On Endurance of Processing in (Nonvolatile) Memory

Salonik Resch
resc0059@umn.edu
University of Minnesota

Husrev Cilasun
cilas001@umn.edu
University of Minnesota

Zamshed Chowdhury
chowh005@umn.edu
University of Minnesota

Masoud Zabihi
zabih003@umn.edu
University of Minnesota

Zhengyang Zhao
zhaox526@umn.edu
University of Minnesota

Jian-Ping Wang
jpwang@umn.edu
University of Minnesota

Sachin Sapatnekar
sachin@umn.edu
University of Minnesota

Ulya R. Karpuzcu
ukarpuzc@umn.edu
University of Minnesota

ABSTRACT

Processing-in-Memory (PIM) architectures have gained popularity due to their ability to alleviate the memory wall by performing large numbers of operations within the memory itself. On top of this, nonvolatile memory (NVM) technologies offer highly energy-efficient operations, rendering processing in NVM especially promising. Unfortunately, a major drawback is that NVM has limited endurance. Even when used for standard memory, nonvolatile technologies face limited lifetimes, which is exacerbated by imbalanced usage of memory cells. PIM significantly increases the number of operations the memory is required to perform, making the problem much worse. In this work, we quantitatively analyze the impact of PIM applications on endurance considering representative memory technologies. Our findings indicate that limited endurance can easily block the performance and energy efficiency potential of PIM architectures. Even the best known technologies of today can fall short of meeting practical lifetime expectations. This highlights the importance of research efforts to improve endurance especially at the device technology level. Our study represents the first step in characterizing the very demanding endurance needs of PIM applications to derive a detailed technology level design specification.

CCS CONCEPTS

• **Hardware** → **Emerging technologies**.

KEYWORDS

processing in memory, endurance, nonvolatile memory

ACM Reference Format:
Salonik Resch, Husrev Cilasun, Zamshed Chowdhury, Masoud Zabihi, Zhengyang Zhao, Jian-Ping Wang, Sachin Sapatnekar, and Ulya R. Karpuzcu. 2023. On Endurance of Processing in (Nonvolatile) Memory . In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23), June 17–21, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3579371.3589114

1 INTRODUCTION

The performance of modern computing systems is limited by the performance of the memory. This is because CPU performance has increased more rapidly than memory performance for the past few decades [15]. This limitation is referred to as the *memory wall*, and it has created the search for new architectures which do not suffer from this problem. A promising candidate is processing-in-memory (PIM) which can perform computation directly in memory. PIM architectures can enhance both performance and energy efficiency of numerous emerging applications significantly [33]. Non-volatile PIM is particularly interesting due to its extreme energy efficiency [6, 20, 21] and high density [31, 43].

Unfortunately, nonvolatile memory (NVM) devices suffer from a low endurance. Memory cells can only be written a certain number of times before failing. The endurance varies significantly between technologies, but all are at risk of premature failure, which would render them impractical. Hence, there has been much research into evenly distributing (standard memory) write operations across NVM cells (i.e., load-balancing) [13, 26, 27]. However, PIM significantly changes the access patterns and drastically increases the number of write operations. For example, our analysis shows that an in-memory multiplication requires *over 150× more write operations* than it would require in a conventional architecture. Not only are there more writes, but they are more difficult to balance in PIM than in standard memory. PIM computations must be triggered where the relevant data is stored in memory. There is less flexibility for remapping PIM computations, and hence the resulting writes, when compared to standard memory writes. We therefore need to revisit strategies which are sufficient for NVM in the context of nonvolatile PIM (NVPIM).

In this work, we characterize the impact of PIM operations on the endurance of representative NVM technologies. We start by investigating write patterns of PIM applications to estimate expected lifetime of PIM arrays. We test basic strategies to improve endurance and show their effectiveness. Our findings indicate that NVPIM is significantly limited by endurance, underlining an urgent need for progress at the device technology level.

The contributions of this work are as follows:

- We show that PIM induces many more writes in memory compared to traditional architectures.
- We show how PIM arrays are particularly susceptible to failed memory cells.
- We provide estimates for expected lifetimes of PIM arrays considering characteristic workloads.
- We adapt load-balancing principles from standard memory to implement generalizable and simple load-balancing strategies for PIM and demonstrate how they can improve lifetime for characteristic workloads.

The rest of the paper is structured as follows: Section 2 provides a background on PIM devices, PIM architectures, and how logic is implemented. Section 3 covers typical data layout patterns and the natural consequences of standard operation. Section 3.1 investigates the impact of limited endurance. Section 3.2 introduces strategies to improve lifetime with load balancing specific to PIM. We discuss the possibility of using PIM arrays with failed devices in Section 3.3. In Section 4 we set up the evaluation, including commonly used PIM

This work was partially supported by a Cisco Research Fellowship.

ISCA '23, June 17–21, 2023, Orlando, FL, USA

ACM 979-8-4007-0095-8/23/06…$15.00
https://doi.org/10.1145/3579371.3589114

applications and how they are laid out in memory. In Section 5 we show write distributions of these applications inside PIM arrays, the impact of basic load-balancing strategies, and remaining problems due to limited endurance. Finally, we conclude in Section 7.

2 BACKGROUND

In this section we cover NV devices used for PIM, how logic operations are performed in memory, and how we can synthesize complex computations using these logic operations.

2.1 Nonvolatile Devices

In this paper, we consider NVMs which hold their state in their resistance. The state of a device can be determined by applying a voltage and sensing the current that flows through it, which constitutes a read. The specifics for changing the state (write operation) varies between technologies. We now cover the most promising representatives.

Magnetic RAM (MRAM) is based on Magnetic Tunnel Junctions (MTJs) which have two magnetic layers, a fixed layer and a free layer. If the layers are aligned, the MTJ has low resistance (parallel state); if not aligned; high resistance (anti-parallel state). The state can be changed by driving a current of sufficiently high magnitude through the MTJ, where the direction of the current determines the state. When electrons flow from the free (fixed) layer to the fixed (free) layer, the MTJ is put into the anti-parallel (parallel) state. The main advantages of MTJs are relatively high density and high endurance with respect to other nonvolatile technologies. Specifically, when it comes to endurance, current MTJs can switch as many as 10^{12} times [23, 34] until permanent failure. In contrast to its competitors, MTJ writes involve no moving atoms. Hence, it is believed that the write endurance can be drastically improved with newer generations of devices [18]. A disadvantage of MTJs is that the resistance difference in the anti-parallel and parallel states is relatively low, which makes them more sensitive to noise such as voltage fluctuations.

Resistive RAM (RRAM) consists of a metal-insulator-metal stack [11]. Applying a voltage differential causes the formation of a conductive filament, creating a low resistance state. Applying a voltage differential in the opposite direction removes this filament and creates a high resistance state [3]. RRAM has more than 2 possible states, as it can take on a range of resistance values between the two extremes. However, it is common in practice to use only the highest and the lowest resistance states to reduce noise. An advantage of RRAM is the high ratio between the high and low resistance states, reducing sensitivity to noise. A major drawback is limited endurance, currently permitting approximately 10^8-10^9 writes before failure [18, 35, 46].

Phase-Change Memory (PCM) has a state based on the structure of the atoms within the channel for electric current [40]. The write operation involves heating up the device by passing an electrical current through it. Quickly reducing the current causes the device to cool rapidly into an amorphous state, which has high resistance. Cooling the device slowly allows it to form a more uniform structure, which has low resistance. PCM currently allows around 10^6 - 10^9 writes before failure [18, 19].

NVM is an emerging technology, hence device characteristics are subject to change. Numerous works have predicted orders of magnitude improvements to the write endurance [18, 37].

2.2 PIM Architectures

PIM architectures modify traditional memory hardware to enable logic operations to occur either within or very close the memory. In this work, we consider architectures which perform Boolean (digital) logic operations directly in memory[1], where the input and outputs of every operation are memory cells in the array. Such architectures maintain the basic memory hardware and operating semantics, allowing them to replace standard memory structures with little modification to the overall architecture. Additionally, hardware modification to the memory array itself is relatively modest. Typical solutions involve adding additional bitlines [29, 31], row-decoders which support multi-row activation [21, 33], or extra transistors in each cell [6, 43]. These architectures have demonstrated high performance and energy efficiency both for use in traditional computing systems [21, 31, 33] and in embedded systems [29, 30] for commercially relevant applications.

Regardless of the specific approach, the mechanics of PIM operations on memory devices are nearly identical. Current is passed through one or two input memory devices, and a single output memory device is written to. Therefore, we can abstract out the specific cell design when analyzing endurance. In the remainder of this section, we discuss PIM operating semantics accordingly.

Basic Logic Operations (Gates): PIM architectures enable logic operations at the bit level directly in memory. Basic logic operations –such as NOT, (N)AND, or (N)OR)– take one or two memory bit cells as input, compute the output, and store the result in another bit cell, which is typically in the same row or column as the inputs. Depending on the architecture, this can be done with or without the involvement of sense amplifiers. For architectures using sense amplifiers [21], the procedure is:

(1) Read multiple input cells simultaneously.
(2) To calculate the output according to the underlying truth table, perform thresholding using the sense amplifier.
(3) Write back the result to the designated output cell.

For architectures which do not use sense amplifiers [31]:

(1) Apply a voltage differential on the bitlines connecting the inputs and outputs.
(2) Current travels through the inputs to outputs, conditionally switching the output according to the truth table of the operation being performed.

Both approaches are shown in Fig.1, for column based computations without loss of generality. Regardless of the approach, input cells effectively go through *read* operations and the output cell goes through a *write* operation. If possible logic operations form a universal set, any computation can be carried out in the respective row or column, limited by the number of memory cells available.

Complex Logic Operations: In traditional architectures, an arithmetic logic unit (ALU) can be used to perform complex logic operations relatively quickly. For example, addition and multiplication can be performed within a few cycles of the system clock. In contrast, PIM architectures require a series of logic gates to perform such operations.

The operation must be decomposed into a set of gates the architecture is capable of (e.g., NOT, AND, NAND). Each of these gates must then be scheduled within the array. When processing within a single column (or row), only a single logic gate can be performed at a time due to *structural hazards* – the hardware used to perform logic is shared by all cells in the column (or row). Hence, even if gates are logically independent (i.e., no *data hazard* applies) they must still be performed sequentially[2]. For example, a full-adder can be implemented with 9 NAND gates, taking 9 time steps, as shown in Fig.2. Hence, optimizing both the latency and energy of a PIM

[1] In contrast to analog architectures which are specialized accelerators, and typically do not maintain standard memory operation.
[2] There are PIM architectures that are exceptions to this [12], however, they require additional transistors which significantly increase complexity.

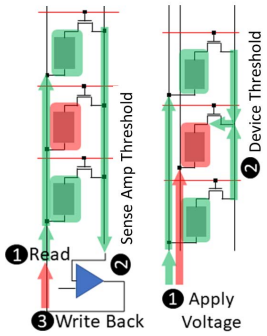


Figure 1: Column based PIM logic approaches. Inputs (output) are (is) shown in green (red). SA: Sense Amplifier.

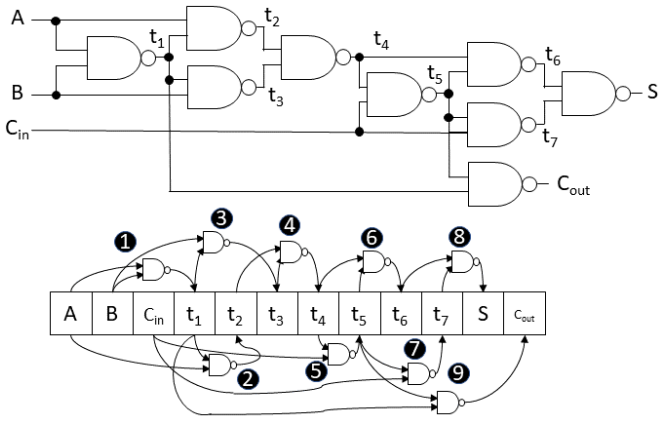


Figure 2: Full-Adder circuit and equivalent in-memory implementation.

computation (within a single row or column) is simply finding the decomposition which requires the fewest logic gates.

b-bit addition can be done with a ripple-carry adder with $b-1$ full-adds and 1 half-add. Note that, while it is slow in traditional digital circuitry, a ripple-carry adder is optimal for PIM as it uses the fewest gates (which must be performed sequentially). A DADDA multiplier [36], on the other hand, can perform b bit multiplication with b^2-2b full-adds, b half-adds, and b^2 AND gates.

While specifics vary across architectures, our discussion so far covers the most representative prior and state of the art work, as listed in Table 1. Here we list each design with the original memory technology used, but for most of these architectures, different memory technologies (MRAM, RRAM, PCM) can be exchanged for one another and the basic operating principles would remain the same.

Parallelism: The sequential nature of logic operations (gates) described in Section 2.2 leads to a high latency for any single operation. However, it is compensated for by high degrees of parallelism. PIM architectures are capable of much higher degrees of parallelism than other architectures, including GPUs. Potentially as many gates as the number of rows (or columns) within an array can be performed at the same time. This is because while a row (column)- parallel PIM architecture can only perform one operation in each row (column) at a time, the same operation can be performed in many rows

Design	Parallelism	Memory Technology
Pinatubo [21]	Column	PCM
MAGIC [20]	Row and Column	RRAM
MAGIC [22]	Column	MTJ
Felix [12]	Row and Column	RRAM
CRAM 2T [6, 43]	Row	MTJ
CRAM 1T [8, 29–31]	Column	MTJ
CRAM [7, 14, 45]	Row	SOT-MTJ

Table 1: Architectures which perform logic gates in memory and follow the operation principle considered in this paper.

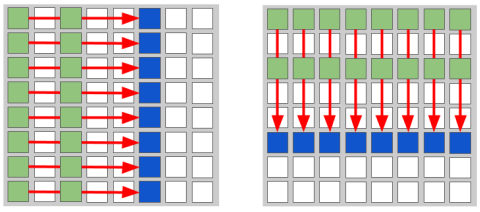


Figure 3: Parallel two-input logic gates in row- and column-parallel architectures. Inputs (output) shown in green/light (blue/dark).

(columns) simultaneously. Whether parallelism comes from the rows or the columns depends on the architecture, both kinds are shown in Fig.3. Within a single array of a row (column)-parallel architecture, gates can be performed simultaneously if:

(1) The logic operation is the same.
(2) Input(s) and outputs are in the same columns (rows).

For example, a common memory array dimension is 512×512, which would allow for 512 parallel operations. Additionally, PIM architectures allow for array level parallelism, as independent logic operations can be performed in different arrays. Hence, the limiting factor for PIM performance at scale is the number of arrays, the energy efficiency of the operations, and the overhead for any communication between arrays.

Row-parallel and column-parallel architectures are logically equivalent, except that in row (column)- parallel architectures the logic operations have the same orientation as (are perpendicular to) the read and write operations. While these differences can largely be accounted for with different data layout optimizations, row- and column-parallel architectures place different constraints on possible optimizations.

In the following, we will use the word ***lane*** to refer to the collection of cells (either in a row or a column) which can work together to perform computation. For column-parallel architectures, a lane is a single column; and for row-parallel architectures, a single row.

Data Layout for Computation: Data layout design is critical and has a significant impact on latency, energy, and endurance. PIM lanes can only compute on values contained within them, hence all data values needed in each step of computation must be properly placed in memory through standard memory writes first. Any data value to be processed which cannot fit into the lane contributes to additional communication (read and write) cost.

Typically, a single primitive operation is mapped onto a single lane. Data is aligned across different lanes to allow parallel processing of independent operations in different lanes. Each lane contains cells dedicated to input data, output data, and temporary workspace, respectively. For example, a 3-bit integer multiplication, $A \times B = C$ can be mapped to a lane as shown in Fig.4: First, space for the bits of

Figure 4: Cells within a lane of a PIM array are dedicated to inputs (A, B), outputs (C), and temporary workspace for the multiplication of two 3-bit integers.

A and B is allocated and the corresponding values are written into the lane. Commonly, one of the operands is static (e.g., the weights of a neural network) and the other is not (e.g., new inputs to a neural network layer). A number of logic gates are then performed in order to produce the product (output). These intermediate logic operations require storage for their outputs and temporary scratch bits. We call this storage the *workspace*. The minimum size for the workspace depends on the algorithm. However, when fully consumed, the workspace needs to be reset (overwritten or re-used) to enable further computation. Once computation is complete, the product C becomes ready in a dedicated set of cells, where they are available to be read out or used in further computation.

Effectively, all PIM architectures with the compute capability discussed in Section 2.2 follow this format. Large scale applications, such as convolutional or fully-connected neural networks, are decomposed into multiplications, additions, and subtractions which are performed within the lanes of the array. Different applications will only result in different data layouts and data transfers.

Application Mapping: Embarrassingly parallel operations can easily be mapped onto lanes: Each independent operation occupies a separate lane, such that all lanes can perform the same logic gate operation on different data values. For example, element-wise multiplication of N-element vectors can be mapped to N multiplications in N lanes[3], following the data layout in Fig.4 for $N = 3$ on a per lane basis. However, many applications are not so easily mapped. For example, an N-element dot-product initially requires the same N parallel multiplications. But then all products must be added together to produce the final sum. This requires read and write operations to move bits scattered across parallel lanes into the very same lane. This mapping process is the subject of prior work [21, 31], and typically involves complex optimization. In summary, any application can be mapped to a PIM architecture, but only those which exploit a high degree of parallelism will be performant [31].

3 IMPACT OF PIM ON ENDURANCE

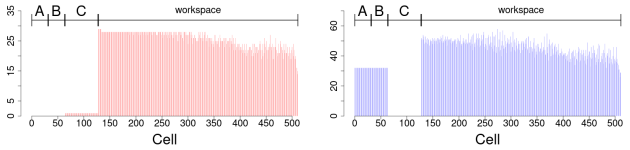
In this section we cover PIM induced performance challenges in the face of endurance limitations and revisit potential mitigation strategies.

3.1 Endurance Demand under PIM

While nonvolatile PIM architectures show a great potential for high performance and extreme energy efficiency, a major problem they face is limited endurance. Underlying memory devices simply break down after having performed a specific number of write operations. While this is a well studied problem for NVM, it has not been addressed for NVPIM.

PIM architectures impose a significantly different access pattern on memory cells than standard memory architectures. This renders a tighter design specification at the memory technology level. Specifically, PIM results in many more memory reads and writes while performing the same computation when compared to a traditional architecture featuring separate memory and logic blocks, which taxes the endurance of memory devices significantly. For

[3]Assuming there is a sufficient number of bits in each lane to complete computation. Practical array sizes (256×256, 512×512, 1024×1024) can easily accommodate the multiplication of 64-bit integer operands.



(a) Write Count (b) Read Count

Figure 5: Number of read and writes per cell in a lane is heavily imbalanced. Workspace cells are used many more times than input cells in producing a single result.

example, 32-bit integer multiplication on a standard architecture entails reading two 32-bit numbers, performing the multiplication using an ALU, and writing the 64-bit product back to memory. In total, this incurs 64 cell reads and 64 cell writes. If 1024 NVM cells are available to facilitate this computation, an average of 0.0625 reads and writes per cell applies. In a PIM architecture, using an in-memory DADDA multiplier [36] as a representative example, the same multiplication requires 9,824 in-memory gates, which incurs 9,824 cell writes and 19,616 cell reads. This produces an average of 19.16 reads/cell and 9.59 writes/cell. Hence, PIM can burn through the endurance of NVM *much* quicker.

An upper limit on the lifetime of a memory array can be quickly calculated. Let us assume that each memory cell is capable of 10^{12} writes before failure [23, 34], which is optimistic for current MTJs and well beyond what current RRAM can support. An 1024×1024 array can perform a total of $1024^2 \times 10^{12}$ writes before failure in this case. Using the multiplication example and assuming perfect load balancing, this array can perform at most

$$\frac{1024^2 \times 10^{12}}{9824} = 1.07 \times 10^{14} \tag{1}$$

32-bit multiplications before total break-down. For high performance, a PIM array typically performs many multiplications in parallel [31]. At full utilization (all 1024 lanes computing in parallel) and assuming a reasonable switching time per gate of 3ns [31, 32], it would take

$$\frac{1024^2 \times 10^{12}}{1024 \times \frac{1}{3\times10^{-9}}} = 3,072,000\ seconds = 35.56\ days \tag{2}$$

until total failure (when every cell breaks down).

Worse, under practical conditions, even a small number of failed devices can cause incorrect operation, so effective failure can occur much sooner. Using current RRAM endurance of approximately 10^8 writes, time to failure would take *just over 5 minutes*. Clearly, PIM comes with a very demanding endurance requirement for the underlying nonvolatile technologies. Endurance has been improving and will undoubtedly continue to improve to enable practical PIM applications as NV technology tailored for PIM comes to maturity. At the same time, higher level mitigation strategies such as load balancing are going to be critical in order to extend the system lifetime as much as possible.

Standard NVM operation may already cause more frequent (read or write) accesses to some rows, which may lead to an imbalance in memory cell lifetime. NVPIM exacerbates such imbalance by performing logic gates in some lanes more frequently than others. Even 32-bit integer multiplication with a data layout similar to Fig.4 can cause a large imbalance in the usage of each cell within the lane, as shown in Fig.5. Specifically, cells dedicated to workspace are used much more frequently, resulting in a significant load imbalance which can make these cells fail significantly sooner than others. Hence, load balancing becomes even more critical in maximizing the lifetime in this case.

3.2 Load Balancing under NVPIM

We will next revisit basic load balancing strategies to increase the lifetime of NVPIM arrays in the face of limited endurance. Any imbalance in cell usage, as noted in Section 3.1, can cause some memory cells to fail much more quickly than others. Load balancing is a well-known strategy to address this problem in the context of NVM applications [13, 24, 39, 42, 47]. The idea is distributing write operations as evenly as possible to all memory cells and thereby preventing premature cell failure due to excessive use. The question is to what extent classic load balancing strategies tailored for NVM can help under NVPIM. Memory cell usage under NVPIM is not just determined by write operations, but does evolve with actual computations. Hence, designing effective load balancing for NVPIM is inherently a different challenge, beyond the capabilities of prior strategies targeting memory functionality only.

Load Balancing for NVM vs. NVPIM: While the rich literature on load balancing for NVM [4, 16, 36] features a variety of (software and/or hardware) techniques, the underlying fundamental strategy never changes: redistribute write operations by modifying the virtual to physical address mapping over time to prevent imbalanced cell wear-out.[4] Here we show why this core mechanism is not directly applicable to PIM, as load balancing for PIM is fundamentally different than load balancing for memory.

Algorithm 1 shows an example code excerpt where the main computation simply corresponds to the bitwise AND of two variables x and y residing in memory. Using this example, Fig.6 illustrates what classic load balancing entails for regular NVM vs. NVPIM. Let us assume that in NVM x gets written to Row 0 and y; to Row 1. If PIM is not the case, the alignment of x and y in their respective rows does not have any direct implication for computing, hence y can be shifted in Row 1 as shown in Fig.6(a) for load balancing purposes. In this case the CPU performs the computation after both variables are read from memory. Any write redirection (in the form of a shift in this example) has no effect on the correctness of the computation. For NVPIM, on the other hand, this type of load balancing would not work, as depicted in Fig.6(b). Correct computation constrains data layout by requiring alignment of the input operands in memory. Specifically, input operands must reside in the same lanes (columns in this example). Hence, the very same "load-balanced" memory layout would lead to a computation error under PIM.

Write re-mapping works for standard memory because a memory word is the typical granularity for data access. This makes relocation relatively easy. The data access granularity of a typical PIM operation (i.e., computation), on the other hand, is the entire array. This is because in-memory operations can use all lanes (i.e., all columns and any set of rows in Fig.6(b)) simultaneously. Hence, relocating any memory word in the conventional fashion can easily corrupt the contents of a PIM array, i.e., the input data of a PIM operation. This complexity holds even if we restrict write redirection to only across rows. For example, y in Fig.6(b) can be remapped into Row 2, but within the same columns as x, potentially allowing for correct completion of the example computation. Even then, we cannot guarantee correctness, since contents of Rows 1 and 2 now become inconsistent. As a result, if Row 1 or 2 is used in PIM operations later in the program, subsets of each may have unexpected contents.

In an nutshell, any memory load-balancing strategy is based on such write redirects. PIM introduces additional constraints that must be satisfied. Importantly, the physical location of variables in memory are dependent on each other. Hence, write operations cannot be remapped without careful consideration of the entire algorithm, a complexity that is not handled by state-of-the-art load-balancing strategies. In the following, we show how basic load-balancing can be tailored to PIM architectures.

[4]Some load-balancing strategies use write cancellation [26] to prevent writes from even occurring in the first place. Unfortunately this is not directly applicable to PIM, where all computation happens in the memory.

Algorithm 1 Example code

```
x ← 5
y ← 6
z ← x&y                                  ▷ Bitwise AND
```

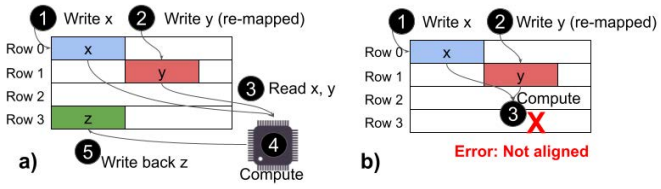


Figure 6: Load balancing for standard NV memory vs. PIM. Remapping writes a) works for standard memory as data layout in memory is decoupled from actual computation; b) does not work for PIM as computation in memory requires input operands to be physically aligned.

Software Load Balancing refers to changes made to the program in order to evenly distribute the write operations. The benefit of software approaches is that they do not require any hardware modifications to the architecture, which can cost extra energy and increase latency for every operation. Significantly, software strategies have a greater ability to re-distribute write operations by arbitrarily modifying the program. However, a key drawback of software approaches is that they require either knowledge from the programmer or compiler support. Additionally, software strategies may require periodic re-mapping (re-compilation) in order to be effective, which incurs a time and energy overhead.

(Software) Load Balancing within Lanes: Fig.5 shows that some cells within a lane are used more often than others, even in undertaking a single computation. Notably, cells holding temporary values are used much more frequently than cells that hold the inputs and the outputs. If this imbalance of usage persists for long periods of time, workspace cells will fail much sooner. Hence, it is desirable to allow cells holding inputs and outputs to also be used as workspace.

An optimized operation (e.g., multiplication or addition) uses the same number of gates with the same inputs and outputs each time. However, individual logic gates that the operation is composed of can have the inputs and the output anywhere within a lane, i.e., logic gates can be re-mapped by modifying their input operand (e.g., row) addresses within the lane (e.g., column). It is conceptually easy to implement fine-grained re-mapping in software by maintaining the logical to physical address mapping for each bit. Programs operate on logical bits and remain intact. Logical to physical mapping can change periodically, arbitrarily re-mapping logcic gate operations within lanes. This process is shown in Fig.7.

As intuition suggests, changing logical to physical address mapping randomly throughout computation can be highly effective at leveling cell wear-out within lanes. We will be referring to this strategy as *Random Shuffling*. Unfortunately, this solution has a significant drawback for row-parallel architectures. For example, an 32-bit variable may reside in consecutive bits in the first 4 bytes of a row (lane). A row parallel architecture can access this variable in

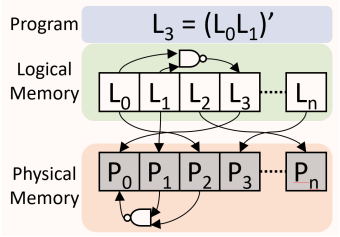


Figure 7: Logical to physical address mapping of bits via software can arbitrarily remap logic gate operations.

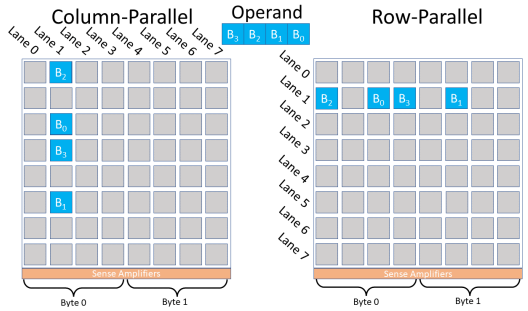


Figure 8: Re-arranging the bits of an operand within a PIM lane can disrupt memory operations. For row-parallel architectures, bits of the operand can be read in parallel, but they will be out of order and potentially in different bytes. Column-parallel architectures read bits out of the lane sequentially, and are less impacted by such re-arrangement.

a single cycle with a standard read or write operation. Re-mapping logic gate operations can cause individual bits of the variable to spread out to different bytes across the lane. Hence, many more bytes may need to be accessed in order to read or update the variable. Additionally, when the bits are read out of the array, they can be in any permutation. This requires external post-processing to re-order the bits. This is less of an issue for column-parallel architectures, as depicted in Fig.8.

One promising strategy which does not complicate memory accesses is, instead of randomizing, simply periodically shifting the logical to physical address mapping. To maintain proper (byte-addressable) read and write operations, shifts should be by an integer number of bytes, hence we call this strategy *Byte-Shifting*. Shifting has been proposed both for standard memory [47] and for crossbars for neural network acceleration [39], and *Byte-Shifting* represents a generic adaptation for PIM.

Re-mapping logical to physical addresses, whether randomized or shifted, has the advantage of no overhead during the execution of the program. However, both require periodic re-compilation in order to balance load. ByteShift is more memory access friendly as it maintains the order and coherency of bits in the memory. The different re-mapping strategies are depicted in Fig.9.

(Software) Load Balancing Between Lanes: Just as with in-lane usage, usage, hence wear, across lanes can vary, as well. However, the causes of imbalance are different. Imbalance in lanes comes from the mapping of individual logic operations, which can be re-mapped. In contrast, imbalance between lanes is more of a function of the application, which restricts re-mapping.

Imbalance between lanes typically is the case when results from many lanes need to be combined. For example, an N-element dot-product requires N parallel multiplications, which can be balanced on N lanes. But all products must be added together to calculate

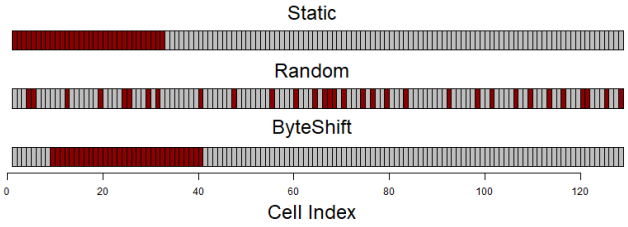


Figure 9: Representative placement of an 32-bit operand within a lane for different re-mapping strategies. *Static* excludes any load balancing.

the final sum. This necessitates a series of memory operations to bring the products into the same lanes where they represent inputs to addition. Reduction operations such as the addition in this example limit lane-level parallelism, hence some lanes get used more than others. Heavily used lanes will wear out sooner. Still, strategies for software load balancing within lanes mostly apply here. Physical bit addresses can be periodically changed over time. Complete randomization is likely to balance load most optimally. However, a byte shift may be more desirable as it keeps addresses aligned for memory accesses.

Hardware Load Balancing: Hardware based load balancing strategies for standard NVM typically rely on estimating write counts per cell and re-directing memory accesses accordingly [13]. However, the incurred complexity is not feasible at the level of a single PIM array: The main benefit of nonvolatile PIM is extreme energy efficiency. Unless exceedingly light-weight by design, hardware dedicated to balancing may easily become the bottleneck. Maintaining counters to track writes at the bit-level is unreasonable. Luckily simpler strategies do exist.

(Hardware) Load Balancing Within Lanes: We observe that the well-known practice of *register renaming* in traditional architectures can be used to perform lightweight hardware load balancing. This process is similar to the software based logical to physical address re-mapping, however it requires logical to physical address translation at execution time and it cannot arbitrarily re-map computations. In the following, we will refer to this strategy as *Hardware re-mapping*.

Hardware re-mapping requires a spare bit which can be used to swap logical addresses. For a lane with N physical bits, there are $N-1$ logical bit addresses and 1 free bit address. Re-mapping in a NVPIM array can be applied upon a write or a logic operation as follows: As an example, without loss of generality, when a write operation is performed to logical bit address A in all lanes, the hardware re-directs the write to the free physical address, overwriting its contents. It then marks the free physical address as logical address A, and assigns the previous physical address of A as the free address. The same procedure seamlessly applies if only a subset of lanes are involved, as well.

For architectures like Pinatubo [21] which perform computation at the array periphery using sense amplifiers, the initial value of the output memory cell does not matter. Re-mapping can occur entirely in-place. In other words, bit re-mapping does not require additional data transfers as all we need is to redirect the output of an operation. Both writes and logic operations can be renamed without complication. However, for architectures like CRAM [6], the initial value of the output cell affects computation and often needs to be preset before computation. For this type of architecture, an additional write operation would be required.

(Hardware) Load Balancing Between Lanes: In principle, the same re-mapping scheme can be applied between lanes, as well.

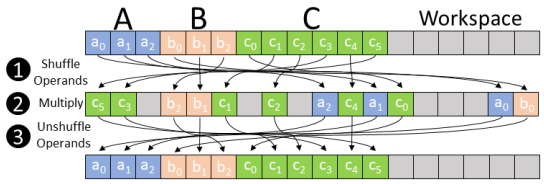


Figure 10: COPY gates can be used to directly shuffle the input operands and the output during execution. Initially, only A and B contain meaningful data and need to be shuffled. After computation, only C contains necessary data and requires un-shuffling.

However, this may not always be practical depending on the architecture. For row-parallel architectures, memory operations can access the entire lane (row) at once. This makes lane re-mapping feasible by swapping row addresses on write operations. However, for column-parallel architectures, memory operations can only access 1 bit from each lane at a time. Reading and writing an entire lane requires accessing each bit sequentially. Since a regular memory write can only overwrite 1 bit in a lane at a time, a lane cannot be renamed using a single memory write. The brute force approach of copying a logical lane (column) from one physical column to another over many sequential accesses would likely be too complex, slow and energy hungry for practical use.

Memory Access Aware Re-mapping: Load balancing strategies we introduced so far focus on PIM, but inevitably have an impact on regular memory operations (i.e., reads and writes). It is possible to re-map computations but return the result of computation to the original addresses afterwards, to prevent any change to regular memory read and write access patterns. Prior to a computation, input operands can be shuffled arbitrarily by performing COPY (or two sequential NOT[5]) gates. Then, the computation can proceed as normal, only with different column (row) addresses for each gate in a row (column)-parallel architecture. After the computation is finished, COPY or NOT gates can again be used to re-order the output bits at the expected destination. This process is depicted for a multiplication operation in Fig.10 considering a row-parallel-architecture without loss of generality.

One drawback of this approach, contrary to previously discussed strategies, is that it requires additional logic gates to implement the shuffling. The relative overhead for this shuffling process depends on the operations being performed in memory. For a precision of b bits, shuffling requires $2 \times b$ COPY gates (or $4 \times b$ NOT gates) to move the two input operands to their new locations. Note that the output and the workspace bits do not need to be physically shuffled as they do not yet contain meaningful data, however, write operations may be required to pre-set them depending on the PIM architecture. The number of gates required to un-shuffle the output depends on the output size. For multiplication, if roll-over is ignored, the output has the same number of bits as the inputs. However, in many applications allocating more bits to the output is useful. We consider this more general case here. For multiplication, the output has twice as many bits, so $2 \times b$ COPY (or $4 \times b$ NOT) gates are required to move the output back to the original location. In total, we need $4 \times b$ COPY (or $8 \times b$ NOT) gates.

This overhead for multiplication is small relative to the number of gates required for computation. A DADDA multiplier [5, 28, 36] requires $b^2 - 2b$ full-adds, b half-adds, and b^2 AND gates. Using 2-input logic gates, a full-add requires a minimum of 5 gates and a half-add requires 2 gates. Hence, a multiplication requires $6b^2 - 8b$ gates in total, and the relative overhead (in terms of the additional number of gates) for shuffling becomes $1/(\frac{3}{2}b - 2)$. For 32-bit numbers, this equates to an extra 2.17%. The relative overhead for addition, on the other hand, is much higher, due to the significantly lower complexity of the algorithm. Ripple-Carry addition (optimal for PIM) requires $b - 1$ full-adds and 1 half-add. The output is 1 bit longer than the inputs, hence the shuffling cost is $3b + 1$ additonal gates. The relative overhead in this case becomes $\frac{3b+1}{5b-3}$. For 32-bit numbers, this equates to an extra overhead of 61.78%. Table 2 captures how this overhead evolves with bit precision.

Table 2: Percentage of extra COPY gates required by memory access aware randomized shuffling during execution. Overhead corresponds directly to extra latency and energy as all gates must be performed sequentially.

Bit Precision	Multiplication (DADDA) Overhead (%)	Addition (Ripple Carry) Overhead (%)
4	25	76.47
8	10	67.57
16	4.55	63.64
32	2.17	61.78
64	1.06	60.88

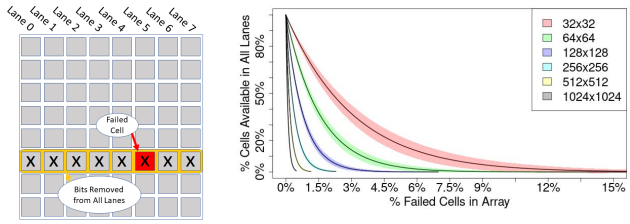
An additional drawback of this approach is that it requires more complex software support. It still requires periodic changes to physical bit addresses. However, these changes cannot be performed by simply modifying a logical to physical address mapping (as it is the case for software-based strategies). Additional logic operations must be inserted directly into the program to perform the shuffling. This requires modification to the original program, hence full-fledged re-compilation. In summary, this approach can load balance without perturbing standard memory read and write access patterns at the expense of a significant logic gate overhead and additional software support.

3.3 Using PIM Arrays with Failed Cells

Provided that nonvolatile cells are subject to failure due to limited endurance, we will next discuss to what extent NVPIM arrays can remain functional in the presence of failed cells. Applications need to exploit the inherent parallelism of NVPIM to achieve high performance. This usually translates into most, if not all, of the lanes participating in computation simultaneously. To this end, cells storing input operands have to reside at the same addresses within each lane. Hence, even a single cell failure in a single lane can deem all cells at the same address in other lanes useless, as shown in Fig.11a. In an $N \times N$ PIM array, there are N^2 cells which can fail, but only N cells in each lane. The available space therefore quickly reduces with failed cells, as captured in Fig.11b. We observe that irrespective of the array size, the number of available cells can quickly reach a point where even multiplication is not possible due to insufficient space.

A workaround solution is to divide lanes into different sets, and to only use lanes in the same set in parallel. This can extend the array lifetime, by increasing the number of usable cells at any given time. However, this comes at a quickly increasing cost in latency, as different sets must run sequentially. In summary, even a few cell failures in a PIM array can significantly disrupt operation.

[5]Some PIM architectures do not natively support COPY [29] and have to use NOT gates instead.



(a) A single cell failure removes bits from all lanes

(b) Percentage of bits in lane that can be used versus percentage of bits in array that have failed.

Figure 11: Failed cells quickly reduce the number of bits available in each lane. A single failed cell prevents the use of bits at the same address in all lanes.

4 EVALUATION SETUP

PIM arrays are used to accelerate computational kernels which can exploit the inherent parallelism in hardware. If used in an embedded device, the device can only function as long as the PIM arrays persist. If used in a server, the accelerator must be replaced once a sufficient number of PIM arrays fail. Hence, we need to analyze how long PIM arrays can function in the face of endurance limitations.

Benchmarks: Large-scale applications must be broken down into computations that can be performed within PIM arrays. PIM arrays can process data independently. As necessary, standard memory read and write operations can handle data transfers between PIM arrays. Our analysis focuses on computations that can be performed within a single array, with the assumption that many such arrays perform similar work in parallel. Without loss of generality, we use three representative case studies which cover extreme ends of potential computations: *1) Embarrassingly parallel multiplications*, *2) Neural network (NN) inference (convolution)*, and *3) Vector dot-products*. Embarrassingly parallel multiplication represents an ideal application for PIM. Each lane of the PIM array can operate independently without any intermediate communication. Dot-products, on the other hand, represent a non-ideal case. Results from all active lanes must eventually be combined into a single output, which leads to considerable data communication. NN inference is a middle ground, where independent computations are typically too large to fit into a single lane, but do not require all lanes at the same time. Computations less complex than multiplication become trivial. Computations more complex than dot-product typically require multiple arrays or are not suitable for PIM. Applications featuring lower degrees of parallelism or incurring high intra- or inter-array data communication traffic perform poorly on PIM architectures.

Embarrassingly Parallel Multiplication: The first benchmark features a simple parallel integer multiplication of 32-bit operands. A single multiplication is performed within each lane (e.g., column). There is no communication between lanes, and all lanes are utilized. Hence, there should be no imbalance between lanes. However, the multiplication algorithm (DADDA multiplier) may have imbalanced usage within each lane.

Dot-Product: A dot-product of two vectors A and B consists of an element-wise multiplication followed by a summation. If each vector has N elements, the final output can be written as:

$$C = \sum_{i=0}^{N-1} A_i \times B_i \tag{3}$$

The dot-product can be performed using the multiplication and addition operations described in Section 2.2. Fig.12 shows two examples of a dot-product of two-element vectors performed in memory. Here, A_0 is multiplied with B_0 and A_1 with B_1, after which, the

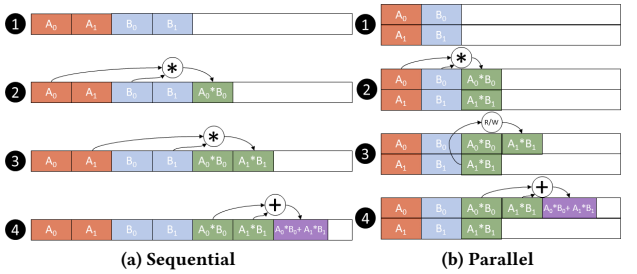


Figure 12: Vector dot-product mapped to a row-parallel PIM array. a) Computation on elements stored in a single row can only proceed sequentially. b) Computation on elements stored in different rows can be done in parallel, but requires data transfer to extract the final result.

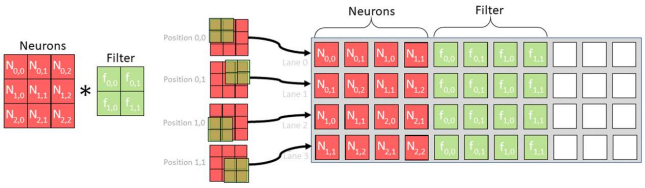


Figure 13: 2-dimensional convolution in a PIM array.

results are added. Assuming a row-parallel architecture, Fig. 12a shows a sequential computation within a single row; Fig. 12b, a parallel computation spread over two rows. In the parallel version the multiplications proceed simultaneously in separate rows. However, this requires an intermediate read and write data transfer to bring the results to the same row so that they can be added. In general, an N element dot-product having $2 \times M$ elements in each row uses $ceiling(\frac{N}{M})$ rows. With larger numbers of rows, a larger number of reads and writes would be required to move intermediate results to the same rows. However, logic operations dominate the latency, energy, and endurance. A single data transfer takes 2 sequential operations (read/write) and at most 1 read and 1 write per cell in a row. A multiplication takes over 20,000 sequential operations (logic gates) and requires roughly 40 reads and writes per cell (Section 3.1). Hence, regardless of the specific data layout chosen, the performance of dot-products within the memory is dominated by the latency and efficiency of the underlying NVPIM technology's logic operations. For our benchmark we use 1024 element vectors with 32-bit operands.

Convolution: Inference applies a *filter* to a set of input *neurons*. The filter consists of a set of weights. The number of weights is typically small relative to the number of input neurons. The filter is "slid" over the neurons. At each location the weights of the filter are element-wise multiplied with the neurons they overlap with. The results of these multiplications are then summed together and finally become subject to some non-linear transformation (such as threshold, sigmoid, or htan). The neurons and filter can be 1-, 2-, or 3-dimensional. Typically they have the same depth (z dimension), and the filter is slid over the neurons in the x and y dimensions. Fig. 13 covers a 2-dimensional example, along with the corresponding data placement in a PIM array.

A filter with K rows and L columns requires $K \times L$ multiplications. Each multiplication requires the corresponding input neuron and filter weight. Fig. 13 shows all $K \times L$ (i.e., 4×4) multiplications occurring on the same lane, hence, each lane contains 4 neurons and 4 weights. However, they may also be distributed to multiple lanes. For example, a single multiplication can be performed in each lane, in which case each lane will have a single neuron and weight.

Following the layout in Fig. 13, the 4 multiplications in each lane proceed sequentially (using the techniques described in Section 2.2). All 4 lanes can perform this computation in parallel. After multiplication, partial results all reside within the same lane. Then, the final sum can be generated using addition. For typical NNs, the sum is read out from each lane, and external circuitry performs the non-linear transformation. However, for binary NNs (BNNs) [9], a simple comparison operation (Section 2.2) can perform a logical threshold operation [31], producing the single bit output. In the case where neurons and weights for a single filter location are placed in separate lanes, the results of each multiplication cannot be directly summed (as they reside in different lanes). Read and write operations become inevitable after the multiplication to move the partial results to a single lane.

In summary, regardless of the specific data layout, convolution in a PIM array consists entirely of multiplications, additions, intermediate reads and writes, and (potentially) a non-linear operation. For our benchmark, we perform two-dimensional convolution with a 4×3 filter on a set of 16×16 neurons with 8-bit precision, using a comparison as the non-linear operation. Three multiplications are performed sequentially and the products are added into a partial sum within each lane. Then the partial sums from 4 lanes are moved to a single lane to compute the final sum and output.

Simulation and Modeling Framework: We choose a PIM array size of 1024×1024, which is a typical subarray size used for NVM [10], large enough to perform non-trivial computations, yet small enough to maintain electrical properties to feasibly enable PIM [44]. We evaluate a column-parallel architecture as a more realistic hardware implementation, requiring few modifications to existing NVM designs [21]. We also account for the overhead for pre-setting the output memory cell of logic operations, as is required by CRAM architectures [6, 8, 29, 31, 43].

Due to temporally fine-grained hardware based re-mapping, each repetition (iteration) of a benchmark can have a different write distribution. Hence, it is necessary to fully simulate a large number of iterations. We simulate each benchmark 100,000 times to obtain an estimate of the overall write distribution over time. We find write distributions for all combinations of load balancing strategies. Specifically, we experiment with two strategies in software, *random shuffling* of addresses and *byte-shifting* of addresses, respectively, which we refer to as *Ra* and *Bs*. We also include a *static* strategy, *St*, which excludes any re-mapping. Each of these strategies can be used within lanes (rows) or between lanes (columns), giving rise to a total of 9 different load balancing configurations (3 row strategies $\times$ 3 column strategies). *Hardware re-mapping*, *Hw*, is applied only within the lane (within columns and across rows) and can be turned on or off. Hence, there is a total of 18 load balancing configurations per benchmark.

Software re-mapping can be invoked every time the program is recompiled. Recompiling does not come for free, hence cannot be performed very frequently. However, more frequent re-mapping is more effective at balancing load. Accordingly, we sweep the re-mapping frequency (i.e., every 10, 100, 1000, and 10000 iterations of the application) to characterize this trade-off space. Hardware re-mapping, on the other hand, does not incur any recompilation overhead. In this case we experiment with the most extreme case of re-mapping on every gate that uses all lanes. For all types of re-mapping, we assume oracular operation, as our focus is finding the upper limit of the benefits of re-mapping[6]. Otherwise, we account for architecture specific latency and energy efficiency overheads.

[6]As we are going to show in Section 5, even in their idealized form with no overhead, these techniques cannot be of much help due to fundamental physical limitations.

We use write distributions to estimate the lifetime of the PIM array by finding when the first memory cell fails. We consider this as the failure of the entire array, because at this point the array can produce incorrect results. Additionally, even a few failed cells can significantly disrupt operation (Section 3.3). The lifetime of the array hence corresponds to:

$$Lifetime = \frac{Cell\ Endurance}{max(WriteCount)} \times Application\ Latency \quad (4)$$

We assume the same endurance for each cell, which makes our analysis more pessimistic as the actual endurance is more likely to vary across cells (our approach can be thought of as using the average endurance for the expected lifetime). Specifically, we base our analysis on MTJs (ReRAM has a much worse endurance as detailed in Section 2) and assume an endurance of 10^{12} writes [23, 34]. Application latency is the time it takes to complete each benchmark. We compute this by summing the latency of all operations (read, write, and logic), assuming 3ns per operation [29, 32].

We assume that the PIM array performs each benchmark repeatedly, i.e., as soon as it computes the final results a new set of inputs is loaded and the process repeats. This is indicative of typical operation, as PIM arrays (whether used in embedded applications or high performance servers) serve as accelerators for computational kernels. For example, an embedded device which performs machine learning will likely only offload dot-products (used for matrix-vector multiplication) or convolution operations to the PIM array. Hence, the PIM array will likely have many repetitions of the same computation.

The code for the benchmarks and the simulator to emulate the PIM array are available at [1]. The benchmarks use a variety of logical operations, such as multiplications, additions, and comparisons. These operations are mapped to sequences of logic gates that operate on a set of logical bits (virtual memory), using rules developed in prior work [31, 43]. For each gate in the program, 1 new bit of logical memory is allocated for the output. Logical bits are freed once they are no longer needed. During simulation, logical bits are mapped to physical bits consistent with the previously described load-balancing strategies (statically, randomly, or periodically shifted). The simulation is instruction-level accurate, and each write to each memory cell is counted. To see effects of load-balancing over time, we simulate for 100,000 iterations.

5 EVALUATION

We start by inspecting the write distributions within the PIM array. The more uniform the write distribution, the better. Even distributions make better use of all cells, increasing the expected time to failure. We use heatmaps to visualize write density as shown in Fig.14 for embarrassingly-parallel multiplication; Fig.15, for convolution; and Fig.16, for dot-product, respectively. Results are labeled by *within lane (row) mapping strategy* $\times$ *between lanes (column) mapping strategy* along with a *+Hw* if hardware re-mapping applies. As detailed in Section 4, the three options for *row mapping strategy* and *column mapping strategy* are: Static *St*, Random shuffling *Ra*, and Byte-shifting *Bs*.

For multiplication (Fig.14), the inputs are only written once, where workspace cells are used many times. Hence, there is a large imbalance across rows when the row mapping is static (i.e., no within lane balancing strategy is applied). On the other hand, as this benchmark uses all columns for computation, there is no imbalance between columns. Row mapping strategies Ra and Bs significantly balance the writes over rows. Adding hardware re-mapping (Hw) on top produces a nearly even write distribution.

The convolution benchmark performs 3×4 convolution, with each neuron-filter product mapped onto four columns. One of

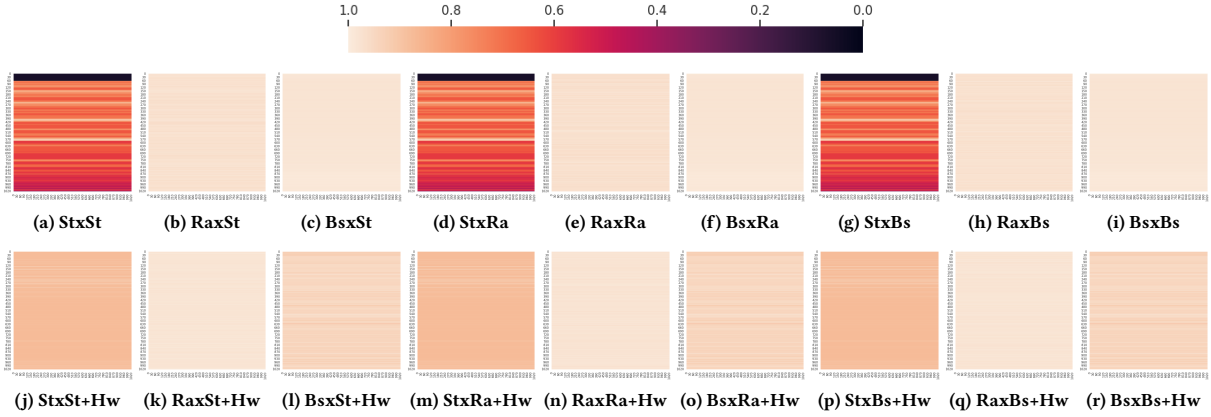


Figure 14: Multiplication write distribution (1: maximum utilization) with re-compilation every 100 iterations.

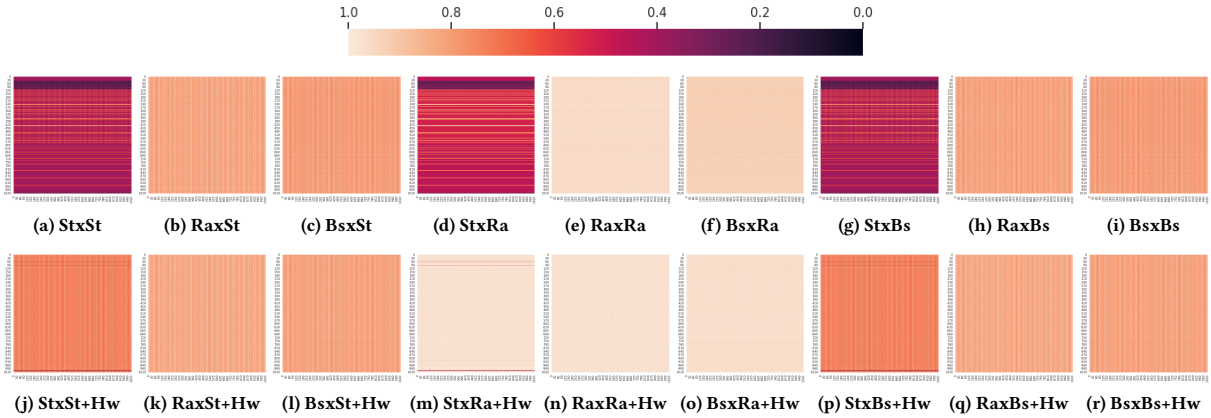


Figure 15: Convolution write distribution (1: maximum utilization) with re-compilation every 100 iterations.

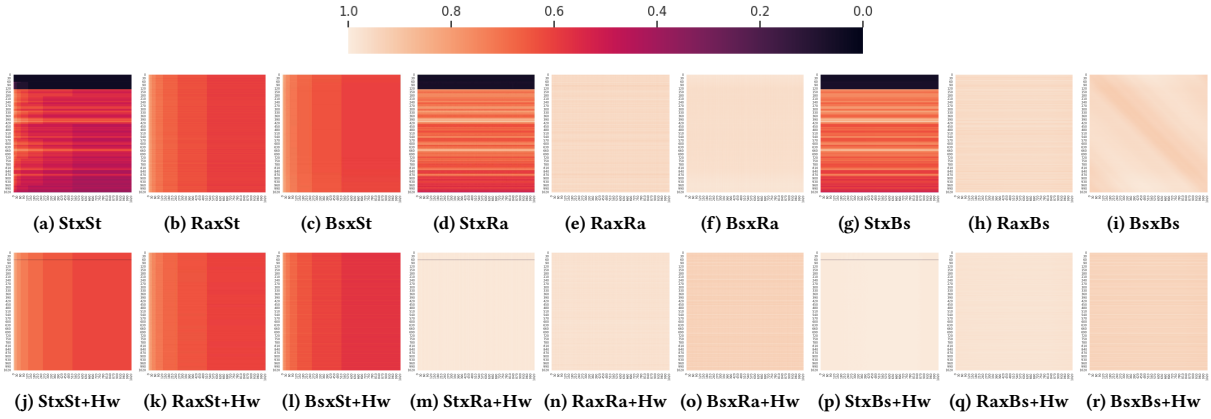
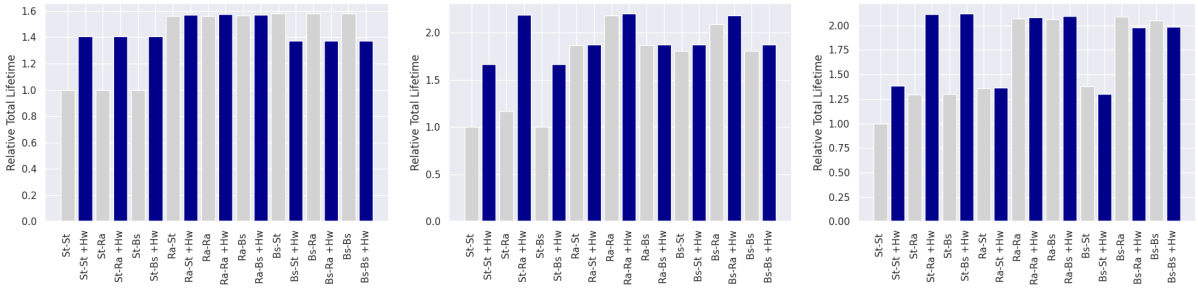


Figure 16: Dot-product write distribution (1: maximum utilization) with re-compilation every 100 iterations.

the four columns is used for the final sum. Hence, convolution (Fig.15) over-utilizes one-fourth; under-utilizes three-fourths of the columns. Convolution also uses an initial parallel multiplication, which results in an imbalance across rows. Row re-mapping strategies are generally effective at balancing out the row usage. For columns, on the other hand, Bs is ineffective as highly used columns overlap when shifted by an integer number of bytes.

Dot-product (Fig.16) heavily uses columns at low addresses, as partial sums are repeatedly moved to lower addresses to perform the reduction sum. Hence, there is a significant imbalance across columns, which both Ra and Bs manage to overcome.

Considering the write distributions, we compute the lifetime of the PIM array with Equation 4. Assuming 3ns per operation [31, 32], and and endurance of 10^{12} writes [23, 34], the lifetime



(a) Multiplication **(b) Convolution** **(c) Dot-Product**

Figure 17: Lifetime improvement under different load-balancing strategies in terms of number of operations before failure. Strategies: Static (St), Random-shuffling (Ra), and Byte-shifting (Bs) in software, along with remapping in hardware (Hw).

Table 3: Lifetime improvement of a 1024×1024 PIM array performing each benchmark continuously.

Benchmark	Avg Lane Utilization	Lifetime Improvement
Multiplication	100%	1.59×
Convolution	84.78%	2.22×
Dot-Product	65.2%	2.11×

in days for a PIM array performing each benchmark non-stop is reported in Table 3. Also reported is the maximum lifetime achieved with load balancing strategies. It should be noted that imbalance impacts lifetime in multiple ways. For example, the large imbalance in dot-product causes some cells to fail sooner, which reduces lifetime. However, the imbalance also means that many columns are inactive for a large percentage of the time (dot-product exploits less parallelism than embarrassingly parallel multiplication, and therefore has fewer writes per unit time). If pre-mature failures resulting from imbalance are avoided, dot-product can result in a longer lifetime as it is less demanding on the PIM array.

Lifetime estimates under different load-balancing strategies relative to no re-mapping (i.e., St × St) are provided in Fig.17a for multiplication; in Fig.17b, for convolution; and in Fig. 17c, for dot-product, respectively. Multiplication has no imbalance between lanes (columns), so it only benefits from within-lane (row) balancing strategies. Specifically, St × Ra and St × Bs do not provide any benefit. Convolution has some imbalance between lanes (columns), and therefore benefits from balancing between lanes (columns). Notably, since convolution is write-heavy in every fourth column, byte shifting (Bs) the columns does not help (St × Bs provides no benefit): Shifting columns by an integer number of bytes re-maps write-heavy columns to other write-heavy columns. Dot-product, which has a large imbalance in both rows and columns, shows significant improvement from load-balancing in both dimensions.

For software strategies, we found that the frequency of re-compiling does not need to be high. Over 100,000 total iterations of the benchmarks, we tested re-mapping every 10000, 1000, 500, 100, 50, and 10 iterations. We found that the expected lifetime saturates at approximately every 50 iterations. Over all benchmarks and configurations that improved from 50 to 10 iterations, the improvement was on average only 1.6%. Hence, re-compiling every 50 iterations over 100,000 iterations (after every 0.05% of the total iterations) provides nearly optimal write distributions. As the PIM array is expected to perform many more than 100,000 iterations, the frequency of re-compilation can be significantly reduced. For benchmarks that were re-compiled every 0.05% of the iterations, the expected lifetime was on average 1.71×10^{11} iterations. Hence, re-compilation on average would only need to be performed approximately every 85,000,000 iterations. This infrequent re-compilation incurs relatively low overhead.

6 RELATED WORK

Prior work primarily has developed load balancing techniques for NVM, which are not directly applicable to NVPIM. In the following we will briefly cover the most representative. Prior to Start-Gap [27] large tables were typically used to track write counts and consequently remap virtual addresses to physical addresses. Start-Gap instead only uses a couple of registers to perform algebraic re-mapping, significantly reducing overhead. Methods to cancel write operations to reduce the total number of writes to memory also exist [26]. WELCOMF [24] develops a word compression algorithm to reduce the number of required writes to NVM. Wen et. al. [38] develop strategies to lower the strain on RRAM crossbars during write operations. WoLFRaM [42] randomly remaps write operations on the fly and avoids failed cells via remapping. Very few papers cover PIM, and only for limited applications. For example, ReNew [39] develops numerous strategies mitigating writes for neural network training on RRAM cross-bars.

7 CONCLUSION

Many NVPIM accelerators have been proposed in recent years [12, 17, 31, 41], promising high performance and energy efficiency. Unlocking this performance and energy efficiency potential is impossible without addressing endurance limitations, especially for high-performance systems such as [17, 21, 31]. Architectures for low-power, embedded applications such as [2, 25, 29], on the other hand, typically have lower duty-cycles (performing computations relatively infrequently) which result in longer lifetimes. Still, endurance limitations remain a critical design challenge.

Unfortunately, despite significant improvements in lifetime due to re-mapping strategies shown in Fig.17, and further highlighted in Table 3, the limitations imposed by endurance remain. Even with aggressive load balancing strategies estimated lifetime of NVPIM arrays remain dramatically lower than the expected lifetime of standard NVM, typically spanning several years [27]. This underlines the need for PIM specific optimizations at the technology (device and material) level to match the unprecedented endurance demand of NVPIM. While endurance limitations make it a challenge to build NVPIM systems with devices available today, major improvements to device endurance [18, 37] are expected in the coming years.

REFERENCES

[1] 2023. PIM Endurance Simulator. https://github.com/SalonikResch/PIMendduranceSimulator.git.

[2] Shahanur Alam, Chris Yakopcic, and Tarek M Taha. 2022. Memristor Based Federated Learning for Network Security on the Edge using Processing in Memory (PIM) Computing. In *2022 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–8.

[3] Elia Ambrosi, Alessandro Bricalli, Mario Laudato, and Daniele Ielmini. 2019. Impact of oxide and electrode materials on the switching characteristics of oxide ReRAM devices. *Faraday discussions* 213 (2019), 87–98.

[4] Zhenhua Cai, Jiayun Lin, Fang Liu, Zhiguang Chen, and Hongtao Li. 2020. NVM-Cache: Wear-Aware Load Balancing NVM-based Caching for Large-Scale Storage Systems. In *2020 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*. IEEE, 657–665.

[5] Peter R Cappello and Kenneth Steiglitz. 1983. A VLSI layout for a pipelined Dadda multiplier. *ACM Transactions on Computer Systems (TOCS)* 1, 2 (1983), 157–174.

[6] Zamshed Chowdhury, Jonathan D Harms, S Karen Khatamifard, Masoud Zabihi, Yang Lv, Andrew P Lyle, Sachin S Sapatnekar, Ulya R Karpuzcu, and Jian-Ping Wang. 2017. Efficient in-memory processing using spintronics. *IEEE Computer Architecture Letters* 17, 1 (2017), 42–46.

[7] Zamshed Chowdhury, S Karen Khatamifard, Salonik Resch, Husrev Cilasun, Zhengyang Zhao, Masoud Zabihi, Meisam Razaviyayn, Jian-Ping Wang, Sachin Sapatnekar, and Ulya R Karpuzcu. 2022. CRAM-Seq: Accelerating RNA-Seq Abundance Quantification using Computational RAM. *IEEE Transactions on Emerging Topics in Computing* (2022).

[8] Hüsrev Cılasun, Salonik Resch, Zamshed Iqbal Chowdhury, Erin Olson, Masoud Zabihi, Zhengyang Zhao, Thomas Peterson, Jian-Ping Wang, Sachin S Sapatnekar, and Ulya Karpuzcu. 2020. Crafft: High resolution fft accelerator in spintronic computational ram. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[9] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830* (2016).

[10] Xiangyu Dong, Cong Xu, Yuan Xie, and Norman P Jouppi. 2012. Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 31, 7 (2012), 994–1007.

[11] Alessandro Grossi, Elisa Vianello, Mohamed M Sabry, Marios Barlas, Laurent Grenouillet, Jean Coignus, Edith Beigne, Tony Wu, Binh Q Le, Mary K Wootters, et al. 2019. Resistive RAM endurance: Array-level characterization and correction techniques targeting deep learning applications. *IEEE Transactions on Electron Devices* 66, 3 (2019), 1281–1288.

[12] Saransh Gupta, Mohsen Imani, and Tajana Rosing. 2018. Felix: Fast and energy-efficient logic in memory. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–7.

[13] Christian Hakert, Kuan-Hsun Chen, Paul R Genssler, Georg von der Brüggen, Lars Bauer, Hussam Amrouch, Jian-Jia Chen, and Jörg Henkel. 2020. Softwear: Software-only in-memory wear-leveling for non-volatile main memory. *arXiv preprint arXiv:2004.03244* (2020).

[14] Zhezhi He, Yang Zhang, Shaahin Angizi, Boqing Gong, and Deliang Fan. 2018. Exploring a SOT-MRAM based in-memory computing for data processing. *IEEE Transactions on Multi-Scale Computing Systems* 4, 4 (2018), 676–685.

[15] John L Hennessy and David A Patterson. 2011. *Computer architecture: a quantitative approach*. Elsevier.

[16] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K Qureshi. 2017. FlashBlox: Achieving Both Performance Isolation and Uniform Lifetime for Virtualized SSDs.. In *FAST*, Vol. 17. 375–390.

[17] Mohsen Imani, Saransh Gupta, Yeseong Kim, and Tajana Rosing. 2019. Floatpim: In-memory acceleration of deep neural network training with high precision. In *Proceedings of the 46th International Symposium on Computer Architecture*. 802–815.

[18] Andrew D Kent and Daniel C Worledge. 2015. A new spin on magnetic memories. *Nature nanotechnology* 10, 3 (2015), 187–191.

[19] SangBum Kim, Geoffrey W Burr, Wanki Kim, and Sung-Wook Nam. 2019. Phase-change memory cycling endurance. *MRS Bulletin* 44, 9 (2019), 710–714.

[20] Shahar Kvatinsky, Dmitry Belousov, Slavik Liman, Guy Satat, Nimrod Wald, Eby G Friedman, Avinoam Kolodny, and Uri C Weiser. 2014. MAGIC—Memristor-aided logic. *IEEE Transactions on Circuits and Systems II: Express Briefs* 61, 11 (2014), 895–899.

[21] Shuangchen Li, Cong Xu, Qiaosha Zou, Jishen Zhao, Yu Lu, and Yuan Xie. 2016. Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In *Proceedings of the 53rd Annual Design Automation Conference*. 1–6.

[22] Jeffry Louis, Barak Hoffer, and Shahar Kvatinsky. 2019. Performing memristor-aided logic (MAGIC) using STT-MRAM. In *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. IEEE, 787–790.

[23] Sadahiko Miura, Koichi Nishioka, Hiroshi Naganuma, TV Anh Nguyen, Hiroaki Honjo, Shoji Ikeda, Toshinari Watanabe, Hirofumi Inoue, Masaaki Niwa, Takaho Tanigawa, et al. 2020. Scalability of Quad Interface p-MTJ for 1X nm STT-MRAM With 10-ns Low Power Write Operation, 10 Years Retention and Endurance> 10^{11}. *IEEE Transactions on Electron Devices* 67, 12 (2020), 5368–5373.

[24] Arijit Nath and Hemangee K Kapoor. 2020. WELCOMF: wear leveling assisted compression using frequent words in non-volatile main memories. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*. 157–162.

[25] Keni Qiu, Nicholas Jao, Mengying Zhao, Cyan Subhra Mishra, Gulsum Gudukbay, Sethu Jose, Jack Sampson, Mahmut Taylan Kandemir, and Vijaykrishnan Narayanan. 2020. ResiRCA: A resilient energy harvesting ReRAM crossbar-based accelerator for intelligent embedded processors. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 315–327.

[26] Moinuddin K Qureshi, Michele M Franceschini, and Luis A Lastras-Montano. 2010. Improving read performance of phase change memories via write cancellation and write pausing. In *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. IEEE, 1–11.

[27] Moinuddin K Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. 2009. Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling. In *Proceedings of the 42nd annual IEEE/ACM international symposium on microarchitecture*. 14–23.

[28] S Ravi, Govind Shaji Nair, Rajeev Narayan, and Harish M Kittur. 2015. Low power and efficient dadda multiplier. *Research Journal of Applied Sciences, Engineering and Technology* 9, 1 (2015), 53–57.

[29] Salonik Resch, S Karen Khatamifard, Zamshed I Chowdhury, Masoud Zabihi, Zhengyang Zhao, Husrev Cilasun, Jian-Ping Wang, Sachin S Sapatnekar, and Ulya R Karpuzcu. 2020. MOUSE: Inference in non-volatile memory for energy harvesting applications. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 400–414.

[30] Salonik Resch, S Karen Khatamifard, Zamshed I Chowdhury, Masoud Zabihi, Zhengyang Zhao, Husrev Cilasun, Jian-Ping Wang, Sachin S Sapatnekar, and Ulya R Karpuzcu. 2021. Energy Efficient and Reliable Inference in Nonvolatile Memory under Extreme Operating Conditions. *ACM Transactions on Embedded Computing Systems (TECS)* (2021).

[31] Salonik Resch, S Karen Khatamifard, Zamshed Iqbal Chowdhury, Masoud Zabihi, Zhengyang Zhao, Jian-Ping Wang, Sachin S Sapatnekar, and Ulya R Karpuzcu. 2019. Pimball: Binary neural networks in spintronic memory. *ACM Transactions on Architecture and Code Optimization (TACO)* 16, 4 (2019), 1–26.

[32] Daisuke Saida, Saori Kashiwada, Megumi Yakabe, Tadaomi Daibou, Naoki Hase, Miyoshi Fukumoto, Shinji Miwa, Yoshishige Suzuki, Hiroki Noguchi, Shinobu Fujita, et al. 2016. Sub-3 ns pulse with sub-100 μA switching of 1x–2x nm perpendicular MTJ for high-performance embedded STT-MRAM towards sub-20 nm CMOS. In *2016 IEEE Symposium on VLSI Technology*. IEEE, 1–2.

[33] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A Kozuch, Onur Mutlu, Phillip B Gibbons, and Todd C Mowry. 2017. Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 273–287.

[34] Yohei Shiokawa, Eiji Komura, Yugo Ishitani, Atsushi Tsumita, Keita Suda, Yuji Kakinuma, and Tomoyuki Sasaki. 2019. High write endurance up to 1012 cycles in a spin current-type magnetic memory array. *AIP Advances* 9, 3 (2019), 035236.

[35] Zainab Swaidan, Rouwaida Kanj, Johnny El Hajj, Edward Saad, and Fadi Kurdahi. 2019. RRAM Endurance and Retention: Challenges, Opportunities and Implications on Reliable Design. In *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. IEEE, 402–405.

[36] Whitney J Townsend, Earl E Swartzlander Jr, and Jacob A Abraham. 2003. A comparison of Dadda and Wallace multiplier delays. In *Advanced signal processing algorithms, architectures, and implementations XIII*, Vol. 5205. International Society for Optics and Photonics, 552–560.

[37] Jian-Ping Wang, Sachin S Sapatnekar, Chris H Kim, Paul Crowell, Steve Koester, Supriyo Datta, Kaushik Roy, Anand Raghunathan, X Sharon Hu, Michael Niemier, et al. 2017. A pathway to enable exponential scaling for the beyond-CMOS era. In *Proceedings of the 54th Annual Design Automation Conference 2017*. 1–6.

[38] Wen Wen, Youtao Zhang, and Jun Yang. 2018. Wear leveling for crossbar resistive memory. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[39] Wen Wen, Youtao Zhang, and Jun Yang. 2019. ReNEW: Enhancing lifetime for ReRAM crossbar based neural network accelerators. In *2019 IEEE 37th International Conference on Computer Design (ICCD)*. IEEE, 487–496.

[40] H-S Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E Goodson. 2010. Phase change memory. *Proc. IEEE* 98, 12 (2010), 2201–2227.

[41] T Patrick Xiao, Christopher H Bennett, Ben Feinberg, Sapan Agarwal, and Matthew J Marinella. 2020. Analog architectures for neural network acceleration based on non-volatile memory. *Applied Physics Reviews* 7, 3 (2020), 031301.

[42] Leonid Yavits, Lois Orosa, Suyash Mahar, João Dinis Ferreira, Mattan Erez, Ran Ginosar, and Onur Mutlu. 2020. WoLFRaM: Enhancing wear-leveling and fault tolerance in resistive memories using programmable address decoders. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*. IEEE, 187–196.

[43] Masoud Zabihi, Zamshed Iqbal Chowdhury, Zhengyang Zhao, Ulya R Karpuzcu, Jian-Ping Wang, and Sachin S Sapatnekar. 2018. In-memory processing on the spintronic CRAM: From hardware design to application mapping. *IEEE Trans. Comput.* 68, 8 (2018), 1159–1173.

[44] Masoud Zabihi, Arvind K Sharma, Meghna G Mankalale, Zamshed Iqbal Chowdhury, Zhengyang Zhao, Salonik Resch, Ulya R Karpuzcu, Jian-Ping Wang, and Sachin S Sapatnekar. 2020. Analyzing the effects of interconnect parasitics in the stt cram in-memory computational platform. *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits* 6, 1 (2020), 71–79.

[45] Masoud Zabihi, Zhengyang Zhao, DC Mahendra, Zamshed I Chowdhury, Salonik Resch, Thomas Peterson, Ulya R Karpuzcu, Jian-Ping Wang, and Sachin S Sapatnekar. 2019. Using spin-Hall MTJs to build an energy-efficient in-memory computation platform. In *20th International Symposium on Quality Electronic Design (ISQED)*. IEEE, 52–57.

[46] Meiran Zhao, Huaqiang Wu, Bin Gao, Xiaoyu Sun, Yuyi Liu, Peng Yao, Yue Xi, Xinyi Li, Qingtian Zhang, Kanwen Wang, et al. 2018. Characterizing endurance degradation of incremental switching in analog RRAM for neuromorphic systems. In *2018 IEEE International Electron Devices Meeting (IEDM)*. IEEE, 20–2.

[47] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. 2009. A durable and energy efficient main memory using phase change memory technology. *ACM SIGARCH computer architecture news* 37, 3 (2009), 14–23.