across banks within a rank.

Figure 9 shows the service timeline and benefits of write-refresh parallelization. There are **two scenarios** when the scheduling policy parallelizes refreshes with writes to increase DRAM's availability to serve read requests. Figure 9a shows the first scenario when the scheduler *postpones* issuing a REF_{pb} command to avoid delaying a read request in Bank 0 and instead serves the refresh in parallel with writes from Bank 1, effectively hiding the refresh latency in the writeback mode. Even though the refresh can potentially delay individual write requests during writeback mode, the delay does not impact performance as long as the length of writeback mode remains the same as in the baseline due to longer prioritized write request streams in other banks. In the second scenario shown in Figure 9b, the scheduler proactively *pulls in* a REF_{pb} command early in Bank 0 to fully hide the refresh latency from the later read request while Bank 1 is draining writes during the writeback mode (note that the read request cannot be scheduled during the writeback mode).

The crucial observation is that write-refresh parallelization improves performance because it avoids stalling the read requests due to refreshes by postponing or pulling in refreshes in parallel with writes without extending the writeback period.

Algorithm 1 shows the operation of write-refresh parallelization. When the memory controller enters the writeback mode, the scheduler selects a bank candidate for refresh when there is no pending refresh. A bank is selected for refresh under the following criteria: 1) the bank has the lowest number of demand requests among all banks and 2) its refresh credit has not exceeded the maximum *pulled-in* refresh threshold. After a bank is selected for refresh, its credit increments by one to allow an additional refresh postponement.

Algorithm 1 Write-refresh parallelization

```
Every $tRFC_{pb}$ in Writeback Mode:
  if refresh_queue[0:N-1].isEmpty() then
    b = find_bank_with_lowest_request_queue_count AND $ref\_credit < 8$
    refreshBank(b)
    ref_credit[b] += 1
  end if
```

4.2.3. Implementation. DARP incurs a small overhead in the memory controller and DRAM without affecting the DRAM cell array organization. There are five main modifications. First, each refresh credit is implemented with a hardware integer counter that either increments or decrements by up to eight when a refresh command is pulled-in or postponed, respectively. Thus, the storage overhead is very modest with 4 bits per bank (32 bits per rank). Second, DARP requires logic to monitor the status of various existing queues and schedule refreshes as described. Despite reordering refresh commands, all DRAM timing constraints are followed, notably *tRRD* and $tRFC_{pb}$ that limit when REF_{pb} can be issued to DRAM. Third, the DRAM command decoder needs modification to decode the bank ID that is sent on the address bus with the REF_{pb} command. Fourth, the refresh logic that is located outside of the banks and arrays needs to be modified to take in the specified bank ID. Fifth, each bank requires a separate row counter to keep track of which rows to refresh as the number of postponed or pulled-in refresh commands differs across banks. Our proposal limits the modification to the least invasive part of the DRAM without changing the structure of the dense arrays that consume the majority of the chip area.

4.3. Subarray Access Refresh Parallelization

Even though DARP allows refreshes and accesses to occur in parallel across different banks, DARP cannot deal with their collision *within a bank*. To tackle this problem, we propose *SARP (Subarray Access Refresh Parallelization)* that exploits the existence of subarrays within a bank. The key observation leading to our second mechanism is that refresh occupies only a few *subarrays* within a bank whereas the other *subarrays* and the *I/O bus* remain idle during the process of refreshing. The reasons for this are two-fold. First, refreshing a row requires only its subarray's sense amplifiers that restore the charge in the row without transferring any data through the I/O bus. Second, each subarray has its own set of *sense amplifiers* that are not shared with other subarrays.

Based on this observation, SARP's key idea is to allow memory accesses to an *idle* subarray while another subarray is refreshing. Figure 10 shows the service timeline and the performance benefit of our mechanism. As shown, SARP reduces the read latency by performing the read operation to Subarray 1 in parallel with the refresh in Subarray 0. Compared to DARP, SARP provides the following advantages: 1) SARP is applicable to both all-bank and per-bank refresh, 2) SARP enables memory accesses to a refreshing bank, which cannot be achieved with DARP, and 3) SARP also utilizes bank-level parallelism by serving memory requests from multiple banks while the entire rank is under refresh. SARP requires modifications to 1) the DRAM architecture because two distinct wordlines in different subarrays need to be raised simultaneously, which cannot be done in today's DRAM due to the shared peripheral logic among subarrays, 2) the memory controller such that it can keep track of which subarray is under refresh in order to send the appropriate memory request to an idle subarray.

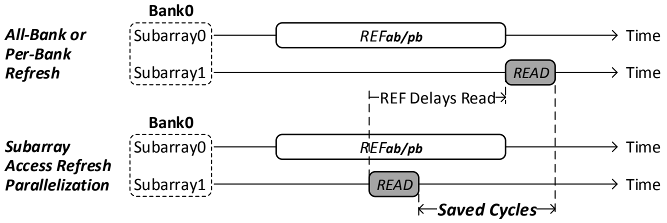


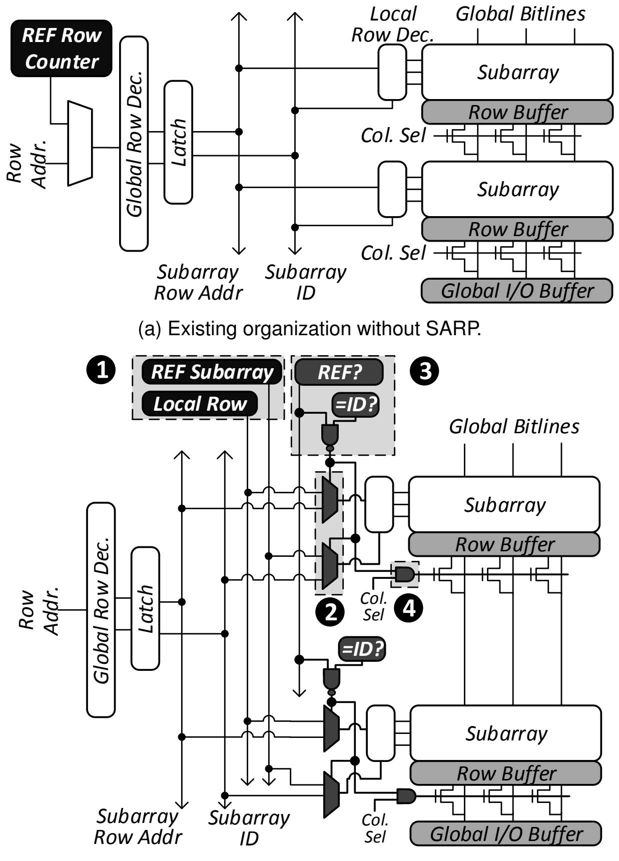
Figure 10: Service timeline of a refresh and a read request to two different subarrays within the same bank.

4.3.1. DRAM Bank Implementation for SARP. As opposed to DARP, SARP requires modifications to DRAM to support accessing subarrays individually. While subarrays are equipped with dedicated local peripheral logic, what prevents the subarrays from being operated independently is the global peripheral logic that is shared by all subarrays within a bank.

Figure 11a shows a detailed view of an existing DRAM bank's organization. There are two major shared peripheral components within a bank that prevent modern DRAM chips to refresh at subarray level. First, each bank has a *global row decoder* that decodes the incoming row's addresses. To read or write a row, memory controllers first issue an `ACTIVATE` command with the row's address. Upon receiving this command, the bank feeds the row address to the *global row decoder* that broadcasts the partially decoded address to all subarrays within the bank. After further decoding, the row's subarray then raises its wordline to begin transferring the row's cells' content to the row buffer.[8] During the transfer, the row buffer also restores the charge in the row. Similar to an `ACTIVATE`, refreshing a row requires the refresh unit to `ACTIVATE` the row to restore its electrical charge (only the refresh row counter is shown for clarity in Figure 11a). Because a bank has only one global row decoder and one pair of address wires (for subarray row address and ID), it cannot simultaneously activate two different rows (one for a memory access and the other for a refresh).

Second, when the memory controller sends a read or write command, the required column from the activated row is routed through the *global bitlines* into the *global I/O buffer* (both of which are shared across all subarrays' row buffers) and is transferred to the I/O bus (all shown in Figure 2). This is done by asserting a *column select* signal that is routed globally to *all* subarrays, which enables *all* subarrays' row buffers to be concurrently connected to the global bitlines. Since this signal connects all subarrays' row buffers to the global bitlines at the same time, if more than one activated row buffer (i.e., activated subarray) exists in the bank, an electrical short-circuit occurs, leading to incorrect operation. As a result, two subarrays cannot be kept activated when one is being read or written to, which prevents a refresh to one subarray from happening concurrently with an access in a different subarray in today's DRAM.

The key idea of SARP is to allow the concurrent activation of multiple subarrays, but to only connect the accessed subarray's row buffer to the global bitlines while another subarray is refreshing.[9] Figure 11b shows our proposed changes to the DRAM microarchitecture. There are two major enablers of SARP.



(a) Existing organization without SARP.

(b) New organization with SARP.

Figure 11: DRAM bank without and with SARP.

The first enabler of SARP allows both refresh and access commands to simultaneously select their designated rows and subarrays with three new components. The first component (❶) provides the subarray and row addresses for refreshes without relying on the global row decoder. To achieve this, it decouples the refresh row counter into a *refresh-subarray* counter and a *local-row* counter that keep track of the currently refreshing subarray and the row address within that subarray, respectively. The second component (❷) allows each subarray to activate a row for either a refresh or an access through two muxes. One mux is a row-address selector and the other one is a subarray selector. The third component (❸) serves as a control unit that chooses a subarray for refresh. The `REF?` block indicates if the bank is currently under refresh and the `=ID?` comparator determines if the corresponding subarray's ID matches with the refreshing subarray counter for refresh. These three components form a new address path for the refresh unit to supply refresh addresses in parallel with addresses for memory accesses.

[8] The detailed step-to-step explanation of the activation process can be found in prior works [18, 22, 37].

[9] As described in Section 2.1, we refer to a subarray as a collection of multiple physical subarrays. A modern refresh operation concurrently refreshes every physical subarray within a collection.

The second enabler of SARP allows accesses to one activated subarray while another subarray is kept activated for refreshes. We add an *AND* gate (❹) to each subarray that ensures the refreshing subarray's row buffer is *not* connected to the global bitlines when the *column select* signal is asserted on an access. At any instance, there is at most one activated subarray among all non-refreshing subarrays because the global row decoder activates only one subarray at a time. With the two proposed enablers, SARP allows one activated subarray for refreshes in parallel with another activated subarray that serves data to the global bitlines.

4.3.2. Detecting Subarray Conflicts in the Memory Controller. To avoid accessing a refreshing subarray, which is determined internally by the DRAM chip in our current mechanism, the memory controller needs to know the current refreshing subarray and the number of subarrays. We create shadow copies of the *refresh-subarray* and *local-row* counters in the memory controller to keep track of the currently-refreshing subarray. We store the number of subarrays in an EEPROM called the *serial presence detect (SPD)* [12], which stores various timing and DRAM organization information in existing DRAM modules. The memory controller reads this information at system boot time so that it can issue commands correctly.[10]

4.3.3. Power Integrity. Because an `ACTIVATE` draws a lot of current, DRAM standards define two timing parameters to constrain the activity rate of DRAM so that `ACTIVATE`s do not over-stress the power delivery network [11, 38]. The first parameter is the *row-to-row activation delay* (*tRRD*) that specifies the minimum waiting time between two subsequent `ACTIVATE` commands within a DRAM device. The second is called the *four activate window* (*tFAW*) that defines the length of a rolling window during which a maximum of four `ACTIVATE`s can be in progress. Because a refresh operation requires activating rows to restore charge in DRAM cells, SARP consumes additional power by allowing accesses during refresh. To limit the power consumption due to `ACTIVATE`s, we further constrain the activity rate by increasing both *tFAW* and *tRRD*, as shown below. This results in fewer `ACTIVATE` commands issued during refresh.

$$PowerOverhead_{FAW} = \frac{4 * I_{ACT} + I_{REF}}{4 * I_{ACT}} \tag{1}$$

$$t_{FAW\_SARP} = t_{FAW} * PowerOverhead_{FAW} \tag{2}$$

$$t_{RRD\_SARP} = t_{RRD} * PowerOverhead_{FAW} \tag{3}$$

I_{ACT} and I_{REF} represent the current values of an `ACTIVATE` and a refresh, respectively, based on the Micron Power Calculator [27]. We calculate the power overhead of parallelizing a refresh over a *four activate window* using (1). Then we apply this power overhead to both *tFAW* (2) and *tRRD* (3), which are enforced during refresh operations. Based on the *IDD* values in the Micron 8Gb DRAM [29] data sheet, SARP increases *tFAW* and *tRRD* by 2.1x during all-bank refresh operations. Each per-bank refresh consumes 8x lower current than an all-bank refresh, thus increasing *tFAW* and *tRRD* by only 13.8%.

4.3.4. Die Area Overhead. In our evaluations, we use 8 subarrays per bank and 8 banks per DRAM chip. Based on this configuration, we calculate the area overhead of SARP using parameters from a Rambus DRAM model at 55*nm* technology [34], the best publicly available model that we know of, and find it to be 0.71% in a 2Gb DDR3 DRAM chip with a die area of 73.5mm^2. The power overhead of the additional components is negligible compared to the entire DRAM chip.

5. Methodology

To evaluate our mechanisms, we use an in-house cycle-level x86 multi-core simulator with a front end driven by Pin [25] and an in-house cycle-accurate DRAM timing model validated against DRAMSim2 [36]. Unless stated otherwise, our system configuration is as shown in Table 1.

Processor	8 cores, 4GHz, 3-wide issue, 8 MSHRs/core, 128-entry instruction window
Last-level Cache	64B cache-line, 16-way associative, 512KB private cache-slice per core
Memory Controller	64/64-entry read/write request queue, FR-FCFS [35], writes are scheduled in batches [4, 20, 42] with low watermark = 32, closed-row policy [4, 17, 35]
DRAM	DDR3-1333 [29], 2 channels, 2 ranks per channel, 8 banks/rank, 8 subarrays/bank, 64K rows/bank, 8KB rows
Refresh Settings	$tRFC_{ab}$ = 350/530/890ns for 8/16/32Gb DRAM chips, $tREFI_{ab}$ = 3.9µs, $tRFC_{ab}$-to-$tRFC_{pb}$ ratio = 2.3

Table 1: Evaluated system configuration.

In addition to 8Gb DRAM, we also evaluate systems using 16Gb and 32Gb near-future DRAM chips [10]. Because commodity DDR DRAM does not have support for REF_{pb}, we estimate the $tRFC_{pb}$ values for DDR3 based on the ratio of $tRFC_{ab}$ to $tRFC_{pb}$ in LPDDR2 [28] as described in Section 3.1. We evaluate our systems with 32ms retention time, which is a typical setting for a server environment and LPDDR DRAM, as also evaluated in previous work [33, 41].

We use benchmarks from *SPEC CPU2006 [40], STREAM [2], TPC [3]*, and a microbenchmark with random-access behavior similar to HPCC RandomAccess [1]. We classify each benchmark as either memory intensive (MPKI $\geq$ 10) or memory non-intensive (MPKI $<$ 10). We then form five intensity categories based on the fraction of memory intensive benchmarks within a workload: 0%, 25%, 50%, 75%, and 100%. Each category contains 20 randomly mixed workloads, totaling to 100 workloads for our main evaluations. For sensitivity studies in Sections 6.1.5, 6.2, 6.3, and 6.4, we run 16 randomly selected memory-intensive workloads using 32Gb DRAM to observe the performance trend.

We measure system performance with the commonly-used *weighted speedup (WS)* [6, 39] metric. To report the DRAM system power, we use the methodology from the *Micron power calculator* [27]. The DRAM device parameters are obtained from [29]. Every workload runs for 256 million cycles to ensure the same number of refreshes. We report DRAM system power as *energy per memory access serviced* to fairly compare across different workloads.

[10] Note that it is possible to extend our mechanisms such that the memory controller specifies the subarray to be refreshed instead of the DRAM chip. This requires changes to the DRAM interface.

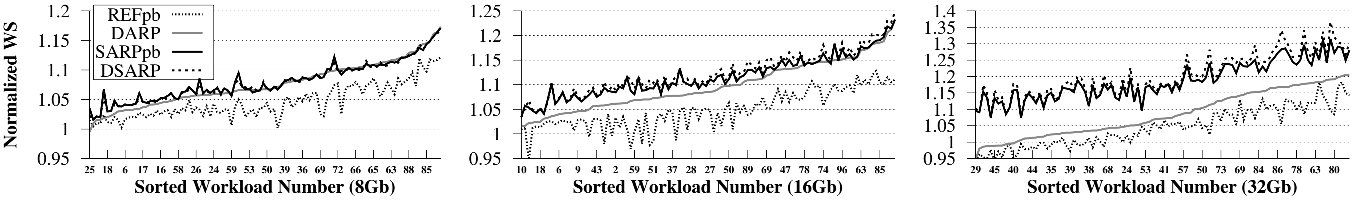


Figure 12: Multi-core system performance improvement over REF_{ab} across 100 workloads.

6. Evaluation

In this section, we evaluate the performance of the following mechanisms: 1) the *all-bank* refresh scheme (REF_{ab}), 2) the *per-bank* refresh scheme (REF_{pb}), 3) elastic refresh [41], 4) our first mechanism, DARP, 5) our second mechanism, SARP, that is applied to either REF_{ab} (SARP_{ab}) or REF_{pb} (SARP_{pb}), 6) the combination of DARP and SARP_{pb}, called DSARP, and 7) an ideal scheme that eliminates refresh. Elastic refresh [41] takes advantage of the refresh scheduling flexibility in the DDR standard: it postpones a refresh if the refresh is predicted to interfere with a demand request, based on a prediction of how long a rank will be idle, i.e., without any demand request (see Section 7 for more detail).

6.1. Multi-Core Results

Figure 12 plots the system performance improvement of REF_{pb}, DARP, SARP_{pb}, and DSARP over the all-bank refresh baseline (REF_{ab}) using various densities across 100 workloads (sorted based on the performance improvement due to DARP). The x-axis shows the sorted workload numbers as categorized into five memory-intensive groups with 0 to 19 starting in the least memory-intensive group and 80 to 99 in the most memory-intensive one. Table 2 shows the maximum and geometric mean of system performance improvement due to our mechanisms over REF_{pb} and REF_{ab} for different DRAM densities. We draw five key conclusions from these results.

First, DARP provides system performance gains over both REF_{pb} and REF_{ab} schemes: 2.8%/4.9%/3.8% and 7.4%/9.8%/8.3% on average in 8/16/32Gb DRAMs, respectively. The reason is that DARP hides refresh latency with writes and issues refresh commands in out-of-order fashion to reduce refresh interference on reads. Second, SARP_{pb} provides significant system performance improvement over DARP and refresh baselines for all the evaluated DRAM densities as SARP_{pb} enables accesses to idle subarrays in the refreshing banks. SARP_{pb}'s average system performance improvement over REF_{pb} and REF_{ab} is 3.3%/6.7%/13.7% and 7.9%/11.7%/18.6% in 8/16/32Gb DRAMs, respectively. Third, as density increases, the performance benefit of SARP_{pb} over DARP gets larger. This is because the longer refresh latency becomes more difficult to hide behind writes or idle banks for DARP. This is also the reason why the performance improvement due to DARP drops slightly at 32Gb compared to 16Gb. On the other hand, SARP_{pb} is able to allow a long-refreshing bank to serve some memory requests in its subarrays.

Fourth, combining both SARP_{pb} and DARP (DSARP) provides additive system performance improvement by allowing even more parallelization of refreshes and memory accesses. As DRAM density (refresh latency) increases, the benefit becomes more apparent, resulting in improvement up to 27.0% and 36.6% over REF_{pb} and REF_{ab} in 32Gb DRAM, respectively.

Fifth, REF_{pb} performs worse than REF_{ab} for some workloads (the curves of REF_{pb} dropping below one) and the problem is exacerbated with longer refresh latency. Because REF_{pb} commands cannot overlap with each other [14], their latencies are serialized. In contrast, REF_{ab} operates on every bank in parallel, which is triggered by a single command that partially overlaps refreshes across different banks [31]. Therefore, in a pathological case, the REF_{pb} latency for refreshing every bank (eight in most DRAMs) in a rank is $8 \times tRFC_{pb} = 8 \times \frac{tRFC_{ab}}{2.3} \approx 3.5 \times tRFC_{ab}$, whereas all-bank refresh takes $tRFC_{ab}$ (see Section 3.1). If a workload cannot effectively utilize multiple banks during a per-bank refresh operation, REF_{pb} may potentially degrade system performance compared to REF_{ab}.

Density	Mechanism	Max (%)		Gmean (%)	
		REF_{pb}	REF_{ab}	REF_{pb}	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}.

6.1.1. All Mechanisms' Results. Figure 13 shows the average performance improvement due to all the evaluated refresh mechanisms over REF_{ab}. The weighted speedup value for REF_{ab} is 5.5/5.3/4.8 using 8/16/32Gb DRAM density. We draw three major conclusions. First, using SARP on all-bank refresh (SARP_{ab}) also significantly improves system performance. This is because SARP allows a rank to continue serving memory requests while it is refreshing. Second, elastic refresh does not substantially improve performance, with an average of 1.8% over all-bank refresh. This is because elastic refresh does not attempt to pull in refresh opportunistically, nor does it try to overlap refresh latency with other memory accesses. The observation is consistent with prior work [33]. Third, DSARP captures most of the benefit of the ideal baseline ("No REF"), performing within 0.9%, 1.2%, and 3.7% of the ideal for 8, 16, and 32Gb DRAM, respectively.

6.1.2. Performance Breakdown of DARP. To understand the observed performance gain in more detail, we evaluate the performance of DARP's two components separately. *Out-of-order per-bank refresh* improves performance by 3.2%/3.9%/3.0%

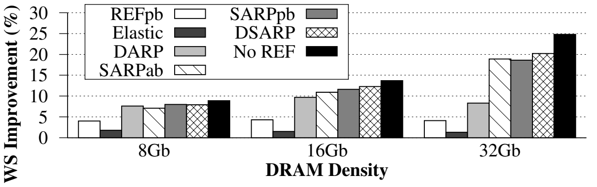


Figure 13: Average system performance improvement over REF_{ab}.

on average and up to 16.8%/21.3%/20.2% compared to REF_{ab} in 8/16/32Gb DRAMs. Adding *write-refresh parallelization* to *out-of-order per-bank refresh* (DARP) provides additional performance gains of 4.3%/5.8%/5.2% on average by hiding refresh latency with write accesses.

6.1.3. Energy. Our techniques reduce energy per memory access compared to existing policies, as shown in Figure 14. The main reason is that the performance improvement reduces average static energy for each memory access. Note that these results conservatively assume the same power parameters for 8, 16, and 32 Gb chips, so the savings in energy would likely be more significant if realistic power parameters are used for the more power-hungry 16 and 32 Gb nodes.

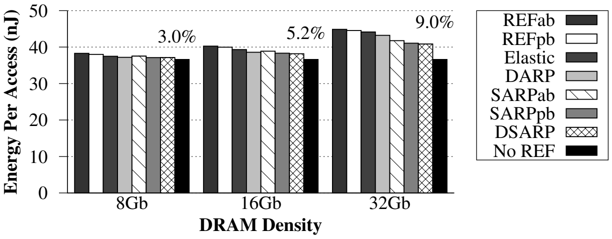


Figure 14: Energy consumption. Value on top indicates percentage reduction of DSARP compared to REF_{ab}.

6.1.4. Effect of Memory Intensity. Figure 15 shows the performance improvement of DSARP compared to REF_{ab} and REF_{pb} on workloads categorized by memory intensity (% of memory-intensive benchmarks in a workload), respectively. We observe that DSARP outperforms REF_{ab} and REF_{pb} consistently. Although the performance improvement of DSARP over REF_{ab} increases with higher memory intensity, the gain over REF_{pb} begins to plateau when the memory intensity grows beyond 25%. This is because REF_{pb}'s benefit over REF_{ab} also increases with memory intensity as REF_{pb} enables more accesses to be be parallelized with refreshes. Nonetheless, our mechanism provides the highest system performance compared to prior refresh policies.

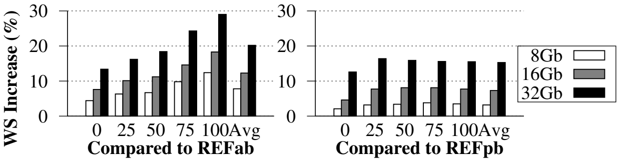


Figure 15: WS improvement of DSARP over REF_{ab} and REF_{pb} as memory intensity and DRAM density vary.

6.1.5. Effect of Core Count. Table 3 shows the weighted speedup, harmonic speedup, fairness, and energy-per-access improvement due to DSARP compared to REF_{ab} for systems with 2, 4, and 8 cores. For all three systems, DSARP consistently outperforms the baseline without unfairly penalizing any specific application. We conclude that DSARP is an effective mechanism to improve performance, fairness and energy of multi-core systems employing high-density DRAM.

Number of Cores	2	4	8
Weighted Speedup Improvement (%)	16.0	20.0	27.2
Harmonic Speedup Improvement [26] (%)	16.1	20.7	27.9
Maximum Slowdown Reduction [5, 16, 17] (%)	14.9	19.4	24.1
Energy-Per-Access Reduction (%)	10.2	8.1	8.5

Table 3: Effect of DSARP on multi-core system metrics.

6.2. Effect of *tFAW*

Table 4 shows the performance improvement of SARP_{pb} over REF_{pb} when we vary *tFAW* in DRAM cycles (20 cycles for the baseline as specified by the data sheet) and when *tRRD* scales proportionally with *tFAW*.[11] As *tFAW* reduces, the performance benefit of SARP_{pb} increases over REF_{pb}. This is because reduced *tFAW* enables more accesses/refreshes to happen in parallel, which our mechanism takes advantage of.

tFAW*/*tRRD (DRAM cycles)	5/1	10/2	15/3	**20/4**	25/5	30/6
WS Improvement (%)	14.0	13.9	13.5	**12.4**	11.9	10.3

Table 4: Performance improvement due to SARP_{pb} over REF_{pb} with various *tFAW* and *tRRD* values.

6.3. Effect of Subarrays-Per-Bank

Table 5 shows that the average performance gain of SARP_{pb} over REF_{pb} increases as the number of subarrays increases in 32Gb DRAM. This is because with more subarrays, the probability of memory requests to a refreshing subarray reduces.

Subarrays-per-bank	1	2	4	**8**	16	32	64
WS Improvement (%)	0	3.8	8.5	**12.4**	14.9	16.2	16.9

Table 5: Effect of number of subarrays per bank.

6.4. Effect of Refresh Interval

For our studies so far, we use 32ms retention time (i.e., $tREFI_{ab} = 3.9\mu$s) that represents a typical setting for a server environment and LPDDR DRAM [14]. Table 6 shows the performance improvement of DSARP over two baseline refresh schemes using retention time of *64ms* (i.e., $tREFI_{pb} = 7.8\mu$s). DSARP consistently provides performance gains over both refresh schemes. The maximum performance improvement over REF_{pb} is higher than that over REF_{ab} at 32Gb because REF_{pb} actually degrades performance compared to REF_{ab} for some workloads, as discussed in the 32ms results (Section 6.1).

Density	**Max (%)**		**Gmean (%)**	
	REF_{pb}	REF_{ab}	REF_{pb}	REF_{ab}
8Gb	2.5	5.8	1.0	3.3
16Gb	4.6	8.6	2.6	5.3
32Gb	18.2	13.6	8.0	9.1

Table 6: Maximum and average WS improvement due to DSARP.

[11] We evaluate only SARP_{pb} because it is sensitive to *tFAW* and *tRRD* as it extends these parameters during parallelization of refreshes and accesses to compensate for the power overhead.

6.5. DDR4 Fine Granularity Refresh

DDR4 DRAM supports a new refresh mode called *fine granularity refresh (FGR)* in an attempt to mitigate the increasing refresh latency ($tRFC_{ab}$) [13]. *FGR* trades off shorter $tRFC_{ab}$ with faster refresh rate ($1/tREFI_{ab}$) that increases by either 2x or 4x. Figure 16 shows the effect of *FGR* in comparison to REF_{ab}, *adaptive refresh policy (AR)* [31], and DSARP. 2x and 4x *FGR* actually reduce average system performance by 3.9%/4.0%/4.3% and 8.1%/13.7%/15.1% compared to REF_{ab} with 8/16/32Gb densities, respectively. As the refresh rate increases by 2x/4x (higher refresh penalty), $tRFC_{ab}$ does not scale down with the same constant factors. Instead, $tRFC_{ab}$ reduces by 1.35x/1.63x with 2x/4x higher rate [13], thus increasing the worst-case refresh latency by 1.48x/2.45x. This performance degradation due to *FGR* has also been observed in Mukundan et al. [31]. AR [31] dynamically switches between 1x (i.e., REF_{ab}) and 4x refresh modes to mitigate the downsides of *FGR*. AR performs slightly worse than REF_{ab} (within 1%) for all densities. Because using 4x *FGR* greatly degrades performance, AR can only mitigate the large loss from the 4x mode and cannot improve performance over REF_{ab}. On the other hand, DSARP is a more effective mechanism to tolerate the long refresh latency than both *FGR* and AR as it overlaps refresh latency with access latency without increasing the refresh rate.

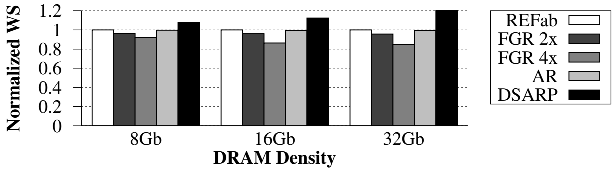


Figure 16: Performance comparisons to FGR and AR [31].

7. Related Work

To our knowledge, this is the first work to comprehensively study the effect of per-bank refresh and propose 1) a refresh scheduling policy built on top of per-bank refresh and 2) a mechanism that achieves parallelization of refresh and memory accesses *within* a refreshing bank. We discuss prior works that mitigate the negative effects of DRAM refresh and compare them to our mechanisms.

Refresh Scheduling. Stuecheli et al. [41] propose elastic refresh, which we discussed and evaluated in Section 6. Elastic refresh postpones refreshes by a time delay that varies based on the number of postponed refreshes and the predicted rank idle time to avoid interfering with demand requests. Elastic refresh has two shortcomings. First, it becomes less effective when the average rank idle period is shorter than $tRFC_{ab}$ as the refresh latency cannot be fully hidden in that period. This occurs especially with 1) more memory-intensive workloads that inherently have less idleness and 2) higher density DRAM chips that have higher $tRFC_{ab}$. Second, elastic refresh incurs more refresh latency when it *incorrectly* predicts a time period as idle when it actually has pending requests. In contrast, our mechanisms parallelize refresh operations with accesses even if there is no idle period and therefore outperform elastic refresh.

Ishii et al. [8] propose a write scheduling policy that prioritizes write draining over read requests in a rank while another rank is refreshing (even if the write queue has not reached the threshold to trigger write mode). This technique is *only* applicable in multi-ranked memory systems. Our mechanisms are *also* applicable to single-ranked memory systems by enabling parallelization of refreshes and accesses at the bank and subarray levels, and they can be combined with Ishii et al. [8].

Mukundan et al. [31] propose scheduling techniques (in addition to adaptive refresh discussed in Section 6.5) to address the problem of *command queue seizure*, whereby a command queue gets filled up with commands to a refreshing rank, blocking commands to *another* non-refreshing rank. In our work, we use a different memory controller design that does not have command queues, similarly to prior work [7]. Our controller generates a command for a scheduled request *right before* the request is sent to DRAM instead of pre-generating the commands and queueing them up. Thus, our baseline design does not suffer from the problem of command queue seizure.

Subarray-Level Parallelism (SALP). Kim et al. [18] propose SALP to reduce bank serialization latency by enabling *multiple accesses* to different subarrays within a bank to proceed in a pipelined manner. In contrast to SALP, our mechanism (SARP) parallelizes *refreshes and accesses* to different subarrays within the same bank. Therefore, SARP exploits the existence of subarrays for a different purpose and in a different way from SALP. As Section 4.3.1 describes, we reduce the sharing of the peripheral circuits for refreshes and accesses, not for arbitrary accesses. As such, our implementation is not only different, but also less intrusive than SALP: SARP does not require new DRAM commands and timing constraints.

Refresh Pausing. Nair et al. [33] propose pausing a refresh operation to serve pending memory requests. To make pausing a refresh possible, the authors assume that DRAM refreshes multiple rows *sequentially*. Thus, there is a short recovery time, called a "refresh pausing point" (RPP), after refreshing each row so that the memory controller can signal the DRAM to stop refreshing subsequent rows. However, DRAM manufacturers currently design their chips to refresh multiple rows in *parallel* (or, in a *staggered/pipelined* way, as explained in [31]).

eDRAM Concurrent Refresh. Kirihata et al. [19] propose a mechanism to enable a bank to refresh independently while another bank is being accessed in embedded DRAM (eDRAM). Our work differs from [19] in two major ways. First, unlike SARP, [19] parallelizes refreshes only across banks, not *within* each bank. Second, there are significant differences between DRAM and eDRAM architectures, which make it non-trivial to apply [19]'s mechanism directly to DRAM. In particular, eDRAMs have no standardized timing/power integrity constraints and access protocol, making it simpler for each bank to independently manage its refresh schedule. In contrast, refreshes in DRAM need to be managed by the memory controller to ensure that parallelizing refreshes with accesses does not violate other constraints.

Retention-Aware Refresh. Prior works (e.g., [24, 43]) propose mechanisms to reduce unnecessary refresh operations by taking advantage of the fact that different DRAM cells have widely different retention times [23]. These works assume that the retention time of DRAM cells can be *accurately* profiled and they depend on having this accurate profile in order to guarantee data integrity [23]. However, as shown in [23], accurately determining the retention time profile of DRAM is an unsolved research problem due to the Variable Retention Time and Data Pattern Dependence phenomena, which can cause the retention time of a cell to fluctuate over time. As such, retention-aware refresh techniques need to overcome the profiling challenges to be viable.

8. Conclusion

We introduced two new complementary techniques, DARP (Dynamic Access Refresh Parallelization) and SARP (Subarray Access Refresh Parallelization), to mitigate the DRAM refresh penalty by enhancing *refresh-access parallelization* at the bank and subarray levels, respectively. DARP 1) issues per-bank refreshes to idle banks in an out-of-order manner instead of issuing refreshes in a strict round-robin order, 2) proactively schedules per-bank refreshes during intervals when a batch of writes are draining to DRAM. SARP enables a bank to serve requests from idle subarrays in parallel with other subarrays that are being refreshed. Our extensive evaluations on a wide variety of systems and workloads show that these mechanisms significantly improve system performance and outperform state-of-the-art refresh policies, approaching the performance of ideally eliminating all refreshes. We conclude that DARP and SARP are effective in hiding the refresh latency penalty in modern and near-future DRAM systems, and that their benefits increase as DRAM density increases.

Acknowledgments

We thank the anonymous reviewers and Jamie Liu for helpful feedback and the members of the SAFARI research group for feedback and the stimulating environment they provide. We acknowledge the support of IBM, Intel, and Samsung. This research was supported in part by the Intel Science and Technology Center on Cloud Computing, SRC (Semiconductor Research Corporation), and an NSF CAREER Award 0953246.

References

[1] "HPCC. RandomAccess," http://icl.cs.utk.edu/hpcc.

[2] "STREAM Benchmark," http://www.streambench.org/.

[3] "TPC," http://www.tpc.org/.

[4] N. Chatterjee *et al.*, "Staged reads: Mitigating the impact of DRAM writes on DRAM reads," in *HPCA*, 2012.

[5] R. Das *et al.*, "Application-aware prioritization mechanisms for on-chip networks," in *MICRO*, 2009.

[6] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE Micro*, pp. 42–53, 2008.

[7] E. Herrero *et al.*, "Thread row buffers: Improving memory performance isolation and throughput in multiprogrammed environments," *IEEE TC*, pp. 1879–1892, 2013.

[8] Y. Ishii *et al.*, "High performance memory access scheduling using compute-phase prediction and writeback-refresh overlap," in *JILP Memory Scheduling Championship*, 2012.

[9] K. Itoh, *VLSI Memory Chip Design*. Springer, 2001.

[10] ITRS, "International technology roadmap for semiconductors executive summary," http://www.itrs.net/Links/2011ITRS/2011Chapters/2011ExecSum.pdf, p. 83, 2011.

[11] JEDEC, "DDR3 SDRAM Standard," 2010.

[12] JEDEC, "Standard No. 21-C. Annex K: Serial Presence Detect (SPD) for DDR3 SDRAM Modules," 2011.

[13] JEDEC, "DDR4 SDRAM Standard," 2012.

[14] JEDEC, "Low Power Double Data Rate 3 (LPDDR3)," 2012.

[15] R. Kho *et al.*, "75nm 7Gb/s/pin 1Gb GDDR5 graphics memory device with bandwidth-improvement techniques," in *ISSCC*, 2011.

[16] Y. Kim *et al.*, "ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers," in *HPCA*, 2010.

[17] Y. Kim *et al.*, "Thread cluster memory scheduling: Exploiting differences in memory access behavior," in *MICRO*, 2010.

[18] Y. Kim *et al.*, "A case for exploiting subarray-level parallelism (SALP) in DRAM," in *ISCA*, 2012.

[19] T. Kirihata *et al.*, "An 800-MHz embedded DRAM with a concurrent refresh mode," *IEEE JSSC*, pp. 1377–1387, 2005.

[20] C. J. Lee *et al.*, "DRAM-Aware last-level cache writeback: Reducing write-caused interference in memory systems," in *HPS Technical Report*, 2010.

[21] C. J. Lee *et al.*, "Improving memory bank-level parallelism in the presence of prefetching," in *MICRO*, 2009.

[22] D. Lee *et al.*, "Tiered-latency DRAM: A low latency and low cost DRAM architecture," in *HPCA*, 2013.

[23] J. Liu *et al.*, "An experimental study of data retention behavior in modern DRAM devices: Implications for retention time profiling mechanisms," in *ISCA*, 2013.

[24] J. Liu *et al.*, "RAIDR: Retention-aware intelligent DRAM refresh," in *ISCA*, 2012.

[25] C.-K. Luk *et al.*, "Pin: Building customized program analysis tools with dynamic instrumentation," in *PLDI*, 2005.

[26] K. Luo *et al.*, "Balancing throughput and fairness in SMT processors," in *ISPASS*, 2001.

[27] Micron Technology, "Calculating Memory System Power for DDR3," 2007.

[28] Micron Technology, "2Gb: x16, x32 Mobile LPDDR2 SDRAM S4," 2010.

[29] Micron Technology, "8Gb: x4, x8 1.5V TwinDie DDR3 SDRAM," 2011.

[30] Y. Moon *et al.*, "1.2V 1.6Gb/s 56nm 6F 2 4Gb DDR3 SDRAM with hybrid-I/O sense amplifier and segmented sub-array architecture," in *ISSCC*, 2009.

[31] J. Mukundan *et al.*, "Understanding and mitigating refresh overheads in high-density DDR4 DRAM systems," in *ISCA*, 2013.

[32] O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems," in *ISCA*, 2008.

[33] P. Nair *et al.*, "A case for refresh pausing in DRAM memory systems," in *HPCA*, 2013.

[34] Rambus, "DRAM Power Model," 2010.

[35] S. Rixner *et al.*, "Memory access scheduling," in *ISCA*, 2000.

[36] P. Rosenfeld *et al.*, "DRAMSim2: A cycle accurate memory system simulator," *IEEE CAL*, 2011.

[37] V. Seshadri *et al.*, "Rowclone: Fast and energy-efficient in-DRAM bulk data copy and initialization," in *MICRO*, 2013.

[38] M. Shevgoor *et al.*, "Quantifying the relationship between the power delivery network and architectural policies in a 3D-stacked memory device," in *MICRO*, 2013.

[39] A. Snavely and D. Tullsen, "Symbiotic job scheduling for a simultaneous multithreading processor," in *ASPLOS*, 2000.

[40] SPEC CPU2006, "Standard Performance Evaluation Corporation," http://www.spec.org/cpu2006.

[41] J. Stuecheli *et al.*, "Elastic refresh: Techniques to mitigate refresh penalties in high density memory," in *MICRO*, 2010.

[42] J. Stuecheli *et al.*, "The virtual write queue: Coordinating DRAM and last-level cache policies," in *ISCA*, 2010.

[43] R. Venkatesan *et al.*, "Retention-aware placement in DRAM (RAPID): Software methods for quasi-non-volatile DRAM," in *HPCA*, 2006.

[44] T. Vogelsang, "Understanding the energy consumption of dynamic random access memories," in *MICRO*, 2010.