Power of One Bit: Increasing Error Correction Capability with Data Inversion

Rakan Maddah[1], Sangyeun Cho[2,1], and Rami Melhem[1]
[1]Computer Science Department, University of Pittsburgh
[2]Memory Solutions Lab, Memory Division, Samsung Electronics Co.
{*rmaddah,cho,melhem*}@*cs.pitt.edu*

***Abstract*—Phase-change memory (PCM) has emerged as a candidate that overcomes the physical limitations faced by DRAM and NAND flash memory. While PCM has desirable properties in terms of scalability and energy, it suffers from limited endurance. Repeated writes cause PCM cells to wear out and get permanently stuck at either 0 or 1. Recovering from stuck-at faults through a proactive error correction scheme is essential for the widespread adoption of PCM.**

In this paper, we propose *data inversion* as a practical technique to increase the number of faults that an error correction code can cover. Since stuck-at cells can still be read, errors are manifested only when a worn-out cell is programmed with a bit value different than the value it is stuck at. After a write operation fails for a given block of data, data inversion attempts another write operation with all original data bits inverted. Inverting the data is likely to bring the number of errors within the nominal capability of the deployed error correction code. Requiring only one additional auxiliary bit, data inversion can double the capability of an error correction code and extends the lifetime by up to 34.5%.

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

I. Introduction

As DRAM and NAND flash memories face physical limitations that hinder their further scaling [1], the quest for an alternative memory technology became a necessity. Amongst several memory candidates, phase-change memory (PCM) is emerging as one of the most promising technologies due to its desirable characteristics in terms of superior scalability, low access latency and negligible standby power. Assessments and measurements show that PCM compete favorably with DRAM and NAND flash in terms of performance while providing improved scalability, density and endurance [2]–[4].

Instead of representing information as the presence or absence of electrical charges, PCM encodes bits in different physical states of chalcogenide alloy that consists of Ge, Sb and Te. Data is stored in PCM devices in the form of either a low resistance crystalline state (SET) or a high resistance amorphous state (RESET). Switching between the states happens through the application of different programming currents that melt and then re-solidify the material into one of the SET/RESET states. Unfortunately, each PCM cell can endure only a limited number of SET/RESET cycles. The heating and cooling process to program a cell leads to frequent expansions and contractions of the material. Consequently, the heating element detaches from the chalcogenide material after sustaining 10^6 to 10^8 writes on average, which results in a stuck-at hard fault that can be subsequently read but not reprogrammed [5].

In order to make PCM a viable memory technology for high volume manufacturing, mitigating the wear-related failures is essential. While wear leveling [6] and suppressing unnecessary bit flips [7] are good techniques to preserve the endurance and lessen the degradation rate of PCM cells, stuck-at faults eventually occur. In addition, parametric and random variations in the manufacturing process results in a non uniform distribution of cell lifetime. Consequently, early failures of cells are common. Therefore, proactive multi-bit error correction schemes are a necessity to gracefully recover from numerous stuck-at faults that accrue within a memory block [8]–[11]. Unlike DRAM and NAND flash, PCM cells are not susceptible to radiation induced transient faults in the foreseeable future due to the high energy required to change the state of a PCM cell [12]. Hence, stuck-at cells are the major failure mode that needs to be alleviated in PCM.

The fact that a failed stuck-at cell is still readable makes the nature of errors to be data dependent [10]. In a sense, a faulty cell is erroneous only when the bit value read is different than the intended bit value to be written. For example, if a cell is stuck-at-1 then an error occurs only when the cell needs to be written with 0. Consequently, a stuck-at cell can be classified into either “stuck-at-wrong” (SA-W) or “stuck-at-right” (SA-R) depending on whether the cell needs to be written with a value different or identical to the value it is stuck at [11]. Henceforth, the errors that an error correction scheme masks correspond to those cells that happen to be SA-W after a given write operation.

This paper proposes *data inversion* to boost the error correction capability of error correction schemes, such as BCH [8] and SECDED [13], at a very small cost. Given an error correction scheme with the capability of masking t errors, a write operation fails only when $t+1$ faults are SA-W. Following a write failure, our technique attempts a second write operation with the original data inverted. By applying data inversion, SA-W cells become SA-R and vice versa. Thus, if the number of SA-R faults in the initial write pattern was smaller than or equal to the nominal capability of the error correction scheme, then applying data inversion makes the second write attempt successful as the number of SA-W faults

*This work is supported in part by National Science Foundation grants CNS-1012070 and CCF-1064976.

becomes within the capability of the error correction scheme.

To increase the error correction capability of a code, the number of auxiliary bits has to be increased. In addition, the complexity of error correction codes increases linearly with the increase in the number of errors that can be tolerated [14]. Data inversion levies very little design complexity as it enables an error correction code to cover more errors through a simple inversion operation without altering the encoding and decoding complexity of the code. As a matter of fact, data inversion introduces an additional auxiliary bit that serves as a polarity bit that is set based on the occurrence of the inversion step. Hence, data inversion reduces the bit costs as well as computational complexity while allowing more errors to be covered.

A write request completes successfully if the number of SA-W cells is smaller than or equal to the nominal capability of the error correction scheme. In this sequel, data inversion can be looked at as a data mapping technique. It offers the choice between two data encodings, one un-inverted and one inverted. If writing the un-inverted pattern happens to induce a number of SA-W cells that is greater than the nominal capability of the error correction scheme, then writing the inverted pattern may induce a number of SA-W cells that is within the nominal capability. Hence, the data to be written is mapped to the pattern that leads to a successful write, if any.

We present two variations of data inversion. In the first, we integrate the polarity bit as part of the codeword. In the second, we take the polarity bit out of the codeword. We show that in the first variation the number of stuck-at faults that can be tolerated depends on the distribution of the faults within the protected block. As for the second variation, we show that the number of stuck-at faults that can be tolerated is doubled. Furthermore, we show that data inversion may require an extra write operation before completing a write request successfully only after the number of stuck-at faults within a protected block exceeds the nominal capability of the error correction code. Nevertheless, the need for an extra write is rare due to the data dependent nature of errors. Our findings reveal that data inversion can increase the lifetime of main memory by up to 34.5% and secondary storage by up to 16.1%.

The remainder of this paper is organized as follows. Section II gives the details of the proposed technique by proving its capability in increasing the number of faults that can form within a memory block before turning defective. Section III presents the flow of execution of data inversion. Section IV analyses potential overheads. Section V presents experimental evaluation. Section VI reports the related work, and finally, Section VII summarizes the paper.

II. Data Inversion

Data inversion is a simple technique to boost the error correction capability of an error correction scheme. When a block write request causes the stuck-at cells to entail a number of errors above the nominal capability of the deployed error correction scheme, data inversion resubmits a new write request after inverting the original write pattern. This simple inversion operation can make the number of manifested errors within the capability of the error correction code. Thus, data inversion is capable of completing otherwise failed write requests successfully.

Before we delve into the details of data inversion, let us start with some preliminary definitions that will help us set the ground for the concepts of data inversion.

Definition 1. *A memory/storage block is non-defective if any data pattern can be written successfully on the block.*

Definition 2. *A memory/storage block is defective if some data patterns cannot be written successfully on the block.*

Definitions 1 and 2 establish the distinction between a defective and a non-defective block. A defective block is not free of stuck-at faults, but these faults can never lead to a pattern of errors uncorrectable by the error correction code. Conversely, the stuck-at faults in a defective block can lead to write failures. However, not every write request necessarily fails on a defective block. In fact, failures are related to specific data patterns that push the number of SA-W bits beyond the capability of a given error correction code.

A. Integrated Protection

When a write request is submitted, the data bits are augmented with the polarity bit set to 0. Subsequently, the error correction code computes the auxiliary information to protect against errors in the original data bits and the polarity bit. Next, the codeword (data bits + polarity bit + auxiliary bits) is physically written. If the codeword manifests a number of SA-W cells larger than the error correction capability of the code, then data inversion kicks in. The codeword is recomputed with inverted data bits and the polarity bit set to 1.

The inversion step has the potential effect of increasing the number of stuck-at faults that can form within the data bits, while preserving the non-defectiveness of the memory block as described in the following theorem.

Theorem 1. *Given a memory/storage block protected by an error correction code that can correct up to t-bit errors, applying data inversion with integrated protection extends the correction capability of the code to $Q + R$ faults such that $Q/2 + R = t$, where Q is the number stuck-at faults in the data bits and R the number of stuck-at faults in the auxiliary bits. That is, the block is defective only if $(Q/2 + R) > t$.*

Proof: By construction, an error correction code of capability t fails only when at least $t + 1$ errors are manifested, i.e., $t + 1$ SA-W faults in the context of the stuck-at fault model. In the worst case, the number of SA-W cells is equal to the number of SA-R cells within the data part after a write failure. Therefore, $Q/2$ stuck-at faults can happen to be SA-W in the worst case after inverting the data part. In addition, at most R errors can be manifested within the auxiliary bits in the worst case as recomputing the auxiliary information does not necessarily change the value of every auxiliary bit. Hence, a memory/storage block becomes defective only if $(Q/2+R) > t$. ■

It follows from Theorem 1 that data inversion makes the defectiveness of memory blocks correlated with the distribution of the stuck-at faults in between the data and the auxiliary bits within the blocks. It is highly likely that the distribution of faults allows increasing the number of faults within a block before it becomes defective. Hence, data inversion can be looked at as increasing the capability of the error correction code.

B. Un-integrated Protection

Data inversion is a post failure technique. As a matter of fact, data inversion interferes with the write operation only after a failure occurs. As long as the number of stuck-at faults accrued within a block is still within the capability of the correction code, no write failure could occur. Accordingly, the polarity bit does not start to be toggled until that point is reached. Even after the number of stuck-at faults gets above the capability of the correction code, the polarity bit is not toggled frequently due to the data dependent nature of errors that makes write failures rare.

Given its infrequent toggling, we consider separating the polarity bit out of the protection scope of the error correction scheme. Such an approach could make data inversion vulnerable to a single point of failure. Nevertheless, the raw endurance of the polarity bit is expected to be resilient enough to sustain its infrequent toggling. Furthermore, the vulnerability to polarity bit failures can be mitigated through redundancy techniques such as triple modular redundancy (TMR). Taking the polarity bit out of the codeword brings two enhancements to data inversion. The first enhancement is to abolish the need to recompute the auxiliary information prior to the second write attempt. With integrated protection, the auxiliary information of the first write attempt protects against errors with the value of the polarity bit is set to 0. In the event of a write failure, the integrated protection scheme recomputes the auxiliary information with the value of the polarity bit set to 1 for the second write attempt. By separating the polarity bit, the data and auxiliary bits are inverted with no need to change the value of any of the original bits. Accordingly, with un-integrated protection the additional write is a simple inversion of the entire codeword. The second and most important enhancement is that separating the polarity bit out of the codeword guarantees doubling the number of stuck-at fault before a block turns into a defective one. Theorem 2 proves the conjuncture.

Theorem 2. *Given a memory/storage block protected by an error correction code that can cover up to t-bit errors, applying data inversion with un-integrated protection can correct up to $2t+1$ stuck-at faults. That is, the block is defective only after $2t+2$ stuck-at faults are accrued.*

Proof: By construction, an error correction code of capability t fails only when at least $t+1$ errors are manifested. Since applying data inversion exchanges the role of SA-W and SA-R, at least $t+1$ SA-R faults in addition to $t+1$ SA-W faults have to exist in the block for the error correction code to fail after inverting the codeword to be written. Thus, a memory/storage block becomes defective only after $2t+2$ stuck-at faults are accrued. ■

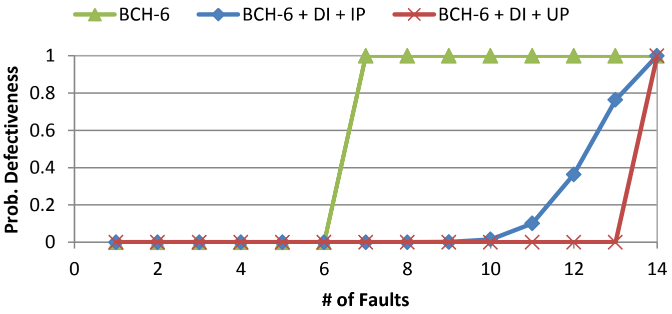

Fig. 1: Probability of defectiveness as a function of the number of stuck-at faults, where a BCH-6 code protects a block of size 512 bits. "DI" denotes data inversion, "IP" denotes integrated protection and "UP" denotes un-integrated protection.

It follows from Theorem 2 that separating the polarity bit out of the codeword allows an error correction code to be capable of extending the non-defectiveness of a block until the number of faults within the block becomes double the capability of the correction code irrespective of the distribution of the faults. Before that point is reached, the probability of defectiveness is 0. As when the polarity bit is integrated with the data part, a probability of defectiveness can form as when the number of faults gets above the capability of the correction code. Nevertheless, the probability of defectiveness increases slowly with the relative increase in the number of stuck-at faults within a memory block as depicted in Fig. 1.

C. Surviving Polarity Bit Defectiveness

Since data inversion with un-integrated protection does not protect the polarity bit, the defectiveness of the polarity bit may result in making data inversion failing to meet the premise of doubling the number of faults that can be tolerated. However, the number of faults that can be tolerated is never below the nominal capability of the error correction code even if the polarity bit is defective. Imagine that a polarity bit is defective by manufacturing and stuck at 1. As the polarity bit is not protected, covering the faulty cell is not possible. However, the codeword can always be inverted to match the status of the polarity bit in which case the error correction code can cover errors within the inverted codeword up to the nominal capability.

To gain insight about the effect of a defective polarity bit on blocks' defectiveness, Table I compares the probability of defectiveness of the integrated protection scheme against the un-integrated scheme where the polarity bit could turn defective for a 512-bit block protected by a BCH-6 code. Similarly to integrated protection, defectiveness with un-integrated protection can now occur when the number of faults exceeds the nominal capability of the error correction code. Yet, it is notable that the probability of defectiveness for the un-integrated scheme remains low until number of faults within the block grows above the threshold that the un-integrated scheme can potentially tolerate. This is because the probability of the polarity bit turning defective is low.

Allowing operation with a defective polarity bit is possible, but it changes an important aspect of data inversion. That is, a second write attempt may be required only after the number of

# of faults	7	8	9	10	11	12	13	14
IP	1.59E-07	3.07E-05	0.001	0.015	0.100	0.363	0.763	1
UP with PL defectiveness	0.012	0.013	0.015	0.017	0.019	0.020	0.022	1

TABLE I: Probability of defectiveness as a function of the number of faults within a 512 bit block protected by BCH-6 code complemented with data inversion with integrated protection (IP) and un-integrated protection (UP) with the possibility of the polarity bit (PL) turning defective.

faults exceeds the nominal capability of the error correction code. Nevertheless, the target of this paper is to present a technique that exploits the major fault model of PCM, which is worn-out cells. In general, memory manufacturers follow a rigorous testing phase to eliminate blocks where certain cells are defective by manufacturing. Even though manufacturers allow variation in cells lifetime, a minimal level of endurance has to be achieved. Accordingly, we do not consider the case of continuing operations with a defective polarity bit. In the event of a defective polarity bit by manufacturing or because of low endurance, one can declare the block associated with the polarity bit defective. Yet, redundancy techniques could be applied to mitigate the unlikely event of polarity bit failure as indicated in the previous section.

III. Execution Flow

The execution of write and read requests differs between data inversion with integrated and un-integrated protection. When a write request is issued, data inversion with integrated protection augments the data bits with the polarity bit set to 0. Subsequently, the error correction code computes the auxiliary information with a correction capability that covers the polarity bit. Next, the codeword (data bits + polarity bit + auxiliary bits) is physically written on the PCM medium. In the event that the codeword exhibits a number of SA-W cells above the capability of the error correction code, a second write operation is attempted. Before submitting the second write attempt, the data bits are inverted, the polarity bit is set to 1 and the auxiliary information is recomputed. If the second write attempt exhibit a number of SA-W cells within the reach of the error correction code, then the write request has completed successfully. At read time, the data bits and the polarity bits are retrieved after the error correction code decodes the codeword and corrects errors. When the value of the polarity bit is 1, the data bits are inverted to finally obtain the intended data that had to be written initially.

Fig. 2 gives an example where a byte is protected by an error correction code of capability 1 complemented with data inversion with integrated protection. Protecting a byte against one error requires a Galois field of size 4, which is large enough to accommodate the need to augment the data bits with the polarity bit. Fig 2 shows that the first write attempt fails as 3 SA-W cells are manifested. Consequently, a second write with inverted data bits, polarity bit set to 1 and recomputed auxiliary bits is attempted. The second write completes successfully as only one SA-W cell is manifested. At read time, the error correction code is capable of recovering from the single error and correctly decodes the data bits and the polarity bit. As a last step, the data bits are inverted since the value of the polarity bit is 1.

Data inversion with un-integrated protection separates the polarity bit from the codeword, which alters the execution flow of write and read requests. When a write request is issued, the error correction code computes the auxiliary information with correction capability that does not cover the polarity bit. Subsequently, the polarity bit is set to 0 and it is physically written along with the codeword (data bits + auxiliary bits) into the PCM meduim. In the event of a write failure, another write operation is attempted with the codeword inverted and the polarity bit set to 1. At read time, the codeword is read inverted if the value of the polarity bit is 1. Finally, the data bits are retrieved through decoding the codeword by the error correction code.

Fig. 3 gives an example where a byte is protected by an error correction code of capability 1 complemented with data inversion with un-integrated protection. The first write attempt fails as a consequence of 3 SA-W cells. Afterwards, a second write is attempted with the codeword inverted and the polarity bit is set to 1. The write completes successfully since the inversion step could turn the SA-W cells into SA-R. At read time, firstly the codeword is retrieved inverted as the value of the polarity bit is 1. Secondly, the codeword is decoded to retrieve the data bits.

So far, our discussion of data inversion did not address two important points. The first is how to determine the successfulness of a write operation while the second is how to handle the failure of the second write attempt. Since PCM's major fault model is stuck-at faults resulting in hard errors, a standard practice is to apply a read-after-write (RAW) operation to verify that the data was written successfully [9]–[11]. Given the data dependent nature of errors, RAW divulges all the SA-W cells. Thus, the successfulness of the write operation is determined based on the number of SA-W cells determined by the RAW operation. Accordingly, data inversion requires one RAW operation after the first write attempted and an additional RAW operation in case a second write attempt is needed. The failure of the second write attempt is an indication of the defectiveness of the block. Consequently, such a block has to be retired and mapped-out of the address space. The execution flow of data inversion for a write request is captured in Fig. 4.

IV. Potential Overheads of Data Inversion

The flow of execution for data inversion reveals that no overhead is incurred before the number of faults accrued

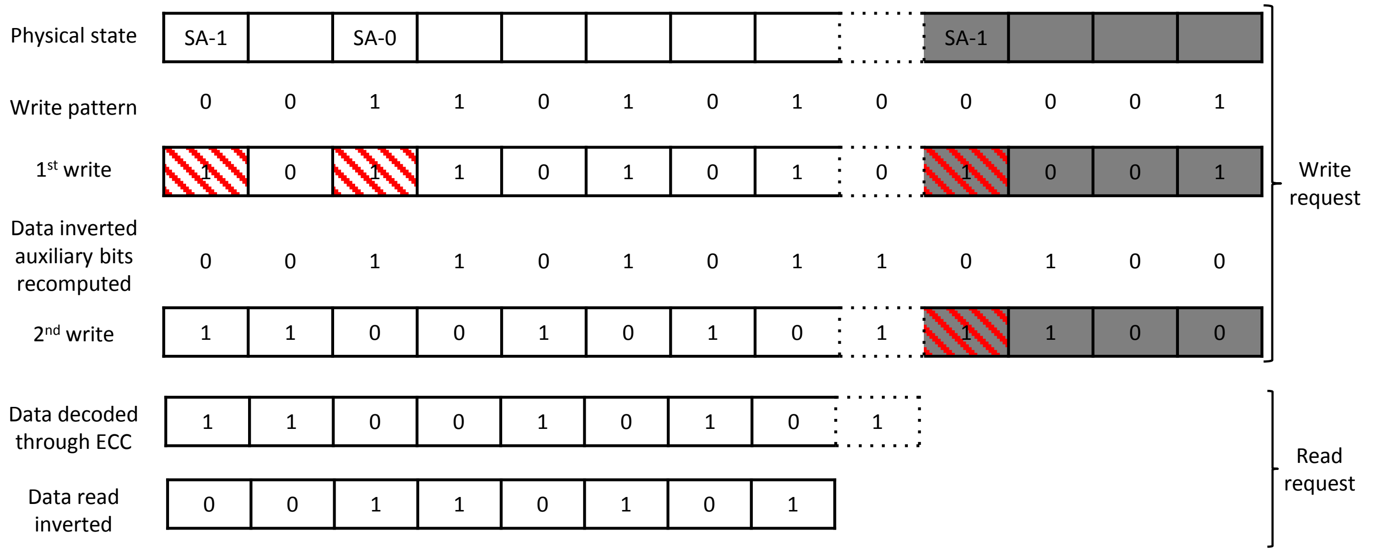

Fig. 2: An example of executing write and read requests with data inversion with integrated protection complementing an error correction code of capability 1. Dotted cells represent the polarity bit, gray cells represent the auxiliary bits and red hashed cells represent errors.

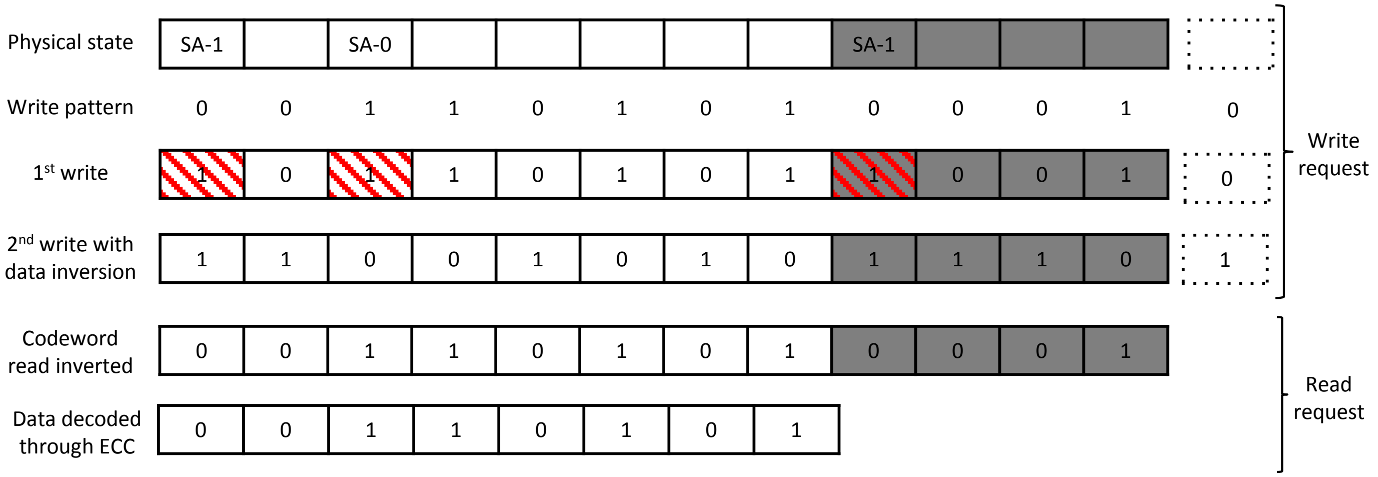

Fig. 3: An example of executing write and read requests with data inversion with un-integrated protection complementing an error correction code of capability 1. Dotted cells represent the polarity bit, gray cells represent the auxiliary bits and red hashed cells represent errors.

within a block exceeds the nominal capability of the deployed error correction scheme. Once the number of faults exceeds the nominal capability, an extra write operation may be required to successfully complete a write request. Nevertheless, the need for an extra write operation is rare due to the data dependent nature of failures. Consider a 4 KB storage block protected by a BCH code of capability 20. For a write operation to fail, two conditions must hold. The first condition is that the block must suffer from at least 21 stuck-faults. The second condition is that at least 21 of the stuck-at faults have to be SA-W after a write operation. Hence, the occurrence of write failure is probabilistic. In Section V-E, we show that the probability of failing to write on a defective block is notably low and quantify the overhead of extra writes.

In PCM, write data are typically buffered before being written on the memory cells in an iterative manner [15], [16]. Hence, the computational overhead of data inversion, when an extra write is required, is incurred while the data is buffered and is off the critical write path. Furthermore, applying inversion does not necessitate an inverter for each cell. Instead, a set of inverters on the bus transferring the data from the data buffer is enough. In fact, data inversion could borrow the same design to incorporate the inverters as in [7], [15].

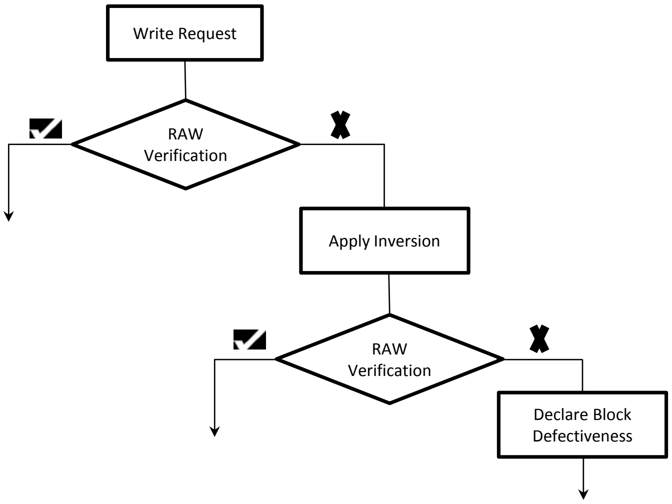

Fig. 4: Flow of executing a write request. “RAW” denotes read after write.

At read time, data inversion requires inverting the data if the polarity bit is set to 1. Consequently, the read path is augmented with a simple logic to retrieve the data inverted, when needed, that incurs marginal overhead. As a matter of fact, the PCM prototype in [17] has a relatively sparse pipeline stages that can easily incorporate the required logic.

To increase the capability of an error correction code, a substantial computational complexity has to be incurred. Complementing an error correction code with data inversion increase the number of faults that the code can tolerate by up to double the nominal capability with a simple inversion operation. In addition to higher computational complexity, increasing the capability of an error correction code requires a significant number of additional auxiliary bits. For example, a BCH code implemented over a Galois field of size 10 requires 60 auxiliary bits to protect against 6 errors. If the BCH capability is to be doubled, then 120 auxiliary bits are required. Nevertheless, data inversion can double the correction capability of the code with a single auxiliary bit i.e. the polarity bit. Thus, data inversion save $\sim$50% of the required auxiliary bits to double the correction capability of a code. Putting it altogether, data inversion increases the error correction capability without incurring a significant encoding/decoding complexity and an appreciable large number of additional auxiliary bits.

In summary, data inversion is a simple technique capable of increasing the number of faults that an error correction code can mask while incurring a minimal additional overhead imposed on read and write operations. Most importantly, data inversion is a post failure technique. Consequently, the additional overhead does not start to be incurred until write failures start to occur. That is, when the number of stuck-at faults within a memory/storage block exceeds the nominal capability of the deployed error correction code.

V. Evaluation

To assess the performance of data inversion, we look at the lifetime that can be achieved when an error correction code is complemented with data inversion in comparison to the lifetime that can be achieved with the same error correction code without data inversion. In addition, we quantify the overhead of the additional write operation that data inversion may require. At last, we study the effect of data inversion on the lifetime of memory devices and chips through changing the variability in cells' endurance. It is worth noting that a number of error correction schemes to mask stuck-at faults have been proposed [9]–[11]. Nevertheless, the roadmap to endorse those schemes in real system implementation is still unclear. Error correction codes, such as BCH, are an industry standard that have been thoroughly implemented and optimized in functional systems. Furthermore, unlike data inversion that is a post-failure technique, other proposed schemes are proactive techniques that try to delay the occurrence of write failures. To the best of our knowledge, data inversion is the first technique that targets extending the usability of memory blocks beyond the nominal capability of the error correction code deployed to recover from faults. Hence, the goal of this paper is not to compare with other existing schemes but to present a simple architectural technique capable of increasing the number of faults that error correction codes can cover.

A. Experimental Setup

To evaluate data inversion, we resort to Monte Carlo simulation. Since a detailed simulation of a large memory capacity is impractical within a reasonable time span, we lay out 2,000 pages of main memory and secondary storage each of size 4KB. For main memory system, we assume that each page is an aggregation of 512-bit cache line size blocks. To protected a cache line, we deploy a BCH code of capability 6 (BCH-6). BCH-6 requires an overhead of auxiliary bits that amounts to 11.7%, which is below the 12% generally accepted overhead. As for secondary storage system, we assume that each page is an aggregation of 512-byte sector size blocks. We protect each block with a BCH code that can tolerate up to 20 errors (BCH-20). Our choice of the error correction code capability follows the need to recover from more than 20 errors in a sector size block in NAND flash [18]. To generate write traffic, we have collected data from various real file types ranging from MPEG videos and JPEG images to PDF documents.

To assign lifetime to the memory cells of simulated blocks, we use a Gaussian distribution with a mean of 10^8 with a standard deviation of 25×10^6 [9]. We assume that an efficient wear leveling scheme distributes writes evenly across the available blocks so that we achieve comparable wear rate. We retire memory blocks after a write request fails to be written successfully as results of a number of errors that could not be masked by either the BCH code or the BCH code complemented with data inversion. In addition, we take into consideration the possibility for the polarity bit to wear out when data inversion with un-integrated protection is in use. That is, in the rare event of a polarity bit wearing out before its associated memory block becomes defective, we retire the

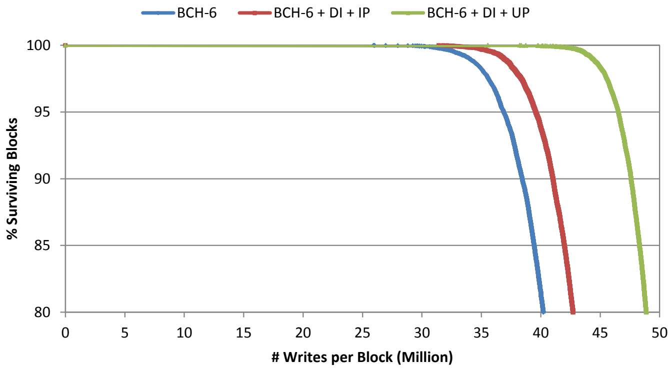

Fig. 5: Lifetime of PCM main memory blocks achieved with BCH-6 and BCH-6 plus data inversion (DI) with integrated protection (IP) and un-integrated protection (UP).

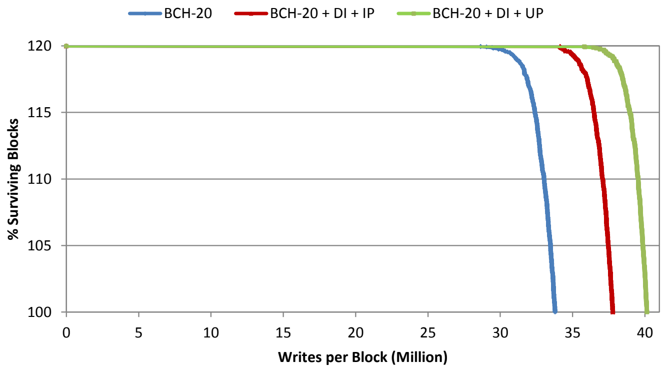

Fig. 6: Lifetime of PCM storage blocks achieved with BCH-20 and BCH-20 plus data inversion (DI) with integrated protection (IP) and un-integrated protection (UP). This experiment assumed that 20% of spare storage capacity was provided.

memory block. At last, our simulation methodology takes into consideration the possible need for extra writes operations and simulates them accurately. Overall, our methodology is similar to the one in [9].

Because the main failure mode of PCM is hard faults, we assume that the BCH capability is dedicated to masking hard faults. In case other failure modes, such as transient errors, become an issue in the future, a separate BCH capability must be provisioned to recover from them, which is orthogonal to this work.

B. Main Memory Lifetime Improvement

To study the effect of data inversion on the lifetime of main memory, we compare the lifetime achieved with data inversion complementing a BCH-6 code to that of the lifetime achieved by the BCH-6 code itself. Our evaluation metric is the total number of writes executed on memory blocks before retirement. When our scheme requires two physical write operations to complete a write request successfully as the first write could fail, we increment to the total write count by 1 only for fairness to the BCH-6 code. Since it is generally accepted for main memory to continue in operation even after retiring a number of memory blocks due to defectiveness, we study the lifetime until 20% of the memory blocks are retired. Fig. 5 shows our results, where the Y axis represents the percentage of surviving blocks as a function of the number of writes per block represented by the X axis.

It is clear from Fig. 5 that data inversion is capable of substantially improving the lifetime of memory chips. After retiring the first memory block, data inversion extends the lifetime over BCH-6 by 21.1% and 34.5% with integrated and un-integrated protection respectively. Furthermore, the shape of the three curves indicates that the achieved lifetime improvement gap is maintained until 20% of the memory blocks are retired.

It is worth noting that data inversion with un-integrated protection significantly passes data inversion with integrated protected in terms of achievable lifetime. This result is a direct consequence of the enhancements that un-integrated protection adds to data inversion. Un-integrated protection retires a memory block after 14 stuck-at faults are accrued. On the other hand, integrated protection could retire a memory block after 7 stuck-at faults are accrued depending on the distribution of the faults within the block.

Overall, protecting memory chips solely with a BCH code leads to an earlier retirement of the blocks. Conversely, deploying data inversion increases the number of faults that the BCH code can tolerate. Thus, data inversion extends the usability of memory blocks beyond a regular BCH code leading to significant lifetime improvements.

C. Secondary Storage Lifetime Improvement

To evaluate the impact of data inversion on storage devices, we note the lifetime in terms of the number of writes per block achieved with a BCH-20 code complemented with data inversion and a regular BCH-20 code. Conversely to main memory chips, block retirements that cause the degradation of the actual storage capacity are not allowed in storage devices. As a matter of fact, block defectiveness is combated through over-provisioned spare blocks. Consequently, we over-provision an additional 20% of the total storage capacity as spares which is a typical practice in server products. In this sequel, a storage device remains in operation until the defectiveness of the first sector that cannot be replaced with a spare. Fig 6 plots the lifetime of storage blocks while providing 20% worth of spares.

The results show that when the first storage block is retired, data inversion extends the lifetime over BCH-20 by 18.1% and 25.2% with integrated and un-integrated protection respectively. This achievement is a consequence of the delay in block defectiveness that data inversion provides through increasing the number of stuck-at faults that can form within a storage block before turning defective. Moreover, it is worth noting that the improvement in lifetime is maintained until the storage device is decommissioned.

Furthermore, it is notable that the difference in lifetime between integrated and un-integrated protection is smaller than that of main memory. As a matter of fact, having a BCH code of high capability and a bigger block size increase the likelihood of having a distribution of faults in between the data bits and the auxiliary bits that preserves the non-defectiveness

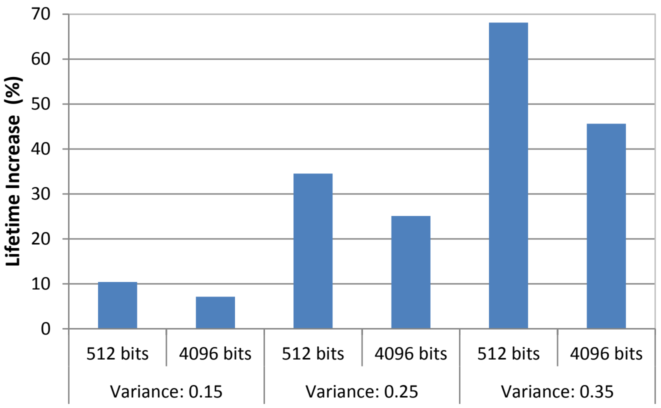

Fig. 7: Lifetime increase with data inversion relative to no inversion with various cell lifetime variance.

of a storage block. Hence, data inversion with integrated protection could perform better with storage devices.

Lastly, it is noticeable that the overall lifetime improvement for a storage device is less than that for a main memory chip. In fact, spares have smaller impact on the lifetime when data inversion complements a BCH code than with the regular code itself. Since data inversion increases the number of stuck-at faults that can be tolerated within storage blocks, it delays the retirement of the blocks. Nevertheless, the number of blocks nearing their lifetime limit increases. Consequently, spares are allocated at a higher rate once defective blocks start to occur. Our findings indicate that at the time the first spare block is allocated when data inversion complements the BCH code, 5% of the spares are allocated when the regular BCH code is used. Yet, the allocation rate of spares with data inversion increases from that point on.

D. Lifetime Variability

As PCM scales to small feature sizes, the manufacturing process is expected to exhibit a non-uniform distribution of cell lifetime. Accordingly, we assess the impact of data inversion on lifetime in light of the imperfect process control. To this end, we evaluate the lifetime of memory and storage blocks while varying the cell lifetime variance. That is, we fix the lifetime mean to 10^8 and set the cell lifetime variance to 0.15, 0.25 and 0.35 respectively. Our evaluation metric is the lifetime increase achieved through complementing a BCH code with data inversion relative to the regular BCH code after the retirement of the first memory block. To calculate this metric, we subtract the total number of writes performed with data inversion from the total number of writes performed by the regular code and divide the difference by the latter. Fig. 7 shows the increase in lifetime with data inversion with the un-integrated protection scheme. The integrated protection scheme showed similar trend and its results were omitted for brevity.

Fig. 7 reveals that data inversion can increase the lifetime across all lifetime variances. Yet, smaller lifetime increase is noted with smaller variability in cells' lifetime. In fact, if the cells within a memory block exhibit low lifetime variability, then a large number of cells are expected to wear out within close proximity. Accordingly, this domino effect makes data inversion less effective in extending the lifetime. Nevertheless, data inversion still manages to increase the lifetime by 10.4% and 7.1% for memory and storage blocks respectively with a variance of 0.15. On the other hand, increasing the number of faults that an error correction code can cover has a significant effect on the lifetime when the variability in cell lifetime is high. For example, a striking lifetime increase is marked with a cell lifetime variance of 0.35. As a matter of fact, a high variability in lifetime implies that cells within a memory block vary from low endurance cells to strong endurance cells. Consequently, increasing the number of faults that can be covered has a significant effect on lifetime as it allows to cover weak cells that fail early in the lifetime and keeps enough capability to cover cells that fails later. Hence, coupling an error correction with data inversion achieves a significant lifetime improvement with high variability in cell lifetime; a case which is common due to the imperfect process control with a very deep sub-micron technology.

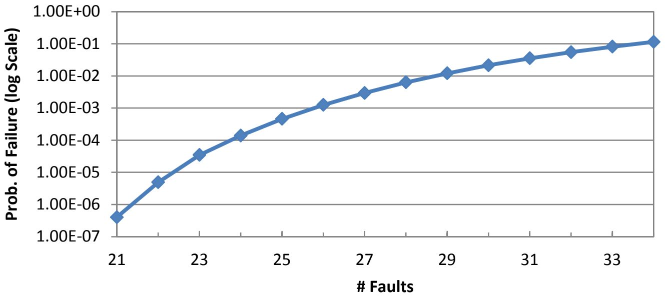

Fig. 8: Probability of failing to write on a 512-bytes storage block protected by a BCH-20 code.

E. Performance Overhead

When a write request fails to complete successfully, data inversion submits another write request after executing the inversion step. Nevertheless, data inversion does not interfere with the write process except when failures actually occur. That is, after the number of faults within a protected block exceeds the nominal capability of the error correction code. Hence, the potential overhead incurred through an extra write operation is not a constant overhead that every write request pays. Even after the number of faults exceeds the capability of the error correction code, an extra write operation is not always required. Consider a 512-byte storage block protected by a BCH code of capability 20. For a write to fail, the block must have at least 21 faults and at least 21 of the faults must happen to be SA-W. Fig. 8 reveals that the probability of failing to write is low even after the number of faults within the block significantly exceeds the capability of the BCH code due to the data dependent nature of errors. For example, with 34 faults the probability of failing to write is only 10%. Hence, writing on a block having a number of faults above the capability of the error correction code would still succeed with a high probability. It is noted that a cache line size block exhibit the same trend and its results are omitted for brevity.

	Data Inversion with Integrated Protection		Data Inversion with Un-Integrated Protection	
	Avg. % of extra writes before nominal capability is exceeded	Avg. % of extra writes after nominal capability is exceeded	Avg.% of extra writes before nominal capability is exceeded	Avg.% of extra writes after nominal capability is exceeded
512 bits	0%	4.9%	0%	13.1%
4096 bits	0%	6.4%	0%	8.9%

TABLE II: Performance evaluation in terms of extra write operations required by data inversion to complete write requests successfully after the number of faults exceeds the nominal capability of the error correction code.

To quantify the overhead of extra writes that could occur, we isolate their performance overhead. To this end, we count the number of extra writes that a block undergoes before it is retired. In Table II, we show the results where we consider a cache line size block and a storage sector size block protected by a BCH code of capability 6 and 20 respectively. We note that we present the results of averaging a million simulation runs each with a block of different cells's lifetime.

Our result shows that no performance overhead in terms of additional writes is incurred before the number of faults within a block exceeds the nominal capability of the error correction code. Thereafter, 4.9% and 6.4% of write requests require an extra write operation for a memory cache line and a storage sector respectively with the integrated protection scheme and 13.1% and 8.9% of write requests require an extra write operation for a memory cache line and a storage sector respectively with the un-integrated protection scheme. It is notable that the need for extra writes is still relatively low even after write failures could start occurring. Therefore, data inversion increases the number of faults that can be tolerated within blocks, salvages otherwise decommissioned blocks and notably augments the write volume of memory blocks while incurring an affordable overhead.

Lastly, an observation is worth noting. The extra overhead of the integrated protection scheme is less than that of the un-integrated scheme. As a matter of fact, our evaluation showed that the integrated protection scheme extends the lifetime of memory blocks less than the un-integrated protection scheme. Consequently, the number of additional successful writes achieved with the integrated protection scheme is less than that of the un-integrated scheme when both schemes are compared against a regular code. Hence, the un-integrated protection scheme incurs more extra writes than the un-integrated protection scheme.

VI. Related Work

Data inversion is a new, simple and innovative technique to increase the number of faults that an error correction code can tolerate. Nevertheless, inverting the write data has appeared in the literature in different contexts. Hence, we present a summary of related work and highlight the differences and similarities to data inversion.

Cho and Lee [7] propose *Flip-N-Write* to invert data if the inversion step leads to a reduction in the number of bits that needs to be physically programmed. When write request is submitted, Flip-N-Write reads the old data written on the physical block. Subsequently, the read data is compared against the new data in its inverted and un-inverted forms. Next, the form that incurs the least number of bit flip is picked as the data that is physically written on the PCM medium. Similarly to data inversion, Flip-N-Write introduces an additional bit that serves as an inversion flag. However, Flip-N-Write does not provide a variation in which the flag bit is protected. It is worth mentioning that Flip-N-Write and data inversion try to achieve two different tasks. The first suppresses unnecessary bit flips in an attempt to increase write bandwidth under write current constraints, while the second attempts to increase the number of stuck-at faults that can be tolerated before a block turns defective. Moreover, Flip-N-Write interferes with every write operation while data inversion starts to interfere in the write process only after the number of faults exceeds the nominal capability of the error correction code and the first write failure occurs. Hence, Flip-N-Write is orthogonal to data inversion. Yet, we would like to note that data inversion is compatible with Flip-N-Write. Since Flip-N-Write's main concern is to reduce the number of bit flips, it does not take into consideration the data dependent nature of errors for stuck-at faults. For example, inverting a given data pattern to reduce bit flips may lead to a number of stuck-at wrong cells that is above the correction capability of the error correction code whereas un-inverted data may not. In this sequel, data inversion reverts the decision taken by Flip-N-Write and completes an otherwise failed write request successfully.

RDIS [11] is an error correction scheme to tolerate stuck-at faults in resistive memories. RDIS identifies a set containing all the stuck-at wrong cells. This set is called the "invertible set" as it needs to be inverted at read time in order to correctly retrieve the stored data. The process of constructing the invertible set goes through several steps that requires inverting the bit values of certain specific cells. Hence, RDIS uses the inversion step as a means to construct the invertible set.

Another scheme that involves a potential inversion of data is *SAFER* [10]. SAFER is an error correction scheme that partitions the bits in a data block into several groups while ensuring that each group contains at most one stuck-at fault. In the event that a stuck-at cell in a group happens to be stuck-at wrong after a write operation, SAFER inverts the data bits that pertains to that group. Similarly to RDIS, the inversion step in SAFER can be viewed as a tool to help encode the data that needs to be written. Stated differently, the inversion step is not an stand-alone technique as in data inversion.

ECP [9] is an error correction scheme that recovers from stuck-at faults through providing a set of spare cells to replace defective cells. To keep track of the position of defective cells, ECP equips each spare cell with a pointer entry that identifies the replaced cell. Once a defective cell is discovered, ECP permanently allocates a correction entry to replace it. Data inversion can be coupled with ECP provided that the allocation of the correction entries changes into a dynamic one. That is, the correction entries are allocated to mask off stuck-at wrong cells that are manifested after the execution of a write operation. Allocating the correction entries dynamically while complementing ECP with data inversion, allows ECP to double the number of faults that can be tolerated.

Lastly, it is worth noting an important difference between data inversion and all other schemes mentioned above. That is, data inversion is a post failure technique while other schemes are proactive techniques. Particularly, data inversion kicks in after write failure occurs while write failure is the halting criterion of other schemes. Moreover, the above mentioned error correction schemes have assumptions of fault-free auxiliary bits. This assumption does not hold for error correction codes such as BCH, but those codes are criticized by their complexity when the number of stuck-at faults that needs to be masked is large. To this end, data inversion is a simple architectural technique that enables error correction codes to increase their capabilities with a minimal overhead. Hence, data inversion paves the way to deploy error correction codes in PCM chips and devices and benefit from their resilient protection capability.

VII. Summary

Phase-change memory (PCM) is an emerging memory technology capable of overcoming the physical challenges faced by DRAM and NAND flash technologies. In order to make PCM a viable memory technology for high volume manufacturing, combating the write endurance problem is essential. Consequently, efficient multi-bit error correction schemes are required. This paper presented *data inversion*, an architectural technique capable of allowing error correction codes such as BCH to tolerate a number of faults greater than their nominal capability. Through exploiting the data dependent nature of errors for stuck-at faults, data inversion inverts the write pattern after a write request fails to complete successfully. Consequently, the inversion step is likely to bring the number of erroneous cells within the capability of the error correction code. Thus, data inversion completes successfully a write request that would otherwise fail and extends the usability of memory blocks.

In this paper, we made the following key contributions:

1) We proved that data inversion is capable of increasing the number of faults that an error correction code can tolerate with a memory block while requiring a single additional polarity bit. In addition, we presented two variations of data inversion. The first protects the polarity bit and could increase the capability of the error correction code based on the distribution of faults within the protected memory block. The second doubles the capability of the error correction code through separating the polarity bit from the protection scope of the code.
2) We showed that data inversion can significantly increase the lifetime of PCM while incurring an affordable performance overhead. Our evaluation shows that the overhead could be incurred only after the number of faults within a block exceeds the nominal capability of the deployed error correction code.
3) We studied the effect of data inversion on lifetime in light of imperfect manufacturing process control. We show that data inversion copes with the variability in cells' endurance and is especially effective when the variability is high, which is expected to be a common manufacturing phenomenon.

In conclusion, data inversion is an architectural technique that effectively addresses the critical endurance reliability issue that PCM suffers from. Data inversion is powerful, yet simple enough to be readily incorporated into computer systems under realistic usage scenarios.

References

[1] ITRS, http://public.itrs.net, 2011.

[2] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *ISCA*, 2009.

[3] J.-Y. Jung, K. Ireland, J. Ouyang, B. Childers, S. Cho, R. Melhem, D. Mosse, J. Yang, Y. Zhang, and A. Camber, "Characterizing a real pcm storage device," in *NVMW*, 2011.

[4] A. Akel, A. M. Caulfield, T. I. Mollov, R. K. Gupta, and S. Swanson, "Onyx: a protoype phase change memory storage array," in *HotStorage*, 2011.

[5] Numonyx, Inc. (Micron Technology, Inc.), "Numonyx phase change memory (p8p)," www.micron.com, 2009.

[6] 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," in *ISCA*, 2010.

[7] S. Cho and H. Lee, "Flip-n-write: a simple deterministic technique to improve pram write performance, energy and endurance," in *MICRO*, 2009.

[8] D. K. Bose, R. C.; Ray-Chaudhuri, "On A Class of Error Correcting Binary Group Codes," *Information and Control*, vol. 3, no. 3, pp. 68–79, march 1960.

[9] S. Schechter, G. H. Loh, K. Straus, and D. Burger, "Use ecp, not ecc, for hard failures in resistive memories," in *ISCA*, 2010.

[10] N. H. Seong, D. H. Woo, V. Srinivasan, J. Rivers, and H.-H. Lee, "Safer: Stuck-at-fault error recovery for memories," in *MICRO*, 2010.

[11] R. Melhem, R. Maddah, and S. Cho, "Rdis: A recursively defined invertible set scheme to tolerate multiple stuck-at faults in resistive memory," in *DSN*, 2012.

[12] E. Ipek, J. Condit, E. B. Nightingale, D. Burger, and T. Moscibroda, "Dynamically replicated memory: building reliable systems from nanoscale resistive memories," in *ASPLOS*, 2010.

[13] R. Hamming, "Error Detecting and Error Correcting Codes," *Bell System Technical Journal*, vol. 26, 1950.

[14] D. Strukov, "The area and latency tradeoffs of binary bit-parallel BCH decoders for prospective nanoelectronic memories," in *ACSSC*, nov. 2006.

[15] H. Chung *et al.*, "A 58nm 1.8V 1Gb PRAM with 6.4MB/s Program BW," in *IEEE ISSCC*, February 2011.

[16] JEDEC, "Jedec publishes lpddr-nvm memory standard," www.jedec.org, 2009.

[17] K.-J. Lee *et al.*, "A 90 nm 1.8 V 512 Mb Diode-Switch PRAM With 266 MB/s Read Throughput," *IEEE JSSC*, vol. 43, January 2008.

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