RDIS: A Recursively Defined Invertible Set Scheme to Tolerate Multiple Stuck-At Faults in Resistive Memory

Rami Melhem, Rakan Maddah and Sangyeun Cho
Computer Science Department
University of Pittsburgh
Pittsburgh, PA 15260 USA
{melhem,rmaddah,cho}@cs.pitt.edu

***Abstract*—With their potential for high scalability and density, resistive memories are foreseen as a promising technology that overcomes the physical limitations confronted by charge-based DRAM and flash memory. Yet, a main burden towards the successful adoption and commercialization of resistive memories is their low cell reliability caused by process variation and limited write endurance. Typically, faulty and worn-out cells are permanently stuck at either '0' or '1'. To overcome the challenge, a robust error correction scheme that can recover from many hard faults is required.**

In this paper, we propose and evaluate *RDIS*, a novel scheme to efficiently tolerate memory stuck-at faults. RDIS allows for the correct retrieval of data by recursively determining and efficiently keeping track of the positions of the bits that are stuck at a value different from the ones that are written, and then, at read time, by inverting the values read from those positions. RDIS is characterized by a very low probability of failure that increases slowly with the relative increase in the number of faults. Moreover, RDIS tolerates many more faults than the best existing scheme—by up to 95% on average at the same overhead level.

***Keywords*-Error Correction Code; Hard Faults; Phase Change Memory; Fault Tolerance; Reliability;**

I. Introduction

Resistive memories are receiving due attention as the scaling of DRAM and flash memory is hindered by physical limitations [1]–[3]. For example, the use of phase-change memory (PCM), spin-transfer torque memory (STT-RAM), and memristor in a platform's memory and storage hierarchy has been explored recently [4]–[12]. Among many resistive memory types, PCM has attracted significant preference in the research community because it is believed to be the closest to mass production; Micron and Samsung are producing working samples of 128 Mbit to 8 Gbit capacity as of 2012 [13]–[15]. Early evaluations (e.g., [6]–[8]) demonstrate that PCM can compete favorably with DRAM (main memory) in terms of performance and beat DRAM in terms of power consumption.

However, a major weakness of resistive memories (especially PCM and memristor), impeding their fast commercialization, is the low cell-level reliability [16]–[18]. There are two major factors in this matter: imperfect process control with a very deep sub-micron technology and repeated writes to a cell (i.e., write cycling). Prior architecture and systems research focus on the latter because manufacturers will ship chips with a minimum guaranteed write endurance. For example, the reported write endurance of PCM and memristor is 10^6 to 10^8 [13], [19]–[21]. A few failure mechanisms exist [22]–[24], and once activated, they interfere with write operations. Hence, weak cells and worn-out cells are typically "stuck-at" a particular value, either '0' or '1' [17], [23].

To address the write endurance problem, it is believed that both aggressive wear leveling and proactive error masking techniques are necessary. Wear leveling spreads writes to the entire memory capacity to evenly wear memory cells (e.g., through periodic, pseudo-randomization of write addresses) [7], [8], [25], [26]. Techniques to suppress unnecessary bit-level writes were proposed [6], [8], [9], [27]. However, due to process variation, memory cells are expected to wear out at different rates, which compromises the chip's lifetime. Accordingly, error masking techniques are required to overcome cell failures.

Error correction code (ECC) such as SEC-DED (single error correction, double error detection) has been successfully used to protect main memory. However, traditional hamming code based ECC is designed for a general fault model and its overhead is unnecessarily large for the stuck-at fault model. This is especially true when the probability of having multiple bit errors is high, as is the case with resistive memories. For example, imagine that many cells in a memory block have reached their write endurance limit simultaneously. To cope with many faults, we must employ a correspondingly stronger ECC, which would incur excessively large space and computation overheads. In fact, for NAND flash memory, also suffering write endurance limitation, it is required to correct 40 or more bits per 512-byte block [28]. Subsequently, recently proposed error masking techniques for resistive memories [16]–[18] combine clever microarchitectural and coding ideas to cut down overheads.

In this paper, we propose *RDIS* (recursively defined invertible set), a novel low-overhead error correction scheme

*This work is supported in part by NSF grants CCF-1064976, CCF-1059283 and CNS-1012070.

to recover from hard errors.[1] RDIS allows for the correct retrieval of data in the presence of stuck-at faults by keeping track of the bits that are stuck at a value different from the ones that are written, and then, at read time, by inverting the values read for those bits. For a write operation, each cell in a data block is either: "non-faulty" (NF), stuck at the opposite of the value being written ("stuck-at-wrong" or SA-W), or stuck at the same value written ("stuck-at-right" or SA-R). For example, trying to write '0' in a cell stuck at '1' makes the cell SA-W. The underlying idea of RDIS is to identify and encode a subset S—out of all cells forming a data block to be updated—containing all the SA-W cells. Later, the members of S are read inverted, which retrieves the data as it was intended to be written originally. RDIS initiates the computation of S after detecting write failure through applying a read-after-write verification operation.

Although it can only guarantee the recovery from three faults, RDIS has a desirable property of effectively recovering from many more faults beyond what it guarantees. Intrinsically, RDIS enjoys a low probability of failure that increases at a very slow rate with the relative increase in the number of fault occurrences. By comparison, current state-of-the-art schemes either cannot recover from a single fault beyond a guaranteed number of faults (e.g., ECC [29] and ECP [16]) or can recover additional faults but with a low probability (e.g., SAFER [17]). Our evaluation shows that RDIS can tolerate 95% more faults on average than SAFER when the protected block size is 1 KB. Given its ability to recover many faults with high probability, RDIS is a very good fit for resistive memories that will experience a growing number of faults over the course of use.

We formally prove the fault tolerance properties of RDIS and by exploring a potential hardware implementation, we find that the required additional logic is surprisingly simple. It is worth mentioning that RDIS error correction capabilities are not limited to main memory. RDIS is capable of tolerating faults significantly within block sizes ranging from cache line size to secondary storage block sector size, while incurring a low overhead. Accordingly, we present a study of RDIS error correction capability at different block sizes.

The remainder of this paper is organized as follows. Section II will first summarize the related work. Section III and Section IV will then give the details of the proposed RDIS scheme by formally describing the concepts and the coverage of the scheme. We also discuss hardware implementation implications. Experimental evaluation of RDIS will be presented in Section V, and finally, Section VI will conclude the paper.

II. Prior Related Work

The exploration of ECC can be traced many years back [29]. Among many ECC schemes, SEC-DED is widely used to protect DRAM in main memory. Since DRAM errors are typically transient and occur infrequently, SEC-DED is adequate in most situations. On the other hand, resistive memories have different failure mechanisms and are subject to multiple bit faults that occur gradually with the lifetime of a chip. Consequently, it is necessary to deploy a multi-bit error correction scheme. Hamming code based BCH code [30] is one such scheme. Yet, codes based on BCH are complex and expensive to implement [28], [31]. As a matter of fact, the complexity increases linearly with the number of faults to be tolerated [31].

There are three recent proposals that target specifically masking errors in resistive memories with higher auxiliary storage efficiency than traditional ECC techniques. First, Error Correcting Pointer (ECP) [16] provides a limited number of programmable "correction entries". A correction entry holds a pointer (address) to a faulty cell within the protected block and a "patch" cell that replaces the faulty one. When a faulty cell is detected, a new correction entry is allocated to cover the cell. A memory block is de-commissioned when the number of faulty cells exceeds that of the correction entries. In essence, ECP provides cell-level spares to each block.

SAFER (Stuck-at-Fault Error Recovery) [17] dynamically partitions a protected data block into a number of groups so that each group contains at most one faulty cell. When the value of the faulty cell is different from the intended value to be written, all cells in the the group are written and read inverted. If the data block is to be partitioned into n groups, then SAFER allows $\log_2 n$ "repartitions". Repartitioning is done whenever a new fault is detected. Therefore, SAFER guarantees the recovery from $\log_2 n + 1$ faults. Any additional fault is tolerated only if it occurs in a fault-free group. Otherwise, the block has to be retired. SAFER was shown to provide stronger error correction than ECC or ECP at the same overhead level.

Free-p (Fine-grained Remapping with ECC and Embedded-Pointers) [18] combines error correction and redundancy, and as such, has two protection layers. First, it uses an ECC to mask faults within a data block. Second, when a block becomes defective, Free-p embeds a pointer within the defective block so that a redundant, non-faulty block can be quickly identified without having to access a separate remapping table. Free-p employs ECC to correct up to four hard errors in a data block of cache line size and relies on the OS to perform block remapping. We note that the block remapping idea of Free-p is orthogonal to RDIS. Hence, RDIS could be used to replace ECC in Free-p.

III. RDIS

This section describe RDIS intuitively using Set Theory. We begin with the idea of invertible sets and how to specify an invertible set given a set of faulty memory cells in a block. We then focus on an algorithm to compute necessary auxiliary information to correctly store and retrieve user information. Finally, we discuss a hardware embodiment of RDIS before we close this section.

A. Basic idea

RDIS applies to a block of memory/storage cells. Let's assume that the block has N cells, $c(0)$, ..., $c(N-1)$, and they store binary information $b(0)$, ..., $b(N-1)$. Each cell $c(i)$ is either non-faulty (NF), stuck at '0' (SA-0), or stuck at '1' (SA-1). Furthermore, RDIS uses a different classification of the faulty

[1]The principles of RDIS are not limited to resistive memories. RDIS is particularly relevant for resistive memories because it can correct many errors with high probability.

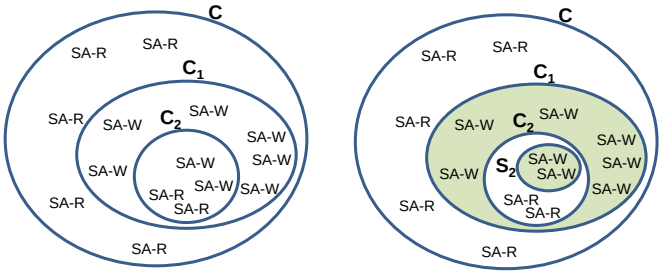


Fig. 1: The invertible set $S = (C_1 - C_2) \cup S_2$.

cells, depending on the value that is to be written in those cells. Specifically, when bit $b(i)$ is to be stored in a faulty cell $c(i)$, then $c(i)$ is stuck at the right value (SA-R) if it is SA-0 and $b(i) = 0$ or it is SA-1 and $b(i) = 1$. Similarly, $c(i)$ is stuck at the wrong value (SA-W) if it is SA-0 and $b(i) = 1$ or it is SA-1 and $b(i) = 0$. Using this classification, each cell $c(i)$ can be in one of three classes: NF, SA-R (when the information to store in the faulty cell is identical to the stuck value), or SA-W (when the information to store in the faulty cell is different from the stuck value).

H-bit auxiliary information is used to allow the correct retrieval of the N stored bits. The value of H will be specified later. For clarity of discussion, we assume that the auxiliary information is maintained in a separate fault-free storage. Alternatively, the auxiliary information can be stored in the same faulty medium as the data but adequately protected by some other technique (see Section V-C for further discussions).

Denoting the memory cells $c(0)$, ..., $c(N-1)$ by C, the main idea of RDIS is to use the auxiliary H bits to identify a subset $S \subset C$ such that every SA-W cell is in S and every SA-R cell is in $C - S$. In other words, S contains all the SA-W cells of C and none of its SA-R cells. We call S an "invertible" subset of C. When the N bits of information are stored, any cell $c(i)$ in $C - S$ will store $b(i)$ intact, while any cell in S will store the complement of $b(i)$. Subsequently, when the information is read, the content of any cell in S is complemented, thus allowing the correct retrieval of all N bits.

A simple way of expressing S is to keep a list of pointers to the SA-W cells. This requires $\log_2 N$ bits of auxiliary information for each cell and hence, to tolerate a maximum of F faults, $H = F \times \log_2 N$ bits of auxiliary information is needed. RDIS introduces a different, yet systematic method for constructing and representing S by allowing it to include NF (not faulty) cells in addition to SA-W cells. Clearly, if a cell $c(i)$ is not faulty, then it is possible to store (and correctly retrieve) the complement of $b(i)$ in $c(i)$. Conceptually, the set S is constructed by computing a sequence of subsets $C_2 \subset C_1 \subset C$ such that:

- All the SA-W cells that are in C, and possibly some SA-R cells, are included in C_1;
- All the SA-R cells that are in C_1, and possibly some SA-W cells, are included in C_2; and
- With a very large probability, the size of C_2 is much smaller than the size of C.

Figure 1 illustrates the idea of the construction of C_1 and C_2. Note that any of C, C_1, and C_2 can contain NF cells as well. However, by definition, $C_1 - C_2$ does not contain any SA-R cells. Clearly, if C_1 does not contain any SA-R cells, then the construction of C_2 is not needed since we can set $S = C_1$.

We consider two cases. First, if C_2 does not contain any SA-W cells, then the invertible set S that we are looking for is $S = C_1 - C_2$ since we are sure that $C_1 - C_2$ contains all the SA-W cells of C and none of its SA-R cells. The second case occurs if C_2 contains some SA-W cells. In this case, we recursively apply the same process to find an invertible set S_2 of C_2 which includes all its SA-W cells and none of its SA-R cells. Therefore, $S = C_1 - (C_2 - S_2) = (C_1 - C_2) \cup S_2$. Figure 1 shows the invertible set S of C as a shaded area.

B. Specifying an invertible subset

One way to identify S, is to arrange the N bits/cells into a logical two-dimensional array of n rows and m columns,[2] and accordingly, re-label the information bits as $b(i, j)$ and the storage cells as $c(i, j)$, where $i = 0, \ldots, n-1$ and $j = 0, \ldots, m-1$. In this section, we will use the example of the 8×8 array shown in Figure 2(a) to illustrate the process of specifying the invertible set. As depicted, C contains 7 SA-W and 7 SA-R faults.

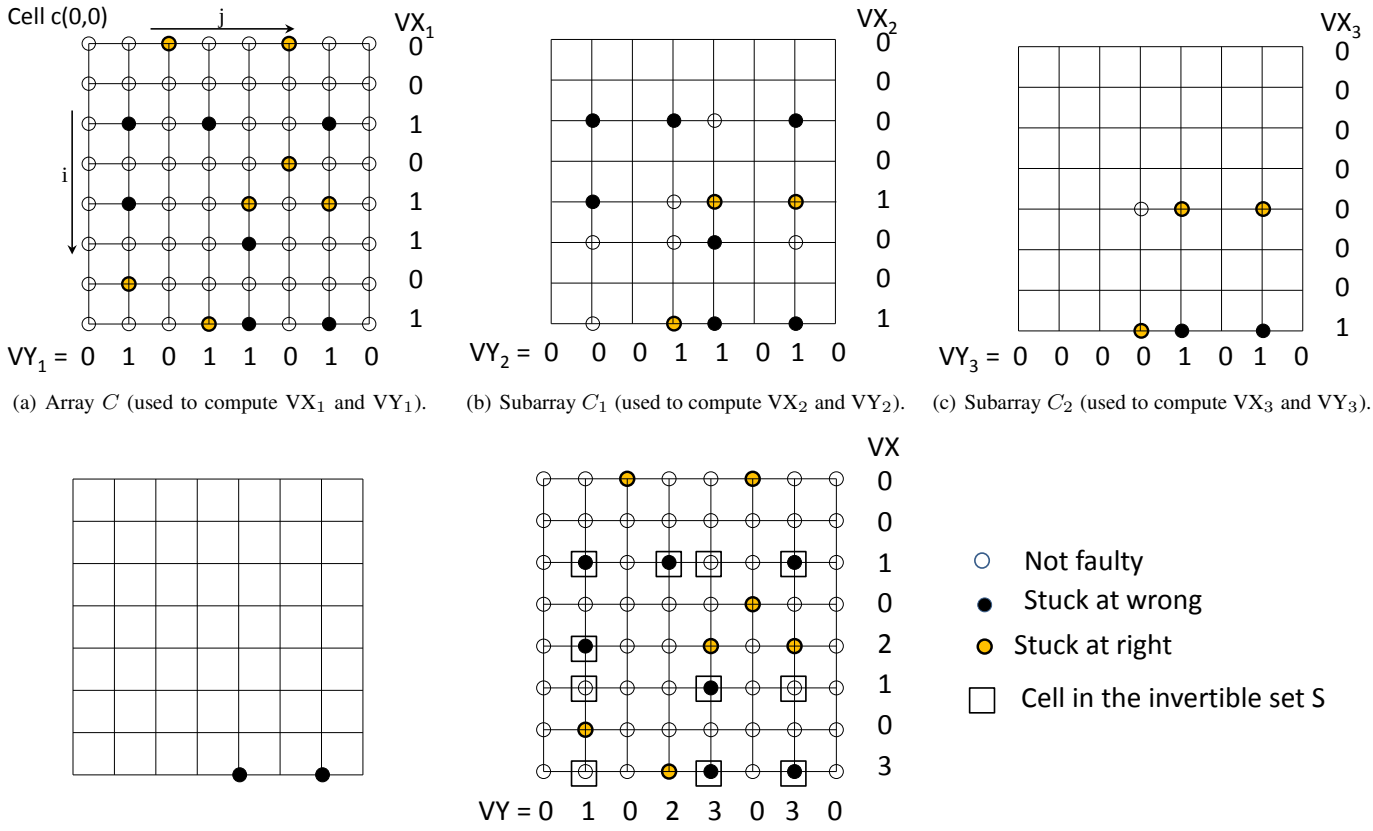
RDIS maintains $n + m$ auxiliary binary flags $\text{VX}_1(i)$, $i = 0, \ldots, n-1$ and $\text{VY}_1(j)$, $j = 0, \ldots, m-1$. These flags are set such that:

- $\text{VX}_1(i) = 1$ if row i of C contains at least one SA-W cell (otherwise $\text{VX}_1(i) = 0$); and
- $\text{VY}_1(j) = 1$ if column j of C contains at least one SA-W cell (otherwise $\text{VY}_1(j) = 0$).

Let n_1 be the number of rows in the $n \times m$ array C that have $\text{VX}_1 = 1$ and let m_1 be the number of columns of C that have $\text{VY}_1 = 1$. Moreover, define C_1 as the subset of cells $\{c(i, j) | (\text{VX}_1(i) = 1) \text{ and } (\text{VY}_1(j) = 1)\}$. In other words, C_1 is the $n_1 \times m_1$ subarray of C that contains: (1) SA-W cells and (2) cells that lie at the intersection of a row that contains a SA-W cell and a column that contains a SA-W cell (these can be either NF or SA-R). In our example, the values of VX_1 and VY_1 are shown in Figure 2(a). The SA-W cells of C are confined to rows 2, 4, 5, 7 and columns 1, 3, 4, 6, and hence, these rows and columns form the subarray C_1 shown in Figure 2(b).

Since C_1 is defined to include all the SA-W cells of C, any cell that is in $C - C_1$ is either NF or SA-R, and thus can hold the correct value of the corresponding information bit. However, the cells that are in C_1 may be NF, SA-W, or SA-R. If C_1 does not contain any SA-R cell (i.e., C_1 contains only NF or SA-W cells), then $S = C_1$. If, however, C_1 contains some SA-R cells (as is the case in Figure 2(b)), then, we need to find a subset, S_1 of C_1, which includes all its SA-R cells and none of its SA-W cells. This will allow us to specify an invertible subset of C as $S = C_1 - S_1$. To obtain S_1, we apply the same procedure used to extract C_1 from C, but after reversing the roles of SA-R and SA-W. Specifically, we define the binary flags

[2]Introducing more than two dimensions is certainly possible, but is beyond the scope of this paper.



(a) Array C (used to compute VX_1 and VY_1).

(b) Subarray C_1 (used to compute VX_2 and VY_2).

(c) Subarray C_2 (used to compute VX_3 and VY_3).

(d) Subarray C_3 (all entries in VX_4 and VY_4 are zeroes).

(e) Invertible subset $S = (C_1 - C_2) \cup C_3$.

Fig. 2: An example for constructing the invertible set.

- $\text{VX}_2(i) = 1$ if row i of C_1 contains at least one SA-R cell (otherwise $\text{VX}_2(i) = 0$); and
- $\text{VY}_2(j) = 1$ if column j of C_1 contains at least one SA-R cell (otherwise $\text{VY}_2(j) = 0$).

Let n_2 be the number of row of C_1 that have $\text{VX}_2 = 1$ and let m_2 be the number of columns of C_1 that have $\text{VY}_2 = 1$. Moreover, define C_2 as the subset of cells $\{c(i,j) | (\text{VX}_2(i) = 1)$ and $(\text{VY}_2(j) = 1)\}$. In other words, C_2 is the $n_2 \times m_2$ subarray of C_1 that contains: (1) SA-R cells and (2) cells that lie at the intersection of a row that contains a SA-R cell and a column that contains a SA-R cell. In the example of Figure 2, we form subarray C_2 to include all the SA-R cells that are in C_1. In Figure 2(c), C_2 is composed of rows 4, 7 and columns 3, 4, 6. By construction any cell that is in $C_1 - C_2$ is either NF or SA-W. Moreover, if C_2 does not contain any SA-W cell, then $S_1 = C_2$ and we can form the invertible set $S = C_1 - C_2$.

Unfortunately, we cannot guarantee that C_2 does not contain any SA-W cell. Fortunately, however, if $C_2 \neq C_1$ (i.e., C_2 is a proper subset of C_1), then we can apply the same procedure used to extract C_1 from C to compute the subset S_2 of C_2 that contains all its SA-W cells and then set $S = (C_1 - C_2) \cup S_2$. The iterative process can continue to compute consecutive subarrays $C_3, \ldots, C_k$. After k iterations, we can have one of three cases:

1. k is odd and C_k contains only SA-W cells. In this case, the invertible set S is defined as

$$S = (C_1 - C_2) \cup (C_3 - C_4) \cup \ldots \cup (C_{k-2} - C_{k-1}) \cup C_k.$$

2. k is even and C_k contains only SA-R cells. In this case the invertible set S is defined as

$$S = (C_1 - C_2) \cup (C_3 - C_4) \cup \ldots \cup (C_{k-1} - C_k).$$

3. The progress stalls because $C_k = C_{k-1}$, in which case the set of faults cannot be masked.

Back to the example of Figure 2, the array C_2 shown in Figure 2(c) includes all the SA-R cells that are in C_1 but also contains two SA-W cells. Hence, we form subarray C_3 to include all the SA-W cells that are in C_2 (see Figure 2(d)). The process terminates with $k = 3$ because C_3 does not include any SA-R cells, and thus, $S = (C_1 - C_2) \cup C_3$ contains all the SA-W cells that are in C and none of its SA-R cells (see Figure 2(e)).

C. Overhead of auxiliary information

The subarrays C_1, C_2, ... are completely specified by the binary flags $\text{VX}_1(i), \text{VX}_2(i), \ldots, i = 0, \ldots, n-1$ and $\text{VY}_1(j), \text{VY}_2(j), \ldots, j = 0, \ldots, m-1$. In other words, these flags form the auxiliary information that has to be maintained to retrieve the correct values stored in the N cells. Note that if $\text{VX}_u(i) = 0$ for some u, then $\text{VX}_v(i) = 0$ for any $v > u$. Similarly, if $\text{VY}_w(i) = 0$ for some w, then $\text{VY}_v(i) = 0$ for any $v > w$.

Hence, the flags can be compressed into two sets of counters (see Figure 2(e)): $\mathrm{VX}(i) = \sum_{k=1}^{u} \mathrm{VX}_k(i)$ for $i = 0, \ldots, n-1$; and $\mathrm{VY}(j) = \sum_{k=1}^{w} \mathrm{VY}_k(j)$ for $j = 0, \ldots, m-1$.

The auxiliary information needed to reconstruct S, thus, consists of the $(n+m)$ counters $\mathrm{VX}(i)$ and $\mathrm{VY}(j)$. If each of these counters can count up to K, then the number of bits, H, needed to keep the auxiliary information is $H = (n+m) \times \lceil \log_2(K+1) \rceil$. Note that by limiting the maximum value of each counter to K, we assume that the recursive construction of S will terminate in K steps. If that is not the case, then the process will fail and the given faults cannot be tolerated.

D. Storing and retrieving information

In order to store and retrieve user data, VX and VY must be computed first. Let us present an algorithm to do that assuming that the locations and nature of faults are known. This information can be kept in a separate storage (e.g., SRAM cache) or discovered on line by a write-read-check process (as described later). Given the fault information and the data that is to be written, we can associate with each cell, $c(i,j)$, a state that is represented by two bits $\phi(i,j)$ and $\sigma(i,j)$ as follows:

- $\phi(i,j) = 1$ and $\sigma(i,j) = 0 \rightarrow$ cell $c(i,j)$ is SA-R.
- $\phi(i,j) = 1$ and $\sigma(i,j) = 1 \rightarrow$ cell $c(i,j)$ is SA-W.
- $\phi(i,j) = 0$ and $\sigma(i,j) = 0 \rightarrow$ cell $c(i,j)$ is NF or the fault was successfully handled.

To compute the values of the counters VX_i for $i = 0, \ldots, n-1$ and VY_j $j = 0, \ldots, m-1$;, Algorithm 1 is applied. In each iteration, k, of the algorithm (line 2), the subarray which contains SA-W cells is formed (by computing the flags VX_k and VY_k - lines 3 to 10). Then, the state of every cell that is not in this subarray is set to (ϕ = 0 and σ = 0) since it is either NF or is SA-R (lines 16 and 17). In preparation for the next iteration, the algorithm then changes the states of every faulty cell in the identified subarray such that SA-W cells become SA-R and SA-R cells become SA-W (lines 18 and 19). The algorithm assumes that the counters $\mathrm{VX}(i)$ and $\mathrm{VY}(j)$are initially set to zero.

The way the counters $\mathrm{VX}(i)$ and $\mathrm{VY}(j)$ are computed implies that if cell $c(i,j)$ is in C_k and not in C_{k+1} then at least one of the two counters $\mathrm{VX}(i)$ or $VY(j)$ is equal to k while the other one is larger than or equal to k. Given this observation, Algorithm 2 can be used to store the data bits.

Similarly, when retrieving the data, the bit read from cell $c(i,j)$ is complemented if the minimum of $\mathrm{VX}(i)$ and $\mathrm{VY}(j)$ is an odd number. Building a hardware circuit to perform this operation is straightforward, especially when the maximum value of VX and VY is small. The next subsection will present an embodiment of RDIS in hardware.

E. Realizing RDIS in hardware

The block diagram of Figure 3(a) depicts the overall system implementation in hardware. A conventional memory chip would include the main storage, data buffer, and write/read hardware. RDIS adds new components to compute and store auxiliary information (VX and VY) based on fault information. It also modifies the write/read hardware. The logic-level hardware implementation of computing auxiliary information (Algorithm 1) is shown in Figure 3(b) and the modified write path (Algorithm 2) in Figure 3(c). While it is not our goal to present a fully optimized hardware design in this section, we find our intuitive hardware implementation surprisingly simple.

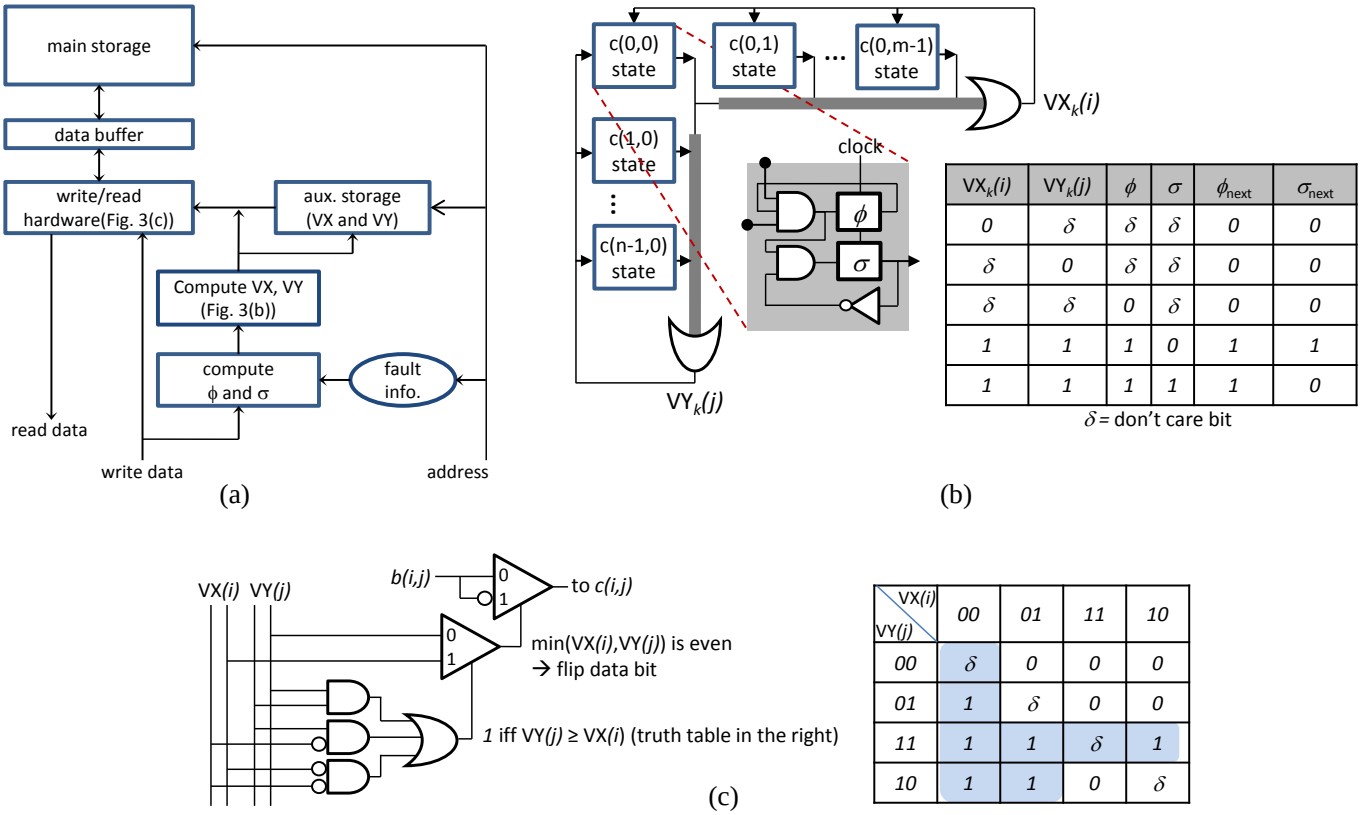
Algorithm 1: Computing VX and VY.

```
1  begin
2      for k ← 1 to K do
3          for i ← 0 to n − 1 do
4              VX_k(i) ← σ(i,0) + ... + σ(i,m−1); // Boolean OR
5              if VX_k(i) = 1 then
6                  VX(i) ← VX(i) + 1;
7          for j ← 0 to m − 1 do
8              VY_k(j) ← σ(0,j) + ... + σ(n−1,j); // Boolean OR
9              if VY_k(j) = 1 then
10                 VY(j) ← VY(j) + 1;
11         if ∀i,j VX_k(i) = 0 and VY_k(j) = 0 then
12             EXIT; // successful completion
13         /* prepare for next iteration */
14         for i ← 0 to n − 1 do
15             for j ← 0 to m − 1 do
16                 if VX_k(i) = 0 or VY_k(j) = 0 then
17                     set φ(i,j) ← 0; σ(i,j) ← 0;
18                 else if φ(i,j) = 1 then
19                     set σ(i,j) ← complement(σ(i,j)); // Bit complement
20     if ∃i,j VX_k(i) > 0 or VY_k(j) > 0 then
21         FAIL; // Given faults can't be masked
```

Algorithm 2: Storing data bits.

```
1  begin
2      for i ← 0 to n − 1 do
3          for j ← 0 to m − 1 do
4              if min(VX(i), VY(j)) is even then
5                  Store b(i,j) in c(i,j);
6              else
7                  Store complement(b(i,j)) in c(i,j);
```

The design in Figure 3(b) spends K cycles to compute VX and VY and maintains two single-bit registers ϕ and σ. These registers are arranged (logically) into a two-dimensional array that mimics the array of storage cells. In each cycle a global OR operation in each row i computes $\mathrm{VX}_k(i)$ and a global OR operation in each column j computes $\mathrm{VY}_k(j)$. The value of $\mathrm{VX}_k(i)$ is then distributed to each cell in row i and the value of $\mathrm{VY}_k(j)$ is distributed to each cell in column j. A local circuit (also illustrated with a truth table) then updates the values of the registers ϕ and σ. To compute $\mathrm{VX}(i)$ and $\mathrm{VY}(j)$, a counter is added to each row, i, and each column j (not shown in the figure). The signal $\mathrm{VX}_k(j)$ is used to increment the counter $\mathrm{VX}(i)$ and the signal $\mathrm{VY}_k(j)$ is used to increment the counter $\mathrm{VY}(j)$. Finally, the logic design (also in a truth table) of Figure 3(c) uses the $\mathrm{VX}(i)$ and $\mathrm{VY}(j)$ counter values to determine if a particular user data bit $b(i,j)$ has to be inverted or not before it is sent to $c(i,j)$.



$VX_k(i)$	$VY_k(j)$	ϕ	σ	ϕ_{next}	σ_{next}
0	δ	δ	δ	0	0
δ	0	δ	δ	0	0
δ	δ	0	δ	0	0
1	1	1	0	1	1
1	1	1	1	1	0

VX(i) / VY(j)	00	01	11	10
00	δ	0	0	0
01	1	δ	0	0
11	1	1	δ	1
10	1	1	0	δ

Fig. 3: (a) A block diagram showing a complete system for writing/reading. (b) Logic to compute auxiliary data (for a cell) and its truth table. (c) Logic to determine write data bit (when $K = 3$) and its truth table.

The proposed hardware implementation infers that the major complexity lies on the write path as RDIS needs to compute the invertible set. The read path is augmented with a simple decoding logic. A recent PCM prototype [20] has a relatively sparse pipeline stages that can easily incorporate the required logic. Write data are typically buffered (e.g., the LPDDR2-NVM interface used in [14]) before being written to the memory cells in an iterative manner. Hence, the computation of the invertible set is done while the data is buffered and is off the critical write path.

As described above, RDIS depends on the knowledge of the fault information (location and stuck-at value). While a read-after-write operation discovers all SA-W cells, it cannot distinguish between the NF and SA-R cells. The latter information can be obtained by testing storage cells on the intersection of a row and column both containing a SA-W cell. Specifically, to test a cell $c(i, j)$, we first read the value, v, stored in that cell, write the complement of v into the cell and read it again. If the value read is not the complement of v, then the cell is SA-R. Otherwise, the cell is NF. One way of avoiding the overhead of error detection before each write operation is to keep a cache which contains information about the faults. Such a cache was proposed in [17] where it was shown that a 128K-entry cache is enough to capture most of the fault information in an 8 Gbit memory. The same cache design can be used in RDIS.

F. Putting it all together

Let us close this section by summing up the overall flow of execution that RDIS follows to detect and mask faults. After writing a block of data, a read operation is performed to verify if data was written correctly. The read verification operation discovers all SA-W cells, if any. Subsequently, SA-R cells are discovered as pointed out in Section III-E. Once the fault information is collected, the computation of the invertible set is executed (refer to Algorithm 1, Algorithm 2 and Figure 3(b)). We note that if the read verification does not discover faults, then data was written correctly and no further action is required. By comparison, ECC always computes the auxiliary information regardless of whether faults occurred or not.

As pointed out in Section III-E, the overhead to obtain the fault information can be eliminated through caching of the fault information (in the "fault information" box of Figure 3(a)). However, the read-after-write operation must always be performed, similar to ECP and SAFER, to detect any new fault that occurs during the writing.

Finally, certain memory blocks may have to be "retired" if they are no longer reliably written. While it is an issue orthogonal to the goal and scope of this work, such retirement-based bad memory block management is commonly used in storage devices [32], [33] and is becoming increasingly important for main memory as well [34], [35].

IV. Coverage of RDIS

The previous section described the basic idea of RDIS as well as the necessary algorithms and their hardware implementation. This section will delve further into the properties of RDIS by studying specific conditions under which RDIS fails to cover a given set of faults. There are two such conditions: (1) the progress stops because for some k, $C_k = C_{k-1}$; and (2) the capacities of the counters VX and VY are exceeded before the recursion terminates. Each of these two situations is caused by specific fault patterns as described next.

A. Coverage failure caused by a loop of faulty cells

In this section, let us first consider the case where the progress of the construction of the invertible set stops because $C_k = C_{k-1}$, for some k. We start with some preliminary definitions.

Definition. *A faulty cell, $c(i,j)$ in C_k is row and column connected (RC-connected) if row i in C_k contains at least one other faulty cell, $c(i,j')$, $j \neq j'$ and column j in C_k contains at least one other faulty cell $c(i',j)$, $i \neq i'$.*

For example, cells $c(7,3)$ in the array of Figure 2(a) is RC-connected while cell $c(0,2)$ is not RC-connected.

Definition. *A loop of faulty cells (or "loop of faults") is a sequence of $2q$ faults ($q > 1$) where every two consecutive faults in the sequence are, alternatively, in the same row or in the same column. More specifically, a loop of faulty cells is of the form $c(i_1,j_1)$, $c(i_2,j_1)$, $c(i_2,j_2)$, $c(i_3,j_2)$, ..., $c(i_q,j_q)$, $c(i_1,j_q)$.*

Definition. *A loop of faults $c(i_1,j_1)$, $c(i_2,j_1)$, $c(i_2,j_2)$, $c(i_3,j_2)$, ..., $c(i_q,j_q)$, $c(i_1,j_q)$ is alternatively-stuck (or "A-stuck") if the faults in the loop alternate between SA-R and SA-W. That is, faulty cells $c(i_1,j_1)$, $c(i_2,j_2)$, ..., $c(i_q,j_q)$, are stuck at a value, while faulty cells $c(i_2,j_1)$, $c(i_3,j_2)$, ..., $c(i_1,j_q)$, are stuck at the opposite value.*

For example, the loop in Figure 4(a) includes the sequence of faulty cells $c(2,6)$, $c(4,6)$, $c(4,4)$, $c(6,4)$, $c(6,0)$, $c(3,0)$, $c(3,1)$, $c(2,1)$. Moreover, this loop is A-stuck since cells $c(2,6)$, $c(4,4)$, $c(6,0)$, $c(3,1)$ are SA-W while cells $c(4,6)$, $c(6,4)$, $c(3,0)$, $c(2,1)$ are SA-R.

Theorem 1. *The process of constructing the invertible set stops with $C_k = C_{k-1}$ for some k, if the original array of cells, C, contains a loop of faults that is A-stuck.*

Proof. Assume that C contains the A-stuck loop of faults, $c(i_1,j_1)$, $c(i_2,j_1)$, $c(i_2,j_2)$, $c(i_3,j_2)$, ..., $c(i_q,j_q)$, $c(i_1,j_q)$. By definition, each of rows i_1, i_2, ..., i_q contains two faults, one SA-R and one SA-W, and each of columns j_1, j_2, ..., j_q contains two faults, one SA-R and one SA-W. Hence, C_1 will include rows i_1, i_2, ..., i_q and columns j_1, j_2, ..., j_q, meaning that it will include the loop of faults. Similarly, we argue that C_2 and any subsequent subarray will include the same loop of faults. Given that the number of faulty cells in C is finite, then the construction of $C_k \subset C_{k-1}$ will eventually terminate with $C_k = C_{k-1}$ for some k. ■

Theorem 2. *The process of constructing the invertible set terminates with C_K being empty for some K if the original array of cells, C, does not contain a loop of faults.*

Proof. First, we observe that if k is odd (a similar argument applies if k is even) and array, C_k, contains some faulty cells but does not contain a loop of faults, then at least one of the faulty cells in C_k, say $c(i,j)$, is not RC-connected. Second, we observe that if $c(i,j)$ is SA-R then during the construction of C_{k+1}, either $\text{VX}_{k+1}(i) = 0$ or $\text{VY}_{k+1}(j) = 0$. This is because either row i does not have a faulty cell besides $c(i,j)$ or column j does not have a faulty cell besides $c(i,j)$. This leads to the exclusion of $c(i,j)$ from C_{k+1}. If, on the other hand, $c(i,j)$ is SA-W then it will be included in C_{k+1} but will lead to $\text{VX}_{k+2}(i) = 0$ or $\text{VY}_{k+2}(j) = 0$ and thus excluded from C_{k+2}. That is, C_{k+2} is a strict subset of C_k. Moreover, given that C_k does not contain a loop of faults, then C_{k+2} does not contain a loop of faults either and the process of excluding faults from consecutive subarray continues until an empty C_K is reached. ■

B. Coverage failure caused by limited counter capacity

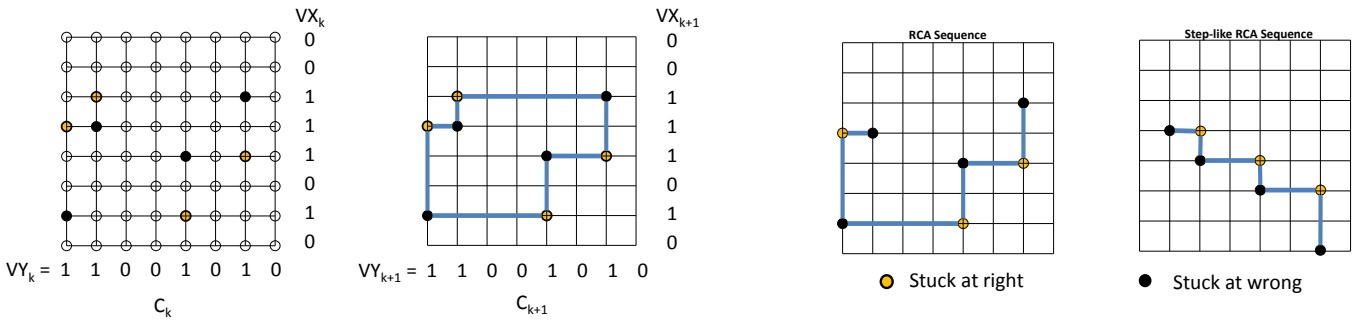
Theorem 2 implies that the process of constructing the invertible set eventually terminates successfully if the fault pattern does not include a loop of faults. However, even in the absence of a loop of faults, the process of constructing the invertible set may fail because of the limited capacity of the counters VX and VY. Specifically, if the maximum capacity of the counters is K and C_K contains both SA-W and SA-R cells, then the construction of the invertible set will fail. We explore the fault configuration that leads to this failure next.

Definition. *A row-column alternating sequence ("RCA sequence") of $2q-1$ faulty cells ($q > 1$) is a loop of $2q$ faulty cells after excluding one node.*

The above definition implies that every two consecutive faults in an RCA sequence are, alternatively, in the same row or in the same column. If the two first cells in the sequence are in the same column, then the sequence is of the form $c(i_1,j_1)$, $c(i_2,j_1)$, $c(i_2,j_2)$, $c(i_3,j_2)$, ..., $c(i_{q-1},j_q)$, $c(i_q,j_q)$, while if the first two cells are in the same row, the sequence is of the form $c(i_1,j_1)$, $c(i_1,j_2)$, $c(i_2,j_2)$, $c(i_2,j_3)$, ..., $c(i_q,j_{q-1})$, $c(i_q,j_q)$. The notation in the following definition encompasses both cases.

Definition. *an RCA sequence of $2q-1$ faulty cells, c_1, c_2, ..., c_{2q-1}, is said to be alternatively-stuck (or "A-stuck") if the first fault in the sequence is SA-W and subsequent faults alternate between SA-R and SA-W. That is, cells c_1, c_3, ..., c_{2q-1} are SA-W, while cells c_2, c_4, ..., c_{2q-2} are SA-R.*

For example, Figure 4(b) shows an RCA sequence of 7 faults which is obtained by removing cell $c(2,1)$ from the loop of faults shown in Figure 4(a). This RCA sequence is A-stuck. The step-like RCA sequence in 4(b) is isomorphic to the RCA sequence and is obtained by interchanging columns 0 and 2, rows 4 and 5, rows 4 and 6 and rows 2 and 7. The proofs of the following theorems are more intuitive if RCA sequences are envisioned as step-like. In general, any RCA sequence can be transformed to a step-like one by a series of row/column interchanges.



(a) (b)

Fig. 4: (a) A loop of faults which cannot be masked ($\text{VX}_k = \text{VX}_{k-1}$ and $\text{VY}_k = \text{VY}_{k-1}$). (b) Two isomorphic RCA sequences of faults that cannot be masked in three iterations.

Theorem 3. *The process of constructing the invertible set fails to terminate after K iterations (with C_K containing only SA-R cells or only SA-W cells) if the original array of cells, C, contains an RCA sequence of at least 2K+1 faults and this sequence is A-stuck.*

Proof. Assume that C contains an RCA sequence c_1, c_2, ..., c_{2q+1} which is A-Stuck. By construction, C_1 contains all the cells in that sequence. However, consider any of the cells c_i, *i = 2,...,2q*. If cell c_i is SA-R, then it is located in a row that contains a SA-W cell and in a column that contains a SA-W cell. Hence, this cell will be included in the subarray C_2. A similar argument applies if c_i is SA-W and consequently C_2 will contain the RCA sequence c_2, c_3, ..., c_{2q}. Applying this argument recursively leads to the conclusion that if $q \geq K$, then the subarray C_K will contain the RCA sequence c_K, ..., $c_{2q+1-(K-1)}$. In other words, if the RCA sequence contains at least *2K + 1* cells, then C_K will contain at least the three cells c_K, c_{K+1} and c_{K+2}. Being three consecutive cells in an RCA sequence, at least one of the cells is SA-R and another is SA-W, which proves the theorem. ■

Theorem 4. *The invertible set can be constructed in at most K iterations if the longest RCA sequence of faults in the original array of cells, C, contains at most $2K-1$ faults.*

Proof. We prove the theorem by induction. Specifically, we prove three Lemmas: the first establishes the base of the induction, while the other two deal with the induction steps. The proofs of the lemmas are based on the observation that the first and last cells in an RCA sequence are not RC-connected.
Lemma 1: If the longest RCA sequence in *C* is c_1, c_2, ..., c_q, then the longest RCA sequence in C_1 is c_{1+u}, ..., c_{q-v}, where *u, v ≥ 0* and both c_{1+u} and c_{q-v} are SA-W. This is because, by construction, any faulty cell that is not RC-connected in C_1 should be SA-W.
Lemma 2: For *k = 1, 3,...,* if the longest RCA sequence in C_k is c_1, c_2, ..., c_q, where c_1 and c_q are SA-W, then the longest RCA sequence in C_{k+1} is c_{1+u}, ..., c_{q-v}, where $u, v > 0$ and both c_{1+u} and c_{q-v} are SA-R. This is because, by construction, any faulty cell in C_{k+1} that is not RC-connected should be SA-R.
Lemma 3: For *k = 2, 4,...,* if the longest RCA sequence in C_k is c_1, c_2, ..., c_q where c_1 and c_q are SA-R, then the longest RCA sequence in C_{k+1} is c_{1+u}, ..., c_{q-v}, where $u, v > 0$ and both c_{1+u} and c_{q-v} are SA-W. This is because, by construction, any faulty cell that is not RC-connected in C_{k+1} should be SA-W.

The above three lemmas prove that for $k = 1, 2, \ldots,$ if the longest RCA sequence in C_k includes q cells, then the longest RCA sequence in C_{k+1} includes $q-2$ cells. Therefore, if the longest RCA sequence in C has $2K-1$ cells then the longest RCA sequence in C_K has one cell (SA-W if K is even and SA-R if K is odd). This proves that C_K includes only one type of faulty cells (SA-R or SA-W). ■

C. Defective blocks of storage cells

Consider a storage block of $n \times m$ cells of which F cells are faulty and assume that RDIS is used for masking the faults with the maximum counter capacity of K. Theorem 1-4 identify the only two types of fault patterns that can cause the failure of RDIS to mask the faults: loops of faults and RCA sequence of length $2K+1$. Hence, we call a block of cells *defective* if it contains a loop of faults or an RCA sequence of at least $2K+1$ faults.

If a block of cells with F faults is not defective, then it can be used to write/read any combination of information bits. For a small number of faults, it is possible to compute the probability of having a defective block analytically. For example, three faults cannot form a loop of faults. With four faults, the probability of having a loop of faults in an $n \times m$ block is given by $\binom{n}{2} \cdot \binom{m}{2} / \binom{n \cdot m}{4}$. Applying this formula, we find that the probability of having a defective fault pattern given four faults is 0.0012 when $n = m = 8$ and 0.00008 when $n = m = 16$. The next section gives a detailed evaluation of the probability of a block being defective in the presence of F faults.

V. Evaluation

In this section, we rely on Monte Carlo simulation to study the various parameters that affect RDIS as well as to compare it to other schemes. We assume that all cells within a storage block have equal probability of failure. To test if a $n \times m$ storage block having F faulty cells is defective, this block is modeled as a bipartite graph of $(n+m)$ nodes, one for each row and one for each column. If a cell $c(i,j)$ is faulty, then an edge connects the nodes representing row i and column j. A simple variation of depth first search algorithm (DFS) is used to detect the occurrence of a loop. To detect RCA sequences, we keep

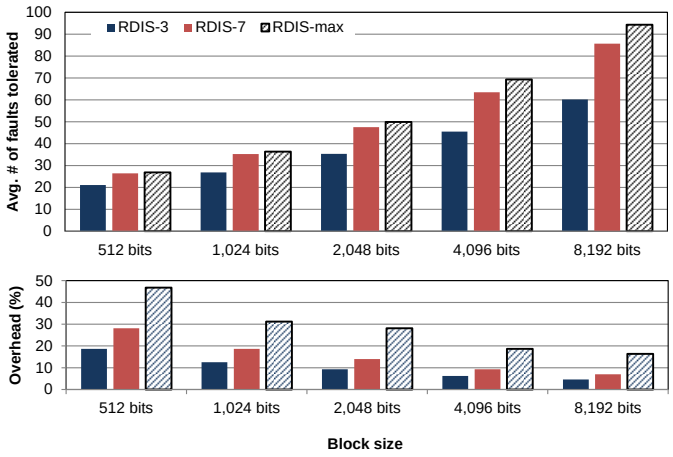


Fig. 5: Average number of faults tolerated within a block and the corresponding overhead.

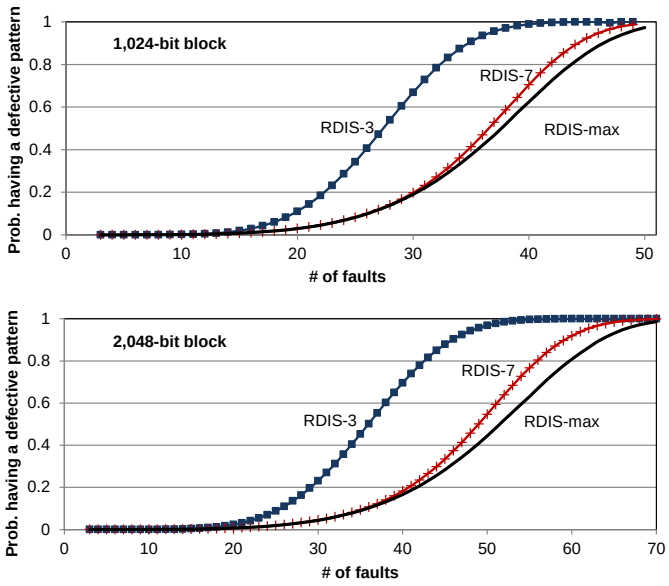


Fig. 6: Probability of failure with F faults.

track of the longest recursion depth executed by DFS while attempting to detect a loop. In other words, our algorithm either detects the existence of a loop, if any, or returns the length of the longest RCA sequence.

In [18], [22], [26], it was shown that stuck-at faults are the dominating failure source. Disturbance and resistance drift failures are prominent in multi-level PCM [36] not targeted by RDIS. Accordingly, we only simulate stuck-at faults.

A. Sensitivity to RDIS Parameters

The block size to be protected, and the overhead of auxiliary counters are the main parameters that affect RDIS. In this section, we study the performance of RDIS in light of these parameters. We simulate RDIS with 5 different block sizes of varying overhead. In addition, each block size is simulated with three variations of RDIS. The first limits the capacity of the auxiliary counters to 3, the second to 7 and the last to the number required to tolerate the maximum possible RCA sequence. Hereafter, we denote these three variations as RDIS-3, RDIS-7 and RDIS-max. For each block size, we report the average number of faults that can be tolerated as well as the probability of failure with F faults for $F = 1, 2, \ldots$. Given a block of size $n \times m$, the corresponding overhead is $(n \cdot s + m \cdot s)/(n \cdot m)$, where s is the size of each auxiliary counter in bits. For example, a 128-byte data block arranged as a 32×32 bit array incurs a 12.5% overhead for $s = 2$. It is to be noted that for a fixed s, the overhead percentage decreases with the increase in the size of the protected storage block.

Figure 5 shows the average number of faults that can be tolerated for various block sizes and the overhead of the three RDIS configurations. The overhead for RDIS-max is calculated based on the maximum length of an RCA sequence that can occur within a block. Specifically, for $n \times m$ block, the maximum length of an RCA sequence is $n + m - 1$ (imagine a step-like RCA sequence starting at $c(0, 0)$ and ending at $c(n - 1, m - 1)$), and thus a counter of size $s = \lceil \log_2((n + m - 1)/2) \rceil$ bits is sufficient for recovery according to Theorem 4. From the results shown in Figure 5, it can be inferred that the average number of faults tolerated within each block increases with the overhead. Hence, the choice of auxiliary counters capacity for RDIS represents a trade-off between the number of faults that can be tolerated and the overhead. RDIS-3 is shown to correct many errors robustly at the smallest overhead.

The main advantage of RDIS is the large probability to tolerate a relatively large number of faults. Figure 6 shows that given F faults, the probability of forming a loop or RCA sequence increases at low pace with the increase of F. Accordingly, RDIS is capable of tolerating a high number of faults beyond what it guarantees. Even though we show the results for blocks of 1,024 bits and 2,048 bits, other block sizes exhibit the same trend and are omitted for brevity. These results show that RDIS-3 is capable of tolerating a notable number of faults while incurring an affordable overhead. As a matter of fact, the relative increase in the number of faults tolerated by increasing the counters capacity beyond three is not proportional to the increase in the overhead. Consequently, we consider only RDIS-3 in the rest of our evaluation.

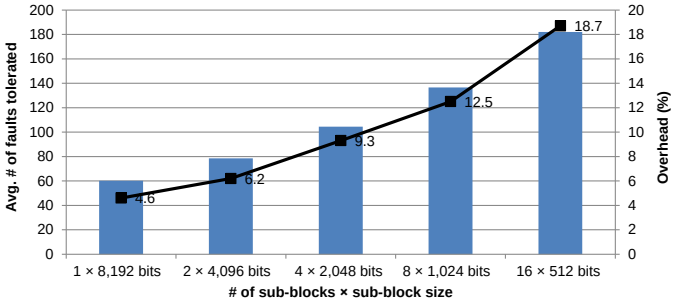


Fig. 7: Average number of faults tolerated in 1 KB memory block (bar) and the corresponding overhead (line).

As indicated earlier, varying the counters capacity is one way of affecting the trade-off between the number of faults tolerated and the overhead. Another way of affecting this trade-off is through protecting a memory block as a combination of

smaller sub-blocks while fixing the counters capacity. Figure 7 shows the average number of tolerated faults in 1 KB of memory. We protect a block of 1 KB of memory through dividing it into smaller sub-blocks. Each sub-block is protected with RDIS-3. The 1 KB block is considered defective as soon as any of its sub-blocks becomes defective. Such an approach leads to a significant increase in the average number of tolerated faults as depicted in Figure 7.

B. Comparison with existing schemes

In this section, we evaluate the performance of RDIS against other schemes. Specifically, we compare RDIS-3 with SAFER which was shown in [17] to be superior to ECP and ECC. The overhead of SAFER depends on the number of groups that a block is partitioned into.[3] Hence, RDIS-3 is compared with two SAFER configurations that have an overhead just smaller and just larger than RDIS-3. Two metrics are used for comparison: (1) the probability of failure with F faults; and (2) the average number of faults that can be tolerated in a storage block.

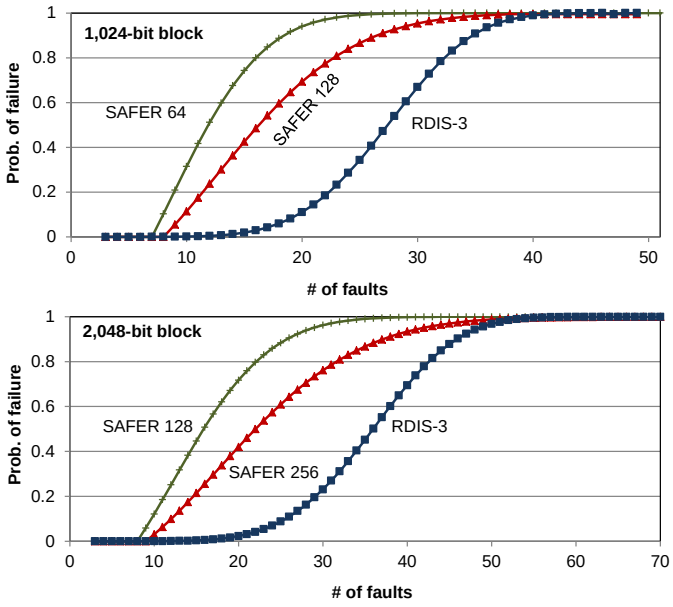


Fig. 8: RDIS vs. SAFER: Probability of failure with F faults.

Both RDIS and SAFER can probabilistically tolerate more faults than what they guarantee. With n groups, SAFER (denoted by SAFER n) guarantees the tolerance of $\log_2 n + 1$ faults while RDIS can always tolerate three faults. Any additional fault is tolerated by both schemes with a certain probability. Figure 8 shows the probability of failure of a storage block after F faults. Though SAFER guarantees the tolerance of more faults than RDIS, the probability of failure after what it guarantees increases at a high rate. On the contrary, the probability of failure for RDIS increases at a substantially low rate. In addition, the probability of failure for RDIS in the interval of faults that SAFER guarantees is remarkably low as depicted in Table I, even when compared with the higher overhead version of SAFER. Though we show the results for two different block sizes for brevity, the same trend is manifested with other block sizes.

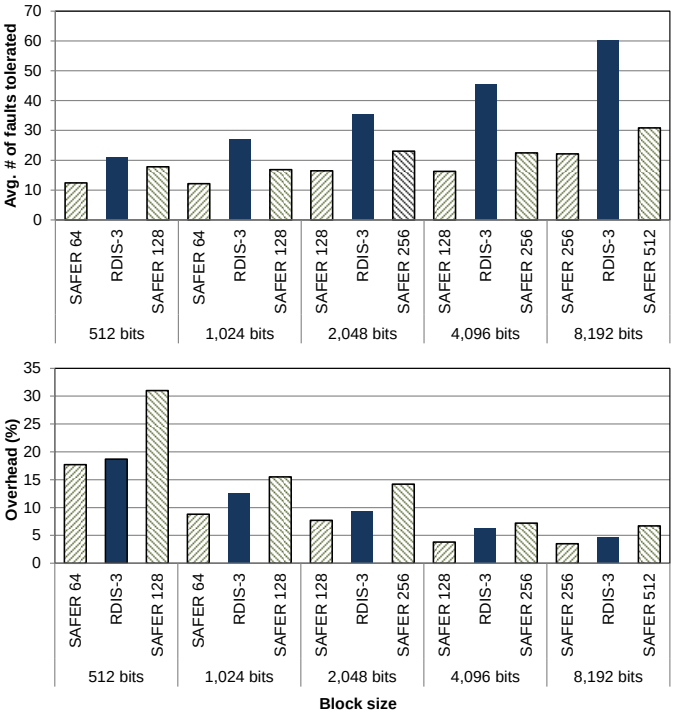


Fig. 9: RDIS vs. SAFER: Average number of tolerated faults and the corresponding overhead.

The advantage of RDIS over SAFER, when it comes to the low probability of failure, is manifested by the average number of faults that each scheme can tolerate as shown in Figure 9. The results show a significant advantage for RDIS over SAFER. For example, RDIS-3 is capable of tolerating 18% more faults than SAFER 128 with a 512-bit block size and 95% more faults than SAFER 512 with a 8,192-bit block. Note that this increase in the average number of faults tolerated is realized with lower overhead.

The presented results demonstrate that RDIS is capable of tolerating a large number of faults on average and is characterize by a probability of failure that increases at a low rate with the increase in the number of faults. With a block size of at least 1,024 bits, the overhead of RDIS is within the 12.5% standard.

C. Protecting auxiliary data

Similarly to SAFER, RDIS cannot recover from faults in the auxiliary bits. Specifically, it is assumed that the storage of those bits is error free. The ECP scheme [16] is different in that regard in the sense that it can protect the cells that replace faulty cells. To this end, we can use ECP to protect the auxiliary counters of RDIS-3 against faults. For this, we can allocate π pointers to protect the auxiliary bits. We simulated RDIS-3 with various values of π and concluded that $\pi = 5$ is a suitable value since it maintains the high number of faults tolerated when counters are assumed to be fault-free.

[3]The overhead of SAFER when used to protect a block of N bits using n groups is: $(\lceil \log_2 n \rceil \times \lceil \log_2 \lceil \log_2 N \rceil \rceil) + \lceil (\log_2 \lceil \log_2 n \rceil + 1) \rceil + n$.

	Faults #	4	5	6	7	8	9	10	11	12	13
1 Kbits	SAFER 128	0	0	0	0	0	0.055	0.11	0.17	0.23	0.30
	RDIS-3	6×10^{-6}	3×10^{-5}	8×10^{-5}	2×10^{-4}	4×10^{-4}	0.00008	0.0015	0.0025	0.0045	0.0074
2 Kbits	SAFER 256	0	0	0	0	0	0	0.03	0.06	0.09	0.13
	RDIS-3	2×10^{-6}	5×10^{-6}	1×10^{-5}	4×10^{-5}	1×10^{-4}	2×10^{-4}	0.00033	0.00057	0.00093	0.0015

TABLE I: RDIS vs. SAFER: Probability of failure.

Hereafter, we denote the scheme that protects the auxiliary bits of RDIS-3 (can be applied to any version of RDIS) as RDIS-3PX.

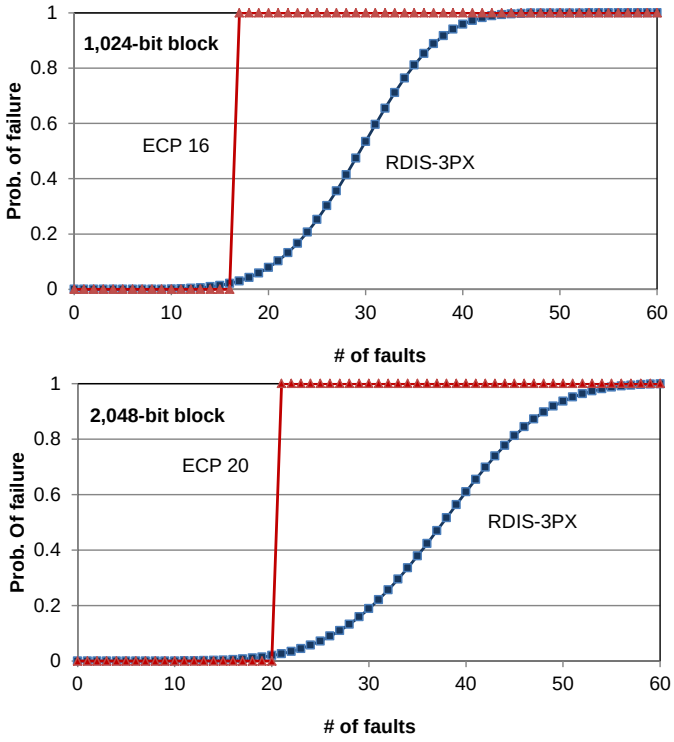


Fig. 10: RDIS-3PX vs. ECP: Probability of failure with F faults.

Subsequently, we compare RDIS-3PX against ECP itself. We assign to ECP the minimum number of pointers, n, that makes its overhead larger than RDIS-3PX and denote the scheme by ECP n.[4] For various block sizes, we study the probability of failure with F faults as well as the average number of tolerated faults achieved by each scheme. When it comes to the probability of failure with F faults, Figure 10 shows that ECP cannot recover from faults beyond the provided number of correction pointers. To the contrary, RDIS is capable of remarkably tolerating faults beyond what it guarantees. Furthermore, RDIS exhibits a notably low probability of failure within the error free window of ECP. Again, these results are manifested in the average number of faults that both schemes can tolerate as depicted in Figure 11. For example, RDIS tolerates up to 81% more faults with block size of 8,192 bits. It is to be noted that RDIS' average number of faults

[4]The overhead of ECP n when used to protect a block of N bits using n pointers is: $n(\lceil \log_2 N \rceil + 1) + 1$.

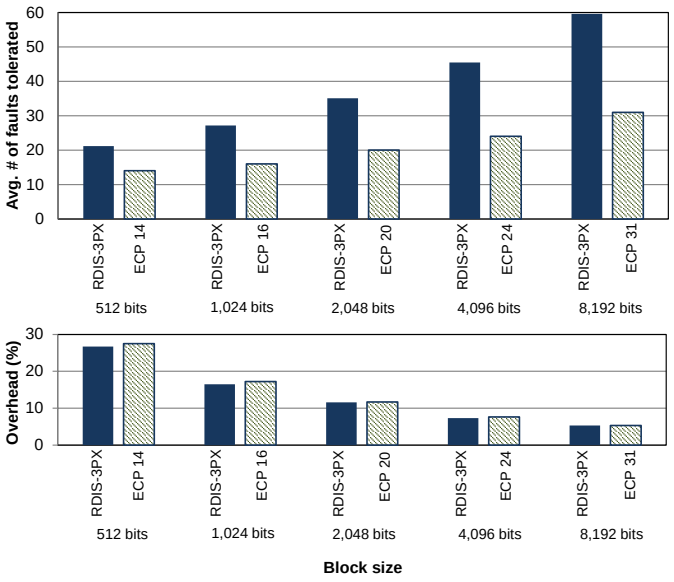


Fig. 11: RDIS-3PX vs. ECP: Average number of tolerated faults and the corresponding overhead.

tolerated corresponds to faults occurring both in the protected block and the auxiliary bits.

The presented results make it clear that RDIS can tolerate more faults with higher probability than previously proposed schemes using the same assumptions and fault model. It is particularly suited for large blocks of 128 bytes or more.

VI. Conclusions

The limited write endurance is the major weakness of emerging resistive memories. Accordingly, robust error recovery schemes are required to mask off hard errors and prolong the lifetime of a resistive memory chip. In this paper, we have presented and evaluated RDIS, a recursively defined invertible set scheme to tolerate multiple stuck-at hard faults. Our extensive evaluation shows that RDIS achieves a very low probability of failure on hard fault occurrences, which increases slowly with the relative increase in the number of faults. This characteristic allows RDIS to effectively recover from a large number of faults. For example, RDIS can recover from 46 hard faults on average when the block size is 512 bytes (storage sector size) while incurring a low overhead of 6.2%. Furthermore, we have shown that realizing RDIS in hardware is fairly straightforward and is off the critical data access path.

Given its high error tolerance potential, RDIS fits the need to recover from many faults in emerging resistive memories.

We believe that RDIS provides a very robust memory substrate to a system and allows system designers to focus their efforts on effective integration and management of resistive memory capacity at higher levels, for better overall system performance and reliability.

REFERENCES

[1] K. Kim, "Technology for sub-50nm DRAM and NAND flash manufacturing," 2005, pp. 323–326.

[2] ITRS, http://public.itrs.net, 2007.

[3] R. F. Freitas and W. W. Wilcke, "Storage-class memory: The next storage system technology," *IBM Journal of Research and Development*, vol. 52, no. 4.5, pp. 439–447, July 2008.

[4] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. H. Lam, "Phase-change random access memory: A scalable technology," *IBM Journal of Research and Development*, vol. 52, no. 4.5, pp. 465–479, July 2008.

[5] G. Dhiman, R. Ayoub, and T. Rosing, "PDRAM: A hybrid PRAM and DRAM main memory system," in *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, july 2009, pp. 664–669.

[6] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," *SIGARCH Comput. Archit. News*, vol. 37, pp. 2–13, June 2009.

[7] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, dec. 2009, pp. 14–23.

[8] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," *SIGARCH Comput. Archit. News*, vol. 37, pp. 14–23, June 2009.

[9] S. Cho and H. Lee, "Flip-N-Write: A simple deterministic technique to improve PRAM write performance, energy and endurance," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, dec. 2009, pp. 347–357.

[10] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "Consistent and durable data structures for non-volatile byte-addressable memory," ser. FAST'11, February 2011.

[11] C. W. Smullen, V. Mohan, A. Nigam, S. Gurumurthi, and M. R. Stan, "Relaxing non-volatility for fast and energy-efficient stt-ram caches," ser. HPCA, February 2011, pp. 50–61.

[12] X. Dong and Y. Xie, "Adams: adaptive mlc/slc phase-change memory design for file storage," ser. ASPDAC '11, 2011, pp. 31–36.

[13] I. Micron Technology, "Phase change memory (pcm)," http://www.micron.com/products/pcm, 2011.

[14] H. Chung, B. H. Jeong, B. Min, Y. Choi, B.-H. Cho, J. Shin, J. Kim, J. Sunwoo, J. min Park, Q. Wang, Y. jun Lee, S. Cha, D. Kwon, S. Kim, S. Kim, Y. Rho, M.-H. Park, J. Kim, I. Song, S. Jun, J. Lee, K. Kim, K. won Lim, W. ryul Chung, C. Choi, H. Cho, I. Shin, W. Jun, S. Hwang, K.-W. Song, K. Lee, S. whan Chang, W. Y. Cho, J.-H. Yoo, and Y.-H. Jun, "A 58nm 1.8V 1Gb PRAM with 6.4MB/s Program BW," in *IEEE ISSCC*, February 2011, pp. 500–502.

[15] Y. Choi, I. Song, M.-H. Park, H. Chung, B. C. Sanghoan Chang, J. Kim, Y. Oh, D. Kwon, J. Sunwoo, J. Shin, Y. Rho, C. Lee, M. G. Kang, J. Lee, Y. Kwon, S. Kim, J. Kim, Y.-J. Lee, Q. Want, S. Cha, S. Ahn, H. Horii, J. Lee, K. Kim, H. Joo, K. Lee, Y.-T. Lee, J.-H. Yoo, and G. Jeong, "A 20nm 1.8V 8Gb PRAM with 40MB/s Program Bandwidth," in *IEEE ISSCC*, February 2012.

[16] S. Schechter, G. H. Loh, K. Straus, and D. Burger, "Use ECP, not ECC, for hard failures in resistive memories," *SIGARCH Comput. Archit. News*, vol. 38, pp. 141–152, June 2010.

[17] N. H. Seong, D. H. Woo, V. Srinivasan, J. Rivers, and H.-H. Lee, "SAFER: Stuck-At-Fault Error Recovery for Memories," in *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, dec. 2010, pp. 115–124.

[18] D. H. Yoon, N. Muralimanohar, J. Chang, P. Ranganathan, N. Jouppi, and M. Erez, "FREE-p: Protecting non-volatile memory against both hard and soft errors," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, feb. 2011, pp. 466–477.

[19] S. Kang, W. Y. Cho, B.-H. Cho, K.-J. Lee, C.-S. Lee, H.-R. Oh, B.-G. Choi, Q. Wang, H.-J. Kim, M.-H. Park, Y. H. Ro, S. Kim, C.-D. Ha, K.-S. Kim, Y.-R. Kim, D.-E. Kim, C.-K. Kwak, H.-G. Byun, G. Jeong, H. Jeong, K. Kim, and Y. Shin, "A 0.1 μm 1.8-V 256-Mb Phase-Change Random Access Memory (PRAM) With 66-MHz Synchronous Burst-Read Operation," *Solid-State Circuits, IEEE Journal of*, vol. 42, no. 1, pp. 210–218, jan. 2007.

[20] K.-J. Lee, B.-H. Cho, W.-Y. Cho, S. Kang, B.-G. Choi, H.-R. Oh, C.-S. Lee, H.-J. Kim, J.-M. Park, Q. Wang, M.-H. Park, Y.-H. Ro, J.-Y. Choi, K.-S. Kim, Y.-R. Kim, I.-C. Shin, K.-W. Lim, H.-K. Cho, C.-H. Choi, W.-R. Chung, D.-E. Kim, Y.-J. Yoon, K.-S. Yu, G.-T. Jeong, H.-S. Jeong, C.-K. Kwak, C.-H. Kim, and K. Kim, "A 90 nm 1.8 V 512 Mb Diode-Switch PRAM With 266 MB/s Read Throughput," *IEEE JSSC*, vol. 43, pp. 150–162, January 2008.

[21] K. Bourzac, "Memristor memory readied for production," http://www.technologyreview.com/computing/25018/, April 2010.

[22] K. Kim and S. J. Ahn, "Reliability investigations for manufacturable high density pram," ser. IRPS, 2005.

[23] S. Lee, J. hyun Jeong, T. S. Lee, W. M. Kim, and B. ki Cheong, "A Study on the Failure Mechanism of a Phase-Change Memory in Write-Erase Cycling," *IEEE Electron Device Letters*, vol. 30, no. 5, pp. 449–450, May 2009.

[24] B. Gleixner, F. Pellizzer, and R. Bez, "Reliability characterization of phase change memory," ser. NVMTS, October 2009.

[25] M. Qureshi, A. Seznec, L. Lastras, and M. Franceschini, "Practical and secure PCM systems by online detection of malicious write streams," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, feb. 2011, pp. 478–489.

[26] N. H. Seong, D. H. Woo, and H.-H. S. Lee, "Security refresh: prevent malicious wear-out and increase durability for phase-change memory with dynamically randomized address mapping," *SIGARCH Comput. Archit. News*, vol. 38, pp. 383–394, June 2010.

[27] B.-D. Yang, J.-E. Lee, J.-S. Kim, J. Cho, S.-Y. Lee, and B.-G. Yu, "A Low Power Phase-Change Random Access Memory using a Data-Comparison Write Scheme," ser. ISCAS, May 2007, pp. 3014–3017.

[28] W. Wong, "A chat about micron's clearnand technology," *electronic design*, December 2010.

[29] R. Hamming, "Error Detecting and Error Correcting Codes," *Bell System Technical Journal*, vol. 26, no. 2, pp. 147 – 160, 1950.

[30] R. Bose and D. Ray-Chaudhuri, "On a class of error correcting binary group codes," *Information and Control*, vol. 3, no. 1, pp. 68–79, 1960.

[31] D. Strukov, "The area and latency tradeoffs of binary bit-parallel BCH decoders for prospective nanoelectronic memories," in *Signals, Systems and Computers, 2006. ACSSC '06. Fortieth Asilomar Conference on*, nov. 2006, pp. 1183–1187.

[32] C. Park, P. Talawar, D. Won, M. Jung, J. Im, S. Kim, and Y. Choi, "A high performance controller for nand flash-based solid state disk (nssd)," in *Non-Volatile Semiconductor Memory Workshop, 2006. IEEE NVSMW 2006. 21st*, feb. 2006.

[33] Y. Cai, E. Haratsch, M. McCartney, and K. Mai, "Fpga-based solid-state drive prototyping platform," in *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, may 2011.

[34] D. Tang, P. Carruthers, Z. Totari, and M. Shapiro, "Assessment of the effect of memory page retirement on system ras against hardware faults," in *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, june 2006.

[35] M. Rahman, B. R. Childers, and S. Cho, "Comet: Continuous online memory test," in *Proceedings of the 17th IEEE Pacific Rim Int'l Symposium on Dependable Computing (PRDC)*, December 2011.

[36] N. Papandreou, H. Pozidis, T. Mittelholzer, G. Close, M. Breitwisch, C. Lam, and E. Eleftheriou, "Drift-tolerant multilevel phase-change memory," in *Memory Workshop (IMW), 2011 3rd IEEE International*, may 2011, pp. 1–4.