Refresh Pausing in DRAM Memory Systems

PRASHANT J. NAIR, CHIA-CHEN CHOU, and MOINUDDIN K. QURESHI,
School of Electrical and Computer Engineering, Georgia Institute of Technology

Dynamic Random Access Memory (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 article 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.

Categories and Subject Descriptors: B.3.1 [**Hardware**]: Semiconductor Memories

General Terms: Dynamic Random Access Memory, Refresh, Subarray, Sub-bank, Performance

Additional Key Words and Phrases: Memory scheduling, memory controller

ACM Reference Format:
Prashant J. Nair, Chia-Chen Chou, and Moinuddin K. Qureshi. 2014. Refresh pausing in DRAM memory systems. ACM Trans. Architec. Code Optim. 11, 1, Article 10 (February 2014), 26 pages.
DOI: http://dx.doi.org/10.1145/2579669

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 work is an extension of the conference paper, "A Case for Refresh Pausing in DRAM Memory Systems" [Nair et al. 2013]. This submission adds the following items that are not present in the original paper. (1) Section 5.3 describes the scalability of Refresh Pausing; (2) Section 5.4 describes the distribution of pauses within any refreshing intervals; (3) Section 5.5 describes impact of Refresh Pausing on forced refreshes; (4) Section 6.3 gives a mathematical overview and insight on the impact of normal and forced refreshes on ER; (5) Section 7 describes the applicability of Refresh Pausing on power-constrained DRAM systems; These add more than 30% newer material in terms of giving greater insight into the behavior of forced refreshes on baseline and elastic refresh, distribution and scalability of pausing, and also applicability of refresh to subarrays and sub-banks.

Authors' addresses: P. J. Nair, C.-C. Chou, and M. K. Qureshi, School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA 30332; email: {pnair6, cchou34, moin}@gatech.edu.

DOI: http://dx.doi.org/10.1145/2579669

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 [JEDEC Committee JC-42.3 2012] specify that DRAM devices must be refreshed every 64ms (32ms at above 85°C temperature). All 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 period 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 7.8μs (3.9μs 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, and only the number of rows refreshed per refresh pulse has increased. For example, for the 8Gb DRAM chips that we consider, each refresh pulse refreshes 8 to 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 [JEDEC-DDR4 2011] 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.

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 takes 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 eight refresh operations can be postponed. Therefore, one can design intelligent scheduling polices [Stuecheli et al. 2010] 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 (1,120 processor cycles for our baseline), the likelihood of finding such a long idle period for a rank is quite low. Thus, refresh scheduling typically cannot hide the latency of refresh completely;

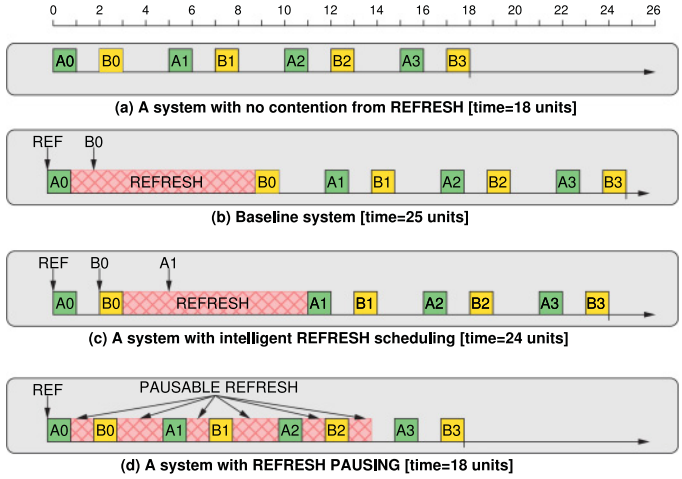

Fig. 1. Latency overheads of doing refresh are significant. Intelligent scheduling of refresh helps reduce this latency overhead but is not sufficient. Refresh Pausing can avoid the latency penalty of refresh operations.

however, it may be able to reduce the penalty. Figure 1(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 noninterruptible 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. Figure 1(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 article 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 and then precharged. After that, the next row is refreshed. We can potentially *Pause* an ongoing refresh operation to service a pending read request. We keep track of the address of rows 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. Thus, the number of rows in a refresh group dictates the number of Refresh Pause Points (RPPs). For a DRAM device containing 8 (or 16) rows, we have 7 (or 15) RPPs. Thus, although 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 and

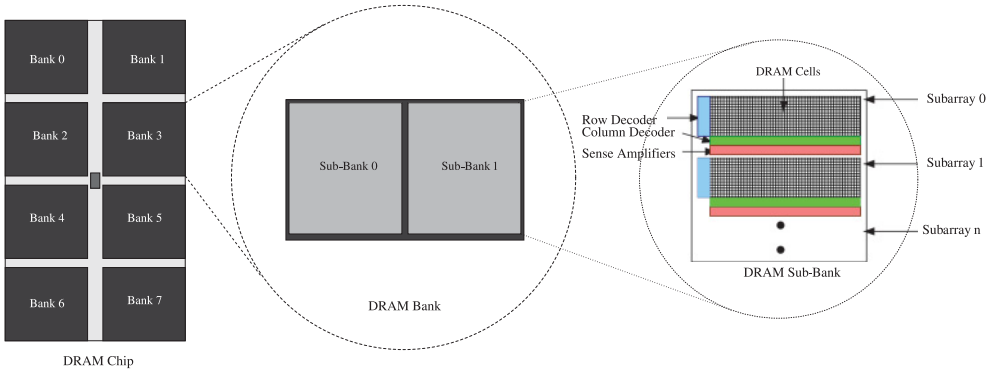

Fig. 2. Each DRAM chip is composed of several banks. Each bank can be composed of sub-banks that allow for concurrent accesses. Each sub-bank is internally organized into subarrays that have mats of DRAM cells, row and column decoders, and sense amplifiers.

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 article is organized as follows: Section 2 provides background and motivation, Section 3 presents the design of Refresh Pausing, Section 4 reviews the methodology, and Section 5 provides our results and analysis. We compare Refresh Pausing with refresh scheduling in Section 6, Refresh Pausing in power-constrained systems in Section 7, and discuss other related work in Section 8.

2. BACKGROUND AND MOTIVATION

2.1. DRAM Refresh: Background and Terminology

Figure 2 shows the internal organization of a DRAM chip. A DRAM chip is composed of 8 to 16 banks. DRAM banks are organized into subarrays and sub-banks [Thoziyoor et al. 2008a, 2008b; Huang et al. 2001]. A subarray contains DRAM cell arrays (rows), row decoder, column decoder, and sense amplifiers. DRAM cells maintain data integrity using refresh operations, a process whereby the data is rewritten to the cell periodically using sense amplifiers. Although DRAM cells have varying retention time [Kim et al. 2011], the JEDEC standards specify a minimum of 64ms retention time (32ms for high temperature), which means that all DRAM rows must be refreshed within this small time period. Let's call this time period *DRAM Retention Time*. Initially, DRAM systems 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 system increased to a few (tens of) thousands, 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 [JEDEC Committee JC-42.3 2012] provided a *Distributed Refresh* mode, whereby a fraction of memory is refreshed at frequent intervals.

Refreshes are performed with some internal parallelism using subarrays. Banks can also be divided into sub-banks. Sub-banking allows multiple portions of the bank to be addressed consecutively. The number of sub-banks is usually small (2 to 4) due to the overhead required for implementing circuits that allow for parallel sub-bank accesses within the bank. Figure 2 shows the internal details of a bank and the organization of subarrays and sub-banks within a bank.

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

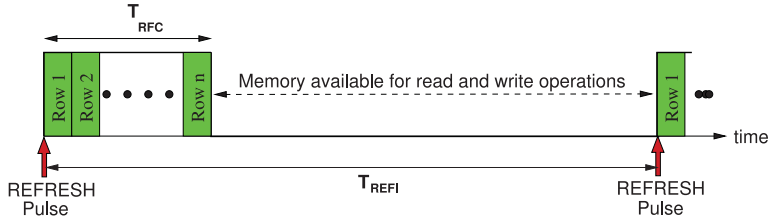

Fig. 3. Timing parameters of distributed DRAM Refresh, the cluster of rows may be split across subarrays or sub-banks and can be operated with some parallelism.

Table I. T_{RC} and T_{RFC} for Different DRAM Densities [JEDEC Committee JC-42.3 2012]

Memory Density	T_{RC}	T_{RFC}	RDC
1Gb	39ns	110ns	2.8%
2Gb	39ns	160ns	5.1%
4Gb	39ns	260ns	6.7%
8Gb	39ns	350ns	9.0%

call this group 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 7.8μs (3.9μs 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 that 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 that 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 a 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}).

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. T_{RFC} may be greater or less than the number of rows in a refresh bundle multiplied by the T_{RC} [Micron 2009; Jacob et al. 2008] and is accounted to the internal parallelism of the refresh operation. Figure 3 illustrates the 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 of time that the memory is doing refresh. RDC can be computed as the ratio of T_{RFC} to T_{REFI}. Ideally, we want a 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 I. The row cycle time has largely remained unchanged; however, the T_{RFC} has increased

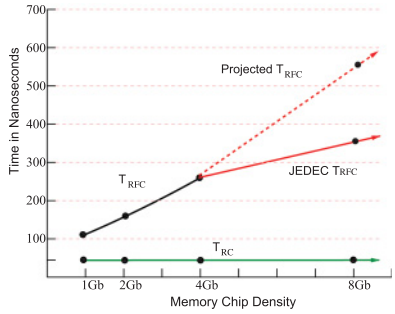

Fig. 4. Variation and projection for T_{RFC} for various densities [JEDEC Committee JC-42.3 2012].

considerably. For high-temperature server operation, T_{REFI} is 3,900ns, so for 8Gb[1] memories available currently [Micron 2010], RDC is 350ns/3900ns = 9%.

Manufacturers implementing these DRAMs also project their timings accordingly for various density of memories. Figure 4 portrays the deviation of the timings of T_{RC} with the value of T_{RFC} for Micron™ DDR3-800 for 1Gb, 2Gb, and 4Gb chips. The projections as per these trends as well as JEDEC predictions are plotted for an 8Gb DRAM chip. One key point to note is that despite the increase in T_{RFC}, the value of T_{RC} remains constant, as it does not depend on the memory density and rather depends on address decoding and precharging latency, making this a key advantage for Refresh Pausing.

Although 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 at 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 behavior increases and a 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 row buffer smaller to reduce overfetch [Udipi et al. 2010]. 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 [Stuecheli et al. 2010]. This value corresponds to a high-temperature operation, typical for dense server environments [Stuecheli et al. 2010]. It also reflects future technologies where variability in devices may dictate a shorter refresh interval even at room temperature.

[1]For meeting the JEDEC specifications of T_{RFC} of 350ns for 8Gb chips, some designs have adopted TwinDie technology [Micron 2010], 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.

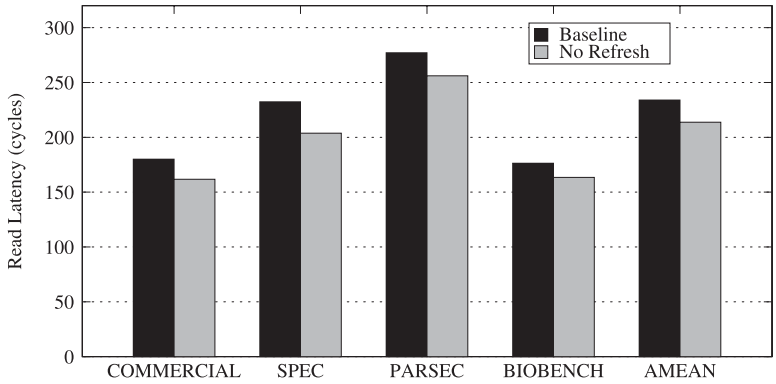

Fig. 5. Impact of refresh on read latency.

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

2.3. Latency Impact of Refresh

Our baseline assumes that 8Gb chips with T_{RFC} equals to 350ns and T_{REFI} of 3,900ns, so the memory system spends 9% of the time doing refresh operations. Figure 5 shows the average latency of baseline as well as if all refresh operations are removed (No Refresh). (Detailed methodology is described later in Section 4, in which Table IV gives the classification of benchmarks.) The bar labeled *AMEAN* represents arithmetic mean over all 18 workloads. The average latency for reads in the baseline system is 234 processor cycles. However, if the contention from refresh operations is removed, then the average read latency would get reduced to 215 cycles.[2]

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 standards provide the ability to postpone refresh operations for up to eight T_{REFI} cycles. The work most related to our work was on scheduling refresh operations called *Elastic Refresh* (ER) [Stuecheli et al. 2010]. 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 ER, are unlikely to give significant benefit, 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.

[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 (thus, 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 $0.09 \times 175\text{ns} = 15.75\text{ns}$, 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.

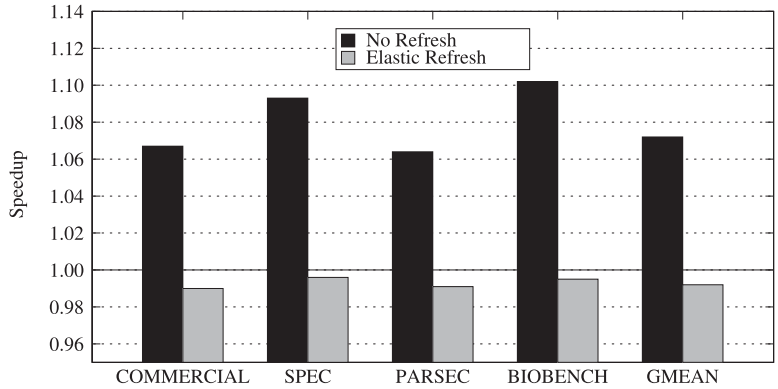

Fig. 6. Potential speedup from removing refresh is significant. However, Elastic Refresh degrades performance (see detailed study in Section 6).

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

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 eight pending refreshes. Compared to this simple refresh scheduling scheme, ER ends up degrading performance. Section 6 will analyze the inefficacy of refresh scheduling algorithms (including ER) in detail.

We need a practical solution that can reduce the latency impact of refreshes. The 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.

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 gets serviced. This section describes the concept, implementation, and implications of Refresh Pausing.

3.1. Refresh Pausing: Concept

Although 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 7(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}, then precharges the row. This

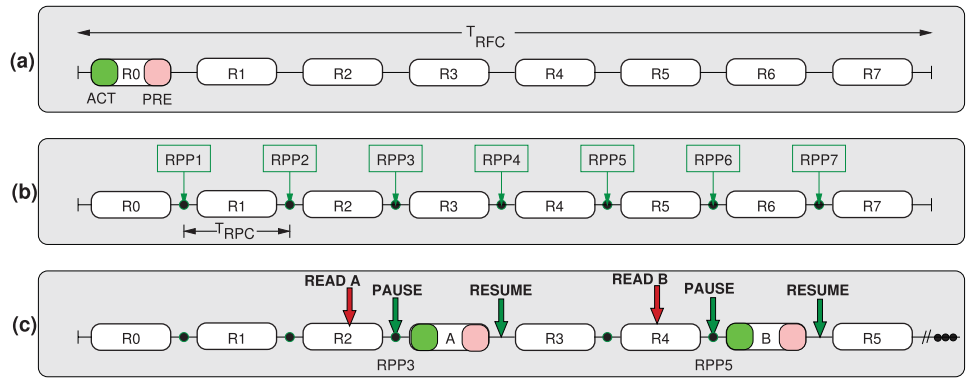

Fig. 7. Enabling Refresh Pausing in DRAM systems. (a) Refresh operation in traditional DRAM memories. (b) Identifying potential pause points (RPPs) for Refresh Pausing. (c) How Refresh Pausing can quickly service pending requests (for simplicity, in this figure we assume that refreshes are performed sequentially on a row-by-row basis and do not account for some internal parallelism).

cycling takes a time equal to T_{RC}. Subsequently, 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 7(b). Each such potential point of pausing is an 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. DRAM vendors do not typically provide details about how the recovery time is calculated or provisioned. In our work, we assume that the recovery time is spread out over all rows. Therefore, we divide the time T_{RFC} into eight (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 7(c) shows the working of memory system with Refresh Pausing. Let's say that a read request for Row A arrives while Row R2 is being refreshed. The memory controller signals the device to pause, and 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, Refresh 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 ongoing refresh operation at the next RPP, if the pause signal is received. To RESUME

[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.

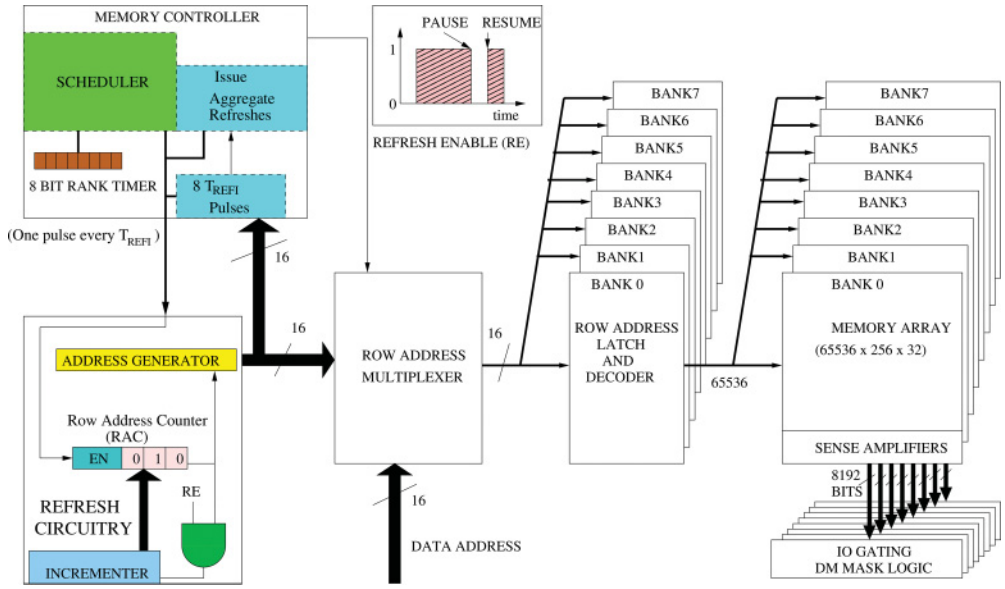

Fig. 8. Implementing Refresh Pausing with (1) reusing the REFRESH ENABLE signal to indicate REFRESH, PAUSE, and RESUME, (2) an AND gate in DRAM refresh circuit to check for RE during refresh, and (3) a one-byte timer in the memory controller.

from the check-pointed state, a paused refresh gets resumed. Figure 8 shows the system that implements Refresh Pausing. Our implementation is geared toward keeping the hardware modifications to minimum.

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 deassertion 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 the RE signal gets asserted, the refresh circuitry probes a register called *Row Address Counter* (RAC) that points to the next row to be refreshed. During every iteration of refresh, 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 until 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 (REFQ) as well as to schedule a PAUSE. A PAUSE must be sent 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 the RE signal during ongoing refresh operation. The DRAM refresh circuitry needs one AND gate. In addition, 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 that 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 [Chatterjee et al. 2012] from the recently conducted Memory Scheduling Championship (MSC) [MSC 2012]. USIMM models the DRAM system in detail, enforcing the various timing constraints. We modified USIMM to conduct a detailed study for refresh operations. We added an 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 II. 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 WRQ. 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 (eight pending requests). Refresh operation takes T_{RFC} cycles to complete. In all scheduling policies, the number of refreshes can be accumulated

Table II. 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	64 byte
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 III. 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	1,120
T_{REFI}	3,120	12,480

Table IV. Workload Characteristics (Suite from MSC [MSC 2012])

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

up to eight without breaking the rules specified by JEDEC standards. We use 8Gb devices for our study, the timing parameters for which are shown in Table III. T_{RFC} is 280 DRAM cycles (350ns [JEDEC Committee JC-42.3 2012]). We use a T_{REFI} of 3.9μs, which translates to 3,120 DRAM cycles.

4.2. Workloads

We use the workloads from the recently held MSC [MSC 2012], as it contains a wide variety of applications. Table IV shows key characteristics of our workloads. The MSC suite contains five commercial applications, *comm*1 to *comm*5. There are nine benchmarks from the PARSEC suite, including two multithread versions of applications fluid

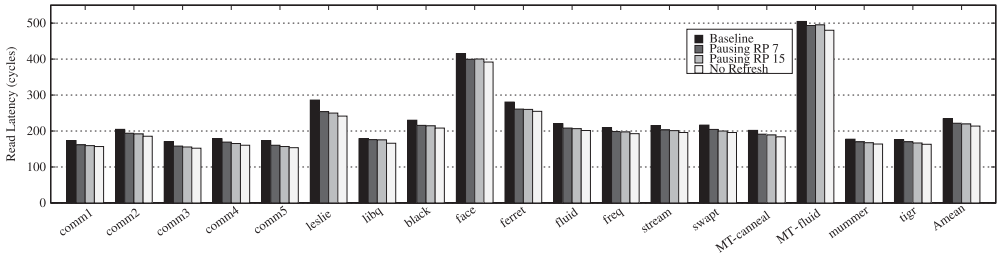

Fig. 9. Average read latency for different systems.

and canneal (marked *MT-fluid* and *MT-canneal*). In addition, 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).

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 (since they use twindie technology with two independent dies, and this number is for a single die), so there can potentially be 7 RPPs. 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 RPPs. 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*. The "Baseline Refresh" uses distributed refresh, segmented in time, and each distributed refresh pulse operates on a channel as soon as it becomes available.

5.1. Impact on Read Latency

The read latency of a system is increased by contention due to refresh operations. Figure 9 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 18 workloads. For the baseline system, the average read latency is 234 cycles; 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 10 shows the speedup from Refresh Pausing (RP-7 and RP-15) and No Refresh. The bar labeled *Gmean* denotes the geometric mean over all 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

[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.

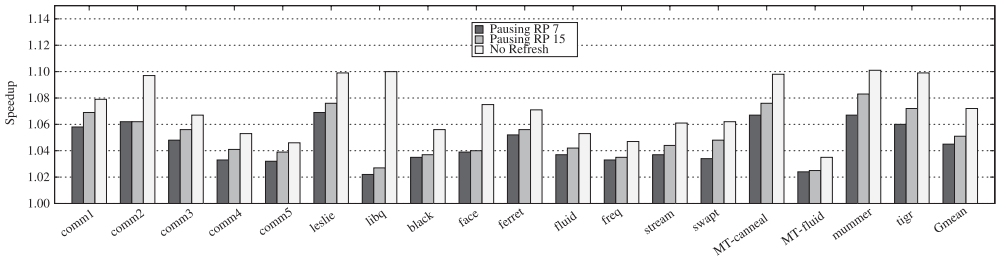

Fig. 10. Performance improvement from Refresh Pausing.

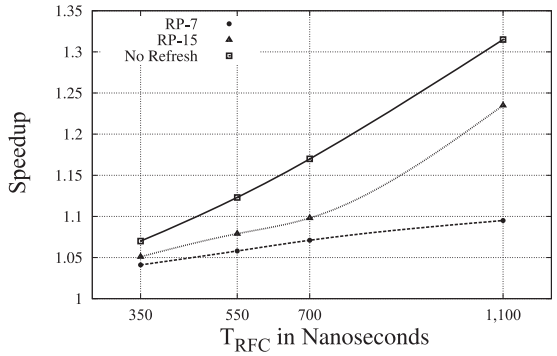

Fig. 11. Scalability of Refresh Pausing with T_{RFC}.

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.

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. Scalability of Refresh Pausing

As memory technologies quest for higher densities and more capacity gets packed into a single chip, the number of rows in the chip is likely to increase. This is expected to increase T_{RFC}. Figure 11 shows the effectiveness of Refresh Pausing as T_{RFC} is varied. The average performance improvement over all workloads is reported. For future 16Gb devices, with similar internal parallelism as the current designs, we assume that they will have a T_{RFC} that is twice of 8Gb devices; hence, we evaluate data points for 700ns and 1,100ns. The data point of 350ns represents the JEDEC specified T_{RFC} of 350ns for 8Gb devices.

If the memory system has a small T_{RFC}, then the potential performance difference from eliminating refresh penalties is small. Refresh Pausing (RP-7 and RP-15) gets most of this potential. Note that maintaining low T_{RFC} is technologically quite hard. As designs move to the future, and T_{RFC} increases, Refresh Pausing becomes even more effective. For example, for a 16Gb memory with T_{RFC} of 1,100ns, there is a potential 32% performance improvement possible with No Refresh, and Refresh Pausing RP-15 gets 24% performance improvement. Although future chips strive toward increasing internal parallelism, Refresh Pausing is robust to increasing device densities with or

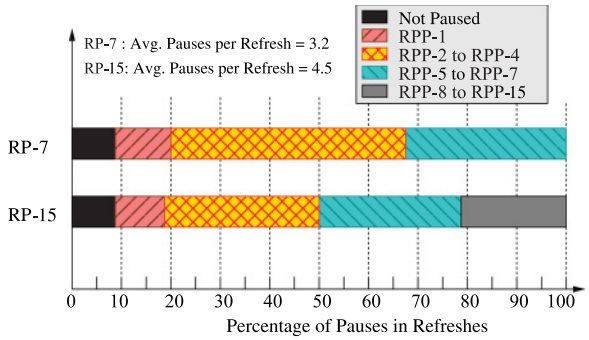

Fig. 12. Frequency of pauses for RP-7 and RP-15 schemes.

without increased parallelism. As device densities increase, so will the number of pause points, which makes Refresh Pausing even more effective for future technologies.

5.4. Frequency of Pausing

With Refresh Pausing, a refresh operation can be paused (and resumed) multiple times. During each pause interval, the memory can schedule many read requests. Once the rank becomes idle, the refresh is resumed and then paused again later if a read arrives. Thus, the breakdown of how many times a refresh is paused serves as a good indicator both for having many pause points and for the inherent burstiness of read accesses. Figure 12 shows such a breakdown. On average, 3.2 pauses and 4.5 pauses happen per refresh in RP-7 and RP-15, respectively. For RP-7, most often the pause point occurs between RPP2 and RPP4. RP-15 shows a drift in the most active pause points, as it has much smaller T_{RPC} than RP-7, so the distribution gets spread out. Figure 12 confirms that Refresh Pausing will help lower contention, as less than 8% refreshes have no pausing, and 92% of the refreshes do encounter a pending read.

5.5. Quantifying Forced Refresh

Table V shows that forced refreshes constitute a maximum of up to 0.02% of the total number of refreshes in the baseline scheme for 4 channels. RP-7 and RP-15 schemes result in an increase in the percentage of forced refreshes of up to 0.06% and 0.08%, respectively. Increasing the read refresh contention by reducing the number of memory channels increases the number of forced refreshes. Table VI shows that for a single channel configuration, forced refreshes constitute a maximum of up to 0.3%. RP-7 and RP-15 schemes result in an increase in the percentage of forced refreshes of up to 3.6% and 8.1%, respectively. Since *Refresh Pausing* allows the RDQ to be empty after the pause points are reached, delaying portions of refreshes. This increases the probability of having refreshes spilling over into the Forced Refresh mode. This behavior is consistent for RP-15 and RP-7, with RP-15 showing a higher percentage of forced refreshes.

5.6. 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 refreshes cannot be paused, there is little scope for Refresh Pausing to improve performance. To analyze Refresh Pausing for highly utilized systems, we reconfigured our baseline to have 1 channel instead of 4 channels.

Table V. Percentage of Refreshes Issued in Forced Mode for a 4-Channel Configuration

Benchmark	Baseline	RP-7	RP-15
black	1.821	1.853	1.908
face	0.009	0.358	2.164
ferret	0.011	1.109	1.176
fluid	0.009	0.009	0.009
freq	0.173	0.273	0.218
stream	0.961	0.969	0.986
swapt	1.583	1.435	1.685
MT-canneal	0.005	0.005	0.006
MT-fluid	0.015	0.475	0.583
mummer	0.005	0.005	0.005
tigr	0.004	0.005	0.005
comm1	0.009	0.011	0.009
comm2	0.007	0.009	0.007
comm3	0.011	0.012	0.011
comm4	0.013	0.014	0.105
comm5	0.016	0.017	0.016
leslie	0.006	0.007	0.007
libq	0.005	1.194	4.325
Gmean	0.023	0.065	0.083

Table VI. Percentage of Refreshes Issued in Forced Mode for a 1-Channel Configuration

Benchmark	Baseline	RP-7	RP-15
black	5.177	6.167	7.699
face	16.359	21.505	32.041
ferret	1.287	1.275	1.427
fluid	0.013	0.614	1.515
freq	1.439	2.739	5.166
stream	2.5	4.103	4.398
swapt	3.632	6.852	6.717
MT-canneal	0.002	57.051	76.957
MT-fluid	0.083	6.513	3.222
mummer	0.002	41.188	78.59
tigr	0.099	74.534	91.479
comm1	0.004	0.45	9.568
comm2	0.098	0.176	4.976
comm3	0.011	0.025	0.477
comm4	0.32	0.684	2.087
comm5	0.944	0.322	1.097
leslie	12.88	33.481	35.365
libq	51.813	72.751	86.448
Gmean	0.307	3.589	8.111

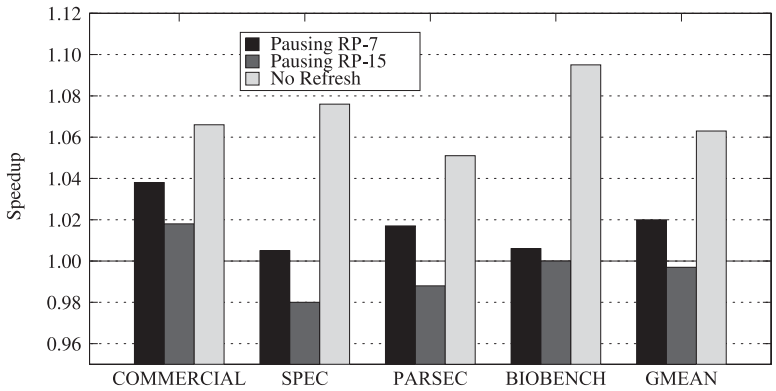

Fig. 13. Performance impact of Refresh Pausing in a highly utilized memory system (number of channels in baseline reduced from 4 to 1).

This increases memory utilization greatly and hence reduces the idle periods available for doing refresh operations.

Figure 13 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. Figure 13 shows performance degradation of SPEC and PARSEC benchmarks due to frequent forced refreshes. In the limit, if the memory is 100% utilized,

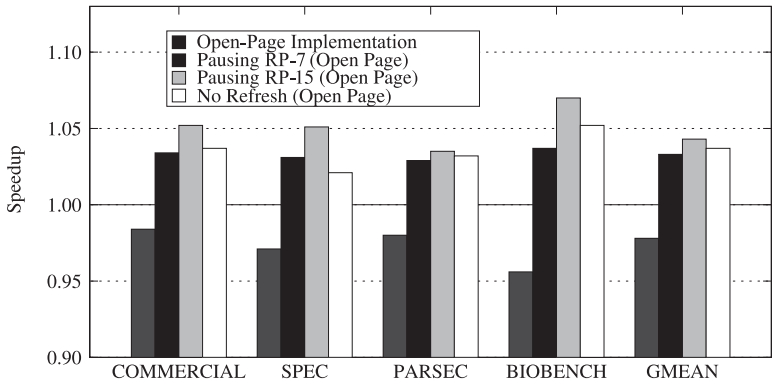

Fig. 14. Speedup from Refresh Pausing on systems that employ open-page policy compared to close-page baseline.

Table VII. 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%

then all refreshes will be done as forced refreshes, leaving no scope for performance improvement with Refresh Pausing.

5.7. Sensitivity to Page Closure Policy

A row buffer in a DRAM memory system is also called a *DRAM Page*. Closing a page involves precharging the row. Two page management policies exist: close page and open page. The close-page policy precharges the row soon after the transaction to the row is complete. The open-page policy keeps the row active for subsequent transactions and precharges only in case of a row conflict in a transaction. 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 14 shows the speedup from Refresh Pausing (RP-7 and RP-15) and No Refresh, all implemented on a system that 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, and 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 MSC [MSC 2012], which 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.

5.8. Sensitivity to Temperature

Table VII 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

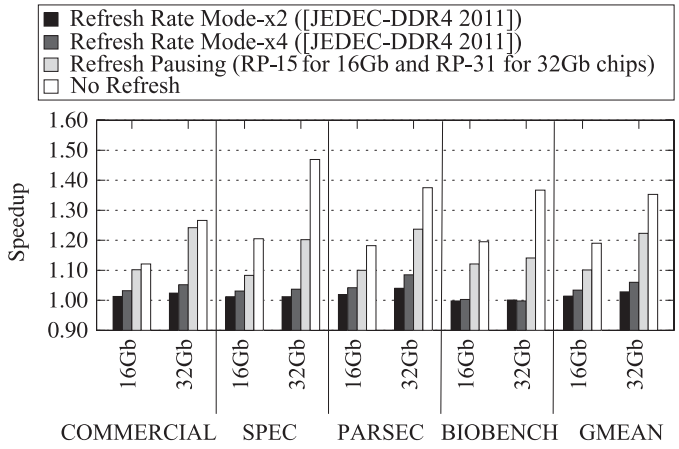

Fig. 15. Effectiveness of Refresh Pausing at high density.

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.

5.9. 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 [JEDEC-DDR4 2011]. One of the refresh proposals being considered for DDR4 is the fine-grained refresh scheme, which lowers the refresh interval T_{REFI} and T_{RFC} both by a factor of either 2x or 4x, called *Refresh Rate Mode-x2* (RRMx2) or *Refresh Rate Mode-x4* (RRMx4) [JEDEC-DDR4 2011]. Unfortunately, these proposals are not as effective at tolerating refresh latency as Refresh Pausing. Furthermore, as memory capacity increases, the granularity of these modes will need to be revised to tolerate longer refresh.

Figure 15 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. In addition, 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.

6. REFRESH PAUSING VERSUS 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, ER [Stuecheli et al. 2010], 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 6 showed that ER degrades performance compared by 1.3% on average compared to our baseline refresh scheduling policy. In this section,

we analyze ER, the requirements for ER 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 until either the rank is idle or the refresh deadline approaches. The critical idea behind 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 in a workload, these idle periods must be spread throughout the program execution. More specifically, the idle period must be within $8 \cdot T_{RFC}$ of the refresh request time. Third, the hardware predictor should predict the start of the long idle period correctly.

6.3. Influence of Forced Refreshes on Effectiveness of Elastic Refresh

Refresh Pausing reduces the average waiting time and thus improves the IPC. The *Read Priority* waits for the RDQs to be empty and then issues the refresh, whereas *Elastic Refresh* goes a step further and issues refreshes only after an idle wait period. These two scheduling policies do not improve the IPC, because read latency of these two are not reduced dramatically. *Expected Delay* is a product of two terms: $Prob_{collision}$ and *Average Waiting Time*. Both scheduling policies, Read Priority and Elastic Refresh, try to eliminate the read collision with refresh, but remaining the same long average waiting time, which do not have positive impact on the read latency. *Refresh Pausing* provides the advantages that the average waiting time is much smaller than these two schemes and is able to reduce the read latency significantly.

We further extend the equation to a more generalized form, as shown in Equation (1). A *ForcedCollision* refers to the collision with a refresh that is issued when the number of pending refreshes exceeds eight, allowed by JEDEC standards. The other type of collision is called *NormalCollision*. When the DRAM is in the *ForcedRefresh* mode, the read latency is penalized heavily since it takes several T_{RFC} cycles to drain out the pending refreshes.

$$\begin{aligned} ExpectedDelay = &(Prob_{NormalCollision} \times T_{NormalWaiting}) \\ &+ (Prob_{ForcedCollision} \times T_{ForcedWaiting}) \end{aligned} \tag{1}$$

As we provide the detailed probability of collision and average waiting time of each policy and configuration in Table VIII, it is clear that not only $Prob_{NormalCollision}$ but also $Prob_{ForcedCollision}$ should be taken into account to reduce the read latency effectively. *Elastic Refresh* tried to reduce $Prob_{NormalCollision}$ to reach smaller read latency but failed, because the average waiting time is dominating and the reduction of probability do not contribute enough to reduce the read latency.

Table VIII. Probability of Collision and Average Waiting Time for Baseline and Elastic Refresh for Different Configurations

Configuration	Policies	P_{Normal}	T_{Normal}	P_{Forced}	T_{Forced}	Expected Delay (Cycles)
1 channel 8Gb	Baseline	0.045	560	0.002	1,120	28
	Elastic Refresh	0.043	560	0.003	1,120	28
4 channel 8Gb	Baseline	0.034	560	0.000	1,120	19
	Elastic Refresh	0.039	560	0.000	1,120	22
1 channel 16Gb	Baseline	0.061	1,120	0.003	2,240	75
	Elastic Refresh	0.058	1,120	0.005	2,240	76
4 channel 16Gb	Baseline	0.043	1,120	0.000	2,240	48
	Elastic Refresh	0.050	1,120	0.000	2,240	56

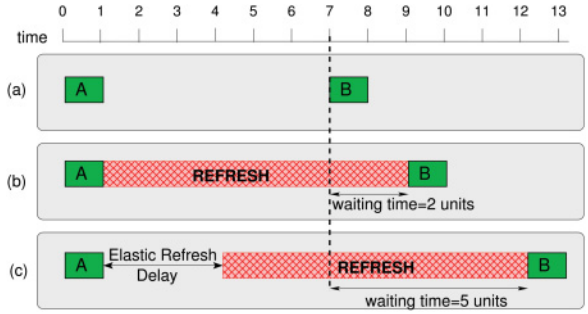

Fig. 16. Waiting time for a system with (a) No Refresh, (b) baseline scheduling, and (c) Elastic Refresh.

6.4. 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 16 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 would 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.

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 an 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 17 shows the Cumulative Density Function (CDF) of idle periods in cycles. The maximum value in the x axis is 1,120 processor cycles at 3.2GHz (280 DRAM cycles at 800MHz), same as T_{RFC} (350ns). The vertical dotted line 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. In addition, about 20% of idle periods are between average idle period and T_{RFC}, which represents a loss scenario for ER. Thus, for our workloads, the loss scenarios with ER are quite common, hence the performance loss.

6.5. Mitigating Loss of Elastic Refresh 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

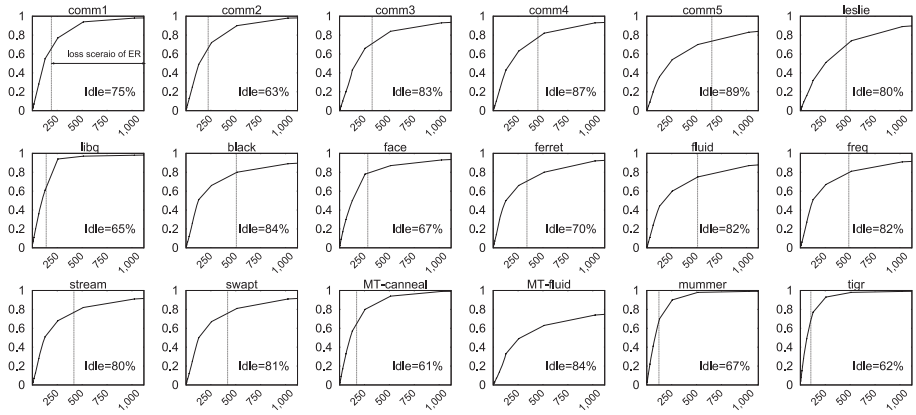

Fig. 17. Cumulative density function of idle periods of a rank in processor cycles. The percentage of execution time that the rank is idle is specified within the figure. The vertical line represents the average length of an idle period. The maximum value of the x axis is 1,120 processor cycles at 3.2 GHz (280 DRAM cycles at 800MHz), similar to T_{RFC}. The average length of idle period for MT-Fluid is 1,330.

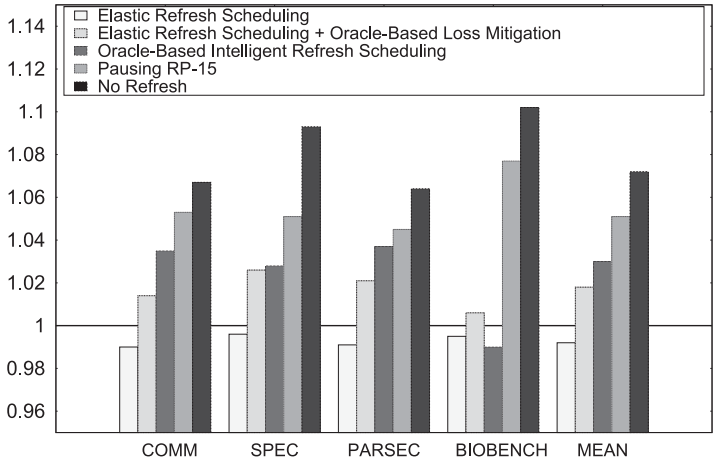

Fig. 18. Speedup for Elastic Refresh and Elastic Refresh with Oracle-Based Loss Mitigation.

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 18 shows the speedup of ER and ER with OBLM. ER 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%.

6.6. 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 regard 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 18 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. Although this is smaller than what we get with Refresh Pausing, it still indicates a good opportunity for future research to develop effective refresh scheduling algorithms.

6.7. Scaling to Future Technologies

As devices move to higher densities, T_{RFC} will become longer, which means that 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. REFRESH PAUSING IN POWER-CONSTRAINED SYSTEMS

Refresh operations consume power, and the number of rows refreshed has increased over time. Future low-power DRAM systems may have power constraints for refresh operations involving a large number of rows. During a refresh operation, sense amplifiers activate and precharge bit lines, consuming power and leading to current spikes. This limits the number of banks that can be activated in a power-constrained system. To mitigate this, we assume that future low-power systems follow the JEDEC-imposed restriction that limits activations of up to 4 banks in a time window T_{FAW} (32 DRAM cycles for 8KB row buffer size). DRAM chips have to wait for T_{FAW} period to end before resuming the activation of additional banks.

Refreshes activate up to 8 banks in the T_{RFC} time period for 8Gb chips. The power-constrained system will have to ensure that every bank can refresh multiple rows within T_{RFC} while tolerating the overhead of current draw. For a 8Gb chip, 8 rows per bank may need to refreshed. These 8 rows can lie in a single subarray in a single sub-bank or can be distributed across several subarrays and lie in separate sub-banks within the same bank. We explore ways in which refresh for such system could potentially be performed while adhering to the current consumption limit for a DRAM chip. We evaluate the applicability of Refresh Pausing in such power-constrained system. We discuss two cases based on concurrent sub-bank activations during the refresh period for each bank. The first method, called *Two Sub-bank Concurrent Refresh* (TSCR), activates 2 sub-banks and multiple subarrays in each sub-bank during every refresh pulse. The second one, called *Four Sub-bank Concurrent Refresh* (FSCR), activates up to 4 sub-banks and multiple subarrays in each sub-bank during every refresh pulse. Figure 19 shows the implementations of these two cases.

Refresh Pausing is still applicable for both TSCR and FSCR, with TSDR allowing us to have 3 pause points (RP-3) and FSCR having 1 pause point (RP-1) for current 8Gb chips. For TSCR, when a read request comes in before $2 \times T_{FAW}$, then refresh can be paused at RPP-1; however, if the read request comes in between T_{FAW} and RPP1, the refresh can only be paused at the next pause point. FSCR allows more time to pause a refresh. Figure 20 shows the potential impact of refresh pausing on speedup for TSCR and FSCR for current 8Gb chips when normalized to a baseline refresh scheme. FSCR shows a speedup of 1.1% due to Refresh Pausing. This increases to 2.7% for a TSCR scheme. As the chip densities increase, the number of pause points for TSCR and FSCR will also increase. DRAM chips having higher density

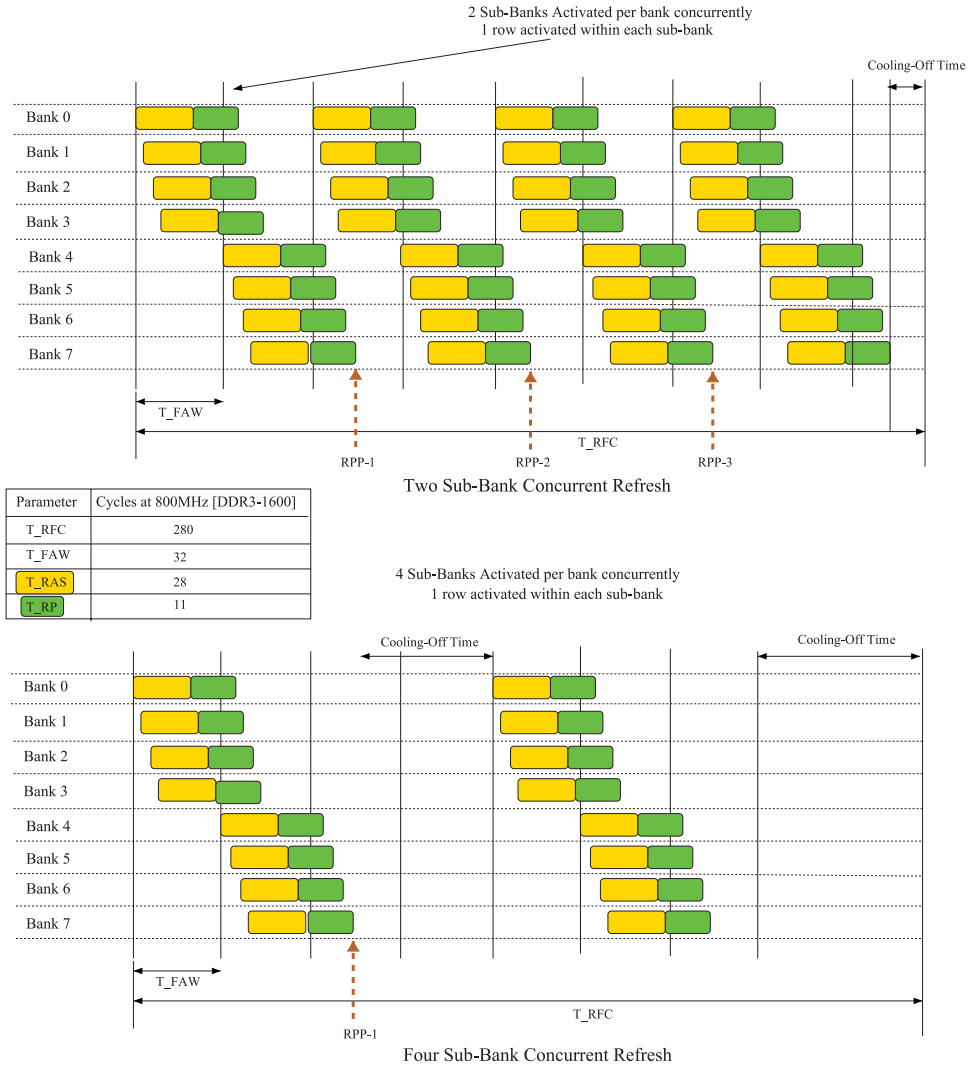

Fig. 19. The two sub-bank concurrent refresh scheme allows for up to 3 possible pause points. The four sub-bank concurrent refresh scheme allows for a single pause point.

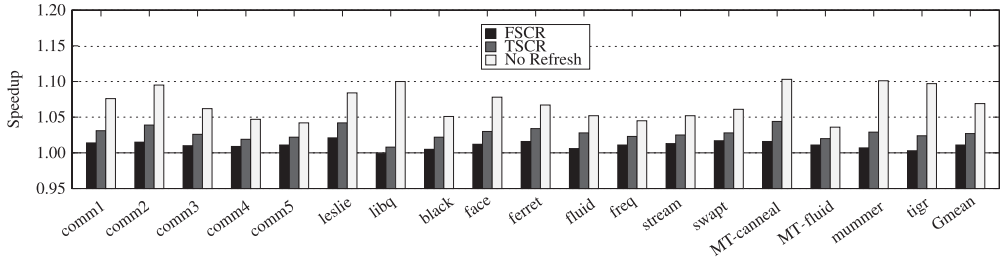

Fig. 20. Speedup from Refresh Pausing for two and four sub-bank concurrent refresh schemes normalized to baseline refresh.

will use higher number of sub-banks or form bank groups that will allow concurrent refreshing.

8. OTHER RELATED WORK

Having more banks and thus doing more refreshes in parallel can reduce the latency of refresh operations. 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 [Venkatesan et al. 2006] or use multirate refresh where high-retention pages (rows) are refreshed infrequently [Kim and Papaefthymiou 2001; Liu et al. 2012; Venkatesan et al. 2006]. These approaches rely on 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 the book on Memory System (Anecdote IV) by Jacon et al. [2008]: "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." As 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 [Ghosh and Lee 2007], when the data is not critical [Isen and John 2009; Liu et al. 2011], or when the system provisions ECC to tolerate data errors [Wilkerson et al. 2010].

Refresh operations can also be done at a finer granularity, such as on a per-bank basis instead of a per-rank basis [Cuppu et al. 1999]. 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.

9. SUMMARY

DRAM is one of the most *forgetful* memory technologies, 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 ongoing 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 a 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 that Refresh Pausing can provide 4.5% to 5.1% performance improvement. Implementing Refresh Pausing entails negligible changes to the DRAM circuitry (one AND gate), reusing existing signal, and a one-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.

REFERENCES

Niladrish Chatterjee, Rajeev Balasubramonian, Manjunath Shevgoor, Seth H. Pugsley, Aniruddha N. Udipi, Ali Shafiee, Kshitij Sudan, Manu Awasthi, and Zeshan Chishti. 2012. *USIMM: The Utah SImulated Memory Module*. Technical Report. University of Utah. UUCS-12-002.

Vinodh Cuppu, Bruce Jacob, Brian Davis, and Trevor Mudge. 1999. A performance comparison of contemporary DRAM architectures. In *ISCA-26*.

Mrinmoy Ghosh and Hsien-Hsin S. Lee. 2007. Smart refresh: An enhanced memory controller design for reducing energy in conventional and 3D die-stacked DRAMs. In *MICRO-40*.

Michael Huang, Jose Renau, Seung-Moon Yoo, and Josep Torrellas. 2001. Energy/Performance design of memory hierarchies for processor-in-memory chips. In *Revised Papers from the 2nd International Workshop on Intelligent Memory Systems (IMS'00)*. Springer-Verlag, London, UK, 152–159. http://dl.acm.org/citation.cfm?id=648002.743089

Ciji Isen and Lizy John. 2009. ESKIMO: Energy savings using Semantic Knowledge of Inconsequential Memory Occupancy for DRAM subsystem. In *MICRO-42*.

Bruce L. Jacob, Spencer W. Ng, and David T. Wang. 2008. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann.

JEDEC Committee JC-42.3 2012. *JESD79-3F*. JEDEC Committee JC-42.3, Arlington, VA.

JEDEC-DDR4. 2011. JS Choi DDR4 Mini Workshop. http://jedec.org/sites/default/files/JS_Choi_DDR4_miniWorkshop.pdf.

Heesang Kim, Byoungchan Oh, Younghwan Son, Kyungdo Kim, Seon-Yong Cha, Jae-Goan Jeong, Sung-Joo Hong, and Hyungcheol Shin. 2011. Characterization of the variable retention time in dynamic random access memory. *IEEE Transactions on Electron Devices* 58, 9, 2952–2958. DOI: http://dx.doi.org/10.1109/TED.2011.2160066

Joohee Kim and M. C. Papaefthymiou. 2001. Block-based multi-period refresh for energy efficient dynamic memory. In *Proceedings of the 14th Annual IEEE International ASIC/SOC Conference.* 193–197. DOI: http://dx.doi.org/10.1109/ASIC.2001.954696

Jamie Liu, Ben Jaiyen, Richard Veras, and Onur Mutlu. 2012. RAIDR: Retention-Aware Intelligent DRAM Refresh. In *ISCA*. 1–12.

Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Ben Zorn. 2011. Flikker: Saving DRAM refresh-power through critical data partitioning. In *ASPLOS-XVI*.

Micron. 2009. *TN-47-16 Designing for High-Density DDR2 Memory*. Micron.

Micron. 2010. *MT41J512M4:8Gb QuadDie DDR3 SDRAM Rev. A 03/11*. Micron.

MSC. 2012. Memory Scheduling Championship (MSC). http://www.cs.utah.edu/~rajeev/jwac12/.

Prashant Nair, Chia-Chen Chou, and Moinuddin K. Qureshi. 2013. A case for Refresh Pausing in DRAM memory systems. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA'13)*. 627–638. DOI: http://dx.doi.org/10.1109/HPCA.2013.6522355

Jeffrey Stuecheli, Dimitris Kaseridis, Hillery C. Hunter, and Lizy K. John. 2010. Elastic refresh: Techniques to mitigate refresh penalties in high density memory. In *MICRO-43*.

Shyamkumar Thoziyoor, Jung Ho Ahn, Matteo Monchiero, Jay B. Brockman, and Norman P. Jouppi. 2008a. A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA'08)*. IEEE Computer Society, Washington, DC, 51–62. DOI: http://dx.doi.org/10.1109/ISCA.2008.16

Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman P. Jouppi. 2008b. *Cacti 5.1*. Technical Report. HP Laboratories, Palo Alto. HPL-2008-20.

Aniruddha N. Udipi, Naveen Muralimanohar, Niladrish Chatterjee, Rajeev Balasubramonian, Al Davis, and Norman J. Jouppi. 2010. Rethinking DRAM design and organization for energy-constrained multi-cores. In *ISCA-37*.

Ravi K. Venkatesan, Stephen Herr, and Eric Rotenberg. 2006. Retention-Aware Placement in DRAM (RAPID): Software methods for quasi-non-volatile DRAM. In *HPCA-12*.

Chris Wilkerson, Alaa R. Alameldeen, Zeshan Chishti, Wei Wu, Dinesh Somasekhar, and Shih-lien Lu. 2010. Reducing cache power with low-cost, multi-bit error-correcting codes. In *ISCA-37*.

Received June 2013; revised October 2013; accepted November 2013