PARBOR: An Efficient System-Level Technique to Detect Data-Dependent Failures in DRAM

Samira Khan[⋆] Donghyuk Lee[†‡] Onur Mutlu[*†]
[⋆]University of Virginia [†]Carnegie Mellon University [‡]Nvidia [*]ETH Zürich

Abstract

System-level detection and mitigation of DRAM failures offer a variety of system enhancements, such as better reliability, scalability, energy, and performance. Unfortunately, system-level detection is challenging for DRAM failures that depend on the data content of neighboring cells (data-dependent failures). DRAM vendors internally scramble/remap the system-level address space. Therefore, testing data-dependent failures using neighboring system-level addresses does not actually test the cells that are physically *adjacent. In this work, we argue that one promising way to uncover data-dependent failures in the system is to determine the location of physically neighboring cells in the system address space. Unfortunately, if done naively, such a test takes 49 days to detect neighboring addresses even in a single memory row, making it infeasible in real systems.*

We develop PARBOR, an efficient system-level technique that determines the locations of the physically neighboring DRAM cells in the system address space and uses this information to detect data-dependent failures. To our knowledge, this is the first work that solves the challenge of detecting data-dependent failures in DRAM in the presence of DRAM-internal scrambling of system-level addresses. We experimentally demonstrate the effectiveness of PARBOR using 144 *real DRAM chips from three major vendors. Our experimental evaluation shows that PARBOR 1) detects neighboring cell locations with only* 66-90 *tests, a* 745,654*X reduction compared to the naive test, and 2) uncovers 21.9% more failures compared to a random-pattern test that is unaware of the neighbor cell locations. We introduce a new mechanism that utilizes PARBOR to reduce refresh rate based on the data content of memory locations, thereby improving system performance and efficiency. We hope that our fast and efficient system-level detection technique enables other new ideas and mechanisms that improve the reliability, performance, and energy efficiency of DRAM-based memory systems.*

1. Introduction

The tremendous growth in DRAM capacity over the last few decades resulted from the continued scaling of DRAM process technology. By making the cell dimensions smaller, more cells can be packed in the same die area, enabling more capacity at the same cost. Unfortunately, scaling cells to smaller technology nodes introduces major reliability issues in DRAM [34, 37, 39, 46, 47, 52, 56, 57, 58, 60, 64]. In order to enable reliable DRAM in future memory systems, prior works proposed to detect and mitigate DRAM failures in the field, while the system is under operation. Such *system-level detection* of DRAM failures has three advantages. First, it can enable better scaling of DRAM by manufacturing smaller and unreliable cells, but providing reliability guarantees by detecting and mitigating failures at the system level [6, 35, 47, 51, 59, 62]. Second, it can provide better reliability against failures that escape the manufacturing tests [30, 39, 53, 67, 74, 75]. Third, it can improve system performance and power/energy consumption by enabling the system to reduce the access latency [18, 27, 43] and refresh rate [46, 48, 62, 80] of robust DRAM cells, while maintaining a higher latency and higher refresh rate for likely-to-fail cells.

Even though a system-level detection and mitigation technique offers better reliability, performance, and energy efficiency, such a technique faces a major unresolved challenge in DRAM failure detection. A large fraction of DRAM failures occur due to interference between cells located in close proximity: some cells fail when specific patterns are stored in neighboring cells, as shown in prior works [18, 35, 39, 43, 45, 47, 64]. We call this phenomenon *data-dependent failures*. These failures can be detected by testing neighboring cells with a data pattern that induces the maximum interference between the cells, which we call the *worst-case data pattern*. Detecting data-dependent failures at the system level is particularly challenging: DRAM vendors internally *scramble* the address space, and therefore, adjacent bit addresses in the system (i.e., *system addresses*) do not map to adjacent cells in the physical DRAM cell arrays (i.e., *physical addresses*). This mapping of system address to physical address, referred to as *address mapping*, is different for each vendor and each chip generation (depending on the internal design of chips), making it difficult to expose it to the system. Without precise knowledge of the address mapping, the worst-case data pattern cannot be applied to induce the maximum interference in physically neighboring cells, since adjacent bit addresses at the system level might not map to cells that are physically adjacent.

Figure 1 shows one example of address scrambling. If the system address and physical address are mapped linearly, data-dependent failure in the cell at the system-level bit address *X* can be detected by writing the worst-case pattern at system-level bit addresses *X-1*, *X*, and *X+1*. Unfortunately, due to scrambled mapping of system addresses to physical addresses, physically neighboring cells can have completely different system addresses. In this example, the left physical neighbor is located at system address *X+1*, the right physical neighbor at *X+5*.

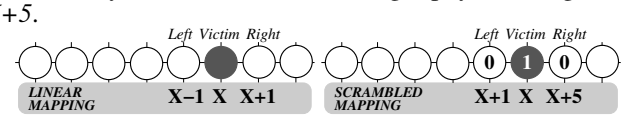


Figure 1: Scrambled mapping of system address to physical

In this work, we show that one way to uncover data-dependent DRAM failures at the system level is to determine the location of physically neighboring cells in the system address space and use that information to devise test patterns that ensure that the physically neighboring cells are tested with the worst-case data pattern. This is a promising approach considering that each generation of chips can have different address mappings, and a technique that can learn the mapping will be generally applicable to any system with any chip. As a cell is primarily affected by two *immediate neighbor* cells [1, 60], such a system-level mechanism should detect the locations of these immediate neighbors. Unfortunately, naively ensuring that all two physically neighboring cells are covered in testing requires exhaustively testing every combination of two bit addresses in a row, an $O(n^2)$ test where n is the number of cells, which takes 49 days of testing even for a *single row* with 8K cells. As cells get smaller and more vulnerable to cell-to-cell interference, it is likely that potentially more neighboring cells will affect each other in the future [2], increasing the test time to 1115 years for three neighbors and 9.1M years for four neighbors. Clearly, it is not feasible to run tests for such a long time to uncover data-dependent failures.

Our goal in this work is to propose a fast and efficient mechanism to determine the locations of the physically neighboring

cells in the system address space. Doing so would help to detect data-dependent failures in the system and thus would enable techniques that improve DRAM reliability, latency, and energy, and aid DRAM technology scaling.

To this end, we develop an efficient test method based on two key ideas. The **first key idea** leverages the observation that cells exhibit variability in the way they get affected by cell-to-cell interference. Even though cells get affected by both left and right neighbors, some cells are *strongly* coupled to only one neighbor (due to process variation) and fail when the data content of *only one* neighbor changes. We call these cells *strongly coupled cells*. Figure 2 shows the variability in coupling in three different cells (A, B, C, each marked as *Victim*). Cells A and B are strongly coupled cells: they fail depending on the content of *only* the left or the right neighbor, respectively. Cell C is a *weakly coupled cell*: it fails only when *both* neighbors contain the worst-case data pattern. Leveraging our observation, our key idea to reduce the test time is to locate the address of *only one* neighboring cell of strongly coupled cells. This approach reduces the test time from $O(n^2)$ to $O(n)$, as each address bit needs to be tested linearly once to determine the address that causes the data-dependent failure in the strongly coupled cell.

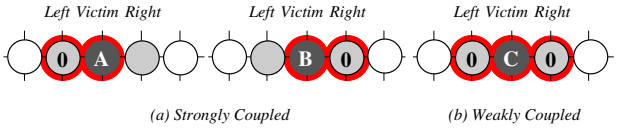


Figure 2: Strong vs. weak coupling of neighboring cells

However, detecting the address of only one neighbor can have a negative effect on the coverage of failures for weakly coupled cells, as these cells depend on the content of both neighbors instead of one. Our **second key idea** enables us to detect *both* neighboring locations with multiple simple linear tests. The idea is based on the observation that internal DRAM organization is mostly regular and repetitive, with an abundance of parallelism in rows. Due to the regularity in internal DRAM address mapping, the distance of the left neighbor in the system address space is the same for *many cells in multiple rows*. Ditto for the distance of the right neighbor. And, due to randomness in process variation, some strongly coupled cells fail based on the content of the left neighbor and some on the right neighbor. By running *parallel tests* in different rows *simultaneously*, we can detect the system address distance of both the left neighbor and the right neighbor for different strongly coupled cells, which together serve as the distances of the left and right neighbors for weakly coupled cells (since internal DRAM address mapping is very regular across rows). Consequently, it is possible to estimate the addresses of *all* neighboring cells by testing multiple cells in different rows *simultaneously* and *aggregating* the distances of the neighboring locations found in those rows.

Based on these two key ideas, we propose **PA**rallel **R**ecursive neigh**BOR (PARBOR)** testing, which detects the locations of physically neighboring cells efficiently by recursively testing multiple rows in parallel. We demonstrate the effectiveness of PARBOR by evaluating it using real DRAM chips. Using an FPGA-based infrastructure, we are able to find the locations of physically neighboring cells in 144 real DRAM chips manufactured by three major vendors with only $66-90$ tests, a 90X and 745,654X reduction, respectively, compared to tests with $O(n)$ and $O(n^2)$ complexity. Using this neighboring cell location information, we devise a new test methodology, which performs only a *small* number of test iterations to uncover data dependent failures in the entire chip. PARBOR uncovers 21.9% more failures than a test with *random data patterns* that is unaware of the locations of neighboring cells in 144 tested DRAM chips.

We show that PARBOR enables not only prior optimizations that rely on system-level detection of data-dependent failures, but also new system-level optimizations that improve the reliability, performance, and energy efficiency of DRAM. We propose and evaluate one such mechanism that improves DRAM performance by reducing refresh operations. We call this new refresh reduction technique *data content-based refresh* (*DC-REF*). The key idea of DC-REF is to employ a high refresh rate only in rows where the data content of the application matches the worst-case pattern that causes failures. Our evaluation shows that DC-REF reduces the number of refreshes by 73% and improves performance by 18% for a system with 32 Gbit DRAM chips and 8 cores running a wide range of applications.

This paper makes the following **contributions**:

- This is the first work to propose an efficient system-level mechanism for locating the addresses of neighboring cells in DRAM devices. Our mechanism, PARBOR, reduces the test time for such detection by exploiting the notion of strongly coupled cells and recursively testing multiple rows in parallel. We use the addresses of physically neighboring cells to devise a new test methodology that can efficiently uncover data-dependent failures.
- We experimentally demonstrate that PARBOR can detect the neighboring cell locations with a small number of tests. Using an FPGA-based infrastructure, we show that PARBOR detects neighboring locations with only $66-90$ tests in 144 real DRAM chips from three major manufacturers, a 90X and 745,654X reduction compared to optimized/naive tests, with respectively $O(n)$ and $O(n^2)$ complexity.
- We show that PARBOR uncovers, on average, 21.9% more failures than a test with *random data patterns* that is unaware of the locations of neighboring cells.
- We show that PARBOR enables new mechanisms to improve future memory systems. Based on the detected patterns that cause failures in cells, we propose a *data content-based* refresh minimization mechanism, DC-REF, which improves performance by 18% for a system with 32 Gbit DRAM chips and 8 cores over a wide range of applications.

2. Background

We provide necessary background on DRAM organization and cell operation to understand the causes of data-dependent failures. We refer the reader to other works for more detail on the DRAM system and its operation [16, 38, 42, 43, 44, 46, 69].

2.1. DRAM Organization

Figure 3a shows the high level organization of DRAM. A DRAM module is connected to the memory controller through a channel and 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. While accessing a 64-byte cache line from DRAM, each chip transfers 64 bits of data in 8 bursts.

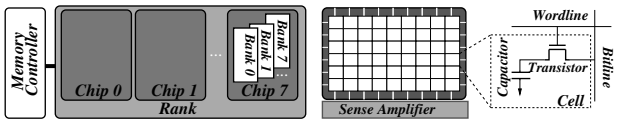
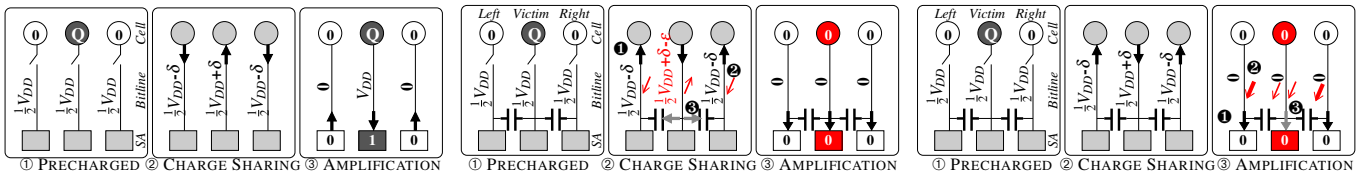


Figure 3: DRAM organization

DRAM banks are organized as multiple 2D arrays of cells. Figure 3b shows a cell array within a bank, and sense amplifiers that are used to latch data values read from cells. A cell consists of a capacitor and a transistor. The capacitor stores data as



(a) *Cell operation* (b) *Failure during charge sharing* (c) *Failure during sensing and amplification*

Figure 4: Example data-dependent failures due to coupling

charge, and the transistor acts as a switch to the capacitor. While accessing a row, all cells in the row are selected in parallel using a wire called *wordline*. An individual cell in each column is accessed through a vertical wire called *bitline*.

2.2. DRAM Cell Operation

Figure 4a describes DRAM cell operation in detail. A cell is represented as a circle and a sense amplifier is represented as a rectangular box. We represent the amount of charge in grayscale where white depicts the discharged state, **0** (data value '0'), dark gray the charged state, **Q** (data value '1'), and light gray the quiescent state. DRAM cell operation can be explained with three major steps:

① PRECHARGED. Initially, each cell in a bank is in charged (**Q**) or discharged (**0**) state, depending on the data value stored in the cell. Bitlines are maintained at a voltage level of $\frac{1}{2}V_{DD}$, to enable access to the bank.

② CHARGE SHARING. Activation of a row raises the voltage in the wordline, connecting the cell capacitor to the bitline. Depending on whether the cell is charged (**Q**) or discharged (**0**), charge flows out of or into the cell, slightly perturbing the bitline voltage from $\frac{1}{2}V_{DD}$ to $\frac{1}{2}V_{DD} \pm \delta$.

③ SENSING AND AMPLIFICATION. After the charge sharing phase, the sense amplifier "senses" the voltage perturbation towards V_{DD} or **0** in its corresponding bitlines and starts to "amplify" this voltage in the same direction. Data values can be read from the sense amplifiers during this time. A cell remains in quiescent state during this transition and is restored to its original state once the bitline voltage stabilizes to V_{DD} or **0**.

Once read or write accesses to the row are complete, the bank is brought back to the PRECHARGED state: the cell capacitor and sense amplifiers are disconnected from the bitlines, and the bitlines are driven back to $\frac{1}{2}V_{DD}$.

So far, we discussed cell operation without considering interference among cells. We next discuss the impact of data values of neighboring cells on cell operation.

2.3. Data-Dependent Failures: Causes and Examples

Adjacent cells can interfere with (i.e., disturb) each other, depending on the values (i.e., charge) stored in them. A bit failure occurs when a cell is disturbed enough to alter the data value stored in the cell. We call such a failure that is a function of the data values of neighboring cells as *data-dependent failure*. The cell experiencing the failure is the *victim cell*. A data-dependent failure is caused by coupling between adjacent bitlines due to parasitic coupling capacitance [1, 45, 47, 60, 64]. This capacitance provides an indirect path between neighboring cells, which affects cell operation and hence value. Here, we discuss the impact of the coupling capacitance between adjacent bitlines on data-dependent failures in an open bitline DRAM architecture [66, 68].[1] We examine data-dependent failures in two groups based on the affected stage of cell operation.

During CHARGE SHARING. The voltage difference in two adjacent bitlines is responsible for data-dependent failures during the CHARGE SHARING phase. A capacitor starts to build up charge in the presence of a voltage difference between its two nodes, but remains quiescent otherwise. In the presence of a voltage difference in neighboring bitlines, coupling capacitance between adjacent bitlines builds up, providing an extra path for charge flow that perturbs the voltage in bitlines. Figure 4b depicts this effect, where we show three neighboring cells in a row. The middle cell is the victim that is in charged state (**Q**), and the two adjacent cells (left and right neighbors) are discharged (**0**). A data-dependent failure during the CHARGE SHARING phase occurs in three steps:

❶ Once the wordline is raised, charge flows out of the victim cell (**Q**), perturbing the bitline voltage to $\frac{1}{2}V_{DD} + \delta$. At the same time, charge flows from the bitline to discharged neighbor cells (**0**), perturbing the neighboring bitlines towards 0 ($\frac{1}{2}V_{DD} - \delta$).

❷ The coupling capacitor between the victim and neighbors now experiences the voltage $\frac{1}{2}V_{DD} + \delta$ at one node and $\frac{1}{2}V_{DD} - \delta$ at the other. Due to the increasing voltage difference between the neighboring bitlines, the coupling capacitance starts to increase.

❸ As a result, some charge from the victim bitline flows towards the parasitic capacitor, decreasing the bitline voltage from $\frac{1}{2}V_{DD} + \delta$ to $\frac{1}{2}V_{DD} + \delta - \varepsilon$. If the voltage decreases to the point that the sense amplifier senses the voltage perturbation as data value '0', it drives the victim bitline towards **0** instead of V_{DD}, resulting in a failure in the victim cell.

Note that this failure is data-dependent because it would not occur if the adjacent cells have the *same* initial data value. In that case, all neighboring bitlines experience similar changes in bitline voltage during the charge sharing phase (either $\frac{1}{2}V_{DD} + \delta$ or $\frac{1}{2}V_{DD} - \delta$ for all bitlines). The coupling capacitor experiences minimal voltage difference, and thus there is no effect on cell access.

During SENSING AND AMPLIFICATION. The time-delay in sensing two adjacent bitlines is responsible for data-dependent failures during the SENSING AND AMPLIFICATION phase. In the presence of a sudden voltage change in one node of a capacitor, the coupling effect changes the voltage of the other node in the same direction. During SENSING AND AMPLIFICATION, due to imperfect timing, if one bitline is sensed and amplified before the neighbors are sensed, the sudden change in the bitline voltage perturbs the neighboring bitlines, which can lead to a data-dependent failure. Figure 4c depicts this effect, where the middle cell is the victim in charged state (**Q**), and the two adjacent cells (left and right neighbors) are discharged (**0**). Once the wordline is raised, voltages in the bitlines are perturbed to $\frac{1}{2}V_{DD} \pm \delta$ depending on the initial state of the cells. A data-dependent failure during the SENSING AND AMPLIFICATION phase occurs in three steps:

❶ Due to time delay, the two neighboring bitlines are sensed and amplified towards **0** *before* the victim bitline is sensed.

❷ As a result, the coupling capacitor between the victim and the neighbor experiences a sudden voltage change towards **0** in the node connected to the neighbor bitline. Consequently, the other node of the capacitor connected to the victim bitline also experiences a similar voltage change towards **0**.

[1] All major DRAM vendors use the open bitline architecture, as it enables higher DRAM density.

❸ When the sense amplifier finally starts to sense the victim bitline, it senses a value less than $\frac{1}{2}V_{DD} + \delta$ and can wrongly sense it as data value '0', resulting in a failure in the victim cell.

This failure is data-dependent because it would not occur if the adjacent cells have the *same* initial data value. In this case, all the bitlines are driven in the same direction during the sensing and amplification phase. Therefore, voltage in the coupling capacitor increases or decreases the voltage of all bitlines towards the original values of the cells. Thus, neighboring cells with the same content do not lead to data-dependent failures.

Manufacturing Tests for Data-Dependent Failures. Manufacturers exhaustively test DRAM cells for data-dependent failures after production. First, they determine the data pattern that introduces maximum cell-to-cell interference through the coupling capacitor, which can lead to a data-dependent failure [3, 4, 19, 33, 36, 70, 71, 78]. This pattern is the *worst-case pattern*. Second, they test the chips with the worst-case pattern, while the cells store the minimum possible amount of charge, to make sure that cells are more vulnerable to failure. DRAM cells leak charge gradually and are refreshed periodically over time. A cell contains the minimum amount of charge just before refresh and is thus more vulnerable to data-dependent failures just before refresh [3, 35, 43, 47, 83]. Manufacturers test for data-dependent failures by writing the worst-case pattern in the cells and waiting for the end of the refresh interval before accessing the cells again. A data-dependent failure is detected if the data value read from a cell does not match the value originally written into the cell. The chip is either discarded or repaired if there is any data-dependent failure.

3. Challenges of System-Level Detection of Data-Dependent Failures

Traditionally, DRAM chips are tested for data-dependent failures at the manufacturing time. However, data-dependent failures are becoming more difficult to test for as DRAM cell size reduces. Recent works propose different system optimizations that can be enabled by detecting data-dependent failures while DRAM is being used in the field, during online system operation. As described in Section 1, such *system-level detection* of data-dependent failures can enable better DRAM scaling [6, 35, 47, 51, 59, 62], reliability [39, 53, 67, 74, 75], and latency and refresh reduction [18, 27, 43, 46, 48, 62, 80]. Unfortunately, detection of data-dependent failures at the system level faces two major challenges.

Challenge 1: Address Scrambling. The system can theoretically detect data-dependent failures by writing the worst-case pattern in the neighboring bit addresses in DRAM. Unfortunately, DRAM vendors scramble the system address space internally in the DRAM chip. As a result, neighboring addresses in the system address space do *not* correspond to neighboring cells in the physical cell array. The mapping of system address space to physical address space, which we call *address mapping*, is not exposed to the system. This mapping could be different for different vendors and generations of DRAM chips, making it difficult to expose it to the system. DRAM manufacturers might also not want to expose this mapping to the system as it might provide competitive information related to DRAM yield.

The reason behind this address scrambling lies in the cost-optimized organization of DRAM internals [70, 78]. During an access, DRAM data is buffered internally in multiple stages before it is sent over the bus through the IO pins [44]. To minimize cost, DRAM-internal buffers are organized in a hierarchical manner, where each level has a different number of entries. Data gets scrambled while passing from a wider to a narrower structure [70, 78].

Figure 5 shows two stages of buffering in a DRAM bank. First, an access to a DRAM bank activates an entire row (8K cells) in multiple cell arrays. The data of the activated row gets latched in the local sense-amplifiers (LSA). As each LSA's width is approximately twice the width of a cell, data from a row is latched in *two* different rows of LSAs [17, 68], one at the top and the other at the bottom of each cell array. Second, upon a read command, data is transferred from the LSAs to global sense-amplifiers (GSA) via long global bitlines. Each bank contains only a small number (64 – 128) of global bitlines and GSAs due to their large width and hence high cost. Because of this large mismatch in the number of LSAs and GSAs, only a fraction of data from the LSAs can be transferred to GSAs at a given time.

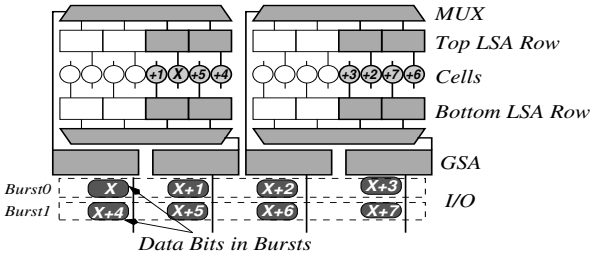


Figure 5: An example of address scrambling

Depending on how (i) cells are connected to LSAs and (ii) LSAs are connected to GSAs, different address mappings are possible in a DRAM chip. Figure 5 illustrates one example mapping. The system writes to bit addresses *X* to *X+7* in two bursts through the I/O pins. The number in a cell in the figure represents the address of that cell as a difference from address *X* (e.g., *X*: *X+0*, *+7*: *X+7*). When the first burst of data arrives at the IO pins, the data first gets distributed into two groups (*X*, *X+1*), (*X+2*, *X+3*), where each group is buffered in GSAs connected to different cell arrays. Then, depending on the way top and bottom LSAs are connected to cells, data bits get swapped when transferred from LSAs to cells (*X*, *X+1* to *X+1*, *X*). Similarly, data in the second burst ends up in different arrays swapped as (*X+5*, *X+4*) and (*X+7*, *X+6*).

As shown in the figure, finally, the neighboring cells of *X* get data from system address bits *X+1* and *X+5*. Due to this internal address scrambling, testing for data-dependent failures with neighboring system addresses (*X*, *X+1*, *X+2*) would *not* test physically neighboring cells within DRAM.

Challenge 2: Long Test Time. Without knowledge of address mapping, detecting data-dependent failures at the system level is hard, as the system does *not* know the addresses of physically neighboring cells. Many prior works that depend on system-level detection of data-dependent failures assume that simple tests with all 0s/1s data patterns can detect data-dependent failures. Unfortunately, prior studies [6, 47] show that a large fraction of failures remain undetected with such simple tests. As a result, mechanisms built on top of such simple tests are impractical as they would face severe reliability issues. One prior work [35] tries to solve this problem by testing DRAM with a large number of random data patterns. Many rounds of tests with random patterns increase the probability of discovering more failures by increasing the likelihood of writing the worst case pattern into the addresses mapped to physically neighboring cells [35]. Unfortunately, system-level testing with random patterns take very long, are expensive, and make it difficult to provide any guarantees on the fraction of data-dependent failures that remain undetected [35] (as we will quantitatively show in Section 7).

One alternate way to uncover data-dependent failures is to determine the location of physically neighboring cells in the system address space and use that information to ensure that the physically neighboring cells are tested with the worst-case pattern. This is a promising approach, since each generation of chips can have different address mappings: a technique that

can quickly learn that mapping is applicable to any system with any chip. As a cell receives the most interference from its two immediate neighbors [1, 60], it is possible to devise an exhaustive test to determine the locations of these two neighbors.

For each potential failing cell (victim), such a test would exhaustively test all combinations of two bit addresses in a DRAM row. The victim would fail when the addresses of the neighbors are being tested, indicating where the neighbors are located in the system address space. For example, when the cell at address X in Figure 5 is tested, out of all two bit address combinations within the row, the cell at X would fail only when content in addresses $X+1$ and $X+5$ is set to the worst case pattern, revealing the system addresses of immediate neighbors. Unfortunately, selecting *every* combination of 2 bit addresses from n bit addresses requires $O(n^2)$ tests and takes 49 days of testing for a row with $n = 8K$ cells (Discussed in Appendix). As cells get smaller and more vulnerable to cell-to-cell interference, it is likely that potentially more neighboring cells will affect each other in the future [2]. Determining the location of k-neighboring cells requires $O(n^k)$ tests, increasing the test time to 1115 years/9.1M years for detecting physical addresses of three/four neighbors. Clearly, it is not feasible to run tests for such a long time.

Our goal in this work is to develop a fast and efficient testing method to determine the locations of physically neighboring cells in the system address space and use that information to detect data-dependent failures in DRAM.

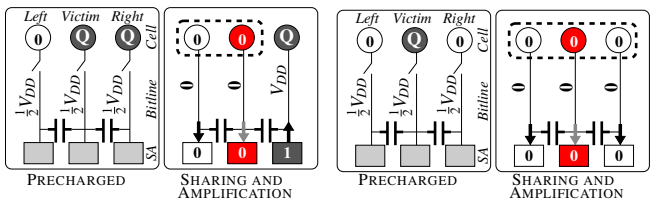
4. Key Ideas

In this work, *(i)* we develop an efficient system-level test method to detect data-dependent failures in DRAM, *(ii)* experimentally demonstrate the effectiveness of our mechanism using real DRAM chips, and *(iii)* discuss new uses cases of our mechanism to improve system reliability and performance. Our system-level test method for determining the location of neighboring cells builds upon two key ideas.

4.1. Key Idea 1: Exploiting Strongly Coupled Cells

Our first key idea is based on the observation that cells affected by bitline coupling due to parasitic capacitance between neighboring bitlines can be divided into two groups based on their *sensitivity* to coupling: *Strongly* vs. *Weakly* coupled.

(i) Strongly Coupled Cells. In Section 2, we discussed that a victim cell experiences the largest interference when it is surrounded by neighboring cells containing the opposite data value (e.g., charged victim cell (**Q**) surrounded by discharged (**0**) neighbors). However, due to process variation, some cells experience a large enough interference to cause a data-dependent failure even when *only one neighbor* has the opposite value. We call such cells *strongly coupled cells*. Figure 6a illustrates a strongly coupled victim cell that fails based on the data content of only the left neighbor, irrespective of the content of the right one. A strongly coupled cell can be coupled with either the left or the right neighbor.



(a) *Strongly coupled cell* (b) *Weakly coupled cell*

Figure 6: Cells have different sensitivity to coupling

(ii) Weakly Coupled Cells. For these cells, the content of only one neighbor *cannot* make the victim fail. Only when both neighbors contain the opposite data value than the victim, the victim bitline experiences a large enough interference that makes the cell fail. Figure 6b shows an example of a weakly coupled cell.

Leveraging our observation, our key idea to reduce test time is to locate the address of *only one neighboring cell* of a strongly coupled cell. Instead of detecting the addresses of both neighbors by testing every combination of *two bit addresses* in a DRAM row, each address now has to be tested linearly *just once*. This approach has two major advantages. First, it reduces the test time from quadratic $O(n^2)$ to linear $O(n)$, where n is the number of cells in a row. Second, it makes other well-known optimizations applicable to this linear test. In this work, we apply a *recursive* test to further reduce test time. Instead of testing one bit address at a time, we divide the address space of an entire row into smaller regions and test all addresses in one region *at once*. Only the region that contains the neighbor address is tested in the next levels of the recursion. When the region size becomes one, we find the exact bit address of the neighboring cell. This divide and conquer mechanism reduces our test time to $\theta(n)$ (See details in Appendix).

4.2. Key Idea 2: Exploiting DRAM Regularity and Parallelism

Our first key idea to detect the address of only one neighbor significantly reduces test time. Unfortunately, it decreases the coverage of failures, as without the knowledge of *both* neighboring addresses, it will not be possible to detect *all* data-dependent failures (i.e., weakly coupled cells). Our second key idea enables us to detect both neighboring locations only with simple linear tests. The idea is based on the observation that DRAM is internally organized as a 2D array of *similar and repetitive tiles*. The regularity in tiles results in regularity in address mapping *within and across* the tiles.[2]

Figure 7 shows a bank with 32K rows and 8K cells per row, where cells are divided into smaller tiles. Two cells located at the same column of a tile likely have the same address mapping for their neighboring cells. For example, the system address offsets of the two neighboring cells of each of X, Y and Z are $+1$ and $+5$, in Figure 7.

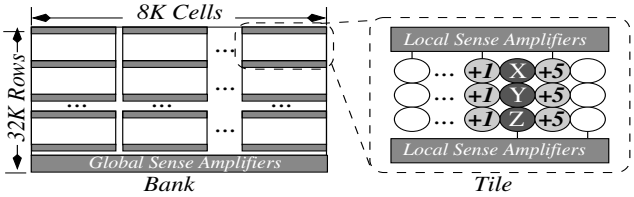


Figure 7: Regularity in address mapping across rows

We propose to utilize this regularity in mapping by running parallel tests in different rows *simultaneously*. Due to randomness in process variation, some strongly coupled cells will fail depending on the content of the right neighbor and some on left. As the mapping is the same in different rows, even if the cells tested are located in different rows, the location of the neighboring cells will be *the same*. By running *parallel tests* in different rows *simultaneously*, we can detect the system address distance of both the left neighbor and the right neighbor for different strongly coupled cells, which together serve as the distances of the left and right neighbors for weakly coupled cells. Consequently, it is possible to estimate the addresses of *all* neighboring cells by testing multiple cells in different rows *simultaneously* and *aggregating* the distances of the neighboring locations found in those rows. For example, in Figure 7, the

[2] A very small fraction of faulty cells are remapped in DRAM [28, 41, 78]. Section 7.3 discusses how remapped cells can be handled.

test of some rows would find *+1* as the neighbor distance (for strongly coupled cells that are coupled with their left neighbors) and the test of some other rows would find *+5*. Aggregating the single neighbor distances found in all rows would provide both distances as the locations of neighboring cells.

5. PARBOR: System-Level Parallel Recursive Neighbor Testing

Based on our two key ideas, we propose an efficient system-level technique for detecting the locations of neighboring cells, called **PA**rallel **R**ecursive neigh**BOR (PARBOR)** testing. We first present the high-level design of PARBOR, and then discuss design challenges and our solutions to solve those challenges.

5.1. Design Overview

At the high level, PARBOR first determines the location of neighbors in the system address space by simultaneously testing multiple rows containing data-dependent failures and aggregating the neighbor locations found in each row. Next, PARBOR uses this information to test every cell in DRAM by applying the worst-case data pattern to the neighboring cells and detects all data-dependent failures. PARBOR has five major steps.

❶ Construct an initial set of victims that fail depending on the data content in the neighboring cells (Section 5.2.1).

❷ Simultaneously test all the rows containing the initial set of victims. Test each row recursively by dividing it into smaller subregions until PARBOR finds the bit addresses of the neighboring cells (Section 5.2.3).

❸ Aggregate the locations of neighboring cells found in each row. Due to the regularity of DRAM tiles, the neighbors of different victim cells are found at common regular distances from each victim cell's system address (Section 5.2.2). Thus, the *union* of all distances found in step ❷ for the initial set of victims provides a set of possible distances where the physically neighboring cells can be located for any cell in the chip.

❹ Filter out the random (non-data-dependent) failures that occurred during the tests, as these failures can potentially include some non-neighbor locations in the set of possible distances determined in step ❸ (Section 5.2.4).

❺ Use the information on neighbor distances to detect data-dependent failures in the entire chip by ensuring that all physically neighboring cells are tested with the worst-case pattern. Test independent bits in the address space simultaneously to reduce the test time (Section 5.2.5).

5.2. Detailed Design

PARBOR has five design challenges. *(i)* How to determine the initial set of victim cells that exhibit data-dependent failures? *(ii)* How to represent the neighbor locations efficiently for all cells? *(iii)* How to recursively test a row? *(iv)* How to identify and filter out random (non-data-dependent) failures during the tests? *(v)* How to efficiently use the location information of neighboring cells to test the entire chip for data-dependent failures? Next, we provide a full overview of these challenges and describe our proposed mechanisms that address these challenges.

5.2.1. Determining the Initial Set of Victim Cells

PARBOR relies on a set of known victim cells that exhibit data-dependent failures to determine the locations of the physically neighboring cells. This initial victim set is needed as testing for neighbor addresses in *any* random cell will likely not work, as that cell might *not* be vulnerable to failure at all. To determine the initial set of victim cells, PARBOR first tests a DRAM chip with multiple different data patterns. A cell that operates correctly for one pattern, but fails when the content in the row changes with a different pattern is *likely* to exhibit data dependence.[3] PARBOR includes such a cell in the initial set of victim cells. We demonstrate the effect of the size of this initial set in Section 7.3.

Note that accurately detecting *data-dependent* failures is challenging as it is difficult to resolve the root cause behind failures. For example, some *weak cells* fail irrespective of the content of the neighbors [47], *soft failures* occur randomly across cells [7, 76], *VRT cells* fail when charge randomly gets trapped in the gate of the transistor and causes the capacitor to leak charge faster [47, 55, 65, 85], etc. Thus, when a failure is found (during the determination of the initial set of victim cells), its cause might not be data dependence. We resolve this issue by making PARBOR robust to such *random* failures. PARBOR first detects a set of failures (i.e., the initial set of victim cells) that are *likely* to be caused by data dependence. It later *filters out* random failures from this set (Section 5.2.4).

5.2.2. Representation of Neighbor Addresses

PARBOR determines the physically neighboring cell addresses by running simultaneous recursive tests in multiple rows to exploit the regularity in DRAM chips. Representing the neighbor addresses efficiently is important as recording the neighbor address for *every bit* in a row will incur a large overhead. PARBOR simplifies the representation of neighbor addresses by expressing each neighbor address as a *distance* (or, offset) from the victim cell. We observe that, due to regularity in address mapping, the distance of any physically neighboring cell from the victim can be represented with *a limited set of numbers*. Figure 8 demonstrates the possible distances of any cell with the example mapping discussed in Figure 5 of Section 3. The system address for the cells in the row runs from 0 to 15, as shown below the cell array. The figure shows that all physically neighboring cells are located at a distance of $\{\pm1, \pm5\}$, irrespective of the absolute address of the victim cell. This *distance-based representation* not only eliminates the necessity to record every single neighbor address, but also simplifies the recursive algorithm, which we discuss next.

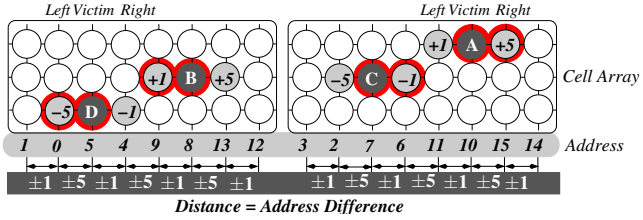


Figure 8: Representing neighbor addresses as *distances*

5.2.3. Recursive Test Algorithm

PARBOR recursively divides a row into smaller regions. The region that contains the victim address is called the *victim region*. Each region is tested to determine if the neighbor address belongs to that region. To test a region, opposite data values are written to the victim address and to the bits in the test region. This ensures that if the neighbor cell is located in the test region, victim and neighbor cells contain the worst-case pattern. If the victim is strongly coupled to the neighbor cell, data value written to the victim will flip due to interference from the neighbor cell. When data is read back from the victim region and compared to the original values written, victim data will result in a mismatch. The region that contains the address of the neighbor that causes the victim to fail is referred to as *neighbor region*. We represent the regions using distance, too. If the neighbor address

[3] DRAM cells can store data value 1 as either charged state or discharged state [47]. To make sure that both types of cells are tested, every pattern tested in PARBOR is also accompanied with the inverse pattern.

is located within the victim region, then the distance is 0; if neighbor address is located in the immediate neighbor region, then distance is ±1, and so on. These neighbor distances found in multiple rows represent possible neighbor locations in the entire chip. The reason is that, due to the regularity in mapping, the left or the right neighbor locations for all cells in the chip can be represented with a limited set of distances, as shown in Figure 8. In order to find all possible neighbor locations, each neighbor distance found in multiple rows is aggregated at each level. In the next level of the recursive algorithm, only those regions where a neighbor is found are subdivided into smaller regions and tested recursively until the region size becomes one. The aggregate list of all found distances provide the neighbor locations for any cell in the entire chip.

We provide a simple example of our recursive testing method. PARBOR starts with an initial set of victims that exhibit data-dependent failures ({A, B, C, D}), as shown in Figure 8. The victims are strongly coupled with one of their neighbors, denoted by red circles in the figure. The distance between the victim and neighbor address is shown within the cell. In this example, PARBOR divides a region into *equal halves* at every level of the recursive algorithm. As the address space is 16 bits, the region size becomes 8 in the first level of the recursion. This region is recursively divided into smaller regions of sizes 4, 2, and 1 in the next levels. In Figure 9, we show the steps of the recursive test for cell D, located at address 5. Each level of the recursive test is labeled as L1, L2, and so on. At each level, the neighbor region found is shown with a *red* outlined box. Figure 10 shows the union of distances found at each level for *all* victim cells.

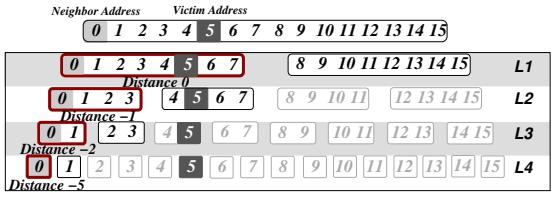


Figure 9: Recursive test for cell D

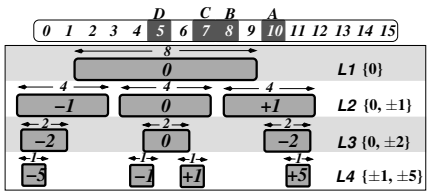


Figure 10: Union of *distances* at each level for A, B, C, and D

❶ At the first level of the recursion (L1), the entire row is divided into two regions, each containing 8 bits: Region 0 (address bits 0–7) and Region 1 (address bits 8–15). D fails while Region 0 is tested. As D also belongs to Region 0, PARBOR determines that the distance between the victim and neighbor region for D is 0. By testing the other rows simultaneously, PARBOR finds that all victims fail at distance 0 at this level.

❷ At the next level, PARBOR divides the neighbor regions found in level 1 into two smaller regions of size 4. For cell D, these two regions contain address bits 0–3 and 4–7 (as shown in the second level in Figure 9, denoted by L2). The neighbor of D is located at address 0, which resides in the region at distance -1. Therefore, D fails while testing the region at distance -1. On the other hand, cell A is located at the region with address bits 8–11 and the neighbor (address 15) is located in the next immediate region with address bits 12–15. Therefore, A fails at distance +1. Similarly, PARBOR determines that B and C fail at distance 0. PARBOR creates a union of the distances at level 2 as {0, ±1} (as shown at L2 in Figure 10), At the next level, only the regions at these distances are tested.

❸ PARBOR keeps dividing the neighbor regions until the region size becomes 1 at level 4. At this level, D fails while testing the region located at distance -5. Thus, we can deduce the location of the neighbor bit address. Note that the union of distances found for only four cells (A, B, C, and D) at this level ({±1, ±5}) provides the possible distances of neighbors for any cell in this chip.

5.2.4. Filtering Random Failures During Tests

PARBOR starts the recursive tests with an approximate set of data-dependent failures. However, some of these failures might be caused by other phenomena (for example, soft failures [7, 76], VRT [55, 65, 85], wordline coupling [39, 64], etc.). Failures might also occur due to *marginal cells* that hold barely enough charge to reliably store data until the end of the refresh interval. Content in these marginal cells are sometimes read correctly, sometimes not. These random failures interfere with PARBOR during the recursive testing. A random failure may occur while testing some region at a level, which wrongfully indicates that region as a neighbor region, even though the region may not contain any physically neighboring cells. PARBOR has to be robust to such random failures, as our goal is to accurately detect the neighboring locations of *data-dependent cell failures*. In order to eliminate the noise induced by random failures, PARBOR takes two steps. First, PARBOR discards the distances found due to a victim cell that repeatedly exhibits failures while testing *most of the regions*. These are likely to be marginal failures as opposed to data-dependent ones. Second, PARBOR *ranks* the distances of neighboring regions at each level. It only considers the distances that occur very frequently. Due to regularity in mapping, victims are likely to fail at a limited, common set of distances. Hence, infrequent distances are likely to occur due to random failures. We provide the frequency of different distances found in real chips in Section 7.3, and demonstrate how *ranking* helps to avoid noise from random failures.

5.2.5. Finding Data-Dependent Failures in the Entire Chip

Once the addresses of the physically neighboring cells are known, well-known test methods, such as neighborhood pattern-sensitive fault (NPSF) tests [19, 77]), can be applied. These tests identify hard faults depending on the data stored in nearby memory cells. Efficient algorithms for detecting NPSFs are based on March algorithms, which repeat a set of operations for each bit. However, testing one bit at a time is not cost-effective at the system level, as multiple data bits are transferred in parallel (typically 8/16 bits per chip) from DRAM to the processor. In order to take advantage of parallel data transfer in real systems, we devise *neighbor location-aware* data patterns that can simultaneously test different independent parts of the row and reduce the number of required tests. The system can test for a data-dependent failure by writing the worst-case pattern in the victim cell and at the neighboring addresses. We observe that multiple victim cells and their potential neighbors can be tested simultaneously, if they do not interfere with each other.

At the end of the recursive algorithm, PARBOR already has a set of distances that contain the possible locations of neighboring addresses. We use this information in two ways to test independent parts of rows in parallel. First, PARBOR determines the maximum distance between a victim and the neighbor, as any address located outside that region does not interfere with the victim cell and therefore, can be tested at the same time. PARBOR divides row addresses into *chunks*, where the length of a chunk is equal to the maximum distance between the victim and neighbors, and tests each of these chunks in parallel. For example, if the neighboring cells are located at distance ±8, each 16-bit chunk in a row can be tested at once. However, within

each chunk, each bit is tested serially. Therefore, this method would require 16 rounds to test each bit in a chunk of size 16. Second, within each chunk, only the victim and the neighbor are dependent. Bits in-between these two locations in the system address space can be tested independently at the same time. In this example, as the neighbor is located at distance 8, the first 8 bits in the 16-bit chunk (bit 0–7) do not interfere with each other and are independent. Therefore, instead of testing one bit at a time within the chunk, the first 8 bits within the chunk are tested at once. As a result, each chunk can be tested in two rounds: the first 8 bits in round 1 and the second 8 bits in round 2. As all the chunks in the system address space are tested in parallel, all cells in the chip can be tested in two rounds.

We conclude that PARBOR is an efficient mechanism that determines the location of physically neighboring cells with a recursive test and uses that information to test the entire chip for data-dependent failures with a small number of neighbor-aware data patterns.

6. Experimental Design and Infrastructure

In order to determine the efficacy of PARBOR in finding the neighboring bit locations and uncovering data-dependent failures, we test PARBOR with real DRAM chips using an FPGA-based infrastructure. Similar to our prior works that studied DRAM failures in off-the-shelf chips [18, 35, 39, 43, 47], our infrastructure consists of a Xilinx ML605 board that is connected to a host PC using a PCIe bus [84]. PARBOR is implemented as a system-level test in the host, interfaced with the memory controller to read and write DRAM modules.

Temperature. The amount of time DRAM cells can retain charge largely depends on the operating temperature and almost halves with every 10 °C increase in temperature [35, 39]. As data-dependent failures get affected by the amount of charge in the victim cells, our experiments isolate the effect of the temperature using a temperature-controlled heat chamber. PARBOR is targeted for systems running in the field, which usually operate at the temperature range of 20 °C–60 °C [24, 43, 49]. Consequently, we run our experiments at 45 °C, with sensitivity tests at 40 °C and 50 °C. We find that neighbor locations determined by PARBOR are *not* dependent on temperature.

Refresh Interval. A large refresh interval makes a cell more vulnerable to failures. Prior works test DRAM at a large refresh interval to model increased cell vulnerability in the future [29, 35, 47, 62]. In this work, PARBOR operates at a higher refresh interval of 4 s (4 s at 45 °C corresponds to 328 ms at 85 °C [47]). We expect that our results will hold at other refresh intervals as prior works show that data-dependent failures exhibit similar characteristics across different refresh intervals [35, 47].

DRAM Modules. We tested PARBOR using 18 DRAM modules containing 144 chips from three different vendors. All modules have a capacity of 2 GB and were manufactured within the last five years (2011–2014).

Source Code and Data Release. We will release the source code of PARBOR and data for all DRAM chips we tested at [61].

7. Results and Analysis

We first show the efficacy of PARBOR in *(i)* finding the distances in the system address space of physically neighboring locations of any cell and *(ii)* uncovering data-dependent failures. Then, we provide detailed analyses of PARBOR.

7.1. Determining Neighbor Locations with PARBOR

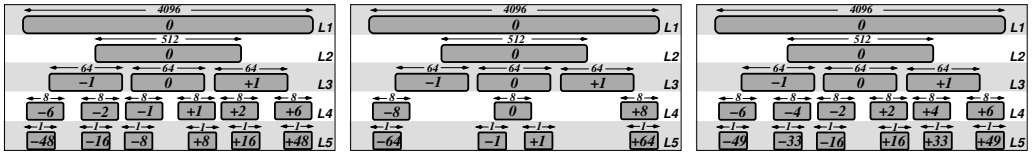
We provide *(i)* the implementation details of the recursive algorithm used in our experiments, *(ii)* neighbor address distances found by PARBOR at each level of the recursion, and *(iii)* the number of tests performed during the recursion.

Details of the recursive algorithm. The tested modules in this work all have 8K cells in a row. In our experiments, PARBOR recursively tests 8K cells in a row by dividing the row into smaller regions until the region size becomes 1. PARBOR uses 5 levels of recursive testing to get to a region size of 1 from a row size of 8K. The row is divided into two regions of size 4096 bits at the first level. At the remaining levels, each region is divided into eight subregions (of sizes 512, 64, 8, and 1 for levels 2 to 5, respectively).

Neighbor distances at each level of the recursion. Figure 11 shows the union of distances of neighbor locations found by PARBOR at each level of the recursion for modules from three different vendors. We found that all modules from a specific vendor and generation exhibit the *same* distances. Figure 11a shows that in the first two levels (L1 and L2), when the row is divided into regions of 4096 bits and 512 bits respectively, modules from **A** always have the neighbor region at distance 0. In the next levels, neighbor regions are found at multiple distances. For example, neighbors are found at distances {0, ±1} at L3, and at distances {±1, ±2, ±6} at L4. Finally, at the last level (L5), when region size becomes 1, neighbors are found at distances {±8, ±16, ±48}. This set represents the possible distances of the physically neighboring cells for any cell in the system address space. We observe that neighbor distances found by PARBOR are different across vendors. For example, distances of neighbor cells for **A** is {±8, ±16, ±48}, whereas it is {±1, ±64} for **B**, and {±16, ±33, ±49} for **C**. We conclude that PARBOR is capable of determining neighbor locations across different vendors and generations.

Number of tests for the recursive algorithm. PARBOR takes the union of neighbor distances found for all victim cells at each level of the recursive test. Neighbor regions found at each level are divided into smaller regions, and these smaller regions are tested serially at the next level to determine which of them contains the neighbor location. As a result, the number of tests at each level directly depends on the number of neighbor regions found at the prior level. If the number of tests at level i is t_i, the number of neighbor regions found in the prior level is N_{i-1}, and each region is divided into S_i subregions at level i, then $t_i = N_{i-1} * S_i$.

Table 1 shows the number of tests performed at each level for modules from three different vendors. At the first level, the entire row is divided into two regions of size 4096 bits, and therefore requires two tests (shown in the column labeled as L1 in the table). Figure 11 shows that for modules from **A**, PARBOR always finds the neighbor region at distance 0 at L1 and L2. As a result, it subdivides this region into 8 smaller regions at the next levels and therefore, requires 1*8=8 tests at L2 and L3, respectively. At L3, for the same modules, PARBOR finds three neighboring regions at distances {0, ±1}. Therefore,



(a) *Modules from* ***A*** (b) *Modules from* ***B*** (c) *Modules from* ***C***

Figure 11: Distances of neighbor regions at each level for modules from A, B, and C

it requires 3*8=24 tests at the next level (L4). The rightmost column of Table 1 shows that the total number of tests performed by PARBOR is 90/66/90 for modules from vendors **A/B/C**, a 90X/745,654X reduction compared to tests with $O(n)/O(n^2)$ runtime. We conclude that PARBOR determines the neighbor locations very efficiently with a small number of tests.

Manufacturer	*L1*	*L2*	*L3*	*L4*	*L5*	Total
A	2	8	8	24	48	90
B	2	8	8	24	24	66
C	2	8	8	24	48	90

Table 1: Number of tests performed by PARBOR

We also find that different modules from a given vendor require the *same* number of tests in our experiments (not tabulated). We believe this is because each vendor may have their own unique address scrambling mechanisms and internal designs that last across multiple DRAM generations.

7.2. Uncovering Data Dependent Failures with PARBOR

After discovering the distances of neighbor locations, PARBOR utilizes this information to develop generalized neighbor-aware test patterns for the entire chip. We provide *(i)* the number of neighbor-aware data patterns for different modules in our experiments, *(ii)* the number of data-dependent failures detected using those patterns, and *(iii)* the fraction of failures detected by PARBOR compared to all known failures (coverage).

Number of neighbor-aware test patterns. As discussed in section 5.2.5, PARBOR generates neighbor-aware data patterns such that independent regions of a row that do not interfere with each other can be tested simultaneously. In our experiments, all tested DRAM chips have neighboring locations within distances of ±64. Consequently, PARBOR can test each 128-bit chunk in a row at the same time.[4] PARBOR further reduces the number of test data patterns when addresses within a 128-bit chunk do not interfere with each other. For example, neighboring cells in modules from **A** are located at distances ±48, ±16, and ±8. As a result, each set of 8 bits in the 128-bit chunk do not interfere with each other and therefore, are tested in parallel for **A**, requiring a total of 16 rounds of tests. In order to test both true and anti cells [47], each pattern is accompanied with the inverse pattern. Therefore, 2*16=32 rounds are required to test every bit in the entire chip for modules from **A**. In our experiments, modules from **B** and **C** require a total of 32 and 16 rounds of tests, respectively.

Increase in detected failures. Figure 12 shows the additional failures detected by PARBOR using the neighbor-aware data patterns compared to tests with random patterns that are unaware of the neighboring locations. We keep the number of performed tests constant in this figure, i.e., the number of random pattern tests is limited to the number of tests performed by PARBOR. The total number of tests in PARBOR is the sum of *(i)* recursive test to determine the neighbor locations (66-90), *(ii)* tests with neighbor-aware patterns (16-32), and *(iii)* initial tests for locating sample victim bits (10). This amounts to 92-132 tests depending on vendor, which takes 38-55 seconds for a 2GB module with a refresh interval of 64 ms (See Appendix for details). The bars in the figure depict the new failures detected by PARBOR and the lines depict the percentage increase in detected failures with PARBOR.

We make two observations from Figure 12. First, PARBOR identifies 1K to 45K *more* failures in every tested module, resulting in a 2–55% increase in total detected failures. Second, the absolute number of failures detected by PARBOR is different

[4]Note that these 128 bits belong to a single chip. As our tested modules access 8 bits per chip in a burst, accessing these 128 bits requires 16 bursts.

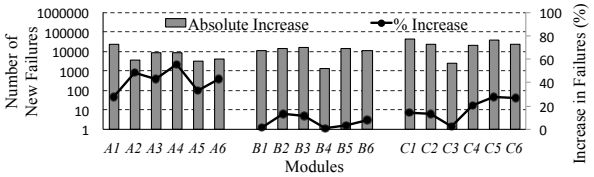
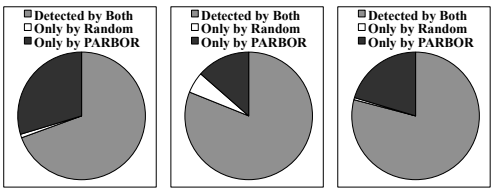


Figure 12: Extra failures uncovered using PARBOR

across vendors and depends on the vulnerability of the chips in the module. For example, on average, modules from **C** are more vulnerable to data dependent failures compared to modules from **A** and **B** (Notice the log scale). We conclude that PARBOR can effectively uncover more data-dependent failures in DRAM by testing with neighbor-aware data patterns instead of random data patterns.

Coverage of failures. We examine the fraction of failures detected by PARBOR compared to all failures uncovered by both PARBOR and random pattern tests. Figure 13 shows the percentage of failures in three specific modules detected by *(i)* only PARBOR, *(ii)* only random tests, and *(iii)* both PARBOR and random tests. We make two major observations from this figure. First, a significant fraction of failures are detected *only* by PARBOR (20-30%). Second, only a small fraction of failures remains undetected by PARBOR (less than 1% for Module A_1 and C_1 and around 5% for Module B_1).



(a) *Module* A_1 (b) *Module* B_1 (c) *Module* C_1

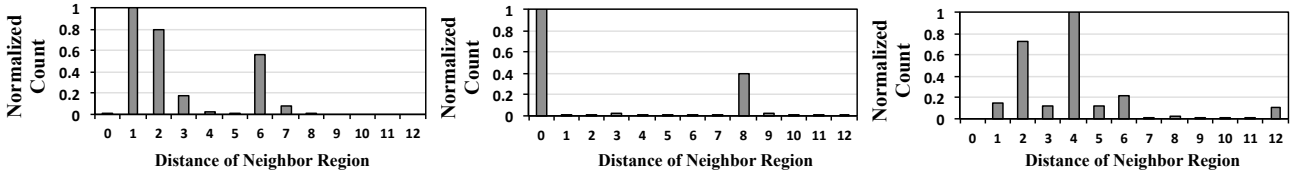
Figure 13: Coverage of failures for A_1, B_1, and C_1

There are two possible sources for failures that occur only in tests with random patterns. First, prior works have shown that DRAM cells are vulnerable to *randomly-occurring failures*, such as soft failures [7, 76], VRT [55, 65, 85], etc. As these failures occur randomly, it is possible that such failures happened to occur during random pattern tests and not during PARBOR tests. Second, PARBOR depends on the regularity of physical address mapping in DRAM. However, a small fraction of faulty columns are remapped to some redundant columns available in the cell array, which reduces regularity in physical address mapping in DRAM. As such redundant columns may have neighbor cells located at different positions than regular cells, PARBOR cannot detect data-dependent failures in these cells. As the number of failures not detected by PARBOR is relatively small, we conclude that our neighbor-aware test method effectively uncovers more failures than testing with random patterns.

7.3. Sensitivity and Analysis

As discussed in Section 5.2.4, PARBOR *ranks* the distances found at each level of the recursive algorithm based on the frequency of distances. It eliminates the *infrequent distances* from consideration, to avoid detecting the random, non-data-dependent failures that occur throughout the tests. In this section, we *(i)* show how these random failures impact the frequency of distances, *(ii)* demonstrate the effect of the size of initial set of victim cells, i.e., sample size, on the ranking of distances, and *(iii)* discuss the limitations of PARBOR.

Avoiding random failures with ranking of distances. Figure 14 shows the ranks of regions at level 4 for three different modules. Each bar at distance X in the figure represents the number of times a neighbor region at distance ±x is discovered,

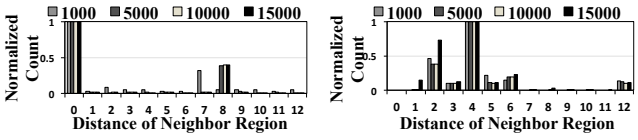


(a) *Module* A_1 (b) *Module* B_1 (c) *Module* C_1

Figure 14: Ranking of regions in recursion level 4 for A_1, B_1, and C_1

normalized to the most frequent distance. We observe that some distances occur very frequently (for example, distances ±1, ±2, ±6 in Module A_1). These are the regions where the neighbor cells are very likely located. On the other hand, some distances occur relatively infrequently (for example, distances ±3 and ±9 in Module B_1). These distances are most likely noise caused by random failures. By not considering such distances that occur very infrequently, PARBOR eliminates the effect of random failures from the determination of distances of neighboring cells. We conclude that ranking of distances is an effective way to filter out the noise caused by random failures.

Effect of the size of the initial set of victim cells (i.e., sample size). PARBOR identifies an initial set of victim cells and determines the neighboring locations for these cells with the recursive testing it employs. The sample size of this initial set can affect the ranking of distances performed at each level of the recursive test. For example, a small set of victim cells might not be enough to differentiate data-dependent failures from random failures that are due to non-data-dependent reasons. Figure 15 shows the effect of sample size on ranking. We perform this sensitivity study in two modules (Module B_1, and C_1) with four different sample sizes (1K, 5K, 10K, and 15K). We observe that for some modules (e.g., B_1), there is a clear partition between frequent and non-frequent regions: the amplitude of the frequent regions is much higher than that of the others. Therefore, random failures do not affect the frequent regions, as we vary the sample size of initial set of victim cells. On the other hand, for some modules (e.g., C_1) the difference between frequent and non-frequent regions is not as clear. A small sample size might negatively affect the ranking of distances in such modules. For example, distance 5 in module C_1 is frequent when testing with a small sample size (1K), but, with a larger sample size, it is clear that this distance is not frequent relative to others. We conclude that a larger sample size for the initial set of victim cells makes ranking more robust to random failures and thus PARBOR more accurate in identifying data-dependent failures.



(a) *Module* B_1 (b) *Module* C_1

Figure 15: Ranking for B_1 and C_1 with different sample sizes

Limitation. PARBOR depends on the regularity of DRAM organization to determine the locations of the neighboring cells. Unfortunately, a small number of cells in DRAM chips can have neighboring cells in different distances than regular cells. These cell are remapped in redundant columns to correct permanent faults in the cell array [28, 41].[5] As the fraction of remapped cells is quite low [78], such cells do not greatly affect the coverage of our mechanism. However, as cells become more vulnerable with scaling, it is likely that DRAM chips will include more redundancy in the future, and PARBOR might become less effective in detecting all data-dependent failures.

We provide a simple way to extend PARBOR to detect data-dependent failures in remapped columns. If a large fraction of cells gets remapped in the future, the initial set of victim cells will likely include some of the remapped cells. Therefore, neighbor regions for these cells will show up as infrequent regions in the ranking stage of PARBOR. In the current implementation, PARBOR discards infrequent neighbor distances to filter out random failures. By taking into account these infrequent regions in intelligent ways, it would be possible to detect the neighboring locations of remapped cells.

8. A New Use Case for PARBOR

PARBOR enables existing mechanisms that require detecting data-dependent failures in the system [6, 35, 46, 47, 48, 59, 80]. On top of that, PARBOR can also enable new mechanisms to improve DRAM reliability, performance, and energy efficiency. We propose one such new use case for PARBOR.

Data content-based refresh. Existing refresh optimization techniques minimize refresh operations by using a lower refresh rate for rows that can reliably retain data at that refresh rate [46, 48, 62, 80]. These works rely on detecting *weak cells* that have low retention times, so that they can refresh the rows such cells reside in at a higher refresh rate, to avoid data loss. However, these techniques do *not* take into account the fact that retention failures in such weak rows occur *only with the worst-case pattern*. Instead, they *always* refresh weak rows frequently. In reality, application data does not always contain the worst-case pattern in weak rows. Therefore, there is an opportunity to further reduce the refresh rate based on the *current data content* of DRAM rows.

We propose a new refresh reduction technique, called Data Content-based REFresh (*DC-REF*). The *key idea* of DC-REF is to employ a high refresh rate *only* when the data content of the row exhibits the worst-case pattern. DC-REF uses PARBOR to determine the location of data-dependent failures and the worst-case pattern that causes the failures. When there is a write to a row containing a cell vulnerable to data-dependent failure, the new data content is checked against the worst-case pattern. *If and only if the new content matches the worst-case pattern*, the row is designated to be refreshed frequently.

Evaluation of DC-REF. We evaluate the performance of DC-REF with Ramulator [40], an open-source cycle-accurate DRAM simulator [63]. We use representative phases of 17 SPEC CPU2006 [73] applications as CPU traces [50]. We evaluate 32 8-core multi-programmed workloads by randomly assigning one application to each core. Our simulation executes at least 256 ms on each core. We report performance as *weighted speedup* [25, 72]. Table 2 shows the simulation parameters.

Figure 16 illustrates the system performance improvement of DC-REF over the baseline system with a uniform refresh interval of 64 ms. On average, DC-REF improves system performance by 18.0% for 32Gbit DRAM across all 32 workloads.[6] Compared to RAIDR [46], a refresh reduction technique that refreshes all rows with weak cells using a high refresh rate, irrespective of the data content of such rows, DC-REF reduces refresh operations by 27.6% and improves performance for 32Gbit

[5] Remapped rows will have the same mapping as the rest of the cell array.

[6] We estimate tRFC (refresh latency) for 16/32Gbit DRAM as 590ns/1us, similarly to the tRFC estimation in [46].

Component	Main Parameters
Processor	8 cores, 3.2GHz, 3-wide issue, 128-entry inst. window
Last-level cache	64B cache-line, 16-way associative, 512KB private cache-slice per core
Memory	DDR3-1600 [31], 2 channels, 2 ranks-per-channel
Refresh Interval	RAIDR [46]: 64/256 ms for 16.4%/83.6% rows DC-REF: 64 ms for rows with the worst-case pattern, 256 ms for the remaining rows

Table 2: Configuration of simulated systems

DRAM by 3.0%. Note that the fraction of weak cells in RAIDR is collected from real chips, using our FPGA-based infrastructure. RAIDR always refreshes 16.4% of the rows with a high refresh rate, while DC-REF reduces that fraction to only 2.7% on average over 17 SPEC benchmarks.

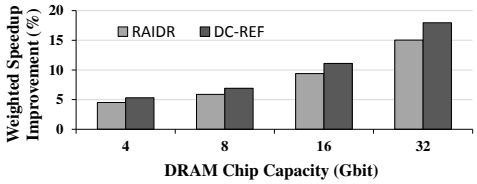


Figure 16: Performance of DC-REF vs. RAIDR

We conclude that DC-REF is an effective mechanism to improve system performance by leveraging both PARBOR and the dynamic data content information of memory rows. We believe similar data-content aware optimizations can also be developed on top of DRAM latency reduction mechanisms [17, 18, 27, 43, 69] to achieve further latency reduction benefits.

9. Related Work

This is the first work to propose a feasible, fast and efficient system-level detection mechanism to determine the locations of physically neighboring DRAM cells and use that information to detect all data-dependent cell failures. Prior works have examined data-dependent failures in DRAM [1, 18, 35, 39, 43, 45, 47, 60, 64], NAND flash memory [8, 9, 10, 11, 12, 13, 14, 15, 26, 54], and SRAM [5, 23, 32, 79, 81, 82]. None of these works developed a method for finding neighboring cell locations. Some prior works focus on detecting data-dependent DRAM failures, either in the system or using specialized logic within DRAM with built-in self tests. Both of these types of works, which we discuss next, have limitations that make them either infeasible or impractical in a real system.

System-level detection. Prior works that depend on detecting data-dependent DRAM failures in the system assume that a simple test with *all 0s/1s* data pattern or random patterns can detect all data-dependent failures [6, 35, 46, 48, 59, 80]. These works are unaware of the location of physically neighboring cells and, therefore, cannot detect all data-dependent failures, as we showed in Section 7 with random patterns. Thus, they could face serious reliability issues if deployed in a real system.

Built-in self test. Prior works propose to detect data-dependent failures by implementing *specialized* built-in self test (BIST) logic within DRAM chips. These works either assume that designers are aware of the locations of the physically neighboring cells [19, 77] or use *exhaustive* testing mechanisms to detect failures over the scrambled address space [20, 22]. Due to their high runtime and complexity, these techniques cannot be used for *system-level detection* of neighbor cell locations or data-dependent failures, which is the focus of our work.

10. Conclusion

We introduced PARBOR, an efficient system-level technique that 1) determines the locations of physically neighboring cells in DRAM and 2) uses this information to *uncover* data-dependent failures. PARBOR greatly reduces the test time required to determine physically neighboring cells by exploiting 1) our new observation that some DRAM cells are strongly affected by only one of their neighbors and 2) the regularity and abundant parallelism found in modern DRAM chips. To our knowledge, this is the first work to provide a fast and practical method to detect data-dependent DRAM failures at the system level, in the presence of scrambling of addresses within DRAM.

We experimentally demonstrate the effectiveness of PARBOR using a large number of real DRAM chips. PARBOR greatly reduces the number of tests (by orders of magnitude) while uncovering significantly more data-dependent failures than state-of-the-art testing methods. We demonstrate that PARBOR enables both previously-proposed and new techniques that improve DRAM reliability, performance and energy efficiency. We introduce the notion of Data Content-based REFresh (DC-REF) as one example new technique, and show that it significantly improves system performance. We hope that PARBOR will also inspire the development of a wide range of new system-level mechanisms that take advantage of efficient dynamic detection of data-dependent DRAM failures.

Acknowledgements

We thank Chris Wilkerson, Uksong Kang, anonymous reviewers, and SAFARI group members for feedback. We acknowledge the support of Google, Intel, Nvidia, and Samsung. This research was supported in part by the ISTC-CC and NSF (grants 1212962, 1320531, and 1566483).

References

[1] Z. Al-Ars et al. Effects of bit line coupling on the faulty behavior of DRAMs. VTS, 2004.
[2] Z. Al-Ars et al. Influence of bit line twisting on the faulty behavior of DRAMs. MTDT, 2004.
[3] Z. Al-ars et al. Space of DRAM fault models and corresponding testing. DATE, 2006.
[4] Z. Al-Ars et al. Defect oriented testing of the strap problem under process variations in DRAMs. TEST, 2008.
[5] Z. Al-Ars and S. Hamdioui. Evaluation of SRAM faulty behavior under bit line coupling. IDT, 2008.
[6] A. Bacchini et al. Characterization of data retention faults in DRAM devices. DFT, 2014.
[7] R. Baumann. The impact of technology scaling on soft error rate performance and limits to the efficacy of error correction. IEDM, 2002.
[8] Y. Cai et al. Error patterns in MLC NAND flash memory: Measurement, characterization, and analysis. DATE, 2012.
[9] Y. Cai et al. Error analysis and retention-aware error management for NAND flash memory. *Intel Technology Journal*, 2013.
[10] Y. Cai et al. Program interference in MLC NAND flash memory: Characterization, modeling, and mitigation. ICCD, 2013.
[11] Y. Cai et al. Threshold voltage distribution in MLC NAND flash memory: Characterization, analysis and modeling. DATE, 2013.
[12] Y. Cai et al. Neighbor-cell assisted error correction for MLC NAND flash memories. SIGMETRICS, 2014.
[13] Y. Cai et al. Data retention in MLC NAND flash memory: Characterization, optimization and recovery. HPCA, 2015.
[14] Y. Cai et al. Read disturb errors in MLC NAND flash memory: Characterization, mitigation, and recovery. DSN, 2015.
[15] J. Cha et al. New fault detection algorithm for multi-level cell flash memories. ATS, 2011.
[16] K. Chang et al. Improving DRAM performance by parallelizing refreshes with accesses. HPCA, 2014.
[17] K. Chang et al. Low-cost inter-linked subarrays (LISA): Enabling fast inter-subarray data movement in DRAM. HPCA, 2016.
[18] K. Chang et al. Understanding latency variation in modern DRAM chips: Experimental characterization, analysis, and optimization. SIGMETRICS, 2016.
[19] K.-L. Cheng et al. Neighborhood pattern-sensitive fault testing and diagnostics for random-access memories. *IEEE TCAD*, 2006.
[20] B. Cockburn and Y.-F. Sat. A transparent built-in self-test scheme for detecting single V-coupling faults in RAMs. MTDT, 1994.
[21] T. H. Cormen et al. *Introduction to Algorithms*. 2001.
[22] D. Das and M. Karpovsky. Exhaustive and near-exhaustive memory testing

techniques and their BIST implementations. *Journal of Electronic Testing*, 10(3), 1997.
[23] R. Dekker et al. A realistic fault model and test algorithms for static random access memories. TCAD, 1990.
[24] N. El-Sayed et al. Temperature management in data centers: Why some (might) like it hot. SIGMETRICS, 2012.
[25] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, 2008.
[26] O. Ginez et al. Embedded flash testing: Overview and perspectives. DTIS, 2006.
[27] H. Hassan et al. ChargeCache: Reducing DRAM latency by exploiting row access locality. HPCA, 2016.
[28] M. Horiguchi and K. Itoh. *Repair for Nanoscale Memories*. Springer Publishing Company, 2011.
[29] C.-S. Hou et al. An FPGA-based test platform for analyzing data retention time distribution of DRAMs. VLSI-DAT, 2013.
[30] A. A. Hwang et al. Cosmic rays don't strike twice: understanding the nature of DRAM errors and the implications for system design. ASPLOS, 2012.
[31] JEDEC. *Standard No. 79-3F. DDR3 SDRAM Specification*, July 2012.
[32] R. R. Julie et al. 12N test procedure for NPSF testing and diagnosis for SRAMs. ICSE, 2008.
[33] D.-C. Kang et al. An efficient built-in self test algorithm for neighborhood pattern and bit-line-sensitive faults in high density memories. ETRI, 2004.
[34] U. Kang et al. Co-architecting controllers and DRAM to enhance DRAM process scaling. In *The Memory Forum*, 2014.
[35] S. Khan et al. The efficacy of error mitigation techniques for DRAM retention failures: A comparative experimental study. SIGMETRICS, 2014.
[36] J. Y. Kim et al. Parallely testable design for detection of neighborhood pattern sensitive faults in high density DRAMs. ISCAS, 2005.
[37] K. Kim. Technology for sub-50nm DRAM and NAND flash manufacturing. IEDM, 2005.
[38] Y. Kim et al. A case for subarray-level parallelism (SALP) in DRAM. In *ISCA*, 2012.
[39] Y. Kim et al. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. ISCA, 2014.
[40] Y. Kim, W. Yang, and O. Mutlu. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE CAL*, 2015.
[41] T. Kirihata, Watanabe, et al. Fault-tolerant designs for 256 Mb DRAM. *JSSC*, 1996.
[42] D. Lee et al. Tiered-latency DRAM: A low latency and low cost DRAM architecture. HPCA, 2013.
[43] D. Lee et al. Adaptive-latency DRAM: Optimizing DRAM timing for the common-case. HPCA, 2015.
[44] D. Lee et al. Simultaneous multi-layer access: Improving 3D-stacked memory bandwidth at low cost. TACO, 2016.
[45] M. J. Lee and K. W. Park. A mechanism for dependence of refresh time on data pattern in DRAM. *Electron Device Letters*, 31(2), 2010.
[46] J. Liu et al. RAIDR: Retention-aware intelligent DRAM refresh. ISCA, 2012.
[47] J. Liu et al. An experimental study of data retention behavior in modern DRAM devices: Implications for retention time profiling mechanisms. ISCA, 2013.
[48] S. Liu et al. Flikker: Saving DRAM refresh-power through critical data partitioning. ASPLOS, 2011.
[49] S. Liu et al. Hardware/software techniques for DRAM thermal management. HPCA, 2011.
[50] C.-K. Luk et al. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.
[51] Y. Luo et al. Characterizing application memory error vulnerability to optimize datacenter cost via heterogeneous-reliability memory. DSN, 2014.
[52] 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.
[53] J. Meza et al. Revisiting memory errors in large-scale production data centers: Analysis and modeling of new trends from the field. DSN, 2015.
[54] M. G. Mohammad and K. K. Saluja. Flash memory disturbances: Modeling and test. VTS, 2001.
[55] Y. Mori et al. The origin of variable retention time in DRAM. IEDM, 2005.
[56] W. Mueller et al. Challenges for the DRAM cell scaling to 40nm. IEDM, 2005.
[57] O. Mutlu. Memory scaling: A systems architecture perspective. *IMW*, 2013.
[58] O. Mutlu and L. Subramanian. Research problems and opportunities in memory systems. *SUPERFRI*, 2014.
[59] P. J. Nair et al. ArchShield: Architectural framework for assisting DRAM scaling by tolerating high error rates. ISCA, 2013.
[60] Y. Nakagome et al. The impact of data-line interference noise on DRAM scaling. *JSSC*, 1988.
[61] PARBOR Source Code. https://github.com/CMU-SAFARI/PARBOR/.
[62] M. Qureshi et al. AVATAR: A Variable-Retention-Time (VRT) Aware Refresh for DRAM Systems. DSN, 2015.
[63] Ramulator Source Code. https://github.com/CMU-SAFARI/ramulator/.
[64] M. Redeker, B. F. Cockburn, and D. G. Elliott. An investigation into crosstalk noise in DRAM structures. MTDT, 2002.
[65] P. J. Restle, J. W. Park, and B. F. Lloyd. DRAM variable retention time. IEDM, 1992.
[66] T. Schloesser et al. $6F^2$ buried wordline DRAM cell for 40nm and beyond. IEDM, 2008.
[67] B. Schroeder et al. DRAM errors in the wild: A large-scale field study. SIGMETRICS, 2009.
[68] T. Sekiguchi et al. A low-impedance open-bitline array for multigigabit DRAM. *JSSC*, 2002.
[69] V. Seshadri et al. RowClone: Fast and efficient In-DRAM copy and initialization of bulk data. MICRO, 2013.
[70] Y. Sfikas et al. Layout-based refined NPSF model for DRAM characterization and testing. VLSI, 2014.
[71] Y. Sfikas and Y. Tsiatouhas. Physical design oriented DRAM neighborhood pattern sensitive fault testing. DDECS, 2009.
[72] A. Snavely and D. Tullsen. Symbiotic job scheduling for a simultaneous multithreading processor. ASPLOS, 2000.
[73] SPEC CPU2006. Standard Performance Evaluation Corporation. http://www.spec.org/cpu2006.
[74] V. Sridharan et al. Memory errors in modern systems: The good, the bad, and the ugly. ASPLOS, 2015.
[75] V. Sridharan and D. Liberty. A study of DRAM failures in the field. SC, 2012.
[76] G. R. Srinivasan et al. Accurate, predictive modeling of soft error rate due to cosmic rays and chip alpha radiation. IRPS, 1994.
[77] D. S. Suk and S. M. Reddy. Test procedures for a class of pattern-sensitive faults in semiconductor random-access memories. *IEEE TC*, 1980.
[78] A. J. van de Goor and I. Schanstra. Address and data scrambling: Causes and impact on memory tests. DELTA, 2002.
[79] A. J. van de Goor and I. B. S. Tlili. Disturb neighborhood pattern sensitive fault. VTEST, 1997.
[80] R. K. Venkatesan et al. Retention-aware placement in DRAM (RAPID): Software methods for quasi-non-volatile DRAM. HPCA, 2006.
[81] B. Wang et al. Reducing test time of embedded SRAMs. MTDT, 2003.
[82] X. Wang et al. Testing of inter-word coupling faults in word-oriented SRAMs. DFTVS, 2004.
[83] C. Weis et al. Retention time measurements and modeling of bit error rates of WIDE I/O DRAM in MPSoCs. DATE, 2015.
[84] Xilinx. *ML605 Hardware User Guide*, Oct. 2012.
[85] D. Yaney et al. A meta-stable leakage phenomenon in DRAM charge storage - Variable hold time. IEDM, 1987.

Appendix

Time for Determining Neighboring Locations with Exhaustive Testing. For each bit address in a row, we need to write the worst-case pattern in the victim and the test cell, wait for 64 ms, and read them back to detect the failures. In order to read/write these two cells, we need to read data from the cells in the row buffer (t_{RCD}), transfer the blocks containing the corresponding bits to the memory controller, and close the row (t_{RP}). According to DDR3-1600 timing, time to read/write two cache blocks is: $t_r = t_{RCD} + t_{CCD} + t_{RP} = 13.75 + 5 * 2 + 13.75 = 42.5$ ns. Therefore, testing one address bit in a row takes 42.5 ns + 64 ms + 42.5 ns $\approx$ 64 ms. Testing all cells in an 8K-cell row ($O(n)$ test) requires $64 * 8192$ ms $= 8.73$ minutes, n^2 tests require $64 * 8192^2$ ms $= 49$ days, n^3 tests require $64 * 8192^3$ ms $= 1115$ years, n^4 tests require $64 * 8192^4$ ms $= 9.1$ M years.

Computational Complexity of the Recursive Divide and Conquer Algorithm. Recurrence relations in divide and conquer algorithms have the following form: $T(n) = aT(n/b) + f(n)$, where the problem of size n is divided into b subproblems at each level, and only a of those subproblems are solved in the next level. $f(n)$ corresponds to the additional work done outside the recursion. The solution for this recursion is: $T(n) = \theta(n \log_b a)$ when $f(n)$=$O(1)$ [21]. In PARBOR, each region is divided into 8 smaller regions at each level. The execution time becomes $T(n) = 8T(n/8) + O(1)$. Solving this recurrence, $T(n) = \theta(n \log_8 8) = \theta(n)$.

Time for Detecting Data-dependent Failures in an Entire Module with PARBOR. To test for data-dependent failures in an entire module, we write a neighbor-aware data pattern in the entire module, wait for 64 ms, and read the module to detect failures. In order to read/write a row, we read data from the cells in the row buffer (t_{RCD}), transfer the row to the memory controller, and close the row (t_{RP}). According to DDR3-1600 timing, time to read/write an $8KB$ row is: $t_r = t_{RCD} + t_{CCD} * (8KB/64B) + t_{RP} = 13.75 + 5 * 128 + 13.75 = 667.5$ ns. In a 2GB module, there are 262144 rows, so reading/writing the entire module would take $t_r * 262144$ ns = 174.98 ms. So, to test a module once, it would take $174.98 + 64 + 174.98$ ms = 413.96 ms. 92 tests with PARBOR take 92*413.96 ms = 32 seconds and 132 tests take 132*413.96 ms = 55 seconds.