Detecting and Mitigating Data-Dependent DRAM Failures by Exploiting Current Memory Content

Samira Khan* Chris Wilkerson† Zhe Wang† Alaa R. Alameldeen† Donghyuk Lee‡ Onur Mutlu*
*University of Virginia †Intel Labs ‡Nvidia Research *ETH Zürich

Abstract

DRAM cells in close proximity can fail depending on the data content in neighboring cells. These failures are called data-dependent failures. *Detecting and mitigating these failures* online, *while the system is running in the field, enables various optimizations that improve reliability, latency, and energy efficiency of the system. For example, a system can improve performance and energy efficiency by using a lower refresh rate for most cells and mitigate the failing cells using higher refresh rates or error correcting codes. All these system optimizations depend on accurately detecting* every possible *data-dependent failure that could occur with any content in DRAM. Unfortunately, detecting* all *data-dependent failures requires the knowledge of DRAM internals specific to each DRAM chip. As internal DRAM architecture is not exposed to the system, detecting data-dependent failures at the system-level is a major challenge.*

In this paper, we decouple the detection and mitigation of data-dependent failures from physical DRAM organization such that it is possible to detect failures without knowledge *of DRAM internals. To this end, we propose* ***MEMCON****, a memory content-based detection and mitigation mechanism for data-dependent failures in DRAM. MEMCON does not detect* every possible *data-dependent failure. Instead, it detects and mitigates failures that occur* only *with* the current content in memory *while the programs are running in the system. Such a mechanism needs to detect failures whenever there is a write access that changes the content of memory. As detection of failure with a runtime testing has a high overhead, MEMCON selectively initiates a test on a write, only when the time between two consecutive writes to that page (i.e., write interval) is long enough to provide significant benefit by lowering the refresh rate during that interval. MEMCON builds upon a simple, practical mechanism that predicts the long write intervals based on our observation that the write intervals in real workloads follow a Pareto distribution: the longer a page remains idle after a write, the longer it is expected to remain idle.*

Our evaluation shows that compared to a system that uses an aggressive refresh rate, MEMCON reduces refresh operations by 65-74%, leading to a 10%/17%/40% (min) to 12%/22%/50% (max) performance improvement for a single-core and 10%/23%/52% (min) to 17%/29%/65% (max) performance improvement for a 4-core system using 8/16/32 Gb DRAM chips.

CCS CONCEPTS

• **Computer systems organization** → Processors and memory architectures; • **Hardware** → Dynamic memory;

KEYWORDS

DRAM, Data-Dependent Failures, System-Level Failure Detection and Mitigation, Performance, Energy, Fault Tolerance, Refresh, Retention Failures, Memory Systems, Reliability.

ACM Reference format

Samira Khan, Chris Wilkerson, Zhe Wang, Alaa R. Alameldeen, Donghyuk Lee and Onur Mutlu. 2017. Detecting and Mitigating Data-Dependent DRAM Failures by Exploiting Current Memory Content. In Proceedings of The 50th Annual IEEE/ACM International Symposium on Microarchitecture, Cambridge, MA, USA, October 14-18, 2017 (MICRO-50), 14 pages. https://doi.org/10.1145/3123939.3123945

MICRO-50, October 14-18, 2017, Cambridge, MA, USA

ACM. ISBN 978-1-4503-4952-9/17/10...$15.00.
DOI: https://doi.org/10.1145/3123939.3123945

1. Introduction

The continued scaling of DRAM process technology has enabled higher density DRAM by placing smaller memory cells close to each other. Unfortunately, the close proximity of cells exacerbates cell-to-cell interference, making cells susceptible to failures [30, 31, 32, 35, 39, 51, 52, 56, 59, 61, 62, 63, 66, 67, 72]. Recent works propose to detect and mitigate these failures in the field, while the system is under operation, as a way to ensure correct DRAM operation while still being able to continue the scaling of process technology. Such *system-level detection and mitigation* of DRAM failures provides better reliability, performance, and energy efficiency in future memory systems [7, 14, 17, 20, 24, 31, 32, 39, 45, 46, 51, 52, 53, 53, 55, 58, 61, 65, 67, 70, 74, 81, 82, 86]. For example, it is possible to improve system energy and performance by using a lower refresh rate, if we can detect the failing cells at a lower refresh rate and mitigate *only* those failing cells using a higher refresh rate [52, 53, 67, 70, 86]. Such ideas that require system-level detection and mitigation rely on *(i)* detecting *every* cell that can fail during the entire lifetime of the system and *(ii)* mitigating failures via a high refresh rate, ECC, and/or remapping of faulty cells to reliable memory regions [32, 51, 65, 70, 88].

Unfortunately, detection and mitigation of DRAM failures caused by cell-to-cell interference is quite challenging, as shown by prior works testing real DRAM chips [21, 31, 32, 39, 51, 67]. A prominent type of interference failure occurs depending on the data content in neighboring cells. Such failures are called *data-dependent failures* [31, 32, 39, 51, 67, 72]. Historically, data-dependent failures have been a major problem for manufacturing reliable DRAM cells since as early as

the Intel 1103, the first commercialized DRAM chip [2]. These failures are inherent to DRAM design as they are caused by the electromagnetic coupling between wires used to access DRAM cells [5, 31, 39, 51, 72]. Manufacturers detect these failures by exhaustively testing neighboring DRAM cells with data patterns that introduce enough cell-to-cell interference to cause failures and then either remap the failed bits or discard the faulty chips to mitigate/avoid the failures. However, detection of data-dependent failures in the system, while the DRAM chip is being used in the field, faces a challenge. The detection of data-dependent failures is closely tied to internal DRAM organization, which is different in each chip and usually not exposed to the system [29, 31, 45, 51, 67, 85]. Without exact knowledge of the internal design of a DRAM chip, it is extremely difficult to detect *all* failures (as discussed in detail in Section 2). As DRAM cells get smaller, more cells fail due to cell-to-cell interference, which poses a significant challenge to DRAM scaling and systems that rely on DRAM scaling for higher memory capacity and performance [30, 31, 39, 51, 56, 59, 61, 62, 63, 66, 72]. Handling such failures during the manufacturing time greatly impacts the testing time that directly increases the cost of DRAM [30, 56, 62, 72]. Therefore, it is important to enable efficient detection and mitigation of data-dependent DRAM failures in the system, during the online operations.

The **goal** of this work is to design a low-overhead detection and mitigation mechanism for data-dependent failures that can be implemented in the system *without* requiring any knowledge about the specifics of the internal DRAM design. We develop a DRAM-transparent mechanism based on the key observation that, in order to ensure correct operation of memory during runtime, it is *not* required to detect and mitigate *every possible* data-dependent failure that can *potentially* occur throughout the lifetime of the system. Instead, it is sufficient to ensure reliability against data-dependent failures that occur with only *the current data content in memory*. Leveraging this key insight, we propose **MEMCON**, a memory content-based detection and mitigation mechanism for data-dependent failures in DRAM. While the system and applications are running, MEMCON detects failures with the current content in memory. These detected failures are mitigated using a high refresh rate for rows that contain the failing cells. MEMCON significantly reduces the mitigation cost as the number of failures with current data content is less than the total number of failures with every possible data content. Using experimental data from real DRAM chips tested with real program data content, we show that program data content in memory exhibits 2.4X-35.2X fewer failures than *all possible failures* with any data content, making MEMCON effective in reducing mitigation cost.

One critical issue with MEMCON is that whenever there is a write to memory, content gets changed and MEMCON needs to test that new content to determine if the new content introduces any data-dependent failures. Unfortunately, testing memory for data-dependent failures while the programs are running in the system is expensive. Testing leads to extra memory accesses that can interfere with critical program accesses and can slow down the running programs. In order to design a cost-effective detection and mitigation technique, we make two critical observations. First, we observe that even though testing has some cost associated with it, it also provides a benefit as a memory row can be refreshed at a lower rate if no failure is found after testing. Consequently, the cost of testing can be amortized if the content remains the same for a long time, providing an opportunity to continue eliminating the unnecessary refresh operations. The longer the content remains the same, the higher the benefit from the reduced refresh operations. In this work, we provide a cost-benefit analysis of testing and show that the cost of testing can be amortized if consecutive tests in a row are performed at a minimum time interval of 448–864 ms, depending on the test mode, refresh rate, and DRAM timing parameters (Section 3). As testing is triggered by a write operation that changes the data content in memory, we refer to this minimum interval as *MinWriteInterval*.

Second, we observe that write intervals in real applications follow the Pareto distribution. A large number of writes occur within a very small interval, but the rest of the writes are spread far enough such that a significant fraction of a program's time is spent on intervals greater than *MinWriteInterval*. For example, we find that on average 81.5% of the total write intervals are spent on intervals greater than 1024 ms in 12 real-world applications, including popular photo, audio, and video editors and streaming applications (details in Section 4.1; our collected data is publicly available online [34]). This result demonstrates that MEMCON can amortize the cost of testing in real workloads.

Based on our model and analysis on write intervals, we propose a simple mechanism to predict the write interval length at each write. Our mechanism, *probabilistic remaining interval length predictor* (PRIL), builds upon a key property of the Pareto distribution of write intervals: the longer a page is idle (not written to), the longer it is expected to remain idle. Hence, the remaining write interval of a page can easily predicted by how long the page has been remained idle after a write.

We demonstrate that MEMCON leads to 64.7%-74.5% reduction in refresh operations compared to a system using an aggressive refresh rate (16 ms) to mitigate failures. However, extra read and write accesses during testing can interfere with memory-intensive applications. We show that performance impact of extra read-write accesses due to testing is minimal. Our evaluation on SPEC [80], STREAM [1], and server [84] benchmarks demonstrates that PRIL improves performance by 10%/17%/40% (min) to 12%/22%/50% (max) for a single-core system and 10%/23%/52% (min) to 17%/29%/65% (max) for a 4-core system using 8/16/32 Gb DRAM chips compared to a baseline system that uses an aggressive refresh rate.

This paper makes the following contributions:

- This is the first work to propose a system-level online data-dependent failure detection and mitigation technique that is *completely decoupled* from the internal physical organization of DRAM. Our online failure detection and mitigation technique, MEMCON, detects failures only for the current memory content of the applications running in the system.
- We analyze and model the cost and benefit of failure detection and mitigation with the *current memory content*. Our analysis demonstrates that the cost of testing for the current content can be amortized if consecutive tests in a row are

performed at a minimum time interval (between 448 and 864 ms according to our evaluation). As testing needs to be performed when data content changes with program writes, we refer to this minimum interval as *MinWriteInterval*.

- Based on our analysis of *write intervals* across a wide range of real applications, we find that the interval of writes in these applications follow a Pareto distribution and applications spend a significant amount of time on very long write intervals. As a result, MEMCON can *(i)* amortize the cost of testing and *(ii)* provide significant benefit by reducing the refresh rate during these long intervals.
- We propose PRIL, a simple mechanism that predicts the long write intervals based on the key property of a Pareto distribution that the amount of time a page is idle after a write can indicate how long it will remain idle in the future. Therefore, MEMCON only initiates testing for pages that has been idle (not written to) for a while and leads to 64.7%-74.5% reduction in refresh operations by using a lower refresh rate during the long write intervals.

2. Background and Motivation

DRAM Organization. A DRAM module is connected to an on-chip memory controller through a channel. The memory controller sends commands and addresses to DRAM through the channel and transfers data to and from DRAM. As shown in Figure 1, each module is hierarchically organized into multiple ranks, chips, and banks. For example, a typical DRAM module can have one rank with 8 chips, where each chip consists of 8 banks. A DRAM bank is organized as multiple 2D arrays of cells. Figure 1 shows a cell array within a bank, where all cells in a row are connected to a wire called *wordline*, and cells in each column are accessed through a vertical wire called *bitline*. While accessing DRAM, an entire row is read and latched into sense-amplifiers. A DRAM cell stores data as charge in a capacitor. As capacitor leaks charge over time, DRAM cells are periodically refreshed every 64 ms (32 ms at high temperature operational mode) to maintain data integrity [52].

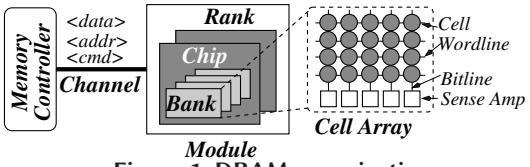


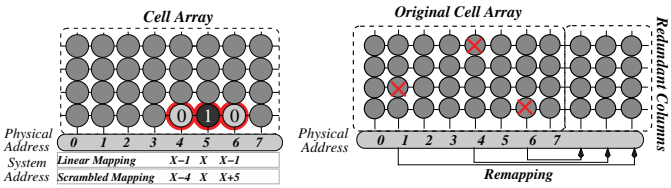
Figure 1: DRAM organization

Data-Dependent Failures. During a DRAM access, the parasitic capacitance between adjacent bitlines provide an indirect connecting path between neighboring cells [5, 72]. Therefore, neighboring cells can interfere each other, depending on the data content stored in the cells. DRAM failures that occur depending on the content of a cell and its neighboring cells are called *data-dependent failures* [31, 32, 39, 51, 72]. In order to detect data-dependent failures, manufacturers exhaustively test *neighboring DRAM cells* with data patterns that introduce enough cell-to-cell interference to cause failures. Experimental studies on DRAM chips demonstrate that a large fraction of DRAM cells exhibit data-dependent behavior in various failure modes [14, 32, 39, 51, 66]. As data-dependent failures occur when the interference is large enough to perturb the charge within the cells, these failures increase when DRAM operates at longer refresh intervals. It is easier to cause a failure perturbing the cells when the cell capacitor contains less charge at higher refresh intervals. Prior works demonstrate that the probability of data-dependent failures increases exponentially with lower refresh rates [66, 72].

Unlike traditional manufacturing time testing, some recent works propose to detect data-dependent failures online, at the system-level to enable system optimizations that improve reliability, latency, and energy efficiency of memory [31, 32, 51, 52, 67, 70]. These works propose to test every cell in a DRAM chip to detect *all possible* cells that are susceptible to data-dependent failures with *any possible data content* in memory. They propose to detect these cells with an initial testing phase and mitigate the failures with ECC, remapping, or higher refresh rates to ensure correct operation for *any content* in memory that can possibly occur during system operation. For example, RAIDR [52] detects such data-dependent failures using an initial test during the system boot-up and uses a low refresh interval to mitigate them, while it refreshes all other cells less frequently to improve performance and energy consumption by reducing the total number of refresh operations. However, there are two major challenges in detecting and mitigating data-dependent failures in the system.

(i) System-level failure detection is challenging due to unknown internal DRAM organization. There are two design issues in modern DRAM chips that make system-level failure detection particularly difficult.

First, DRAM vendors internally *scramble* the address space within DRAM. Neighboring addresses in the system address space do *not* correspond to neighboring physical cells [29, 31, 44, 45, 51, 85]. Consequently, testing neighboring cells for data-dependent failures by writing a specific data pattern in neighboring addresses at the system-level does *not* test adjacent cells in the DRAM cell array. Figure 2a shows one example of scrambled address space in DRAM. Neighbors of the cell at address **X** are expected to be located at adjacent addresses **X-1** and **X+1** with a regular linear mapping of physical address space to system address space. However, due to scrambling of the address space, neighbors of **X** are located at system addresses **X-4** and **X+5**. This internal address mapping from physical to system-level address is *not* exposed to the system, e.g., the memory controller [31, 44, 45, 51, 85]. To make things worse, vendors scramble the addresses differently for each generation of DRAM chips [31, 45].



(a) *Address scrambling* (b) *Column remapping*

Figure 2: Address scrambling and column remapping in DRAM

Second, DRAM vendors repair some of the faulty cells detected during manufacturing tests by remapping the faulty columns to available redundant columns in DRAM [22]. Fig-

ure 2b shows an example of remapping where three faulty columns at physical column addresses **1**, **4**, and **6** are remapped to the redundant columns located at the right of the original cell array. Due to this remapping, cells in remapped columns now depend on neighbors located in the redundant columns. For example, the neighbors of cells located at physical column address **1** are now located at physical addresses **4** and **7**. Remapping makes neighboring cell information different for each chip based on the location of faulty cells and remapped columns, making it necessary to design specific failure detection tests targeted for each individual chip in the system.

Vendors consider the internal DRAM design as proprietary information and do not expose it outside of the manufacturing organization. Even if the vendors expose the details of DRAM internals, scrambling of address space and remapped columns make system-level failure detection tightly coupled with each specific DRAM chip. Exposing such information efficiently for each chip and designing a generalized detection mechanism in the system that will work for all commercially available DRAM chips is rather challenging.

(ii) System-level failure mitigation is very expensive. It is expected that systems will have to mitigate a large number of failures in the future as cells become more vulnerable to cell-to-cell interference with DRAM scaling [39, 61, 66]. Prior works have shown that mitigating a *large* number of failures with ECC, remapping, or higher refresh rates adds a significant storage, performance, and energy overhead, making mitigation very expensive [32, 65].

The **goal** of this work is to design a low-overhead technique for detecting and mitigating data-dependent failures that does not require *any* knowledge about the internal DRAM organization. To this end, we propose to decouple the detection and mitigation of data-dependent failures from information about the internal organization of a DRAM chip.[1] We limit the scope of our mechanism to data-dependent DRAM failures, as other types of interference failures [39, 61, 66] and random failures [24, 58, 70] can be mitigated using ECC or orthogonal failure-specific mechanisms [11, 32, 39, 61, 65, 70, 88]. We also do not provide mechanisms to handle variation in data-dependent DRAM failures with the change in temperature. Prior works showed that it is possible to protect against these variations using well-known and experimentally-validated temperature models and adding an appropriate guardband to the mitigation technique [32, 46, 51, 67]. In the next section, we describe our DRAM-transparent online data-dependent failure detection and mitigation mechanism that relies only on the change in memory content during the execution of applications in the system.

3. MEMCON: Memory Content-Based Detection and Mitigation of DRAM Failures

In this work, we make the argument that it is *not* necessary to detect and mitigate *every possible* data-dependent failure that can potentially occur with *any* memory content during the lifetime of the system. Instead, it is sufficient to detect and mitigate the failures that can occur *only* with the *current content* in memory, stored by programs running in the system, and ensure a reliability guarantee that there will be *no failure* in the system with the *current memory content*. Doing so makes system-level detection and mitigation independent of the internal DRAM architecture as it eliminates the need to detect *every* susceptible cell that can fail due to *any data content* in memory.

3.1. *High-Level Design of MEMCON*

Based on this observation, we make a case for *memory content-based detection and mitigation* of data-dependent failures in DRAM, which we refer to as **MEMCON**. While the programs are running, MEMCON detects failures with the current memory content, and uses a higher refresh rate for faulty rows to mitigate those failures. Therefore, it detects failures dynamically while programs are running and mitigates *only* failures that can be triggered by the current data content of the programs.

As we will demonstrate, MEMCON *(i)* eliminates the need for detecting *every possible* data-dependent failure, which requires knowledge of DRAM internals, and *(ii)* reduces the mitigation cost as the number of failures with the current memory content is much smaller than the total number of failures with every possible combination of data content in memory.

To motivate MEMCON, Figure 3 experimentally demonstrates that cells fail conditionally depending on the memory content. In this figure, each dot on a vertical line represents a particular cell failing with some specific data content (i.e, data pattern) when a DRAM chip is tested with an FPGA-based infrastructure (details in Section 5). We observe that a particular cell may or may not fail depending on the data content in memory. As a result, instead of detecting every cell that can fail with any possible content, detecting and mitigating failures with just the current program content can significantly reduce the mitigation cost.

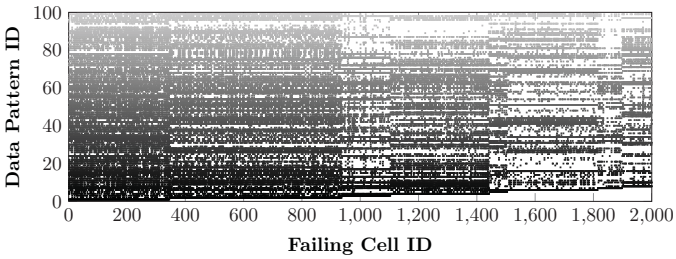


Figure 3: DRAM cells failing with different data content

Figure 4 shows the percentage of rows that contain data-dependent failures with *current memory content* compared to *every possible* data-dependent failure detected in memory. The fraction of the rows represents the number of failed rows for each workload when its memory footprint is duplicated in the entire DRAM module to ensure that the whole memory is occupied by program content. We present the percentage of failing rows (averaged over every 100 million instructions across a set of representative phases [69]) for 20 SPEC CPU 2006 [80] benchmarks.[2] This figure shows that only 0.38%-

[1] Note that detection of retention failures that occur due to *weak cells* that always fail *irrespective* of the data content is *not* a major problem. These failures can be easily detected as these cells fail *every time* a chip is tested.

[2] Even if we demonstrate the fraction of failing rows for only SPEC benchmarks, we believe that other workloads with larger working sets will also exhibit similar results. The number of failures does *not* depend on the size of the working set, but depends on the data content in memory.

5.6% of the rows encounter failures with the program content, where 13.5% of all rows encounter failures when tested with any possible memory content (represented by ALL FAIL in the figure). Thus, the number of failures is 2.4X-35.2X less with the current program content than with every possible content.

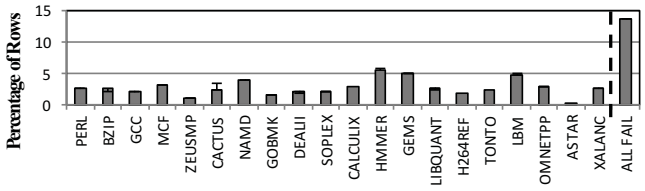


Figure 4: Percentage of rows that exhibit failures

3.2. *Design Challenge of MEMCON*

MEMCON ensures correct DRAM operation by detecting and mitigating possible failures with the *current* memory content. As long as the content remains the same, MEMCON comfortably ensures reliability since it provides the right level of protection (e.g., the appropriate refresh rate) for that content. Read accesses do not alter memory content and therefore, cannot introduce any new failures. However, whenever there is a write to a row in memory, content gets changed and MEMCON needs to *(i)* test that new content to detect whether or not that content introduces any data-dependent failure and *(ii)* if so, find the right level of protection for the rows failing with the new content.

Unfortunately, detecting data-dependent failures *while* the programs are running in the system could cause performance degradation. Testing current memory content for detecting data-dependent failures involves keeping the the row that is being tested (i.e., the *in-test row*) idle until the end of the refresh period so that cells in that row are tested with the lowest possible charge, when they are the most vulnerable to cell-to-cell interference. As any access to a row fully charges all cells in that row, program accesses during the testing period *cannot* be serviced from the in-test row and have to be serviced by temporarily buffering the content of the row in a different region. Therefore, there are two sources of overhead in detecting data-dependent failures with current memory content. First, testing involves reading the row content into the memory controller and buffering the read content of the row, keeping the row idle for the test period, and reading the entire row again to compare with the buffered row content to determine if there are any data-dependent failures. As a result, all the cache blocks in the in-test row have to be read at least twice to compare their contents before and after the test. Second, temporarily buffering the content of the in-test row in some other region (either inside the memory controller or inside memory) involves extra read and write traffic to copy the in-test row to that region. These additional read and write requests for testing purposes increase bandwidth consumption and can interfere with critical program accesses. The key design challenge MEMCON is to minimize the overhead of the testing.

3.3. *Cost-Benefit Analysis of MEMCON*

As testing at the system-level is expensive, it is necessary to analyze the cost-benefit of MEMCON on application runtime to demonstrate its effectiveness. The cost of testing (i.e., its performance and/or energy overhead) depends on the extra memory requests issued for testing. The benefit of testing comes from our technique of optimizing the refresh rate based on the results of testing: MEMCON initially refreshes each row very frequently to avoid any failure; after the row content is tested and no failure is detected for a row, that row is refreshed at a lower rate. Therefore, the benefit of testing comes from the reduction in refresh operations enabled by testing.

Figure 5 examines the tradeoff between the cost of testing (in terms of latency) and the frequency of testing of a single row to demonstrate that the cost of frequent testing can potentially outweigh the benefit of testing. There are three different costs associated with MEMCON: *(i)* Without any testing, all rows have to be refreshed aggressively so that failures do not get exposed to running applications. This cost of aggressive refresh (i.e., the periodic latency required to refresh a row) is represented as the ***HI-REF*** state in the figure. *(ii)* The cost of testing (i.e., the latency required for extra read-write accesses), which is represented as the highest bar in the figure, labeled as ***TESTING***. The figure shows that testing is more expensive than the ***HI-REF*** state due to the extra read-write accesses incurred for testing. *(iii)* When testing for data-dependent failures is done and no failures are found in a tested row, the row can be refreshed less frequently. This low-cost refresh state (the latency required for infrequent refresh operations) is represented as ***LO-REF*** in the figure.

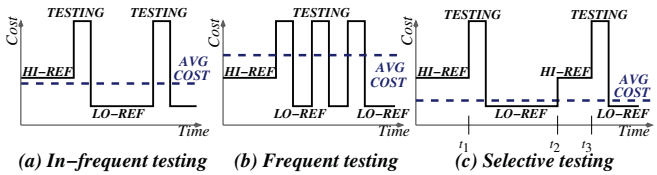


Figure 5: Tradeoff between frequency of testing vs. cost

Figure 5 demonstrates the average cost of MEMCON over some specific period of time, when testing is performed at different rates for a row. First, if testing is infrequent (i.e., writes to a row are separated far apart in time), the benefit of the ***LO-REF*** state overshadows the cost of testing, such that the average cost remains equal to or lower than that of the ***HI-REF*** state (shown in Figure 5(a)). In this case, a longer interval between two consecutive tests leads to a higher benefit and the average cost gets lower. Second, as testing is very expensive, frequent testing can increase the average cost to a level higher than the ***HI-REF*** state (Figure 5(b)). In this case, it is better to just use frequent refreshes (***HI-REF***), instead of detecting any failures, to minimize the average cost. Therefore, there is a trade-off between the cost of testing vs. the frequency of testing. In order to minimize the overall cost (and thus maximize the benefit of testing), MEMCON should *not* initiate a test every time there is a new write to a row. Instead, it should test the row on a write, *only when* the cost of testing can be amortized by the future infrequent refreshes to the tested row. Therefore, MEMCON should skip testing for cases where the interval between two consecutive writes to a row is *not* large enough to amortize the cost of testing.[3] Such selective testing

[3]MEMCON can be further optimized by eliminating testing if the row gets read frequently enough such that it does not need refresh. We leave such an optimization for future work.

for MEMCON is illustrated in Figure 5(c). MEMCON tests the row at t_1, as the interval between two writes (t_1 and t_2) is large enough to amortize the cost of testing. It skips testing for the write that arrives at t_2, as the interval between t_2 and t_3 is too small. Instead, MEMCON moves to the ***HI-REF*** state during that interval to avoid the high cost associated with testing.

In order to determine the minimum interval between two consecutive writes to a row that amortizes the cost of testing, we compare the total cost for two configurations: *(i)* when a row is refreshed aggressively (always at the ***HI-REF*** state), and *(ii)* with MEMCON, i.e., when a row is tested and then refreshed at a lower rate if appropriate (***LO-REF*** state only after testing). The total accumulated cost would increase linearly with time, as rows are refreshed periodically in both cases. However, initially, MEMCON would have a higher cost than ***HI-REF*** because of the high cost associated with testing. As MEMCON moves to the low refresh state after testing, its total cost would increase at a slower rate compared to the ***HI-REF*** configuration. The point in time when the total cost of ***HI-REF*** would become higher than the total cost of MEMCON indicates the time interval between two writes that can amortize the cost of testing. We refer to this interval as *minWriteInterval* in this work.

Figure 6 shows the total accumulated cost over time (in terms of latency) for both MEMCON and the ***HI-REF*** configuration. In order to determine the *minWriteInterval*, we model the cost of ***HI-REF*** and MEMCON based on the latency required to perform refresh and read-write operations. The ***HI-REF*** configuration refreshes the rows every 16 ms, which is 4X more frequent than the typical refresh interval in modern DRAM devices.[4] Thus, the cost of refresh in terms of required latency per row is 39 ns for every 16 ms (details in Appendix). Therefore, the cost for ***HI-REF*** increases sharply with time (represented as the red line in the figure). In this work, we compare two modes of testing for MEMCON (Read and Compare vs. Copy and Compare), based on where data content is buffered to serve accesses during the test. The cost of testing for each mode determines the frequency of testing that can amortize the cost.

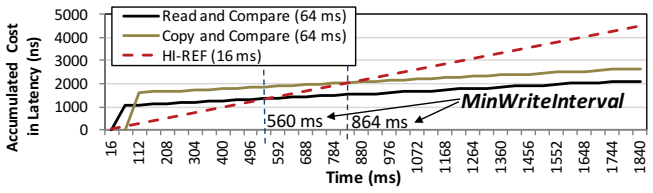


Figure 6: Determining *MinWriteInterval*

Read and Compare. In this mode, an in-test row is buffered in the memory controller. This mode involves *reading* the entire row into the memory controller twice (once before the test and once after the test) to *compare* the old and new content and determine any occurrence of failure. The latency of reading an entire 8K row twice from memory is 1068 ns (details in Appendix).

Copy and Compare. One problem with the prior test mode is that testing a large number of rows simultaneously requires a large buffer in the memory controller. As an alternative, in this second mode of testing, the contents of the in-test rows are temporarily stored in a different, special region of memory to service requests to in-test rows during the test. In this mode, the in-test row is *copied* to another region in memory by reading the row into the memory controller and then writing it to that special region in memory. Only the ECC information is calculated and stored in the memory controller. After the test, the content of the row is read back again into the memory controller to *compare* the old and new ECC values to determine any occurrence of failure. As a result, this Copy and Compare mode involves reading the entire row into the memory controller twice (once before the test and once after the test) and writing the entire row once into a new location. The cost for Copy and Compare in terms of latency is 1602 ns (details in Appendix).[5] [6]

Figure 6 shows that both of these test modes pay the high latency cost of testing in the beginning (1068 ns and 1602 ns, respectively). After that, the system is refreshed once every 64 ms.[7] Therefore, the total cost increases more slowly over time compared to that of the ***HI-REF*** configuration, where rows are refreshed every 16 ms all the time. The figure shows that the total cost of testing becomes lower than the cost of the ***HI-REF*** configuration, if the system can be at the ***LO-REF*** state for at least 560 ms and 864 ms, respectively for Read and Compare and Copy and Compare test modes. Thus, the MinWriteInterval should be 560 ms/864 ms for these two configurations. We also evaluate the *MinWriteInterval* for the ***LO-REF*** state with a refresh interval of 128 ms and 256 ms, found it to be 480 ms and 448 ms, respectively.

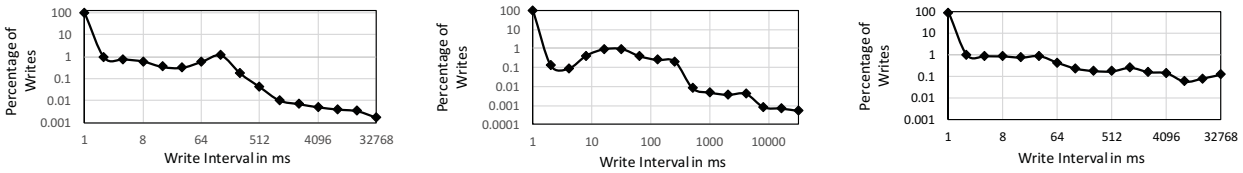
We conclude that MEMCON can amortize the cost of testing if tests are done at a minimum interval of 448-864 ms, depending on the combination of test mode and refresh interval. Note that the *MinWriteInterval* is quite long, almost half a second to a full second. This requirement for *long* write intervals implies that MEMCON would provide benefit *only* when the applications spend a large fraction of their execution time on long write intervals. In the next section, we analyze the write intervals in real applications and, based on our analysis, propose a simple prediction mechanism to determine when to initiate a test such that the cost of testing is amortized.

[4] Modern DRAM chips use frequent refresh operations to ensure data integrity (e.g., LPDDR2 uses 32 ms refresh intervals at 85°C [26], DDR3/4 uses 32 ms refresh intervals at 95°C [27, 28]). As DRAM scaling is getting worse, we believe that a 16 ms refresh interval is reasonable for future DRAMs. However, we also compare our results with a less aggressive baseline of a 32 ms refresh interval (Figure 16 in Section 6.2).

[5] The storage overhead of this mode, i.e., the amount of memory required to copy the in-test rows to, is modest as only a small fraction of rows are tested concurrently. For example, reserving 512 rows per bank in a 2 GB module with 8 banks, results in only a 1.56% loss in DRAM capacity; details in Appendix. This mode also requires memory requests to the in-test rows to be redirected by the memory controller to the appropriate region of memory, which can be accomplished with little storage overhead.

[6] Note that this mode can become significantly faster by *(i)* performing copy operations completely within DRAM, using mechanisms like RowClone [76] and LISA [16], *(ii)* exploiting subarray-level parallelism [41], tiered-latency DRAM [47], or a Dual-Data-Port DRAM [42], or *(iii)* performing comparison operations inside DRAM or in the logic layer of 3D-stacked memory [3, 4, 23, 75, 77, 78, 79]. We leave the evaluation of such optimizations for future work.

[7] We use 64 ms refresh interval is this figure, but we also report summary results for 128 ms and 256 ms later (at the end of the same paragraph).



(a) *ACBrotherhood* (b) *Netflix* (c) *SystemMgt*

Figure 7: Distribution of write intervals in three representative workloads

4. Write Interval Prediction for MEMCON

MEMCON selectively initiates a test on a write to a page, only when the time interval between two consecutive write requests to the page is predicted to be long enough (i.e., at least 448-864 ms, as shown in Section 3.3). Therefore, when there is a write access to a page, MEMCON *(i)* predicts the write interval and *(ii)* if the interval is predicted to be long enough to amortize the cost of testing, tests the page with the current content. In this section, we first demonstrate the characteristics of the write intervals in real applications (Section 4.1). Based on those characteristics, we propose a simple mechanism to predict the write intervals (Section 4.2).

4.1. Write Interval Characteristics in Real Applications

We use an FPGA-based memory tracing system, similar to HMTT [8], to track the characteristics of writes to each page in various real programs. These traces span several minutes of execution time after the initialization phase. Our evaluated applications include popular games, video editors and players, system management software, video streaming applications, and others (see Table 1; details of our methodology in Section 5).

Figure 7 demonstrates the distribution of write intervals for three representative workloads: ACBrotherhood, Netflix, and System Management (Win 7). The x-axis represents different write intervals, from 1 ms to 32768 ms, and the y-axis represents the percentage of writes that are within the corresponding write interval range. We make two observations from this figure. First, a large majority (more than 95%) of the writes occur within 1 ms. Fortunately, writes within 1 ms do *not* trigger testing because such frequent accesses naturally *refresh* the row *before* the refresh interval and therefore, do *not* cause any data-dependent failures. Thus, our mechanism is *not* affected by such short-interval writes. Second, only a small fraction of the writes exhibit write intervals that are equal to or greater than the *MinWriteInterval* that can amortize the cost of testing. For example, on average, less than 0.43% of the total number of writes exhibit write interval lengths greater than 1024 ms. We refer to such a write interval with length greater than 1024 ms, where there is significant opportunity to lower the refresh rate for a long time, as a *long write interval*. Even though writes with long intervals constitute a small fraction of the *total* number of writes, the corresponding write intervals are *very long*. As a result, our programs spend a very large fraction of their execution time in these long write intervals. We demonstrate this phenomenon in Figure 9, which shows the fraction of the execution time each workload spends on long write intervals. We observe that, on average, write intervals greater than 1024 ms constitute 89.5% of the total time spent on write intervals.

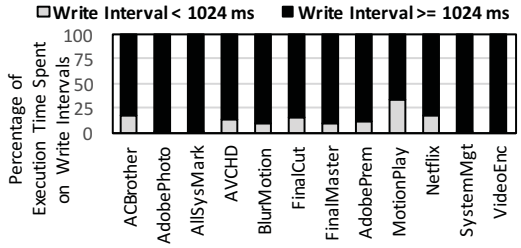
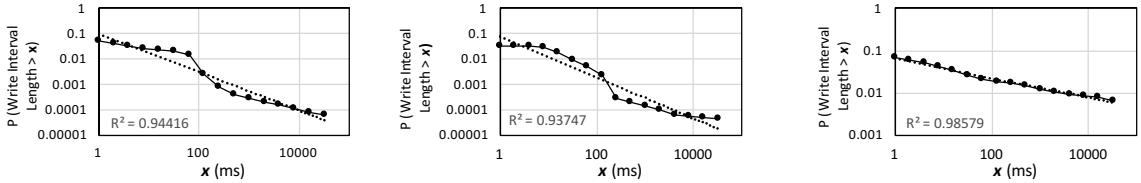


Figure 9: Execution time is dominated by long write intervals (i.e., write intervals with length greater than 1024 ms)

Our empirical observations on write intervals indicate that write intervals in real applications follow a Pareto distribution [6]. The intervals exhibit a heavy-tailed property, where the number of long intervals are very few, but these long write intervals comprise a very large fraction of the total execution time. Mathematically, a Pareto distribution of the write intervals means that the probability of the write interval being greater than x follows this function: *Probability (Write Interval Length* $> x) = k.x^{-\alpha}$, where $k, \alpha > 0$ are constants. Figure 8 demonstrates that the probability distribution of write interval lengths in three representative workloads closely follow this equation: the plotted curves match a linear fit on the log-log scale with very high R^2 values, indicating that the Pareto distribution is indeed a good fit. Various system properties have been shown to follow a Pareto distribution, such as system job lengths [19, 73], Unix process lifetimes [19], sizes of files and web traffic [18, 68, 73], lengths of memory episodes [40], and memory errors in servers [58].

In this work, we leverage the decreasing hazard rate (DHR) property [9] of a Pareto distribution to predict the write interval



(a) *ACBrotherhood* (b) *Netflix* (c) *SystemMgt*

Figure 8: Pareto distribution of write intervals in three workloads

on a write access. This property implies that the elapsed time in an interval can be used to predict the remaining length: the longer a page is idle (i.e., not written to) after a write, the longer it is expected to remain idle (i.e., the more likely it has a long write interval). On the other hand, if the elapsed time after a write to a page is short, it is expected that the write interval will end soon.

We define two parameters for a write interval, as shown in Figure 10. *(i) Current interval length* (CIL), the elapsed time after the latest write to the page at the current time T, and *(ii) remaining interval length* (RIL), the remaining length of the write interval. We show that the probability that RIL is long increases with a longer CIL in real applications (Figure 11). For this experiment, we conservatively set the RIL to be 1024 ms as the *MinWriteInterval* required to amortize the cost of testing ranges from 448-864 ms. Therefore, Figure 11 illustrates the probability that the *remaining interval length* is greater than 1024 ms when we vary the *current interval length* from 1 ms to 32768 ms. This figure shows that the probability that the *remaining interval length* is greater than 1024 ms is very low when the *current interval length* is within 1 to 256 ms. However, the probability increases with increasing *current interval length* and becomes approximately 50%-80% at a *current interval length* of 512 ms. The probability approaches 1 when the *current interval length* is higher than 16384 ms. Based on this observation, we can devise a mechanism that predicts the RIL, after it has observed the length of the CIL, for a given page. For example, if such a mechanism predicts that the RIL would be greater than 1024 ms, when it has already observed that the CIL so far has been 512 ms, the probability of misprediction would be relatively low. We conclude that the probability distribution we observe for write interval lengths (shown in Figure 11) can be used to predict the long write intervals for each page *only* by tracking the length of the *current write interval*.

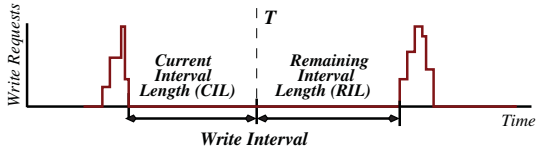


Figure 10: Terminology used for write intervals

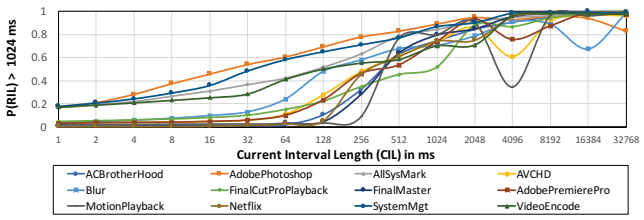


Figure 11: Probability that *RIL* is greater than 1024 ms, as a function of *CIL*

A longer CIL provides better accuracy, but at the same time reduces the length of RIL. Thus, there is a tradeoff between how long we should wait to observe the length of CIL to avoid mispredictions vs. how much opportunity we will lose to reduce the refresh rate if we wait longer. We define the *coverage* of our prediction as the fraction of the time spent in write intervals that has been correctly predicted. A higher CIL increases accuracy, but reduces coverage, e.g., waiting for the CIL to reach 32768 ms to predict that the RIL is greater than 1024 ms would provide a high-accuracy prediction; however, it would lead to a low coverage. Figure 12 shows the coverage of the time spent in write intervals when *current interval length* varies from 1 ms to 32768 ms. We observe that the coverage of total write interval time becomes lower as we increase CIL. A CIL of 512-2048 ms provides a prediction accuracy of 50-80% with a reasonable coverage (on average 65-85%). Based on these observations, we conclude that it is possible to devise a simple write interval prediction mechanism that achieves both high accuracy and high coverage, simply by waiting for a specific amount of time after a write (e.g., 512-2048 ms as the *current interval length*).

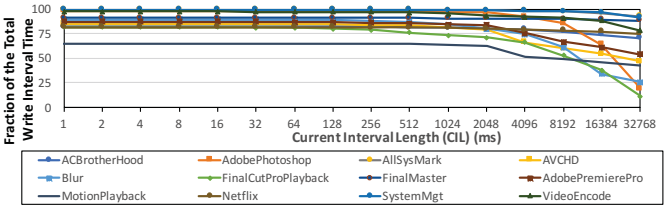


Figure 12: Coverage of the time spent in the write intervals when P(RIL) > 1024 ms, as a function of *CIL*

4.2. Probabilistic Remaining Interval Length (PRIL) Prediction

In order to exploit long write intervals for lowering the refresh rate, we propose a simple mechanism to predict the write interval after each write. Our mechanism, *probabilistic remaining interval length prediction* (PRIL), builds upon our observation that if the elapsed time after a write to a page (i.e., the *current interval length*), has been long enough, it is highly likely that the *remaining interval length* would be long. Therefore, it is possible to predict the the *remaining interval length* of a write interval, just by measuring how much time has elapsed after the last write to the page (i.e., since the write interval started). However, tracking the elapsed time after the last write for each page using a counter can introduce high overhead. To avoid such overhead, we propose a simple mechanism that tracks writes to pages across coarse-grained time quanta. The key idea of PRIL is to divide the execution time into fixed-length quanta, where the quantum size is set to be equal to a *current interval length* that provides a high-accuracy and a high-coverage prediction (e.g., 1024 ms), and detect the pages that remain *idle* (i.e., without writes) for at least one quantum.

High-Level Design. We describe *one* implementation of PRIL that aims to minimize the hardware overhead while achieving most of the opportunity available from an ideal prediction mechanism. In order to detect the pages that receive no writes for at least one quantum, PRIL keeps track of the pages (i.e., row addresses) that *(i)* have been written to *only once* during each quantum and *(ii)* remain idle (i.e., receive no writes) in the *next* quantum.[8] This mechanism ensures that

[8]Note that tracking the pages that are written to *only once* is a design choice we make to limit the number of page addresses we track in the write buffer. Removing pages that are written to multiple times during a time quantum greatly reduces the number of addresses MEMCON tracks while minimally affecting the accuracy of our prediction mechanism. The effect on accuracy is minimal because pages that are written to multiple times within a short interval are unlikely to have long write intervals.

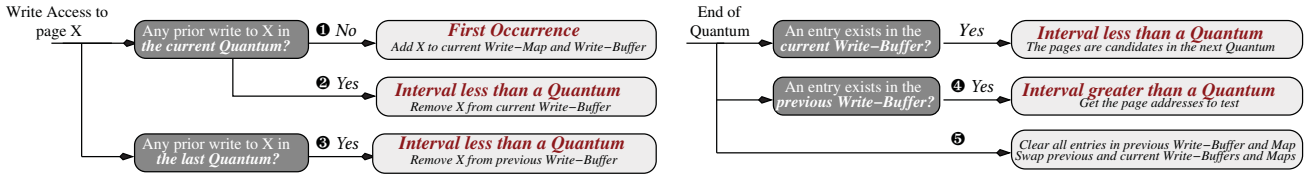


Figure 13: Workflow for PRIL: Steps associated with (i) the arrival of a write and (ii) the end of a quantum

the elapsed time after the write is greater than the quantum length (even if the write occurs at the very last moment in the first quantum). Therefore, PRIL uses two consecutive quanta to detect the long-interval writes. At the end of the second quantum, all the pages that have received only one write in the first quantum and no writes in the second quantum, are predicted to have a long *remaining interval length* and MEMCON initiates testing for those pages.[9]

Implementation. Figure 13 shows the operations that take place in each quantum. PRIL uses two structures to keep track of the pages that receive only one write access during one quantum. First, it uses a bit vector (called *write-map*) to mark the pages that receive *at least* one write access during a quantum. The appropriate bit in the *write-map* is set when there is a write access to a page. Second, it uses a *write-buffer* to store the addresses of the pages that have been written to *only once* during that quantum. When there is a write access to a page, the address of that page is inserted in the *write-buffer* only if the corresponding bit in the *write-map* indicates that this is the first write to that page (indicated by ❶ in Figure 13).[10] Otherwise, if the *write-map* indicates that the page has already received at least one write access before in the current quantum (i.e., the corresponding bit in the *write-map* is set), that address is deleted from the *write-buffer*, because the write interval is clearly less than the quantum length (indicated by ❷).As PRIL tracks two consecutive quanta to determine the pages that have *current interval length* greater than the quantum length, there are two *write-maps* and two *write-buffers* in PRIL: the *current write-map* and the *current write-buffer* track the occurrence of writes in the *current quantum*, where as the *previous write-map* and the *previous write-buffer* have already been tracking the addresses of the pages that have received only one write access during the *last quantum*. PRIL deletes any address from the *previous write-buffer*, if that address receives a write access in the *current quantum* (indicated by ❸). As a result, the entries in the *previous write-buffer* contain only the addresses of the pages that have received only *one* write access in the last quantum and *no* write access in the current quantum; these pages become candidates for testing at the end of a quantum (indicated by ❹). At the end of each quantum, MEMCON *(i)* gets the recorded page addresses from the *previous write-buffer* to initiate testing on them and *(ii)* clears all entries in the *previous write-buffer and write-map*. Then, the current and previous *write-buffers and maps* are swapped, and the writes in the next quantum start being tracked by the *current write-buffer and current write-map* (as indicated by ❺). We discuss the storage overhead of PRIL in Section 6.4. We leave the exploration of cheaper implementations of PRIL for future work.

5. Methodology

In this paper, we demonstrate the feasibility of MEMCON using three different evaluation infrastructures. First, we use an FPGA-based DRAM testing infrastructure based on SoftMC [21], to test real DRAM chips for data-dependent failures (results shown in Section 3). Second, we use an FPGA-based testing infrastructure to collect memory traces from long-running applications (results in Section 4). Third, we use a cycle-accurate simulator [36, 37] to study the impact of MEMCON on system performance (Section 6). Next, we discuss our methodology for each of these evaluation techniques.

FPGA infrastructure to test real DRAM chips. We use an infrastructure based on SoftMC [21], as done in a variety of DRAM characterization studies [12, 13, 14, 17, 31, 32, 38, 39, 43, 45, 46, 51, 67, 70]. Our infrastructure provides the ability to: *i)* dump application memory content in the test DRAM chip with different refresh rates, *ii)* provide an interface from a host machine to the FPGA test infrastructure. We use a Xilinx ML605 board [87] that includes an FPGA-based memory controller connected to a DDR3 SO-DIMM socket. The FPGA board is connected to the host machine using the PCIe bus. We run experiments on DRAM chips in a temperature-controlled environment. We test DRAM chips with a refresh interval of 4 s at 45°C, which corresponds to a refresh interval of 328 ms at 85°C [51]. There are three steps involved in our tests: *(i)* Generate the memory content traces for 20 SPEC CPU2006 benchmarks [80] and replicate the content in the tested DRAM chip such that the entire memory gets filled up with the collected real data content from a specific time. *(ii)* Keep memory idle for the refresh interval and *(iii)* Read back the content to determine the failing content. While a program executes in the system, data content in memory changes over time as dirty cache blocks get written back to memory. We generate memory content traces for each workload at every 100 million instructions and test DRAM chips to determine the number of failures with each data content trace. We report the average percentage of failing rows over 0.5 billion instructions (with the maximum and minimum shown as the error bars) in Figure 4 in Section 3.

FPGA infrastructure to generate memory traces for long-running applications. We use another FPGA-based infrastructure, similar to HMTT [8], to generate memory traces from real applications. This infrastructure intercepts the memory bus and tracks commands, associated addresses, and timestamps of every memory request sent to DRAM. The traces span several minutes of application runtime (as shown in Ta-

[9] MEMCON can become even more effective if we can recognize silent writes to a page (i.e., writes that do not change the value in memory) [10, 48, 49, 50] and not initiate testing on such writes.

[10] If the write buffer is full, we discard the new page and set its refresh state to HI-REF. By doing so, MEMCON loses the potential opportunity to reduce the refresh rate for that page, but this does not affect correctness as a discarded page will be refreshed at the ***HI-REF*** state.

ble 1) and are collected after a fixed time interval passes to avoid tracing an application's initialization phase. We use this infrastructure to track the write intervals in some popular applications taken from different domains, such as gaming, video editing, playback, and streaming, photo editing, system management. Table 1 provides the names and characteristics of these long-running applications. We made the write interval characteristics of these traces publicly available online [34].

Application	Type	Time (s)	Mem (GB)	Threads
ACBrotherHood	Game	209.1	2.8	8
AdobePhotoshop	Photo editing	149.2	3.0	4
AllSysMark	Media creation	2064	3.4	4
AVCHD	Video playback	217.3	5.2	2
BlurMotion	Image processing	93.4	0.2	2
FinalCutPro	Video editing	76.9	3.0	2
FinalMaster	Movie display	248.1	2.0	2
AdobePremiere	Video editing	298.8	5.0	2
MotionPlayBack	Video processing	233.9	5.6	2
Netflix	Video streaming	229.4	4.6	2
SystemMgt	Win 7 managing	466.2	7.6	2
VideoEncode	Video encoding	299.1	7.3	4

Table 1: Evaluated long-running workloads

Cycle-accurate simulation. We use a cycle-accurate open-source simulator, Ramulator [36, 37], to evaluate the performance impact of MEMCON. It is driven by a frontend based on Pin [54]. It is not possible to perform a cycle-accurate simulation of *minutes* of actual execution time. As a result, we get the reduction in refresh rate from the write interval characteristics of the long-running traces for MEMCON and model the corresponding refresh rate with the overhead of testing to evaluate MEMCON using our simulator. We use multiprogrammed workloads that contain benchmarks from SPEC CPU2006 [80], TPC-C and TPC-H [84]. We combine 4 randomly-selected applications from these benchmark suites to generate 30 multi-programmed workload mixes for our multi-core evaluations. Table 2 shows our system configuration.

Processor	1-4 cores, 4GHz, 4-wide, 128-entry instruction window
Last-Level Cache	64B cache-line, 16-way associative 512KB private cache-slice per core
Main Memory	8GB DIMM DDR3-1600 (800MHz clock rate, 1.25ns cycle time) Baseline (t_{REFI}/t_{RFC}): 1.95us/350ns MEMCON: t_{REFI}: LO-REF 7.8us, HI-REF 1.95us MEMCON: t_{RFC}: **530/890/1600ns (16/32/64Gb)**

Table 2: Evaluated system configuration

6. Results

We first evaluate the effect of MEMCON on refresh reduction (Section 6.1) and performance (Section 6.2). Then, we compare the performance of MEMCON to other refresh optimizations (Section 6.3). Finally, we provide an analysis of our write interval prediction mechanism, PRIL (Section 6.4).

6.1. Reduction in Refresh Operations

In this section, we demonstrate that MEMCON is effective at reducing the number of refreshes by identifying the long write intervals in programs when a row can be refreshed less frequently (in the ***LO-REF*** state). We evaluate MEMCON's reduction in refresh operation count compared to a baseline system that refreshes every row aggressively at a refresh interval of 16 ms. MEMCON uses a 64 ms refresh interval (***LO-REF*** state) for rows that are *(i)* identified as read-only, and *(ii)* predicted to be idle (i.e., not written to) for more than 1024 ms after a write. The remaining rows are refreshed at a 16 ms interval (***HI-REF*** state). If all rows were refreshed at a 64 ms interval (***LO-REF*** state), MEMCON would reduce the refresh count by 75% (which is the *upper bound* refresh reduction achievable with the ***HI/LO-REF*** parameters we evaluate).

Figure 14 shows MEMCON's reduction in refresh count with three different values of *current interval length (CIL)*: 512, 1024, and 2048 ms. With these three values, PRIL provides both high accuracy and high coverage when predicting the long write intervals (as shown in Figure 12). We make two observations from Figure 14. First, on average, MEMCON's reduction in refresh count comes very close to the upper bound (75%). Depending on the characteristics of the applications, the reduction ranges from 64.7% to 74.5%. This result implies that our simple prediction mechanism, PRIL is very effective at identifying the long write intervals. Second, we observe that the reduction in refresh count does *not* change significantly when we vary CIL from 512 ms to 2048 ms. This is because a significant fraction of the execution time is dominated by very long write intervals that are on the order of minutes and a small variation in CIL (in the range of milliseconds) *does not* affect the coverage of write intervals in a significant way (as discussed in Section 4.1). From these observations, we conclude that MEMCON achieves a refresh reduction that is very close to the upper bound.

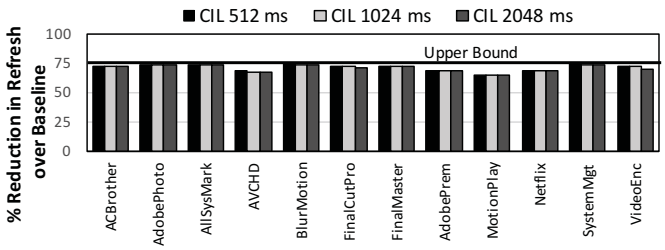


Figure 14: Reduction in refresh count with MEMCON

6.2. Performance Impact of Refresh Reduction

In this section, we evaluate the performance improvement of MEMCON resulting from the reduction in refresh operation count. The actual execution of our long-running applications ranges from seconds to several minutes (Table 1). As cycle-accurate simulation of such long execution times cannot be feasibly performed with current tools, we provide an alternative method to evaluate the performance impact of MEMCON. The performance impact of MEMCON depends on two parameters: *(i)* the reduction in refresh count, and *(ii)* the execution time overhead of testing. We evaluate the performance impact of MEMCON in simulation *(i)* by modeling the refresh reduction we found in Section 6.1 for our long-running applications and *(ii)* injecting extra memory accesses that are required for testing. Section 6.1 demonstrates that the reduction in refresh count ranges from 64.7% to 74.5% in our long-running applications. We model *(i)* two refresh reduction amounts (60% to 75%

reduction compared to the baseline) on 30 single-core and four-core workloads from SPEC [80] and server [84] applications, and *(ii)* 256 concurrent memory tests performed every 64 ms by injecting extra read/write accesses. Figure 15 shows that MEMCON's performance improvement ranges from 10%/17%/40% to 12%/22%/50% in a single-core system and 10%/23%/52% to 17%/29%/65% in a 4-core system for 8/16/32 Gb DRAM chips.[11] We conclude that *(i)* MEMCON provides significant performance improvement due to the large reduction in refresh count it enables, and *(ii)* MEMCON's performance improvement increases with DRAM chip capacity.

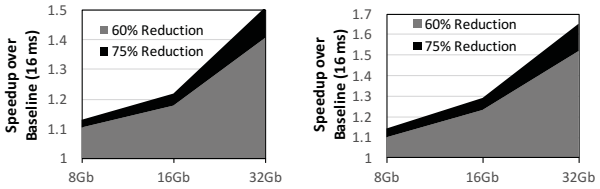


Figure 15: MEMCON's range of performance improvement with 60-75% refresh reduction over the baseline (16 ms refresh)

Analysis of testing overhead. In this section, we model 256-1024 concurrent tests every 64 ms in order to demonstrate the impact of extra memory accesses that are required for testing. Each test initiates extra memory requests to read, write, and compare data content according to the test mechanism described in Section 3.3. Running 1024 concurrent tests every 64 ms can test 8192 rows during every quantum (when the quantum length is 512 ms). Table 3 shows the average performance loss due to testing compared to the ideal case where testing has no overhead. The results show that the performance impact of extra accesses due to testing is very low, only 0.5%-1.9% on average for single-core systems. The overhead is lower in the multi-core system that can handle the extra accesses better with more parallelism. We conclude that the extra accesses due to testing have a negligible impact on performance. Note that MEMCON's full performance results in Figure 15 already include this testing overhead.

	Number of Concurrent Tests		
	256	512	1024
Single-core	0.54%	1.03%	1.88%
Four-core	0.05%	0.09%	0.48%

Table 3: Performance loss due to the extra accesses required for testing by MEMCON

6.3. Performance Comparison to Other Refresh Optimizations

MEMCON uses a 16 ms refresh interval for the ***HI-REF*** state and a 64 ms of refresh interval for the ***LO-REF*** state. In this section, we compare MEMCON with systems using other refresh intervals and refresh optimization techniques. First, we compare the performance benefit of MEMCON to a system that *always* uses a 32 ms refresh interval. This configuration represents a system that uses a less aggressive baseline than ours. Second, we compare MEMCON to a prior refresh optimization work, RAIDR [52], which refreshes all rows with *any possible failure* (for *any* possible data content in memory) at the ***HI-REF*** state. This mechanism depends on identifying every failing cell with an initial test and thus, relies on having access to the internal DRAM layout to detect these failures. The ***LO/HI-REF*** state evaluated for RAIDR is the same as ours (64/16 ms). We model that 16% of all rows are refreshed at the ***HI-REF*** state (when memory failures are randomly distributed with an error rate of 10^{-5}), which matches our experimental data from real DRAM chips in Figure 4. Third, we compare MEMCON with the ideal system that always uses the ***LO-REF*** state (i.e., 64 ms) for the entire memory, without any testing overhead (this is the upper bound shown in Section 6.1).

Figures 16a and 16b provide three major observations. First, MEMCON provides significant performance benefits even over a baseline with 32 ms refresh rate. On average, MEMCON improves performance by 4%/6%/13% and 5%/8%/17% for 8/16/32 Gb DRAM chips in a single- and multi-core system over the baseline that always uses a 32 refresh interval. Second, MEMCON consistently provides better performance than RAIDR. This is because that RAIDR refreshes more rows at the ***HI-REF*** state compared to MEMCON, as it detects every possible failure with an initial test and *cannot* exploit the fact that not all content triggers errors. Note that we assume that the error profiling information used for RAIDR is robust (and hence the RAIDR mechanism does *not* lead to incorrect execution), even though prior works have demonstrated that the worst-case profiling of every possible failure with every possible data content in memory is extremely challenging [31, 32, 51, 67, 70]. Third, MEMCON performs within 3-5% of the performance of the 64 ms refresh configuration. This is expected because the refresh reduction of MEMCON is close the the 75% upper bound (as shown in Section 6.1).

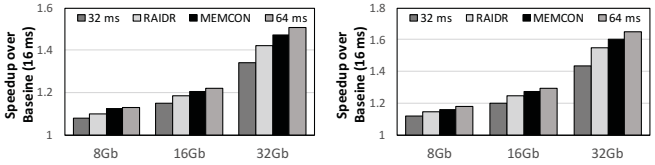


Figure 16: Comparison with other refresh mechanisms

We conclude that dynamically detecting and mitigating data-dependent failures at the system-level using the current memory content, while detection and mitigation occurs simultaneously with program execution is a feasible and cost-effective approach that performs better than prior approaches.

6.4. Evaluation of PRIL

We evaluate the coverage and the misprediction overhead of PRIL when identifying the long write intervals. We also analyze the storage overhead to implement PRIL.

Coverage of execution time. Figure 17 shows the fraction of total execution time spent while operating at the ***LO-REF*** state (referred to as coverage) when our prediction mechanism uses three different *current interval lengths*. On average, 95%

[11] We scale up the refresh operation count with DRAM chip capacity, as done in prior works [13, 15, 31, 51, 67, 70].

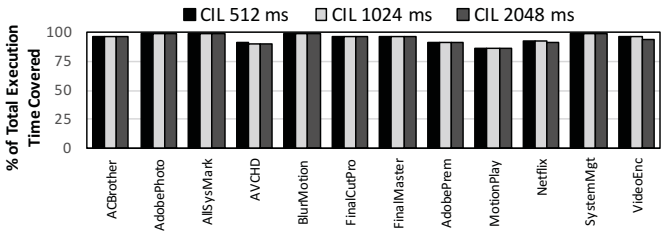


Figure 17: Execution time coverage of PRIL (operating in the *LO-REF* state)

of the total execution time is spent in the ***LO-REF*** state. We conclude that PRIL is an effective mechanism that can detect the long write intervals in real applications with high coverage.

Misprediction Overhead. MEMCON initiates testing for any page that is predicted to be remain idle (e.g., likely not be written to for a long time interval). However, as we have demonstrated in Section 4.1, the prediction mechanism is not completely accurate and MEMCON can initiate testing for some pages that may receive write requests relatively soon in the future. For these pages, the cost of testing would not be possible to amortize, as these pages have to be refreshed at the ***HI-REF*** state after the next write access. Figure 18 represents the fraction of time MEMCON spends on testing (for both correctly predicted and mispredicted pages) and refresh operations normalized to the amount of time spent on refresh in the baseline system. In the baseline all pages are refreshed with a 16 ms refresh interval (***HI-REF*** state). This figure shows that the time spent on testing (for both correctly and mispredicted pages) is very small and constitutes, on average, only 0.01% of the time spent on refresh operations in the baseline system. We conclude that testing memory simultaneously with program execution introduces negligible overhead even when some pages are tested unnecessarily due to misprediction of long write intervals with PRIL.

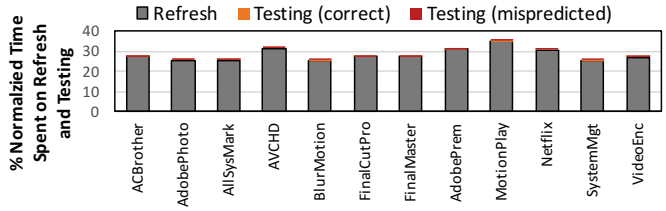
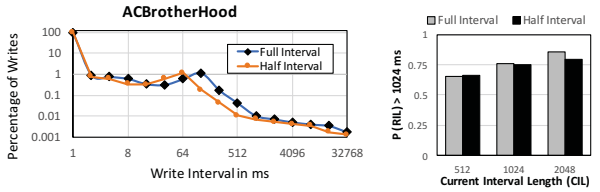


Figure 18: Fraction of time MEMCON spends on refresh operations and testing normalized to time spent on refresh in the baseline with 16 ms refresh

Storage Overhead. The storage overhead of PRIL comes mainly from the *write-maps* and *write-buffers* (Section 4.2). *Write-maps* are simple bit vectors, where each bit represents one memory page. Therefore, the overhead of a *write-map* is 128KB for an 8GB memory with 8KB pages. Our workloads access at most around 100K pages at each quantum, so we cache the *write-maps* in a 12KB direct-mapped cache in the memory controller. Note that predicting write intervals is off the critical path and does not affect performance. *Write-buffers* store the addresses of pages that are idle for the length of the quantum. We found that on average 4000 entries are enough to hold the addresses in each quantum (in the applications that we evaluate), leading to a storage overhead of 17KB in the memory controller. Note that these overheads can be optimized by leveraging prior techniques that use the last-level cache to store metadata [88] or co-locate data and metadata in memory to reduce extra accesses for metadata [57, 71, 89]. We leave the exploration of such optimizations for future work.

Sensitivity to Cache Size. Our mechanism relies on real applications spending a large fraction of their execution time on long write intervals. However, the distribution of writes to memory can change depending on how data gets evicted from the last-level cache based on cache contention or other microarchitectural events. We analyze the probability of the *remaining interval length* being greater than 1024 ms, when the interval between two consecutive writes to a page becomes smaller. To do so, we halve the write intervals for each application. Figure 19a demonstrates that the distribution of the write intervals *slightly* moves to the left when all intervals are halved. As the write interval distribution shows an exponential behavior (Section 4.1), most of the write intervals are less than 1 ms. Due to this behavior, Figure 19b shows that the probability that the *remaining interval length* is greater than 1024 ms does *not* change significantly compared to the configuration where all write intervals were twice as long. We conclude that cache size does *not* significantly impact MEMCON.



(a) (b)

Figure 19: (a) Write interval distribution and (b) the probability of RIL > 1024 ms, when the write intervals get halved

7. Related Work

To our knowledge, this is the first work to propose a system-level detection and mitigation technique for data-dependent failures that does not require knowledge of DRAM internals. No prior work *(i)* proposes a technique that can detect and mitigate failures based on program data content, *(ii)* demonstrates the properties of write intervals in real applications, and *(iii)* proposes a simple prediction mechanism to predict write intervals in real applications.

We discuss some related and mostly complementary works on DRAM failures and optimizing DRAM refresh. Note that, even though we demonstrate the benefits of MEMCON in the context of refresh optimization, MEMCON can be used to provide better reliability and scalability by detecting and mitigating failures at the system-level [32, 51, 67, 70] or to lower latency/power of DRAM by detecting cells that fail when we reduce the DRAM timing parameters/voltage [14, 17, 20, 45, 46].

System-Level DRAM Failure Detection. Prior works demonstrate the difficulty of detecting data-dependent DRAM failures at the system-level [31, 32, 51, 52, 67, 70]. Many of these failures depend on the internal architecture of DRAM, such as true-anti cell organization [51], open-closed bitline architecture [16, 45], remapping of faulty cells [31], address scrambling [31, 85], etc. Each DRAM chip can have failures with different data and access patterns and addresses. Some

of this information can potentially be reverse-engineered with elegant techniques [29, 31, 45]. However, prior works demonstrate that, even then, some failures remain undetected [31]. Our work is the first to attempt to detect and mitigate data-dependent failures *without* requiring information about DRAM internal organization.

Multi-Rate DRAM Refresh. Prior works propose multi-rate DRAM refresh mechanisms to reduce the performance and energy impact of refresh operations [52, 53, 70, 86]. Many of these works first profile the DRAM failures by simply writing 0/1s in memory and detecting data-dependent failures. They refresh the rows with failures more frequently and rows without failures less frequently, thereby reducing the refresh operation count. Recent works demonstrate that *simple* testing mechanisms *cannot* detect *all possible* data-dependent failures and employing such simple mechanisms would result in bit flips and unreliable DRAM operation [32, 51, 67, 70]. MEMCON is the first detection and mitigation mechanism that does *not* depend on detecting all possible data-dependent failures.

Refresh Optimization. Other works reduce the performance overhead of refresh by scheduling refresh operations in a flexible way such that they interfere less with program memory requests [15, 25, 60, 64, 83]. Our work is orthogonal, as MEMCON can still be applied on top of these techniques to reduce the overall refresh operation count and improve the performance and energy efficiency of the system.

Pareto Distribution in Real Workloads. Prior studies demonstrate that jobs running in real systems follow the Pareto distribution: only a small fraction of the jobs are the largest ones, but they comprise most of the total load in the system [19, 73]. Similarly, Unix process lifetimes [19], sizes of files and web traffic [18, 68, 73], lengths of memory episodes [40], and memory errors across servers [58] are shown to follow the Pareto distribution. No prior work demonstrates that the write intervals in real workloads follow the Pareto distribution.

8. Conclusion

We introduce MEMCON, the first system-level detection and mitigation technique for data-dependent DRAM failures that completely decouples failure detection from internal DRAM organization. MEMCON detects failures with the *current content* in memory by running online testing simultaneously with program execution. As testing a page for data-dependent failures after every write to the page (i.e., whenever the data content changes) incurs a large performance and energy overhead, MEMCON uses selective testing to initiate a test only when a write to a page is predicted to be followed by a long interval without any writes. MEMCON builds upon a simple, practical mechanism that predicts such long write intervals based on our observation that the write intervals in real workloads follow a Pareto distribution: the longer a page remains idle (i.e., not written to) after a write, the longer it is expected to remain idle. Our detailed experimental analysis shows that MEMCON greatly reduces refresh operation count and improves system performance. We believe that our analysis and experimental results will inspire future works to further develop and utilize memory content-based failure detection and mitigation techniques in real systems.

Acknowledgments

We thank the anonymous reviewers for their valuable suggestions. We thank David Zimmerman, Kevin J. Long, and Matthew A. Blackmore from Intel for their support with the workload traces. We acknowledge the support of our industrial partners: Google, Huawei, Intel, Microsoft, Samsung, VMware. This research was partially supported by NSF, Semiconductor Research Corporation, and the Intel Science and Technology Center for Cloud Computing. We thank the reviewers of IEEE CAL for the positive and helpful feedback on an earlier version of this work [33].

References

[1] "STREAM Benchmark," http://www.streambench.org/.
[2] "Oral history of Joel Karp," *Computer History Museum*, 2003.
[3] J. Ahn *et al.*, "A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing," in *ISCA*, 2015.
[4] J. Ahn *et al.*, "PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture," in *ISCA*, 2015.
[5] Z. Al-Ars *et al.*, "Effects of bit line coupling on the faulty behavior of DRAMs," in *VTS*, 2004.
[6] B. Arnold, *Pareto distributions*. Chapman & Hall/CRC Monographs on Statistics & Applied Probability, 1983.
[7] A. Bacchini *et al.*, "Characterization of data retention faults in DRAM devices," in *DFT*, 2014.
[8] Y. Bao *et al.*, "HMTT: A platform independent full-system memory trace monitoring system," in *SIGMETRICS*, 2008.
[9] R. E. Barlow *et al.*, "Properties of probability distributions with monotone hazard rate," *The Annals of Mathematical Statistics*, 1963.
[10] G. B. Bell *et al.*, "Characterization of silent stores," in *PACT*, 2000.
[11] S. Cha *et al.*, "Defect analysis and cost-effective resilience architecture for future DRAM devices," in *HPCA*, 2017.
[12] K. Chandrasekar *et al.*, "Exploiting Expendable Process-margins in DRAMs for Run-time Performance Optimization," in *DATE*, 2014.
[13] K. K. Chang, "Understanding and improving the latency of DRAM-based memory systems," Ph.D. dissertation, CMU, 2017.
[14] K. K. Chang *et al.*, "Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization," in *SIGMETRICS*, 2016.
[15] K. K. Chang *et al.*, "Improving DRAM performance by parallelizing refreshes with accesses," in *HPCA*, 2014.
[16] K. K. Chang *et al.*, "Low-Cost Inter-Linked Subarrays (LISA): Enabling Fast Inter-Subarray Data Movement in DRAM," in *HPCA*, 2016.
[17] K. K. Chang *et al.*, "Understanding reduced-voltage operation in modern DRAM devices: Experimental characterization, analysis, and mechanisms," in *SIGMETRICS*, 2017.
[18] M. E. Crovella and A. Bestavros, "Self-similarity in world wide web traffic: Evidence and possible causes," *IEEE/ACM Trans. Netw.*, 1997.
[19] M. Harchol-Balter and A. B. Downey, "Exploiting process lifetime distributions for dynamic load balancing," in *SIGMETRICS*, 1996.
[20] H. Hassan *et al.*, "ChargeCache: Reducing DRAM Latency by Exploiting Row Access Locality," in *HPCA*, 2016.
[21] H. Hassan *et al.*, "SoftMC: A Flexible and Practical Open-Source Infrastructure for Enabling Experimental DRAM Studies," in *HPCA*, 2017.
[22] M. Horiguchi and K. Itoh, *Repair for Nanoscale Memories*. Springer, 2011.
[23] K. Hsieh *et al.*, "Accelerating pointer chasing in 3D-stacked memory: Challenges, mechanisms, evaluation," in *ICCD*, 2016.
[24] A. A. Hwang *et al.*, "Cosmic Rays Don't Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design," in *ASPLOS*, 2012.
[25] C. Isen and L. John, "ESKIMO - Energy savings using semantic knowledge of inconsequential memory occupancy for DRAM subsystem," in *ISCA*, 2009.
[26] JEDEC, *JEDEC Standard: Low Power Double Data Rate 2 (LPDDR2)*, 2010.
[27] JEDEC, *Standard No. 79-3F. DDR3 SDRAM Specification*, 2012.
[28] JEDEC, *Standard No. 79-4B. DDR4 SDRAM Specification*, 2017.
[29] M. Jung *et al.*, "Reverse engineering of DRAMs: Row hammer with crosshair," in *MEMSYS*, 2016.
[30] U. Kang *et al.*, "Co-Architecting Controllers and DRAM to Enhance DRAM Process Scaling," in *The Memory Forum*, 2014.
[31] S. Khan *et al.*, "PARBOR: An Efficient System-Level Technique to Detect Data-Dependent Failures in DRAM," in *DSN*, 2016.
[32] S. Khan *et al.*, "The Efficacy of Error Mitigation Techniques for DRAM

Retention Failures: A Comparative Experimental Study," in *SIGMETRICS*, 2014.
[33] S. Khan *et al.*, "A case for memory content-based detection and mitigation of data-dependent failures in DRAM," in *IEEE CAL*, 2016.
[34] S. Khan *et al.*, *MEMCON Data Repository*, https://github.com/samirakhan/MEMCON-data, 2017.
[35] K. Kim, "Technology for sub-50nm DRAM and NAND flash manufacturing," in *IEDM*, 2005.
[36] Y. Kim *et al.*, "Ramulator: A Fast and Extensible DRAM Simulator," in *IEEE CAL*, 2015.
[37] Y. Kim *et al.*, *Ramulator Repository*, https://github.com/CMU-SAFARI/ramulator, 2015.
[38] Y. Kim, "Architectural techniques to enhance DRAM scaling," Ph.D. dissertation, CMU, 2015.
[39] Y. Kim *et al.*, "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," in *ISCA*, 2014.
[40] Y. Kim *et al.*, "ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers." in *HPCA*, 2010.
[41] Y. Kim *et al.*, "A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM," in *ISCA*, 2012.
[42] D. Lee *et al.*, "Decoupled Direct Memory Access: Isolating CPU and IO Traffic by Leveraging a Dual-Data-Port DRAM," in *PACT*, 2015.
[43] D. Lee, "Reducing DRAM latency at low cost by exploiting heterogeneity," Ph.D. dissertation, CMU, 2015.
[44] D. Lee *et al.*, "Reducing DRAM latency by exploiting design-induced latency variation in modern DRAM chips," in *ArXiv*, 2016.
[45] D. Lee *et al.*, "Design-induced latency variation in modern DRAM chips: Characterization, analysis, and latency reduction mechanisms," in *SIGMETRICS*, 2017.
[46] D. Lee *et al.*, "Adaptive-latency DRAM: Optimizing DRAM timing for the common-case," in *HPCA*, 2015.
[47] D. Lee *et al.*, "Tiered-latency DRAM: A low latency and low cost DRAM architecture," in *HPCA*, 2013.
[48] K. M. Lepak and M. H. Lipasti, "Silent stores for free," in *MICRO*, 2000.
[49] K. M. Lepak and M. H. Lipasti, "On the value locality of store instructions," in *ISCA*, 2000.
[50] K. M. Lepak and M. H. Lipasti, "Temporally silent stores," in *ASPLOS*, 2002.
[51] J. Liu *et al.*, "An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms," in *ISCA*, 2013.
[52] J. Liu *et al.*, "RAIDR: Retention-Aware Intelligent DRAM Refresh," in *ISCA*, 2012.
[53] S. Liu *et al.*, "Flikker: Saving DRAM Refresh-power Through Critical Data Partitioning," in *ASPLOS*, 2011.
[54] C.-K. Luk *et al.*, "Pin: Building customized program analysis tools with dynamic instrumentation," in *PLDI*, 2005.
[55] Y. Luo *et al.*, "Characterizing application memory error vulnerability to optimize datacenter cost via heterogeneous-reliability memory," in *DSN*, 2014.
[56] J. A. Mandelman *et al.*, "Challenges and future directions for the scaling of dynamic random-access memory (DRAM)," *IBM J. of Res. and Dev.*, 2002.
[57] J. Meza *et al.*, "Enabling efficient and scalable hybrid memories using fine-granularity DRAM cache management," *IEEE CAL*, vol. 11, 2012.
[58] J. Meza *et al.*, "Revisiting Memory Errors in Large-Scale Production Data Centers: Analysis and Modeling of New Trends from the Field," in *DSN*, 2015.
[59] W. Mueller *et al.*, "Challenges for the DRAM cell scaling to 40nm," in *IEDM*, 2005.
[60] J. Mukundan *et al.*, "Understanding and mitigating refresh overheads in high-density DDR4 DRAM systems," in *ISCA*, 2013.
[61] O. Mutlu, "The rowhammer problem and other issues we may face as memory becomes denser," in *DATE*, 2017.
[62] O. Mutlu, "Memory scaling: A systems architecture perspective," *IMW*, 2013.
[63] O. Mutlu and L. Subramanian, "Research problems and opportunities in memory systems," *SUPERFRI*, 2014.
[64] P. Nair *et al.*, "A case for refresh pausing in DRAM memory systems," in *HPCA*, 2013.
[65] P. Nair *et al.*, "ArchShield: Architectural framework for assisting DRAM scaling by tolerating high error rates," in *ISCA*, 2013.
[66] Y. Nakagome *et al.*, "The impact of data-line interference noise on DRAM scaling," *JSSC*, 1988.
[67] M. Patel *et al.*, "The reach profiler (REAPER): Enabling the mitigation of DRAM retention failures via profiling at aggressive conditions," in *ISCA*, 2017.
[68] V. Paxson and S. Floyd, "Wide area traffic: The failure of Poisson modeling," *IEEE/ACM Transactions on Networking*, 1995.
[69] E. Perelman *et al.*, "Using SimPoint for accurate and efficient simulation," in *SIGMETRICS*, 2003.
[70] M. Qureshi *et al.*, "AVATAR: A Variable-Retention-Time (VRT) Aware Refresh for DRAM Systems," in *DSN*, 2015.
[71] M. Qureshi and G. H. Loh, "Fundamental latency trade-off in architecting DRAM caches: Outperforming impractical SRAM-Tags with a simple and practical design," in *MICRO*, 2012.
[72] M. Redeker *et al.*, "An investigation into crosstalk noise in DRAM structures," in *MTDT*, 2002.
[73] B. Schroeder and M. Harchol-Balter, "Evaluation of task assignment policies for supercomputing servers: The case for load unbalancing and fairness," *Cluster Computing*, vol. 7, no. 2, Apr. 2004.
[74] B. Schroeder *et al.*, "DRAM errors in the wild: A large-scale field study," in *SIGMETRICS*, 2009.
[75] V. Seshadri *et al.*, "Fast bulk bitwise AND and OR in dram," in *IEEE CAL*, 2015.
[76] V. Seshadri *et al.*, "RowClone: Fast and Energy-efficient in-DRAM Bulk Data Copy and Initialization," in *MICRO*, 2013.
[77] V. Seshadri *et al.*, "Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology," in *MICRO*, 2017.
[78] V. Seshadri *et al.*, "Gather-scatter DRAM: In-DRAM address translation to improve the spatial locality of non-unit strided accesses," in *MICRO*, 2015.
[79] V. Seshadri and O. Mutlu, *Simple Operations in Memory to Reduce Data Movement.* Advances in Computers, 2016.
[80] SPEC CPU2006, "Standard Performance Evaluation Corporation," http://www.spec.org/cpu2006.
[81] V. Sridharan *et al.*, "Memory errors in modern systems: The good, the bad, and the ugly," in *ASPLOS*, 2015.
[82] V. Sridharan and D. Liberty, "A Study of DRAM Failures in the Field," in *SC*, 2012.
[83] J. Stuecheli *et al.*, "Elastic refresh: Techniques to mitigate refresh penalties in high density memory," in *ISCA*, 2010.
[84] Transaction Processing Performance Council, "TPC 2011," http://www.tpc.org/.
[85] A. J. van de Goor and I. Schanstra, "Address and Data Scrambling: Causes and Impact on Memory Tests," in *DELTA*, 2002.
[86] R. Venkatesan *et al.*, "Retention-Aware Placement in DRAM (RAPID): Software Methods for Quasi-Non-Volatile DRAM," in *HPCA*, 2006.
[87] Xilinx, *ML605 Hardware User Guide*, 2012.
[88] D. H. Yoon and M. Erez, "Virtualized and Flexible ECC for Main Memory," in *ASPLOS*, 2010.
[89] X. Yu *et al.*, "Banshee: Bandwidth-efficient DRAM caching via software/hardware cooperation," in *MICRO*, 2017.

Appendix

Cost of Read and Compare. There are three steps involved: (i) *read* and store the in-test row in the memory controller, (ii) keep the in-test row idle in memory for the duration of the target refresh interval to make sure victim cells are tested with the least possible charge in them, and (iii) read back the row again into the memory controller to *compare* its content to determine the failures. Therefore, Read and Compare mode involves reading the entire row into the memory controller twice. The cost of reading one row into the memory controller includes activating the row (tRCD), reading the cache blocks into the memory controller (128 * tCCD for a typical 8K row), and closing the row by precharging (tRP) it. Therefore, the cost for two row reads in terms of latency is 2*(tRCD + 128 * tCCD + tRP) = 1068 ns, using DDR3-1600 timing parameters [27].

Cost of Copy and Compare. The Copy and Compare mode involves reading the entire row into the memory controller twice (once before the test and once after the test) and writing the entire row once into a new row. The cost of Copy and Compare in terms of latency is 3*(tRCD + 128 * tCCD + tRP) = 1602 ns, using DDR3-1600 timing parameters [27].

Cost of a Refresh Operation. A row is refreshed by activating (tRAS) and precharging (tRP) it, making the cost of one refresh operation tRAS + tRP = 39 ns, using DDR3-1600 timing parameters [27].

Storage Overhead of Copy and Compare. A 2 GB module consists of 32768 rows per bank (a total of 262144 rows in 8 banks). Reserving 512 rows per bank (4K rows in total for all banks) for the special memory region to hold the content of the in-test rows results in 4096 / 262144 * 100 = 1.56% overhead of the total DRAM capacity.