2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)

On Error Correction for Nonvolatile Processing-In-Memory

Hüsrev Cılasun, Salonik Resch, Zamshed I. Chowdhury, Masoud Zabihi,
Yang Lv, Brandon Zink, Jian-Ping Wang, Sachin S. Sapatnekar, Ulya R. Karpuzcu
University of Minnesota, Twin Cities
{cilas001, resc0059, chowh005, zabih003, lvxxx057, zinkx030, jpwang, sachin, ukarpuzc}@umn.edu

***Abstract*—Processing in memory (PiM) represents a promising computing paradigm to enhance performance of numerous data-intensive applications. Variants performing computing directly in emerging nonvolatile memories can deliver very high energy efficiency. PiM architectures directly inherit the vulnerabilities of the underlying memory substrates, but they also are subject to errors due to the computation in place. Numerous well-established error correcting codes (ECC) for memory exist, and are also considered in the PiM context, however, they typically ignore errors that occur throughout computation. In this paper we revisit the error correction design space for nonvolatile PiM, considering *both* storage/memory and computation-induced errors, surveying several self-checking and homomorphic approaches. We propose several solutions and analyze their complex performance-area-coverage trade-off, using three representative nonvolatile PiM technologies. All of these solutions guarantee single error correction for both, bulk bitwise computations and ordinary memory/storage errors.**

I. Introduction

Processing in memory (PiM) is a promising computing paradigm for data-intensive applications. The core idea is performing logic operations directly within the memory system to minimize, if not eliminate, lengthy and power hungry data transfers. PiM features a rich design space spanned by (i) the underlying memory technology, (ii) where in the memory hierarchy the computation takes place, and (iii) how logic operations are performed. More traditional PiM architectures based on SRAM [2], DRAM [50], STT-MRAM or ReRAM [39] perform computation at the memory array periphery by exploiting sense amplifiers or by using dedicated logic blocks. In this case, the result of each logic operation has to be written back to the corresponding memory element. Computing directly within the memory arrays is also possible. This time, memory elements get seamlessly updated with the results of computation, in-situ, using different memory technologies such as DRAM [21], [60], ReRAM [30], [33], STT-MRAM [56] or SOT/SHE-MRAM [57]. *In this paper we target an especially promising class of PiM for energy efficiency, which can perform universal Boolean computation directly in nonvolatile memory (NVM) [30], [33], [48].*

By construction, PiM architectures directly inherit the reliability characteristics of the underlying memory substrates, but they are also vulnerable to errors due to computation. Unfortunately, error detection and correction for PiM is not well-characterized. Traditional memory systems typically utilize error correcting codes (ECC) to detect and correct storage/memory-induced errors, which are not designed to protect dynamically changing data (i.e., against computation-induced errors which PiM implies). Adapting conventional fault tolerance techniques to protect computation is also possible, but only if PiM logic blocks in charge of computation and the memory arrays represent separate entities, which is not always the case. *Targeted PiM architectures in this paper fuse logic and memory, where each memory cell can directly act as an input or as an output to a Boolean operation, and where computation strictly happens within the array. In this case, neither classical ECCs for storage/memory, nor classical fault tolerance techniques for computation directly apply and represent a comprehensive solution. We focus on this more challenging problem.*

Numerous well-established ECCs for memory [8], including NVM [45], exist. A few studies consider ECCs in the PiM context: One example extends MAGIC based processing in (resistive) memory to support two dimensional parity bits, which enables error detection and correction in idle data only, excluding computation-induced errors [32], [36]. Another example covers Triple-Modular Redundancy (TMR) [37] for MAGIC based PiM in ReRAM. Redundancy here comes in two flavors: time and space. TMR (or generalized N-modular redundancy) is trivially simple and covers computation-induced errors, but can incur a significant time and/or space overhead.

In this paper we make the distinction between conventional *memory* errors, that PiM systems inherit from the underlying memory, and *logic* errors, that stem from computing in memory. Logic errors do not necessarily always manifest themselves as memory errors, especially when computing continuously in memory without any interruption. In this case, corruptions due to logic errors can easily propagate before periodic ECC checks to catch conventional memory errors kick in. Accordingly, we revisit the error correction design space for nonvolatile PiM, considering *both* memory and logic errors. To this end, we explore classical self-checking and homomorphic approaches, and introduce several solutions which can guarantee single error correction for three representative nonvolatile PiM technologies supporting in-array computing semantics.

The paper is organized as follows: Section II covers PiM and fault tolerance basics; Section III, ECC design space for PiM;

This work was in part supported by Cisco fellowships.

979-8-3503-2658-1/24/$31.00 ©2024 IEEE
DOI 10.1109/ISCA59077.2024.00055

Section IV, practical error correction approaches for PiM; Sections V and VI, the evaluation; Section VII, the related work; and Section VIII, a summary of our findings.

II. Background

A. Nonvolatile PiM Basics

Without loss of generality, in this paper we evaluate non-volatile (resistive) PiM architectures which strictly perform logic operations within the memory array. Computing directly in the arrays using memory cells already challenges ECC design due to faster and more frequent data updates in place. This problem becomes especially acute for resistive PiM architectures capable of performing very large numbers of bulk bitwise operations in parallel, energy efficiently. In this case, only a minimum overhead ECC solution can help.

Resistive memory cells encode two distinct levels of device resistance, low (R_{low}) and high (R_{high}), respectively, to logic values (0 and 1, in STT and SOT/SHE MRAM; 1 and 0, in ReRAM). Fig.1 covers three representative examples based on ReRAM (a), STT-MRAM (b), and SOT/SHE-MRAM (c), which facilitate computation directly within the memory arrays using resistive memory cells. In each case, memory functions (i.e., reads and writes) closely follow the operation semantics of the underlying memory technology. For reads, this translates into passing a small current through the memory device (by appropriately biasing the control lines), which gets modulated by the resistance (hence, logic state) of the device, and which in turn is captured by sense amplifiers to extract the actual logic state. For writes in ReRAM, applying a voltage at memory device's on-(off-)threshold changes the resistance to low(high)/1(0). Writes in STT- or SOT/SHE-MRAM, on the other hand, have a single current threshold and the direction of the current through the memory device determines the final state. SOT/SHE-MRAM retains most of the operation principles of STT-MRAM except that each memory device features a Spin-Orbit-Torque (SOT) or Spin-Hall Effect (SHE) channel to enhance the energy efficiency of writes.

For logic operations, all three designs can form universal Boolean gates within the array where each memory cell can act as an input or as an output. Electrically, each gate corresponds to a resistive network as shown in Fig.1(d). First the designated output cell is preset to a known value. Then, by appropriately biasing the control lines, the network in Fig.1(d) is formed. Finally, a gate specific voltage (V_{bias}) is applied across the network, to enforce switching of the output cell according to the underlying truth table, as a function of the logic states of the input cells.

As an example, the output preset value for a NOR gate in MRAM is logic 0. Under any V_{bias} the combined current through the input cells that passes through the output cell monotonically decreases with increasing input resistance levels. Therefore it is possible to determine a NOR specific V_{bias} = V_{NOR} such that the output cell switches –the combined current through the input cells is above the critical current, I_C, for switching the output cell– only if both input cells are in R_{low} state. In ReRAM, a similar procedure applies. All of the systems we cover can support many different universal Boolean gates or gate sets besides NOR. In the following, we will use NOR based bitwise logic without loss of generality.

Due to this thresholding principle in implementing logic operations, not all logic functions can be performed in a single step. An example is XOR which, as depicted in Table I, can take 3 steps (NOR, CP, and the thresholding gate THR applied in a sequence). $\texttt{in}_1$ and $\texttt{in}_2$ here represent the inputs; $\texttt{s}_1$, the output of NOR; $\texttt{s}_2$, a copy of the NOR output; and out, the actual output of XOR, which is the output of the 4-input thresholding gate THR. THR takes $\texttt{in}_1$, $\texttt{in}_2$, $\texttt{s}_1$, and $\texttt{s}_2$ as inputs. The preset for THR output is logic 0, which only switches to 1 if three or more of its inputs are 0. All of the the PiM technologies we consider support 2-output NOR gates, NOR_{22}, which makes finishing the NOR and CP gates from Table I in one step possible, rendering a 2-step XOR function (2-output NOR and THR performed in a sequence). In the Appendix, we provide a detailed electrical characterization for 2-output gate operation.

$\texttt{in}_1$	$\texttt{in}_2$	$\texttt{s}_1$ = NOR($\texttt{in}_1$, $\texttt{in}_2$)	$\texttt{s}_2$ = CP($\texttt{s}_1$)	out = THR($\texttt{in}_1$, $\texttt{in}_2$, $\texttt{s}_1$, $\texttt{s}_1$)
0	0	1	1	0
0	1	0	0	1
1	0	0	0	1
1	1	0	0	0

TABLE I: 3-step XOR [14].

All three PiM technologies support three different levels of parallelism: (1) Partition-level parallelism [38] where each row can be divided into several partitions using transistors, such that multiple logic operations can be performed in each row; (2) Row-level parallelism, where each row can perform the same Boolean gate simultaneously; (3) Array-level parallelism where each memory array can perform computational tasks in parallel.

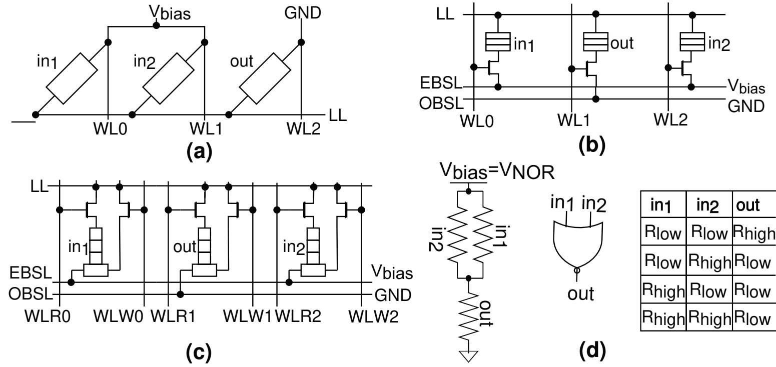
B. Application Mapping

A generic PiM compiler flow incorporates three major steps as sketched in [48]:

1) **Intermediate code generation** involves identifying (multi-bit) PiM operations as well as the data layout for the input, output, and scratch spaces, i.e., cells where the inputs and the outputs reside and where the intermediate gate operations take place, respectively.

2) **Gate-level opcode generation** translates multi-bit operations from 1) into actual Boolean logic gates from the PiM library. HDL synthesis tools can be adopted for this level of translation. Also possible is taking advantage of emerging homomorphic computing tools, where software transpilers can map arbitrary software to Boolean gates [7], [9], [22], [23], [34].

3) **Binary instruction translation** represents the last step, where gate-level opcodes get mapped to their binary equivalents to drive voltages. This mapping is specific to the array architecture and involves the bias voltages for BSLs/WLs from Fig.1.



in1	in2	out
Rlow	Rlow	Rhigh
Rlow	Rhigh	Rlow
Rhigh	Rlow	Rlow
Rhigh	Rhigh	Rlow

Fig. 1: Logic gate construction in a row of **(a)** ReRAM [33], **(b)** STT-MRAM [12], **(c)** SOT/SHE-MRAM [61] arrays; **(d)** Electrical equivalent, circuit symbol and truth table in terms of input and output resistance levels (for STT/SHE-MRAM as a representative example) for 2-input NOR. V_{bias} is a gate-specific voltage applied to Bit Select Lines (BSL). (b) and (c) make a distinction between even (EBSL) and odd (OBSL) BSLs. (c) also distinguishes between Word Lines (WL) for read (WLR) and writes (WLW).

This broad classification can take different shapes depending on the location of the underlying PiM substrate in the memory hierarchy. If, e.g., the PiM substrate represents a custom accelerator, 3) or both 2) and 3) can be undertaken by the accelerator. Otherwise, the memory controller of the respective level in the memory hierarchy can perform the translation in 3), while the rest can be handled purely in software.

C. Fault Tolerance

Errors in a computing system can broadly be classified as *hard* (permanent) and *soft* (temporary) errors. Previous work in the PiM context further distinguishes between soft errors induced by intended operations such as a faulty write or an incorrect logic operation (referred to as *direct* errors) and other soft errors (referred to as *indirect* errors) [37]. Direct errors form the focus of our study. Bulk bitwise logic operations in NVM can be subject to different reliability issues. For example, memristive devices with voltage dependent switching may suffer from resistance fluctuations [65], or the switching of spintronic devices may probabilistically change due to the initial magnetization or thermal fluctuations [41]. In case of ReRAM another major error source is the resistance state confusion [43], which can be quantitatively characterized by the overlap between high and low resistance regions. More precisely, ReRAM is vulnerable to diffusion of oxygen vacancies, ion strikes or environmental factors, in a both temporally correlated and uncorrelated fashion [36]. MRAM is prone to thermal noise [27], retention failures, read disturbance, write errors, write pulse/current variations [46], [51], tunneling magnetoresistance ratio variations [64], and bias voltage variations [62]. For key parameters such as MRAM tunneling magnetoresistance ratio [27] or ReRAM threshold voltage [40], Gaussian distributions apply. Regardless of the origin, however, such nonidealities mainly manifest themselves as single bit flips for STT-MRAM [44], SOT-MRAM [27], or ReRAM [40]. For gate operations, such single bit flips stem from the output not being able to change state when it is supposed to, or vice versa. As these technologies are not mature enough, computation-induced errors in PiM are not well characterized. We also should note that to be feasible for practical applications, respective gate error rates should not significantly exceed the error rates of conventional memory technologies of today. Aside from technology specific adjustments, our error model is based on previous work [36], [37], where, without loss of generality, errors in Boolean gate operations are uniformly distributed in each PiM array throughout row-parallel computation.

Self-checking Circuits have the ability to discern incorrect operation by monitoring their own outputs [4], [55]. This is enabled by adding redundancy in the form of check symbols to the data representation. Two types of self-checking circuits exist: Type-I features *systematic*; Type-II, *non-systematic* codes. In systematic codes, the actual data and check symbols exist as separate distinct components in the overall codeword. Here, direct access to the data to be protected is possible. However, error vulnerability of the check symbols makes it difficult to distinguish between errors in the actual data vs. errors in the check symbols. In non-systematic codes, actual data and check symbols are combined in an interleaved fashion to form codewords. The obvious disadvantage is lack of direct access to actual data, while error assessment becomes easier.

Modular Redundancy in broad terms entails using multiple copies of unreliable components to improve reliability. In Dual Modular Redundancy (DMR), two copies of the exact same computational primitive are executed in parallel or series, the outputs are compared, and an error is signaled in case of a mismatch. DMR can detect but not correct errors. Only if the outputs match the computation is considered correct. Hence, for DMR to work, probability of two simultaneous errors (one in each copy) should be lower than the probability of a single error. Similarly, Triple Modular Redundancy (TMR) relies on three copies where the computation is deemed correct if a strict majority of the outputs match. For TMR to work, two simultaneous errors (one in each copy) should be less likely than a single error, as well. TMR, however,

can correct up to one error. In the context of PiM operations, N-modular redundancy translates into performing N copies of the respective gate operation, which necessitates working with N copies of the corresponding input operands, as well. Targeted PiM architectures in this paper are promising due to the capability of performing bulk bitwise operations in a massively parallel fashion, and the flexibility of being able to fire computation in any row within the memory array. Hence, even for modest N (which is the case for DMR or TMR), the area and/or time overhead can easily become prohibiting.

Hamming Codes [24] are linear block codes[1] featuring a Hamming distance of 3 between any two codewords, therefore, can correct (detect) single (double) bit errors. In formal terms, n bit Hamming codewords are obtained by multiplying each k bit binary vector corresponding to raw data by a binary generator matrix $G_{k\times n}$. Error checks translate into multiplying another binary matrix $H_{(n-k)\times n}$ termed parity-check matrix by codewords, where

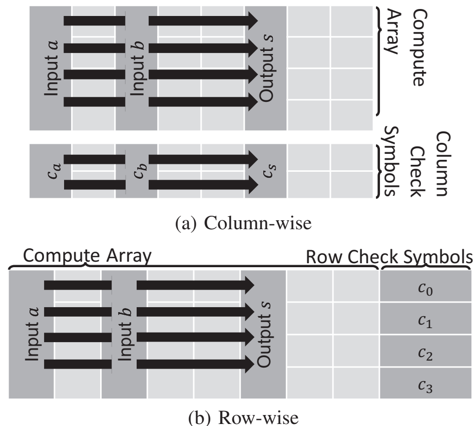
$$G = \{I_k | - A^T\} \qquad H = \{A | I_{n-k}\} \tag{1}$$

applies. A here is a $(n-k)\times k$ binary submatrix; I_k, the $k \times k$ identity matrix; T, the transpose operation; and $|$, horizontal concatenation. A provides a mapping from raw data to codewords such that information in redundant bits can pinpoint erroneous bit flip locations. Considering matrix dimensions, multiplying by G extends the k bit raw data vector by $n-k$ bits. During check, the n bit codeword is multiplied by H, which produces an $n-k$ bit vector called the *syndrome*. If the syndrome is all zero, there is no error. Otherwise, each possible syndrome value points to a unique location for an error in the actual (raw) data vector, which in turn can be corrected by a simple bit flip. Codeword generation (encoding) and syndrome computation translate into modulo-2 matrix multiplications. Hence, a sequence of ANDs suffice for element-wise multiplication, where modulo-2 addition reduces to XOR. Hamming codes can be implemented in systematic or non-systematic form; and conversion between the two forms takes elementary matrix transformations. While the coverage is similar to classical TMR, number of check bits grow as $\log{(n+1)}$ with increasing code word length n.

III. ECC Design Space

We can envision each ECC codeword as a combination of raw data and *check symbols*, i.e., the accompanying redundant information that enables error detection and/or correction. Check symbols in a codeword can be totally isolated from the raw data bits (systematic ECCs); or interleaved (non-systematic ECCs). In the following, we stick to systematic ECCs where data to be protected can be accessed directly, which by construction enables a more modular design, especially useful in the PiM context. In this case, as computation is performed in each row, the check symbols can be organized either column-wise (Fig.2a) or row-wise (Fig.2b). In this illustrative example, all rows compute in parallel, processing a single bit of the inputs a and b to generate a single bit of the output s.

[1]Linear combination of the codewords is another codeword. Block codes can be applied on chunks of limited size data independently.



(a) Column-wise

(b) Row-wise

Fig. 2: Check symbol layout.

A. Column-wise ECC

In Fig.2a, a dedicated portion of the memory array is allocated for computation, where bulk bitwise logic operations are performed. Each column has its check symbols in a separate set of rows, which may form an isolated portion in the same memory array or a separate array. Most significantly, column check symbols of the output column are calculated using the column check symbols of the input columns, which can proceed simultaneously with the actual computation. Deriving the output column check symbols from the input check symbols in this manner, however, is only practical if the following three criteria are satisfied:

1) The bitwise operation in Fig.2a is generic. We can assume that it corresponds to any of the universal Boolean gate operations NAND or NOR, where $s = \bar{a} \vee \bar{b}$ or $s = \bar{a} \wedge \bar{b}$ applies, respectively. In this case, the check symbols should satisfy $s = \bar{a} \vee \bar{b} \longleftrightarrow c_s = f(c_a, c_b)$ or $s = \bar{a} \wedge \bar{b} \longleftrightarrow c_s = f(c_a, c_b)$, where c_a and c_b correspond to the check symbols for a and b; $\longleftrightarrow$ depicts logic equivalence; and f is an appropriately defined ECC operator that generates c_s, i.e., the check symbols for the output of the NAND or NOR. $f(c_a, c_b)$ in this case only depends on c_a and c_b (and not on actual data). This enables ECC updates using check symbols only. However, not all ECCs can satisfy this criterion.
2) The check symbols c_a, c_b should have a modest storage requirement compared to the actual (raw) data.
3) Provided that 1) applies, as the main contributor to the ECC overhead, calculation of $f(c_a, c_b)$ should be computationally cheap.

The first criterion directly restricts the type of applicable ECCs. This criterion essentially is after homomorphic operation, which guarantees that computation on raw data can always be mapped to computation on check symbols without any ambiguity. Under typical homomorphism, bitwise logic operations on raw data map to element-wise addition

and multiplication between very long codewords, which often translates into very complex numerical operations. It is hard to justify the computational overhead, especially for bulk bitwise operations, where a single Boolean logic gate is of concern. As a result, promising homomorphic linear block codes such as Reed-Muller [11] satisfy 1), but not at all 2) and 3). This generally applies to other candidates including homomorphic arithmetic codes such as Berger codes [42]. *Typical homomorphic codes fall short of satisfying 2) and 3) for bulk-bitwise logic in PiM, rendering column-wise ECC infeasible.*

B. Row-wise ECC

The alternative is keeping check symbols on a per row basis, as depicted in Fig.2b. In this case, each operation on the compute array requires an update on the row check symbols, which, when compared to the column-wise alternative, can incur a higher time overhead (especially if only one gate operation can be performed in each row at a time). However, even under partition-level parallelism (where each row is chunked into multiple sub-rows and where a gate operation can be performed in each sub-row simultaneously) check symbol updates cannot be performed *fully* parallel to logic operations on actual data bits, in stark contrast to column-wise schemes. This is because row check symbols c_0, c_1, c_2, c_3 each represent a function of one bit of the output s, and hence, cannot be computed without the corresponding bit of the output being ready. On the other hand, for the row-wise case no other restrictive criterion (such as homomorphism for the column-wise alternative) applies. The only practical criterion for the row-wise case is a time- and space-efficient implementation of check symbol updates. As we are going to cover in the next section, satisfying this criterion is possible by overlapping check symbol updates and actual computation on data bits using Hamming codes.

C. Putting It All Together

Although column-wise ECC (Section III-A) provides a more elegant mathematical solution, the overhead due to complex arithmetic remains much higher than row-wise ECC. This, however, doesn't necessarily imply that row-wise ECC is practical. Row-wise ECC serializes check symbol updates after actual computation, which can incur significant time overhead, easily on a par with trivial modular redundancy. In the following, we will demonstrate to what extent overlapping actual computation with ECC updates can help address this challenge.

IV. Error Correction in Nonvolatile PiM

A. Coverage: Single Error Protection (SEP)

We target novel nonvolatile technologies that can perform logic gates directly in memory arrays, which are still being developed. Device level reliability has to be improved for any practical use scenario. For example, today, the spin-transfer torque magnetic random-access memory (STT-MRAM) [10], [18], [20], [58] is the most mature MTJ technology, with commercial offerings from Everspin, GLOBALFOUNDRIES, IBM, Intel, Samsung, TSMC, and Avalanche Technology, albeit with great restrictions on the structure of the MTJ array. One of the chief benefits of STT-MRAM over competing non-volatile technologies lies in its endurance, energy efficiency, and speed. In laboratory conditions, it has been reported to exhibit endurance up to 10^{14} write cycles [35], energy consumption as low as 90fJ/bit [5], delay as low as less than 200ps [63], and Tunneling Magnetoresistance Ratio as high as 600% [28]. Gate error rate is very sensitive to Tunneling Magnetoresistance Ratio, and can significantly decrease with modest increases in this ratio. Though all of these metrics may not be simultaneously achievable today (for example, a recent Samsung's product [35] offers 10^{14} cycles endurance, 100ns write speed, and 25pJ/bit write energy consumption, where a product line by Avalanche Technology [53] offers 10^{14} cycles endurance, 20ns write speed, and 10pJ/bit write energy consumption), rapid technology developments are promising. Accordingly, once these technologies mature, they will reach reliability levels similar to conventional memory technologies of today, where TMR (Triple Modular Redundancy) level coverage is adequate. Accordingly, we target single error protection (SEP) in this paper.

B. Full System Design: Macroscopic View

In the following, we will explore the design space for novel PiM specific SEP solutions considering two approaches. The first approach relies on fine grain parity calculation to efficiently implement Hamming codes. The second approach rethinks ordinary TMR to protect in-memory computations. We will refer to the resulting designs as ECiM and TRiM respectively.

The key design challenge is keeping the overhead of error detection and correction at bay. The end performance primarily depends on at what granularity in time and space error detection takes place, and upon error detection, how/where error correction is performed:

- Ideally, to prevent error propagation, a check for errors should happen after each logic operation to trigger correction immediately as necessary. While the proposed ECiM and TRiM designs can accommodate this, *to keep the SEP overhead at bay, we perform checks at logic-level granularity*. This does not necessarily compromise coverage, as Boolean operations within the same logic level are typically not data-dependent. This way, we can guarantee SEP in the main computation – as well as parity calculations for ECiM and redundant operations for TRiM.
- If error correction happens directly in memory, the correction logic becomes subject to the very same errors as the main computation, and can become a bottleneck. To address this, our designs feature dedicated error correction blocks, hardened by design, outside of (but near) PiM arrays.

Error Detection & Correction Semantics: Fig.3 provides the full system overview. *Checker* blocks depict external logic blocks dedicated to error detection and correction. For ECiM,

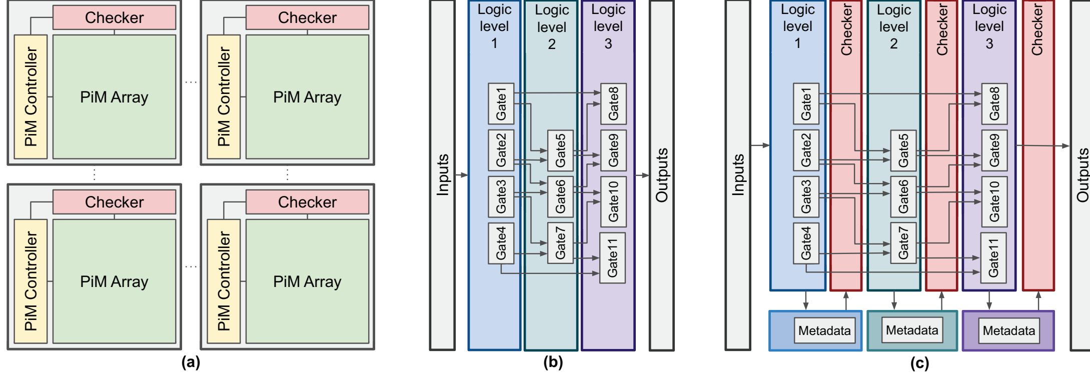
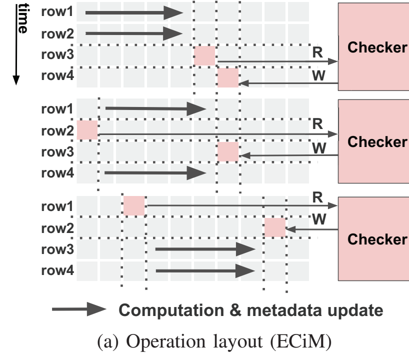


Fig. 3: **(a)** Overall system architecture. **(b)** Main computation in (one row of) memory with logic levels explicitly shown. **(c)** Main computation (in one row of memory) interleaved with error detection/correction as performed by *Checker* blocks. For error detection, ECiM *Checker* blocks are in charge of syndrome generation; TRiM *Checker* blocks, majority vote calculation. *Metadata* translates into parity bits for ECiM; two redundant computation outputs, for TRiM.

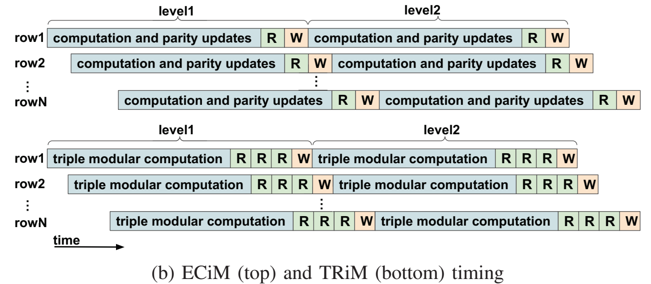
each *Checker* processes the parity information from a PiM array to calculate the syndrome for error detection. For TRiM, each *Checker* processes the output of the main computation along with the two redundant copies to calculate the majority vote for error detection. Error detection happens at logic level granularity. Each *Checker* block is also in charge of error correction and sending the corrected data back to the respective PiM array.

ECiM updates parity information in memory after each logic operation, as detailed in Section IV-C. However, checking for errors happens at the granularity of logic levels outside each PiM array. Whenever the computation of all Boolean gates in a logic level along with the accompanying parity updates is complete, we transfer the result of the computation as well as parity bits to the *Checker* through a conventional memory read. Each logic level occupies a single row of the PiM array. The first phase of the check (decoding in more formal terms) is to multiply the H matrix (from Equation 1, which is hardwired in the *Checker*) with the vector (codeword) incorporating main computation results and parity bits to obtain the syndrome vector. Additions in matrix-vector multiplication reduce to XORs; and bitwise multiplications, to ANDs. The next phase is correction, implemented by XORing the syndrome vector with the vector carrying main computation results. ECiM *Checkers* therefore represent relatively light-weight hardware blocks. If an error is detected, the final phase is writing the corrected logic level output back to the PiM array.

TRiM calculates two redundant copies in memory, in the same row. Whenever the computation of all Boolean gates in a logic level along with two redundant copies is complete, we transfer the result of the computation as well as redundant outputs to the *Checker* through a conventional memory read. The first phase of the check is to take the majority vote among the three copies for error detection. If an error is detected (as signaled by a mismatch in the outputs), the final phase is writing the correct logic level output (the output with the majority vote) back to the PiM array. *Checkers* in this case primarily rely on majority voting logic and small multiplexers, hence, constitute relatively light-weight hardware blocks, as well.



(a) Operation layout (ECiM)



(b) ECiM (top) and TRiM (bottom) timing

Fig. 4: Operation layout (a) and timing (b). Each row independently computes the same gates, logic level by logic level, on different data. To overlap computations in one row with reads or writes in other rows, computations in each row start in a delayed fashion (b). More specifically, recall that we use universal NOR gates as core building blocks for computation; all computations are synthesized using NOR. With delayed start, when r^{th} row executes s^{th} NOR, $(r+1)^{\text{st}}$ row would execute $(s-1)^{\text{st}}$ NOR of the same level, on different data.

Practical Considerations: If not carefully orchestrated, data communication with *Checker* blocks can become a performance bottleneck. In the PiM execution model each row operates in parallel, performing (all logic levels of) the same

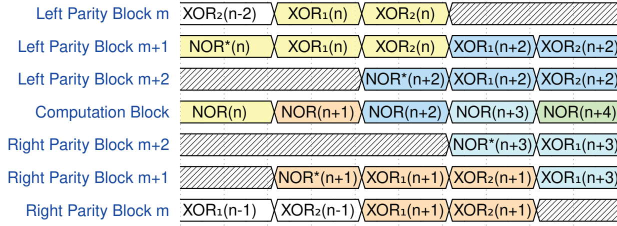


Fig. 5: Timing diagram for parity updates vs. main computation. Each waveform captures the activity in a specific block over time. The hatch pattern depicts an idle block. The computation (NOR) that triggered the parity update (i.e., the two steps of XOR) and the corresponding parity update operations are labeled using the same color. NOR* and NOR in this diagram point to the very same NOR gate in actual computation, whereNOR* indicates the calculation of the second output of NOR_{22}^{n} in a different block. XOR_1 and XOR_2 denote the two steps of XOR; NOR_{22} and THR, respectively. For each gate, the corresponding step in computation is indicated in parenthesis: NOR(n+1) is the NOR gate initiated at Step n+1.

computation on different data (Section II). We cannot initiate error checking for a row until the computations and metadata (parity for ECiM; redundant copies for TRiM) calculations in the same row are complete, at the logic level granularity. However, we can interleave R(ead) and W(rite) operations in a given row (as induced by the communication with the *Checker*) with computations and metadata updates in other rows to reduce the overall time overhead – as demonstrated by the operation layout in Fig.4a (for ECiM, without loss of generality); and timing, in Fig.4b. The goal is to fully utilize the communication bandwidth at the PiM array interface. To this end, we (i) partition columns, similar to rows (Section IV-C); (ii) adjust the layout of PiM operations in each row.

C. Metadata Updates in ECiM

In this section we take a closer look into metadata updates. We start with how ECiM updates parity information after every (NOR) operation throughout computation. For ECiM, this forms the backbone for generating Hamming codes in memory.

Parity Updates in Memory: Targeted PiM architectures perform computations in rows. Hence, actual computation and parity updates can happen in separate, dedicated columns, closely following row-wise error detection semantics from Fig.2b. PiM technologies that we consider in this paper can implement multiple-output gates with identical outputs (Section II), which we can exploit for seamless parity updates. To this end, for example, we can use the two-input/two-output NOR_{22} gate instead of a standard two-input/single output NOR_{21} gate. The first output is produced in dedicated computation columns; the (identical) second output, in dedicated parity columns. We keep a running parity bit in each row, per logic level. The second output gets XOR-ed with the running parity bit to update the instantaneous parity information. Whenever a NOR_{22} output is one, it flips the corresponding running parity bit. Each parity update triggers the XOR of (the instantaneous value of) the respective parity bit with one of the identical outputs of the NOR_{22} (corresponding to the instantaneous step of computation). Each such XOR can be performed as a two step operation consisting of a NOR_{22} and a thresholding THR gate (Section II). Hence, each parity update (triggered by a NOR_{22} operation in the computation columns) gives rise to two extra gate operations (NOR_{22} and THR).

Fig.5 captures how main computation gets interleaved with parity update operations to form a pipeline: The PiM array is partitioned into *left parity columns*, *compute columns* in the middle, and *right parity columns*. Parity columns keep intermediate data during parity updates. Specifically, left and right parity columns keep the parity after alternating steps of computation, respectively, and help pipeline parity update and actual computation operations to minimize the time overhead. Moreover, left and right columns are organized as independent blocks, where each block corresponds to a separate partition. Essentially each block corresponds to a number of neighboring columns, which are separated from the rest of the columns by switches in the logic lines (LLs from Fig.1). Partitioning semantics closely follow [37], by en/disabling switches in logic lines (which establish electrical connections between inputs and outputs of logic operations). No more than one logic gate operation can be in progress in one partition at a time, while gate operations can span (have inputs and outputs distributed over) multiple partitions as long as there are no other overlapping simultaneous gate operations in progress in any of the partitions.

To be more specific, the left and the right side each keeps a separate parity bit, each corresponding to a sequence of alternating gate operations (which we can think of as the odd and even numbered gate operations in the compute columns, respectively, if gate operations were hypothetically enumerated in a sequence at runtime). Every computation in the compute columns triggers a parity update of this form, on one specific side of the parity blocks. Recall that each parity update (triggered by a NOR_{22} operation in the compute columns) gives rise to an XOR implemented by two extra gate operations, NOR_{22} and THR, respectively, which we will refer to as $\mathrm{XOR}_1 = \mathrm{XOR}_{\mathrm{NOR}_{22}}$ and $\mathrm{XOR}_2 = \mathrm{XOR}_{\mathrm{THR}}$. Partitioning parity blocks on the left and right helps to streamline these operations to optimize throughput. This way, in one array, three gate operations can be active in each row at a time, as opposed to a mere one. We keep multiple partitions (i.e., blocks) in the parity columns on each side. On each side, parity computations start in the left- or right-most blocks, respectively. After each step in computation, the next block closer to the computation columns gets used, moving one block at a time. Due to parity block partitioning, each step of computation requires one parity block.

Generating Hamming Codes in Memory: A Hamming code with parameters n and k, denoted as $Hamming(n, k)$, has n bit codewords with $n - k$ redundant bits (corresponding to check symbols) to enable error detection and correction, which we can think of as the equivalent of "parity bits". Specifically,

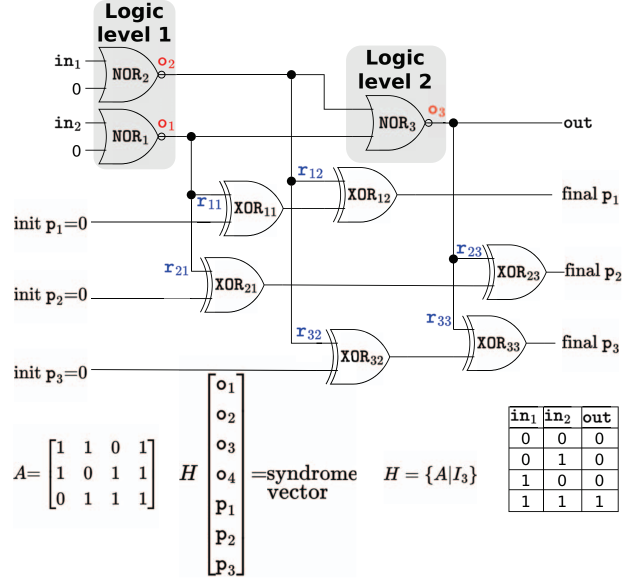
in Equation (1), the kth row of A^T indicates which parity bits need to be updated when the kth bit in the original data vector is updated (as a result of computation). Since A^T is a constant $k \times (n-k)$ matrix, the parity update operation per bit in the original data reduces to up to $n-k$ XOR operations. Hence, all we need to generate Hamming codes is to update $n-k$ parity bits per operation instead of only one parity bit, using the very same semantics and the data layout as in the previous discussion. Since Hamming code has the equivalent of $n-k$ parity bits, once the second output of the $\mathtt{NOR}_{22}$ is produced in the left or right parity blocks, up to $n-k$ XOR computations follow. And as XOR is implemented as a two-step operation, this requires up to $2(n-k)$ gate operations.

D. Metadata Updates in TRiM

TRiM's metadata corresponds to the two redundant copies of the computation results. Using standard, single output gates, we can generate these copies in the same row in a partitioned fashion, similar to the parity generation for ECiM, *by implementing the same gate in three different column partitions simultaneously.* At the same time, we can make use of multiple output gates that the targeted PiM technologies support, to generate the redundant copies in one shot, simultaneous with the actual computation. This can be achieved by using 3-output gates, in the same way as the 2-output gates for ECiM parity updates (Section IV-C). We cover the electrical characterization for multiple output gates in the Appendix. Multiple-output gates have a higher power consumption when compared to their single-output counterparts, which grows linearly with the number of outputs.

E. Single Error Protection Guarantee

Just adapting Hamming codes or TMR does not suffice to guarantee single error correction. The granularity in space and time at which we perform error checks dictates to what extent we can prevent errors in intermediate results to propagate to final outputs. Fig.6 provides an illustrative ECiM example using $Hamming(7,4)$. In a conventional (non-PiM) setting, $Hamming(7,4)$ can correct single errors by construction. The actual computation in this example is a sequence of three multi-output NOR gates – $\mathtt{NOR}_1$, $\mathtt{NOR}_2$, $\mathtt{NOR}_3$ – implementing an AND gate. Each $\mathtt{o}$ represents a gate output, be it intermediate or not, from the main computation. Each $\mathtt{r}$, on the other hand, corresponds to one redundant NOR output used for parity updates. As detailed in the Appendix, such $\mathtt{o}$ and $\mathtt{r}$ for multi-output gates are independent. Because each XOR gate processes one independent $\mathtt{r}$ input, we can guarantee that any error in a given $\mathtt{r}$ affects only a single parity bit and does not spread to other parity bits. A matrix from Fig.6 assigns parity bits to gate outputs (Section II-C); e.g., $\mathtt{p}_1$ protects $\mathtt{o}_1$, $\mathtt{o}_2$. To be more specific, $\mathtt{r}_{\mathtt{ij}}$ corresponds to $\mathtt{i}^{\mathtt{th}}$ parity bit and $\mathtt{j}^{\mathtt{th}}$ logic gate ($\mathtt{NOR}_\mathtt{j}$), where $\mathtt{i},\mathtt{j} = 1,2,3$. Each NOR gate in this example has 3 outputs. Each XOR corresponds to a cascade of a 2-output NOR and a threshold gate. $\mathtt{XOR}_{\mathtt{ij}}$ updates the $\mathtt{i}^{\mathtt{th}}$ parity bit with $\mathtt{r}_{\mathtt{ij}}$. We should also note that, even though XOR is implemented by two gates, any single error in XOR – be it in inputs or component gates – can only cause one bit flip at the respective XOR output.



Error Site	Number of Errors in Logic Level Output	Number of Errors in Final Output	Final Error Outcome
$\mathtt{o}_1$ or $\mathtt{o}_2$	1	3	Error propagates to $\mathtt{out}(\mathtt{o}_3)$, and two respective parity bits if not fixed after logic level 1
$\mathtt{o}_3$	1	1	Error in out
$\mathtt{r}_{11}$ or $\mathtt{r}_{12}$	1	1	Error in $\mathtt{p}_1$
$\mathtt{r}_{21}$ or $\mathtt{r}_{23}$	1	1	Error in $\mathtt{p}_2$
$\mathtt{r}_{32}$ or $\mathtt{r}_{33}$	1	1	Error in $\mathtt{p}_3$

Fig. 6: Single error protection (SEP) guarantee. $\mathtt{p}_1, \mathtt{p}_2, \mathtt{p}_3$ correspond to parity bits; out, to the final result of computation. One data bit ($\mathtt{o}_4$) remains unused, as our small illustrative example does not match a standard size Hamming code.

The table in Fig.6 covers all possible cases for single errors (bit flips), where the (second) third column tabulates the resulting number of errors at the output of (the respective logic level in) the main computation. The first logic level comprises $\mathtt{NOR}_1$ and $\mathtt{NOR}_2$. An error in $\mathtt{o}_1$ ($\mathtt{o}_2$) would result in a single bit error at the output of the first logic level and $\mathtt{NOR}_3$, as well as in the corresponding parity bits $\mathtt{p}_1$, $\mathtt{p}_2$ ($\mathtt{p}_1$, $\mathtt{p}_3$). Therefore, to ensure that the number of errors remain at most one at the time of each error check – such that the *Checker* can guarantee perfect correction – we perform error checks at logic level granularity.

Since ECC is maintained row-wise, potentially correlated errors along the columns are not a concern – they have independent ECC by construction. Error correlation along a row, on the other hand, can be temporal or spatial. Spatial correlation implies errors in confined portions of a row or a partition, where one gate operation would be performed at a time. Each cell (that serves as a gate output) is preset before performing a gate operation, and presets happen alongside error correction. A preset may overwrite an erroneous cell value and mask the error. Otherwise, if the preset itself is erroneous, the corresponding error can be corrected as a logic

error. Spatially correlated errors can result in multiple errors per logic level if the correlation distance is smaller than the row space allocated for a logic level. Temporal correlation implies multiple errors happening back-to-back in time (not necessarily in the same cell), which can lead to multiple errors per logic level if temporally correlated errors are in the same logic level. We can always use stronger codes to protect against multi-bit errors as shown in Fig.8, which our pipeline supports off-the-shelf. Errors in the outputs of multi-output gates can only be correlated if a shared circuit parameter such as the gate voltage is erroneous – which can be avoided by strictly using single-output gates. However, unshared device parameters represent major sources of errors, such as critical current [51], temperature stability factor [59] or tunneling magnetoresistance ratio [64].

F. Putting It All Together: Error Correction Design Space

Table II provides a comparison for different ECiM and TRiM design points, constrained with the same area budget as the unprotected counterparts. ECiM and TRiM designs therefore need to reclaim area throughout the execution, which adds to the time and energy overhead. The overhead of area reclaims strongly depends on the application, and therefore, asymptotic trends in Table II only cover the overheads for metadata updates and maintenance.

Update Granularity and *Check Granularity* refer to the granularity of metadata updates and error checks, respectively. By definition, *Check Granularity* cannot be finer than *Update Granularity*. We consider *gate* and *logic level* granularity. *Gate* implies updating meta data and/or triggering checks in the *Checker* after performing each Boolean gate operation; *logic level*, after performing all Boolean gates in a logic level (which are not data-dependent by construction). A *Check Granularity* of *circuit* is also possible, by triggering checks after performing all logic levels in the computation, however, cannot guarantee single error protection – irrespective of the underlying *Update Granularity*, even a single gate error can propagate to gates in subsequent logic levels to result in multiple errors before the check takes place. Aside from *Time* and *Energy* overhead, we also report how much space the *Checker* has to allocate for metadata – which also serves as a proxy for the communication overhead between PiM arrays and *Checker* blocks.

TRiM with *Check Granularity* = *Gate* and *Update Granularity* = *Gate* corresponds to classic triple modular redundancy in time, which incurs a time overhead of 3×. Error checks (which boil down to majority logic calculation) happen after each gate operation. Timing optimizations like overlapping a gate operation in one row with the checks for another row, while possible, are less effective in this case, as the latency of checks and single gate operations are not well balanced. Under *Check Granularity* = *Logic Level* such overlapped execution becomes more effective, by overlapping checks in one row with the computation of a logic level in another row – which typically features many more gates than one. Accordingly, with sufficiently large logic levels, it is possible to mask the 3× time overhead of classic TMR.

ECiM with *Check Granularity* = *Gate* and *Update Granularity* = *Gate*; i.e., *Hamming(3,1)*, reduces to TRiM operating at the very same granularity. Under *Check Granularity* = *Logic Level*, time and energy overhead evolve with the *Checker* metadata overhead, which becomes a logarithmic function of the number of bits (i.e., gate outputs) to be protected.

Note that [37] also proposes a TMR implementation for PiM induced errors. However, [37] uses PiM arrays for error correction, which limits protection capabilities. TRiM, on the other hand, only performs redundant computations in PiM arrays, and uses an external *Checker* for error detection and correction. TRiM also features several design optimizations to minimize latency compared to an iso-area unprotected baseline.

V. Experimental Setup

We use a behavioral simulator for functional validation, alongside a cycle-accurate timing simulator which can extract energy consumption and latency using the technology parameters in Table III. The simulator decomposes fixed point integer arithmetic into Boolean arithmetic and manages scratch space using a greedy memory allocator, which reclaims cells (whose data is no longer needed) whenever the array runs out of available scratch space. We utilize NVSim [17] to estimate the peripheral circuitry overhead induced by sense amplifiers, column decoders, predecoder, charge/precharge, and driving control lines. To estimate the Hamming decoder and majority voter overhead in hardware, we utilize NanGate 45nm open cell library [1] and OpenROAD flow [3].

We experiment with representative benchmarks of different scales: fixed-point Dense Matrix Multiplication, a Multi-layer Perceptron (MLP), and a larger scale FFT. In dense matrix multiplication, we experiment with 8x8 (*mm8*), 16x16 (*mm16*), 32x32 (*mm32*), and 64x64 (*mm64*) matrices. We use a two layer perceptron with 64 hidden neurons to classify MNIST, considering 1–4 bits of weight precision (*mnist1, mnist2, mnist3, mnist4*). We also include a variant of the in-memory FFT implementation from [16] as a representative larger scale benchmark for sensitivity analysis. We experiment with 8, 16, 32, and 64 element FFTs (*fft8, fft16, fft32, fft64*). While FFT can be implemented by ordinary sparse matrix-vector multiplications, our FFT benchmark is optimized for performance through butterfly arithmetic with complex numbers. We map all benchmarks to the underlying PiM substrates by synthesizing a PiM gate schedule in space and time. We use a fleet of PiM arrays, responsible for performing bulk bitwise logic as well as for maintaining parity state for ECiM and redundant computations for TRiM. Depending on the application/problem size, we experiment with various array sizes in order to maximize the utilization. All benchmarks are mapped to (no more than 16) 256×256 arrays for the whole computation. In order to match array dimensions and maximize array interface utilization, we use $Hamming(255, 247)$ code with $n = 255$ and $k = 247$.

	Update Granularity	Check Granularity	SEP Guarantee	Time	Energy	Checker Metadata
	Gate	Gate	✓	$3N$	$3N$	$2N$
TRiM	**Gate**	**Logic Level**	✓	3N, but can be fully masked	$3N$	$2N$
	Gate	Gate	Reduces to TRiM			
ECiM	**Gate**	**Logic Level**	✓	$N(1+\log N)$	$N(1+\log N)$	$N \log N$

TABLE II: Single Error Protection (SEP) design space for protecting N PiM gate outputs. As highlighted in the table, our TRiM and ECiM designs perform metadata updates at gate; error checks, at logic level granularity.

Parameter	STT	SOT/SHE	ReRAM
$R_{low}/R_{ON}/R_P$ $(K\Omega)$	3.15 [61]	253.97 [14]	10 [54]
$R_{high}/R_{OFF}/R_{AP}$ $(K\Omega)$	7.34 [61]	507.94 [14]	1000 [54]
R_{SHE} $(K\Omega)$	-	64 [14]	-
I_C (μA)	50 [61]	3 [14]	-
V_{OFF}/V_{ON} (V/V)	-	-	0.3/-1.5 [52]
t_{switch} (ns)	1 [14]	1 [14]	1.3 [52]
NOR Energy (fJ)	10.5	2.45	19.68
THR Energy (fJ)	11.2	1.31	20.99
Write Energy (fJ)	1.03	0.01	23.8

TABLE III: Technology parameters. R_{SHE} captures the resistance of the SHE channel; t_{switch}, the switching time (i.e., gate delay), respectively.

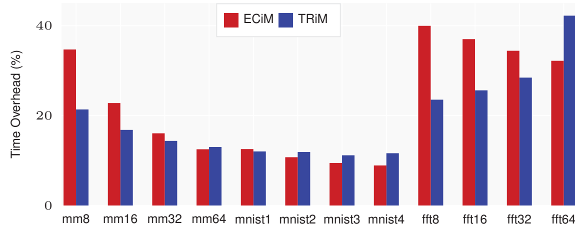


Fig. 7: Time overhead compared to an unprotected iso-area baseline.

VI. Evaluation

In the following, we characterize the time and energy overhead of ECiM and TRiM under an *iso-area* budget when compared to an unprotected baseline for each benchmark. ECiM and TRiM designs therefore need to reclaim area throughout the execution, which adds to the time and energy overhead. While the area budget with fault tolerance is likely to be higher than the unprotected area budget, area increases to totally eliminate area reclaims may not always be feasible considering asymptotic trends and practical limitations (Table II). In this regard, the reported overheads can be regarded as worst-case overheads.

Time Overhead: Fig.7 shows the time overhead with respect to an unprotected iso-area baseline, using multi-output gates. For both, ECiM and TRiM, the main contributing factor to the latency is metadata updates along with area reclaims, i.e., the number of times used scratch space is recycled to meet the area budget. Since TRiM metadata takes up notably more space than ECiM, in TRiM, less scratch space is available for main computation. As a consequence, larger problem sizes incur more area reclaims, and the time overhead for TRiM grows more significantly. Overall, TRiM outperforms ECiM for smaller problem sizes. The opposite applies at scale. For the largest problem we considered (*fft64*), ECiM has a lower

	mm				mnist				fft			
	8	16	32	64	1	2	3	4	8	16	32	64
ECiM	10	22	44	90	224	427	718	1162	26	42	60	87
TRiM	40	82	166	334	658	1414	2447	4295	118	241	445	1109

TABLE IV: Number of area reclaims.

time overhead (29%) than TRiM (42%). This trend (Fig.7) is in line with the number of area reclaims from Table IV. With growing problem sizes the logarithmic latency overhead of ECiM (Table II) gets amortized, while metadata overheads (including the data communication with the *Checker*) become even more significant for TRiM.

Energy Overhead: Table V summarizes the energy overhead compared to an iso-area unprotected baseline, considering different technologies. We also make a distinction between ECiM and TRiM implementations using single-output and multi-output gates. Overall, in smaller benchmarks, ECiM has a higher energy overhead due to parity updates being performed for every NOR operation in the actual computation. Still, depending on the application requirements and error characteristics, the efficiency in the area vs. latency trade-off can mask this energy drawback. Different benchmarks give rise to different circuit topologies, which dictate the number of logic levels (circuit depth). Logic level count tends to increase with problem size, and determines the frequency of checks, hence the communication overhead with the *Checker*. For larger benchmarks, area reclaims and communication overheads become more significant, and more so for TRiM, as TRiM incurs more area reclaims and higher-volume data communication with the *Checker*.

The differences in the energy overhead of different technologies for the same design configuration mainly stem from the differences in the relative energy of gate operations with respect to writes. Generally, if error correction in memory is more computation heavy (due to more complex metadata updates as for ECiM) than data communication (due to larger volume data transfers to/from the *Checker* as for TRiM), relatively higher energy of writes with respect to gate operations can render lower energy overheads and vice versa, granted that a break even point exists.

The energy overheads in Table V reflect practical factors such as the energy overhead of area reclaims or data communication with the *Checker* as well as the *Checker* energy, which are not considered in Table II.

Extension to Higher-Coverage Codes: Beyond Hamming codes, ECiM can also support a class of codes that can correct multiple errors. One example is BCH codes [6], [26], where multiple error correction is possible at the expense of an

	ECiM						TRiM					
	ReRAM		STT-MRAM		SOT-MRAM		ReRAM		STT-MRAM		SOT-MRAM	
	m-o	s-o	m-o	s-o	m-o	s-o	m-o	s-o	m-o	s-o	m-o	s-o
mm8	19.55	81.23	43.26	355.03	360.56	1715.01	**4.92**	17.98	6.42	46.50	37.67	175.82
mm16	4.07	13.86	7.67	56.48	57.38	269.67	**2.30**	6.44	2.23	11.30	7.80	33.43
mm32	3.29	10.58	5.50	38.40	38.45	179.43	**2.65**	7.91	2.34	12.19	6.54	27.40
mm64	**3.05**	9.51	4.40	29.23	28.47	131.90	3.42	11.05	2.76	15.66	5.89	24.30
mnist1	**32.97**	122.27	31.38	247.01	207.62	984.11	53.89	201.84	37.37	295.90	36.84	171.57
mnist2	**39.20**	151.33	28.90	229.16	167.46	793.48	87.04	339.91	57.87	466.58	38.43	179.22
mnist3	**46.30**	184.38	28.96	231.31	149.64	708.98	117.75	474.00	76.55	624.02	39.47	184.26
mnist4	**56.70**	230.97	31.55	253.94	147.60	699.49	167.31	688.08	107.25	881.51	46.06	215.71
fft8	6.81	25.82	14.11	110.53	112.41	532.00	**2.54**	7.58	2.78	15.93	12.41	55.41
fft16	6.43	24.00	13.20	102.75	104.64	494.93	**2.70**	8.20	2.96	17.38	13.30	59.63
fft32	6.10	22.69	12.39	96.06	97.67	461.70	**2.90**	9.11	3.19	19.32	14.48	65.27
fft64	5.82	21.59	11.69	90.26	91.58	432.71	**3.72**	12.63	4.20	27.79	20.37	93.34

TABLE V: Energy overhead compared to an unprotected iso-area baseline. *m-o*(*s-o*) denote multi(single)-output gate designs. Highlighted are lowest overhead designs for each benchmark.

increased number of parity bits, as captured by Fig.8. We use $Hamming(255, 247)$ by default with $n = 255$ and $k = 247$. BCH codes are characterized by the same parameters, and we sweep k for a fixed n in this analysis. Similar to Hamming codes, BCH codes can be updated in memory just by updating the corresponding parity bits from the non-identity part of the generator matrix G (*i.e.*, $-A^T$) and can be implemented following the exact same principles outlined in Section IV-C. Latency and energy overhead of ECiM is proportional to the number of parity bits to be maintained, which grows in a sublinear fashion in the more general case of BCH codes.

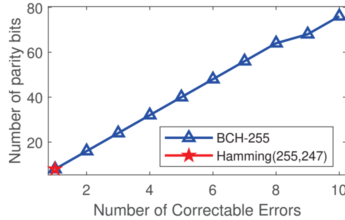


Fig. 8: Number of parity bits vs. correctable errors.

VII. Related Work

Homomorphic codes are appealing candidates for ECC in PiM applications, since extracting output parity solely using input parities maximizes efficiency. For example, Reed-Muller codes support additive and multiplicative homomorphism and are promising for homomorphic computing [11]. However, addition in original data becomes XOR-addition in codewords, while multiplication becomes even-costlier cyclic convolution. Simple operations on original data translate into orders-of-magnitude costlier (codeword) computations, resulting in excessive overhead for bulk bitwise operations. Other linear block codes don't support binary element-wise multiplication cost-efficiently [25], required to implement universal Boolean logic. Arithmetic codes such as Berger, Residue, AN, ANB, and ANBD inherently support homomorphism for addition and multiplication [49]. Only Berger codes allow direct computation of bitwise operations homomorphically, but still not cost-effectively, as output check symbols don't only depend on input check symbols.

Reliable-Simpler-MAGIC supports general-purpose in-memory ECC, capable of detecting and correcting only storage related errors, and not logic errors induced by computations in memory [32], [36]. The idea is diagonally calculating parities once the data is written into the PiM substrate, before and after sensitive tasks. The only relevant work which can correct general-purpose PiM logic errors is the TMR approach in [37], which (with several extra optimizations to maximize fairness) we consider in our evaluation. This paper does not specify how to perform majority voting, data transfers to/from the voting module, or operation under memory size constraints. We cover all of these aspects in our MR-based solution TRiM. More precisely: (1) We introduce an external checker module to guarantee correct operation, while majority voting for TMR is not protected in [37]. (2) We propose a pipeline to mask data communication overheads with the external checker at logic level granularity. (3) We provide a detailed quantitative characterization of time and energy overheads under strict area constraints, which is not provided in [37]. Moreover, TRiM is just one of the designs we propose in our study. We also introduce a PiM-specific Hamming code based design (ECiM) of minimal time overhead under strict area budget constraints.

More recent work [40] on ECC for PiM with memristive devices uses four extra gate operations to protect a single Boolean gate computation, incurring a significantly higher overhead than our solutions by construction. Another recent study, FAT-PiM [66] focuses on only detection in crossbars. FAT-PiM is based on a summation idea that cannot accommodate general-purpose bulk bitwise computation in PiM technologies we consider. This also is the case for [19], which uses arithmetic codes in a dot-product crossbar setting specialized for matrix-vector multiplication. Arithmetic (more specifically, AN) codes can efficiently protect linear algebraic computations, but not bitwise logic [49]. Only Berger codes support bitwise logic, which are not feasible for PiM due to complex data dependencies. In contrast, our paper targets general purpose computations in memory via bulk bitwise logic – capable of performing matrix vector multiplication, but not limited to linear algebraic computations due to the support for universal logic gates. Nevertheless, being tightly tailored to an application domain, designs like [19] incur less

overhead than our general purpose solutions. Finally, while only detection can still be acceptable for applications with a sufficient budget for recomputation upon error detection, direct correction is desirable for wider spread adoption.

VIII. Conclusion & Discussion

While memory errors are extensively studied, error detection and correction for processing in memory (PiM) requires rethinking due to the highly dynamic nature of the data to be protected. In this study, we investigate various techniques to improve the reliability of nonvolatile PiM operations. We compile the specification for a PiM-oriented ideal error correction scheme and explore the design space. We propose several solutions based on Hamming codes (ECiM) and TMR (TRiM), and characterize their time and space complexity along with energy efficiency under iso-error-coverage (guaranteed single error correction), considering representative nonvolatile PiM technologies – spanning ReRAM and MRAM variants. Key novel aspects which apply to all design points include: Single error protection guarantee due to error correction and detection at *logic level* granularity; full system design featuring an external checker with optimized data transfer to/from the checker; design modularity and straight-forward extension to stronger codes with protection guarantees for larger number of errors (such as BCH). ECiM and TRiM by construction provide guaranteed protection against computation-induced errors and inherently cover memory/storage errors, including potential errors in the input data.

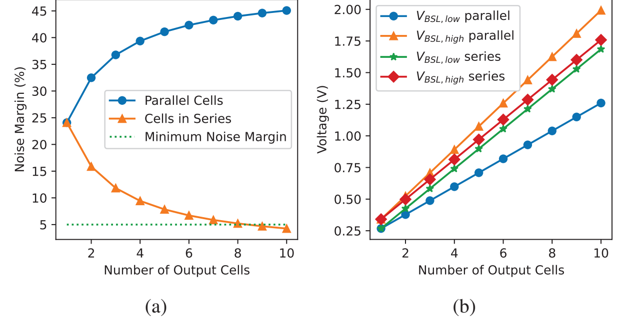
We assume that the PiM substrate corresponds to a standalone accelerator, which may be connected to a host machine but which is not tightly integrated into the memory hierarchy of the host. In this case we can provide full coverage for both logic and memory. We can catch memory errors if corresponding memory cells serve as gate inputs or outputs. When used as a dedicated accelerator for large scale memory intensive applications, this generally is the case for the vast majority of the cells [13], [15], [29], [31], [47]. However, even if memory cells do not serve as logic gate inputs or outputs, we can designate the "computation" as a trivial buffering operation to cover pure memory errors. If, on the other hand, the PiM substrate is tightly integrated with the memory hierarchy and is only occasionally used for smaller scale computations (as opposed to a dedicated accelerator), it is plausible to assume that the portion of the memory contained in the PiM substrate remains unchanged most of the time. Accordingly, lower-overhead classical fault tolerance techniques may become applicable.

Appendix: Electrical Characterization

Correct operation relies on the following conditions:

1) **Support for 2-output gates:** The 2-step XOR operation requires 2-output NOR (NOR_{22}) gates.

2) **Bias voltage matching:** Since multiple partitions in the same array are controlled using the same column control lines, i.e., WLs and WLW/Rs, they need to operate under the same bias voltages. Therefore, we need to ensure that different types of logic components in the implementation, i.e., NOR_{22} and 4-input THR gates, work within the same bias voltage range.



(a) (b)

Fig. 9: Noise margins (a) and voltages (b).

STT and SOT/SHE: Extending the analysis in [61] to the serial and parallel (with respect to how the resistances of the two outputs are connected in the resistive division network corresponding to the underlying logic gate) multiple output gate operation, using the "Today's MTJ" parameter set (of [61]), we obtain Equation (2) for the parallel case:

$$\begin{aligned} V_{BSL,low(\text{parallel})} &= N\frac{(TMR+1)R_P}{TMR+2} + \frac{R_P}{N}I_C, \\ V_{BSL,high(\text{parallel})} &= N\frac{(TMR+1)R_P}{2} + \frac{R_P}{N}I_C \end{aligned} \tag{2}$$

Equation (3) captures the serial case:

$$\begin{aligned} V_{BSL,low(\text{series})} &= \frac{(TMR+1)R_P}{TMR+2} + R_P N I_C, \\ V_{BSL,high(\text{series})} &= \frac{(TMR+1)R_P}{2} + R_P N I_C \end{aligned} \tag{3}$$

The low and high voltages are obtained by Kirchoff's laws on the equivalent resistances for marginally non-switching and switching cases in the truth table. Here, TMR is the Tunnel Magneto-Resistance; R_P, the MTJ parallel state (i.e., low) resistance; I_C, the critical switching current; and N, the number of output cells. Noise margin (%) is defined as $\frac{V_{\text{high}}-V_{\text{low}}}{\frac{V_{\text{high}}+V_{\text{low}}}{2}}$ in [61] as a measure of error tolerance of the gate implementation. We show the resulting noise margins for the multiple-output case, considering both serial and parallel connectivity, in Fig.9a. We assume a 5% minimum noise margin. The corresponding bias voltages are provided in Fig.9b. This analysis reveals that multi-output implementations are feasible and more efficient when the output MTJs are placed in parallel. Further, THR's bias voltage (V_{bias}) should satisfy:

$$\begin{aligned} I_C\,(R_P \parallel R_P \parallel R_P \parallel R_{AP}+R_P) < V_{bias} < \\ I_C\,(R_P \parallel R_P \parallel R_{AP} \parallel R_{AP}+R_P) \end{aligned} \tag{4}$$

To ensure that both THR and NOR correctly work in the same bias voltage range, we introduce D dummy inputs to NOR gates. D is 4 for STT; 5, for SOT/SHE; and 2, for ReRAM. For an N-output NOR with D dummy inputs to match the voltage

range of the thresholding gate, we can characterize the voltage requirements as follows:

$$NI_C \left(R_P \parallel R_P \parallel \frac{R_P}{D} + \frac{R_P}{N} \right) < V_{bias} < NI_C \left(R_P \parallel R_{AP} \parallel \frac{R_P}{D} + \frac{R_P}{N} \right) \quad (5)$$

`D` depends on the technology parameters, and can be easily tuned to find an overlapping bias voltage range within which both the `NOR` and the `THR` gates can operate. In the SHE case, besides the changes in the technology parameters, the only difference in Eqs. 4 and 5 is the output resistance, which now becomes the resistance of the SHE channel, respectively. Note that here $\frac{R_P}{N}$ is the output resistance and the remaining resistance values in the parenthesis correspond to the marginal input combination. Our observations so far hold otherwise.

ReRAM: The low and high resistance states are characterized by R_{ON} and R_{OFF}; threshold voltages, by V_{OFF} and V_{ON}. We can derive `THR` bias voltage from

$$\frac{V_{OFF}}{R_{ON}} (R_{ON} + R_{OFF} \parallel R_{OFF} \parallel R_{ON} \parallel R_{ON}) < V_{bias} < \frac{V_{OFF}}{R_{ON}} (R_{ON} + R_{OFF} \parallel R_{OFF} \parallel R_{OFF} \parallel R_{ON}) \quad (6)$$

and the $\texttt{NOR}_{22}$ bias voltage from

$$\frac{V_{OFF}}{\frac{R_{ON}}{N}} \left(\frac{R_{ON}}{N} + R_{OFF} \parallel R_{ON} \parallel \frac{R_{ON}}{D} \right) < V_{bias} < \frac{V_{OFF}}{\frac{R_{ON}}{N}} \left(\frac{R_{ON}}{N} + R_{OFF} \parallel R_{OFF} \parallel \frac{R_{ON}}{D} \right) \quad (7)$$

We can match the `NOR` and `THR` bias voltage range in a similar manner to STT or SOT/SHE otherwise.

References

[1] "Nangate 45nm open cell library," 2009.

[2] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute caches," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 481–492.

[3] T. Ajayi, V. A. Chhabria, M. Fogaça, S. Hashemi, A. Hosny, A. B. Kahng, M. Kim, J. Lee, U. Mallappa, M. Neseem *et al.*, "Toward an open-source digital flow: First learnings from the openroad project," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–4.

[4] D. A. Anderson, *Design of self-checking digital networks using coding techniques*. University of Illinois at Urbana-Champaign, 1971.

[5] K. Ando, S. Fujita, J. Ito, S. Yuasa, Y. Suzuki, Y. Nakatani, T. Miyazaki, and H. Yoda, "Spin-transfer torque magnetoresistive random-access memory technologies for normally off computing (invited)," *Journal of Applied Physics*, vol. 115, no. 17, 04 2014, 172607. [Online]. Available: https://doi.org/10.1063/1.4869828

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

[7] S. Carpov, P. Dubrulle, and R. Sirdey, "Armadillo: a compilation chain for privacy preserving applications," in *Proceedings of the 3rd International Workshop on Security in Cloud Computing*, 2015, pp. 13–19.

[8] C.-L. Chen and M. Hsiao, "Error-correcting codes for semiconductor memory applications: A state-of-the-art review," *IBM Journal of Research and development*, vol. 28, no. 2, pp. 124–134, 1984.

[9] E. Chielle, O. Mazonka, N. G. Tsoutsos, and M. Maniatakos, "E3: A framework for compiling c++ programs with encrypted operands." *IACR Cryptol. ePrint Arch.*, vol. 2018, p. 1013, 2018.

[10] Y.-D. Chih, Y.-C. Shih, C.-F. Lee, Y.-A. Chang, P.-H. Lee, H.-J. Lin, Y.-L. Chen, C.-P. Lo, M.-C. Shih, K.-H. Shen, H. Chuang, and T.-Y. J. Chang, "A 22nm 32Mb Embedded STT-MRAM with 10ns Read Speed, 1M Cycle Write Endurance, 10 Years Retention at 150°C and High Immunity to Magnetic Field Interference," in *2020 IEEE International Solid- State Circuits Conference - (ISSCC)*, 2020, pp. 222–224.

[11] J. Cho, Y.-S. Kim, and J.-S. No, "Homomorphic computation in reed-muller codes," *IEEE Access*, vol. 8, pp. 108 622–108 628, 2020.

[12] Z. Chowdhury, J. D. Harms, S. K. Khatamifard, M. Zabihi, Y. Lv, A. P. Lyle, S. S. Sapatnekar, U. R. Karpuzcu, and J.-P. Wang, "Efficient in-memory processing using spintronics," *IEEE Computer Architecture Letters*, vol. 17, no. 1, pp. 42–46, 2017.

[13] Z. I. Chowdhury, S. K. Khatamifard, S. Resch, H. Cılasun, Z. Zhao, M. Zabihi, M. Razaviyayn, J.-P. Wang, S. S. Sapatnekar, and U. R. Karpuzcu, "Cram-seq: Accelerating rna-seq abundance quantification using computational ram," *IEEE Transactions on Emerging Topics in Computing*, vol. 10, no. 4, pp. 2055–2071, 2022.

[14] Z. I. Chowdhury, S. K. Khatamifard, Z. Zhao, M. Zabihi, S. Resch, M. Razaviyayn, J.-P. Wang, S. Sapatnekar, and U. R. Karpuzcu, "Spintronic in-memory pattern matching," *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits*, vol. 5, no. 2, pp. 206–214, 2019.

[15] H. Cılasun, S. Resch, Z. I. Chowdhury, E. Olson, M. Zabihi, Z. Zhao, T. Peterson, K. K. Parhi, J.-P. Wang, S. S. Sapatnekar *et al.*, "Spiking neural networks in spintronic computational ram," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 18, no. 4, pp. 1–21, 2021.

[16] H. Cılasun, S. Resch, Z. I. Chowdhury, E. Olson, M. Zabihi, Z. Zhao, T. Peterson, J.-P. Wang, S. S. Sapatnekar, and U. Karpuzcu, "Crafft: High resolution fft accelerator in spintronic computational ram," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[17] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, "Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 7, pp. 994–1007, 2012.

[18] D. Edelstein, M. Rizzolo, D. Sil, A. Dutta, J. DeBrosse, M. Wordeman, A. Arceo, I. C. Chu, J. Demarest, E. R. J. Edwards, E. R. Evarts, J. Fullam, A. Gasasira, G. Hu, M. Iwatake, R. Johnson, V. Katragadda, T. Levin, J. Li, Y. Liu, C. Long, T. Maffitt, S. McDermott, S. Mehta, V. Mehta, D. Metzler, J. Morillo, Y. Nakamura, S. Nguyen, P. Nieves, V. Pai, R. Patlolla, R. Pujari, R. Southwick, T. Standaert, O. van der Straten, H. Wu, C.-C. Yang, D. Houssameddine, J. M. Slaughter, and D. C. Worledge, "A 14 nm Embedded STT-MRAM CMOS Technology," in *2020 IEEE International Electron Devices Meeting (IEDM)*, 2020, pp. 11.5.1–11.5.4.

[19] B. Feinberg, S. Wang, and E. Ipek, "Making memristive neural network accelerators reliable," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 52–65.

[20] W. Gallagher, E. Chien, T.-W. Chiang, J.-C. Huang, M.-C. Shih, C. Wang, C.-H. Weng, S. Chen, C. Bair, G. Lee, Y.-C. Shih, C.-F. Lee, P.-H. Lee, R. Wang, K.-H. Shen, J. J. Wu, W. Wang, and H. Chuang, "22nm STT-MRAM for Reflow and Automotive Uses with High Yield, Reliability, and Magnetic Immunity and with Performance and Shielding Options," in *2019 IEEE International Electron Devices Meeting (IEDM)*, 2019, pp. 2.7.1–2.7.4.

[21] F. Gao, G. Tziantzioulis, and D. Wentzlaff, "Computedram: In-memory compute using off-the-shelf drams," in *Proceedings of the 52nd annual IEEE/ACM international symposium on microarchitecture*, 2019, pp. 100–113.

[22] S. Gorantala, R. Springer, S. Purser-Haskell, W. Lam, R. Wilson, A. Ali, E. P. Astor, I. Zukerman, S. Ruth, C. Dibak *et al.*, "A general purpose transpiler for fully homomorphic encryption," 2021.

[23] R. A. Hallman, K. Laine, W. Dai, N. Gama, A. J. Malozemoff, Y. Polyakov, and S. Carpov, "Building applications with homomorphic encryption," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2160–2162.

[24] R. W. Hamming, "Error detecting and error correcting codes," *The Bell system technical journal*, vol. 29, no. 2, pp. 147–160, 1950.

[25] X. Han and F. Gao, "Homomorphic error-control codes for linear network coding in packet networks," *China Communications*, vol. 14, no. 9, pp. 178–189, 2017.

[26] A. Hocquenghem, "Codes correcteurs d'erreurs," *Chiffers*, vol. 2, pp. 147–156, 1959.

[27] B. Hoffer and S. Kvatinsky, "Performing stateful logic using spin-orbit torque (sot) mram," in *2022 IEEE 22nd International Conference on Nanotechnology (NANO)*. IEEE, 2022, pp. 571–574.

[28] S. Ikeda, J. Hayakawa, Y. Ashizawa, Y. M. Lee, K. Miura, H. Hasegawa, M. Tsunoda, F. Matsukura, and H. Ohno, "Tunnel magnetoresistance of 604% at 300K by suppression of Ta diffusion in CoFeB/MgO/CoFeB pseudo-spin-valves annealed at high temperature," *Applied Physics Letters*, vol. 93, no. 8, 08 2008, 082508. [Online]. Available: https://doi.org/10.1063/1.2976435

[29] M. Imani, S. Gupta, Y. Kim, and T. Rosing, "Floatpim: In-memory acceleration of deep neural network training with high precision," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 802–815.

[30] M. Imani, Y. Kim, and T. Rosing, "Mpim: Multi-purpose in-memory processing using configurable resistive memory," in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2017, pp. 757–763.

[31] M. Khalifa, R. Ben-Hur, R. Ronen, O. Leitersdorf, L. Yavits, and S. Kvatinsky, "Filtpim: In-memory filter for dna sequencing," in *2021 28th IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*. IEEE, 2021, pp. 1–4.

[32] S. Kvatinsky, "Making real memristive processing-in-memory faster and reliable," in *2021 17th International Workshop on Cellular Nanoscale Networks and their Applications (CNNA)*. IEEE, pp. 1–3.

[33] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "Magic—memristor-aided logic," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 61, no. 11, pp. 895–899, 2014.

[34] D. Lee, W. Lee, H. Oh, and K. Yi, "Optimizing homomorphic evaluation circuits by program synthesis and term rewriting," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 503–518.

[35] T. Y. Lee, J. M. Lee, M. K. Kim, J. S. Oh, J. W. Lee, H. M. Jeong, P. H. Jang, M. K. Joo, K. Suh, S. H. Han, D.-E. Jeong, T. Kai, J. H. Jeong, J.-H. Park, J. H. Lee, Y. H. Park, E. B. Chang, Y. K. Park, H. J. Shin, Y. S. Ji, S. H. Hwang, K. T. Nam, B. S. Kwon, M. K. Cho, B. Y. Seo, Y. J. Song, G. H. Koh, K. Lee, J.-H. Lee, and G. T. Jeong, "World-most energy-efficient MRAM technology for non-volatile RAM applications," in *2022 International Electron Devices Meeting (IEDM)*, 2022, pp. 10.7.1–10.7.4.

[36] O. Leitersdorf, B. Perach, R. Ronen, and S. Kvatinsky, "Efficient error-correcting-code mechanism for high-throughput memristive processing-in-memory," in *58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021.

[37] O. Leitersdorf, R. Ronen, and S. Kvatinsky, "Making memristive processing-in-memory reliable," in *2021 28th IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*. IEEE, 2021, pp. 1–6.

[38] O. Leitersdorf, R. Ronen, and S. Kvatinsky, "Partitionpim: Practical memristive partitions for fast processing-in-memory," *arXiv preprint arXiv:2206.04200*, 2022.

[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 *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2016, pp. 1–6.

[40] Z. Li, H. Long, X. Zhu, Y. Wang, H. Liu, Q. Li, N. Xu, and H. Xu, "Error detection and correction method toward fully memristive stateful logic design," *Advanced Intelligent Systems*, vol. 4, no. 5, p. 2100234, 2022.

[41] H. Liu, D. Bedau, J. Sun, S. Mangin, E. Fullerton, J. Katine, and A. Kent, "Dynamics of spin torque switching in all-perpendicular spin valve nanopillars," *Journal of Magnetism and Magnetic Materials*, vol. 358, pp. 233–258, 2014.

[42] J.-C. Lo, S. Thanawastien, T. Rao, and M. Nicolaidis, "An sfs berger check prediction alu and its application to self-checking processor designs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 11, no. 4, pp. 525–540, 1992.

[43] Y. Long, X. She, and S. Mukhopadhyay, "Design of reliable dnn accelerator with un-reliable reram," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 1769–1774.

[44] Y. Lv, B. R. Zink, R. P. Bloom, H. Cılasun, P. Khanal, S. Resch, Z. Chowdhury, A. Habiboglu, W. Wang, S. S. Sapatnekar *et al.*, "Experimental demonstration of magnetic tunnel junction-based computational random-access memory," *arXiv preprint arXiv:2312.14264*, 2023.

[45] R. Micheloni, A. Marelli, and R. Ravasio, *Error correction codes for non-volatile memories*. Springer Science & Business Media, 2008.

[46] S. Motaman, S. Ghosh, and N. Rathi, "Impact of process-variations in sttram and adaptive boosting for robustness," in *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2015, pp. 1431–1436.

[47] H. Nejatollahi, S. Gupta, M. Imani, T. S. Rosing, R. Cammarota, and N. Dutt, "Cryptopim: In-memory acceleration for lattice-based cryptographic hardware," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[48] S. Resch, S. K. Khatamifard, Z. I. Chowdhury, M. Zabihi, Z. Zhao, H. Cilasun, J.-P. Wang, S. S. Sapatnekar, and U. R. Karpuzcu, "Mouse: Inference in non-volatile memory for energy harvesting applications," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 400–414.

[49] U. Schiffel, "Hardware error detection using an-codes," 2011.

[50] 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 *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2017, pp. 273–287.

[51] Z. Sun, X. Bi, and H. Li, "Process variation aware data management for stt-ram cache design," in *Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design*, 2012, pp. 179–184.

[52] N. Talati, S. Gupta, P. Mane, and S. Kvatinsky, "Logic design within memristive memories using memristor-aided logic (magic)," *IEEE Transactions on Nanotechnology*, vol. 15, no. 4, pp. 635–650, 2016.

[53] A. Technology, "Data Endurance, Retention and Field Immunity in STT-MRAM," 6 2021.

[54] M. S. Truong, E. Chen, D. Su, L. Shen, A. Glass, L. R. Carley, J. A. Bain, and S. Ghose, "Racer: Bit-pipelined processing using resistive memory," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 100–116.

[55] J. F. Wakerly, "Partially self-checking circuits and their use in performing logical operations," *IEEE Transactions on Computers*, vol. 100, no. 7, pp. 658–666, 1974.

[56] J.-P. Wang and J. D. Harms, "General structure for computational random access memory (cram)," Dec. 29 2015, uS Patent 9,224,447.

[57] J.-P. Wang, S. S. Sapatnekar, U. R. Karpuzcu, Z. Zhao, M. Zabihi, M. S. Resch, Z. I. Chowdhury, and T. Peterson, "Computational random access memory (cram) based on spin-orbit torque devices," Nov. 16 2021, uS Patent 11,176,979.

[58] L. Wei, J. G. Alzate, U. Arslan, J. Brockman, N. Das, K. Fischer, T. Ghani, O. Golonzka, P. Hentges, R. Jahan, P. Jain, B. Lin, M. Meterelliyoz, J. O'Donnell, C. Puls, P. Quintero, T. Sahu, M. Sekhar, A. Vangapaty, C. Wiegand, and F. Hamzaoglu, "A 7Mb STT-MRAM in 22FFL FinFET Technology with 4ns Read Sensing Time at 0.9V Using Write-Verify-Write Scheme and Offset-Cancellation Sensing Technique," in *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*, 2019, pp. 214–216.

[59] Y. Xie, B. Behin-Aein, and A. W. Ghosh, "Fokker—planck study of parameter dependence on write error slope in spin-torque switching," *IEEE Transactions on Electron Devices*, vol. 64, no. 1, pp. 319–324, 2016.

[60] X. Xin, Y. Zhang, and J. Yang, "Elp2im: Efficient and low power bitwise operation processing in dram," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 303–314.

[61] M. Zabihi, Z. I. Chowdhury, Z. Zhao, U. R. Karpuzcu, J.-P. Wang, and S. S. Sapatnekar, "In-memory processing on the spintronic cram: From hardware design to application mapping," *IEEE Transactions on Computers*, vol. 68, no. 8, pp. 1159–1173, 2018.

[62] Y. Zhang, X. Wang, H. Li, and Y. Chen, "Stt-ram cell optimization considering mtj and cmos variations," *IEEE transactions on magnetics*, vol. 47, no. 10, pp. 2962–2965, 2011.

[63] H. Zhao, B. Glass, P. K. Amiri, A. Lyle, Y. Zhang, Y.-J. Chen, G. Rowlands, P. Upadhyaya, Z. Zeng, J. A. Katine, J. Langer, K. Galatsis, H. Jiang, K. L. Wang, I. N. Krivorotov, and J.-P. Wang, "Sub-200 ps spin transfer torque switching in in-plane magnetic tunnel junctions with interface perpendicular anisotropy," *Journal of Physics*

D: Applied Physics, vol. 45, no. 2, p. 025001, dec 2011. [Online]. Available: https://doi.org/10.1088/0022-3727/45/2/025001
[64] W. Zhao, Y. Zhang, T. Devolder, J.-O. Klein, D. Ravelosona, C. Chappert, and P. Mazoyer, "Failure and reliability analysis of stt-mram," *Microelectronics Reliability*, vol. 52, no. 9-10, pp. 1848–1852, 2012.
[65] X. Zhu, H. Long, Z. Li, J. Diao, H. Liu, N. Li, and H. Xu, "Implication of unsafe writing on the magic nor gate," *Microelectronics Journal*, vol. 103, p. 104866, 2020.
[66] K. A. Zubair, S. K. Jha, D. Mohaisen, C. Hughes, and A. Awad, "Fat-pim: Low-cost error detection for processing-in-memory," 2022. [Online]. Available: https://arxiv.org/abs/2207.12231