In the proceedings of the International Symposium on High Performance Computer Architecture (HPCA-2013)

A Case for Refresh Pausing in DRAM Memory Systems

Prashant Nair Chia-Chen Chou Moinuddin K. Qureshi

School of Electrical and Computer Engineering
Georgia Institute of Technology
{pnair6, cchou34, moin}@gatech.edu

Abstract

DRAM cells rely on periodic refresh operations to maintain data integrity. As the capacity of DRAM memories has increased, so has the amount of time consumed in doing refresh. Refresh operations contend with read operations, which increases read latency and reduces system performance. We show that eliminating latency penalty due to refresh can improve average performance by 7.2%. However, simply doing intelligent scheduling of refresh operations is ineffective at obtaining significant performance improvement.

This paper provides an alternative and scalable option to reduce the latency penalty due to refresh. It exploits the property that each refresh operation in a typical DRAM device internally refreshes multiple DRAM rows in JEDEC-based distributed refresh mode. Therefore, a refresh operation has well defined points at which it can potentially be Paused *to service a pending read request. Leveraging this property, we propose* Refresh Pausing*, a solution that is highly effective at alleviating the contention from refresh operations. It provides an average performance improvement of 5.1% for 8Gb devices, and becomes even more effective for future high-density technologies. We also show that Refresh Pausing significantly outperforms the recently proposed Elastic Refresh scheme.*

1. Introduction

Dynamic Random Access Memory (DRAM) has been the technology of choice for building main memory systems for the past four decades. Technology scaling of DRAM has allowed higher density devices, enabling higher capacity memory systems. As systems integrate more and more cores on a chip, the demand for memory capacity will only increase, further motivating the need to increase DRAM densities.

The fundamental unit of storage in a DRAM system is a DRAM cell consisting of one transistor and one capacitor. Data is represented in the DRAM cell, as the amount of electrical charge stored in the capacitor. If a DRAM cell stays idle without any operation for a certain amount of time, the leakage current drains out the stored charge, which can lead to data loss. To maintain data integrity, DRAM devices periodically perform *Refresh* operations.

JEDEC standards [1] specify that DRAM devices must be refreshed every 64 millisecond (32 millisecond at above 85°C temperature). All the DRAM rows must undergo refresh within this time period. The total time incurred in doing refresh is thus proportional to the number of rows in memory, and approximately doubles as the number of rows in the DRAM array is doubled. Initial DRAM designs performed *Burst Refreshes* whereby refresh for all DRAM rows happened in succession; however, this mode makes memory unavailable for a long periods of time. To avoid this long latency, JEDEC standards support *Distributed Refresh* mode. In this mode, the total number of rows in a bank is divided into 8K *groups*, and each group is refreshed within a time period equal to 7.8 µsecond (3.9 µsecond at high temperatures). This time duration is referred to as *Refresh Interval* or T_{REFI} . The DRAM controller sends a refresh pulse to DRAM devices once every T_{REFI} . The standard for T_{REFI} was developed when memory banks typically had 8K rows; therefore each refresh pulse refreshed exactly one row. Over time, as the size of memory has increased, the T_{REFI} has remained the same, only the number of rows refreshed per refresh pulse has increased. For example, for the 8Gb DRAM chips we consider, each refresh pulse refreshes 8-16 rows. Therefore, the latency to do refresh for one group is almost an order of magnitude longer than a typical read operation.

When a given memory bank is performing refresh operations, the bank becomes unavailable for servicing demand requests such as reads and writes. Thus, a read request arriving at a bank that is undergoing refresh waits until the refresh operation gets completed. This increases the effective read latency and degrades system performance. As memory technology scales to higher densities, the latency from refresh worsens from being significant to severe. In fact, as JEDEC updates its specifications from DDR3 to DDR4, the refresh circuitry is expected to undergo significant revision [2] primarily because of lack of scalability of current refresh schemes.

We explain the problem of contention from refresh, and our solution to mitigate that, with a simple example. Consider a memory system that takes 1 unit of time for a read request and 8 units of time for refresh. Requests A0-B0, A1-B1, A2-B2, and A3-B3 are to be serviced. A request of type B arrives one unit of time after a request of type A is serviced, and request of type A arrives two units after type B is serviced. Figure 1(a) shows the timing for a system that does not have any refresh-related penalties. It would be able to service these requests in a time period equal to 18 units.

978-1-4673-5587-2/13/$31.00

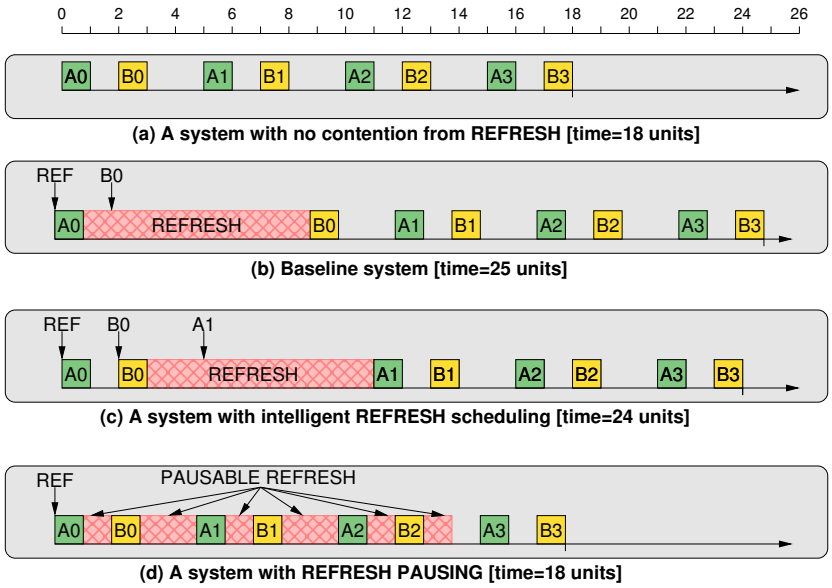

Figure 1: Latency overheads of doing refresh is significant. Intelligent scheduling of refresh helps reduce this latency overhead but is not sufficient. Refresh Pausing can avoid the latency penalty of refresh operations.

Figure 1(b) shows the timing for the baseline system where a refresh operation arrives shortly after A0 is scheduled. The baseline will start the refresh as soon as A0 is serviced, and the refresh will continue for 8 time units. A later arriving read request B0 must wait until the refresh is completed. Therefore B0 gets delayed, and the entire sequence of requests take a time period equal to 25 units. Thus, the overall time has increased significantly compared to a system with no refresh.

A system does not have to schedule a refresh operation as soon as it becomes ready. JEDEC standards specify that a total of up-to 8 refresh operations can be postponed. Therefore, one can design intelligent scheduling polices [3] that try to schedule refresh in periods of low (idle) memory activity. However, given that refresh operations are very long compared to memory read operations (1120 processor cycles for our baseline) the likelihood of finding such a long idle period for a rank is quite low. So, refresh scheduling typically cannot hide the latency of refresh completely, however it may be able to reduce the penalty. Figure1(c) shows the timing of our system with intelligent refresh scheduling. Instead of scheduling a refresh after A0, it waits until after B0, to get a longer idle time. However, this reduces the penalty by only 1 unit, and the entire sequence of request takes 24 units. Thus, refresh scheduling can help, but it is not enough.

Traditional systems treat refresh as a non-interruptible operation. Once refresh is scheduled, the memory gets committed for the time period equal to T_{RFC} (8 units for our example). Assume (for now) that refresh operation can be paused at arbitrary points. Figure1(d) shows the timing of our system with *Pausable Refresh*. Refresh operations now occur only during periods of no activity, and as soon as a read request arrives they relinquish the memory for servicing the pending read. A given refresh operation can be paused and resumed multiple times. With pausing, the entire sequence now takes a time period of 18 units, similar to the system with no refresh penalty. Thus, an interruptible and pausable refresh can reduce (or avoid) the latency penalty due to refresh operations.

This paper proposes *Refresh Pausing*, an interruptible and pausable refresh architecture. It exploits the behavior that for each pulse (every T_{REFI}), a typical DRAM device performs refresh of multiple rows in succession. To refresh a given row, that row is activated, then precharged. And, then the next row is refreshed. We can potentially *Pause* an on-going refresh operation to service a pending read request. We keep track of the address of row undergoing refresh and store that address when the refresh is paused. After the pending read operation finishes, the refresh of the group is resumed using the row address information stored during pause. The number of *Refresh Pause Points (RPP)* is thus dictated by the number of rows in a refresh group. For a DRAM device containing 8 (or 16) rows, we have 7 (or 15) RPP. Thus, while pausing at an arbitrary point may not be practical, with our proposal it becomes possible to pause at many well-defined RPPs.

Our evaluations with a detailed memory system simulator (USIMM) shows that on average removing refresh related penalties has the potential for 7.2% performance improvement and Refresh Pausing provides 5.1% performance improvement. Our implementation of Refresh Pausing avoids extra signal pins between the processor memory interface, and incurs the hardware of only one AND gate and one byte per rank. It reuses the existing pins to indicate pausing.

The paper is organized as follows: Section 2 provides background and motivation, Section 3 design of Refresh Pausing, Section 4 methodology, Section 5 results and analysis. We compare Refresh Pausing with Refresh Scheduling in detail in Section 6, and discuss other related work in Section 7.

2. Background and Motivation

2.1. DRAM Refresh: Background and Terminology

DRAM cells maintain data integrity using refresh operations, a process whereby the data is rewritten to the cell periodically. While DRAM cells have varying retention time [4], the JEDEC standards specify a minimum of 64ms retention time (32ms for high temperature), which means all DRAM rows must be refreshed within this small time period. Let's call this time period as *DRAM Retention Time*. Initially, DRAM arrays had relatively few rows, so the total time in performing refresh operations was small. Therefore it was acceptable to refresh all rows using one refresh pulse every DRAM Retention Time. This is referred to as *Burst Mode* refresh.

As the number of rows in a typical DRAM array increased to few (tens of) thousand, the latency penalty of Burst Mode became unacceptable, as it tied up the memory banks for a latency equivalent to tens of thousands of read operations. To overcome this long latency, JEDEC [1] provided a *Distributed Refresh* mode, whereby a fraction of memory is refreshed at frequent intervals. When JEDEC standards were formed, memories typically had approximately 8K rows per bank, so memory array was divided into 8K groups. For discussion, let's call this group as a *Refresh Bundle*. To ensure that all refresh bundles get refreshed in the DRAM Retention Time, a refresh pulse is now required at a much smaller time period, called *Refresh Interval (T_{REFI})*. The time period for T_{REFI} is simply DRAM Retention Time divided by 8K groups, so it is 7.8µsec (3.9µsec for high temperature). The T_{REFI} remains constant across DRAM generations. A constant T_{REFI} across generations means that a system can mix-and-match different DRAM DIMMs, or upgrade to a different DIMM while still using the same refresh infrastructure that sends one refresh pulse every T_{REFI} interval. The refresh activity is handled entirely inside the DRAM chip, and is triggered by the refresh pulse.

The size of DRAM memory has continued to increase, which means current DRAM banks have more than 8K rows. This is simply handled by refreshing multiple rows for each refresh pulse. For example, for the 8Gb device we consider, there are 8 rows per Refresh Bundle. When the DRAM array gets a refresh pulse, it refreshes 8 rows, one after another. Thus, the time incurred to perform refresh operation for each refresh pulse is a function of number of rows per refresh bundle. This time is referred to as *Refresh Cycle Time (T_{RFC})*.

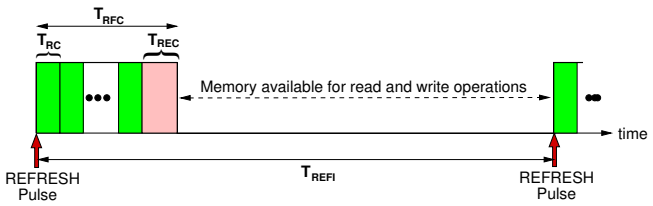

Figure 2: Timing parameters of distributed DRAM Refresh

The time taken to refresh one row is bounded by the *Row Cycle Time (T_{RC})*, which is the time to activate and precharge one row. In general T_{RFC} is greater than the number of rows in a refresh bundle multiplied by the T_{RC}, as refresh operation is also provisioned with Recovery Time (T_{REC}) [5][6] to subside the effects of large current draw. Figure 2 illustrates the different timing parameters related to DRAM refresh performed in distributed refresh mode.

2.2. The Latency-Wall of Refresh

When the DRAM array receives a refresh pulse, the memory gets tied up and then released only after the refresh operation is completed. Thus, the memory is unavailable during refresh period. Lets, define *Refresh Duty Cycle (RDC)* as the percentage time that the memory is doing refresh. RDC can be computed as the ratio of T_{RFC} to T_{REFI} . Ideally, we want small RDC so that the memory is available for servicing demand requests. Unfortunately, RDC is increasing.

The increase in T_{RFC} across technology generations is shown in Table 1. The row cycle time has largely remained unchanged, however the T_{RFC} has increased considerably. For high-temperature server operation, T_{REFI} is 3900ns, so for 8Gb[1] memories available currently[7], RDC is 350ns/3900ns=9%.

Table 1: T_{RC} and T_{RFC} for different DRAM Densities

Memory Density	T_{RC}	T_{RFC}	RDC
1 Gb	39ns	110ns	2.8%
2 Gb	39ns	160ns	5.1%
4 Gb	39ns	300ns	7.7%
8 Gb	39ns	350ns	9.0%

While RDC has been increasing at almost an exponential rate (theoretically about 2x every DRAM generation) in the past, it is expected to increase at an even higher rate in the future because of the combination of the following reasons:

1. **High Density:** As the number of rows in the DRAM array increases, so does the number of rows in a refresh bundle. T_{RFC} can be expected to increase linearly with memory capacity. Thus, RDC would increase in proportion to memory capacity.
2. **High Temperature Operation:** At higher temperature, DRAM cells leak at a faster rate. Therefore, JEDEC specifications dictate that above 85°C memories should be refreshed at 2x the rate at normal temperature. This reduces the T_{REFI} from 64ms to 32ms. A 2x reduction in T_{REFI} corresponds to doubling of RDC.
3. **Increasing Device Variability:** As DRAM devices get pushed into smaller geometries, the variability in per-cell

[1] For meeting the JEDEC specifications of T_{RFC} of 350ns for 8Gb chips, some designs have adopted TwinDie technology[7] which combines two 4Gb dies to create one 8Gb chip. Thus, meeting the JEDEC specifications of T_{RFC} may necessitate significant changes to the DRAM chip architecture.

behavior increases and larger number of weak bits gets placed into the array. To handle such weak bits, the typical refresh rate of DRAM devices could be reduced to 32ms, reducing T_{REFI} by 2x, and increasing RDC by 2X.

4. **Reduction in Row Buffer Size:** The energy efficiency of DRAM memories can be improved by making the the row-buffer smaller in order to reduce over-fetch [8]. Such optimizations increase the total number of rows in memory, and hence the number of rows in a refresh bundle. As T_{RC} remains unaffected, T_{RFC} would increase in proportion to the number of rows, and increase RDC proportionally.

For our studies, we use a refresh rate of 32ms, similar to prior work [3]. This value corresponds to a high temperature operation, typical for dense server environments [3]. It also reflects future technologies where variability in devices may dictate a shorter refresh interval even at room temperature.

Thus, while current DRAM systems spend about 7%-9% of the time performing refreshes, future systems can be expected to spend an even more time. This high refresh duty cycle makes the memory unavailable for longer time, and is the impending *Latency-Wall* of refresh.

2.3. Latency Impact of Refresh

Our baseline assumes 8Gb chips, with T_{RFC} equals to 350ns and T_{REFI} of 3900ns, so the memory system spends 9% of the time doing refresh operations. Figure 3 shows the average latency of baseline as well as if all refresh operations are removed (No Refresh). Detailed methodology is described in Section 4. The bar labeled *AMEAN* represents arithmetic mean over all 18 workloads. The average latency for reads in the baseline system is 234 processor cycles. Whereas, if the contention from refresh operations is removed then the average read latency would get reduced to 215 cycles.[2]

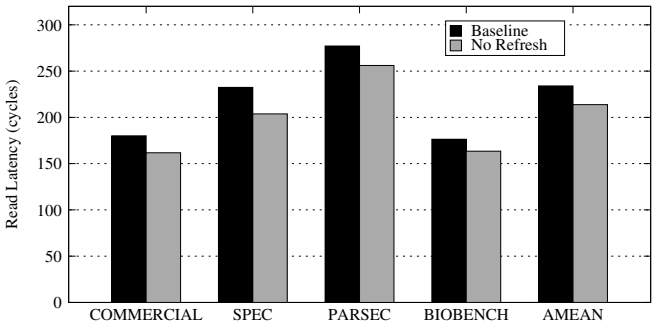

Figure 3: Impact of refresh on read latency

[2]One may simplistically estimate the latency impact from refresh as a product of *collision probability* and *Average delay under collision*. Collision probability is related to RDC (so, it will get approximated as 9%), and average delay under collision is half of T_{RFC} , so 175ns. Therefore, one may estimate that the average delay due to refresh as 0.09*175ns=15.75ns, or 63 processor cycles. However, the implicit assumption in such simple estimation is that a read request is equally likely to come during refresh, as during other times. We found that this key assumption is invalid, as refresh delays the read, which stops or slows down the subsequent read stream; hence this method of estimating latency impact of refresh is incorrect.

2.4. Mitigating Latency Impact via Refresh Scheduling

Refresh operations have a significant impact on read latency of memory system. Reducing this impact can improve read latency and thereby system performance. One potential option to alleviate the latency impact of refresh is exploiting the flexibility in scheduling refresh operations. JEDEC standard provide the ability to postpone refresh operations for up-to 8 T_{REFI} cycles. The work most related to our work, was on scheduling refresh operations, called *Elastic Refresh*[3]. Instead of scheduling a pending refresh operation as soon as the memory becomes idle, this scheme delays the pending refresh for some time. This time is determined based on average time duration of idle periods of the memory queues.

Refresh scheduling schemes, including Elastic Refresh, are unlikely to give significant benefit though, as they need to frequently accommodate a very long latency operation. Finding the memory idle for that long on a regular basis is difficult for memory intensive workloads. Furthermore, scheduling a refresh after a period of time has passed could increase the waiting time for a later arriving read request.

2.5. Performance Potential

Figure 4 shows the performance improvement over baseline if we remove all the refresh operations (No Refresh) and if the baseline adopts Elastic Refresh. The bar labeled *GMEAN* represents geometric mean over all 18 workloads throughout this paper.

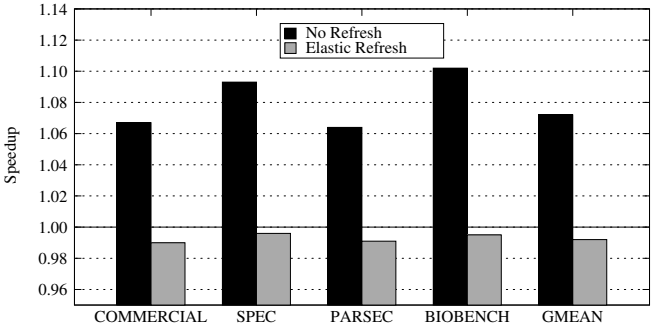

Figure 4: Potential Speedup from removing refresh is significant. However, Elastic Refresh degrades performance (detailed study in Section 6).

The "No Refresh" system has potential for significant performance improvement, 7.2% compared to the baseline. Our baseline has read priority scheduling, so refreshes are delayed in favor of reads, and get done in a forced manner if there are 8 pending refreshes. Compared to this simple refresh scheduling scheme, Elastic Refresh ends up degrading performance. Section 6 will analyze the inefficacy of refresh scheduling algorithms (including Elastic Refresh) in details.

We need a practical solution that can reduce the latency impact of refreshes. Next section presents a scheme that greatly reduces the contention from refreshes, and easily scales to future technologies/situations when RDC will be quite high.

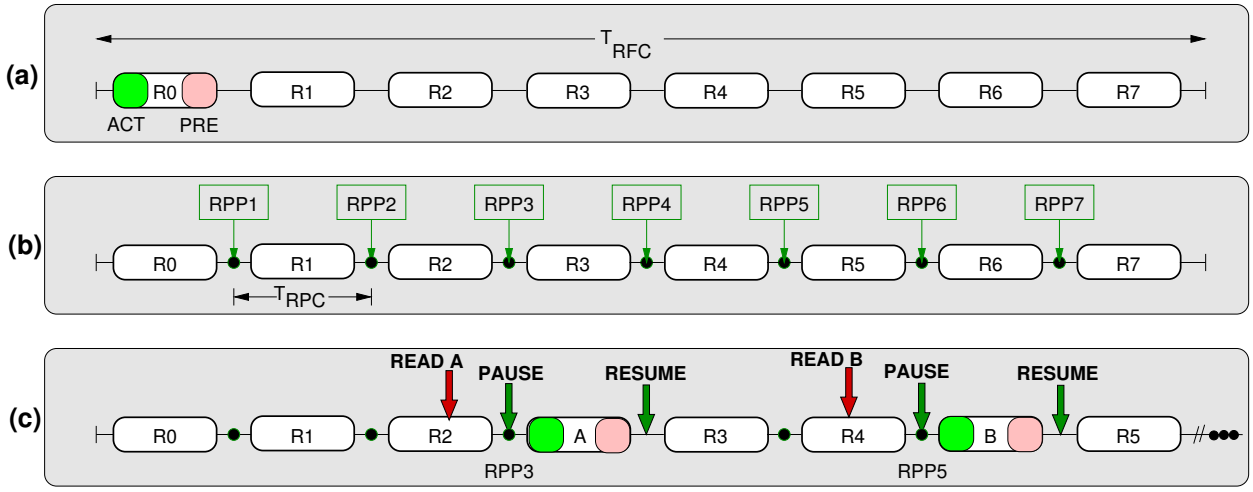

Figure 5: Enabling Refresh Pausing in DRAM systems (a) Refresh operation in traditional DRAM memories (b) Identifying potential pause points (RPP) for Refresh Pausing and (c) How Refresh Pausing can quickly service pending requests

3. Refresh Pausing in DRAM Systems

Traditionally, refresh operations are considered uninterruptible, therefore once a refresh is scheduled later arriving demand requests must wait until the refresh gets completed. The longer the refresh operation, the longer is the expected waiting time for a pending demand request. We avoid this latency penalty due to refresh by making refresh operations interruptible, and propose *Refresh Pausing* for DRAM memory systems. With an interruptible refresh, the refresh can be paused to service a pending read request, and then resumed once the read request get serviced. This section describes the concept, implementation, and implications of Refresh Pausing.

3.1. Refresh Pausing: Concept

While it may not be possible to Pause a refresh at an arbitrary point, there are some well defined points during refresh, where it can potentially be paused in order to service a pending read request. Consider the refresh operation done in traditional DRAM systems, as shown in Figure 5(a). In a time interval of T_{RFC} , the DRAM array refreshes say 8 rows, numbered R0 to R7. To refresh a row, the given row is activated, and then the bank waits for a time period equal to T_{RAS}, and then precharges the row. This cycling takes a time equal to T_{RC}. Then shortly after, the next row is refreshed and so on, until all rows R0-R7 are refreshed.

When one row is refreshed, we can potentially pause the refresh operation and relinquish the DRAM array for servicing some other operation, as shown in Figure 5(b). Each such potential point of pausing is called *Refresh Pause Point (RPP)*. For a memory with N rows in a refresh bundle, there would be (N-1) RPP. In practice, the time interval of T_{RFC} is longer than simply the sum of row cycle times because of recovery time. Details about how the recovery time is calculated, or provisioned, are not typically provided by DRAM vendors. In our work, we assume that the recovery time is spread out over all rows. Therefore, we divide the time T_{RFC} into 8 (in general N) equal time quanta, and call this duration *Refresh Pause Cycle* (T_{RPC}).[3] A memory array that supports Pausing can potentially Pause at an interval of every T_{RPC}.

Figure 5(c) shows the working of memory system with Refresh Pausing. Let's say a read request for row A arrives while Row R2 is being refreshed. The memory controller signals the device to pause, the device pauses refresh at the next RPP, which is RPP3. The memory then services A, and then the memory controller can signal the refresh circuit to RESUME the refresh operation. A refresh operation can be paused and resumed multiple times, as shown for a request for Row B which arrives while refreshing Row R4.

Refresh operations cannot be paused indefinitely if there is a heavy read traffic. When the refresh operation is done because it has reached the refresh deadline ($8\times T_{REFI}$), then it cannot be paused. We refer to such refresh operations as *Forced Refresh*. To maintain data integrity, and to confirm to JEDEC standards, pausing is disallowed for forced refresh.

3.2. Refresh Pausing: Implementation

To facilitate Refresh Pausing, we need to make minor changes to the memory controller and the DRAM devices. The task of the memory controller is to decide if and when to PAUSE an on-going refresh, and when to RESUME a paused refresh, depending on the occupancy of the memory queues. The task of the DRAM refresh circuit is to PAUSE the on-going refresh operation at the next RPP, if the pause signal is received. And to RESUME from the check-pointed state, a paused refresh gets resumed. Figure 6 shows the system that implements Refresh Pausing. Our implementation is geared toward keeping the hardware modifications to minimum.

[3]We make the assumption of equal time quanta only for simplicity. DRAM vendors can adapt the definition of T_{RPC} and placement of RPP, depending on their specific implementation of recovery time management.

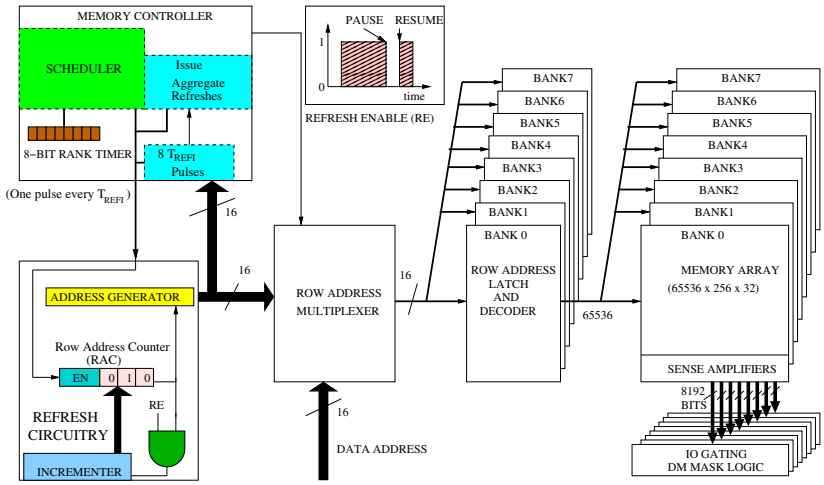

Figure 6: Implementing Refresh Pausing with: (1) reusing REFRESH ENABLE signal to indicate REFRESH, PAUSE, and RESUME (2) an AND gate in DRAM refresh circuit to check for RE during refresh (3) A one-byte timer in the memory controller.

Signaling REFRESH, PAUSE, and RESUME: A naive implementation may provision additional signal pins for PAUSE and RESUME. However, extra signal pins are costly and a deterrent for adoption in standards, so we simply reuse the existing signal REFRESH ENABLE (RE). A DRAM refresh circuitry starts the refresh procedure once RE gets asserted. The role of RE during the refresh is unimportant for traditional systems. To facilitate Refresh Pausing, we simply require that RE must remain asserted during the entire refresh period for an uninterruptible refresh. A de-assertion of RE indicates a request for PAUSE. A RESUME is treated same as REFRESH, in that it starts a regular REFRESH operation, but only refreshes the rows remaining in the refresh bundle.

Changes to DRAM Refresh Circuit: When REFRESH ENABLE (RE) signal gets asserted, the refresh circuitry probes a register called *Row Address Counter (RAC)* that points to the next row to be refreshed. In each refresh iteration, the row pointed to by the RAC gets refreshed and the RAC is incremented. This is done until the number of rows in a refresh bundle gets refreshed. Thus, after the refresh for one pulse is done the RAC stores the row address for the next refresh pulse. To support PAUSE, the refresh circuitry simply checks if the RE remains asserted at each RPP. If not, the refresh operation gets stalled. On RESUME, the refresh operation gets performed till RAC reaches the end of the refresh bundle. Thus, to support Refresh Pausing, we need only *one additional AND* gate in the DRAM refresh circuitry.

Changes to Memory Controller: The memory controller needs to keep track of the amount of time that a refresh has completed, to remove it from the refresh queue as well as to schedule a PAUSE. A PAUSE must be send at-least one cycle before the RPP point. To enable such time tracking, we keep a one-byte timer for each rank. For the refresh operation in service or paused this timer indicates the time spent in doing refresh. Thus, even with Refresh Pausing, the direction of signals is still from memory controller to DRAM circuits, and the operation of DRAM still remains deterministic.

3.3. Summary of Hardware/Interface Support

Our implementation of Refresh Pausing avoids extra pins or signals. However, it relies on modifying the specification of RE signal during on-going refresh operation. The DRAM refresh circuitry needs one AND gate. And, the memory controller needs one byte for time keeping. The hardware for AND gate and time keeping is incurred per rank, as refresh is typically done on a per-rank basis. Thus, implementing Refresh Pausing requires negligible support in terms of hardware and interfaces.

3.4. Implication on Reducing Latency Overhead

With Refresh Pausing, the maximum time that a later-arriving read request has to wait gets shortened from T_{RFC} to T_{RPC}, about 8x if we have 8 rows in the refresh bundle. The average waiting time can be expected to reduce by 8x as well, assuming the refresh is not done in Forced mode. Such a significant reduction in waiting time greatly reduces the latency impact of refresh and improves system performance.

3.5. Implication on Scalability to Future Technologies

As the density of DRAM memories increases and more and more rows get packed into a DRAM array, the specified T_{RFC} is expected to increase at an alarming rate. This would make traditional memory designs unavailable for significant periods of time and increase latency greatly. However, with Refresh Pausing the contention remains bounded to T_{RPC}, almost independent of T_{RFC} and memory size. Thus, Refresh Pausing can enable future memory designs to overcome the latency wall due to refresh induced memory unavailability.

4. Experimental Methodology

4.1. System Configuration

We use the memory system simulator USIMM [9] from the recently conducted MSC (Memory Scheduling Championship) [10]. USIMM models DRAM system in detail, enforcing the various timing constraints. We modified USIMM to conduct a detailed study for refresh operations. We added a refresh queue (REFQ) in addition to the existing Read Queue (RDQ) and write queue (WRQ). Refresh operations thus become part of scheduling decisions. The REFQ is incremented every T_{REFI} . The scheduler for a channel can issue a read, a write, or a refresh to a rank every memory cycle.

The parameters of system configuration are shown in Table 2. We model a quad-core system operating at 3.2GHz. The memory system is configured as a 4-channel design operating at 800MHZ. The memory system is composed of channels, ranks, and banks. The RDQ and WRQ are on a channel basis, and the REFQ is provisioned on a rank basis. We use the default write scheduling policy of USIMM that services writes at the lowest priority, using high and low watermarks to decide when to drain the write queue. To schedule memory requests, we adopt close-page policy, which is known as a better scheduling policy in multiprogram platform.

The refresh scheduling policy in our baseline favors reads over refresh requests unless the REFQ becomes full (8 pending requests). Refresh operation takes T_{RFC} cycles to complete. In all scheduling policies, the number of refresh can be accumulated up to eight without breaking the rules specified by JEDEC standards. We use 8Gb devices for our study, the timing parameters for which is shown in Table 3. T_{RFC} is 280 DRAM cycles (350ns [1]). We use a T_{REFI} of 3.9 micro seconds, which translates to 3120 DRAM cycles.

Table 2: Baseline System Configuration(default USIMM)

Number of cores	4
Processor clock speed	3.2GHz
Processor ROB size	160
Processor retire width	4
Processor fetch width	4
Processor pipeline depth	10
Last Level Cache (Private)	1MB per core
Cache line size	64Byte
Memory bus speed	800MHz
DDR3 Memory channels	4
Ranks per channel	2
Banks per rank	8
Rows per bank	128K/256K
Columns(cache lines) per row	128
Write queue capacity	64
Write queue high watermark	40
Write queue low watermark	20

Table 3: DRAM timing parameters for our memory system

Timing Parameters	DRAM Cycles (at 800MHZ)	Processor Cycles (at 3.2GHz)
T_{RCD}	11	44
T_{RP}	11	44
T_{CAS}	11	44
T_{RC}	39	156
T_{RAS}	28	112
T_{FAW}	32	128
T_{RFC}	280	1120
T_{REFI}	3120	12480

4.2. Workloads

We use the workloads from the recently held Memory Scheduling Championship (MSC) [10], as it contains a wide variety of applications. Table 4 shows key characteristics of our workloads. The MSC suite contains five commercial applications, $comm1$ to $comm5$. There are nine benchmarks from the PARSEC suite, including two mutlithread-versions of applications fluid and canneal (marked $MT\text{-}fluid$ and $MT\text{-}canneal$). Also, there are two benchmarks each from the SPEC[4] suite and the biobench suite.

We execute these benchmarks in rate mode on the quad-core processor. We compute the execution time as the time to finish the last benchmark in the workload (as the benchmarks are executed in rate mode, the variation in execution time of individual benchmarks within the workload is negligible).

Table 4: Workload Characteristics (suite from MSC [10]).

Suites	Workloads	MPKI	Read Latency	IPC
COMMERCIAL	comm1	6.6	186	1.73
	comm2	7.5	221	1.30
	comm3	3.2	186	2.28
	comm4	2.2	195	2.63
	comm5	1.4	195	2.89
SPEC	leslie	6.4	313	1.15
	libq	13.6	191	0.94
PARSEC	black	2.8	252	2.27
	face	6.0	455	1.66
	ferret	4.8	305	1.98
	fluid	2.4	246	2.46
	freq	2.7	226	2.53
	stream	3.4	232	2.25
	swapt	2.9	229	2.35
	MT-canneal	13.2	215	2.88
	MT-fluid	1.4	539	0.97
BIOBENCH	mummer	19.3	187	0.81
	tigr	26.9	184	0.79

[4]We also evaluated other memory intensive benchmarks from the SPEC suite and found that Refresh Pausing provides similar performance improvement for memory intensive SPEC workloads, as it does for the MSC suite.

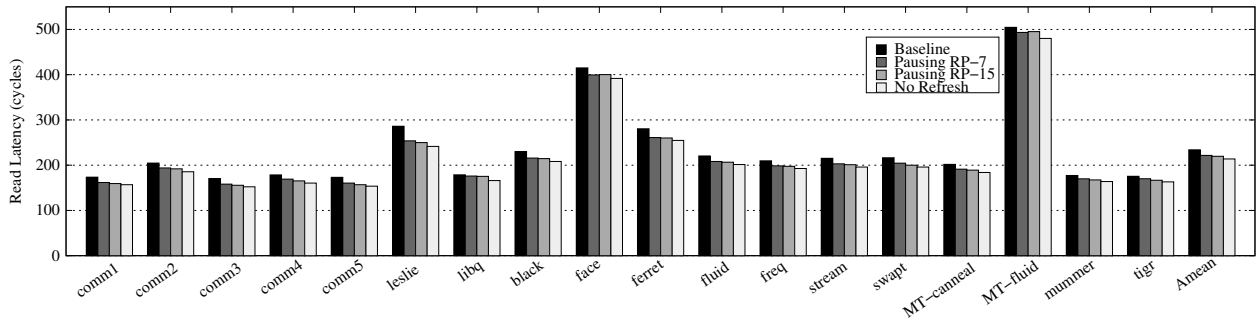

Figure 7: Average read latency for different systems

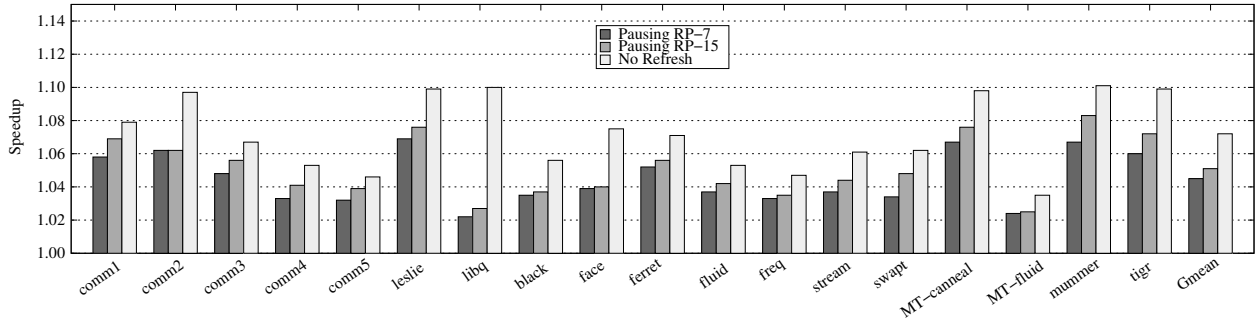

Figure 8: Performance Improvement from Refresh Pausing

5. Results and Analysis

In this section, we analyze the effectiveness of Refresh Pausing. For our baseline memory system with 8Gb chips, there are 8 rows in a refresh bundle, so there can potentially be 7 refresh pause points. However, for future memory systems, the number of rows in a refresh bundle is expected to increase (both because of increase in capacity and decrease in row buffer size). So, to indicate the effectiveness of Refresh Pausing for future memory systems we will also consider a version that has 16 rows in the refresh bundle, therefore 15 refresh pause points. We will refer to the configuration that implements Refresh Pausing with 7 pause points as *RP-7* and the one with 15 pause points as *RP-15*.

5.1. Impact on Read Latency

The read latency of a system is increased by contention due to refresh operations. Figure 7 shows the read latency of our baseline system, the baseline system with Refresh Pausing (RP-7 and RP-15), and the ideal-bound system with No Refresh. We report latency in terms of processor cycles. The bar labeled *Amean* denotes the arithmetic average over all the 18 workloads. For the baseline system, the average read latency is 234 cycles, and for the system with No Refresh it is 215 cycles. Thus, a significant fraction of read latency (19 cycles) is due to contention from refresh. With Refresh Pausing, this latency impact gets reduced to 7 cycles (RP-7) and 4 cycles (RP-15). Thus, Refresh Pausing can remove about half to two-thirds of the delay induced by refresh operations.

5.2. Impact on Performance

The reduction in read latency with Refresh Pausing translates into performance improvement. Figure 8 shows the speedup from Refresh Pausing (RP-7 and RP-15) and No Refresh. The bar labeled *Gmean* denotes the geometric mean over all the 18 workloads. For a system with No Refresh, there is a potential gain of approximately 7.2% on average. Refresh Pausing obtains 4.5% improvement with RP-7 and 5.1% improvement with RP-15. Thus, with Refresh Pausing we can get about half to two-thirds of the potential performance gains. Few workloads, such as *comm1*, *MT-canneal*, and *mummer* obtain significantly better performance improvement with RP-15 than with RP-7.

We observed that on an average 2.7 pauses and 3.6 pauses per refresh was incurred for the RP-7 and RP-15 schemes respectively, with 90% of the refreshes incurring pausing.

Our default configuration consists of 1MB per core LLC. We varied the LLC size from 512KB per core to 2MB per core. For 512KB per core, RP-7 provides 4.7%, RP-15 provides 5.2% and No Refresh provides 7.5% performance improvement, on average. With 2MB per core, these become 4%, 4.6%, and 6.4% respectively. The performance benefit of Refresh Pausing is robust to cache size.

5.3. Sensitivity to Page Closure Policy

Similar to typical server systems, our baseline employs a close-page policy, as we found that close-page policy had better performance than open-page policy. However, our proposal is applicable to open-page systems as well. Figure 9

shows the speedup from Refresh Pausing (RP-7 and RP-15) and No Refresh, all implemented on a system employs open page policy (with row buffer friendly data mapping policy). No Refresh still gains approximately 5.9% on average and Refresh Pausing becomes even more effective, RP-7 improves performance by 5.5% and RP-15 by 6.8%, over the open page policy. A paused refresh serves as an implicit page closure, so an access that would get a row-buffer conflict with open page avoids the row-precharge latency. These results are consistent with the evaluations reported in Memory Scheduling Championship, that showed that the default close page policy was better than most of the open-page policies, due to contention in the memory reference streams from different cores. Thus the performance of RP-15 is found to exceed that of No Refresh for a system that employs open page policy.

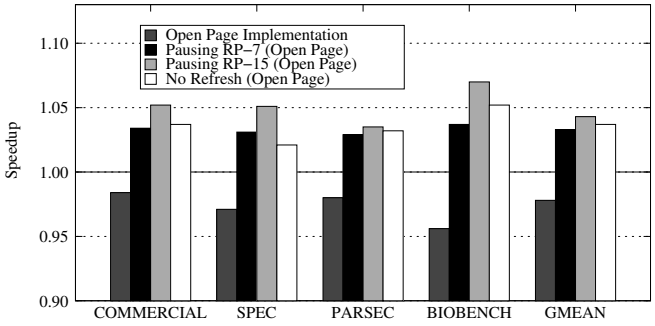

Figure 9: Speedup from Refresh Pausing on systems that employ open-page policy compared to close-page baseline

5.4. Refresh Pausing in Highly Utilized Systems

Refresh Pausing tries to exploit idle cycles in memory for doing refresh, and relinquishing refresh as soon as memory is required for a demand read request. If the memory is almost always busy with servicing reads and writes, then the refresh operations get done in a forced manner. Since Forced Refresh cannot be paused, there is little scope for Refresh Pausing to improve performance. To analyze Refresh Pausing for highly utilized systems, we re-configured our baseline to have 1-channel instead of 4-channels. This increases memory utilization greatly and hence reduces the idle periods available for doing refresh operations.

Figure 10 shows the performance improvement with Refresh Pausing (RP-7 and RP-15) and with No-Refresh. The potential performance improvement with No Refresh is approximately 6.3%. However, the improvement with Refresh Pausing is reduced, with RP-7 providing 2% performance improvement and RP-15 providing negligible 0.003% performance degradation respectively. The RP-15 is less effective because it gets paused often, and the refreshes eventually get done in Forced mode, which cannot be paused. In the limit, if the memory is 100% utilized then all refreshes will be done as Forced Refresh, leaving no scope for performance improvement with Refresh Pausing.

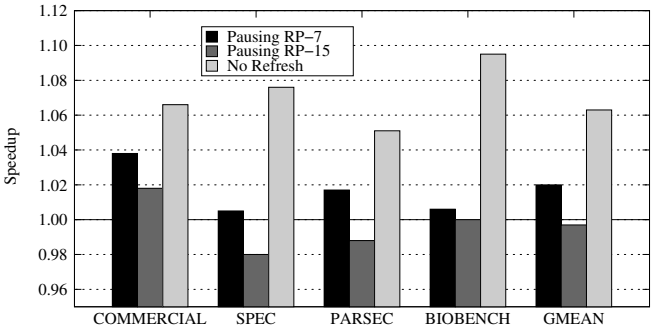

Figure 10: Performance impact of Refresh Pausing in a highly utilized memory system (number of channels in baseline reduced from 4 to 1)

5.5. Comparisons with Alternative Proposals for DDR4

As DRAM chips scale from 8Gb node to higher densities, JEDEC is updating the DDR specifications from DDR3 to DDR4. One of the critical elements that is likely to change in DDR4 is the refresh circuitry [2]. One of the refresh proposal that is being considered for DDR4 is fine-grained refresh scheme which lowers the refresh interval T_{REFI} and T_{RFC} both by a factor of either 2x or 4x, and are called *Refresh Rate Mode-x2 (RRMx2)* or *Refresh Rate Mode-x4 (RRMx4)* [2]. Unfortunately, these proposals are not as effective at tolerating refresh latency as Refresh Pausing. Furthermore, as memory capacity increases, these granularity of these modes will need to be revised to tolerate longer refresh.

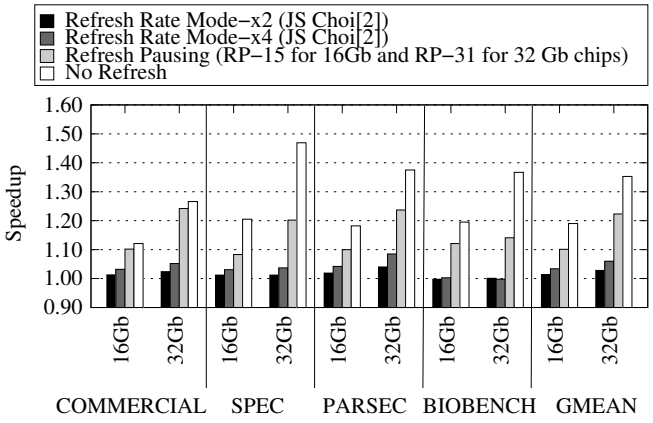

Figure 11: Effectiveness of Refresh Pausing at High Density

Figure 11 compares Refresh Pausing for High Density (16Gb and 32Gb) chips with RRMx2 and RRMx4. Refresh pausing gives significant performance gains of 10% for 16Gb and 22% for 32Gb. Comparatively, RRMx2 gives 1.2% for 16Gb, and 2.8% for 32Gb. And, RRMx4 gives 3.4% for 16Gb and 6% for 32Gb. The limited effectiveness of RRMx happens because it still incurs the penalty of locking up the rank for long refresh periods more frequently. Thus, for high density devices (say 32Gb), Refresh Pausing provides more than double the performance improvement compared to one of the

aggressive proposal being considered by JEDEC (RRMx4). Given the low implementation complexity (one AND gate in refresh controller) and high effectiveness, Refresh Pausing is a strong candidate for future standards.

5.6. Sensitivity to Temperature

Table 5 shows the performance improvement with Refresh Pausing and No Refresh, for different operating temperature at different chip densities. Thus, even for low operating temperature ($< 85°C$), Refresh Pausing provides 11.5% performance improvement for 32Gb chips. As chip densities increase, future chips can use Refresh Pausing to mitigate the latency impact of refresh across all operating temperatures.

Table 5: Performance Improvement of Refresh Pausing at different temperature ranges and densities

	No Refresh		Refresh Pausing	
Density	< 85°C	> 85°C	< 85°C	> 85°C
8Gb	3.5%	7.2%	2.6 %	5.1 %
16Gb	9.7%	19%	5%	10.1%
32Gb	18.4%	35.3%	11.5%	22.3%

6. Refresh Pausing vs Refresh Scheduling

We proposed Refresh Pausing to mitigate refresh-related latency penalties. An alternative option to tolerate the delay from such long-latency refresh operations is to do intelligent Refresh Scheduling. The scheduler can place the refresh operations in a time period when memory is idle. A key prior work, Elastic Refresh [3], performed such refresh scheduling and their study indicated that simply doing refresh scheduling can mitigate almost all of the performance loss from refresh. Our evaluations, however, show that this is unlikely to be the case for memory intensive workloads. The initial set of results presented in Figure 4 showed that Elastic Refresh degrades performance compared by 1.3% on average compared to our baseline refresh scheduling policy. In this section, we analyze Elastic Refresh, the requirements for Elastic Refresh to be effective, understand the reasons for performance loss, place approximate bounds on intelligent refresh scheduling, and compare Refresh Pausing and Refresh Scheduling for scaling to future technologies.

6.1. Elastic Refresh

A system can delay the servicing of a refresh request for a time period of up-to 8 T_{REFI}. Therefore, if the system is busy servicing reads, current systems would delay the refresh till either the rank is idle or the refresh deadline approaches. The critical idea behind Elastic Refresh (ER), is to not schedule a refresh even if the rank is idle but rather have a wait-and-watch decision. After a rank becomes idle, ER waits for a given time period (say t_{ER}), as determined by the average idle time of the rank and the number of pending refreshes. If the rank does receive a demand request within t_{ER}, then the wait-and-watch time gets reset. However, if the rank does not receive any request within t_{ER}, it schedules the pending refresh. The key objective is to avoid a long-latency of refresh when a read is predicted to come within a short time period.

6.2. Requirements for Elastic Refresh to be Effective

There are three key requirements for ER to be effective. First, there must be a large number of idle periods with duration longer than T_{RFC}. Otherwise, waiting for the right time to schedule refresh will be rendered ineffective. Second, even if there are a large number of long idle periods (LIP) in a workload, these idle periods must be spread throughout the program execution. More specifically, idle period must be within $8 \cdot T_{RFC}$ of the refresh request time. Third, the hardware predictor should predict the start of LIP correctly.

6.3. The Loss Scenarios for Elastic Refresh

ER does not schedule a pending refresh operation even if the rank is idle. This does have a potential disadvantage compared to a simple scheme that schedules refresh operation if no other requests are available. Consider the example shown in Figure 12 for a system where read takes 1 unit of time and refresh 8 units of time. When A is serviced (at time 1) the rank is idle. The simple policy will schedule refresh immediately. A later arriving read request B at time 7 will have to wait for 2 time units. However, with ER if we delay the start of refresh by say 3 time units, and the read request B arrives at time 7, it will have to wait for 5 time units. Thus, the wait-and-watch policy of ER can degrade performance.

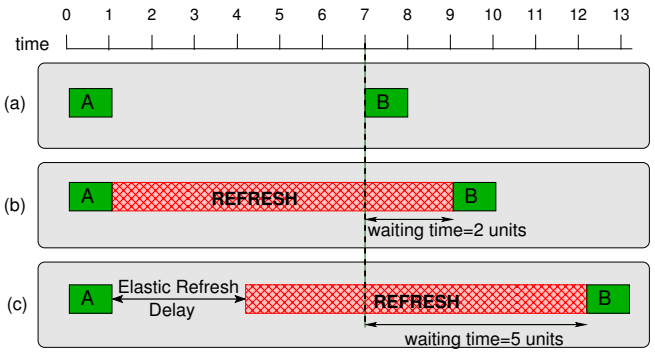

Figure 12: Waiting time for a system with (a) No Refresh (b) baseline scheduling (c) Elastic Refresh

The implicit assumption in design of ER, is that idle periods are either less than average delay or longer than T_{RFC}. If a request has idle period within these two values, it will result in higher latency with ER than with baseline scheduling which services a pending refresh operation as soon as the rank becomes idle.

Figure 13 shows the Cumulative Density Function (CDF) of idle periods in cycles. The maximum value of in x-axis is 1120 cycles, same as T_{RFC} (350ns). The vertical dotted line

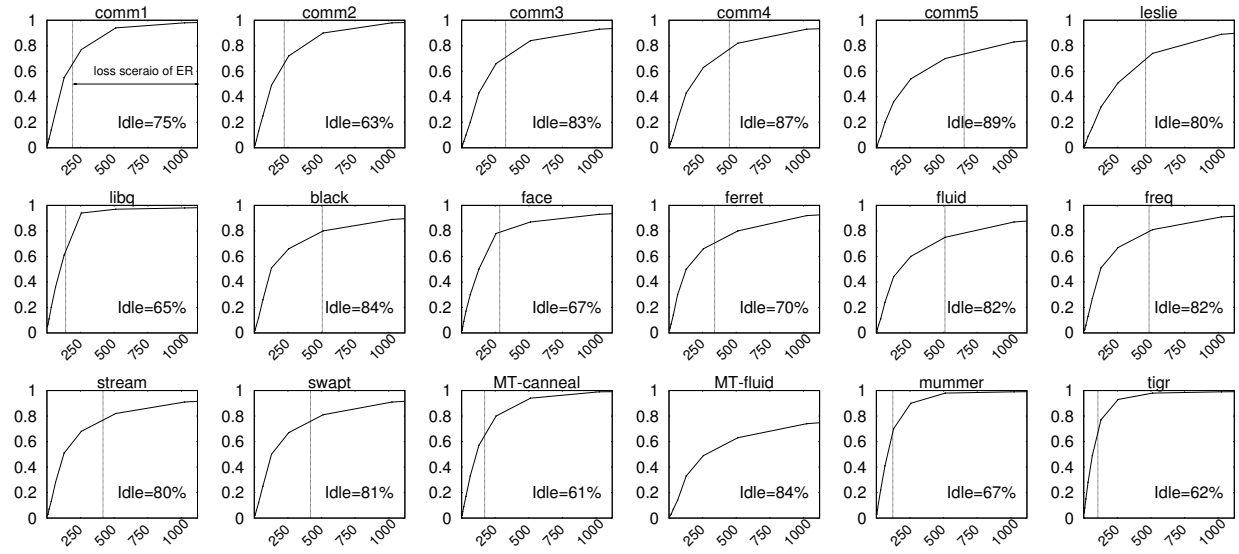

Figure 13: Cumulative density function of idle periods of a rank in processor cycles. The percentage of execution time that the rank is idle is specified within the figure. The vertical line represents the average length of an idle period. The maximum value of x-axis is 1120 cycles, similar to T_{RFC}. The average length of idle period for MT-Fluid is 1330.

indicates the average idle time of the rank, one of the parameters used by ER. Note that idle periods longer than T_{RFC} are very few. And, about 20% of idle periods are between average idle period and T_{RFC}, which represents a loss scenario for ER. So, for our workloads the loss scenarios with ER are quite common, hence the performance loss.

6.4. Mitigating Loss of ER with Oracle Information

To validate our hypothesis that the performance degradation of ER is indeed due to the wait-and-watch policy of ER based on average idle time, we conducted the following study. We assumed that ER is extended to have oracle information, such that when ER decides to schedule a refresh after a time period t_{ER}, we give ER prior credit, and schedule the refresh operation as if it was scheduled as soon as the idle period began. Note this is not for practical implementation but only a study to gain insights. With such an *Oracle-Based Loss Mitigation (OBLM)*, ER would avoid the loss scenario.

Figure 14 shows the speedup of Elastic Refresh and Elastic Refresh with OBLM. Elastic Refresh degrades performance by 1.3% on average; however with OBLM it improves performance by 2.0%. Thus, the wait-and-watch based on average delay is costing ER a performance loss of about 3.3%.

6.5. Potential Performance of Refresh Scheduling

With ER+OBLM, the decision of whether to schedule a refresh or not is still with the ER scheduler, which makes this decision based on average idle period. We also tried to estimate the performance potential of intelligent refresh scheduling, without regards to hardware implementation. At the end of each idle period, we decide whether or not refresh should have been scheduled at the start of the idle period. If the idle period is greater than a threshold we assume that the pending refresh was issued at the beginning of the idle period. We reduce this threshold linearly with the number of pending refreshes. Figure 14 also shows the performance of such *Oracle-Based Linear Refresh Scheduling*. We observe that with perfect information about the future we can get approximately 3.7% performance improvement on average. While this is smaller than we get with Refresh Pausing, it sill indicates a good opportunity of future research to develop effective refresh scheduling algorithms.

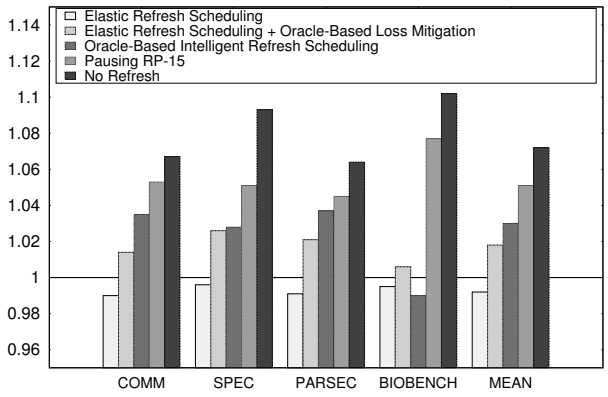

Figure 14: Speedup for Elastic Refresh and Elastic Refresh with Oracle-Based Loss Mitigation

6.6. Scaling to Future Technologies

As devices move to higher densities, T_{RFC} will become longer, which means refresh scheduling will have to accommodate even longer refresh operations. Finding larger idle periods is harder than finding smaller ones. On the other hand, Refresh Pausing would require accommodating only a delay of T_{RPC} (similar to row cycle time). Therefore, Refresh Pausing is more scalable and effective at higher densities. However, both techniques are orthogonal and can be combined.

7. Other Related Work

The latency of refresh operations can be reduced by having more banks and thus doing more refreshes in parallel. However, refresh is already a current limited operation, so increasing parallel refresh operations may not be practical.

Another option is to tune the refresh operations based on the DRAM retention characteristics. Such schemes, can either decommission high refresh pages [11] or use multi-rate refresh where high-retention pages (rows) are refreshed infrequently [12][13][11]. These approaches rely on a having retention characteristics of DRAM cells available, and that these characteristics do not change. Unfortunately, such an assumption can result in data-loss in practice, as captured in Jacob et al.'s [6] book on Memory System (Anecdote IV): "The problem is that these [proposals to exploit variability in cell leakage] ignore another, less well-known phenomenon of DRAM cell variability, namely that a cell with a long retention time can suddenly (in the frame of second) exhibit a short retention time". Such, *Variable Retention Time (VRT)* would render these proposals functionally erroneous. Avoiding refresh operations may still be possible in specific scenarios. For example when the row has recently been accessed [14], or when the data is not critical [15][16], or when the system provisions ECC to tolerate data errors [17].

Refresh operations can also be done at a finer granularity, such as on a per-bank basis instead of a per-rank basis [18]. This would enable better refresh scheduling, exploiting the time periods when the bank is idle (while the rank may still be busy). Our proposal is applicable to (and remains effective for) such per-bank implementations as well.

8. Summary

DRAM is one of the most *forgetful* memory technology, with retention time in the range of few milliseconds. It relies on frequent refresh operations to maintain data integrity. The time required to do refresh is proportional to the number of rows in memory array; therefore this time has been increasing steadily. Current high density 8Gb DRAM chips spend 9% of the time doing refresh operations, and this fraction is expected to increase for future technologies.

Refresh operations are blocking, which increases the wait time for read operations, thus increasing effective read latency and causing performance degradation. We mitigate this latency problem from long-latency refresh operations, by breaking the notion that refresh needs to be an uninterruptible operation, and make following contributions:

1. We propose Refresh Pausing, a highly effective solution that significantly reduces the latency penalty due to refresh. Refresh Pausing simply pauses an on-going refresh operation to serve a pending read request.
2. We show that Refresh Pausing is scalable to future technologies, such as DDR4, which will have very high (T_{RFC}). With Refresh Pausing, the latency impact of refresh is determined less by T_{RFC} , and is simply a function of row cycle time T_{RC}, which tends to remain unchanged.
3. We show that Refresh Pausing is much more effective than simply relying on Refresh Scheduling. Refresh Pausing not only significantly outperforms Refresh Scheduling algorithms, but it becomes even more attractive as design move to future technology nodes.

Our evaluations, for current 8Gb chips, using a detailed memory system simulator show that removing refresh can provide 7.2% performance improvement and Refresh Pausing can provide 4.5%-5.1% performance improvement. Implementing Refresh Pausing entails negligible changes to the DRAM circuitry (one AND gate), reusing existing signal, and a 1-byte timer in memory controller. Given the impending latency wall of refresh, the simplicity of our proposal makes it appealing for adoption in future systems and standards.

Acknowledgments

Thanks to Jeff Stuecheli and Rajeev Balasubramonian for discussions and feedback. Moinuddin Qureshi is supported by NetApp Faculty Fellowship and Intel Early Career Award.

References

[1] *JESD79-3F*, JEDEC Committee JC-42.3 Std. DDR3, 2010.
[2] (2011) Js choi ddr4 miniworkshop. [Online]. Available: http://jedec.org/sites/default/files/JS_Choi_DDR4_miniWorkshop.pdf
[3] J. Stuecheli, D. Kaseridis, H. C.Hunter, and L. K. John, "Elastic refresh: Techniques to mitigate refresh penalties in high density memory," in *MICRO-43*, 2010.
[4] H. Kim, B. Oh, Y. Son, K. Kim, S.-Y. Cha, J.-G. Jeong, S.-J. Hong, and H. Shin, "Characterization of the variable retention time in dynamic random access memory," *Electron Devices, IEEE Transactions on*, vol. 58, no. 9, pp. 2952 –2958, sept. 2011.
[5] *TN-47-16 Designing for High-Density DDR2 Memory*, Micron, 2009.
[6] B. L. Jacob, S. W. Ng, and D. T. Wang, *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2008.
[7] *MT41J512M4:8Gb QuadDie DDR3 SDRAM – Rev. A 03/11*, Micron, 2010.
[8] A. Udipi *et al.*, "Rethinking dram design and organization for energy-constrained multi-cores," in *ISCA-37*, 2010.
[9] N. Chatterjee, R. Balasubramonian, M. Shevgoor, S. Pugsley, A. Udipi, A. Shafiee, K. Sudan, M. Awasthi, and Z. Chishti, "USIMM: the Utah SImulated Memory Module," University of Utah, Tech. Rep., 2012, uUCS-12-002.
[10] (2012) Memory scheduling championship (msc). [Online]. Available: http://www.cs.utah.edu/~rajeev/jwac12/
[11] R. K. Venkatesan, S. Herr, and E. Rotenberg, "Retention-aware placement in dram (rapid):software methods for quasi-non-volatile dram," in *HPCA-12*, 2006.
[12] J. Kim and M. Papaefthymiou, "Block-based multi-period refresh for energy efficient dynamic memory," in *ASIC/SOC Conference, 2001. Proceedings. 14th Annual IEEE International*, 2001, pp. 193 –197.
[13] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu, "Raidr: Retention-aware intelligent dram refresh," in *ISCA*, 2012, pp. 1–12.
[14] M. Ghosh and H.-H. S. Lee, "Smart refresh: An enhanced memory controller design for reducing energy in conventional and 3d die-stacked drams," in *MICRO-40*, 2007.
[15] C. Isen and L. John, "Eskimo: Energy savings using semantic knowledge of inconsequential memory occupancy for dram subsystem," in *MICRO-42*, 2009.
[16] S. Liu *et al.*, "Flikker: saving dram refresh-power through critical data partitioning," in *ASPLOS-XVI*, 2011.
[17] C. Wilkerson *et al.*, "Reducing cache power with low-cost, multi-bit error-correcting codes," in *ISCA-37*, 2010.
[18] V. Cuppu, B. Jacob, B. Davis, and T. Mudge, "A performance comparison of contemporary dram architectures," in *ISCA-26*, 1999.