Nonblocking Memory Refresh

Kate Vy H Nguyen

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science and Applications

Xun Jian, Chair

Godmar V. Back

Ali R. Butt

May 21, 2018

Blacksburg, Virginia

Keywords: Memory Systems, Dependable Architectures, Security

Nonblocking Memory Refresh

Kate Vy H Nguyen

(ABSTRACT)

Since its inception half a century ago, DRAM has required dynamic/active refresh operations that block read requests and decrease performance. We propose refreshing DRAM in the background without stalling read accesses to refreshing memory blocks, similar to the static/background refresh in SRAM. Our proposed Nonblocking Refresh works by refreshing a portion of the data in a memory block at a time and uses redundant data, such as Reed-Solomon codes, in the block to compute the block's refreshing/unreadable data to satisfy read requests. For proof of concept, we apply Nonblocking Refresh to server memory systems, where every memory block already contains redundant data to provide hardware failure protection. In this context, Nonblocking Refresh can utilize server memory system's existing per-block redundant data in the common-case when there are no hardware faults to correct, without requiring any dedicated redundant data of its own. Our evaluations show that on average across five server memory systems with different redundancy and failure protection strengths, Nonblocking Refresh improves performance by 16.2% and 30.3% for 16gb and 32gb DRAM chips, respectively.

Nonblocking Memory Refresh

Kate Vy H Nguyen

(GENERAL AUDIENCE ABSTRACT)

Main memory is an essential component of computers, which stores data being actively used. The dominant type of computer main memory is Dynamic Random Access Memory (DRAM). DRAM is divided into thousands of memory cells. Each cell stores a single bit of data as a charge on a capacitor. Charges may leak over time, causing the data stored to be lost. To maintain the data stored in memory, DRAM must periodically restore charges held by memory cells through an operation known as memory refresh. Refresh operations decrease system performance because they stall read requests to refreshing memory blocks. A memory block refers to the unit of data transferred per memory request. Conventional memory systems refresh all the data within the block at a time, therefore the entire memory block is inaccessible while it is being refreshed. Our proposed Nonblocking Refresh reduces the amount of data in a memory block which is inaccessible due to refresh by refreshing only a portion the memory block at a time. To satisfy read requests, the block's refreshing/inaccessible data is computed using redundant data. Nonblocking Refresh improves DRAM performance by refreshing DRAM in the background without stalling read accesses to refreshing memory blocks. For proof of concept, we apply Nonblocking Refresh to server memory systems, where every memory block already contains redundant data to provide hardware failure protection. In this context, Nonblocking Refresh can utilize server memory system's existing redundant data to improve performance, without adding additional redundancy overhead. Our evaluations show that on average across five server memory systems with different redundancy and failure protection strengths, Nonblocking Refresh improves performance by 16%-30%.

Acknowledgments

Thank you to my advisor, Dr. Xun Jian, for his guidance and support throughout my journey to obtain my master's degree. Dr. Jian navigated me through my transition from undergraduate to graduate school, helping me develop the skills I needed to become a successful graduate student. The knowledge I have gained under his mentorship will carry with me in every aspect of my career. Dr. Jian's dedication and commitment to his work is truly inspiring.

In addition, I would also like to thank the members of my committee, Dr. Godmar Back and Dr. Ali Butt, for their valuable insight and feedback. I had the pleasure of being a student in their classes during my time as an undergraduate. Dr. Butt and Dr. Back teach with a passion that motivated me to do well as a student. It was an honor to have two of my favorite professors serve on my committee.

I am thankful for the support I received from the Cybercorps: Scholarship for Service program, which allowed me to focus on my education without worry. Thank you to my scholarship advisor, Dr. Kira Gantt, for her advice and reassurance during my education.

Thank you to my brothers and my entire extended family for their love and support every step of the way. I would also like to thank Neel Arora, for his unwavering patience and encouragement all the way from the other side of the country. Finally, thank you to my parents, Diep Cao and Hoang Nguyen. Words cannot fully express how much I appreciate everything they have done for me. The sacrifices my parents have made inspire me to work hard and do my best every day. This work would not have been possible without their encouragement and guidance.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

For half a century, Dynamic Random Access Memory (DRAM) has been the dominant computer main memory. Despite its important role, DRAM has an inherent physical characteristic that contributes to its inferior performance compared to its close relative - SRAM (Static RAM). While DRAM and SRAM are both volatile, DRAM requires dynamic/active refresh operations that stall read requests to refreshing data; in comparison, SRAM relies on latch feedback to perform static/background refresh without stalling any read accesses.

Stalled read requests to DRAM's refreshing data slow down system performance. Prior works have looked at how to reduce the performance impact due to memory refresh [8, 13, 25, 32, 35, 39, 40, 41, 42, 43]. Some of them have explored intelligent refresh scheduling to block fewer pending read requests [13, 39, 40]; however, they provide limited effectiveness. As refresh latency increases, many later works have explored how to more aggressively address memory refresh performance overheads by skipping many required memory refresh operations [8, 25, 32, 41, 42] at the cost of reducing memory security and reliability [15, 21, 24, 27, 31]; however, this is inadequate for systems that do not wish to sacrifice security and reliability for performance.

To effectively address increasing refresh latency without resorting to skipping refresh, we propose *Nonblocking Refresh* to refresh DRAM without stalling reads to refreshing memory blocks. A memory block refers to the unit of data transferred per memory request. Nonblocking Refresh works by refreshing only some of the data in a memory block at a time and uses redundant data, such as Reed-Solomon code, to compute the inaccessible data in the refreshing block to complete read requests. Compared to the conventional approach of refreshing all the data in a block at a time, Nonblocking Refresh makes up for refreshing only some of the data in a block at a time by operating more frequently in the background. Nonblocking Refresh transforms DRAM to behave like SRAM at the system-level by enabling DRAM to refresh in the background without stalling read requests to refreshing memory blocks.

For proof of concept, we apply Nonblocking Refresh to server memory systems, which value security and reliability. We observe server memory systems already contain redundant data to provide hardware failure protection via an industry-standard server memory feature commonly known as chipkill-correct, which tolerates from bit errors up to dead memory chips [5, 17, 19]. Because redundant data are budgeted to protect against worst-case hardware failure scenarios, they are often under-utilized when there is minor or no hardware fault. As such, in the context of server memory, we can safely utilize existing under-utilized redundant data to implement Nonblocking Refresh in the common-case, without requiring any dedicated redundant data. Our evaluation shows that across five server memory systems with different failure protection strengths, Nonblocking Refresh improves average performance by 16.2% and 30.3% for 16gb and 32gb DRAM chips, respectively. The performance of memory systems with Nonblocking Refresh is 2.5%, on average, better than systems that only performs 25% of the required refresh.

We make the following contributions in this paper:

- We propose Nonblocking Refresh to avoid stalling accesses to refreshing memory blocks in DRAM.

- We apply Nonblocking Refresh in the context of server memory systems, where existing redundant memory data can be leveraged without increasing storage overhead.

- We find that Nonblocking Refresh improves average performance by 16.2% and 30.3% for server memory systems with 16gb and 32gb DRAM chips, respectively.

Chapter 2

Background

2.1 DRAM Structure

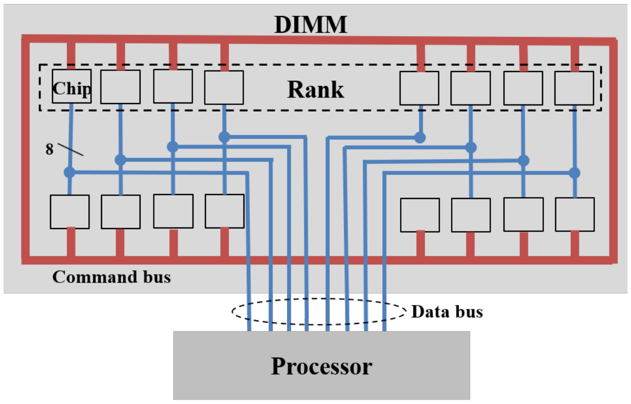

Figure 2.1: Memory system layout

The lowest-level structure in memory is a cell, which contains one bit of data. Each memory chip consists of billions of cells. Chips accessed in lockstep are referred to as a rank. A rank is the smallest unit that can be addressed in memory commands. When accessing memory, all chips in a rank operate in lockstep to transmit a unit of data called a memory block. Each chip in the rank contributes an equal amount of data to a memory block, usually four or eight bytes; memory chips that access four and eight bytes of data per memory request are referred to as x4 and x8 chips, respectively. Multiple ranks form a memory module, which is commonly referred to as a DIMM (dual in-line memory module). One or more DIMMs form a memory channel. Each channel has a data bus and command bus that are shared by all ranks in the channel (see Figure 2.1). The processor's memory controller (MC) manages accesses to each channel by broadcasting commands over each channel's command bus.

2.2 Memory Refresh

A memory cell stores a single bit of data as charge in a capacitor. A cell loses its data if it loses this charge. The charge in a cell may leak or degrade over time; thus, memory refresh is needed to periodically restore the charge held by memory cells. Memory standards dictate that a cell refresh its charge every 64ms [22]. A cell refreshes in lockstep with the other cells in its row. Each chip maintains a counter that determines which rows to refresh.

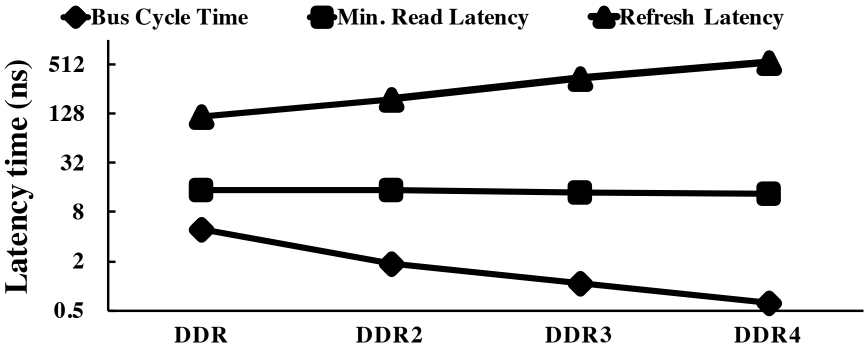

Figure 2.2: Historical trends of memory latencies [18, 36, 37, 38]

To refresh a row, a memory chip reads data from a row into its row buffer and then rewrites the data back to the row, thus restoring the charge. Chips refresh multiple rows per refresh interval. The duration of a single refresh interval is called refresh cycle time ($tRFC$). The MC sends a single refresh command to refresh all chips in a rank simultaneously. The duration between refresh commands for one rank is the refresh interval time ($tREFI$). MC can "pull-in" or issue refresh commands earlier than $tREFI$ to allow scheduling flexibility [3]. MC can pull in up to eight refresh commands to reduce the number of refresh commands required later [3].

Historically, *tRFC* has increased for every new generation of chips, growing 50% between the last two generations (8gb to 16gb) chips [38]. This increase is attributed to growth of chip density because the time for refresh correlates to the number of rows in memory. In contrast, other memory related latencies have remained steady or decreased across generations. Historical data collected from Micron datasheets, as seen in Figure 2.2, reveal the improvement of bus cycle time and minimum read latency in comparison with worsening refresh latency [18]. As these trends continue, memory refresh stands out as one of the determining factors in overall memory system performance.

Refreshing chips are unable to service memory requests until their refresh cycle has completed. The inability to access data from refreshing chips stalls program execution. *tRFC* has been steadily increasing because each new generation of DRAM has higher capacity and, therefore, contains more memory cells to refresh. Using refresh latency from the last four DRAM generations [38], we apply best fit regression to project the refresh latency for the next two generations of memory chips in Figure 2.3. *tRFC* will become 880ns and 1200ns in 32gb and 64gb devices, respectively.

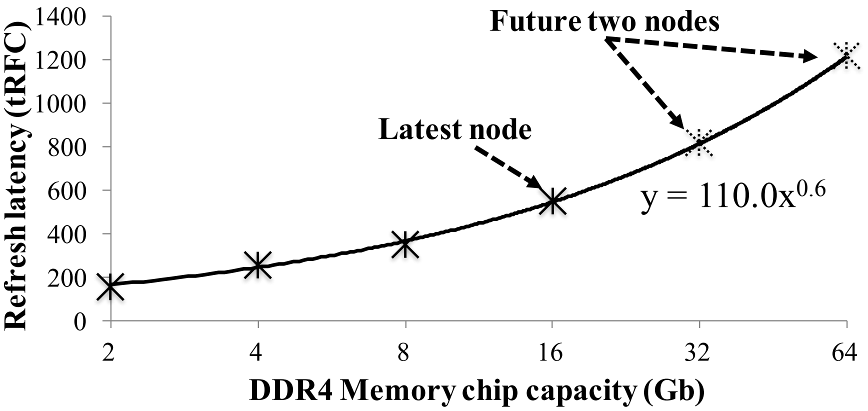

Figure 2.3: Historical [38] and projected refresh latency.

2.3 Skipping Refresh

Many recent works propose skipping many refresh operations, by increasing refresh interval, to improve performance [8, 25, 32, 35, 41, 42, 43]. For example, RAIDR [35] profiles the charge retention time of DRAM cells in each row in memory and skips refresh operations to memory rows with long retention time. However, skipping refresh reduces the average amount of charge stored in DRAM cells and, therefore, significantly increases DRAM vulnerability to read disturb errors [27]. This in turn significantly increases system vulnerability to software attacks that have exploited DRAM read disturb errors [15, 21, 27, 31]. Operating memory out-of-spec at reduced refresh rate may also increase memory fault rates because retention profiling cannot always identify all weak cells; higher memory fault rate in turn can degrade reliability. Reliability is important for server systems because an hour of server downtime can often lead to millions of dollars loss in revenue [20]. As such, data-center operators and decision-makers are often averse to adopting techniques with unquantifiable reliability risks [10]. Furthermore, out-of-spec operations can also void warranty and system-level agreements and thus degrade serviceability.

In summary, new solutions are needed to address memory refresh performance overheads for systems that have strict security, reliability, and serviceability requirements.

2.4 Motivation

Because server memory systems often contain many (i.e., 100s to 1000s) memory chips to provide high memory capacity, they need to protect against memory chips failing during system lifetime. As such, every memory block in server systems contains significant redundant data (see Figure 2.4) for hardware failure protection. The ratio of redundant data to program data in each block ranges from 12.5% to 40.6%.

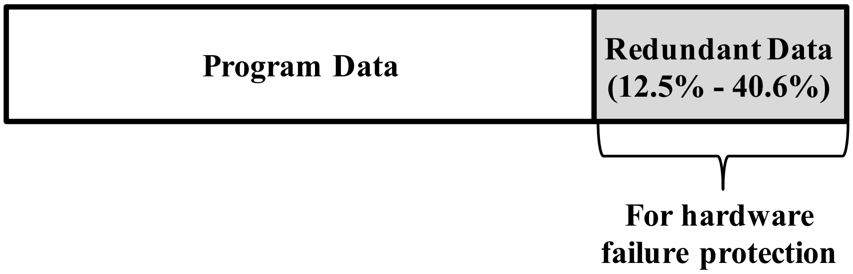
Program Data
Redundant Data (12.5% - 40.6%)
For hardware failure protection

Figure 2.4: Composition of a memory block in server memory.

Refreshing memory chips behave similarly to dead memory chips in that data stored in chips is inaccessible in both cases; as such, it should be possible to reuse the existing redundant data in server memory intended for chip failure protection to also compute data stored in inaccessible refreshing memory chips. To reuse redundant server memory to improve performance, we observe that individual memory chips are highly reliably as evidenced by the fact that systems with few memory chips, such as personal computers, mostly do not provide memory chip failure protection. Because individual chips are highly reliable, only a small fraction of memory locations, on average, experience hardware faults. As such, we can leverage the under-utilized redundant data in common-case fault-free memory locations to implement Nonblocking Refresh.

Figure 2.5 quantifies the expected fraction of memory pages that have not yet encountered any hardware fault by the N^{th} year of operation.[1] On average across seven years of operations, 97% of memory pages are not affected by any faults. While only a small fraction of memory pages experience fault, server systems protect memory pages with uniform redundant data because chip failures are stochastic events, whose time and location are difficult to predict.

[1] Figure 2.5 is calculated from the memory chip failure rate and patterns reported in a recent large-scale field study of memory failures [45], assuming eight ranks per channel and 18 chips per rank.

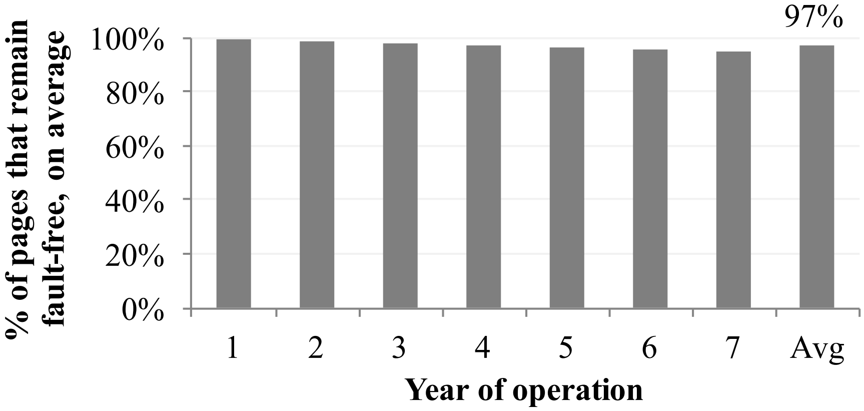

Figure 2.5: Expected fraction of memory pages that have not yet been affected by any fault as a function of time.

The general trend in the ratio of redundant data to program data in server memory is also increasing. The JEDEC memory standard reduces data bus width per channel from 64 bits in DDR4 to 32 bits for the upcoming DDR5 [4]. While reducing the width of the data bus naturally reduces the number of data chips per rank, the number of redundant chips per rank used to provide chipkill-correct remains the same; this doubles the ratio of redundant data to program data from 12.5% in a DDR4 rank to 25% in a DDR5 rank. Due to the increasingly disparity between the large amount of redundant data in server memory and the small fraction of that data actually being used to correct errors, we argue redundant data is an underutilized resource that can be reused to also improve memory performance.

Chapter 3

Nonblocking Refresh

We propose Nonblocking Refresh to refresh memory blocks while allowing read requests to access the refreshing blocks; it works by refreshing just a portion of the data in a memory block at any point in time, and uses per-block redundant data, such as Reed-Solomon codes, to reconstruct the unreadable/refreshing data in the block to satisfy read requests to the refreshing block. Compared to refreshing an entire block at a time as do conventional blocking refresh, Nonblocking Refresh can make up for refreshing only a portion of data in a block at a time by refreshing more frequently in the background. Nonblocking Refresh transforms DRAM to become functionally similar to SRAM in terms of refresh; under Nonblocking Refresh, DRAM refreshes continuously in the background without blocking read requests to refreshing memory blocks.

In this paper, we focus on exploring Nonblocking Refresh in the context of server memory systems. In this context, Nonblocking Refresh can exploit existing abundant redundant data in server memory to compute refreshing data in each block, without requiring any dedicated redundant data of its own. Designing Nonblocking Refresh for server memory requires addressing three main challenges: 1) How to reuse existing redundant data in server memory to perform Nonblocking Refresh? 2) How to perform the same aggregate amount of refresh as the conventional approach of refreshing an entire block at a time? 3) Redundant data must preserve its original purpose of hardware failure protection in the event that memory faults do suddenly occur. Therefore, a third challenge is how to preserve baseline failure protection while leveraging redundant data to implement Nonblocking Refresh?

3.1 How to Utilize Existing Redundant Server Memory Data?

Conventional server memory systems cannot exploit redundant data to compute inaccessible data stored in refreshing chips because they refresh all chips in a rank at the same time. As such, all data will be missing from a memory block read from a refreshing rank (see Figure 3.1A), making it impossible for redundant data to compute any missing data.

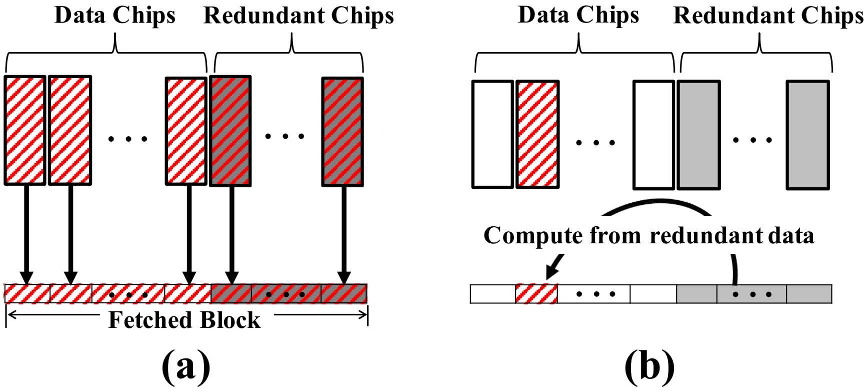

Figure 3.1: (a) Conventional refresh. (b) Nonblocking Refresh. Red represents inaccessible data stored in refreshing memory chips.

To compute inaccessible data stored in refreshing chips, the amount of inaccessible data in each block must be less than the maximum amount of data that the block's redundant data can reconstruct. We propose refreshing few chips in a rank at a time so that only a small fraction of the data in each block are inaccessible due to refresh. Figure 3.1B shows an example that refreshes only one chip at a time. The MC uses the block's redundant data to compute the missing data in the block to complete the read request to the block.

Computing the missing data is fast because the MC already knows which memory chip(s) are refreshing, unlike regular error correction, where the MC needs to locate the error before computing the error value. Computing the value of errors whose locations are known is called *erasure correction*. The vast majority of latency during error correction is to locate the error; computing the error value after knowing the error location incurs only a few cycles of latency [29]. Erasure correction also only consumes small amount of power. Prior study using 180nm transistor process technology report that erasure correction only consumes 200-500uW [29]; it should be even lower in today's 14nm process technology. Enhancing the MC to perform erasure correction for Nonblocking Refresh also incurs little to no area overhead because the error correction logic in conventional server systems' MCs already contains the hardware to compute the correct values of located errors.

To enable Nonblocking Refresh, the chips in each rank are logically partitioned into *refresh groups*. A Nonblocking Refresh operation refreshes a single refresh group. Since conventional server memory systems refresh all chips in a rank simultaneously, some hardware modifications are needed to refresh each refresh group individually.

One possible implementation of refresh groups is to refresh the refresh groups in a round-robin fashion and modify each memory chip to ignore refresh commands designated for other refresh groups; modifying a chip to ignore some refresh commands is similar to a recent work that skips refresh [8]. For a rank with N refresh groups, the memory chips belonging to a refresh group ignores $N - 1$ out of every N refresh commands such that each command refreshes only one refresh group. By refreshing the refresh groups in a round-robin fashion, the MC can track which refresh group is refreshing by counting the past Nonblocking Refresh operations via a modulo counter. Since current DRAM standards dictate that a refreshing chip should not receive any valid commands, the chips also need to be modified to ignore other commands while refreshing. When a refresh group exits refresh, it may be out-of-sync with the row buffer state of the remaining chips in the rank. The MC can synchronize all chips in the rank by issuing a precharge_all command to the rank.

Another possible implementation of refresh groups is to modify the DIMM rather than the memory chips themselves. We observe that a memory chip ignores all commands, including refresh commands, unless its chip select (CS) input bit is asserted [22]. To refresh individual refresh groups, we can simply devote a CS bit to each refresh group, instead of devoting a CS bit to an entire rank as do conventional systems. The MC initiates Nonblocking Refresh for a refresh group by asserting only the CS bit of the desired refresh group when issuing a refresh command.

3.2 How to Ensure Each Chip Performs Same Amount of Refresh as Conventional Blocking Refresh

One obvious challenge with refreshing only some of the chips in a rank at a time is how to perform same amount of refresh in each chip as the conventional approach of refreshing all chips in a rank at the same time. The MC must issue Nonblocking Refresh more frequently than conventional blocking refresh to make up for refreshing fewer chips at a time. We observe that because Nonblocking Refresh does not block read requests, the MC can refresh memory continuously in the background with minimum performance impact. Conventional systems with blocking refresh, on the other hand, can only refresh each rank infrequently to avoid excessively blocking read requests. Figure 3.2 contrasts the timeline of Nonblocking Refresh with the timeline of conventional refresh.

Since Nonblocking Refresh is performed more frequently than conventional blocking refresh, Nonblocking Refresh can incur command bus bandwidth overheads. Assuming a single rank per channel and $tRFC = 550ns$ [38], if the MC issues refresh commands back to back

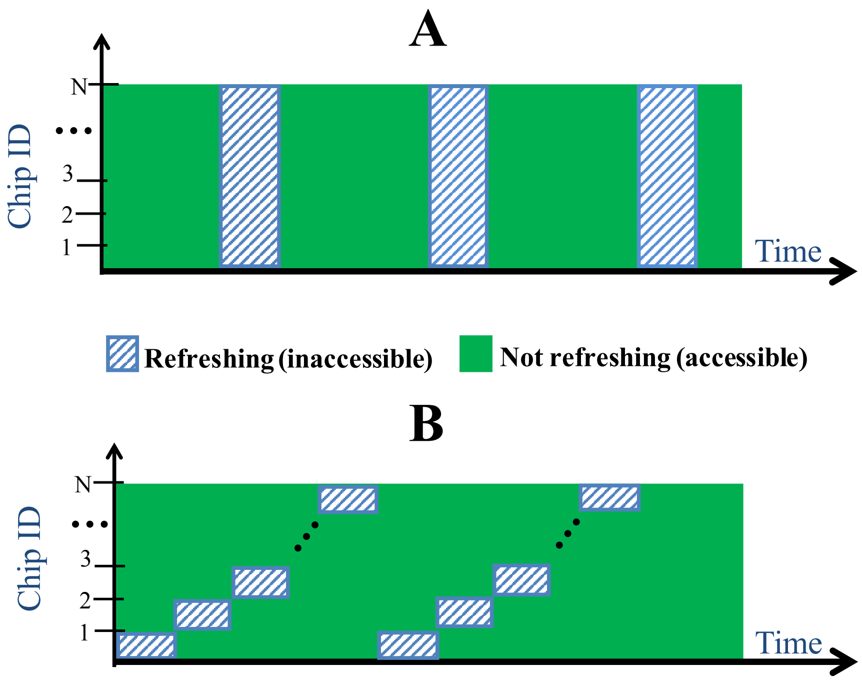

Figure 3.2: Timelines of (a) blocking refresh and (b) Nonblocking Refresh

after every tRFC, the aggregate command bus bandwidth is only $0.2 - 0.4\%$. However, this command bus bandwidth overhead increases proportionally with the number of ranks in the channel; this may translate to non-negligible (e.g., 5%) command bus bandwidth utilization for very large channels. One effective solution for very large channels is to let multiple ranks (e.g., all ranks in the same DIMM) in the same channel perform Nonblocking Refresh in parallel for each refresh command MC places on the command bus.

Depending on the refresh group size and tREFI, Nonblocking Refresh may not always fully keep up the conventional approach of refreshing entire blocks at a time. In this scenario, a memory system with Nonblocking Refresh may need to occasionally perform conventional blocking refresh to meet requirement. Even in this scenario, Nonblocking Refresh still helps to avoid many conventional blocking refresh and, therefore, improves performance compared to only performing conventional refresh. A memory system with Nonblocking Refresh may use a per-rank hardware counter to count the number of past Nonblocking Refresh operations;

after a rank has performed the same number of Nonblocking Refresh as there are refresh groups, the MC does not need to issue a blocking refresh to the rank at the next tREFI time interval.

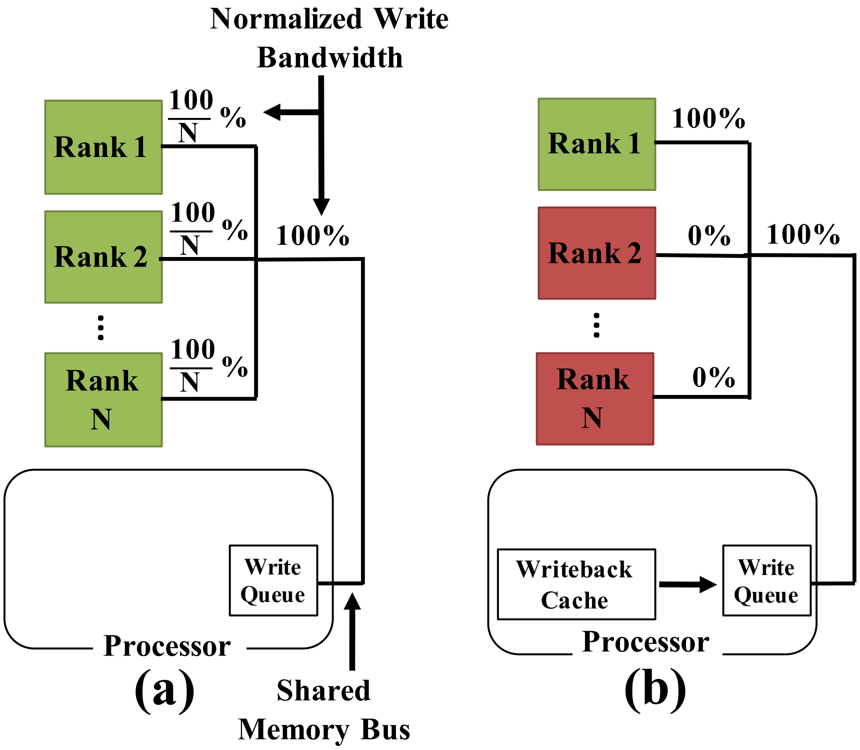

Figure 3.3: Write distribution in (a) conventional and (b) proposed memory systems. Green ranks are not refreshing and, therefore, writable; red ranks are refreshing and, therefore, not writable.

Unlike read requests, write requests can be negatively impacted when each rank refreshes frequently/continuously. Writes to a rank cannot proceed in parallel with refreshing the rank because data in a chip cannot be updated while a chip is refreshing. Write requests still need to wait for a rank to complete any in-flight refresh operations before they can proceed. Therefore, refreshing each rank more frequently can potentially increase write latency and reduce write bandwidth. We note that increasing write latency does not degrade performance because memory writes are not on the critical path of program execution; however, reducing write bandwidth can degrade performance because it can reduce the throughput of memory store instructions.

To maintain memory write bandwidth while frequently performing Nonblocking Refresh, we make two observations. First, since all ranks in the same channel share the same memory bus, the MC can only write to one rank at a time. Therefore, total write bandwidth in a channel is divided across all the ranks in the channel, as shown in Figure 3.3A. Second, logically adjacent memory pages are often interleaved across ranks to minimize read latency overheads due to row conflicts. This interleaving causes write requests to distribute fairly evenly among all ranks in the channel. Based on these observations, we propose re-ordering write requests to concentrate each channel's write bandwidth to a few ranks at a time as shown in Figure 3.3B. This maintains the same channel-level write bandwidth while allowing the remaining ranks in the channel to continuously perform Nonblocking Refresh.

We propose logically grouping the ranks in a channel into separate *write groups*, such that each channel with N ranks contain K write groups, with N/K ranks per write group. During each $tRFC$ interval, the MC writes to one of the K write groups while performing Nonblocking Refresh to the remaining $K-1$ write groups. The remaining ranks will complete their current Nonblocking Refresh after each $tRFC$ interval. At the same time, the MC selects a different write group to write to and again puts the remaining ranks under Nonblocking Refresh. This approach can provide the channel-level write bandwidth of conventional systems while allowing the majority (i.e., $(K-1)/K$) of the ranks to benefit from Nonblocking Refresh. Server memory often contains many ranks per channel to provide adequate capacity; as such, they can often benefit from a large $(K-1)/K$ value (e.g., 3/4 for channels with just four ranks per channel).

Re-ordering write requests to only one write group per $tRFC$ interval requires modifying the MC to buffer more writes. We use Little's Law [34] to estimate the size of the write buffer needed to match the outgoing rate of the write buffer in the worst-case arrival rate of write requests. Little's Law states that the average number of elements in a queue is $L = \lambda \cdot W$, where λ is the average arrival rate and W is the average time each element waits in the queue [34]. In the context of the write buffer, L is buffer size, λ is the memory write bandwidth, and W is how long, on average, a block needs to wait in the buffer until its write group is selected for writes. Assuming write requests account for at most half of total memory requests because a processor typically needs to first fetch a block from memory before writing to the block, $\lambda =$ 12.8GBps for a 3.2ghz and 64-bit wide channel. With K write groups, a newly arrived block waits, on average, $K \cdot tRFC$ before its write group is selected; as such, we pessimistically estimate $W = K \cdot tRFC$. $W = 4 \cdot 550 = 2200ns$ assuming a server system with 16gb chips (550ns $tRFC$) [38] and four write groups per channel. Together, the new size of the write buffer for the channel should be $L = 12.8 \cdot 2200 = 28$kB.

We implement the write buffer as a set-associative writeback cache. When the MC receives an evicted dirty block, the MC places the block in the writeback cache instead of immediately placing it in the write queue used by the memory command scheduler. At the end of each $tRFC$ interval, the MC selects an active write group to write to for the next $tRFC$ interval; the MC first determines the most occupied set in the writeback cache and then selects the write group with the most cachelines in that set as the active write group. However, there are two special cases. If the most occupied set in the writeback cache has less than a threshold occupancy (e.g., 75% in our evaluation), the MC does not select an active write group so that all write groups can continue to perform Nonblocking Refresh during the next $tRFC$ interval. On the other hand, $tREFI$ may not be evenly divisible by $tRFC$; if a blocking refresh is required at the next $tREFI$ interval and there is not enough time for perform a Nonblocking Refresh, all ranks become active write groups. After selecting one or more active write groups, the MC drains the write group(s)' dirty blocks from the writeback cache to the write queue whenever it has available entries, starting from the most occupied cache set to the adjacent set in a round-robin fashion. The memory command scheduler only scans the write queue to schedule write commands; it is oblivious of the writeback cache.

3.3 How to Preserve Failure Protection?

Nonblocking Refresh improves system performance by reusing the redundant data in server memory to compute the inaccessible data in refreshing memory chips. This should not detract from the original purpose of redundant data - hardware failure protection. The following lists a set of sufficient conditions that, if all true, enables a server memory system with Nonblocking Refresh provide equal hardware failure protection as a conventional system with same amount of redundant data: A) Memory systems with Nonblocking Refresh should not increase the physical/raw fault rate of memory chips compared to conventional systems. B) Both systems should have identical error *detection* strength. C) Both systems should have identical error *correction* strength.

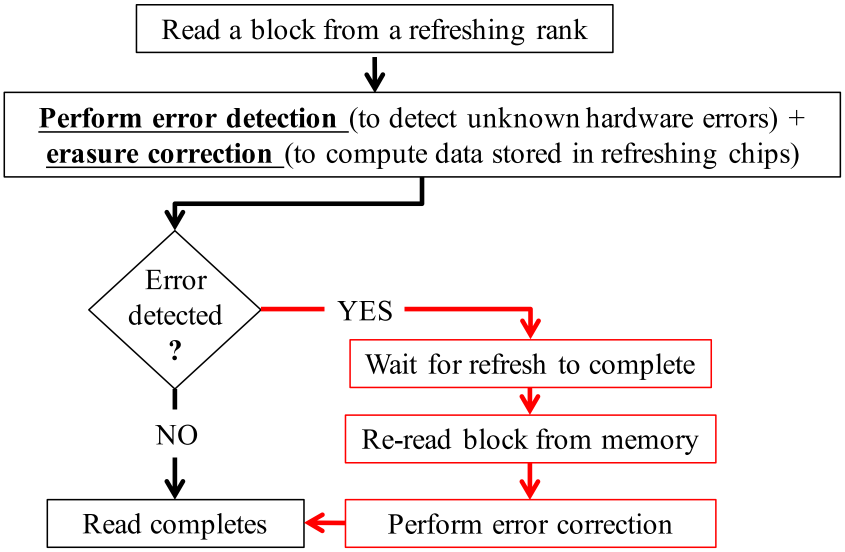
Read a block from a refreshing rank
Perform error detection (to detect unknown hardware errors) +
erasure correction (to compute data stored in refreshing chips)
Error detected ?
YES
NO
Wait for refresh to complete
Re-read block from memory
Perform error correction
Read completes

Figure 3.4: Action flow for reading from a rank under Nonblocking Refresh.

We meet A) because memory systems with Nonblocking Refresh can perform the same amount of refresh as conventional memory systems (see Section 3.2) and, therefore, can maintain baseline memory system's physical fault rates. To meet B), we observe that each block contains some redundant data for error detection; as such, we can meet B) by using the same amount of redundant data to detect fault-induced random errors for each read request as baseline memory systems. To meet C), only when no random errors are detected does Nonblocking Refresh opportunistically reuse the redundant data intended for error correction to compute data missing due to refresh. When random errors are detected in a fetched block, the MC waits for the rank to finish its in-flight refresh and then re-read the same block from memory, as shown in Figure 3.4. Since the rank is no longer refreshing when the second read is performed, the re-fetched block no longer misses any of its data due to refresh; as such, the redundant data in the re-fetched block can correct fault-induced random errors in the exact same way as baseline memory system and, therefore, preserve baseline error correction strength. We examine three specific server memory systems to further demonstrate how to meet B) and C) in more detail.

Many Intel and AMD server systems protect memory with single chipkill-correct (SCC) [5, 47]. SCC memory systems guarantee detection and correction of one faulty chip per rank. SCC memory systems protect K data bytes, each from a different data chip in a rank, with two check bytes, each from a different redundant chip in a rank. We observe that the same check bytes in a codeword can be used in many different ways [33]. R check bytes can guarantee detection and correction of R/2 unknown error bytes; as such, the two check bytes per codeword in SCC memory systems can guarantee detection and correction of one random error byte. Meanwhile, the same R check bytes can also be used instead to guarantee detection of Q unknown error bytes and correct another P erasures (i.e., missing bytes at a known locations), where $P + Q = R$ [33]. When applying Nonblocking Refresh to SCC memory systems, the refresh group size should be one; as such, MC only uses $P = 1$ check byte per codeword for erasure correction. Since there are two check bytes per codeword in SSC memory systems, each codeword can still guarantee detection of $Q = 2 - 1 = 1$ unknown error byte per codeword and, therefore, guarantee single chip failure detection just like baseline SSC systems that only perform conventional blocking refresh. Figure 3.5A shows a detailed example for SCC memory systems where the third data chip in a rank is being refreshed. Figure 3.5B shows a corresponding codeword read from the refreshing rank; the third byte in the codeword is missing because the third chip in the rank is refreshing. The MC can use any one of the codeword's two check bytes to compute the missing byte via erasure correction; the remaining check byte can guarantee detection of any single random error byte in the codeword and, therefore, guarantee detection of one chip failure.

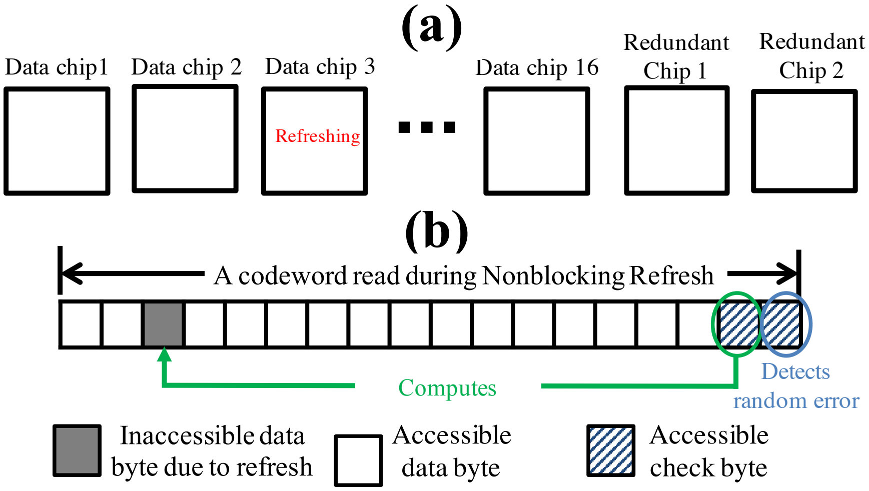

Figure 3.5: (a) A SSC memory rank under Nonblocking Refresh. (b) a codeword read from the refreshing rank.

Many IBM servers provide MCC in their memory systems to correct multiple faulty chips per rank as long as a second chip does not fail before the first faulty chip has been logically replaced [17]. Under MCC, each rank has four redundant chips, two used in the same manner as SCC to guarantee single-chip failure detection and two spare chips to logically replace up to two previously observed faulty data chips [17]. When applying Nonblocking Refresh, a

MCC memory system can modify each codeword to store four check bytes, instead of two check bytes and two spare data bytes. With four check bytes per codeword, a MCC memory system can use one check byte per codeword to guarantee single-chip failure detection at the rank level and the remaining three check bytes per codeword to implement a refresh group size of three. In the uncommon-case when a MCC memory system needs to replace a faulty data chip, it can revert the faulty rank back to storing two check bytes and two spare bytes per codeword.

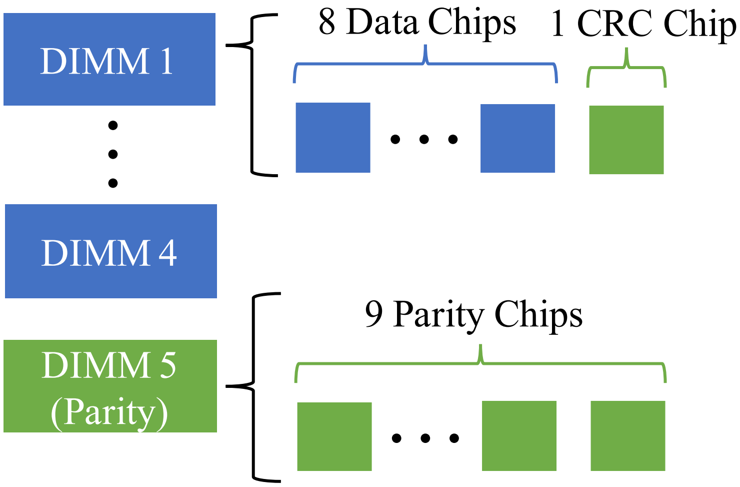

Figure 3.6: Layout of a rank with RAIM protection.

High-end IBM servers protect their memory systems with RAIM to tolerate the complete failure of an entire DIMM [16]. A RAIM memory system contains 45 chips per rank, organized in groups of $45/5 = 9$ chips across five different DIMMs, as shown in Figure 3.6. Four of the groups store data; the fifth group stores a bitwise parity of the four data groups to provide error correction. Each data group also contains one redundant chip to store CRC guarantee detection of a single chip failure per data group. When applying Nonblocking Refresh, the refresh group size is nine. Eight check bytes from the parity group can compute the program data missing due to Nonblocking Refresh, while the 9th check byte from the parity group can compute the error detection byte for the eight computed data bytes to guarantee the same single chip failure detection as a conventional RAIM system. When the parity group itself is refreshing, the MC does not need to reconstruct any data for fetched blocks because no program data are missing from these blocks when only the parity group is refreshing.

One potential performance bottleneck with opportunistically reusing existing redundant data intended for hardware failure protection to implement Nonblocking Refresh is that requiring a second read whenever an error is detected can effectively increase error correction latency to $tRFC$. If a rank experiences a permanent chip failure, every read to the require will require error correction. To address this problem, the MC can dynamically decide to only perform conventional blocking refresh for faulty ranks. We will quantify the performance impact of permanent chip faults in Section 4.2.

Chapter 4

Evaluation

4.1 Methodology

4.1.1 Baselines

We evaluate a conventional memory refresh baseline that refreshes each rank every tREFI time interval; we refer to this baseline as *Conventional Refresh*. We also evaluate a baseline that completely skips 75% of refresh operations and optimistically assume that it requires no other operations or overheads; this baseline represents the best-case of all prior works that propose skipping refresh [8, 25, 32, 41, 42]. We refer to this baseline as *Skipping Refresh*.

4.1.2 Processor and Workloads

We simulate a 16-core out-of-order processor using Gem5 [9], a cycle-accurate micro-architectural simulator. Table 4.1 lists the micro-architectural parameters used for simulation. We ob-

Table 4.1: Processor Microarchitecture

Core	16 cores, 3GHz, 4-issue OOO 128 ROB entries, 64B cacheline size
L1 d-cache, i-cache	2-way, 64kB, 1 cycle
Private L2 cache	8-way, 512kB, 3 cycles
Shared L3 cache	32-way, 32MB, 14 cycles

Table 4.2: Mixed Workload Composition

mixA	4 omnetpp, 4 mcf, 4 wrf, 4T ocean_cp
mixB	4 bwaves, 4 cactusADM, 4 wrf, 4T ocean_cp
mixC	4 sjeng, 4 cactusADM, 4 radiosity, 4T radix
mixD	4 mcf, 4 GemsFDTD, 4T barnes, 4T radiosity
mixE	4 cactusADM, 4 bwaves,4 sjeng, 4T fft
mixF	4 mcf, 4 omnetpp,4 astar, 4T fft
mixG	4 GemsFDTD,4 astar, 4 bwaves, 4T barnes

tain cache access latencies from CACTI for the 32nm technology node [30]. We evaluate 16 threads per workload, seven single-application NASBench [2] workloads and seven multi-programmed workloads (see Table 4.2 for composition); only native and reference inputs are used. The workloads' memory footprints range from 10GB to 35GB and are 17GB on average. We fast forward each workload until all multi-threaded application(s) have initialized and then by another 20 simulated seconds. Next, we warm up the caches by 20 simulated milliseconds and then perform cycle-accurate simulation for the next 10 milliseconds. Since all workloads contain multi-threaded applications, we measure throughput not by total instructions, but by FLOPs for workloads with only FP benchmarks and by instructions that access main memory for the remaining workloads. Figure 4.1 characterizes the memory behavior of these workloads during the 10 millisecond cycle-accurate measurement.

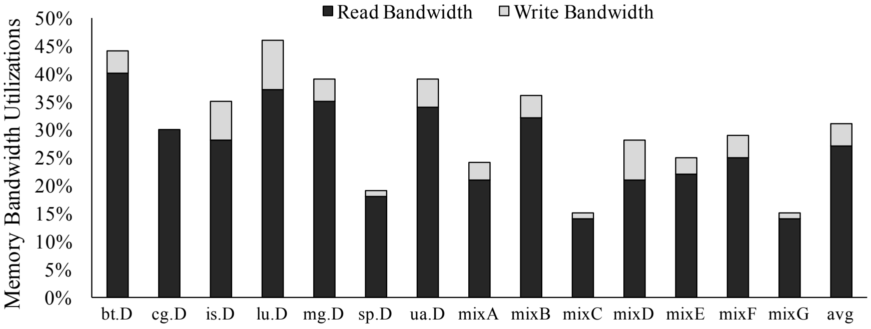

Figure 4.1: Workload characterization.

4.1.3 Memory System Modeling

We use Ramulator [28] and DRAMPower [12] to measure memory performance and power, respectively, for 3200Mhz DDR4 DRAM by using the latency and current values reported in [38] as input parameters. We model the $tRFC$ of the latest 16gb chips and the future 32gb chips by using the values given in Figure 2.3. When modeling the future 32gb chips, we pessimistically assume latencies unrelated to refresh remain the same as current DRAMs, instead of keep reducing according to historical trends as shown in Figure 2.2. We simulate the FR-FCFS scheduling policy and the open-page row buffer policy and prioritize reads over writes. We evaluate an address mapping policy that interleaves logically adjacent pages across channels, banks, and then ranks. We evaluate four ranks per channel. Each channel contains a 64-entry read queue, 64-entry write queue, and 64-entry command queue. For Nonblocking Refresh, we model a 36-way 36KB writeback cache per 64-bit channel and four write groups per channel, where each rank is a write group. For the baselines, we model staggered refresh, similar to prior works [8, 13], and optimize staggered refresh by applying DARP [13] at the rank level.

We evaluate the three memory systems described in Section 3.3 - SCC, MCC, and RAIM memory systems. Commercial SCC memory systems and MCC memory systems use X4 and X8 memory chips, respectively [5, 17]; we refer to them as SCC_X4 and MCC_X8. To explore the effectiveness of Nonblocking Refresh when applied to server memory systems with different redundancy, we also evaluate SCC and MCC implementations using X8 and X4 memory chips, respectively; we refer to these implementations as SCC_X8 and MCC_X4, respectively. We implement SCC_X8 by cutting the number of chips per rank in MCC_X8 by half; we implement MCC_X4 by replacing all the chips in MCC_X8 with X4 chips and doubling the number of data chips per rank. Table 4.3 summarizes the memory organization for the evaluated memory systems.

Table 4.3: Evaluated Memory Configurations

System	Chip Width	Chips/rank	Channels	Redundant chips
SCC_X4	X4	18	four	two (12.5%)
SCC_X8	X8	10	four	two (25%)
MCC_X4	X4	36	two	four (12.5%)
MCC_X8	X8	20	two	four (25%)
RAIM	X4	45	two	twelve (40.6%)

When modeling Nonblocking Refresh, we set refresh group size to one for SCC_X4 and SCC_X8. We set refresh group size to nine for RAIM memory systems. For MCC_X8 memory systems, there are six refresh groups with three chips and one refresh group with two chips because each rank contains 20 chips, which is not divisible by three. We set refresh group size to three for MCC_X4 memory systems. We model the latency of erasure correction as four clock cycles; this corresponds to the latency of the Forney algorithm, which computes the correct values of located errors [29].

4.2 Results

4.2.1 Performance Comparison

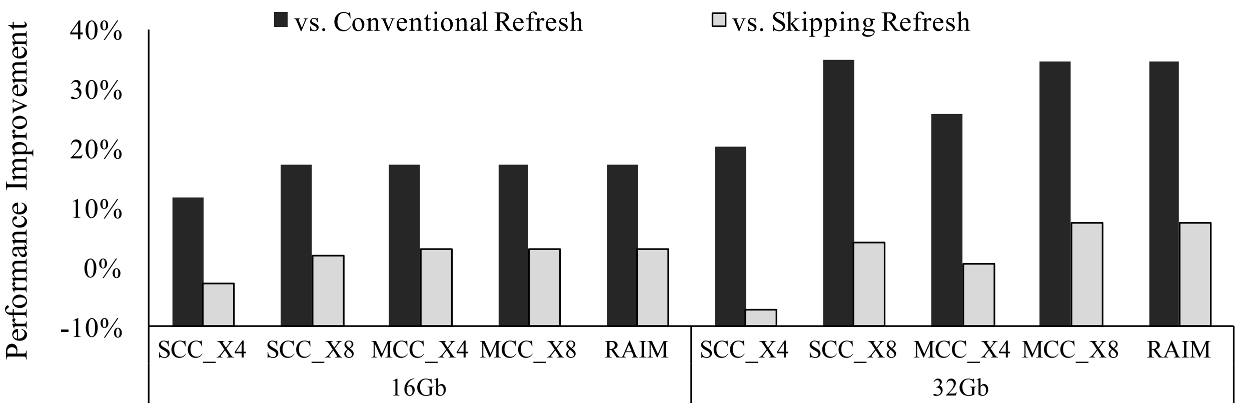

Figure 4.2: Average performance improvement vs. Conventional Refresh and Skipping Refresh for 16gb and 32gb DRAM.

Figure 4.2 shows the average performance improvement of Nonblocking Refresh over Conventional Refresh for 16gb and 32gb DRAM. Each bar (e.g., the bar for "SCC_X4") in Figure 4.2 shows the average performance improvement across all 14 workloads when Nonblocking Refresh and Conventional Refresh are applied to the same memory system (e.g., "SCC_X4"). On average across the five memory systems, Nonblocking Refresh provides 16.2% and 30.3% performance improvement for 16gb and 32gb DRAM, respectively. The performance improvement is higher for 32gb DRAM because 32gb DRAM has a longer refresh latency than 16gb DRAM; when refresh latency is longer, reducing the performance overhead of memory refresh can yield greater overall system-level performance benefit.

SCC_X4 memory systems receive the least performance improvement; it is only 13% for 16gb DRAM and 21.0% for 32gb DRAM. In comparison, the average performance improvement obtained under the remaining memory systems are $16.9\% - 17.1\%$ for 16gb DRAM and $27\% - 35\%$ for 32gb DRAM. SCC_X4 memory systems receive the least performance benefit because Nonblocking Refresh can only refresh $1/18^{th}$ of each rank at a time, the lowest among all five memory systems. As a result, SCC_X4 memory systems with Nonblocking Refresh must perform the most blocking refresh operations among all evaluated memory systems.

Figure 4.2 also shows the average performance improvement of Nonblocking Refresh over Skipping Refresh for 16gb and 32gb DRAM. On average across the five memory systems, Nonblocking Refresh provides 2.3% and 3% performance improvements compared to Skipping Refresh for 16gb and 32gb DRAM, respectively. Nonblocking Refresh can sometimes perform better than Skipping Refresh because Skipping Refresh still requires performing some blocking refresh; on the other hand, when Nonblocking Refresh can completely keep up with blocking refresh, *all* blocking refreshes can be prevented. More memory systems show performance improvement relative to Skipping Refresh under 16gb DRAM chips than under 32gb DRAM chips because 16gb chips have shorter refresh latency, which enables Nonblocking Refresh to more easily keep up with blocking/full-rank refresh. Note that while the performance of Nonblocking Refresh is within 3%, on average, of Skipping Refresh, Nonblocking Refresh meets the required amount of refresh and, therefore, is applicable to systems with strict security and reliability requirements.

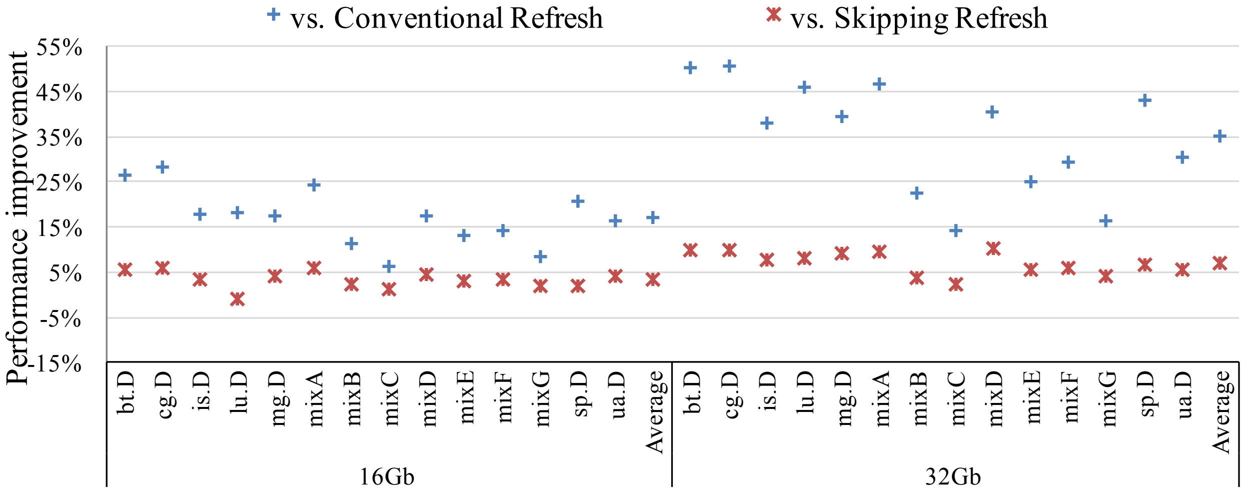

Figure 4.3: Performance improvement for MCC_X8 memory systems.

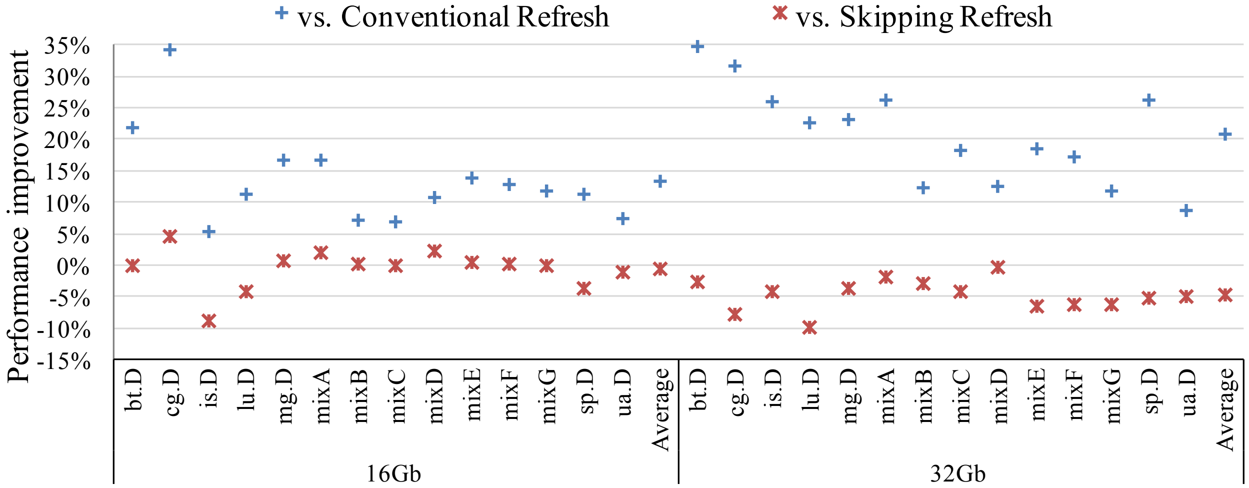

Figure 4.4: Performance improvement for SCC_X4 memory systems.

Figure 4.3 shows that Nonblocking Refresh consistently provides higher performance than conventional refresh for all the workloads with a MCC_X8 memory system. *cg.D* enjoys the highest performance improvement - 28% and 51% - for 16gb and 32gb DRAM, respectively. Figure 4.3 shows both memory-intensive workloads such as *bt.D* and *lu.D* (see Figure 4.1) and workloads with low bandwidth utilization, such as *mixC* and *mixG*, can benefit from Nonblocking Refresh; workloads with low bandwidth utilization also benefit from reducing blocking refreshes because the long refresh latency of $> 500ns$ can still stall execution for a long time even if memory accesses are less frequent. Figure 4.4 shows the performance improvement of Nonblocking Refresh for a SCC_X4 memory system follows similar trends.

4.2.2 Power Comparison

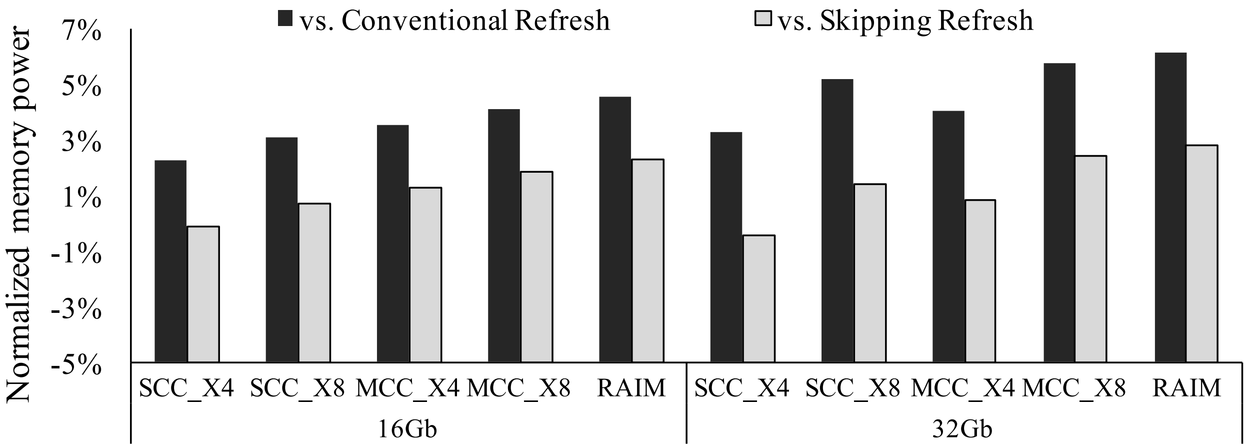

Figure 4.5: Memory power vs. Conventional Refresh and Skipping Refresh.

Figure 4.5 shows the power consumption of memory systems with Nonblocking Refresh normalized to memory systems with conventional blocking refresh and Skipping Refresh. The power consumption of memory systems with Nonblocking Refresh is higher than memory systems with conventional refresh. This is because Nonblocking Refresh improves performance compared to conventional refresh; as such, memory systems with Nonblocking Refresh can complete more read requests and, therefore, consume more power. For this reason, Figure 4.5 shows that the power increase for systems which gain the least performance benefit from Nonblocking Refresh, such as SCC_X4, is lower than that of systems which gain the most performance benefit from Nonblocking Refresh, such as RAIM and MCC_X8. Note that while Nonblocking Refresh improves performance by 17% to 35%, it increases memory power consumption by only 2% to 6%; this is because Nonblocking Refresh reduces the average power consumption of memory read requests since they are ignored by refreshing chips in a rank when the rank is performing Nonblocking Refresh.

4.2.3 Performance Analysis for Faulty Ranks

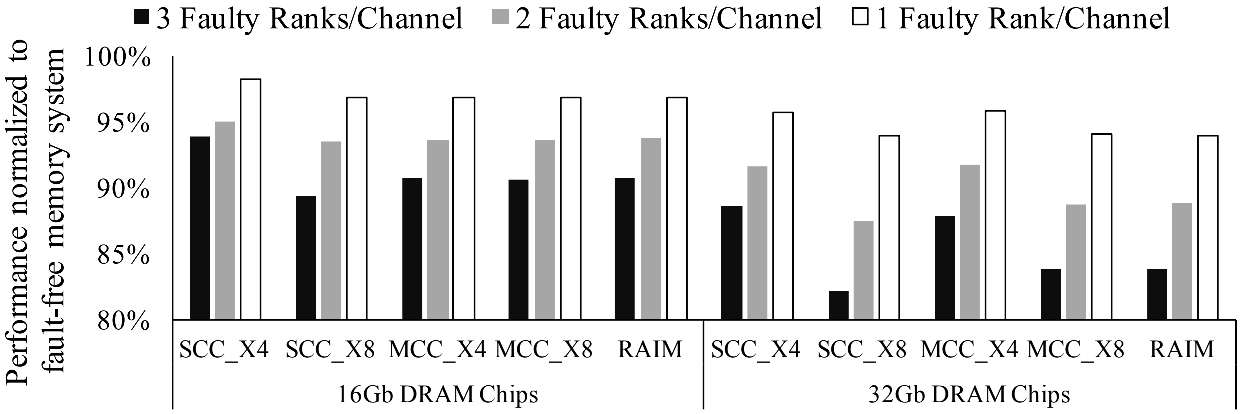

Figure 4.6: Sensitivity analysis for Nonblocking Refresh: performance of systems with faulty chips normalized to performance of fault-free systems.

To quantify the effects of permanent chip failure on Nonblocking Refresh, we evaluate the performance degradation when a rank falls back on performing only conventional blocking refresh. Figure 4.6 shows the performance of Nonblocking Refresh for each memory configuration when one, two, or three ranks out of the four ranks per channel only perform conventional blocking refresh; this is normalized to the performance of a memory system where all four ranks are performing Nonblocking Refresh. The presence of faulty ranks impacts the performance of memory systems with 32gb DRAM more than memory systems with 16gb DRAM because the longer refresh latency in the former impacts system performance more than the latter; as such, memory systems with 32gb DRAM have more to lose when some of their ranks cannot perform Nonblocking Refresh.

To estimate the average performance degradation due to chip failures, we assume that a rank falls back on performing only conventional blocking refresh after encountering a single permanent multi-bank or multi-rank fault [45]; the memory system retires all pages affected by smaller faults, such as permanent bank, column, or row faults, because each such fault affects only 0.4% – 0.8% of each evaluated system's total memory capacity. Assuming the above and the memory chip fault rates reported in [45], each rank falls back on performing only conventional blocking refresh < 1% of the time, on average across a seven-year lifetime.

4.2.4 Writeback Cache Size Sensitivity Analysis

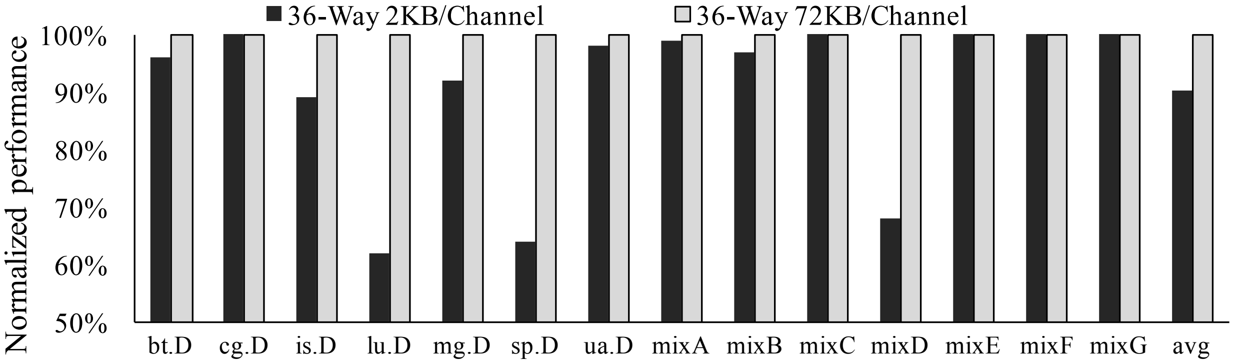

Figure 4.7: Performance of Nonblocking Refresh with different writeback cache sizes normalized to 32KB writeback cache.

The writeback cache is an important component of Nonblocking Refresh. To evaluate how the writeback cache size can affect Nonblocking Refresh performance, we measure the performance of Nonblocking Refresh with increased and decreased writeback cache sizes. Figure 4.7 shows the performance of Nonblocking Refresh in SCC_X4 memory systems with a smaller, 36-Way 2KB per channel writeback cache and with a larger, 36-Way 72 KB per channel writeback cache; this is normalized to the performance of Nonblocking Refresh in a SCC_X4 memory system using 32gb DRAMs with the proposed 36-Way 32KB per channel writeback cache. Our evaluation shows that Nonblocking Refresh performance does not benefit from a bigger 72KB writeback cache; this is because a 36KB writeback cache is already large enough to maintain the same write bandwidth for Nonblocking Refresh as conventional systems. On the other hand, an insufficient writeback cache size causes a significant degradation in Nonblocking Refresh performance. For workloads such as *lu.D* and *mixD*, which have high write bandwidth utilization (see Figure 4.1), using a small 2KB writeback cache can degrade performance by almost 40%.

Chapter 5

Discussion

5.1 Related Works

5.1.1 Other Works that Leverage Redundant Data

To enable future DRAM density scaling, memory manufacturers may embed error correcting code (ECC) bits and ECC logic within future memory chips [11, 23]; this is known as *on-die ECC*. In the context of DDRx server memory, on-die ECC has been proposed to correct bit errors due to manufacturing defects in future DRAMs [11, 23]; one can envision a strong-enough (and expensive-enough) on-die ECC implementation that also reduces the refresh rate of future DRAMs by correcting errors in DRAM cells with lower retention time than average. However, how much refresh rate can be reduced is limited by the mean/median retention time, which keeps reducing as DRAM cell size reduces. As such, Nonblocking Refresh, which can directly tackle the performance overheads of high-rate refresh head on without reducing refresh rate, provides orthogonal/additive performance benefits/scaling beyond techniques that do require reducing DRAM refresh rate to improve memory performance.

Erasure coding has been used in other contexts to compute data in temporarily inaccessible storage or memory devices. For example, BitTorrent, a popular peer-to-peer file sharing network, creates redundant file chunks using erasure codes to enable clients to compute inaccessible file chunks stored in temporarily off-line peers from redundant file chunks distributed across the network [44]. Shibo et al. [46] propose adding redundant HMCs (Hybrid Memory Cubes) storing erasure codes to compute the data stored in HMCs currently placed in inaccessible power-down modes. Yan et al. [48] propose using erasure coding to compute data in Flash chips that are occupied by on-going garbage collection operations. Mohammad et al. [6] propose adding redundant PCM (Phase Change Memory) chips to store erasure codes to compute inaccessible data stored in PCM chips occupied by long latency writes. However, we apply erasure codes in the context of DRAM refresh and address many new challenges specific to this new application context.

5.1.2 Fine-Grained Refresh

Other works have proposed leveraging fine-grained control of refresh scheduling to enhance parallelization of refresh and access to DRAM. Alternative refresh modes that operate at a finer granularity than traditional refresh break each refresh operation into smaller units. Although this offers some performance benefits by reducing the size of inaccessible memory region in each refresh cycle, access to a refreshing block still stalls; as such, fine-grained refresh is a stopgap solution that do not scale well in the face of growing DRAM density.

One type of fine-grained refresh is per-row refresh, which refreshes one row every refresh command. This requires significantly more refresh commands than traditional refresh, leading to increased consumption of command bus bandwidth. Support for per-row refresh in standard DRAM has been deprecated due to its high command bus overhead.

DDR4 DRAM includes a Fine Granularity Refresh (FGR) feature with 2x and 4x refresh modes as an alternative to traditional 1x mode. By refreshing fewer rows per command than 1x mode, 2x and 4x refresh modes have a shorter $tRFC$ at the cost of issuing commands twice and four times more frequently, respectively. The success of FGR is limited because

$tRFC$ does not decrease proportionally with increasing refresh rate. More specifically in a 16gb DDR4 system, 2x mode takes almost 30% longer than 1x mode refresh the same number of rows [22]. Due to this overhead, previous works have found that FGR offer small performance benefits [39].

Per-bank refresh is another form of fine-grained refresh which refreshes one bank in a rank at a time in LPDDR4 DRAM [1]. Ideally this mitigates refresh overhead by allowing parallel access to the remaining banks which are not refreshing; however accesses to the refreshing bank must still wait. The ratio of $tRFC_{ab}$ -to- $tRFC_{pb}$ is how long a all-bank refresh command takes compared to a per-bank refresh command. In LPDDR DRAMs, $tRFC_{ab}$ -to- $tRFC_{pb} = 2$; however, LPDDR DRAMs need to issue per-bank refresh commands 8 times as frequently as all-bank refresh because there are 8 banks in a chip [1]. As DRAM becomes denser, $tRFC$ increases, making it difficult for per-bank refresh to keep up with the required refresh frequency. To examine how increasing density impacts per-bank refresh, consider a 32gb DDR4 server memory system with $tRFC_{ab} = 880ns$. With 16 banks per channel, $tRFC_{pb} = 440ns$ is almost equal to time between per-bank refresh commands $tREFI_{pb} = tREFI_{ab}/16 = 487ns$. As a result, an entire bank in each chip is inaccessible almost all of the time. Even worse, for a 64gb server system $tRFC_{pb} = 600ns$ while $tREFI$ remains $487ns$ so per-bank refresh cannot maintain the same amount of refresh as all-bank refresh even by constantly refreshing an entire bank in memory. Chang et al. [13] propose Dynamic Access Refresh Parallelization (DARP) to improve per-bank refresh by selectively refreshing banks with fewer memory requests; however this does not solve the fact that at least an entire bank per rank is almost always inaccessible due to refresh. In comparison, refresh frequency is not an issue for Nonblocking Refresh where read access can proceed regardless of whether memory is refreshing.

5.2 Generality of Nonblocking Refresh

While we apply Nonblocking Refresh in the context of server memory systems, Nonblocking Refresh is applicable to DRAM-based memory systems in general. For example, desktop/laptop memory systems use the same rank architecture as server memory systems; therefore, they can perform Nonblocking Refresh by adding a redundant chip to the rank and then use the same Nonblocking Refresh implementation as we described for server memory systems.

Nonblocking Refresh is also applicable to memory systems that access only one DRAM chip per memory request, such as High Bandwidth Memory (HBM) and smartphone memory (i.e., LPDDRX DRAM), because the internal organization with each DRAM die mirrors a memory channel's organization. Consider HBM for example. There are multiple banks sharing a common data bus in each DRAM die [14], just like there are multiple ranks in a channel sharing a common data bus. There are also multiple sub-arrays per bank just like there multiple chips per rank [14]. In addition, each memory block is spread across multiple sub-arrays in one bank of a DRAM die, just like how a memory block is spread across a rank [14]. As such, HBM devices can implement Nonblocking Refresh by refreshing a portion of the sub-arrays in a bank at a time and adding redundant sub-arrays to each bank to compute the inaccessible data in refreshing sub-arrays.

In addition to improving raw system performance, avoiding read stalls due to DRAM refresh also reduces performance variability. Performance variability is a major concern for real-time systems because it complicates task scheduling. Conventional blocking DRAM refresh introduces a significant source of performance variability for real-time systems [7, 26]. As such, applying Nonblocking Refresh to the memory systems of real-time systems also provides an added benefit of simplifying task scheduling.

Chapter 6

Conclusion

Modern DRAM requires increasingly frequent refresh operations that block memory read requests and, therefore, slow down system performance. To effectively tackle the increasing performance overhead of memory refresh, many prior works have proposed skipping refresh operations; however, this can reduce security and reliability. A new solution is needed for systems with strict security and reliability requirements.

To effectively address increasing refresh latency without resorting to skipping refresh, we propose *Nonblocking Refresh* to refresh DRAM without stalling reads to refreshing memory blocks. Nonblocking Refresh works by refreshing only some of the data in a memory block at a time and uses redundant data, such as Reed-Solomon code, to compute the inaccessible data in the refreshing block to complete read requests. Compared to the conventional approach of refreshing all the data in a block at a time, Nonblocking Refresh makes up for refreshing only some of the data in a block at a time by operating more frequently in the background. Nonblocking Refresh transforms DRAM to behave like SRAM at the system-level by enabling DRAM to refresh in the background without stalling read requests to refreshing memory blocks.

For proof of concept, we apply Nonblocking Refresh to server memory systems, which value security and reliability. We observe that modern server memory systems contain redundant data to recover from memory chip failures; because this redundant data is budgeted for the worst-case memory hardware failure scenarios, a large fraction of the redundant data is not being used in the common case when there are no/little hardware errors to correct. As such, we propose utilizing the under-utilized redundant data in server memory systems to compute the inaccessible data stored in refreshing chips. Our evaluations show that on average across five server memory systems with hardware failure protection strengths, Nonblocking Refresh improves performance by 16.2% and 30.3% for 16gb and 32gb DRAM chips, respectively.

Bibliography

[1] Low power double data rate 4 - jedec. http://www.jedec.org/.

[2] Nas parallel benchmarks. http://www.nas.nasa.gov/publications/npb.html.

[3] Jedec standard ddr4 sdram, June 2017. https://www.jedec.org/sites/default/files/docs/JESD79-4.pdf.

[4] Barbara Aichinger. Ddr5: The new jedec standard for computer main memory, 2017. https://www.futureplus.com/ddr5-the-new-jedec-standard-for-computer-main-memory/.

[5] AMD. BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors, 2013. URL http://support.amd.com/TechDocs/42301_15h_Mod_00h-0Fh_BKDG.pdf.

[6] M. Arjomand, M. T. Kandemir, A. Sivasubramaniam, and C. R. Das. Boosting access parallelism to pcm-based main memory. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 695–706, June 2016. doi: 10.1109/ISCA.2016.66.

[7] B. Bhat and F. Mueller. Making dram refresh predictable. In *2010 22nd Euromicro Conference on Real-Time Systems*, pages 145–154, July 2010. doi: 10.1109/ECRTS.2010.23.

[8] I. Bhati, Z. Chishti, S. L. Lu, and B. Jacob. Flexible auto-refresh: Enabling scalable and energy-efficient dram refresh reductions. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 235–246, June 2015. doi: 10.1145/2749469.2750408.

[9] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011. ISSN 0163-5964. doi: 10.1145/2024716.2024718. URL http://doi.acm.org/10.1145/2024716.2024718.

[10] K. W. Cameron. Energy efficiency in the wild: Why datacenters fear power management. *Computer*, 47(11):89–92, Nov 2014. ISSN 0018-9162. doi: 10.1109/MC.2014.315.

[11] S. Cha, S. O, H. Shin, S. Hwang, K. Park, S. J. Jang, J. S. Choi, G. Y. Jin, Y. H. Son, H. Cho, J. H. Ahn, and N. S. Kim. Defect analysis and cost-effective resilience architecture for future dram devices. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 61–72, Feb 2017. doi: 10.1109/HPCA.2017.30.

[12] Karthik Chandrasekar, Christian Weis, Yonghui Li, Sven Goossens, Matthias Jung, Omar Naji, Benny Akesson, Norbert Wehn, and Kees Goossens. Drampower: Open-source dram power & energy estimation tool. http://www.drampower.info.

[13] K. K. W. Chang, D. Lee, Z. Chishti, A. R. Alameldeen, C. Wilkerson, Y. Kim, and O. Mutlu. Improving dram performance by parallelizing refreshes with accesses. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 356–367, Feb 2014. doi: 10.1109/HPCA.2014.6835946.

[14] B. Giridhar, M. Cieslak, D. Duggal, R. Dreslinski, Hsing Min Chen, R. Patti, B. Hold, C. Chakrabarti, T. Mudge, and D. Blaauw. Exploring dram organizations for energy-efficient and resilient exascale memories. In *2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12, Nov 2013. doi: 10.1145/2503210.2503215.

[15] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A remote software-induced fault attack in javascript. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*, DIMVA 2016, pages 300–321, New York, NY, USA, 2016. Springer-Verlag New York, Inc. ISBN 978-3-319-40666-4. doi: 10.1007/978-3-319-40667-1_15. URL http://dx.doi.org/10.1007/978-3-319-40667-1_15.

[16] Dave Hayslett. System z Redundant Array of Independent Memory. URL http://thebrainhouse.ch/gse/silvio/74.GSE/Silvio's%20Corner%20Doc%20Jukebox/System%20z%20Redundant%20Array%20of%20Independent%20Memory.pdf.

[17] Daniel Henderson. Power8 processor-based systems ras, October 2014.

[18] Micron Technology Inc. Speed vs. latency: why cas latency isn't an accurate measure of memory performance, 2015. https://pics.crucial.com/wcsstore/CrucialSAS/pdf/en-us-c3-whitepaper-speed-vs-latency-letter.pdf.

[19] Intel. Intel E7500 Chipset MCH Intel x4 SSDC, 2002. http://www.intel.com/content/www/us/en/chipsets/e7500-chipset-mch-x4-single-device-data-correction-note.html.

[20] ITIC. Itic 2015 - 2016 global server hardware, server os reliability report, 2015. http://www.lenovo.com/images/products/system-x/pdfs/white-papers/itic_2015_reliability_wp.pdf.

[21] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. Sgx-bomb: Locking down the processor via rowhammer attack. *Proceedings of the 2nd Workshop on System Software for Trusted Execution (SysTEX)*, October 2017.

[22] JEDEC Memory Specifications. Jedec memory specifications, 2004. http://www.jedec.org/.

[23] Uksong Kang, Hak soo Yu, Churoo Park, Hongzhong Zheng, John Halbert, Kuljit Bains, SeongJin Jang, and Joo Sun Choi. Co-architecting controllers and dram to enhance dram process scaling. *THE MEMORY FORUM*, 2014.

[24] Samira Khan, Donghyuk Lee, Yoongu Kim, Alaa R. Alameldeen, Chris Wilkerson, and Onur Mutlu. The efficacy of error mitigation techniques for dram retention failures: A comparative experimental study. *SIGMETRICS Perform. Eval. Rev.*, 42(1):519–532, June 2014. ISSN 0163-5999. doi: 10.1145/2637364.2592000. URL http://doi.acm.org/10.1145/2637364.2592000.

[25] Samira Khan, Chris Wilkerson, Zhe Wang, Alaa R. Alameldeen, Donghyuk Lee, and Onur Mutlu. Detecting and mitigating data-dependent dram failures by exploiting current memory content. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 27–40, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4952-9. doi: 10.1145/3123939.3123945. URL http://doi.acm.org/10.1145/3123939.3123945.

[26] H. Kim, D. Broman, E. A. Lee, M. Zimmer, A. Shrivastava, and J. Oh. A predictable and command-level priority-based dram controller for mixed-criticality systems. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 317–326, April 2015. doi: 10.1109/RTAS.2015.7108455.

[27] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 361–372, June 2014. doi: 10.1109/ISCA.2014.6853210.

[28] Y. Kim, W. Yang, and O. Mutlu. Ramulator: A fast and extensible dram simulator. *IEEE Computer Architecture Letters*, 15(1):45–49, Jan 2016. ISSN 1556-6056. doi: 10.1109/LCA.2015.2414456.

[29] A. Kumar and S. Sawitzki. High-throughput and low-power architectures for reed solomon decoder. In *Conference Record of the Thirty-Ninth Asilomar Conference onSignals, Systems and Computers, 2005.*, pages 990–994, October 2005. doi: 10.1109/ACSSC.2005.1599906.

[30] HP Labs. Cacti 6.5. http://www.hpl.hp.com/research/cacti/cacti65.tgz.

[31] Mark Lanteigne. How rowhammer could be used to exploit weaknesses in computer hardware. march 2016. http://www.thirdio.com/rowhammer.

[32] D. Lee, Y. Kim, G. Pekhimenko, S. Khan, V. Seshadri, K. Chang, and O. Mutlu. Adaptive-latency dram: Optimizing dram timing for the common-case. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 489–501, Feb 2015. doi: 10.1109/HPCA.2015.7056057.

[33] Shu Lin and Daniel J. Costello. *Error Control Coding, Second Edition*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2004. ISBN 0130426725.

[34] John D. C. Little and Stephen C. Graves. *Little's Law*, pages 81–100. Springer US,

Boston, MA, 2008. ISBN 978-0-387-73699-0. doi: 10.1007/978-0-387-73699-0_5. URL https://doi.org/10.1007/978-0-387-73699-0_5.

[35] Jamie Liu, Ben Jaiyen, Richard Veras, and Onur Mutlu. Raidr: Retention-aware intelligent dram refresh. *SIGARCH Comput. Archit. News*, 40(3):1–12, June 2012. ISSN 0163-5964. doi: 10.1145/2366231.2337161. URL http://doi.acm.org/10.1145/2366231.2337161.

[36] MICRON. 2Gb: x4, x8, x16 DDR2 SDRAM. *MICRON*, 2006.

[37] MICRON. 2Gb: x4, x8, x16 DDR3 SDRAM, 2006. https://www.micron.com/~/media/Documents/Products/Data%20Sheet/DRAM/DDR3/2Gb_DDR3_SDRAM.pdf.

[38] *8Gb: x4, x8, x16 DDR4 SDRAM.* Micron, 2015.

[39] Janani Mukundan, Hillery Hunter, Kyu-hyoun Kim, Jeffrey Stuecheli, and José F. Martínez. Understanding and mitigating refresh overheads in high-density ddr4 dram systems. *SIGARCH Comput. Archit. News*, 41(3):48–59, June 2013. ISSN 0163-5964. doi: 10.1145/2508148.2485927. URL http://doi.acm.org/10.1145/2508148.2485927.

[40] P. Nair, C. C. Chou, and M. K. Qureshi. A case for refresh pausing in dram memory systems. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 627–638, Feb 2013. doi: 10.1109/HPCA.2013.6522355.

[41] Minesh Patel, Jeremie S. Kim, and Onur Mutlu. The reach profiler (reaper): Enabling the mitigation of dram retention failures via profiling at aggressive conditions. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 255–268, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4892-8. doi: 10.1145/3079856.3080242. URL http://doi.acm.org/10.1145/3079856.3080242.

[42] M. K. Qureshi, D. H. Kim, S. Khan, P. J. Nair, and O. Mutlu. Avatar: A variable-retention-time (vrt) aware refresh for dram systems. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 427–437, June 2015. doi: 10.1109/DSN.2015.58.

[43] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: Approximate data types for safe and general low-power computation. *SIGPLAN Not.*, 46(6):164–174, June 2011. ISSN 0362-1340. doi: 10.1145/1993316.1993518. URL http://doi.acm.org/10.1145/1993316.1993518.

[44] S. Spoto, R. Gaeta, M. Grangetto, and M. Sereno. Bittorrent and fountain codes: friends or foes? In *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8, April 2010. doi: 10.1109/IPDPSW.2010.5470926.

[45] Vilas Sridharan, Jon Stearley, Nathan DeBardeleben, Sean Blanchard, and Sudhanva Gurumurthi. Feng shui of supercomputer memory: Positional effects in dram and sram faults. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 22:1–22:11, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2378-9. doi: 10.1145/2503210.2503257. URL http://doi.acm.org/10.1145/2503210.2503257.

[46] S. Wang, Y. Song, M. N. Bojnordi, and E. Ipek. Enabling energy efficient hybrid memory cube systems with erasure codes. In *2015 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 67–72, July 2015. doi: 10.1109/ISLPED.2015.7273492.

[47] Thomas Willhalm. Independent channel vs. lockstep mode - drive your memory

faster or safer, July 2014. https://software.intel.com/en-us/blogs/2014/07/11/independent-channel-vs-lockstep-mode-drive-you-memory-faster-or-safer.

[48] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in NAND ssds. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 15–28, Santa Clara, CA, 2017. USENIX Association. ISBN 978-1-931971-36-2. URL https://www.usenix.org/conference/fast17/technical-sessions/presentation/yan.