

# Making Memristive Processing-in-Memory Reliable

Orian Leitersdorf, Ronny Ronen, and Shahar Kvatinsky

Viterbi Faculty of Electrical and Computer Engineering, Technion – Israel Institute of Technology, Haifa, Israel  
orianl@campus.technion.ac.il, ronny.ronen@technion.ac.il, shahar@ee.technion.ac.il

**Abstract**—Processing-in-memory (PIM) solutions vastly accelerate systems by reducing data transfer between computation and memory. Memristors possess a unique property that enables storage and logic within the same device, which is exploited in the memristive Memory Processing Unit (mMPU). The mMPU expands fundamental stateful logic techniques, such as IMPLY, MAGIC and FELIX, to high-throughput parallel logic and arithmetic operations within the memory. Unfortunately, memristive processing-in-memory is highly vulnerable to soft errors and this massive parallelism is not compatible with traditional reliability techniques, such as error-correcting-code (ECC). In this paper, we discuss reliability techniques that efficiently support the mMPU by utilizing the same principles as the mMPU computation. We detail ECC techniques that are based on the unique properties of the mMPU to efficiently utilize the massive parallelism. Furthermore, we present novel solutions for efficiently implementing triple modular redundancy (TMR). The short-term and long-term reliability of large-scale applications, such as neural-network acceleration, are evaluated. The analysis clearly demonstrates the importance of high-throughput reliability mechanisms for memristive processing-in-memory.

## I. INTRODUCTION

Emerging processing-in-memory (PIM) technologies possess ample potential for massive computational parallelism. By nearly eliminating the CPU-memory data transfer (*memory wall*) [1], PIM is capable of vastly accelerating computing systems [2]. The traditional memory read/write interface is supplemented with in-memory operations that perform logic on stored data without explicit read/write.

The memristor [3] enables true processing-in-memory as a fundamental device that supports both storage and logic. The resistance of a memristor may represent binary information (low resistance for logical 1, high resistance for logical 0). Crucially, the unique property of the memristor enables modifying resistance with an applied voltage. For memristors stored in a crossbar array structure, this unique property enables computing *stateful logic* between the memristors in the crossbar without explicitly reading/writing the memristor values [4]. As the same memristors are responsible for both storage and logic, memristive processing-in-memory is inherently supported.

Stateful logic within crossbar arrays supports massive inherent parallelism, which is utilized in the memristive Memory Processing Unit (mMPU) for efficient *high-throughput* operations. Stateful logic techniques include IMPLY [5], MAGIC [6], and FELIX [7]. These techniques support gates such as NOT/NOR, NAND, OR and Minority3 within rows/columns of crossbar arrays. They also inherently support

parallelism: the same in-row (in-column) gate can be repeated across all rows (columns) with the exact same latency. Therefore, the inherent logic capabilities of memristive memory can be utilized for massive parallelism with minimal peripheral overhead. The memristive Memory Processing Unit (mMPU) constructs an entire memory from crossbar arrays that each support this parallelism simultaneously [4], achieving high-throughput operation. Through a sequence of basic PIM gates, the unit is capable of advanced logic operation, such as matrix-vector multiplication [8]–[10] and image convolution [8], [11].

Reliability is an open challenge for memristive processing-in-memory. Soft-errors are categorized into those that alter memristor states over time (indirect), and those that lead to incorrect operations (direct). The former are traditionally addressed via the error correcting code (ECC) technique, while the latter are addressed through techniques such as triple modular redundancy (TMR) [12]. Processing-in-memory is especially vulnerable to these errors as they may propagate undetected through future computations, without the data being explicitly read. Previous works that address soft-errors for PIM are throughput-limited as they require periphery that is not compatible with PIM parallelism [13]–[15]. Therefore, we discuss novel techniques for high-throughput solutions that support reliable PIM by exploiting parallelism themselves.

This paper details architectural innovations for mMPU reliability, contributing the following:

- *Memristor Soft Errors*: Memristive soft errors are analyzed and categorized in the context of high-throughput memristive processing-in-memory.
- *Error-Correction-Codes (ECC)*: We detail non-traditional techniques that efficiently support high-throughput mMPU operations by exploiting mMPU parallelism themselves, expanding our previous work [16].
- *Triple-Modular-Redundancy (TMR)*: We propose efficient high-throughput in-memory TMR. A trade-off between latency, area and throughput is presented.
- *Case Study – Neural Network Accelerator*: We evaluate the proposed high-throughput ECC and TMR solutions in a case-study. This provides an insight into large-scale reliability for real high-throughput applications.

## II. BACKGROUND

### A. Stateful Logic

Memristive crossbar arrays locate memristors at the cross-points of vertical bitlines and horizontal wordlines. Each

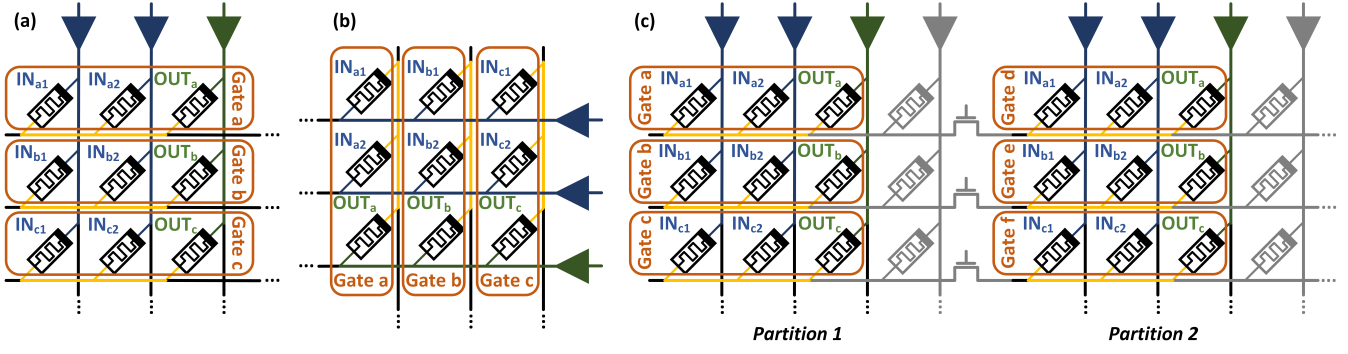


Fig. 1. MAGIC NOR (a) in-row and (b) in-column operations, each scenario performed simultaneously. The logical inputs are the resistive states of the input memristors prior to the operation, and the final output resistances represent the logical outputs. (c) Transistors may divide the crossbar into partitions that may function as independent computation units, further increasing parallelism by also enabling multiple in-row (in-column) gates within the same row (column).

memristor stores binary information through resistance (*i.e.*, high resistance for logical 0, and low resistance for logical 1). Stateful logic considers performing logic operations between the states of the memristors, using the memristors, without explicitly reading/writing their values. This is achieved by exploiting the unique property of the memristor: voltage affects resistance. By applying voltages on the bitlines (wordlines) of the crossbar, it is possible to induce logic between memristors in the same row (column). The resistance of the input memristors prior to the operation represents the logical inputs, and the resistance of the output memristor afterwards represents the logical output. Emerging stateful logic techniques include IMPLY [5], MAGIC [6], and FELIX [7]. They support logical gates such as NOT/NOR, NAND, OR and Minority3.

Stateful logic inherently supports row/column parallelism within crossbar arrays. As stateful logic operations are performed by applying voltages along bitlines (wordlines), the same voltages may induce logic in all of the rows (columns) of the crossbar at the same time, as seen in Figures 1(a,b). Additionally, partitions may dynamically divide the crossbar array to support multiple in-row (in-column) gates in the same row (column) in parallel [7], as shown in Figure 1(c).

### B. Memristor Soft Errors

Memristors are vulnerable to soft errors, similar to other memory technologies [17]. Soft errors alter the logical state of the memristor (*i.e.*, the resistance) without harming its functionality. Conversely, hard errors permanently damage memristor functionality. Hard errors can be addressed with various testing circuits [18], [19], yet soft errors are more difficult to detect. Soft errors in PIM may propagate through consecutive operations without the data ever being explicitly read, while traditional reliability techniques are designed across the read/write interface. Furthermore, the high-throughput mMPPU may access/update massive amounts of data at once, which increases the potential for soft errors. We categorize soft errors into *indirect* and *direct* soft errors.

1) *Indirect Soft Errors*: These soft errors generally occur over time, and are divided into the following types:

- *Retention and state-drift*: The resistance of a memristor drifts over time, *e.g.*, due to diffusion of oxygen vacancies [19]–[21]. Furthermore, read (logic) operations may induce state-drift in the (input) memristors [22]–[25].
- *Proximity*: When operations on nearby memristors affect this memristor, *e.g.*, read/write disturbance [19].
- *Abrupt*: Rare occurrences that change the resistive state at once, *e.g.*, due to ion-strikes [26]. Statistically, these typically affect data stored over time.

2) *Direct Soft Errors*: These soft errors occur at once, due to a failure in a specific operation, such as:

- *Write failures*: When a write is unsuccessful [27].
- *Incorrect logic*: When a stateful logic gate is incorrect, *e.g.*, due to variability in the resistances and properties of the input and output memristors [13], [21]–[25], [28].

## III. MEMRISTIVE MEMORY PROCESSING UNIT (mMPPU)

The mMPPU constructs an entire memory architecture based on memristive stateful logic for high-throughput computation.

### A. Parallelism

The mMPPU enables massively high-throughput operation by utilizing three forms of parallelism:

- *Row/Column Parallelism*: Logic repetition across multiple rows/columns, as explained in Section II-A. Initially, this was utilized to reduce latency [29], [30], yet recent efforts focus on maximizing throughput [8]–[11], [31], [32]. Single-row (single-column) arithmetic is performed within a single row (column), enabling concurrent execution across multiple rows (columns) for vector operations.
- *Partition Parallelism*: Transistors may dynamically divide the crossbar array into memristive partitions to enable further parallelism [7]. Partitions within rows (columns) enable concurrent execution of multiple in-row (in-column) logic gates. As a recently emerging technology, partitions appear to possess vast potential [7], [9], [16], [33], [34].
- *Crossbar Parallelism*: The mMPPU consists of multiple crossbar arrays that can operate in parallel [35].

These three parallelism forms enable massive throughput, capable of tremendously accelerating many applications.

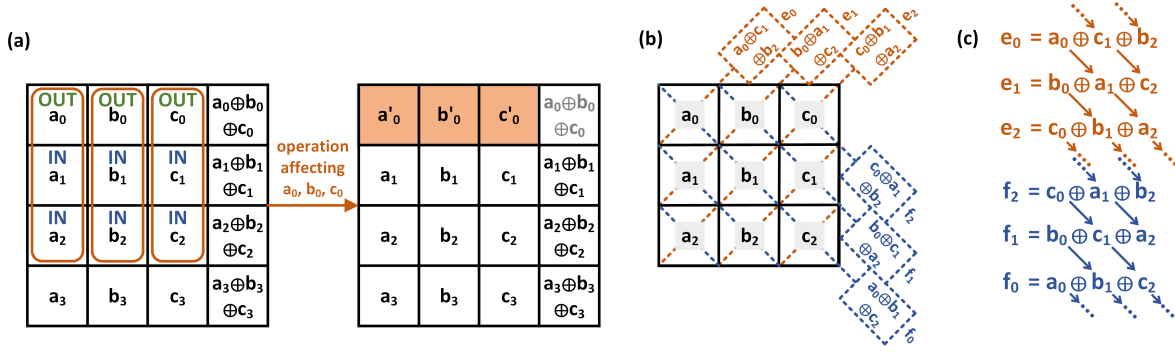


Fig. 2. (a) Naive ECC solution that stores parity bits horizontally. An in-column operation is not compatible with this solution as the top-right check-bit has all  $n$  of its data-bits updated (in one cycle). Thus,  $O(n)$  cycles are required for ECC update. (b) The proposed solution that stores parity along diagonals to enable high-throughput operation in all cases. (c) The shift pattern in the diagonal technique that enables implementation via barrel shifters [30], [36].

### B. Arithmetic Functions

The mMPU architecture expands the basic stateful logic gates (e.g., NOR) to intra-crossbar arithmetic functions (e.g., addition [7]–[9], [35], [37], [38], multiplication [9], [32], [33]). The mapping of the Boolean functions to in-memory logic gates is performed either manually [9], [32], [33] or via automated tools [31], [39]. The Boolean functions are typically mapped to a single row (column) to allow concurrent execution across multiple rows (columns). This provides vectored operations with low latency, such as vector addition and element-wise vector multiplication. Additional functions, such as matrix-vector multiplication [8]–[10], are also supported.

The mMPU performs operations at the arithmetic scale, which are utilized for larger applications. The mMPU controller [40], [41] receives instructions from the CPU to perform an arithmetic function (e.g., vector addition) within certain crossbar arrays, and converts the function to stateful logic gates via the aforementioned mapping techniques [9], [31]–[33], [39]. Larger applications utilize these in-memory arithmetic functions with libraries at the CPU-level to enable more complex operations, while still requiring low CPU/memory data transfer as only the instructions are communicated.

### IV. MEMRISTIVE ERROR-CORRECTING-CODES

The error correcting code (ECC) technique utilizes redundant data (check-bits) to improve the long-term reliability of the original information (data-bits) [12]. ECC for the mMPU primarily addresses *indirect* soft errors by protecting the stored data over time.

The proposed ECC mechanisms are provided on a per-function basis. Each arithmetic function that the mMPU supports utilizes input, intermediate, and output memristors. The proposed solution verifies the correctness of the input data before function execution, and updates the ECC for the output memristors following the execution. This follows from the fact that the function outputs may serve as future inputs. Soft-errors in intermediate memristors are addressed in Section V.

Traditionally, ECC is implemented along the memory read/write interface with low throughput [27]. The lack of such an interface in the mMPU presents the first challenge:

data may be accessed and altered within the memory without flowing through the memory interface. Yet, a larger challenge arises from the high-throughput computation supported by the mMPU. As the mMPU is capable of processing vast amounts of data at once (e.g. approximately 100 TB/sec for 8192 crossbars, each sized  $1024 \times 1024$ , consuming only 1GB of memory [35]), the ECC solutions must be capable of *high-throughput* verification and computation. This is achieved by utilizing the mMPU parallelism itself. When an arithmetic function is repeated across all rows (columns), then the inputs/outputs of the function (which require ECC verify/compute) are located across columns (rows) of data. Therefore, high-throughput ECC must support efficient verification/computation for columns/rows of data.

The naive solution would be to store check-bits within the memory in a horizontal configuration. For example, designating the eighth bit of every horizontal byte as a parity bit, as illustrated in Figure 2(a). Consider an in-row logic gate (e.g., NOR) that is repeated along all of the crossbar rows (similar to Figure 1(a)). Assuming that the crossbar starts at a valid state (parity bits matching stored data), we seek to reach a valid state that reflects the newly-stored output data. An entire column of information was altered within a single cycle, and the parity bits can be updated in  $O(1)$  cycles by utilizing exclusive-or linearity with the same multi-row parallelism (new parity bit can be computed given only old parity bit, old data bit, and new data bit). Unfortunately, when an in-column operation is performed along all columns (similar to Figure 1(b)), then  $O(n)$  cycles are needed to update the parity-bits, as illustrated in Figure 2(a). Therefore, high-throughput ECC is only supported in the naive solution for some of the potential logic and arithmetic operations.

The mMPU-compatible solution stores parity-bits along wrap-around diagonals instead of horizontally/vertically [16]. The diagonal pattern emerges from the unique computation capabilities of the mMPU, enabling  $O(1)$  overhead support for all potential user operations. By storing parity-bits along both leading and counter diagonals, single error-correction is achieved, through multi-dimensional parity [42], per  $m \times m$

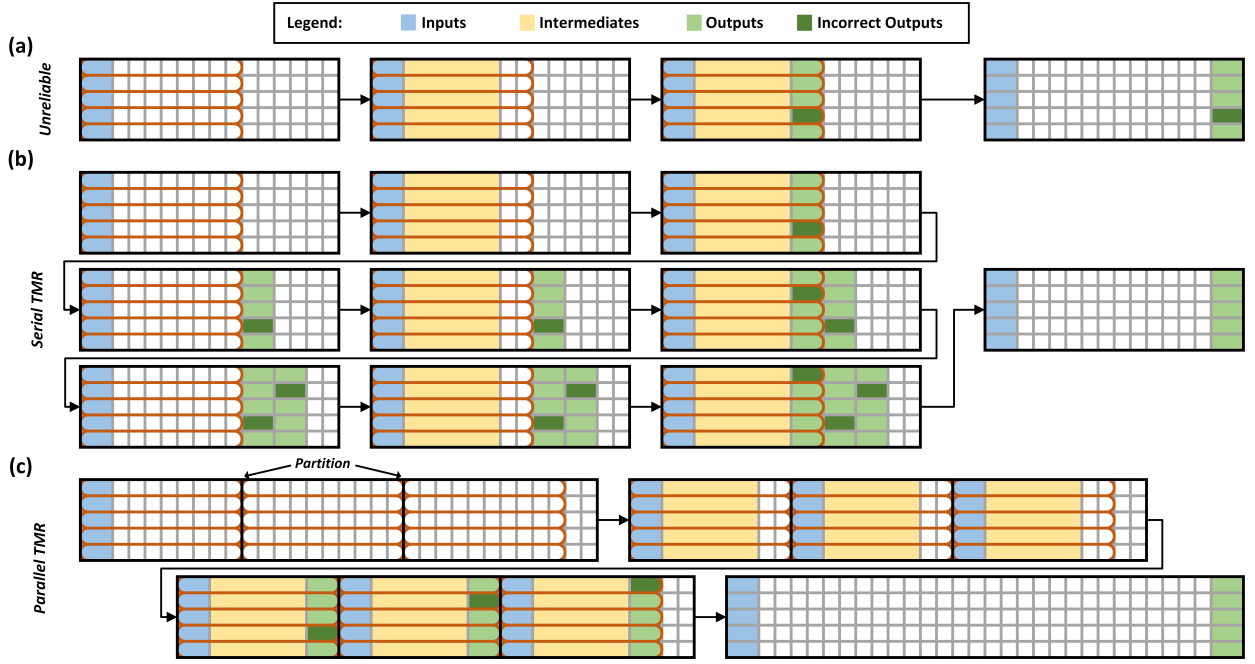


Fig. 3. (a) Unreliable computation of a single-row function in parallel across rows. Reliable computation of the same function through the proposed (b) serial and (c) parallel TMR solutions.

block in the  $n \times n$  crossbar (e.g.,  $m \approx 16, n \approx 1024$ ). The diagonal parity-bits are illustrated in Figure 2(b) as diagonal extensions to the block (for illustration purposes). The check-bits are stored in a dedicated extension that is also based on memristive memory. The communication between the crossbars utilizes a barrel shifter [30], [36] to emulate diagonal wires, following the pattern shown in Figure 2(c). While the barrel shifter is peripheral, the communication between the crossbars remains stateful (similar to partitions). The dedicated extension may work in parallel to the main memory, enabling moderate latency overhead of 26% on average [16].

## V. MEMRISTIVE TRIPLE MODULAR REDUNDANCY

The triple modular redundancy (TMR) technique improves logic reliability through a simple concept: compute the same function three times and vote between the results [12]. TMR for the mMPU primarily addresses *direct* soft-errors.

Consider a single-row arithmetic function (e.g., multiplication [9]) that is repeated across all of the crossbar rows (e.g., element-wise vector multiplication [9]), as illustrated in Figure 3(a). Errors in the stateful gates of the function, in any of the rows, may<sup>1</sup> cause incorrect output. That incorrect output may propagate erroneous values when used as a future input.

The proposed serial solution increases reliability through high-throughput TMR. The naive solution is repeated three times, using the same inputs and re-using the intermediate memristors. The outputs are stored in three separate copies. At the end, voting is accomplished using the Minority3 logic gate [7] with parallelism across all of the rows. As shown in Figure 3(b), while each iteration encountered a soft-error,

the final output is correct. This solution increases latency by approximately  $3\times$ , while area is only slightly increased (as input/intermediate memristors are shared across iterations).

The proposed parallel solution also utilizes high-throughput TMR, yet reduces the latency overhead with memristive partitions. Rather than performing the naive solution three times in an iterative fashion, the iterations are executed concurrently with memristive partitions [7]. In this case, inputs and intermediates cannot be shared without compromising partition independence. Hence, while latency remains constant (voting time is negligible due to high-throughput voting using the Minority3 gate), area increases by  $3\times$  (as reuse is not possible). Note that a semi-parallel solution that does not utilize partitions is possible by repeating the function computation across additional rows (thereby reducing *throughput* by  $3\times$ ).

As the proposed voting is performed per-bit, effective reliability is increased. That is, incorrect final output requires that at least two copies have errors at the exact same bits. For example, consider a scenario where voting is performed between outputs 1000, 0100, 0010. Per-element voting would result in an error as no two copies agree on the final output. Conversely, per-bit voting will choose 0000. Per-bit voting may only increase reliability over per-element voting, as they differ only when per-element voting is undefined.

The proposed TMR solutions efficiently support the high-throughput mMPU. By exploiting the same PIM parallelism as the mMPU for *high-throughput* TMR, the TMR overhead is relatively low, i.e.,  $3\times$  latency,  $1\times$  area (serial) or  $1\times$  latency and  $3\times$  area (parallel) compared to an *unreliable baseline*. Conversely, solutions that utilize crossbar periphery may not be compatible with PIM parallelism, thereby requiring up to  $1024\times$  latency increase (for 1024 rows) [13], [14].

<sup>1</sup>Some gate errors are masked and do not result in incorrect function output.

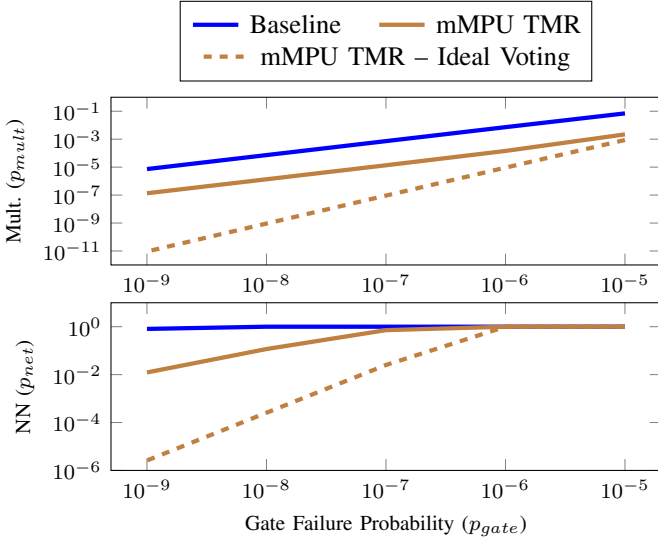


Fig. 4. The computation reliability for the unreliable baseline and the proposed TMR, comparing both multiplication failure probability (top) and neural network (NN) failure probability (bottom). At  $p_{gate} = 10^{-9}$ , the baseline network has a soft-error-induced miss-classification rate of 74%, while the TMR network has approximately 2% (below the network’s inherent accuracy). The proposed TMR with ideal voting is shown in dashed brown.

## VI. CASE STUDY: NEURAL-NETWORK ACCELERATION

This section considers the reliability of large-scale applications, assessing the combination of ECC and TMR. We begin by discussing the *computation* reliability of a single fixed-point multiplication, based on the MultPIM algorithm [9] and the proposed TMR solutions. We then utilize these results to consider the feed-forward compute reliability of an in-memory neural-network accelerator, such as FloatPIM [8], under the presence of *direct* soft-errors. Lastly, we analyze the weight degradation over time due to *indirect* soft-errors.

### A. Multiplication Reliability

We modify the simulator from MultPIM [9] to evaluate the overall reliability provided by the proposed solutions under the presence of *direct* soft-errors. We consider 32-bit full-precision multiplication according to the state-of-the-art algorithm [9]. The simulation accounts for error *masking* in the algorithm. The original simulator involved requests from the algorithm micro-code to perform stateful gates; we inject soft-errors into these requests and measure the logical masking. For the baseline, the MultPIM algorithm is run as usual, and then the final output is compared to the true product. For the proposed solution, the algorithm is performed three times and then voting is performed using in-memory Majority3 (also vulnerable to soft-errors). The probability of multiplication failure,  $p_{mult}$ , is shown in Figure 4 (top) for varying probabilities of single-gate error,  $p_{gate}$ . Interestingly, the non-ideal voting becomes the bottleneck near  $p_{gate} = 10^{-9}$ , as shown by the dashed line that indicates a *theoretical* ideal-voting solution.

### B. Neural Network Reliability

The reliability of a large-scale neural network accelerator is considered, based on the FloatPIM [8] accelerator,

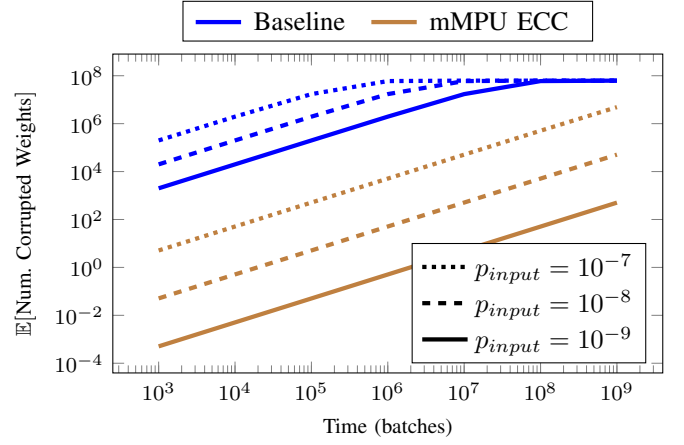


Fig. 5. Expected weight corruption weights for the baseline and the mMPU ECC, at varying times (number of batches) with varying  $p_{input}$ .

the AlexNet [43] model (32-bit fixed-point), and the ImageNet [44] classification dataset. Neural network reliability consists of two aspects: reliable feed-forward *computation* with direct soft-errors, and weight *degradation* due to indirect soft-errors. Both affect the accuracy of the network, while the former affects the short-term accuracy and the latter causes accuracy degradation over time.

1) *Feed-Forward Computation Reliability*: Feed-forward computation reliability is evaluated based on the multiplication compute reliability, and on previous works that explore error propagation in deep neural networks. Multiplication accounts for the vast majority of the computation in the FloatPIM implementation of AlexNet; thus, we focus only on multiplications. The model requires  $M = 612 \cdot 10^6$  multiplication per sample.

The network has inherent logical masking characteristics, thus incorrect multiplications may have no effect on the final classification. We consider the soft-error propagation analysis performed by G. Li *et al.* [45] as the multiplication compute errors here are analogous to the soft-errors they explore (they affect the same intermediate states of the network). For AlexNet, they find that only  $p_{mask} = 0.03\%$  of soft-errors affect the final network classification. Therefore, the probability that a single multiplication causes an incorrect classification is approximately  $p_{mask} \cdot p_{mult}$ . Assuming independence across multiplication reliability, the probability of incorrect final classification due to a soft error is  $1 - (1 - p_{mask} \cdot p_{mult})^M$ . These results are plotted in Figure 4 (bottom) for varying  $p_{gate}$ . We find that the feed-forward compute error is approximately 2% for  $p_{gate} \leq 10^{-9}$  using the proposed mMPU TMR (non-ideal voting). Considering that the neural network itself has classification error  $\approx 27\%$ , this compute error is negligible.

2) *Weight Degradation*: Weight-degradation is evaluated based on *indirect* soft-errors. As the neural network accelerator constantly accesses all of the  $W = 62\text{M}$  weights for every batch, we focus on indirect soft-errors that occur as a result of accessing the memristors. Assuming that accessing a single bit may corrupt that bit with probability  $p_{input}$ , the probability that a single batch corrupts a 32-bit weight is extrapolated. Then, the probability that that specific weight is corrupted



over  $T$  consecutive batches is extrapolated, and shown in Figure 5. We find that the baseline (no ECC) results in nearly all of the weights corrupted after only  $10^7$  batches, while the mMPU ECC maintains an expectation of approximately a single corrupted weight at  $10^7$  batches with  $p_{input} = 10^{-9}$ .

## VII. CONCLUSION AND DISCUSSION

The memristive Memory Processing Unit (mMPU) achieves high-throughput operation by exploiting inherent stateful logic parallelism, and by efficiently expanding fundamental logic gates to fast arithmetic functions. Unfortunately, this high-throughput is not compatible with the traditional throughput-limited reliability techniques that address *indirect* and *direct* soft-errors. We detail non-conventional, high throughput, error-correcting-code (ECC) techniques that are designed specifically for the mMPU to address *indirect* errors, as well as high throughput triple-modular-redundancy (TMR) solutions to address *direct* errors. The case study of in-memory neural-network acceleration demonstrates the large-scale implications of the mMPU ECC and TMR, considering both short-term feed-forward accuracy and long-term weight degradation.

The neural network case study presents a unique perspective as the classifier possess massive logical masking capabilities, and as the inherent miss-classification rate is already relatively high. Applications that do not possess these traits, such as database acceleration, will require stronger reliability mechanisms to address memristor soft-errors. Further, such mechanisms may relax the required soft error rates in neural-network acceleration, bridging the gap between the architectural requirements and the physical technology. These mechanisms may include generalizations of TMR, and error-correcting codes with higher correction capability. They must remain high-throughput to efficiently support the mMPU, ideally by utilizing the same mMPU parallelism for reliability.

## ACKNOWLEDGMENT

This work was supported in part by the European Research Council through the European Union's Horizon 2020 Research and Innovation Programme under Grant 757259, and in part by the Israel Science Foundation under Grant 1514/17.

## REFERENCES

- [1] A. Pedram *et al.*, "Dark memory and accelerator-rich system optimization in the dark silicon era," *IEEE Design & Test*, 2017.
- [2] R. Balasubramanian and B. Grot, "Near-data processing," *Micro*, 2016.
- [3] L. Chua, "Memristor-the missing circuit element," *IEEE TCT*, 1971.
- [4] S. Kvatinsky, "Real processing-in-memory with memristive memory processing unit (mMPU)," in *IEEE ASAP*, 2019.
- [5] J. Borghetti *et al.*, "Memristive switches enable 'stateful' logic operations via material implication," *Nature*, 2010.
- [6] S. Kvatinsky *et al.*, "MAGIC—memristor-aided logic," *TCAS-II*, 2014.
- [7] S. Gupta *et al.*, "FELIX: Fast and energy-efficient logic in memory," in *ICCAD*, 2018.
- [8] M. Imani *et al.*, "FloatPIM: In-memory acceleration of deep neural network training with high precision," in *ISCA*, 2019.
- [9] O. Leitersdorf *et al.*, "MultiPIM: Fast stateful multiplication for processing-in-memory," *TCAS-I*, 2021.
- [10] A. Eliahu *et al.*, "abstractPIM: Bridging the gap between processing-in-memory technology and instruction set architecture," in *VLSI-SOC*, 2020.
- [11] A. Haj-Ali *et al.*, "IMAGING: In-memory algorithms for image processing," *TCAS-I*, 2018.

- [12] M. L. Shooman, *Reliability of Computer Systems and Networks: Fault Tolerance, Analysis, and Design*. USA: John Wiley & Sons, Inc., 2002.
- [13] J. Xu *et al.*, "In situ aging-aware error monitoring scheme for IMPLY-based memristive computing-in-memory systems," *TCAS-I*, 2021.
- [14] X. Zhu *et al.*, "Memristive stateful logic with N-modular redundancy error correction design towards high reliability," in *EDTM*, 2021.
- [15] F. M. Puglisi *et al.*, "SIMPLY: Design of a RRAM-based smart logic-in-memory architecture using RRAM compact model," in *ESSDERC*, 2019.
- [16] O. Leitersdorf *et al.*, "Efficient error-correcting-code mechanism for high-throughput memristive processing-in-memory," in *DAC*, 2021.
- [17] C. Slayman, "Soft error trends and mitigation techniques in memory devices," in *RAMS*, 2011.
- [18] P. Liu *et al.*, "Fault modeling and efficient testing of memristor-based memory," *TCAS-I*, 2021.
- [19] S. Swami and K. Mohanram, "Reliable nonvolatile memories: Techniques and measures," *IEEE Design & Test*, 2017.
- [20] A. M. S. Tossou *et al.*, "RRAM refresh circuit: A proposed solution to resolve the soft-error failures for HfO<sub>2</sub>/Hf 1T1R RRAM memory cell," in *GLSVLSI*, 2016.
- [21] S. Wiefels *et al.*, "Reliability aspects in resistively switching valence change memory cells," *Lehrstuhl für Werkstoffe der Elektrotechnik II und Institut für Werkstoffe*, Tech. Rep., 2021.
- [22] A. Siemon *et al.*, "Memristive device modeling and circuit design exploration for computation-in-memory," in *ISCAS*, 2019.
- [23] M. Escudero *et al.*, "Memristive logic in crossbar memory arrays: Variability-aware design for higher reliability," *TNANO*, 2019.
- [24] N. Wald and S. Kvatinsky, "Understanding the influence of device, circuit and environmental variations on real processing in memristive memory using memristor aided logic," *MEJ*, 2019.
- [25] X. Zhu *et al.*, "Implication of unsafe writing on the MAGIC NOR gate," *Microelectronics Journal*, 2020.
- [26] A. Abubakr *et al.*, "The impact of soft errors on memristor-based memory," in *New Generation of CAS*, 2017.
- [27] D. Niu *et al.*, "Low power memristor-based ReRAM design with error correcting code," in *ASP-DAC*, 2012.
- [28] X. Zhu *et al.*, "Unsafe writing impacts on the stateful memristor gates," in *ISCAS*, 2020.
- [29] R. Ben Hur *et al.*, "SIMPLE MAGIC: Synthesis and in-memory mapping of logic execution for memristor-aided logic," in *ICCAD*, 2017.
- [30] M. Imani *et al.*, "Ultra-efficient processing in-memory for data intensive applications," in *DAC*, 2017.
- [31] R. Ben-Hur *et al.*, "SIMPLER MAGIC: Synthesis and mapping of in-memory logic executed in a single row to improve throughput," *IEEE TCAD*, 2020.
- [32] A. Haj-Ali *et al.*, "Efficient algorithms for in-memory fixed point multiplication using MAGIC," in *ISCAS*, 2018.
- [33] Z. Lu *et al.*, "RIME: A scalable and energy-efficient processing-in-memory architecture for floating-point operations," in *ASP-DAC*, 2021.
- [34] M. R. Alam *et al.*, "Sorting in memristive memory," 2021. [Online]. Available: arXiv:2012.09918
- [35] R. Ronen *et al.*, "The bitlet model: A parameterized analytical model to compare PIM and CPU systems," *JETC*, 2021.
- [36] S. Gupta *et al.*, "NNPIM: A processing in-memory architecture for neural network acceleration," *IEEE TC*, 2019.
- [37] N. Talati *et al.*, "Logic design within memristive memories using memristor-aided logic (MAGIC)," *TNANO*, 2016.
- [38] S. Kvatinsky *et al.*, "Memristor-based material implication (IMPLY) logic: Design principles and methodologies," *TVLSI*, 2014.
- [39] F. Wang *et al.*, "STAR: Synthesis of stateful logic in RRAM targeting high area utilization," *TCAD*, 2020.
- [40] R. Ben Hur and S. Kvatinsky, "Memristive memory processing unit (MPU) controller for in-memory processing," in *ICSEE*, 2016.
- [41] N. Talati *et al.*, "CONCEPT: A column-oriented memory controller for efficient memory and PIM operations in RRAM," *IEEE Micro*, 2019.
- [42] J. M. Shea and T. F. Wong, *Multidimensional Codes*. John Wiley & Sons, Inc., 2003.
- [43] A. Krizhevsky *et al.*, "Imagenet classification with deep convolutional neural networks," *NeurIPS*, 2012.
- [44] J. Deng *et al.*, "Imagenet: A large-scale hierarchical image database," in *IEEE CVPR*, 2009.
- [45] G. Li *et al.*, "Understanding error propagation in deep learning neural network (DNN) accelerators and applications," in *SC*. ACM, 2017.