ECC Replay: A Practical and Efficient Scheme To Tolerate High Stuck Bit Rate in Emerging Memories

Lunkai Zhang
lunkai.zhang@wdc.com
Western Digital
San Jose, California, USA

Nate Franklin
Nate.Franklin@wdc.com
Western Digital
San Jose, California, USA

ABSTRACT

While emerging memories are considered promising technologies to replace DRAM, it has been proven difficult to penetrate the mature DRAM market and achieve profitability. Therefore, it is crucial to reduce the cost of the emerging memory media. One important way to reduce the cost is to make the memory system more fault tolerant, thus the production process can be simplified and media yield can be improved. This paper focuses on stuck failures, which are dominant in endurance failures and manufacturing failures. We present *ECC Replay*, an efficient and practical scheme which handles the stuck failures by utilizing existing ECC techniques in a typical emerging memory system. When a codeword read results in an uncorrectable, *ECC Replay* detects the stuck bits of the codeword by writing all-0/all-1 data into the media and then reading it back. *ECC Replay* then tries to get a correctable result by reperforming ECC correction with all the possible data combinations of the stuck bits. We also propose special techniques to minimize the *Miscorrection Rate*, which is often the reliability bottleneck of memory systems, and read tail latency. With minimal change in the memory controller architecture and no change to the emerging memory media, *ECC Replay* achieves 30× improvement of stuck-bit tolerance in a typical emerging memory system with negligible impact on memory bandwidth and read latency.

1 INTRODUCTION

In recent years, Large Scale Language Models (LLM) has been proven to be a transformative technology to reshape computing [3]. Companies, big and small, are all racing to implement and utilize their own LLMs. The inherent memory capacity requirement of LLMs fuels the growth of the memory component market, which is dominated by DRAM [1]. The hunger for memory capacity, however, is hampered by another trend–the scaling of DRAM technology is slowing and may reach its limit [32]. Which means it is getting increasingly harder to reduce the per-GB cost of DRAM.

As a result, the market is calling for a cheaper DRAM replacement technology with better scalability. Candidates for such a replacement, sometimes called *Emerging Memories* or *Storage Class Memories*, include PCM [7][19], ReRAM [34][5] and MRAM [10][15], etc. and they are often combined with the crossbar array technology [18] to reduce the cell size. However, it takes great efforts, both in time and capital spending, to develop and market a memory technology to replace DRAM. Unfortunately, the only commercialized emerging memory technology, Optane, has been discontinued partially due to its inability to turn a profit [2] in years. The sunset of Optane serves as a cautionary tale for its potential sucessors that it is crucial to manage the cost of emerging memories in order to compete with DRAM and achieve commercial success.

One important way to reduce cost of emerging memories is to use error correction code (ECC) to tolerate bit errors and thus reduce media manufacturing complexity as well as improve yield. We will show that, a reasonable baseline BCH-based ECC setup can tolerate up-to 1E-5 scale bit error rate (*BER*). For comparison, MLC (including TLC and QLC) 3-D NAND Flash, the only mature memory technology scaling better and cheaper than DRAM, can tolerate a *BER* of up to 1E-2 scale [4]. However, the high *BER* tolerance of MLC 3-D NAND Flash is achieved by using large codewords and complex ECC mechanism such as LDPC, both of which incur long ECC latency and thus are not suitable for latency sensitive DRAM replacement technologies.

To achieve high error tolerance in emerging memories without exccessive latency or high system complexity, it is crucial to understand the system implications of different types of media failures. Among many failures of emerging memories, we are especially interested in stuck failures in which the value of stored bit is stuck at 0 or 1, because they are the major failure types of endurance and manufacturing failures [27][36][9][26]. There have been a number of proposals targeting stuck failures in emerging memories. However, to our best knowledge, none of these proposals is effective in a practical product setting. Especially, none of them discussed their impact on *Miscorrection Rate*, which is often the reliability bottleneck of the emerging memory systems.

This paper presents *ECC Replay*, an efficient and practical scheme which handles the stuck failures by utilizing existing ECC mechanisms in a typical emerging memory system. When a codeword read results in an uncorrectable, *ECC Replay* first detects the stuck bits of the codeword by writing all-0/all-1 data into the same media codeword and then reading it back. *ECC Replay* then tries to achieve a correctable result by reperforming ECC correction with all the possible data combinations of the stuck bits. We also introduce measures to minimize *Miscorrection Rate* as well as avoid extra tail latency. Our evaluation shows that, by utilizing a typical ECC mechanism (BCH-6) with only 1E-5 scale *Bit Error Rate* (BER) capability, *ECC Replay* achieves enterprise-scale UBER (*Uncorrectable Bit Error Rate*) and *Miscorrection Rate* requirements on an emerging memory media with a much higher 3E-4 *Stuck Bit Rate*, which is a 30× improvement over the BCH-6 baseline. Furthermore, *ECC Replay* achieves such aggressive tolerance of stuck bits with negligible performance degradation and no extra storage overhead.

The rest of the paper is presented as follows. We first introduce the background of emerging memory in Section 2. Then Section 3 models the reliability of our baseline systems and provides the problem statement. Section 4 presents the schemes of our *ECC Replay* proposal and models their impact on system reliability. Section

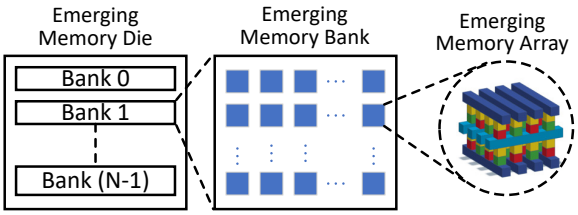

Figure 1: The Structure of a Typical Emerging Memory Die [11]. The die consists of multiple banks and each bank has many crossbar arrays. For a single codeword access, each bit is contributed by different crossbar arrays. Such bit access isolation indicates that it is unlikely to have clustered bit fails. As a result, it is fair to model the bit fails using binomial distribution.

5 evaluates the performance impact of *ECC Replay* with a cycle-accurate simulator. Section 6 discusses the effectiveness of *ECC Replay* under different error correction capabilities. Finally, Section 7 discusses the related work and Section 8 draws the conclusion.

2 BACKGROUND OF EMERGING MEMORY SYSTEMS

This section first discusses a typical emerging memory die structure and its implication on bit fail characteristics. We then discuss the soft and stuck failures of emerging memories, and introduce two types of reliability failures: *uncorrectables* and *miscorrections*. Finally we introduce the fundanmentals of emerging system reliability including Bit Error Rate (BER), Uncorrectable Bit Error Rate (UBER) and Miscorrection Rate (MISC Rate).

2.1 Emerging Memory Die Structure and Its Implication on Non-Clustering Bit Fails

Figure 1 shows the structure of a typical emerging memory die[11], which has multiple banks, similar with DRAM. The structure of an emerging memory bank, however, is fundanmentally different from that of a DRAM:

- A DRAM Bank contains a *single* data array. Each codeword/memory line access reads/writes the same row of the array. In case of the row failure, all the bits retrieved by a DRAM die during the codeword read are subject to errors. As a result, clustering bit fails are serious concerns in DRAM and thus countermeasures like chipkill are required for high reliability use cases.
- An emerging memory bank [11] contains many independent crossbar arrays (sometimes referred as tiles). In a codeword access, each codeword bit is contributed by different crossbar arrays. Such bit access isolation indicates that it is unlikely to have clustered bit fails. Thus it is fair to assume all bit failures are independent and thus the bit failure is following binomial distribution.

2.2 Stuck Bit Errors and Soft Bit Errors

We can categorize all the failure mechanisms in emerging memories into two groups:

- **Soft Bit Errors**, which refer to the failures that can be resolved by reread or rewrite the media codeword. The failure mechanisms of soft errors include retention failure, read disturb, write disturb, etc.
- **Stuck Bit Errors**, which refer to the failures in which the value of stored bit is stuck at value 0 or 1. The failure mechanisms of stuck bit errors include endurance failures and manufactural failures.

In this paper we focus on *Stuck Bit Errors* because: (1) endurance failure is usually the major failuretype in end-of-life media; (2) endurance failure bits can provide extra information for recovering, which provides opportunities for more efficient failure recovery schemes such as ECC Replay.

2.3 Uncorrectables and Miscorrections

Any practical ECC scheme has its capability limit and is subject to two types of ECC failures:

- **Uncorrectables.** If the decoder cannot provide a valid ECC outcome, there is an uncorrectable.
- **Miscorrections.** If the decoder provides a valid ECC outcome but it is different from the original data, there is a miscorrection. *Miscorrections* are also referred as *silent data corruption* (SDC) [16].

Both uncorrectables and miscorrections are serious reliability issues for the memory/storage systems. As is shown in Section 3, there are established metrics (UBER and MISC Rate) to measure their impact on system reliability.

3 FAILURE MODELING AND PROBLEM STATEMENT

In this section, we will model in detail of the reliability metrics of memory system with both hard errors and soft errors, and then present the problem statement.

3.1 Reliability Metrics of Memory System

In memory industry, There are three established reliability metrics:

- Bit Error Rate (*BER*), which is the total bit errors divided by the total number of bits read. *BER* is the direct metric of how erroneous the media is.
- Uncorrectable Bit Error Rate (*UBER*), which is the total *uncorrectable* codewords divided by the total number of bits read (*not the total number of codewords*). *UBER* shows how vulnerable the memory system is to *uncorrectables*.
- Miscorrection Rate (*MISC Rate*) [24], which is the total miscorrected codewords divided by the total number of codewords read. *MISC Rate* shows how vulnerable the memory system is to *miscorrections*.

3.2 BCH ECC for Emerging Memories

Since emerging memory media has higher BER, it is logical to protect it through strong ECC mechanism such as LDPC [29], Reed-Solomon [35] or BCH [13]. Among them, BCH code has received additional attention because of its storage effectiveness on short codewords and potential to construct low latency decoding scheme [12]. Table 1 summarizes the capabilities and overheads of different

Table 1: ECC Capacity Overhead and Correctable Bits of Different BCH Schemes for 64B Codewords

ECC Schemes	[n, k, d] Representation	User Data Bits	ECC Bits	ECC Overhead	Correctable Bits
BCH-4	[552,512,9]	512	40	7.8%	4
BCH-6	[572,512,13]	512	60	11.7%	6
BCH-8	[592,512,17]	512	80	15.6%	8
BCH-10	[612,512,21]	512	100	19.5%	10
BCH-12	[632,512,25]	512	120	23.4%	12
BCH-14	[652,512,29]	512	140	27.3%	14
BCH-16	[672,512,33]	512	160	31.3%	16

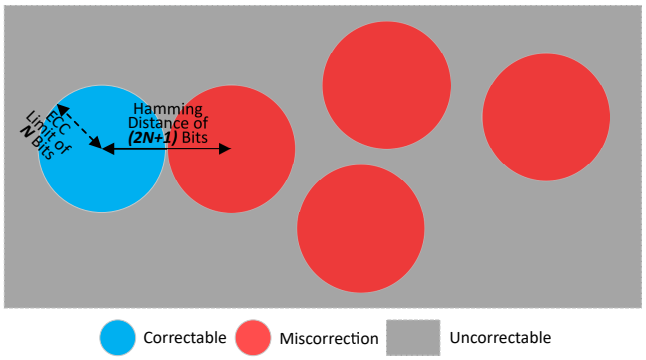

Figure 2: Correctables, Uncorrectables and Miscorrections in BCH

BCH schemes [33]. This table also includes the [n, k, d] representations of each BCH scheme, in which:

- n is the full length of a codeword;
- k is the number of data/information bits of a codeword;
- d is the minimum Hamming distance between any two valid codewords.

Without loss of generality, we assume our baseline system uses a BCH-6 ECC which can corrects up to 6 bits and has an overhead of 11.7%.

3.3 Basic BCH Failure Modeling with a Unified BER

The first step is to model the BCH scheme with a unified BER following binomial distribution. Figure 2 shows where *correctables*, *uncorrectables* and *miscorrections* are located in BCH code space: assume we have a BCH code with *ECC Limit* of *N*, the nearest different codeword has a hamming distance of *(2N+1)*. If the number of bit errors is within *N*, the codeword is always *correctable*; otherwise the codeword is mostly *uncorrectable* with a small chance of falling into the correctable space of another codeword, which causes *miscorrection*. In BCH code with long enough codeword size and large enough *ECC Limit*, the miscorrection probability in uncorrectable scenarios can be roughly considered as a constant [24](i.e., *BCH_Misc_Prob*). Section 9.1 provides the details of how to estimate *BCH_Misc_Prob*. The detailed formula of *UNC Rate* (uncorrectable rate), *UBER* and *MISC Rate* of BCH are shown in Equations 1–3.

$$UNC\_Rate = \sum_{errors=ecc\_limit+1}^{cw\_size} binom.pmf(errors, cw\_size, BER) \quad (1)$$

$$UBER = \frac{UNC\_Rate}{cw\_size} \quad (2)$$

$$Misc\_Rate = UNC\_Rate \times BCH\_Misc\_Prob(ecc\_limit, cw\_size) \quad (3)$$

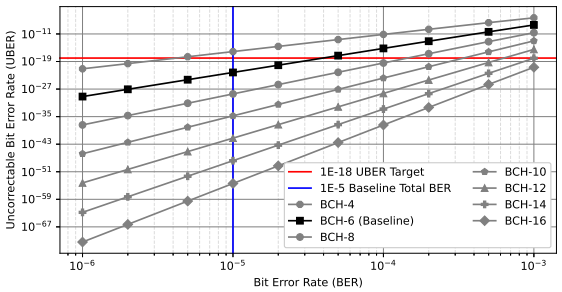

Figure 3: UBERs of Different BCH Codes with Different BERs

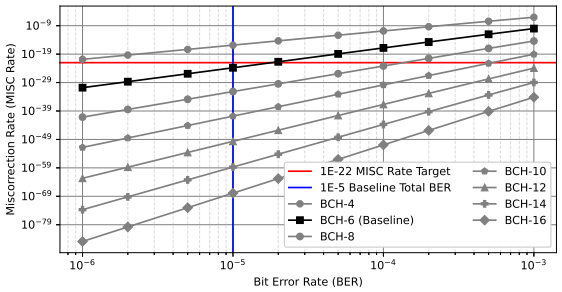

Figure 4: MISC Rates of Different BCH Codes with Different BERs

Figures 3 and 4 show *UBERs* and *MISC Rates* of different BCH codes with different Unified BERs. Without loss of generality, we assume a baseline system using *BCH-6* and with a *UBER* target of 1E-18 and a MISC Rate Target of 1E-22. We can see that, BCH-6 can meet the *UBER* target with up to 3E-5 Unified BER; at the same time, it misses the *MISC Rate* target with 2E-5 Unified BER. **This indicates *MISC Rate* is in many cases the reliability bottleneck of a memory system, not *UBER*. Therefore, for bit error tolerance related proposals, it is of great importance to investigate their impact on *MISC Rate*.**

3.4 Failure Modeling with BCH Schemes with Separate Considerations of Soft Bit Errors and Stuck Bits

We now do a more detailed modeling with separate considerations of soft bit errors and stuck bits. Figure 5 summaries our approach:

- We assume the stuck bits are randomly distributed in the whole memory and the addresses of accesses are random. Under such assumptions, the number of stuck bits in a read codeword is following binomial distribution with *Stuck Bit Rate*.
- A stuck bit error happens only when a stuck bit is written with the opposite data (e.g., Stuck-At-1 bit written with 0). We define such probability as *S2E Rate* (Stuck-to-Error Rate). As a result, the number of stuck errors of a given number of stuck bits is also following the binomial distribution. Without loss of generality, we assume a 50% *S2E Rate* for the rest of the paper.

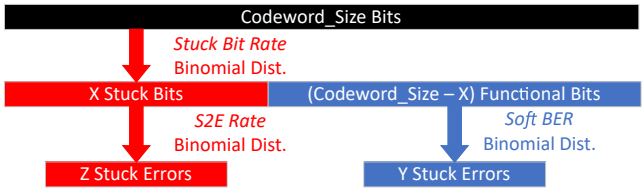

Figure 5: Failure Modeling of Stuck Bit Errors and Soft Bit Errors

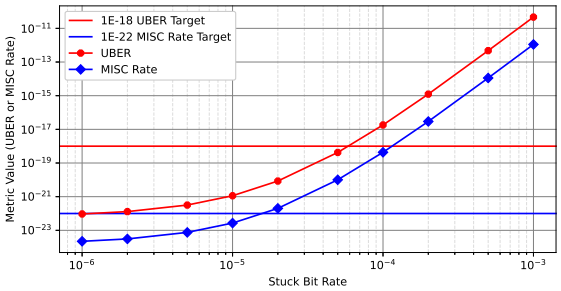

Figure 6: UBER and MISC Rate of Baseline BCH-6 with 1E-5 Soft BER and Different Stuck Bit Rate

- For the functional (not stuck) bits, they can have soft errors following binomial distribution with *Soft BER*.

Figure 6 shows *UBER* and *MISC Rate* of the baseline BCH-6 with 1E-5 *Soft BER* and different *Stuck Bit Rate*. For more modeling details, please refer to Section 9.2. The *MISC Rate* becomes out-of-target when *Stuck Bit Rate* reaches 2E-5; while UBER becomes out-of-target when *Stuck Bit Rate* reaches 6E-5.

3.5 Problem Statement

We can see that the baseline BCH-6 provides insufficient stuck bit tolerance. A straightforward solution would be increase the ECC capability, which is costly in terms of both media storage overhead and ECC latency/complexity. Therefore, a more efficient approach is desired to tackle high *Stuck Bit Rate*.

4 ECC REPLAY

In this Section, we first propose the ECC hierarchy and discuss the requirements for each ECC layer in Section 4.1. In Section 4.2, we then discuss how to address the missed *MISC Rate* target for *Normal Read* ECC layer. Afterwards, we present the detailed architecture, scheme and modeling of *ECC Replay* layer in Section 4.3. Section 4.4 shows *ECC Replay* can be further enhanced by caching the many-stuck codewords. Finally, Section 4.5 summarizes the reliability capability of *ECC Replay*.

4.1 Proposed ECC Hierarchy and Requirements

Figure 4.1 shows the overview of our proposed Normal Read—ECC Replay hierarchy. For each read request, *ECC Replay* gets triggered only when the initial media read (i.e., *Normal Read*) fails. To meet the system UBER and MISC Rate targets, such a hierarchy has different requirements for each layer:

- **Normal Read Layer.** For *Normal Read Layer*, *UBER* doesn't matter anymore because it will be resolved in the next layer

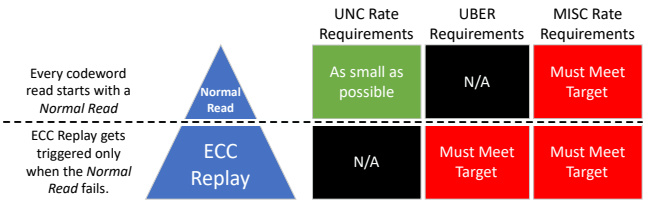

Figure 7: Our Hierarchical Proposal and Its Requirements on UNC Rate, UBER and MISC Rate.

(i.e., *ECC Replay*). However, it must meet the *MISC Rate* target to ensure we meet the system-wise *MISC Rate* target. Also the *UNC Rate* (i.e., *ECC Replay* trigger rate) of this layer should be as small as possible to minimize the performance impact of *ECC Replay*.

- **ECC Replay Layer.** As the last resort in ECC hierarchy, *ECC Replay* must meet both *UBER* and *MISC Rate* targets.

4.2 Undercorrecting Normal Reads to Reduce MISC Rate

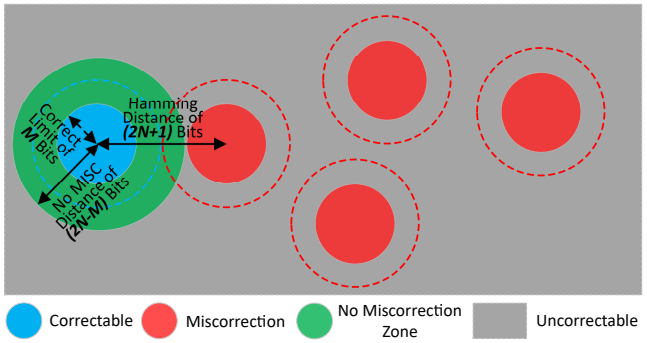

Figure 8: Undercorrection Helps Reduce the Probability of Miscorrection.

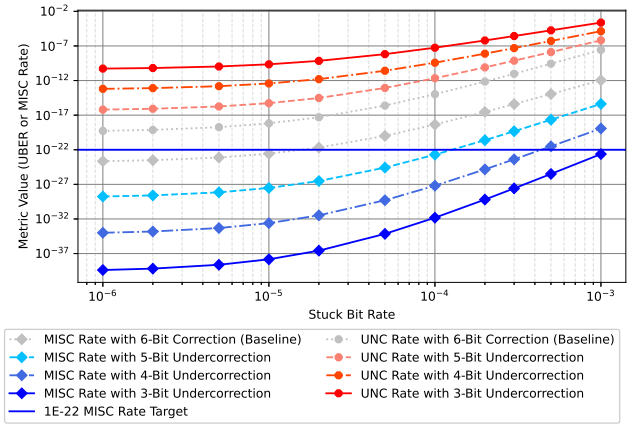

Figure 9: Undercorrection Helps Meet the Miscorrection Rate Target.

As is shown in Figure 6, baseline *Normal Read* doesn't meet the *MISC Rate* target with 1E-5 *Soft BER* and a *Stuck Bit Rate* equal or higher than 2E-5. To tolerate a higher *Stuck Bit Rates*, we need to first bring down the *MISC Rate* of *Normal Read* layer. To achieve this, we propose simple *Undercorrection* scheme–it has a *Correct Limit* which is smaller than the full BCH *ECC Limit*. Using *Undercorrection*, we consider a codeword *uncorrectable* if its error bits are more than *Correct Limit*, even if it is correctable by the full *ECC Limit*.

Figure 8 explains two effects of *Undercorrection* helps reduce the *MISC Rate* (see Section 9.3 for details):

- **Smaller Miscorrectable Probability for Uncorrectables.** This is because *Undercorrection* reduces the size of *correctable space* of each codeword. As a result, the probability of falling into *correctable space* of other codewords is reduced.
- **Expansion of No-MISC Space.** As we recall, the minimal Hamming distance between two codewords is *2N+1*, where N is the *ECC Limit* of corresponding BCH code. For baseline BCH scheme, the No-MISC limit is N bits, same with the limit of correctables. With *Undercorrection*, however, the No-MISC limit becomes (2N-M), where M is the *Correct Limit*.

Figure 9 shows the effectiveness of *Undercorrection* for our baseline BCH-6 ECC scheme. We can see that, when we undercorrect, the *MISC Rate* is significantly reduced at the expense of higher *UNC Rate*. Even with an aggressive *Stuck Bit Rate* target of 1E-3, the 1E-22 *MISC Rate* can still be met when we *undercorrecting* to 3 bits. Meanwhile, the *UNC Rate* of this scenario is 2E-4, which is still very low.

4.3 The Architecture and Scheme of ECC Replay

Figure 10 shows the architecture of *ECC Replayer*, the main module to perform *ECC Replay* and its position inside a typical memory controller. *ECC Replayer* is a standalone module which talks to the *Processing Queue(s)* of the memory controller. It has the following components:

- A *Raw UNC Data* register to record the raw uncorrectable data of *Normal Read*;
- A *Stuck Bit Mask* register to record the stuck bit positions inside the uncorrectable codeword;
- A *Replay Combination Generator* to generate *Stuck Bit Combinations* based on *Stuck Bit Mask*;
- A *Current Stuck Bit Replay Combination* register to record the current *Stuck Bit Combination*.
- A *Current ECC Replay Data* to record the current ECC replay data to go through the in-module ECC;
- A dedicated *ECC Decoder* to perform ECC decoding of the *ECC Replay Data*;
- A *Correctable Result Buffer* to interpret the results of *ECC Replay*.

Figure 11 shows the workflow of *ECC Replay*.

(1) When a *Normal Read* returns uncorrectable, *ECC Replayer* first records the raw uncorrectable data in *Raw UNC Data* register.
(2) **Stuck Bits Detection.** For a given emerging memory media, the endurance failures are manifested as either open or short, as a result the stuck bits are overwhelming biased in either 1 or 0. Without loss of generality, we assume the stuck bits are stuck-at-1. *ECC Replayer* writes the all-0 pattern to the media codeword and reads it back, bypassing main ECC Decoder. The value-1 bits indicate the locations of the stuck bits. Such information is stored into the *Stuck Bit Mask* register. To reduce the overall ECC Replay latency, these stuck detection write and read commands have the highest priority. In case there are both stuck-at-1 and stuck-at-0 bits, separate *Stuck Bit Detect Read and Write* are needed for each type of the stuck bits.
(3) **Stuck Combination Replay.** *Replay Combination Generator* takes the value of *Stuck Bit Mask* register, and generates all the possible data combinations of the stuck bits. For each *Replay Combination*, *ECC Replayer* replaces the stuck bits with the value of the *Replay Combination* to form the current *ECC Replay Data* and decode with the dedicated *ECC Decoder*.
(4) **Result Interpretation.** As is shown in Figure 12, after all the *Replay Combinations* get processed through ECC, *ECC Replayer* summarizes and reports the results:
    - **Case #1:** One or multiple *ECC Replay Data* are correctable, and the ECC outcomes are the same. In this case, *ECC Replayer* trusts the result and report the ECC outcome as *correctable*.
    - **Case #2:** Multiple *ECC Replay Data* are correctable, and there are more than one ECC outcomes. In this case, *ECC Replayer* doesn't know which ECC outcome is correct. To avoid miscorrection, it reports this case as *uncorrectable*.
    - **Case #3:** If none of the *ECC Replay Data* are correctable, *ECC Replayer* returns *uncorrectable*.

Table 2: Best ECC Replayed Stuck Combination is correctable if the ECC decoder can correct the soft bit errors.

	Stuck Bits	Soft Bit Errors
Observed Uncorrectable Data	111..11	010...01
Best ECC Replayed Stuck Comb With No Error In Stuck Bits	010..01 (no error)	010...01

Table 2 explains why ECC Replay manages to lower both *Uncorrectable Rate*/*UBER* and *MISC Rate*—out all the stuck bit combinations replayed, there must be a best combination which has no errors in the stuck bits. For this best combination, we can get a correctable if the number of soft bit errors is within *ECC Limit*. As a result, in cases where number of *soft bit errors* is within *ECC Limit* (refer to more details in Section 9.4):

- ECC Replay removes all the *uncorrectables* of *ECC Replay Data* since we already have a *correctable* result;
- ECC Replay captures all the *miscorrections* of *ECC Replay Data* and translates them into *uncorrectables*.

As is shown in Figure 13, with the help of enforced uncorrectable (Case #2), *ECC Replay* keeps the MISC Rate at the same level when we sweep *Stuck Bit Rate* from 1E-5 to 1E-3. Due to the limit of 1E-18

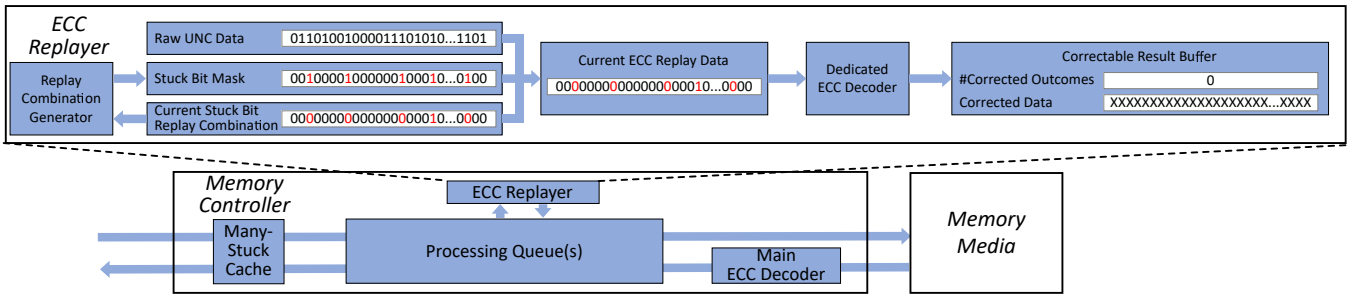

Figure 10: The Architecture of ECC Replay.

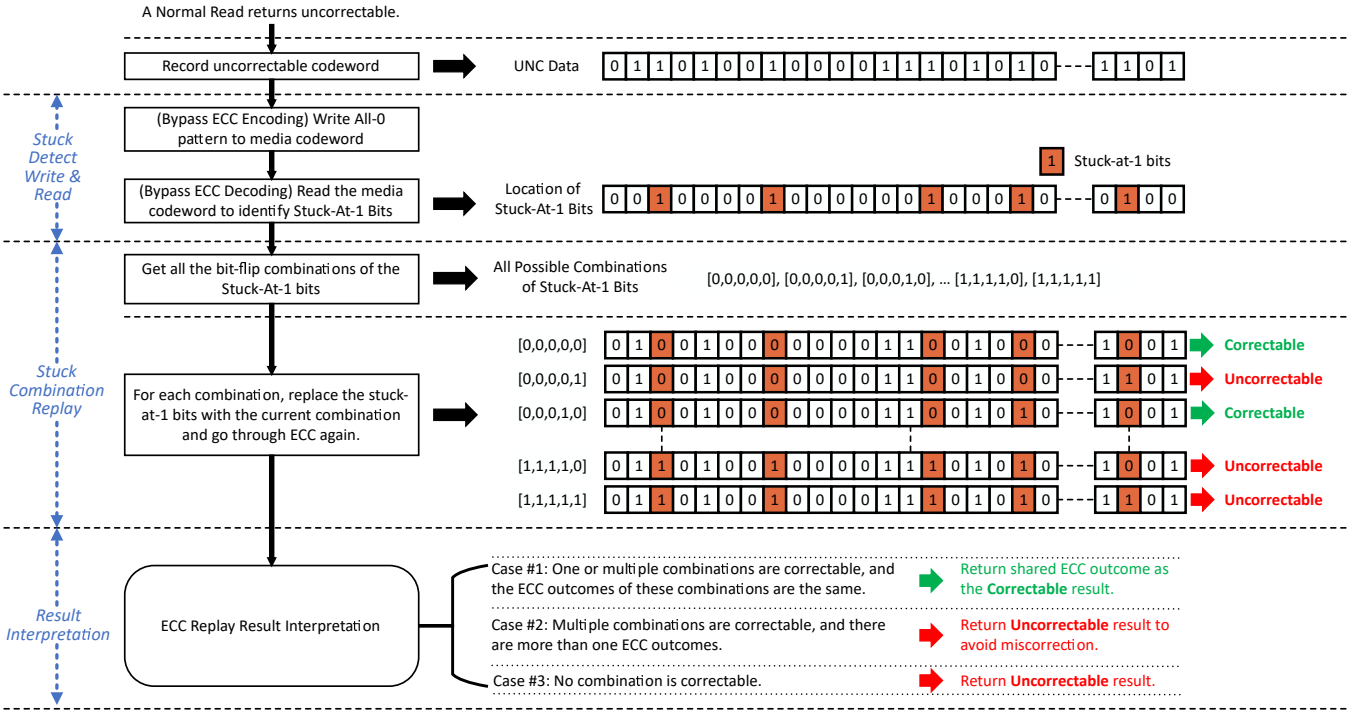

Figure 11: The Scheme of ECC Replay. Without loss of generality, we assume stuck bits are all stuck-at-1.

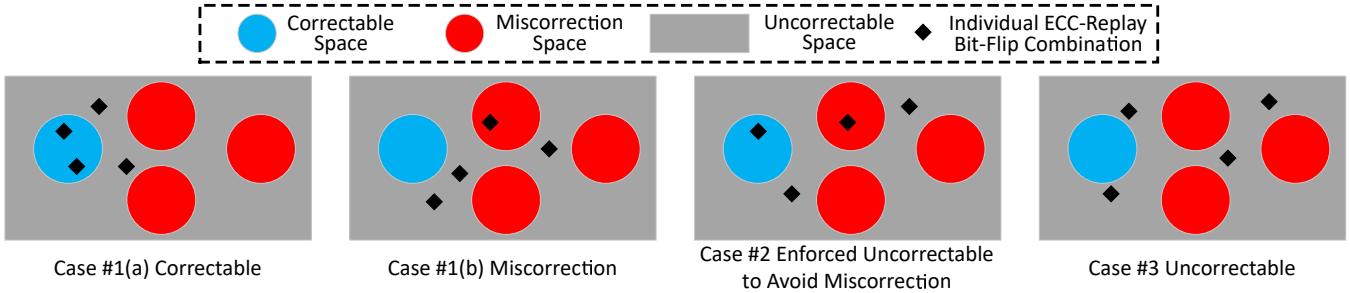

Figure 12: The Result Interpretation of ECC Replay.

UBER target, *ECC Replay* can support up to 3E-4 *Stuck Bit Rate*, a **30x improvement** of the baseline BCH-6 scheme.

4.4 Capping ECC Replay Tail Latency By Caching Many-Stuck Codewords

One potential issue of *ECC Replay* is it might take too much time replaying all the *Stuck Combinations* when there are many stuck bits. Figure 14 shows the number of codewords with a given number of stuck bits in a 256GB emerging memory under different Stuck Bit

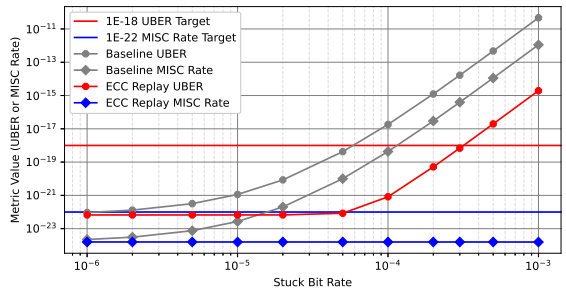

Figure 13: UBER and Miscorrection Rate of ECC Replay with BCH-6 and 1E-5 Soft BER.

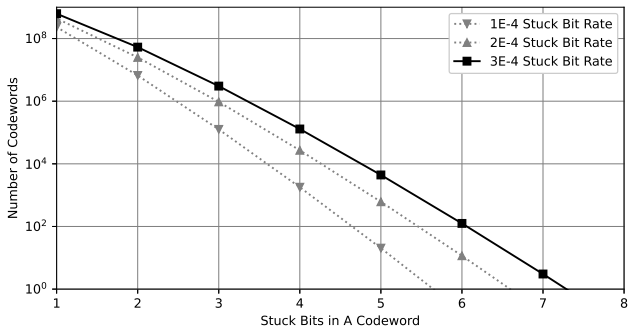

Figure 14: Number of Codewords with A Given Number of Stuck Bits in a 256GB emerging memory following binomial distribution. We can see that, there are less than 10 codewords with more than 6 stuck bits in an aggressive 3E-4 Stuck Bit Rate.

Rates. We can see that, even with an aggressive 3E-4 Stuck Bit Rate, there are only less than 10 codewords with more than 6 stuck bits. To efficiently handle these few codewords with many stuck bits, we can have a simple *Many-Stuck Cache* in front of the memory controller to cache these codewords with many stuck bits to cap the tail latency of *ECC Replay*. To fill the *Many-Stuck Cache*, we could periodically (e.g., once a day) scrub through the entire memory using the same stuck detection write and read commands in *ECC Replay*.

4.5 Reliability Summary

Figure 15 summaries the achievable reliability of *ECC Replay* with BCH-6 ECC and *Soft BER* of 1E-5. We can see that, *ECC Replay* hierarchy meet the 1E-18 *UBER* and 1E-22 *MISC Rate* targets with up to a 3E-4 *Stuck Bit Rate*, representing a 30× improvement over 1E-5 tolerable *Stuck Bit Rate* of baseline *Normal Read Only* scheme. Meanwhile, the trigger rate of *ECC Replay* is only 2.7E-6. The next section will show that such low trigger rate has negligible impact on the performance of memory system.

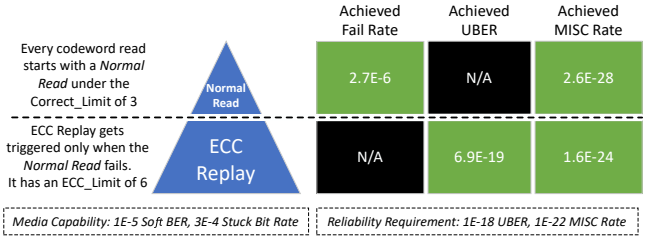

Figure 15: ECC Replay Achievable Reliability Summary. With BCH-6 ECC and Soft BER of 1E-5, ECC Replay can tolerate up-to 3E-4 Stuck Bit Rate. This represents a over 30× improvement over the baseline *Normal Read Only*, which can only tolerate 1E-5 Stuck Bit Rate.

Table 3: Simulation Parameters

Emerging Memory Media	
Memory Frequency	1.6GHz (DDR3200)
IO Interface	DDR-like IO organization: – Shared Command IO Bus by all dies; – Bidirectional Data IO Bus per die.
Banks	16, Codeword Interleave
Bank Cycle Time	40ns
Read Latency	35ns
Write Latency	10ns
Burst Length	16 (5ns for each codeword data transfer)
Soft Bit Error Rate	1E-5 following binomial distribution
Stuck Bit Rate	3E-4 following binomial distribution
Baseline Memory Controller Configuration	
Codeword Size	572 bits (512 user bits and 60 ecc bits)
ECC Decoder	Correcting up-to 6 bits 10ns decoding latency Pipelined, getting result every 5ns
Max Read Requests	32
Max Write Requests	8
Processing Queue Design	Separate Read and Write Queues per Bank
Read Queue Depth	16
Write Queue Depth	16
Write Drain Watermarks	10 (high) & 8 (low)
ECC Replayer Configuration	
Trigger Threshold	3 (When more than 3 bit errors in a *Normal Read*, ECC Replay is triggered)
ECC Decoder	Same configuration with the ECC decoder in baseline system

5 PERFORMANCE EVALUATION

5.1 Methodology

Without loss of generality, we simulate *ECC Replay* using a cycle-accurate in-house simulator following framework of DRAMSim3 [20]. Table 3 summarizes the simulation parameters.

On media side, our target emerging memory is similar with a commercial DDR4/5 system, with shared command bus by all dies, and bidirectional Data bus. Each die has 16 banks and operating at 1.6Ghz (DDR3200). The media read and write latencies are 35ns and 10ns, respectively. The cycle time of a bank is 40ns, meaning it can issue another read or write command at least 40ns after issuing the first one. The burst length of the data IO is 16, as a result it takes 5ns to transfer each codeword. To improve performance in sequential workloads, the banks are low-level interleaved. For failure modeling, we simulate a *Soft BER* of 1E-5 and *Stuck Bit Rate* of 3E-4, both of which following binomial distribution.

On memory controller side, it uses BCH-6 ECC with each codeword has 64B (512bit) user bits and 60 ECC bits. The ECC can correct up to 6 bits. The ECC decoder works in a pipeline fashion with each decoding taking 10ns and it can produce one ECC result every 5ns

(same with the data burst time of the media). The controller has separate read and write queues per bank, with each queue taking 16 commands. The write drain watermarks are 10 (high) and 8 (low). The write draining starts when any bank has more than 10 write commands, and stops when all banks have less than or equal to 8 write commands.

For *ECC Replayer*, we set its trigger threshold as 3. The separate ECC decoder in *ECC Replayer* has the same capability of the main ECC Decoder in the baseline system. The memory controller has a *Many-Stuck Cache* to bypass the codewords with more than 6 stuck bits, so *ECC Replayer* replays at most 6 stuck bits.

For workloads, we use five synthetic stress workloads which are typically used in Reliability Demonstrate Test (RDT) [8] flow:

- **Rand All-Rd**, which is an all-read workload with random generated address.
- **Rand 2R1W**, in which uses the access pattern of 2 reads followed by 1 writes with randomly generated addresses.
- **Seq All-Rd** and **Seq 2R1W** , which are the same with **Rand All-Rd** and **Rand 2R1W** except for using sequential addresses instead of random ones.
- **JEDEC-like** [17], which loosely follows JESD219 endurance test spec:
  - 60/40 read-write ratio;
  - 50% accesses to 5% of the memory space;
  - 30% accesses to 15% of the memory space;
  - 20% accesses to remainder of the memory space.

5.2 Results

Table 4: Performance Impact of ECC Replay

Workload	ECC Replay Trigger Rate	Avg. Read Req. Lat. (ns)	Max Read Req. Lat. (ns)	Avg. ECC-Replayed Read Req. Lat. (ns)	Max ECC-Replayed Read Req. Lat. (ns)
Rand All-Rd	3.63E-06	123.16	1199.38	371.36	798.13
Rand 2R1W	4.76E-06	177.51	1781.25	711.00	1167.50
Seq All-Rd	1.30E-06	120.93	781.25	351.56	631.88
Seq 2R1W	4.13E-06	171.40	1180.63	475.83	901.25
Jedec-like	4.71E-06	195.11	1918.13	437.71	849.38

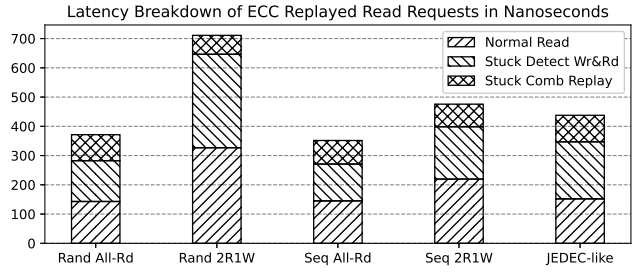

Figure 16: Latency Breakdown of ECC Replayed Read Requests.

The overall performance impact of *ECC Replay* is negligible– both average latency and bandwidth loss are within 0.2% in all five workloads. This is because *ECC Replay* has a really low trigger rate in 1E-6 range, as shown in Table 4. Table 4 also shows that, though the average latency of *ECC Replayed* read requests is longer than the average read request latency, the max *ECC Replayed* read request latency is within 1000ns and always smaller than the max read request latency. This indicates *ECC Replay* doesn't introduce extra tail latency for read requests.

Figure 16 shows that the breakdown of average ECC Replayed read request latency.We can see that:

- Expensive as it may sound, replaying stuck combinations actually represents a small fraction (less than 100ns in all workloads) of the overall latency of *ECC Replayed* read requests.
- The stuck detection read and write commands are responsible for a larger but still very affordable fraction of overall *ECC Replay* latency.

Table 5: ECC Replay Effectiveness On Different Target Systems

Target System			Error Tolerance of Baseline (Max BER)	Error Tolerance of ECC Replay		
ECC Config	UBER Target	MISC Rate Target		Max Soft BER	Max Stuck Bit Rate	Stuck Tolerance Ratio
BCH-4	1E-15	1E-17	5E-6	5E-6	2E-5	4×
BCH-6	1E-18	1E-22	1E-5	1E-5	3E-4	30×
BCH-8	1E-21	1E-25	5E-5	5E-5	6E-4	12×
BCH-10	1E-22	1E-29	1E-4	1E-4	1E-3	10×
BCH-12	1E-22	1E-32	2E-4	2E-4	4E-3	20×
BCH-14	1E-22	1E-33	5E-4	5E-4	9E-3	18×
BCH-16	1E-20	1E-34	1E-3	1E-3	3E-2	30×

6 DISCUSSION: EFFECTIVENESS OF ECC REPLAY ON OTHER ECC CONFIGURATIONS

We believe ECC Replay is a general scheme which can be used in a wide variety of ECC schemes and target systems. To verify this, we apply ECC Replay to wide range of target systems with different BCH algorithms and reliability targets. For each configuration:

- We first calculate the max tolerable BER for the baseline system only using *Normal Read*. Note that, here the bit errors contain both soft errors and stuck bit errors.
- We then calculate the error tolerance of ECC Replay, which contains the *Max Soft BER* and *Max Stuck Bit Rate*.
- We then calculate the *Stuck Tolerance Ratio* of ECC Replay, which the ratio of *Max Stuck Bit Rate* of ECC Replay to *Max BER* of baseline.

As is shown in Table 5, the max soft BER of ECC Replay is same with the max BER of the baeline, indicating that ECC Replay doesn't degrade the soft error correction capability of the ememory system. At the same time, ECC Replay manages to achieve high *Stuck Tolerance Ratio* (4–30 ×), indicating that ECC Replay effectively tolerates high stuck bit rate in all the configurations.

7 RELATED WORK

The concept of replaying stuck bit combinations has actually been covered in a 1980s patent [6]. Our paper, based on detailed modeling and simulation, shows that it is an ideal foundation to solve the stuck bit problem of emerging memories. Our paper also proposed crucial enhancements (i.e., *Undercorrection*, *Replay Result Interpretation*) to

achieve the required *MISC Rate* target and avoid creating a large tail latency.

Using undercorrection of BCH code to reduce miscorrection rate has been proposed in patent [14]. This patent targets a layered read scheme for emerging memories, where:

- A fast but more errornous read is performed first;
- If the fast read returns an error, the memory controller issues a slow but more accurate read.

The issue of this layered read scheme is that the fast read layer has an unacceptable high miscorrection rate because of fast read's high BER (bit error rate). The patent [14] proposes to *undercorrect* BCH decoding of the fast reads, therefore the *Miscorrection Rate* is dramatically reduced at the expense of higher fast read fail rate.

Flexible stuck correction approaches, including ECP [30], SAFER [31], Free-P [37], PAYG [28], etc., are based on the same observation that not all the codewords have many stuck bits. Same with ECC Replay, these schemes typically have a default read scheme with relatively limited error correction capability, and if the default read scheme fails, a fallback read mechanism is triggered to handle the stuck bits. Different from ECC Replay, the recovery schemes of these prior approaches usually involve pointer chasing to read additional codewords containing the stuck bits information, which adds substantial complexity to the system. Also, the storage efficiency of these approaches are not necessarily better than state-of-the-art ECC approach like BCH—for example, ECP6 [30] has an overhead of 11.9% and can correct up to 6 *stuck-only* bits; FREE-p [37] has an overhead of 12.5% and can correct up to 4 error bits. Both ECP6 [30] and FREE-p [37] are actually inferior than our baseline BCH-6 ECC, which has an overhead of 11.7% and can correct up to 6 error bits of *any* kind. Compared to these prior approaches, *ECC Replay*, which achieves aggressive 30x tolerance to Stuck Bit Rate with low complexity, negligible performance degradation and no storage overhead, is more advanced and practical.

Another class of solutions to stuck bit issue is stuck bit encoding [21–23, 25]. Based on the observation that a stuck bit only manifests as a bit error when encoded the opposite data, stuck bit encoding tries to encode the codeword so the stuck bits do not show as bit errors. Though well-intentioned, stuck bit encoding faces some practical difficulties: first, the controller needs to know the location of stuck bits in write encoding, so for each write it needs to perform an additional read to the codeword to get the stuck bit location information, which is costly in terms of write performance; second, such multi-encoding schemes likely increase the probability of miscorrection. However to our best knowledge, none of such prior proposals has studied their impact on *MISC Rate*.

8 CONCLUSION

Stuck bits of emerging memories are typically caused by media endurance and manufacturing failures, which are dominant failures of emerging memories. As a result, tolerating more stuck bits can greatly help simplify the manufacturing process and improve yield, and thus reduce the cost. In this paper, we first quantify *UBER*, *MISC Rate* requirements of a practical emerging memory system with both *soft errors* and *stuck bits*. The modeling results show that *MISC Rate* is often the system reliability bottleneck and has to handled properly. With this in mind, we present a layered ECC framework which triggers a recovery scheme (i.e., *ECC Replay*) if the initial read is an *uncorrectable*. *ECC Replay* detects the stuck bits by writing specific data and then reading back, then exhausts all the data combinations of the stuck bits and decodes them again through a separate *ECC decoder*. We also propose crucial schemes, including *Undercorrection* and conservative *ECC Replay* results interpretation, to minimize *MISC Rate*. Our modeling and simulation results show that *ECC Replay* achieves 30× improvement of stuck-bit tolerance with negligible impact on overall system performance in terms of memory bandwidth and read latency distribution. Moreover, our scheme achieves this aggressive goal with minimal change in the memory controller architecture and no change to the emerging memory media, making it an ideal solution to practical emerging memory products.

ACKNOWLEDGMENTS

We would like to thank Dr. Martin Hassner for his generous help in the initial phase of this project.

REFERENCES

[1] "DRAM Market to Reach USD 272.5 Billion," https://finance.yahoo.com/news/dram-market-reach-usd-272-010000616.html.
[2] "Intel's Optane Business Haemorrhaged Over Half a Billion Dollars in 2020," https://www.tomshardware.com/news/intel-optane-massive-losses.
[3] "Large language model," https://en.wikipedia.org/wiki/Large_language_model.
[4] "Soft-Decoding in LDPC based SSD Controllers," https://www.eetimes.com/soft-decoding-in-ldpc-based-ssd-controllers/.
[5] H. Akinaga and H. Shima, "Resistive random access memory (reram) based on metal oxides," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2237–2251, 2010.
[6] D. L. Arlington, C.-L. Chen, and E. K. Evans, "Extended error correction for package error correction codes," Apr. 28 1987, uS Patent 4,661,955.
[7] F. Bedeschi, C. Resta, O. Khouri, E. Buda, L. Costa, M. Ferraro, F. Pellizzer, F. Ottogalli, A. Pirovano, M. Tosi, R. Bez, R. Gastaldi, and G. Casagrande, "An 8mb demonstrator for high-density 1.8v phase-change memories," in *2004 Symposium on VLSI Circuits. Digest of Technical Papers (IEEE Cat. No.04CH37525)*, 2004, pp. 442–445.
[8] N. Bidokhti, "Ssd next gen rdt," in *2016 Annual Reliability and Maintainability Symposium (RAMS)*, 2016, pp. 1–4.
[9] C.-Y. Chen, H.-C. Shih, C.-W. Wu, C.-H. Lin, P.-F. Chiu, S.-S. Sheu, and F. T. Chen, "Rram defect modeling and failure analysis based on march test and a novel squeeze-search scheme," *IEEE Transactions on Computers*, vol. 64, no. 1, pp. 180–190, 2014.
[10] T. Endoh, H. Honjo, K. Nishioka, and S. Ikeda, "Recent progresses in stt-mram and sot-mram for next generation mram," in *2020 IEEE Symposium on VLSI Technology*, 2020, pp. 1–2.
[11] A. Fazio, "Advanced technology and systems of cross point memory," in *2020 IEEE International Electron Devices Meeting (IEDM)*. IEEE, 2020, pp. 24–1.
[12] M. Ferrari, P. Amato, C. Laurent, M. Sforzin, L. Barletta, and S. Bellini, "Ultra-fast error correction and detection for low-latency storage applications with emerging memories," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2018, pp. 1–5.
[13] G. Forney, "On decoding bch codes," *IEEE Transactions on information theory*, vol. 11, no. 4, pp. 549–557, 1965.
[14] M. Hassner, M. N. A. Tran, W. Parkinson, M. Grobis, N. Franklin, and R. Ramanujan, "Programmable ecc for mram mixed-read scheme," Apr. 30 2024, uS Patent 11,972,822.
[15] S. Ikegawa, F. B. Mancoff, and S. Aggarwal, "Commercialization of mram–historical and future perspective," in *2021 IEEE International Interconnect Technology Conference (IITC)*. IEEE, 2021, pp. 1–3.
[16] S. Jaffer, S. Maneas, A. Hwang, and B. Schroeder, "The reliability of modern file systems in the face of ssd errors," *ACM Transactions on Storage (TOS)*, vol. 16, no. 1, pp. 1–28, 2020.
[17] JESD219, "Solid-state drive (ssd) endurance workloads," 2010.
[18] S. H. Jo, T. Kumar, S. Narayanan, W. D. Lu, and H. Nazarian, "3d-stackable crossbar resistive memory based on field assisted superlinear threshold (fast) selector," in *2014 IEEE International Electron Devices Meeting*, 2014, pp. 6.7.1–6.7.4.
[19] D. Kau, S. Tang, I. V. Karpov, R. Dodge, B. Klehn, J. A. Kalb, J. Strand, A. Diaz, N. Leung, J. Wu, S. Lee, T. Langtry, K. wei Chang, C. Papagianni, J. Lee, J. Hirst, S. Erra, E. Flores, N. Righos, H. Castro, and G. Spadini, "A stackable cross point phase

change memory," in *2009 IEEE International Electron Devices Meeting (IEDM)*, 2009, pp. 1–4.
[20] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob, "Dramsim3: A cycle-accurate, thermal-capable dram simulator," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 106–109, 2020.
[21] R. Maddah, S. Cho, and R. Melhem, "Power of one bit: Increasing error correction capability with data inversion," in *2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing*, 2013, pp. 216–225.
[22] R. Maddah, S. Cho, and R. Melhem, "Symbol shifting: Tolerating more faults in pcm blocks," *IEEE Transactions on Computers*, vol. 65, no. 7, pp. 2270–2283, 2016.
[23] R. Maddah, R. Melhem, and S. Cho, "Rdis: Tolerating many stuck-at faults in resistive memory," *IEEE Transactions on Computers*, vol. 64, no. 3, pp. 847–861, 2015.
[24] A. Marelli and R. Micheloni, "False decoding probability (detection) of bch and ldpc codes," *Flash Memory Summit*, vol. 11, 2016.
[25] R. Melhem, R. Maddah, and S. Cho, "Rdis: A recursively defined invertible set scheme to tolerate multiple stuck-at faults in resistive memory," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, 2012, pp. 1–12.
[26] G. Panagopoulos, C. Augustine, and K. Roy, "Modeling of dielectric breakdown-induced time-dependent stt-mram performance degradation," in *69th Device Research Conference*. IEEE, 2011, pp. 125–126.
[27] A. Pirovano, A. Redaelli, F. Pellizzer, F. Ottogalli, M. Tosi, D. Ielmini, A. L. Lacaita, and R. Bez, "Reliability study of phase-change nonvolatile memories," *IEEE Transactions on Device and Materials Reliability*, vol. 4, no. 3, pp. 422–427, 2004.
[28] M. K. Qureshi, "Pay-as-you-go: Low-overhead hard-error correction for phase change memories," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 318–328.
[29] W. E. Ryan *et al.*, "An introduction to ldpc codes," *CRC Handbook for Coding and Signal Processing for Recording Systems*, vol. 5, no. 2, pp. 1–23, 2004.
[30] S. Schechter, G. H. Loh, K. Strauss, and D. Burger, "Use ecp, not ecc, for hard failures in resistive memories," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 141–152, 2010.
[31] N. H. Seong, D. H. Woo, V. Srinivasan, J. A. Rivers, and H.-H. S. Lee, "Safer: Stuck-at-fault error recovery for memories," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2010, pp. 115–124.
[32] S. Shiratake, "Scaling and performance challenges of future dram," in *2020 IEEE international memory workshop (IMW)*. IEEE, 2020, pp. 1–3.
[33] D. Strukov, "The area and latency tradeoffs of binary bit-parallel bch decoders for prospective nanoelectronic memories," in *2006 Fortieth Asilomar Conference on Signals, Systems and Computers*, 2006, pp. 1183–1187.
[34] K. Tsunoda, K. Kinoshita, H. Noshiro, Y. Yamazaki, T. Iizuka, Y. Ito, A. Takahashi, A. Okano, Y. Sato, T. Fukano, M. Aoki, and Y. Sugiyama, "Low power and high speed switching of ti-doped nio reram under the unipolar voltage source of less than 3 v," in *2007 IEEE International Electron Devices Meeting*, 2007, pp. 767–770.
[35] S. B. Wicker and V. K. Bhargava, *Reed-Solomon codes and their applications*. John Wiley & Sons, 1999.
[36] J. Yang-Scharlotta, M. Fazio, M. Amrbar, M. White, and D. Sheldon, "Reliability characterization of a commercial tao x-based reram," in *2014 IEEE International Integrated Reliability Workshop Final Report (IIRW)*. IEEE, 2014, pp. 131–134.
[37] D. H. Yoon, N. Muralimanohar, J. Chang, P. Ranganathan, N. P. Jouppi, and M. Erez, "Free-p: Protecting non-volatile memory against both hard and soft errors," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. IEEE, 2011, pp. 466–477.

9 APPENDIX

We use the target media capability of 1E-5 Soft BER, 3E-4 Stuck Bit Rate and 50% S2E Rate, and BCH-6 error correction code.

9.1 Getting Miscorrection Probability of BCH (*BCH_Misc_Prob*)

$$Total\_Cw\_Comb = 2^{cw\_size} \tag{4}$$

$$Correctable\_Comb = \sum_{errors=0}^{ecc\_limit} C(cw\_size, errors) \tag{5}$$

$$Uncorrectable\_Comb = Total\_Cw\_Comb - Correctable\_Comb \tag{6}$$

$$Misc\_Comb = (2^{user\_bits} - 1) \times Correctable\_Comb \tag{7}$$

$$BCH\_Misc\_Prob = \frac{Misc\_Comb}{Uncorrectable\_Comb} \tag{8}$$

This subsection shows how to approximate *BCH_Misc_Prob* of baseline BCH code:

- Equation 4 calculates the total combinations of the codeword space.
- Equation 5 provides the correctable combinations of the target codeword, which is the sum of the combinations with number of bit errors within its *ECC Limit*.
- Equation 6 provides the uncorrectable combinations (including miscorrections), which is the total combinations of the codeword space except for the correctable combinations.
- Equation 7 provides the miscorrection combinations of the target codeword, which is the sum of correctable combinations of all other codewords.
- Equation 8 provides the probability of miscorrection when the number of bit errors is larger than *ECC Limit*. Since this probability can be estimated as a constant, it can be approximated as the combinations of all correctables in all codewords divided by the total combinations of the codeword space. In our baseline BCH-6 ([572, 512, 13]) ECC configuration, this value is 4.2E-5.

Algo. 1: Computing the probability of having X stuck bits and Y soft bit errors given the *Stuck Bit Rate* and *Soft BER*.

```
Stuck_Bit_Rate = 3E-4; Soft_BER = 1E-5; ECC_Limit = 6;
def computeMediaFailureProb(Total_Bits, Stuck_Bit_Rate, Soft_BER):
  Media_Fail_Prob_Array = [][];
  for Stuck_Bits in range(0, Total_CW_Bits):
     Stuck_Bits_Prob = binom.pmf(Stuck_Bits,Total_Bits,
          Stuck_Bit_Rate);
     Func_Bits = Total_CW_Bits-Stuck_Bits;
     for Soft_Errors in range(0, Total_Bits-Stuck_Bits):
        Soft_Error_Prob = binom.pmf(Soft_Errors,Func_Bits,Soft_BER
             );
        Combined_Prob = Stuck_Bits_Prob * Soft_Error_Prob;
        Media_Fail_Prob_Array[Stuck_Bits][Soft_Errors] =
             Combined_Prob;
  return Media_Fail_Prob_Array;
```

		Soft Bit Errors							
		0	1	2	3	4	5	6	7
Stuck Bits	0	0.8375	0.0048	1.4E-05	2.6E-08	3.7E-11	4.2E-14	4.0E-17	3.2E-20
	1	0.1438	8.2E-04	2.3E-06	4.4E-09	6.3E-12	7.1E-15	6.7E-18	5.4E-21
	2	0.0123	7.0E-05	2.0E-07	3.8E-10	5.4E-13	6.1E-16	5.7E-19	4.6E-22
	3	7.0E-04	4.0E-06	1.1E-08	2.1E-11	3.0E-14	3.4E-17	3.2E-20	2.6E-23
	4	3.0E-05	1.7E-07	4.8E-10	9.1E-13	1.3E-15	1.5E-18	1.4E-21	1.1E-24
	5	1.0E-06	5.8E-09	1.6E-11	3.1E-14	4.4E-17	4.9E-20	4.6E-23	3.7E-26
	6	2.9E-08	1.6E-10	4.6E-13	8.7E-16	1.2E-18	1.4E-21	1.3E-24	1.0E-27
	7	7.0E-10	4.0E-12	1.1E-14	2.1E-17	3.0E-20	3.3E-23	3.1E-26	2.5E-29

Algo. 2: Computing the probability of X stuck bits resulting in Y stuck errors.

```
S2E_Rate = 0.5; #The rate of a stuck bit becoming a stuck error.
def computeStuck2ErrorProb(Total_CW_Bits, S2E_Rate):
    Stuck2Error_Prob_Array = [][];
    for Stuck_Bits in range(0, Total_CW_Bits):
         for Stuck_Errors in range(0, Stuck_Bits+1):
              S2E_Prob = binom.pmf(Stuck_Errors,Stuck_Bits,
                   S2E_Rate);
              Stuck2Error_Prob_Array[Stuck_Bits][Stuck_Errors] =
                   S2E_Prob;
    return Stuck2Error_Prob_Array;
```

		Stuck Errors							
		0	1	2	3	4	5	6	7
Stuck Bits	0	1.000							
	1	0.500	0.500						
	2	0.250	0.500	0.250					
	3	0.125	0.375	0.375	0.125				
	4	0.063	0.250	0.375	0.250	0.063			
	5	0.031	0.156	0.312	0.312	0.156	0.031		
	6	0.016	0.094	0.234	0.312	0.234	0.094	0.016	
	7	0.008	0.055	0.164	0.273	0.273	0.164	0.055	0.008

9.2 Failure Modeling of Stuck Bits and Soft Bit Errors

In this subsection, we present the details of failure modeling BCH code with given *Soft BER*, *Stuck Bit Rate* and *S2E Rate*. We first calculate the *UNC Rate*:

- Following Section 3.4, Algorithm 1 calculates a 2D probability array of having X stuck bits and Y soft bit errors, and the following table shows the result.
- Algorithm 2 calculates the probability of having Y *stuck errors* out of X *stuck bits*, and the following table shows the result.
- With the results of Algorithms 2, Algorithm 3 calculates a probability 2D array of having an *Uncorrectable* with X stuck bits and Y soft bit errors, and the following table shows the result.
- Combining the result of Algorithms 1 and 3, Algorithm 4 calculates the *UNC Rate* with the given media capability.

We now calculate the *MISC Rate*:

- With the results of Algorithm 2, Algorithm 5 calculates a 2D array of the probability of having a *Miscorrection* with X stuck bits and Y soft bit errors, and the following table shows the result.
- Combining the result of Algorithms 1 and 5, Algorithm 6 calculates the *MISC Rate*.

Algo. 3: Computing the probability of having an uncorrectable when a codeword has X stuck bits and Y soft bit errors

```
Total_CW_Bits=572; ECC_Limit = 6; S2E_Rate = 0.5;
def computeMediaFail2UncProb(Total_CW_Bits,S2E_Rate,ECC_Limit):
  MediaFail2Unc_Prob_Array = [][];
  S2E_Prob_Array=computeStuck2ErrProb(Total_CW_Bits,S2E_Rate);
  for Stuck_Bits in range(0, Total_CW_Bits+1):
    for Soft_Errors in range(0, Total_CW_Bits-Stuck_Bits+1):
      MediaFail2Unc_Prob = 0;
      for Stuck_Errs in range(0, Stuck_Bits+1):
        if Stuck_Errs + Soft_Errs > ECC_Limit:
          MediaFail2Unc_Prob +=  S2E_Prob_Array[Stuck_Bits][
               Soft_Errs];
      MediaFail2Unc_Prob_Array[Stuck_Bits][Soft_Errors] =
           MediaFail2Unc_Prob;
  return MediaFail2Unc_Prob_Array;
```

		Soft Bit Errors							
		0	1	2	3	4	5	6	7
Stuck Bits	0	0	0	0	0	0	0	0	100%
	1	0	0	0	0	0	0	50.0%	100%
	2	0	0	0	0	0	25.0%	75.0%	100%
	3	0	0	0	0	12.5%	50.0%	87.5%	100%
	4	0	0	0	6.25%	31.3%	68.8%	93.8%	100%
	5	0	0	3.13%	18.8%	50.0%	81.2%	96.9%	100%
	6	0	1.56%	10.9%	34.4%	65.6%	89.1%	98.4%	100%
	7	0.78%	6.25%	22.7%	50.0%	77.3%	93.7%	99.2%	100%

9.3 Failure Modeling with Undercorrected BCH Code

In this subsection, we present the details of failure modeling baseline BCH code with *Undercorrection*. We first approximate *MISC Rate* of BCH code with *Undercorrection*:

- Equation 9 provides the correctable combinations of a single codeword with *Undercorrection*, which is the sum of combinations of flipping the number of bits within the chosen *Correct Limit* (instead of *ECC Limit*).
- Equation 10 provides the correctable combinations with *Undercorrection* of all codewords.
- Equation 11 calculates the max number of errors to be miscorrection-free (i.e., *NO MISC Distance*) with the chosen *Correct Limit*.
- Equation 12 estimates the probability of miscorrection when the number of bit errors is larger than *NO MISC Distance*.

Table 6 shows the probability of miscorrection of BCH-6 code with different *Correct Limits*. We can see that, reducing *Correct Limit* can effectively reduce the miscorrection probability of BCH-6 code– Compared with no *Undercorrection*, a *Correct Limit* of 3 reduces the miscorrection probability by more than **6 decades**.

$$Single\_CW\_UnderCorrect\_Comb = \sum_{Error\_Bits=0}^{Correct\_Limit} C(Total\_Bits, Error\_Bits) \quad (9)$$

$$All\_CWs\_UnderCorrect\_Comb = 2^{User\_Bits} \times Single\_CW\_UnderCorrect\_Comb \quad (10)$$

$$No\_MISC\_Distance = 2 \times ECC\_Limit - Under\_Correct\_Limit \quad (11)$$

$$BCH\_MISC\_Prob \approx \frac{All\_CWs\_UnderCorrect\_Comb}{Total\_CW\_Comb} \quad (12)$$

Algo. 4: Computing *Uncorrectable Rate* based on Algothrims 1–3

```
Total_CW_Bits=572; Stuck_Bit_Rate = 3E-4;
Soft_BER = 1E-5; ECC_Limit = 6; S2E_Rate = 0.5;
def computeUncRate(Total_CW_Bits,ECC_Limit,Stuck_Bit_Rate,Soft_BER
      ,S2E_Rate):
  Media_Fail_Prob_Array = computeMediaFailureProb(Total_CW_Bits,
       Stuck_Bit_Rate, Soft_BER);
  MediaFail2Unc_Prob_Array = computeMediaFail2UncProb(
       Total_CW_Bits,S2E_Rate,ECC_Limit);
  UNC_Rate = 0;
  for Stuck_Bits in range(0, Total_CW_Bits+1):
    for Soft_Errors in range(0, Total_CW_Bits-Stuck_Bits+1):
      UNC_Rate = UNC_Rate + Media_Fail_Prob_Array[Stuck_Bits][
           Soft_Errors] * MediaFail2Unc_Prob_Array[Stuck_Bits][
           Soft_Errors];
  return UNC_Rate;
```

Algorithms 7 and 8 calculate *UNC Rate* and *MISC Rate* with *Undercorrection*. The red texts highlight the key differences from their *no-undercorrection* counterparts, which are:

- Using *Correct Limit* instead of *ECC Limit* as the *uncorrectable* boundary.
- Using *No MISC Diantance* instead of *ECC Limit* as the *miscorrection* boundary.
- Having different *BCH_MISC_Prob* for different *Correct Limit*.

Figure 9 shows *UNC Rates* and *MISC Rates* of BCH-6 under different *Correct Limits*.

9.4 Failure Modeling of ECC Replay

Compared with the modeling baseline BCH ECC (Section 9.2), the only difference of modeling *ECC Replay* is having different *UNC and MISC Probability Arrays*. Following Section 4.3 and Table 2, Figure 17 shows how to derive these arrays from the *UNC and MISC Probability Arrays* of baseline BCH ECC.

Table 6: BCH_MISC_Prob of BCH-6 ECC Code with Different Correct_Limit

Correct Limit	3	4	5	6 (No Undercorrection)
BCH_MISC_Prob	2.7E-11	3.9E-09	4.4E-07	4.2E-05

Algo. 5: Computing the probability of having a miscorrection when a codeword has X stuck bits and Y soft bit errors

```
Total_CW_Bits=572;  ECC_Limit = 6; S2E_Rate = 0.5;
def computeMediaFail2MiscProb(Total_CW_Bits,S2E_Rate,ECC_Limit):
  Media_Fail2Misc_Prob_Array = [][];
  S2E_Prob_Array=computeStuck2ErrProb(Total_CW_Bits,S2E_Rate);
  for Stuck_Bits in range(0, Total_CW_Bits+1):
    for Soft_Errors in range(0, Total_CW_Bits-Stuck_Bits+1):
      Media_Fail2Misc_Prob = 0;
      for Stuck_Errs in range(0, Stuck_Bits+1):
        if Stuck_Errs + Soft_Errs > ECC_Limit:
          Media_Fail2Misc_Prob+= (S2E_Prob_Array[Stuck_Bits][
               Soft_Errs] * BCH_MISC_Prob);
      Media_Fail2Misc_Prob_Array[Stuck_Bits][Soft_Errors]=
           Media_Fail2Misc_Prob;
  return Media_Fail2Misc_Prob_Array;
```

		Soft Bit Errors							
		0	1	2	3	4	5	6	7
Stuck Bits	0	0	0	0	0	0	0	0	4.2E-05
	1	0	0	0	0	0	0	2.1E-05	4.2E-05
	2	0	0	0	0	0	1.0E-05	3.1E-05	4.2E-05
	3	0	0	0	0	5.2E-06	2.1E-05	3.6E-05	4.2E-05
	4	0	0	0	2.6E-06	1.3E-05	2.9E-05	3.9E-05	4.2E-05
	5	0	0	1.3E-06	7.8E-06	2.1E-05	3.4E-05	4.0E-05	4.2E-05
	6	0	6.5E-07	4.5E-06	1.4E-05	2.7E-05	3.7E-05	4.1E-05	4.2E-05
	7	3.2E-07	2.6E-06	9.4E-06	2.1E-05	3.2E-05	3.9E-05	4.1E-05	4.2E-05

Algo. 6: Computing *MISC Rate* based on Algothrims 1, 2&5

```
Total_CW_Bits=572; Stuck_Bit_Rate = 3E-4;
Soft_BER = 1E-5; ECC_Limit = 6; S2E_Rate = 0.5;
def computeMiscRate(Total_CW_Bits,ECC_Limit,Stuck_Bit_Rate,
      Soft_BER,S2E_Rate):
  Media_Fail_Prob_Array = computeMediaFailureProb(Total_CW_Bits,
       Stuck_Bit_Rate, Soft_BER);
  Media_Fail2Misc_Prob_Array =  computeMediaFail2MiscProb(
       Total_CW_Bits,S2E_Rate,ECC_Limit);
  MISC_Rate = 0;
  for Stuck_Bits in range(0, Total_CW_Bits+1):
    for Soft_Errors in range(0, Total_CW_Bits-Stuck_Bits+1):
      MISC_Rate += (Media_Fail_Prob_Array[Stuck_Bits][Soft_Errors
           ]*Media_Fail2Misc_Prob_Array[Stuck_Bits][Soft_Errors]);
  return MISC_Rate;
```

Algo. 7: Computing *Uncorrectable Rate* with *Undercorrection*. The differences from full strength case are marked in red.

```
Total_CW_Bits=572; Stuck_Bit_Rate = 3E-4;
Soft_BER = 1E-5; ECC_Limit = 6; S2E_Rate = 0.5;
Correct_Limit = a value in range from 3 to 6;
def computeUncRate(Total_CW_Bits,Correct_Limit,Stuck_Bit_Rate,
      Soft_BER,S2E_Rate):
  Media_Fail_Prob_Array =  computeMediaFailureProb(Total_CW_Bits,
       Stuck_Bit_Rate, Soft_BER);
  MediaFail2Unc_Prob_Array = computeMediaFail2UncProb(
       Total_CW_Bits,S2E_Rate,Correct_Limit);
  UNC_Rate = 0;
  for Stuck_Bits in range(0, Total_CW_Bits+1):
    for Soft_Errors in range(0, Total_CW_Bits-Stuck_Bits+1):
      UNC_Rate = UNC_Rate + Media_Fail_Prob_Array[Stuck_Bits][
           Soft_Errors] * MediaFail2Unc_Prob_Array[Stuck_Bits][
           Soft_Errors];
  return UNC_Rate;
```

Media Fail to Uncorrectable Probability Array of Baseline ECC (BCH-6)

Stuck Bits \ Soft Bit Errors	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	100.00%	100.00%
1	0	0	0	0	0	0	50.00%	100.00%	100.00%
2	0	0	0	0	0	25.00%	75.00%	100.00%	100.00%
3	0	0	0	0	12.50%	50.00%	87.50%	100.00%	100.00%
4	0	0	0	6.25%	31.25%	68.75%	93.75%	100.00%	100.00%
5	0	0	3.13%	18.75%	50.00%	81.25%	96.87%	100.00%	100.00%
6	0.00%	1.56%	10.94%	34.38%	65.63%	89.06%	98.44%	100.00%	100.00%
7	0.78%	6.25%	22.66%	50.00%	77.34%	93.75%	99.22%	100.00%	100.00%
8	3.52%	14.45%	36.33%	63.67%	85.55%	96.48%	99.61%	100.00%	100.00%

Media Fail to Uncorrectable Probability Array of ECC Replay with BCH-6

Stuck Bits \ Soft Bit Errors	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	100.00%	100.00%
1	0	0	0	0	0	0	2.08E-05	100.00%	100.00%
2	0	0	0	0	0	1.04E-05	3.12E-05	100.00%	100.00%
3	0	0	0	0	5.19E-06	2.08E-05	3.63E-05	100.00%	100.00%
4	0	0	0	2.60E-06	1.30E-05	2.86E-05	3.89E-05	100.00%	100.00%
5	0	0	1.30E-06	7.79E-06	2.08E-05	3.37E-05	4.02E-05	100.00%	100.00%
6	0	6.49E-07	4.54E-06	1.43E-05	2.73E-05	3.70E-05	4.09E-05	100.00%	100.00%
7	3.25E-07	2.60E-06	9.41E-06	2.08E-05	3.21E-05	3.89E-05	4.12E-05	100.00%	100.00%
8	1.46E-06	6.00E-06	1.51E-05	2.64E-05	3.55E-05	4.01E-05	4.14E-05	100.00%	100.00%

Media Fail to Miscorrection Probability Array of Baseline ECC (BCH-6)

Stuck Bits \ Soft Bit Errors	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	4.15E-05	4.15E-05
1	0	0	0	0	0	0	2.08E-05	4.15E-05	4.15E-05
2	0	0	0	0	0	1.04E-05	3.12E-05	4.15E-05	4.15E-05
3	0	0	0	0	5.19E-06	2.08E-05	3.63E-05	4.15E-05	4.15E-05
4	0	0	0	2.60E-06	1.30E-05	2.86E-05	3.89E-05	4.15E-05	4.15E-05
5	0	0	1.30E-06	7.79E-06	2.08E-05	3.37E-05	4.02E-05	4.15E-05	4.15E-05
6	0	6.49E-07	4.54E-06	1.43E-05	2.73E-05	3.70E-05	4.09E-05	4.15E-05	4.15E-05
7	3.25E-07	2.60E-06	9.41E-06	2.08E-05	3.21E-05	3.89E-05	4.12E-05	4.15E-05	4.15E-05
8	1.46E-06	6.00E-06	1.51E-05	2.64E-05	3.55E-05	4.01E-05	4.14E-05	4.15E-05	4.15E-05

Media Fail to Miscorrection Probability Array of ECC Replay with BCH-6

Stuck Bits \ Soft Bit Errors	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	4.15E-05	4.15E-05
1	0	0	0	0	0	0	0	4.15E-05	4.15E-05
2	0	0	0	0	0	0	0	4.15E-05	4.15E-05
3	0	0	0	0	0	0	0	4.15E-05	4.15E-05
4	0	0	0	0	0	0	0	4.15E-05	4.15E-05
5	0	0	0	0	0	0	0	4.15E-05	4.15E-05
6	0	0	0	0	0	0	0	4.15E-05	4.15E-05
7	0	0	0	0	0	0	0	4.15E-05	4.15E-05
8	0	0	0	0	0	0	0	4.15E-05	4.15E-05

Figure 17: Quantifying *UNC Rate* and *MISC Rate* of ECC Replay. As is mentioned in Section 4.3, if the number of *Soft Bit Errors* is within *ECC Limit* (6 for BCH-6), ECC Replay (1) removes all the baseline *Uncorrectables*; (2) transfers all the baseline *Miscorrections* to *Uncorrectables*.

Algo. 8: Computing *Miscorrection Rate* with *Undercorrection*. The differences from full strength case are marked in red.

```
Total_CW_Bits=572; Stuck_Bit_Rate = 3E-4;
Soft_BER = 1E-5; ECC_Limit = 6; S2E_Rate = 0.5;
Correct_Limit = a value in range from 3 to 6;
#Returns A 2D array of probability of  having an uncorrectable
     when a codeword
#has X stuck bits and Y soft bit errors
Total_CW_Bits=572;  ECC_Limit = 6; S2E_Rate = 0.5;
def computeMediaFail2MiscProb(Total_CW_Bits,S2E_Rate,ECC_Limit,
     Correct_Limit):
  Media_Fail2Misc_Prob_Array = [][];
  No_Misc_Diantance = ECC_Limit * 2 - Correct_Limit;
  S2E_Prob_Array=computeStuck2ErrProb(Total_CW_Bits,S2E_Rate);
  for Stuck_Bits in range(0, Total_CW_Bits+1):
    for Soft_Errors in range(0, Total_CW_Bits-Stuck_Bits+1):
      Media_Fail2Misc_Prob = 0;
      for Stuck_Errs in range(0, Stuck_Bits+1):
        Total_Errs = Stuck_Errs + Soft_Errs;
        if Total_Errs > No_Misc_Diantance:
          Media_Fail2Misc_Prob+= (S2E_Prob_Array[Stuck_Bits][
               Soft_Errs] * BCH_MISC_Prob[Total_Errs]);
      Media_Fail2Misc_Prob_Array[Stuck_Bits][Soft_Errors]=
           Media_Fail2Misc_Prob;
  return Media_Fail2Misc_Prob_Array;

def computeMiscRate(Total_CW_Bits,ECC_Limit,Correct_Limit,
     Stuck_Bit_Rate,Soft_BER,S2E_Rate):
  Media_Fail_Prob_Array = computeMediaFailureProb(Total_CW_Bits,
       Stuck_Bit_Rate, Soft_BER);
  Media_Fail2Misc_Prob_Array = computeMediaFail2MiscProb(
       Total_CW_Bits,S2E_Rate,ECC_Limit,Correct_Limit);
  MISC_Rate = 0;
  for Stuck_Bits in range(0, Total_CW_Bits+1):
    for Soft_Errors in range(0, Total_CW_Bits-Stuck_Bits+1):
      MISC_Rate += (Media_Fail_Prob_Array[Stuck_Bits][Soft_Errors
           ]*Media_Fail2Misc_Prob_Array[Stuck_Bits][Soft_Errors]);
  return MISC_Rate;
```