FLOWER and FaME: A Low Overhead Bit-level Fault-map and Fault-tolerance Approach for Deeply Scaled Memories

Donald Kline, Jr†, Jiangwei Zhang†, Rami Melhem‡, and Alex K. Jones†‡
†*Electrical and Computer Engineering* ‡*Computer Science*
University of Pittsburgh, Pittsburgh, PA 15260 USA
Email: {dek61, jiz148, melhem, akjones}@pitt.edu

***Abstract*—To maintain appropriate yields in deeply scaled technologies requires fault-tolerance of increasingly high fault rates. These fault rates far exceed traditional general approaches such as ECC, particularly when faults accrue over time. Effective fault tolerance at such high fault rates requires detailed bit-level knowledge of the location of faulty cells. We provide a solution to this problem in the form of a space efficient, bit-level fault map called FLOWER. FLOWER utilizes Bloom filters to provide detailed fault characterization for a relatively small overhead. We demonstrate how FLOWER can enable improved fault tolerance at high fault rates by enhancing existing fault tolerance proposals and yielding 10–100× improvements. Using in-memory processing, FLOWER can maintain a <2% performance overhead at 10^{-4} fault rates with less than 2% loss of memory density to report bit-level faults with high accuracy. Using a tuned novel hashing technique called MinCI, FLOWER for memory achieves considerably lower false positives than with disk-level hashing techniques at a fraction of the performance overhead. With a new technique to protect against errors during in-memory operations, PETAL bits, FLOWER can remain resilient against random errors while efficiently targeting predictable errors. Furthermore, we propose a new fault tolerance scheme called FaME, which provides ultra-efficient bit-level sparing by using the FLOWER fault map to identify the location of faults. FLOWER+FaME can achieve 14× longer PCM memory lifetime with half the area overhead versus SECDED ECC.**

***Keywords*-Fault tolerance, Bloom filters, Hash functions, DRAM, Phase Change Memory, Processing in memory.**

I. Introduction

A critical limiting factor for computing performance remains the ability to move increasingly large data sets to and from the processor. As a result, the development of increasingly dense main memories to hold these growing working sets continues to be a focus for future computing systems. DRAM fabrication continues to push to smaller geometries while relying on increasingly complex multi-patterning lithography, which in turn leads to larger process variation. With continued issues related to capacitor scaling and smaller processes on the horizon, future memories will have to contend with further increasing process variation.

Unfortunately, increasing variation has important and non-negligible side effects that negatively impact memory reliability [1], [2]. Increased variation leads to more failed cells reaching a 10^{-4} fault rate at fabrication time [3], [4]. In DRAM higher charge variance leads to lower tolerable noise margins, increased coupling exacerbating crosstalk [4]–[9], and reduced retention times [9], [10]. In limited endurance memories, such as Phase Change Memory (PCM) [11], variation leads to faster wear-out.

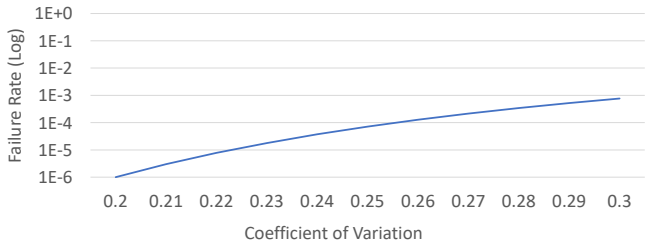

Figure 1. Change in cell failure rate with increasing CoV.

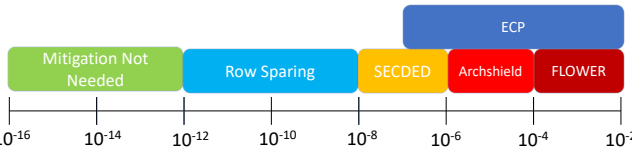

Figure 2. Fault mitigation requirements as fault rate increases, expanding upon a similar figure shown in previous work on memory fault-tolerance [3].

Figure 1 links variation and faulty cell rate. A popular assumption for process variation of recent technologies is a 0.2 coefficient of variation (CoV) [12], [13]. If the initial fault rate for 0.2 CoV was 10^{-6}, increasing the CoV to 0.25 or 0.3 increases the faulty cell rate to 10^{-4} or 10^{-3}, respectively.

Figure 2 qualitatively compares fault rate versus practicality of different error correction strategies, while Figure 3 quantitatively shows the area overhead required to achieve a 10^{-12} uncorrectable bit error rate (UBER). At cell fault rates $<10^{-12}$ no error correction is necessary. Between 10^{-8} and 10^{-12}, incorporating a small number (*i.e.,* $< 1/84{,}000$) of spare memory rows is sufficient to replace any row that contained a faulty cell with a fault free spare row [3]. Between 10^{-6} and 10^{-8} word-level 64,72 SECDED (single error correction, double error detection) ECC is more space efficient than row sparing. Up to 10^{-4}, word-level fault maps and spares (*e.g.,* ArchShield) become necessary to be space efficient [3]. Exceeding 10^{-4}, word-level replication becomes impractical. The sharply increasing space overhead of 23% and 92% at 10^{-3} and 10^{-2} fault rates, respectively, for ArchShield

2378-203X/20/$31.00 ©2020 IEEE
DOI 10.1109/HPCA47549.2020.00037

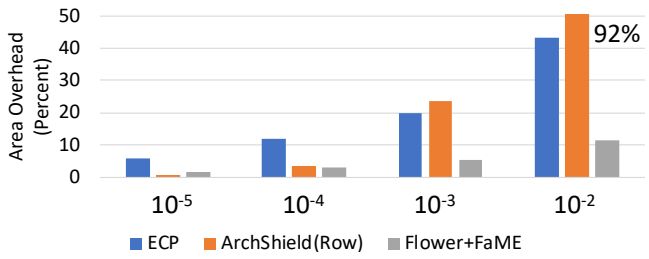

Figure 3. Faultmap area overhead to achieve a 10^{-12} effective fault rate for different initial fault rates.

indicate bit-level strategies are necessary. Error correction pointers (ECP) [14], a form of bit-level fault map and spares, perform better than word level replication, which is in an encouraging result for bit-level fault tolerance, but its space overhead is still prohibitive.

In this paper, we propose FLOWER (Fine-grained, Low Overhead Weak- and Error-prone-cell Registry), a space and energy efficient bit-level fault map to enable main memory fault tolerance in the presence of high fault incidence rates (*i.e.*, $\geq 10^{-4}$). FLOWER uses Bloom filters for its implementation to ensure that it is both space efficient and has a low access latency. While FLOWER can report a small number of "phantom faults" or false positives due to overlapping of the hashing functions, FLOWER is guaranteed to report all "true faults." We demonstrate that the number of false positives are easily minimized through effective multiple dimension hashing while maintaining the same storage overhead. Further, leveraging processing-in-memory techniques, we demonstrate how FLOWER can efficiently construct fault vectors from the multi-dimensional fault map using the two most prominent instances of main memory available: traditional DRAM and newly-released PCM (Optane Persistent Memory) DIMMs [15].

We also demonstrate a new fault tolerance technique enabled by FLOWER called **Fa**ult **M**ap **E**nabled (FaME) error correction. FaME distributes spare storage cells as small numbers of auxiliary bits for each memory row. For each memory access, FLOWER reports the location of the faulty cells in that row and the FaME spare storage cells are used in place of those bit locations. This avoids requiring pointers to the fault locations within the row (ECP). FaME also naturally protects auxiliary bit failures by including the auxiliary bits in the fault map and sparing faulty auxiliary bits. In contrast, ECP requires two pointers to correct one failure in the auxiliary bits. As shown in Figure 3, FLOWER with FaME has a significant advantage in space efficiency over ECP.

Recent studies of commodity DRAM have demonstrated potential fault rates reaching 10^{-4} [4]. FLOWER and FaME provides the same 10^{-12} uncorrectable bit error rate (UBER) with a 53% lower area overhead compared to ArchShield at this fault rate. Moreover, endurance limited memories like PCM may experience even higher fault rates. FLOWER and FaME is scalable to a 10^{-2} fault rate using 12.5% area overhead, a respective 3× and 7× reduction compared to ECP and Archshield, as shown in Figure 3.

In this paper we provide the following contributions:

- We propose FLOWER, a compact, efficient, bit-level fault map to protect main memory, which supports dynamic updates and scales efficiently at higher fault rates of current and future deeply scaled technology.
- We propose *minimum-cumulative intersection*, a tuned hashing technique, which has lower collisions for moderately dense data than traditional (*e.g.,* disk-level hashing) techniques along with better performance.
- We show how FLOWER can use processing-in-memory to construct fault vectors from multiple dimensions to minimize memory traffic, and propose novel *PETAL bits* for error correction of in-memory operations.
- We propose FaME, a new error correction strategy that leverages the FLOWER bit-level fault map to dramatically reduce storage overheads while maintaining high fault tolerance for high fault rates.
- We provide area, fault tolerance, lifetime, and performance evaluations for FLOWER and FaME compared to state-of-the-art fault-tolerance mechanisms.

II. Background

In this section we discuss background on Bloom filters; the origin, variety, and tolerance of faults for conventional DRAM and emerging PCM; and prior work on fault maps.

A. Bloom Filters

Bloom filters [16] improve space and time efficiency of checking set membership. Their application includes search engines [17], DNA sequencing [18], and packet classification [19]. Bloom filters typically appear in the form of a bit array with a hash function, where an item hashed to the corresponding bit in the array was set to '1.' Multi-dimensional Bloom filters use multiple hashes where each hash location in the array is set to '1.' On checking set membership, the bit corresponding to each hash function is accessed, and only considered a member if each hashed value returns '1.' Because Bloom filters reduce the amount of stored information, there is a chance to categorize a non-set member as a member. While this situation, referred to as a *false positive*, may occur, the opposite case, where a member is incorrectly classified as not being in the set (or a *false negative*), cannot occur. While Bloom filter construction (adding set members) is simple, items cannot be removed without outside knowledge.

Many Bloom filter modifications have been developed to suit particular applications [20]–[22]. In addition to adjusting the Bloom filter structure, many studies and enhancements have been performed on their hashing algorithms. The goal of a multidimensional Bloom filter hash algorithm is to produce independent random variables. This minimizes the risk to hash into the same bin and ultimately, the

occurrence of false positives. While traditional cryptographic hash functions such as SHA-2 work well for this purpose, faster non-cryptographic hash functions which approximate k-independence are often sufficient, such as Pearson's hash [23] and MurmurHash, the latter of which is used in Hadoop [24].

B. Memory Failures and Mitigation

In this section we inspect DRAM and PCM errors and the potential of FLOWER to assist in their fault tolerance.

1) DRAM: Certain DRAM *victim cells* experience word-line crosstalk faults when their stored charge is drained prematurely due to the repeated charging of adjacent word-lines without being refreshed. Studies have demonstrated that as few as 30,000 accesses to adjacent rows can manifest in an incorrectly stored data bit [2]. Proposed solutions include ECC, counters [25], probabilistic refreshing [2], more frequent refresh, and protection of weak (prone to crosstalk) DRAM cells with use of a word-level fault map [3].

Bitline crosstalk manifests as fluctuations in the read bit values due to bitline voltage levels connected to the same sense amplifiers [7]. It can be observed in cases within a row where a weak cell and its two neighbors share the same charge [4]. The weak cell may flip and produce incorrect data [26]. Periodic Flip Encoding (PFE) has been recently proposed to encode data to avoid these problematic patterns. Knowledge of the location of weak cells from a fault map allows PFE to guarantee to correct three faults with only two auxiliary bits per block [27].

Weak cells can also manifest with reduced retention times. This hampers power saving efforts by reducing refresh intervals [28], [29] or requires additional error correction [30]–[32]. While significant work has addressed each of these problems, many of the correction schemes developed for one fault either directly exacerbate another problem (bitline twisting, increased refresh time) or require significant area overhead for correction (e.g., multi-bit ECC).

2) PCM: PCM has been proposed as a potential replacement of DRAM for main memory to gain storage density and save static power. It has recently gained commercial traction in 3D Xpoint (Optane) memory [15]. Other than the challenge of higher write energy, the primary concern with PCM utilization is the limited write endurance (circa 10^8 writes) before cells become permanently "stuck-at" a particular value. While these cells can be read, they can no longer transition from their current amorphous or crystalline phase [11].

Additionally, early failures of weak cells due to process variation will severely limit the useful lifetime of PCM. By extending the fault-tolerance capability from 10^{-4} to 10^{-2} using techniques like FLOWER and FaME, the lifetime can be increased by more than 100% for a coefficient of variation of 0.2. This improvement grows to 3.8× and 16.4× as CoV grows to 0.25 and 0.3, respectively [33].

C. Existing Fault Maps

ECP [14] is essentially a fault-map combined with spare bits distributed through row-level auxiliary bits. For stuck-at faults at low error rates, ECP is more efficient than ECC but sacrifices protection of transient faults. ECP has also been adapted for DRAM [32], [34]. ArchShield [3] is a word-level fault map combined with sparing proposed for DRAM. In its design, each row contains two bits to set whether the row has zero, one, or multiple faulty words in a row. A combination of error correction (when possible) and replication is used to maintain fault tolerance. Recently, SFaultMap, a bit-level fault map focused on minimizing energy and targeting sustainability [35] was proposed. SFaultMap is essentially a compact hardware sparse matrix representation but has a high performance penalty due to sequential searching of the map. To add new faults, SFaultMap requires either pre-allocating for new faults expansion per row or completely rebuilding the fault map for every new fault. This reduces its value to systems with endurance faults such as PCM.

III. The FLOWER Fault Map

FLOWER is a bit-level fault map which scales to extremely high fault rates to enable protection of bits which are either susceptible to faults (*i.e.,* weak) or have permanently failed. This allows FLOWER to be applied to many memory technologies and fault types. A particular strength of FLOWER is its relatively low area overhead compared to other techniques which protect against very high error rates due in part to its Bloom filter-based design. However, FLOWER provides several key innovations to the Bloom filter concept to make it suitable for high fault-rate memory instrumentation.

FLOWER is implemented using a Bloom filter and returns a *fault vector* that is the same width as the memory row. Each location with a fault is represented with a '1' and fault free cells with a '0.' In a conventional Bloom filter design a *unified array* is used to store each dimension of the filter. As such, a memory row addressed with N bits is mapped

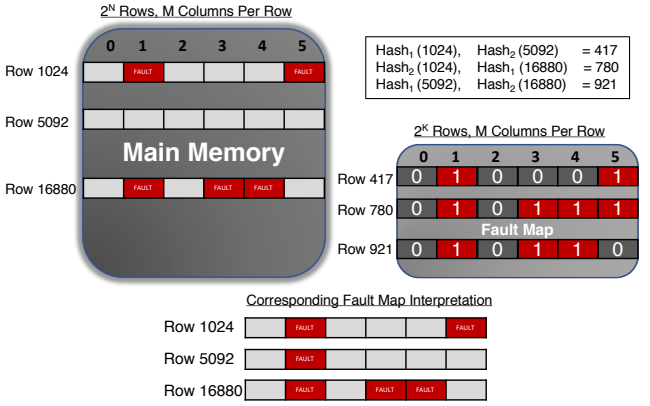

Figure 4. Two-Dimensional, unified-array FLOWER example. Red represents faulty cells. M is the length of each row, N are the bits in the address space for the memory, and k are the bits which result from the hash function

to an array of rows addressed by k bits, such that $k < N$. The Bloom filter uses a hash function to map each of the 2^N memory rows to the 2^k rows of the Bloom filter storage. Of course, as $k < N$, multiple memory rows will map to the same row in the Bloom filter storage. Thus, faults recorded in the Bloom filter will always be reported for the row in which they exist, but false positives will also be inadvertently reported for other rows that map to this location.

To minimize false positives with d-dimensional Bloom filters, d independent hash functions map each row to d different locations, effectively superimposing d Bloom filters into a unified array. For a fault to be reported, all hash functions, each of which returns a *partial* fault vector (PFV), must agree the fault is present in the row into which each hash is mapped. False positives are dramatically reduced, although not entirely eliminated, using this method.

A two-dimensional example of this mapping to faults is shown in Figure 4. Row 1024 has faulty bits at columns 1 and 5 and row 16880 has faulty bits at columns 1, 3, and 4. Row 1024 is translated to rows 417 and 780 by hash functions one and two, respectively. Thus, for these rows, bits 1 and 5 are set to '1.' Row 16880 maps to row 780 and 921 for hashes one and two, respectively. In these rows, bits 1, 3, and 4 are set to '1.' In general, for a d-dimensional Bloom filter, each faulty cell will result in d cells set to one in the Bloom filter, one for each hash.

In the example, row 5092 in the memory has no faults, but maps to rows 417 and 921 in the Bloom filter. Because bit position 1 is set in both of these rows in the filter, when accessing this row in the fault map, it reports a false positive: that location 1 is faulty due to the collision of data from rows 1024 and 16880. For a d-dimensional fault map, fault vectors for a given row are constructed through the `AND` function of the PFVs accessed from each of the d hash functions in the filter. Note, the faults for rows 1024 and 16880 are still reported correctly. The prevalence of false positives directly correlates to the scalability and effectiveness of FLOWER; thus we will discuss techniques to minimize them in the next section followed by a discussion of the FLOWER architecture along with an in-memory processing extension in Section III-B.

A. MinCI: A Tuned Hash for FLOWER

The application of a Bloom Filter for constructing a fault map that reaches high fault rates can create more dense structures than commonly employed in other applications. Thus, it is important to develop tuned hash functions that maximize the amount of information retained in the hashes to minimize collisions in the filter. Moreover, there are several opportunities to tune the hash function to support architecture implementation. First, dividing the d-dimensional Bloom filter storage into d smaller, independent subarrays, one for each of the d hash functions, has some advantages for access latency through parallelism. Second, employing simpler hashing functions can further streamline the implementation of FLOWER. In this section, we discuss these techniques to adjust our Bloom filter implementation to be more efficient.

In order to motivate our approach we simulated several fault maps with different storage capacities to study the use a unified array versus multiple arrays, one for each dimension. Maps of faulty cells were created using normal distributions to mimic the impact of process variation and include spatial correlation of faults [36] (more details in Section VI-B).

First we consider the impact of using a Bloom filter using Murmur hashes with separate subarrays. Using a block-based partitioning where for a d-dimensional fault map, each dimension was allotted $1/d$ the storage space as the unified array. The comparison of unified vs separate subarrays for the Murmur hash in a FLOWER fault map are show in the blue and orange bars, respectively, of Figure 5. The results show that multiple independent arrays performed at least as well as the unified array often reducing false positive rates. Moreover, this allows partial fault vectors for different hashes to be stored in separate sub-arrays in-memory which can support parallel combination (*e.g.,* using in-memory processing).

Murmur hashes, essentially pseudorandom hash functions, were designed to minimize collisions in the unified array Bloom filter. In separate arrays, the d hashes no longer have to consider collisions/overlap with one another to minimize false positives. They can instead focus on collectively retaining the most information possible. Moreover, while simple for software, the most efficient Murmur hash implementations still require approximately 50 instructions *each time* the address is hashed. This is a significant overhead for a hardware implementation. Thus, to simultaneously minimize the complexity of the hashes while maximizing the information retained by the filter we propose a *minimum cumulative intersection* (MinCI) of the address space.

Our MinCI hashes are designed offline for a particular memory size and fault map overhead and are implemented at runtime by simple address bit masking. We define MinCI for a given number of dimensions (d), address range (N), and hash size ($h = k - log_2 d$) to obey the following properties. Each bit in the address space is used a (or $a-1$) times. Each hash shares b (or $b-1$) bits with each other hash. Each hash shares c (or $c-1$) bits total, across all hashes. a, b, and c are minimized given the particular d, N, and h. These conditions for the minimum cumulative intersection can be intuitively explained using an overlap matrix, which shows how many times each hash overlaps with each other hash. An example of an overlap matrix is shown Figure 6. Each cell contains the $b_{i,j}$ value corresponding to the intersection of two hashes specified by the matrix row and column, while c refers to the sum of overlap for a particular hash along each column (or row). When the conditions on a, b, and c are met, the sum of the elements of the overlap matrix will be minimized

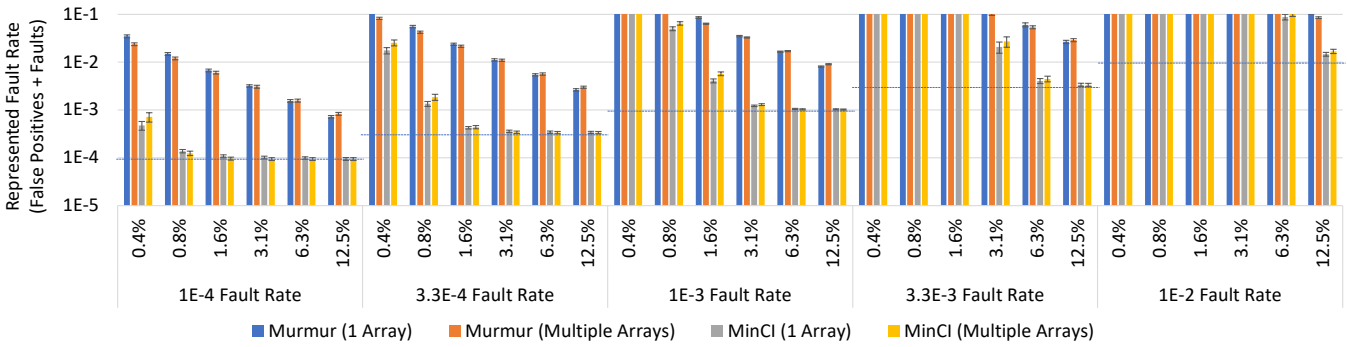

Figure 5. False positive rates for 4D fault maps with different storage overheads and initial fault rates.

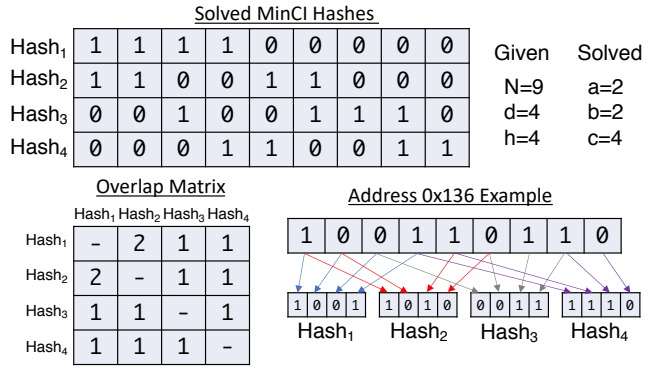

Figure 6. 4 masks for MinCi hash functions assuming $N = 9$ address bits, $d = 4$ dimensions, and $h = 4$ hash bits.

and equal to the value given by Eq. 1:

$$overlap_{minci} = 2 * \sum_{i=1}^{d-1} H(h*d - i*N) * (h*d - i*N) \quad (1)$$

where H refers to the Heaviside function [37]. Once MinCI masks are computed for a particular N, d, and h, they can be used for any system which uses the same N, d, and h.

A simple MinCI example is also shown in Figure 6. In this example, $d = 4$, $N = 9$, and $h = 4$. Each of the N bits is used by either one or two hashes, thus $a = 2$. Hash_1 shares two bits with Hash_2, and one bit with both Hash_3 and Hash_4, for a total of four bits shared. Hash_{2-4} have similar overlaps such that b=2 and c=4. The gray and yellow bars in Figure 5 show the false positive rate for the MinCI for both single (unified) and multiple arrays, which indicate the MinCI approach considerably reduces false positives over Murmur hashes. The multiple arrays version of MinCI degrades slightly over the unified array; however, the difference is small compared to the improvement of MinCI over Murmur. Thus, using the multiple array version of MinCI allows for several efficiency optimizations in the memory architecture while providing effective false positive minimization for FLOWER.

B. FLOWER Architecture

FLOWER uses the memory it is protecting to store the fault-map. An example of this is illustrated in Figure 7, where FLOWER is stored in one bank of a DRAM chip,

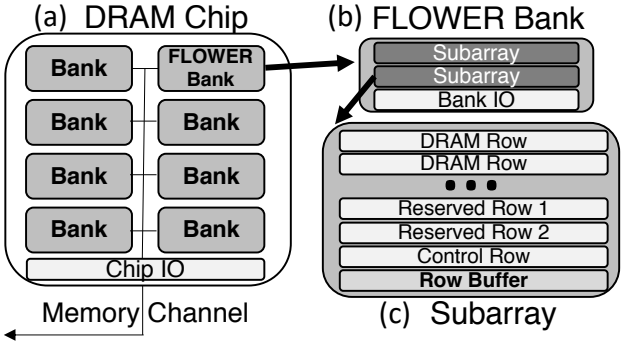

Figure 7. DRAM Organization for FLOWER using in-memory logical operations.

protecting the other banks in the chip. Efficiently storing and accessing FLOWER directly within the memory also requires several adjustments to the memory system architecture. A traditional solution for implementing d dimensions requires d memory reads, each retrieving a PFV corresponding to each hash function. These PFVs can be combined with an `AND` operation directly at the memory controller. These fault-map memory accesses are a considerable overhead for each memory access; therefore we propose an in-memory approach to reduce this overhead where possible for both DRAM [38] and PCM [39]. In our experimental evaluation we compare the performance of both memory controller and in-memory based combination of FLOWER PFVs.

1) In-Memory FLOWER Access for DRAM: To reduce the latency and energy of multiple DRAM accesses on the memory bus we leverage in-memory cloning and logical operations, proposed by Ambit [38]. A row can be cloned if the row is read into the row buffer and then a second row is activated as the row buffer will override what is stored in the row [38]. Logical operations can be conducted by activating two rows simultaneously, along with a third *control* row [38]. Along each bit line the voltages are combined such that if $\geq$2 of the three rows contains a V_{DD} the voltage will be driven to V_{DD} and if $\leq$1, the value is driven to V_{SS}.

Reading from the in-DRAM FLOWER fault-map is shown in Figure 8. First in steps 1 and 2, the two DRAM rows corresponding to PFVs from the first two hash functions, H_1, H_2–in the example rows `A1`, `A2`–are cloned to reserved rows `R1` and `R2`. In step 3 the control row `Ctrl` is initialized to

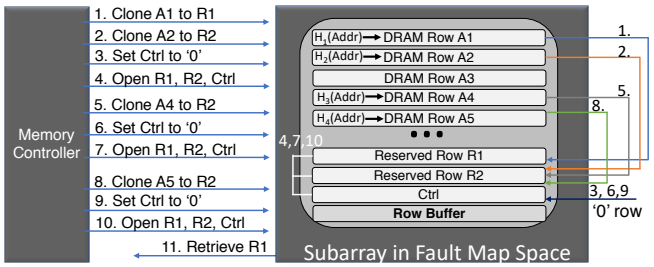

Figure 8. In-memory 4D FLOWER reading example.

'0's. In step 4, rows `R1`, `R2`, `Ctrl` are opened simultaneously to conduct the `AND` operation. Note the result of the `AND` operation is left in all three rows and the row buffer [38]. Subsequent `AND` operations can be accomplished by cloning the next dimension's (*e.g.,* H_3) PFV, row `A4` in the example, into one of the two reserved rows (*e.g.,* `R2`) [Step 5], resetting the '0' row [Step 6], and completing the `AND` operation [Step 7]. After Step 10, the complete fault vector from a 4D map is available in the row buffer and can be returned to the memory controller [Step 11]. For d dimensions, this method requires d cloning, $d-1$ `AND`, and $d-1$ `zeroing` operations.

When applying MinCI in separate arrays to FLOWER with in-memory operations for the example in Figure 8, steps 1-4 for dimensions 1 and 2 can be completed in parallel to similar steps for dimensions 3 and 4 in independent subarrays. Then using similar steps to 5-7 the partial fault vector of dimensions 1 and 2 can be combined with the partial fault vector for dimensions 3 and 4 and steps 8-10 can be eliminated. For d dimensions this reduces the depth of computation from d-1 `AND` operations to $\log_2 d$ `AND` operations.

2) In-Memory FLOWER for PCM: In-memory logical operations in PCM can be completed by simultaneously opening multiple rows, as in the DRAM case, using the technique developed by Pinatubo [39]. In contrast to the DRAM in-memory operations, instead of using a control row, the result of the `AND` function can be obtained by shifting the sensing threshold. For example, standard PCM sensing tests between resistance (Ω) Ω_{high} and Ω_{low}. By opening two rows simultaneously each cell resistance is combined along the bitline and can be tested in parallel. By adjusting the threshold to be halfway between $\Omega_{low+low}$ and $\Omega_{low+high}$ allows computation of the `AND` operation [39]. Theoretically, more than two `AND` operations can be completed in a single operation [39]. One key advantage of PCM in-memory logical operations over DRAM is that no initial copying is needed prior to activating two rows directly. Thus, in-PCM memory access of FLOWER functions in a similar fashion to the in-DRAM access from Figure 8 with some simplifications due to the non-destructive read operation. Consequently, the process of receiving a FLOWER result from PCM simply requires $d-1$ comparisons. Similarly to DRAM, in-memory PCM bitwise operations can function across rows in different sub-arrays or banks on the same die with a longer latency.

3) Updating the Fault Map: While many faulty cells can be determined at the time of manufacturing and testing memory devices, endurance-based faults typically occur throughout the lifetime of the memory. PCM, Flash, and even DRAM can experience cells that fail due to use. FLOWER naturally supports adding new faults into the fault map. These faults can either be detected via the common read-after-write policy for non-volatile memories [40], or through complementary ECC designed for the correction of transient faults. Once a new fault is detected, the previous fault vector (fv) for that row is updated with the new fault (fv'). The Bloom filter is updated by replacing each hash function's partial fault vector (pfv_i) with $pfv'_i = fv'$ `OR` pfv_i. Similarly to Sections III-B1 and III-B2, this can be accomplished using in-memory `OR` operations, possible in both DRAM [38] and PCM [39]. However, when using FLOWER with volatile main memory such as DRAM, upon system startup, the DRAM map must be populated from secondary, non-volatile storage. Thus, to protect against resident data loss, any new faults added to FLOWER have a write-through policy, and must immediately write updated FLOWER addresses to secondary storage. While this might seem expensive, for all relevant fault modes new failures occur very infrequently with respect to the lifetime of the device in question. We quantify this infrequency of new failures in Section VI-C. Thus, fault map updates have only a minor, negligible impact on performance.

C. Implementation Details

The in-memory architecture, including combination of multiple partial fault vectors using in-memory logical operations is about 20% faster and upwards of 40× more energy efficient for DRAM [38], with potentially even higher speedups for PCM, compared to combining partial fault vectors at the memory controller. However, FLOWER access is still a significant overhead. Moreover, using a potentially faulty memory to store the fault map creates a challenge to ensure that the fault map itself does not accrue faults.

1) Improving Performance: Whether performing the join of FLOWER dimensions in-memory or at the memory controller, the cost of accessing the fault map for every address would be prohibitive, and unreasonable at low error rates. Thus, FLOWER implementations for low-fault rates and for faults which accrue over time should have an *address filter* at the memory controller. This address filter is a simple, small, and fast first-level Bloom filter that tracks which rows have at least one fault [34]. If a row does not hit in the address filter, the row is fault free and accessing the fault map can be skipped. This filter allows the fault map to avoid significant performance degradation through a 10^{-4} fault incidence rate. The 10^{-4} fault rate is an important threshold, as it is the tolerable limit for many existing schemes. Beyond this limit, the FLOWER fault map is still capable of effectively representing fault rates of as much as 10^{-2} where the first level filter is less effective. Detailed performance results

demonstrating the fault rate and performance tradeoff are presented in Section VI-C.

2) Storing FLOWER Reliably: Storing FLOWER within the memory it is protecting raises the question of reliable operation in a potentially faulty memory. For most cases, the special operation of the fault map makes it unlikely to experience these faults. Endurance-based faults and faults dependent on excessive write operations (such as wordline crosstalk in DRAM) can be safely ignored due to insufficient write activity in the fault map segment of the memory. Each cell is only ever initialized to '0' and written once to '1' to represent a faulty cell. Moreover, each row of the fault map is written *at most M* times (where *M* is the number of bits in the row) over the course of the memory's lifetime. Additionally, write faults can be mitigated through read-after-write. Memories with high vulnerability to read disturbance can use probabilistic refresh [2] or counter-based trees [25] to protect FLOWER with nominal overhead. Also, FLOWER could be protected by manufacturing the FLOWER region with a less-aggressive fabrication process, as is assumed in previous fault-mitigation strategies [41], [42].

Furthermore, a FLOWER implementation that combines partial fault vectors at the memory controller can use ECC for protection. However, when electing to use in-memory logical operations for FLOWER, these operations may introduce transient faults and cannot be directly protected by ECC [38], because the ECC correction is not preserved over the AND operation required to combine the Bloom filter dimensions together. Our FLOWER-specific solution to this problem will be discussed in the context of FaME in Section V-B2. The next section describes how FLOWER can be used to support existing fault tolerance techniques for various types of technology specific faults (*e.g.,* endurance faults) or those due to deep technology scaling (*e.g.,* crosstalk).

IV. FLOWER for Fault Tolerance

There are several fault tolerance techniques [3], [27], [41]–[45] that assume *a priori* knowledge of fault locations in order to provide fault tolerance. Most of these techniques assume a runtime method to detect the faults (such as read after write) [46] and possibly employ a fault cache to avoid adding these expensive tests [4], [42]–[44]. ArchShield uses a *word*-level fault map [3]. Unfortunately, for high fault rates, fault caches and word-level fault maps are ineffective. Additionally, read after write tests are often unreliable as faults can be probabilistic or instigated by unrelated accesses. Moreover, they can accelerate further cell failures in the case of endurance failures. Thus, FLOWER provides an alternative which we discuss in the context of two examples, bitline crosstalk in DRAM and stuck-at faults in PCM.

A. DRAM Bitline Crosstalk

Recall from Section II, bitline crosstalk in DRAM occurs probabilistically when a "weak" cell and its surrounding cells are written with the same charge ("111" or "000"), which we refer to as "bad patterns." These weak cells are discoverable through system testing [4] and can be used to populate the FLOWER fault map. Periodic Flip Encoding (PFE) [27], is a technique to mitigate these bad patterns and significantly benefits from knowledge of weak cell locations. As shown in Figure 9, PFE either flips the first, second, or third bit (or no bits) of each 3-bit grouping to break up bad patterns. Red cells show a bad pattern overlapping with a weak cell (high probability for a fault) and gold show a bad pattern without a weak cell (no fault). PFE successfully avoids the fault in the (c) "10" encoding even though some bad patterns exist in the encoding, due to the knowledge of the location weak cells from the fault map. The authors of PFE specifically discuss the need for a bit-level fault map for fault-aware protection [27], which provides an $100,000\times$ improvement over fault-oblivious protection at a 10^{-4} fault incidence rate.

B. PCM Stuck-at Faults

Yielding optimized dependability assurance (YODA) [45] is a technique to use stuck-at fault information to extend ECP from correcting p to $2p+1$ faults[1], where p is the number of pointers, by adding one bit for inversion. YODA matches data to faults to determine stuck-at right (SA-R) or wrong (SA-W) and points only to the SA-W values. YODA dynamically writes rows of '0's and '1's to discover the faults, significantly degrading lifetime (between $2\times$–$3\times$).

FLOWER can keep separate maps for different errors, such as stuck-at '1' and stuck-at '0' faults, which can avoid lifetime reduction. However, in this instance, a single fault vector of generic stuck-at locations along with the data currently stored is a sufficient, lower overhead method to identify stuck-at '1' and stuck-at '0' faults. Nonetheless, the ability to segregate types of faults that must be handled differently, such as stuck-at faults and cells susceptible to write disturbance in PCM [47], is a useful capability of FLOWER.

V. FaME Error correction

In Section IV, we discussed how FLOWER can enable and enhance previously developed fault-tolerance techniques. Most importantly, however, FLOWER enables *new* fault-tolerance techniques which otherwise would not be possible.

[1]This is classified as YODA_1 [45].

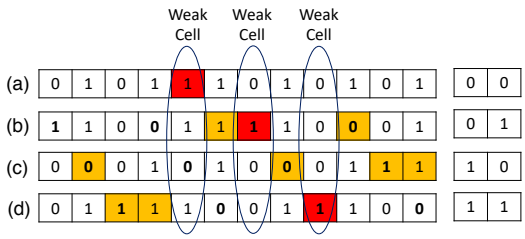

Figure 9. Encoding "0x5D5" into a 12-bit (11...0) with weak cells as positions 7, 5, and 3, using PFE.

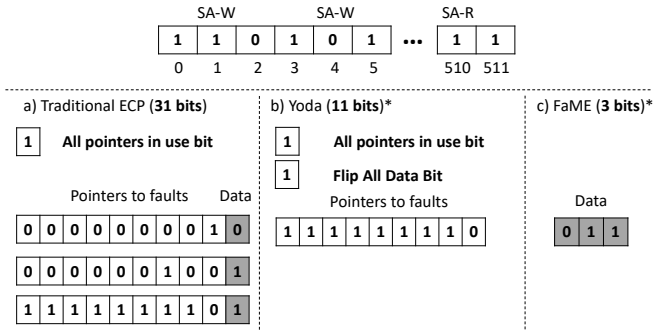

Figure 10. Auxiliary bit example for a 512-bit PCM row for a) traditional ECP, b) Yoda, and c) FaME. SA-W and SA-R refer to stuck-at-wrong and stuck-at right cells. FaME and Yoda overheads do not include the fault map.

In this section, we propose a new technique for correcting stuck-at faults in PCM: **Fa**ult **M**ap **E**nabled Error Correction.

A. FaME Design

In FaME, f bits are added to each memory row to provide protection against f faults. Instead of requiring f pointers, FaME utilizes the FLOWER fault vector, utilizing spare bits for each reported fault location, *in order*. A comparison of the auxiliary bit overheads to protect three faults for ECP, YODA, and FaME are shown in Figure 10. Traditional ECP requires 31 ECP bits per 512-bit row to protect three faults: 3×9 pointer bits, three spare bits, and one flag bit signifying all pointers are used. ECP does not require knowledge of SA-R or SA-W. YODA requires only 11-bits to correct three faults: one ECP pointer, or nine pointer bits, one flag bit, and one inversion bit (see Section IV-B). YODA also requires knowledge of SA-'1' and SA-'0' fault locations. FaME requires only three auxiliary bits to protect against three faults[2]. FaME requires knowledge of fault *locations*, but does not distinguish between SA-'1' or SA-'0.'

Because FaME only requires f bits per row to correct f faults per row, there are several substantial advantages over previous fault tolerance implementations, such as ECP. The primary advantage is that FaME combined with FLOWER can achieve a substantial area savings over ECP. For example, if a 3.13% area overhead FLOWER fault map is used, six fault protection per row requires an additional area overhead of $^{6}/_{512}$ or 1.2% for a total overhead of 4.3%. In contrast ECP requires 61-bits per row for a $^{61}/_{512}$ or a nearly 12% overhead. Even if the nominal lost fault tolerance from false positives is accounted for, FaME still provides substantial benefits in area with iso-tolerance, fault tolerance with iso-area, or both improved fault tolerance and area.

Faults in the auxiliary bits are protected in same way data faults are represented, by extending the fault vectors with auxiliary bits in the fault map. Thus, FaME can discover if an auxiliary bit has a stuck-at fault. In this case, the auxiliary bit identified as faulty will be skipped, such that FaME with f auxiliary bits protects against f faults reported by FLOWER across both the data and auxiliary bits.

[2]FaME works properly if FLOWER reports false positives, but each false positive will consume a spare bit and limit the fault tolerance capability of FaME.

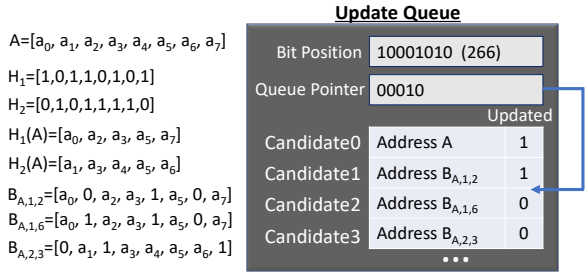

Figure 11. FaME update cache.

B. Accumulated Faults

FaME's dependence on fault vectors for its encoding and decoding operations allows it to use extremely small auxiliary bit overheads. This dependence causes complications in systems where faults accrue over time. Before updating a row's partial fault vectors, as discussed in Section III-B3, data rows whose fault vectors will be affected by adding the new fault must have their auxiliary bits updated, even if the new fault is not a true fault (*e.g.,* a false positive).

MinCI with multiple arrays allows some simplifications, not possible with Murmur hashes, to allow discovery of all the rows which might require updates. For example, consider an 8-bit address (n=8), with a d=2 (2D) fault map that requires a 25% overhead (k=5). Assume address A=[a_0,a_1, ...a_7] has a hash function $H_j(A) \rightarrow [h_{j0},h_{j1},...h_{j4}]$. Due to the properties of MinCI, $H_j(A)$ essentially strips $n-k$ bits—in this case arbitrarily bits at positions 1, 4, and 6—to form a vector [a_0,a_2,a_3,a_5,a_7]. Assuming a vector U that contains all possible permutations of $n-k$ bits, vector B_{Aj} can be created from $H_j(A)$ and U, such that B_{Aj}= [a_0,u_0,a_2, a_3,u_1,a_5,u_2,a_7]. Rows other than $u_0 = a_1$, $u_1 = a_4$, and $u_2 = a_6$ are the "conflicting rows" from the hash. Determining these conflicting rows for all d hashes produces the pool of memory rows that may require updates to their FaME bits. The size of the pool is $d2^{n-k}$. However, the number of writes can be substantially reduced after checking the current partial fault vectors for a potential candidate. Address B_{Aj} only requires updates if, for the new fault position p and for all j, bit p of the partial fault vector addressed at $H_j(B_{Aj})$ is '1' or $H_j(B_{Aj})$=$H_j(A)$.

1) Memory Update Queue: To avoid memory stalls, any rows which satisfy the aforementioned conditions are added to an update queue shown in Figure 11, while memory accesses continue. The queue stores the bit position to update in the fault map and a list of update candidates. Each update candidate entry stores the address and a flag bit to indicate whether the address is updated (re-written with the corrected auxiliary bits) in memory. If an address in the update queue is accessed and the auxiliary bits have been updated in memory, the fault vector from the fault map is updated with the stored bit position. Otherwise, the access proceeds using

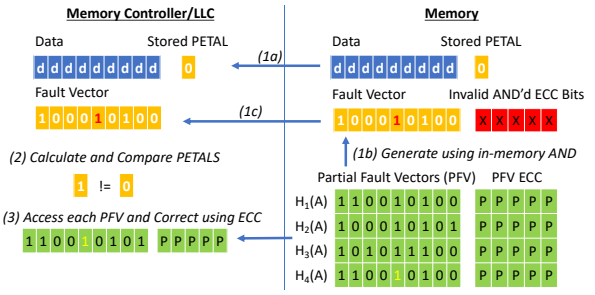

Figure 12. (1) Data/FLOWER access, (2) PETAL comparison, and (3) Error correction on petal detection.

the unmodified fault vector. Once all the candidates in the update queue have been written, the fault vector for row A is updated with the procedure in Section III-B3 and the queue is flushed. In the rare case a second fault is discovered, for example in row C, prior to completing the first fault update, the write to row C stalls until the update to row A is complete.

2) PETALs: In-Memory FLOWER Correction: Revisiting error detection and correction for FLOWER in the context of protecting the fault map against transient faults (such as errors during in-memory AND operations), the FaME update queue can also be used to support a fault tolerance structure for the in-memory version of FLOWER. We add a FLOWER PETAL (Parity Enabling Tolerance for Accelerating Logic-in-memory) bit to each memory row (not just memory rows storing FLOWER) requiring a $1/512$ or 0.2% overhead, as shown in Figure 12. The PETAL bit stores the parity of the number of faults for the *corresponding FLOWER fault vector* to that row. If there is a fault accessing a *partial* fault vector, it *may* appear as a false positive ('0' read as '1' where all other hashes are '1') or a false negative ('1' read as '0' and all other hashes are '1'). The PETAL bit will detect one fewer or one additional reported fault, which triggers individual access of the partial fault vectors. These vectors can be corrected using their ECC at the memory controller. Note that the in-memory AND result of the ECC bits (shown as 'X's in red) is unusable [38] and discarded. Furthermore, multiple rows may require updates to the PETAL bit when a new fault is added to FLOWER. This follows the same process used to identify the rows requiring FaME bit updates.

The UBER of using PETALs to protect in-memory partial fault vector combination is shown in Table I and compared with using SECDED ECC at the memory controller across different initial fault rates. Our goal was for the PETAL approach to match ECC-1 when combining fault vectors at the memory controller combination. PETALs provide slightly higher error protection making the in-memory access approximately as safe as ECC at the memory controller.

VI. Results

In this section we evaluate the effectiveness of FLOWER and FaME compared to existing fault tolerance schemes.

Table I
UBER for in-memory combination protected with PETALs versus memory controller combination protected with ECC-1 for a 4D FLOWER fault map.

	10^{-10}	10^{-8}	10^{-6}	10^{-4}
PETAL	$2{\cdot}10^{-19}$	$4{\cdot}10^{-16}$	$4{\cdot}10^{-12}$	$4{\cdot}10^{-8}$
ECC-1	$6{\cdot}10^{-19}$	$6{\cdot}10^{-15}$	$6{\cdot}10^{-11}$	$6{\cdot}10^{-7}$

In particular, we present simulations of error rate analysis, lifetime improvement in endurance limited memories (*e.g.,* PCM), and performance impact comparisons.

We first conducted a sensitivity study on the false positive rates when using one, two, four, and eight fault map dimensions for both Murmur and MinCI hash functions. In general we found that eight dimensions best minimized false positives while one and two dimensions performed significantly less well. However, four dimensions performed nearly as well as eight, while reducing the number of in-memory logical operations, making four the best compromise. A comparison of Murmur and MinCI hashing for four dimensions is shown in Figure 5. The figure also shows at which area overhead point the adopted MinCI hash achieves fault reporting rate with minimal false positives—*i.e.,* approaching the stated incidence fault rate indicated by a dashed line. Thus, unless otherwise specified, further results are reported for 4D MinCI hashes targeting separate arrays.

A. FLOWER for Enhanced Fault Tolerance

To begin, we analyze the improvement FLOWER-augmented fault tolerance provides to uncorrectable bit error rate (UBER). We focus on the two cases from Section IV, Bitline Crosstalk Correction in DRAM, and Stuck-at Fault Correction in PCM.

1) Bitline Crosstalk Correction (DRAM): Figure 13 provides an iso-area study of the PFE algorithm [27] with and without FLOWER for a faulty cell rate of 10^{-3}. It also provides a comparison with a perfect "ideal" fault map for the same instantiations of the FLOWER fault map. The PFE labels indicate the encoded block size. For example, PFE16 adds two auxiliary bits for each 16-bit block resulting in a 12.5% overhead, with PFE32, PFE64, and PFE128 requiring 6.25%, 3.13%, and 1.56% overhead, respectively. When utilizing FLOWER, half of the auxiliary bits are repurposed for storing a fault map of the same size. For example, the 4D fault map for PFE32 uses 6.25% overhead to encode faults and 6.25% to store a fault map, making it comparable to PFE16 without a fault map.

For 12.5% total area overhead, fault-aware PFE32 with FLOWER outperforms fault-oblivious PFE16 implementation by $>125\times$. Comparing the achievable correction capability of FLOWER to an perfect (100% overhead) bit-level fault map, PFE32+FLOWER is within 50% of the ideal fault map with over $8\times$ less area. At 6.25% total area overhead, PFE+FLOWER has a similar advantage over no fault map and is close to the ideal map. At the more coarse granularity

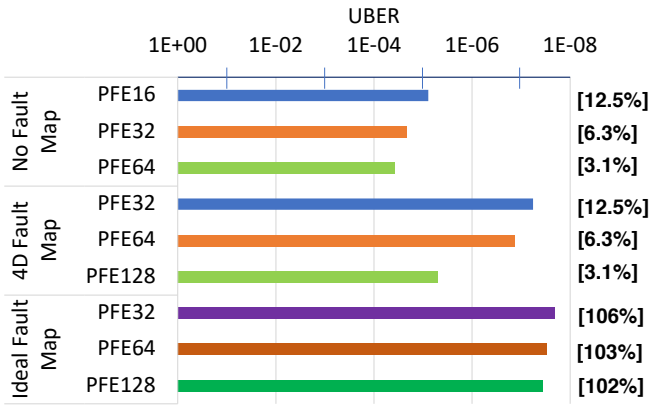

Figure 13. Iso-area PFE correction capability at 10^{-3} incidence weak-cell rate. Total overhead including fault map and encoding bits shown in "[]."

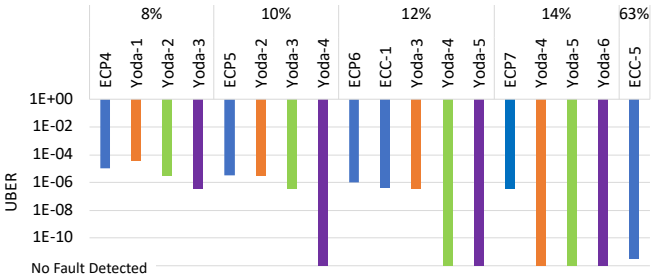

Figure 14. Iso-area analysis of UBER for ECP with FLOWER, ECP's extension Yoda with FLOWER, and ECC-5 per 64-bit word at 10^{-3} fault incidence rate. Colors indicate different fault map allocations: Orange: 6.25%, Green: 3.13%, Purple: 1.56%, Blue: 0%.

dictated by 3.13% total overhead, PFE+FLOWER provides less than an order of magnitude advantage. In this case, the loss of fault tolerance seems to originate from significant reported false positives from FLOWER, as using an ideal fault map would improve this result by two orders of magnitude.

2) Stuck-at Fault Correction (PCM): Figure 14 demonstrates the benefits of adding FLOWER to fault pointers for permanent faults for a 10^{-3} fault incidence rate. The areas are set approximately equal by replacing some pointers with two fault maps (SA-'1's and SA-'0's) and an additional "flip" auxiliary bit (see Section IV-B). For example, ECP4 dedicates four pointers and spares (8% overhead) but can only handle four total faults. YODA uses one, two, and three pointers with two 6.25%, 3.13%, and 1.56% overhead FLOWER maps, respectively, while having the ability to handle three, five, and seven reported faults, respectively. Total fault tolerance overheads are varied from 8% to 14%. At 8% area overhead, YODA-2 and YODA-3 with FLOWER are superior to ECP4, with the latter nearing two orders of magnitude improvement. By 10% area overhead, YODA-4 with FLOWER was sufficient to correct all possible faults in our SPEC [48] benchmark simulations, which without FLOWER requires ECP9. Thus, at this fault rate YODA (unlike PFE) is tolerant of FLOWER's false positives, making lower investments in fault map size and higher investments in auxiliary bits appropriate for this scenario. In contrast, ECC-5 per 64-bit word resulted in less tolerance (UBER higher than 10^{-12}) while requiring a much larger area overhead.

B. FaME Lifetime Improvement for PCM

To determine the impact of YODA+FLOWER on the lifetime of PCM memory we conducted experiments on memory traces to a 1MB memory segment (16K 64-byte rows), consistent with previous studies [43], [45]. Our custom simulator determines the expected lifetimes of each cell in the memory based on a probabilistically determined failure map. The failure map is computed from the mean lifetime (10^8 writes) and coefficient of variance (CoV) according to a normal probability distribution with spatial correlation of faults [36], [49]. The memory traces averaged over 12 SPEC benchmarks using the largest datasets (SUITE), synthetic traces simulating thrashing between two memory rows (THRASH), and perfectly uniform wear-leveling (LEVEL) were run in the simulator for 20 different failure maps[3] at CoV=0.2,0.3. During each experiment, once the number of writes to a cell exceeds its lifetime, dictated in that particular failure map, it is considered stuck-at its last recorded value. Once two rows in the 1MB memory segment cannot be corrected, we consider the memory to have failed and reached the end of its life.

Figure 15 shows the lifetime results for iso-area (12.5%) fault tolerance area overhead for different correction schemes with and without FLOWER, all compared against the significantly higher 63% storage overhead ECC-5 as a baseline. As expected, because of the high CoV, the system fails much earlier than the mean cell failure rate of 10^8 writes. For the SPEC benchmarks, on average, "No Correction" fails before the 10^6th write to the 1MB memory segment for a 0.3 CoV. ECC1 and ECP6 improve this correction substantially, to close to 10^8 writes to the memory segment before failure.

Assuming a CoV of 0.2 and dedicating 1.56% area overhead for the fault map(s), YODA-5 extends the lifetime compared to ECP6 by 4×. With the same encoding overhead, FaME can provide 51 bit spares and extends the lifetime by 17× and 14× over ECP6 for CoV of 0.2 and 0.3, respectively. Comparing to larger fault maps (6.25%) at the expense of fewer auxiliary bits, we see an interesting behavior. For SUITE, YODA achieves a longer lifetime with more pointers and smaller fault maps (1.56%), while FaME achieves a longer lifetime with larger maps (6.25%) and fewer auxiliary bits. This is because FaME benefits from the drastically reduced false positives, which more than make up for the 20 fewer spare bits. For LEVEL at 0.2 CoV, FaME31 with the larger FLOWER fault map improves lifetime versus no correction by 2,895× and 28× compared ECP6. This improvement can be treated as an upper bound of the lifetime improvement for the system. For THRASH, the benefit is,

[3]20 maps were deemed to have sufficient coverage as the difference in results after including more than 10 maps was insignificant.

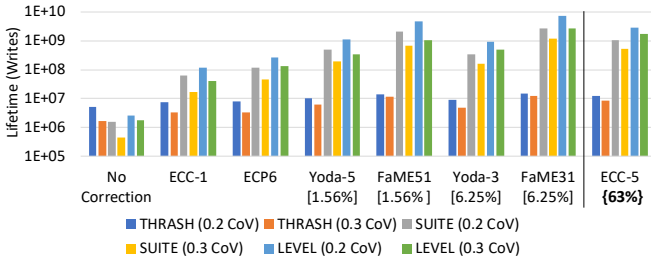

Figure 15. Lifetime to 1MB memory for 12.5% area budget (iso-area). FLOWER overheads shown in "[]."

unsurprisingly, tempered—YODA and FaME improve over ECP6 by only up to 1.8× and 3.4×, respectively at CoV=0.3. However, as many systems aim to implement improved wear leveling strategies, this THRASH (worst-case) improvement can typically be avoided, and a behavior between SUITE and possibly close to LEVEL is more likely. It is also important to note that the trends bear out that more overhead dedicated to the fault map has a better impact on maintaining fault rates with higher CoV values. For both YODA and FaME, the difference between CoV of 0.2 and 0.3 is tempered when more space is dedicated to the map. The best performing 12.5% overhead approach (FaME31) achieves 22%–2.6× better lifetime than ECC-5 using 5× less area.

C. Performance Impact (DRAM)

The fault tolerance enabled by FLOWER incurs a performance overhead from accessing the fault map. While somewhat mitigated by filtering rows at low fault rates and leveraging in-memory processing, FLOWER access latencies are non-negligible and block subsequent fault map accesses. To quantify the FLOWER performance impact we studied different overheads at different initial fault rates. Performance was simulated with the SNIPER full system simulator [52] with a DRAMSim2 backend [53] using the architecture parameters from Table II. Each memory access includes the delay of the first-level filter and upon a hit indicating at least one fault, the detailed fault-map is accessed as described in Section III-B1. In these results we assume a 4D FLOWER implementation. The results shown are for an average of 10 different fault maps[3] using MinCI with independent arrays, and are reported for an average 12 SPEC CPU benchmarks. Results are conservatively reported for DRAM due to the longer relative latency of in-memory AND operations compared to PCM.

Table II
ARCHITECTURE PARAMETERS

CPU	Cache
4 out-of-order cores	Private L1 32KB Inst, 32KB Data
4 issue width, 4GHz clk	Private L2 256KB/Core
45 nm Technology	Associativity: 8 (L1 data and L2)
1GHz Frequency	Block Size: 64B
Memory: DDR3-1066 (8-8-8) [50] (4GB)	
2 channels, 1 rank per channel, 8 banks per rank	
4KB address filter, 3ns latency [34]	
50ns (Subarray), 155ns (Inter-Bank) RowClone [51]	

Figure 16 shows detailed performance results, in terms of instructions per cycle (IPC), for MinCI with multiple arrays at 6.25% fault map area overhead at different represented fault rates and compared to turning off the first-level fault filter, which has the same performance independent of error rate. The striped bars indicate the additional IPC achieved by using in-memory operations instead of bringing the four dimensions individually into the memory controller. Certain benchmarks, such as `h264ref` and `libquantum`, are not significantly impacted by the FLOWER detailed fault map access, while others, such as `perlbench`, `gcc`, and `gromacs`, can have over 25% reductions in performance at 10^{-2}. However, a general trend across all of the benchmarks is that $\leq 10^{-4}$ fault rates, the performance impact of FLOWER is within 2% of an unprotected system, while still benefiting from the fault map, due to the first-level filter. At 10^{-6}, this 2% degradation comes almost entirely from the access overhead of the address filter. Thus, FLOWER is competitive with previous techniques that attempt to reach 10^{-4} fault rates. Moreover, FLOWER can operate successfully at fault rates up to 10^{-2}, which for compute oriented benchmarks is still competitive with a fault free system. The significant performance impacts only occur for some applications, and only once the concentration of faults reaches the 10^{-4} threshold where other correction schemes no longer operate. Compared to the performance of ArchShield, which was reported as a 1% performance degradation and handled up to 10^{-4} incidence fault rates, FLOWER maintains within 2% performance degradation through 10^{-4}. In this way, FLOWER allows the system to perform correctly in a gracefully degraded mode after a 10^{-4} fault rate threshold.

Figure 17 shows the average normalized performance in IPC as fault rates increase from 10^{-6} to 10^{-2}. Note, smaller area budgets become ineffective when a sufficient fault rate is reached. Missing bars are excluded intentionally as they represent fault maps that report fault rates higher than 10^{-1}, thus providing insufficient accuracy. Depending on the desired level of fault tolerance, the figure can help guide what fault map area budget is sufficient. There is $<$1% performance degradation for $\leq 10^{-5}$ fault rate for each of the fault map storage overheads. At 10^{-4}, the viable limit of many other proposed correction schemes such as ArchShield [3], MinCI still maintains $<$2% performance overhead for fault map storage overheads of $\geq$1.56%. For fault rates above 10^{-4}, the performance degrades, as expected, due to increased hits in the first-level filter requiring more FLOWER accesses. At 10^{-3}, the performance of MinCI with multiple arrays sees a 7% to 10% degradation, depending on the storage budget with the smallest fault map overheads now insufficient to provide sufficient fault resolution. By 10^{-2}, the first level filter no longer provides any benefit (all rows appear faulty) and at least a 6.25% area overhead is required.

In addition to the aforementioned fault map access delays, there are also update delays for adding faults. Independent

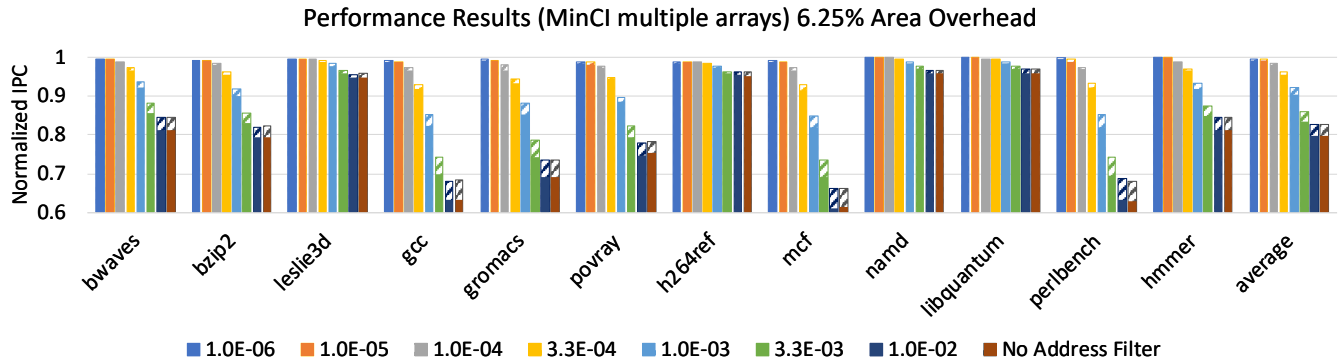

Figure 16. IPC over 12 SPEC Benchmarks at 6.25% area overhead at different error rates (10^{-6} to 10^{-2}) for FLOWER. Striped bars indicate IPC improvement from using in-memory operations.

of memory size, our experiments for an endurance limited memory indicate a 512-bit row, on average, can expect approximately 2.5 million and 1.5 million accesses between new failures when experiencing a fault rate of 10^{-4} and 10^{-2}, respectively. For a 4GB memory with perfectly uniform writes, this translates into $2.6 \cdot 10^6$ and $1.6 \cdot 10^6$ writes between failures at 10^{-4} and 10^{-2}, respectively. These fault rates represent points late in the memory lifetime, after they have accrued significant endurance faults. Thus, new faults appear $< 10^6$ system writes, which is relatively infrequent.

VII. Conclusion

We present FLOWER, a low-overhead, bit-level fault map that facilitates unprecedented levels of fault detection while enabling previously infeasible correction schemes. Additionally, we present FaME, a new fault tolerance scheme for PCM endurance fault mitigation. Our novel hashing technique tuned for FLOWER, MinCI, reduces false positives and improves performance over traditional (*e.g.,* disk-level) hashes such as Murmur. Our in-memory strategy for multi-dimensional fault maps allows faster and more efficient fault map accesses than traditional techniques conducted at the memory controller. PETAL bits provides novel protection for in-memory operations to enhance their reliability in this context. FLOWER greatly enhances the correction capability of a leading DRAM bitline crosstalk correction scheme, PFE, providing an UBER improvement of over 125×. Similarly, FLOWER enables a pointer-based PCM endurance fault tolerance scheme, YODA, to operate with enhanced lifetimes. FLOWER combined with FaME provides a 17× improvement in fault tolerance compared to ECP6 at the same area overhead for SPEC workloads. Compared to Archshield at 10^{-4}, a fault rate supported by physical crosstalk studies of DRAM released within the last five years [4], FLOWER+FaME can provide equal memory protection while reducing the area overhead by 53% and when scaling to higher (*e.g.,* 10^{-2}) fault rates, this overhead advantage exceeds 7×. Further, we extend PCM lifetime using FLOWER+FaME fault tolerance, which can reach beyond 10^{-4} to 10^{-2} faults after continued use. While fault map accesses do have performance overheads, in the typical range of operation for existing systems (fault rates reaching 10^{-4}), FLOWER has an average IPC degradation of $< 2\%$ with a 1.6% area overhead across SPEC benchmarks. Thus, to ensure long lifetimes in gracefully degraded operation, FLOWER can tolerate up to 10^{-2} fault rates with approximately 80% and 83% of original average performance, for memory controller and in-memory processing, respectively, at 6-12.5% area overheads. Beyond traditional computing, FLOWER/FaME will be critical for the success of embedded, implantable, and IoT devices, which must be very small and rely on dense memories with high inherent fault rates.

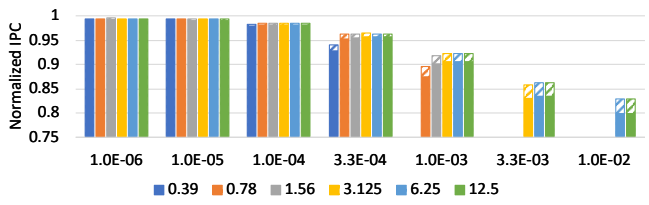

Figure 17. Average IPC over 12 SPEC Benchmarks for different fault map area overheads (0.39% to 12.5%) and error rates (10^{-6} to 10^{-2}). Striped bars indicate IPC improvement from using in-memory operations.

References

[1] K. Kim, "Technology for sub-50nm dram and nand flash manufacturing," in *Elec. Dev. Meeting*. IEEE, 2005.

[2] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them," in *ISCA*, 2014.

[3] P. J. Nair, D.-H. Kim, and M. K. Qureshi, "Archshield: Architectural framework for assisting dram scaling by tolerating high error rates," in *ISCA*, 2013.

[4] S. Khan, D. Lee, and O. Mutlu, "Parbor: An efficient system-level technique to detect data-dependent failures in dram," in *DSN*, 2016.

[5] K. K. Chang, A. Kashyap, H. Hassan, S. Ghose, K. Hsieh, D. Lee, T. Li, G. Pekhimenko, S. Khan, and O. Mutlu, "Understanding latency variation in modern dram chips," in *SIGMETRICS*, 2016.

[6] S. Khan, D. Lee, Y. Kim, A. R. Alameldeen, C. Wilkerson, and O. Mutlu, "The efficacy of error mitigation techniques for dram retention failures," in *SIGMETRICS 2014*.

[7] Y. Konishi *et al.*, "Analysis of coupling noise between adjacent bit lines in megabit drams," *IEEE Journal of Solid-State Circuits*, vol. 24, 1989.

[8] K.-N. Lim *et al.*, "A 1.2 V 23nm 6F 2 4Gb DDR3 sdram with local-bitline sense amplifier, hybrid lio sense amplifier and dummy-less array architecture," in *Solid-State Cir.*, 2012.

[9] J. Liu, B. Jaiyen, Y. Kim, C. Wilkerson, and O. Mutlu, "An experimental study of data retention behavior in modern dram devices," in *ISCA*, 2013.

[10] J. Mandelman, R. Dennard, G. Bronner, J. DeBrosse, R. Divakaruni, Y. Li, and C. Radens, "Challenges and future directions for the scaling of dram."

[11] C. J. Xue, G. Sun, Y. Zhang, J. J. Yang, Y. Chen, and H. Li, "Emerging non-volatile memories," in *CODES+ISSS*, 2011.

[12] M. K. Tavana, A. K. Ziabari, and D. Kaeli, "Live together or die alone," in *DATE*, 2017.

[13] S. Swami, P. M. Palangappa, and K. Mohanram, "Ecs: Error-correcting strings for lifetime improvements in nonvolatile memories," *ACM Trans. Archit. Code Optim.*, vol. 14, 2017.

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

[15] E. Hamilton, "Intel announces optane dc persistent memory dimms," *TechSpot*, Apr 2019. [Online]. Available: https://www.techspot.com/news/79483-intel-announces-optane-dc-persistent-memory-dimms.html

[16] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, 1970.

[17] B. Goodwin, M. Hopcroft, D. Luu, A. Clemmer, M. Curmei, S. Elnikety, and Y. He, "Bitfunnel: Revisiting signatures for search," in *SIGIR*, 2017.

[18] Y. Heo, X.-L. Wu, D. Chen, J. Ma, and W.-M. Hwu, "Bless: bloom filter-based error correction solution for high-throughput sequencing reads," *Bioinformatics*, vol. 30, 2014.

[19] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, "Fast hash table lookup using extended bloom filter," in *SIGCOMM*. ACM, 2005.

[20] F. Deng and D. Rafiei, "Approximately detecting duplicates for streaming data using stable bloom filters," in *SIGMOD*. ACM, 2006.

[21] F. Putze, P. Sanders, and J. Singler, "Cache-, hash-and space-efficient bloom filters," *Experimental Algorithms*, 2007.

[22] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *CoNEXT*, 2014.

[23] P. K. Pearson, "Fast hashing of variable-length text strings," *Communications of the ACM*, vol. 33, 1990.

[24] D. Borthakur, "The hadoop distributed file system: Architecture and design," *Hadoop Project Website*, vol. 11, 2007.

[25] S. M. Seyedzadeh, A. K. Jones, and R. Melhem, "Mitigating wordline crosstalk using adaptive trees of counters," in *ISCA*, June 2018.

[26] Z. Yang and S. Mourad, "Crosstalk induced fault analysis and test in drams," *Journal of Electronic Testing*, vol. 22, 2006.

[27] S. M. Seyedzadeh, D. Kline Jr, A. K. Jones, and R. Melhem, "Mitigating bitline crosstalk noise in dram memories," in *MEMSYS*. ACM, 2017.

[28] J. Kim and M. C. Papaefthymiou, "Block-based multiperiod dynamic memory design for low data-retention power," *TVLSI*, vol. 11, 2003.

[29] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu, "Raidr: Retention-aware intelligent dram refresh," in *ISCA*, 2012.

[30] C. Wilkerson, A. R. Alameldeen, Z. Chishti, W. Wu, D. Somasekhar, and S.-l. Lu, "Reducing cache power with low-cost, multi-bit error-correcting codes," in *ISCA*. ACM, 2010.

[31] P. G. Emma, W. R. Reohr, and M. Mete3relliyoz, "Rethinking refresh: Increasing availability and reducing power in dram for cache applications," *IEEE micro*, vol. 28, 2008.

[32] C.-H. Lin, D.-Y. Shen, Y.-J. Chen, C.-L. Yang, and M. Wang, "Secret: Selective error correction for refresh energy reduction in drams," in *ICCD*. IEEE, 2012.

[33] D. Kline, R. Melhem, and A. K. Jones, "Counter advance for reliable encryption in phase change memory," *IEEE Computer Architecture Letters*, 2018.

[34] Y. H. Son, S. Lee, O. Seongil, S. Kwon, N. S. Kim, and J. H. Ahn, "Cidra: A cache-inspired dram resilience architecture," in *HPCA*. IEEE, 2015.

[35] D. Kline, R. Melhem, and A. K. Jones, "Sustainable fault management and error correction for next-generation main memories," in *IGSC*, 2017.

[36] T. Yuan, S. Z. Ramadan, and S. J. Bae, "Yield prediction for integrated circuits manufacturing through hierarchical bayesian modeling of spatial defects," *Transactions on Reliability 2011*.

[37] J. R. Carson, "The heaviside operational calculus," *The Bell System Technical Journal*, vol. 1, 1922.

[38] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology," in *MICRO*. ACM, 2017.

[39] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *DAC*. ACM, 2016.

[40] S. Swami and K. Mohanram, "Reliable nonvolatile memories: Techniques and measures," *IEEE Design & Test*, vol. 34, 2017.

[41] 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.

[42] 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 *MICRO*, 2010.

[43] J. Fan, S. Jiang, J. Shu, Y. Zhang, and W. Zhen, "Aegis: Partitioning data block for efficient recovery of stuck-at-faults in phase change memory," in *MICRO*, 2013.

[44] J. Zhang, D. Kline Jr, L. Fang, R. Melhem, and A. K. Jones, "Dynamic partitioning to mitigate stuck-at faults in emerging memories," in *ICCAD*. IEEE, 2017.

[45] J. Zhang, D. Kline, L. Fang, R. Melhem, and A. K. Jones, "Yoda: Judge me by my size, do you?" in *ICCD*. IEEE, 2017.

[46] A. P. Ferreira, M. Zhou, S. Bock, B. Childers, R. Melhem, and D. Mossé, "Increasing pcm main memory lifetime," in *DATE*, 2010.

[47] L. Jiang, Y. Zhang, and J. Yang, "Mitigating write disturbance in super-dense phase change memories," in *DSN*. IEEE, 2014.

[48] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, 2006.

[49] Z. Al Ars, *DRAM fault analysis and test generation*. TU Delft, Delft University of Technology, 2005.

[50] A. JEDEC, "Jesd79-3f: Ddr3 sdram specification," 2012.

[51] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Rowclone: fast and energy-efficient in-dram bulk data copy and initialization," in *MICRO*. ACM, 2013.

[52] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *High Perf. Comp. Net. Stor. and Analysis*. ACM, 2011.

[53] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *IEEE Computer Architecture Letters*, vol. 10, 2011.