A Case for Memory Content-Based Detection and Mitigation of Data-Dependent Failures in DRAM

Samira Khan*, Chris Wilkerson†, Donghyuk Lee‡, Alaa R. Alameldeen†, Onur Mutlu*‡
*University of Virginia †Intel Labs ‡Carnegie Mellon University *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 optimizations that improve reliability, latency, and energy efficiency of the system. All these 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. *Our goal* in this work is to 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 with *the current content in memory* while the programs are running in the system. Using experimental data from real machines, we demonstrate that MEMCON is an effective and low-overhead system-level detection and mitigation technique for data-dependent failures in DRAM.

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 [13, 15, 16, 18, 23, 24, 27, 30–32, 34, 39]. A prominent type of interference failure occurs depending on the data content in neighboring cells. Such failures are called *data-dependent failures* [14, 15, 18, 24, 39]. 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 [36]. These failures are inherent to DRAM design as they are caused by the electromagnetic coupling between wires used to access DRAM cells [1, 15, 18, 24, 39]. 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. Therefore, these failures can significantly affect the yield and manufacturing cost of DRAM chips. As DRAM cells get smaller, more cells fail due to cell-to-cell interference, posing a significant challenge to DRAM scaling [13, 15, 24, 27, 30–32, 34, 39]. Prior works proposed 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 [3, 6, 7, 10, 14, 15, 18, 20, 23–26, 28, 33, 38, 40, 43, 44, 46]. System-level detection and mitigation relies on detecting *every* cell that can fail during the entire lifetime of the system and mitigating failures via ECC and/or remapping of faulty cells to some other region [14, 33, 47]. Unfortunately, detection and mitigation of data dependent failures face two major challenges. First, 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 [12, 15, 22, 24, 45]. Without the exact knowledge of the internal design of a DRAM chip, it is not possible to detect *all* failures (discussed in detail in Section 2). Second, detecting *a*ll possible data-dependent failures, which constitute a very large number of failures, is time consuming [14, 15, 24], and mitigating such a large number of failures with ECC or remapping results in a large space overhead [14, 33]. It is expected that systems will have to detect and mitigate an even larger number of data-dependent failures in the future as cells become more vulnerable to interference with DRAM scaling [24, 34].

The **goal** of this work is to decouple the detection and mitigation of data-dependent failures from DRAM internals and design a low overhead mechanism that can be implemented in the system *without* requiring any knowledge about the specifics of 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 *e*very 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 observation, 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 content is less than the total number of failures with every possible combination of data content. Using experimental data from real DRAM chips tested with real program 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 simultaneously 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. On the other hand, the benefit of testing comes from using a lower refresh rate once no failure is found in a row. The longer the content remains the same, the higher the benefit from the reduced refresh operations. Therefore, there is a trade-off between the cost of testing vs. frequency of testing. In this work, we show that the cost of testing can be amortized by the reduction in refresh operations, if consecutive tests in a row are performed at a minimum time interval. As testing is triggered by a write operation that changes the data content in memory, we refer to this minimum interval as *MinWriteInterval*. We find that *MinWriteInterval* is 448–864 ms, depending on the test mode, refresh rate, and DRAM timing parameters (Section 3).

We profile real applications and make a case for MEMCON with two experiments. First, we demonstrate that a significant fraction of the time programs spend on intervals greater than *MinWriteInterval* (on average 71.8% of the execution time), which shows that MEMCON can amortize the cost of testing in real workloads. Second, we show that the impact of extra requests due to MEMCON is negligible on program performance (only 0.5-1.8% compared to the ideal no-test case). We conclude that MEMCON is an effective and low-overhead online detection and mitigation technique for data-dependent failures in DRAM.

This paper makes the following contributions:

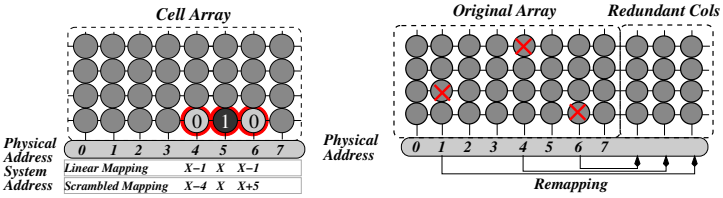
- This is the first work to propose a system-level data-dependent failure detection and mitigation technique that is completely decoupled from the internal physical organization of DRAM. Our detection and mitigation technique, MEMCON, depends only on the current memory content of the applications.
- 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 *MinWriteInterval*, we make a case for MEMCON with two experimental studies. By profiling real applications we show that *(i)* applications running in real machines spend a significant amount of time on long write intervals, providing an opportunity to maximize the benefit of testing, and *(ii)* the impact of extra requests due to MEMCON is *negligible* on program performance.

2 Motivation

Due to cell-to-cell interference in DRAM, some DRAM cells can fail depending on the data content stored in neighboring cells. We refer to these failures as data-dependent failures [14, 18, 24, 38, 39]. In order to detect these failures, manufacturers exhaustively test *neighboring DRAM cells* with data patterns that introduce enough cell-to-cell interference to cause failures. Unlike traditional manufacturing time testing, some recent works propose to detect data-dependent failures at the system-level to enable system optimizations that improve reliability, latency, and energy efficiency of memory [14, 15, 23, 24, 38]. 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 content in memory. They propose to detect these cells with an initial testing phase and mitigate the failures with ECC or remapping to ensure correct operation for *any content* in memory that can possibly occur during system operation. However, there are two major challenges in detecting and mitigating data-dependent failures in the system.

(i) 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 [12, 15, 22, 24, 45]. 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 1a 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-1** and **X+5**. This internal address mapping from physical to system-level address is not exposed to the system [15, 22, 24, 45]. To make things worse, vendors scramble the addresses differently for each generation of DRAM chips [15].



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

Fig. 1: 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 [9]. Figure 1b 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 cell array. 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 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 the manufacturing organization. Even if the vendors expose the details of DRAM internals, scrambling of address space and remapped columns make system-level 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 past and future commercially available DRAM chips is rather challenging.

(ii) 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 [18, 34]. Prior works have shown that mitigating a large number of failures with ECC or remapping adds a significant storage overhead, making mitigation very expensive [14, 33].

The **goal** of this work is to design a low-overhead technique for detecting and mitigating data-dependent failures that can be implemented in the system, without requiring *any* knowledge about the internal DRAM organization. To this end, we propose to decouple the detection and mitigation of data-dependent failures from DRAM internals.[1] We limit the scope of our mechanism to data-dependent failures, as other interference [18, 34] and random failures [10, 28, 38] can be mitigated using ECC or orthogonal mechanisms proposed in prior works [14, 18, 33, 38]. We also do not provide mechanisms to handle variation in data-dependent 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 [14, 20, 24]. In the next section, we describe our DRAM-transparent 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 an argument that it is *not* necessary to detect and mitigate *every possible* data-dependent failure that can occur with *any* memory content during the lifetime of the system. Instead, it is sufficient to detect and mitigate the failures that 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 dependent on the *change in memory content* with execution time and eliminates the need to detect every susceptible cell that can fail due to data dependence with any content in memory.

Based on this observation, we make a case for *memory content-based detection and mitigation* for 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. As a result, it detects failures dynamically while programs are running and mitigates only failures that can be triggered by the programs. 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 less than the total number of failures with every possible combination of data content in memory. Figure 2 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. The failure rate is generated by testing real DRAM chips with an FPGA-based infrastructure [6, 8, 14, 15, 18, 20, 22, 24, 38]. We test DRAM chips with a refresh interval of 4 s at 45C, which corresponds to a refresh interval of 328 ms at 85C [24]. We present the percentage of failing rows (averaged over every 100 million instructions across a set of representative phases [37]) for 20 SPEC CPU 2006 [42]

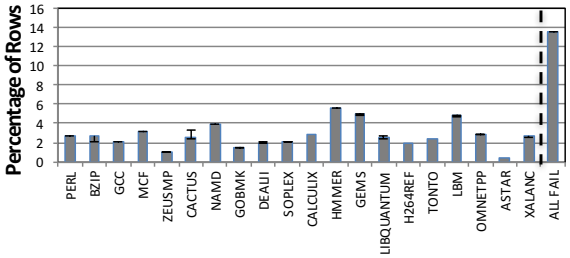


Fig. 2: Percentage of rows that exhibit failures

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.

benchmarks.[2] We also show the maximum and minimum fraction of failing rows of each benchmark as an error bar in the figure. This figure shows that programs encounter 2.4X-35.2X fewer failures than every possible failure in DRAM (represented by ALL FAIL in the figure).

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 memory, content gets changed and MEMCON needs to test that new content to detect whether or not that content introduces any data-dependent failure and 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 caching 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, keeping the row idle for the test period, and reading the entire row again to compare with the prior 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, caching 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.

Cost-Benefit Analysis of MEMCON. As testing at the system-level is expensive, it is necessary to analyze the cost-benefit of MEMCON during application runtime in order 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 mitigation technique of optimizing the refresh rate: MEMCON initially refreshes each row very frequently to avoid any failures; after the row content is tested and no failures are 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 3 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 overshadow the benefit of testing. We provide the detailed cost-benefit analysis in Section 4.1. 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 (the periodic latency required to refresh a row) is represented as the ***HI-REF*** state in the figure. *(ii)* The cost of testing (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 3 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, the benefit of ***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 3(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 3(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 for MEMCON is illustrated in Figure 3(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.

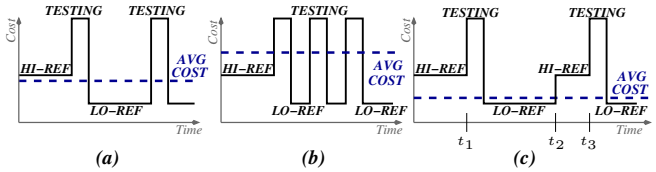


Fig. 3: Tradeoff between cost of testing vs. frequency of testing

4 A Case for MEMCON

In this section, we make a case for MEMCON by answering three questions: *(i)* How to determine the frequency of testing such that cost of testing is amortized? (Section 4.1) *(ii)* Are the intervals between writes to a row large enough to amortize the cost of testing in real programs? (Section 4.2) *(iii)* How much performance degradation can testing cause to the applications running in the system? (Section 4.3)

4.1 Amortizing the Cost of Testing

In order to determine the minimum interval between two writes that can amortize 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 for ***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 4 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 typical refresh interval in modern DRAM devices. 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, 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.

2. Even if we demonstrate the fraction of failing rows for only SPEC benchmarks, we believe that other workloads with a larger working set 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.

3. MEMCON can be optimized even further by considering the interval between read accesses and eliminating testing if the row gets read frequently enough such that it does not need refresh. We leave such an optimization for future work.

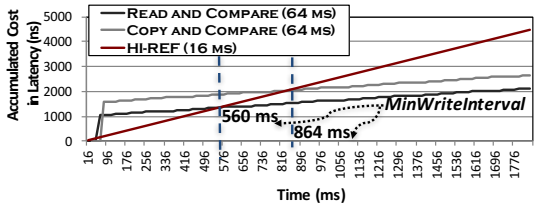


Fig. 4: Determining *MinWriteInterval*

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 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 content of the in-test rows are temporarily stored in a different, special region of memory to service requests 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).[4] [5]

Figure 4 shows that both of these test modes pay the high cost of testing in the beginning (1068 ns and 1602 ns, respectively). After that, the system is refreshed once every 64 ms.[6] Therefore, the total cost increases more slowly over time compared to 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 ***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 evaluated 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.

4.2 Distribution of Writes In Programs

In order to evaluate the effectiveness of MEMCON in amortizing the cost of testing in real applications, we need to consider two issues regarding write intervals in applications. First, there have to be writes in applications occurring at an interval greater than *MinInterval*: otherwise, according to our evaluation in Section 4.1, aggressively using the ***HI-REF*** state is more cost effective than MEMCON. Second, programs have to spend a significant fraction of their runtime in those long intervals to make MEMCON effective in lowering the total cost.

4. The storage overhead of this mode is modest as only a small fraction of rows are tested concurrently (e.g., reserving 512 rows per bank for the special memory region, 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 a very little storage overhead.

5. Not that this mode can become significantly faster by performing copy operations within DRAM, using mechanisms like RowClone [41] and LISA [5]. Exploiting subarray-level parallelism [17], tiered-latency DRAM [19], or a Dual-Data-Port DRAM [21] can also reduce the performance impact of such copy operations. We do not evaluate such optimizations and leave them for future work.

6. We use 64 ms is this figure, but we also report summary results for 128 ms and 256 ms later.

We evaluate two metrics: percentage of writes with interval greater than *MinWriteInterval* is shown in Figure 5(a) and the fraction of time programs spend on those intervals is shown in Figure 5(b). We use an FPGA-based memory tracing system that reads memory bus signals to trace DRAM traffic. The tracer keeps track of memory commands and addresses from the CPU to a single DRAM channel. Traces span minutes of application runtime after the initialization phase. Each trace includes all memory commands, associated addresses (if applicable) and timestamps during the tracing time frame. In this experiment, in addition to SPEC 2006 benchmarks, we evaluate some modern applications found in typical mobile systems (the popular game Angry Birds [2] and a background application that remains mostly idle) and server systems (database server Oracle [35] and search engine Bing [4]). We make two observations from these figures. First, Figure 5(a) shows that most (92%) of the writes to a row occur very frequently (within 1 ms) in real applications. Fortunately, writes within 1 ms do *not* need to be tested because such frequent accesses refresh the row *before* the refresh interval, avoiding any data-dependent failures. Second, only a small fraction of writes exhibit large intervals in real programs; the percentage of writes with intervals greater than 512 ms is between 0.3% to 2.3%. We also observe that, even though writes with large intervals are infrequent, writes happen in bursts and these intervals between the bursts are very long Thus, a program spends a large fraction of its execution time in these intervals: on average 71.8% of execution time, as shown in Figure 5(b). Therefore, when MEMCON performs testing, it would lead to significant benefit by reducing refresh operations for the duration of these very long intervals that account for most of the program execution time. **We conclude that MEMCON not only amortizes the cost of testing, but also significantly reduces the total cost by using infrequent refreshes for a large fraction of program execution time.**[7]

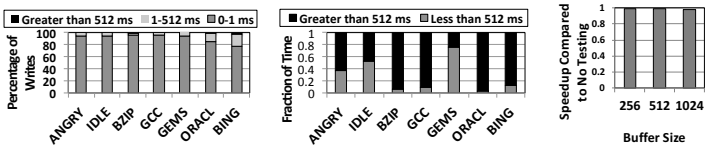


Fig. 5: (a) Distribution of write intervals in rows, (b) Fraction of time programs spend in long interval writes, (c) Impact of extra accesses due to testing.

4.3 Impact of Testing on Program Accesses

So far, we have shown that real applications exhibit write intervals greater than the *MinWriteInterval* and spend a significant fraction of time in those intervals, making MEMCON highly effective. However, in order to make a complete case for MEMCON, we also need to evaluate the impact of testing on program accesses. As testing injects extra accesses to memory, theses accesses would interfere with program accesses and can slow down the running applications. We evaluate the impact of testing on a wide range of applications (30 applications from SPEC, server, and stream [29]), when the detection of failure and execution of these applications occur simultaneously. In this experiment, whenever there is a write, memory controller puts the write address of the row in a queue and starts running tests on such rows in memory using the READ AND COMPARE mode. However, we restrict the number of rows that are tested concurrently by using a fixed size buffer in the memory controller for in-test rows. Figure 5(c) shows the average slowdown in performance when MEMCON can concurrently test 256-1024 rows. This figure shows that the impact of extra accesses due to testing on performance is very low, slowing down the running applications by only 0.5%-1.8% on average. **We conclude that the extra accesses due to testing have a negligible impact on performance.**

Based on our model, analysis, and experimental results, we conclude that dynamically detecting and mitigating data-dependent

7. Note that obtaining this benefit requires predicting the write intervals ahead of time in order to make a decision as to whether or not to initiate testing upon encountering a write to a row. We leave the design and evaluation of such prediction mechanisms for future work.

failures in the system using the current memory content, where detection and mitigation occurs simultaneously with program execution is a feasible and cost-effective approach.

5 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. In this work, we make a case for MEMCON, showing that the overhead of such a concurrent detection technique can be negligible. We believe that our analysis and experimental results will inspire future works to design, build, and evaluate memory content-based detection and mitigation techniques in real systems.

Acknowledgments

We thank anonymous reviewers and SAFARI group members for feedback. We acknowledge the support of Google, Intel, Nvidia, Seagate, and Samsung. This research was supported in part by the ISTC-CC, an NSF CAREER Award (CCF-0953246), and NSF grants (CCF-1212962, CNS-1320531, and CCF-1566483).

References

[1] Zaid Al-Ars et al. Effects of bit line coupling on the faulty behavior of DRAMs. VTS, 2004.
[2] Angry Birds. *https://www.angrybirds.com*.
[3] Angelo Bacchini et al. Characterization of data retention faults in DRAM devices. DFT, 2014.
[4] Bing. *https://www.bing.com*.
[5] Kevin Chang et al. Low-cost inter-linked subarrays (LISA): Enabling fast inter-subarray data movement in DRAM. HPCA, 2016.
[6] Kevin Chang et al. Understanding latency variation in modern DRAM chips: Experimental characterization, analysis, and optimization. SIGMETRICS, 2016.
[7] Hasan Hassan et al. ChargeCache: Reducing DRAM latency by exploiting row access locality. HPCA, 2016.
[8] Hasan Hassan et al. SoftMC: A flexible and practical open-source infrastructure for enabling experimental DRAM studies. HPCA, 2017.
[9] Masashi Horiguchi and Kiyoo Itoh. *Repair for Nanoscale Memories*. Springer, 2011.
[10] Andy A. Hwang et al. Cosmic rays don't strike twice: Understanding the nature of DRAM errors and the implications for system design. ASPLOS, 2012.
[11] JEDEC. *Standard No. 79-3F. DDR3 SDRAM Specification*, July 2012.
[12] Matthias Jung et al. Reverse engineering of DRAMs: Row hammer with crosshair. MEMSYS, 2016.
[13] Uksong Kang et al. Co-architecting controllers and DRAM to enhance DRAM process scaling. In *The Memory Forum*, 2014.
[14] Samira Khan et al. The efficacy of error mitigation techniques for DRAM retention failures: A comparative experimental study. SIGMETRICS, 2014.
[15] Samira Khan et al. PARBOR: An efficient system-level technique to detect data-dependent failures in DRAM. DSN, 2016.
[16] Kinam Kim. Technology for sub-50nm DRAM and NAND flash manufacturing. IEDM, 2005.
[17] Yoongu Kim et al. A case for subarray-level parallelism (SALP) in DRAM. In *ISCA*, 2012.
[18] Yoongu Kim et al. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. ISCA, 2014.
[19] Donghyuk Lee et al. Tiered-latency DRAM: A low latency and low cost DRAM architecture. HPCA, 2013.
[20] Donghyuk Lee et al. Adaptive-latency DRAM: Optimizing DRAM timing for the common-case. HPCA, 2015.
[21] Donghyuk Lee et al. Decoupled Direct Memory Access: Isolating CPU and IO traffic by leveraging a Dual-Data-Port DRAM. PACT, 2015.
[22] Donghyuk Lee et al. Reducing DRAM latency by exploiting design-induced latency variation in modern DRAM chips. ArXiv, 2016.
[23] Jamie Liu et al. RAIDR: Retention-aware intelligent DRAM refresh. ISCA, 2012.
[24] Jamie Liu et al. An experimental study of data retention behavior in modern DRAM devices: Implications for retention time profiling mechanisms. ISCA, 2013.
[25] Song Liu et al. Flikker: Saving DRAM refresh-power through critical data partitioning. ASPLOS, 2011.
[26] Yixin Luo et al. Characterizing application memory error vulnerability to optimize datacenter cost via heterogeneous-reliability memory. DSN, 2014.
[27] Jack A Mandelman et al. Challenges and future directions for the scaling of dynamic random-access memory (DRAM). *IBM J. of Res. and Dev.*, 2002.
[28] Justin Meza et al. Revisiting memory errors in large-scale production data centers: Analysis and modeling of new trends from the field. DSN, 2015.
[29] Thomas Moscibroda and Onur Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. USENIX Security Symposium, 2007.
[30] W Mueller et al. Challenges for the DRAM cell scaling to 40nm. IEDM, 2005.
[31] Onur Mutlu. Memory scaling: A systems architecture perspective. *IMW*, 2013.
[32] Onur Mutlu and Lavanya Subramanian. Research problems and opportunities in memory systems. *SUPERFRI*, 2014.
[33] Prashant J. Nair et al. ArchShield: Architectural framework for assisting DRAM scaling by tolerating high error rates. ISCA, 2013.
[34] Yoshinobu Nakagome et al. The impact of data-line interference noise on DRAM scaling. *JSSC*, 1988.
[35] Oracle: Integrated Cloud Applications and Platform Service. *https://www.oracle.com/index.html*.
[36] Oral History of Joel Karp. *http://archive.computerhistory.org/resources/access/text/2012/07/102658274-05-01-acc.pdf*.
[37] Erez Perelman et al. Using SimPoint for accurate and efficient simulation. SIGMETRICS, 2003.
[38] Moinuddin Qureshi et al. AVATAR: A Variable-Retention-Time (VRT) Aware Refresh for DRAM Systems. DSN, 2015.
[39] Michael Redeker, Bruce F. Cockburn, and Duncan G. Elliott. An investigation into crosstalk noise in DRAM structures. MTDT, 2002.
[40] Bianca Schroeder et al. DRAM errors in the wild: A large-scale field study. SIGMETRICS, 2009.
[41] V. Seshadri et al. RowClone: Fast and efficient In-DRAM copy and initialization of bulk data. MICRO, 2013.
[42] SPEC CPU2006. Standard Performance Evaluation Corporation. http://www.spec.org/cpu2006.
[43] Vilas Sridharan et al. Memory errors in modern systems: The good, the bad, and the ugly. ASPLOS, 2015.
[44] Vilas Sridharan and Dean Liberty. A study of DRAM failures in the field. SC, 2012.
[45] Ad. J. van de Goor and Ivo Schanstra. Address and data scrambling: Causes and impact on memory tests. DELTA, 2002.
[46] Ravi K. Venkatesan et al. Retention-aware placement in DRAM (RAPID): Software methods for quasi-non-volatile DRAM. HPCA, 2006.
[47] Doe Hyun Yoon and Mattan Erez. Virtualized and flexible ECC for main memory. ASPLOS, 2010.

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 least possible charge, and (iii) read back the row again in the memory controller to *compare* the content to determine 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 in the memory controller (128 * `tCCD` for a typical 8K row), and closing the row by precharging (`tRP`) it. Therefore, the cost for two row accesses in terms of latency is 2*(`tRCD` + 128 * `tCCD` + `tRP`) = 1068 ns, using DDR3-1600 timing parameters [11].

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 [11].

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 [11].

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.