Invited Paper: Fault Tolerant In-Memory Computing based on Emerging Technologies for Ultra-Low Precision Edge AI Accelerators

Akul Malhotra
malhot23@purdue.edu
Purdue University
West Lafayette, Indiana, USA

Sumeet Kumar Gupta
guptask@purdue.edu
Purdue University
West Lafayette, Indiana, USA

ABSTRACT

Edge Artificial Intelligence (AI) demands ultra-low power data processing on highly resource-constrained platforms, mandating a departure from conventional computing architectures. In this context, in-memory computing (IMC) coupled with ultra-low precision (ULP) neural architectures have gained traction. Furthermore, non-volatile memories such as Resistive RAMs (ReRAMs), ferroelectric-transistors (FeFETs) and others have shown an immense promise to further enhance the efficiency of deep neural network (DNN) accelerators by enabling compact and low-leakage solutions. However, the design of ULP IMC platforms must counter the impairment of inference accuracy due to manufacturing defects such as stuck-at faults (SAFs), especially those based on relatively immature emerging technologies. To that end, we present two training-free and IMC-compatible fault tolerant techniques that utilize the unique properties of ULP (binary and ternary) DNNs to mitigate the impact of SAFs. For binary neural networks (BNNs), we present BNN-Flip, a weight transformation technique that inverts rows and columns of the weight matrices to convert harmful unmasked faults to innocuous masked faults, while preserving the correctness of matrix-vector multiplication operation. Our experiments show that BNN-Flip recovers the inference accuracy of binary-precision edge devices by up to 10.55% with an energy overhead of < 3%. For ternary neural networks (TNNs),we propose TFix, a technique that exploits the natural redundancy of ternary memory arrays and high weight sparsity of TNNs to enhance fault tolerance. Our experiments show that TFix reprograms a majority of the faulty weights to their correct values, regaining the inference accuracy by up to 11.07%, with an energy overhead of < 6%.

KEYWORDS

Binary neural networks, edge AI, in-memory computing, stuck-at faults, ternary neural networks.

1 INTRODUCTION

Deep neural networks (DNNs) have been immensely successful for a wide range of tasks, including recognition and sensory processing and generative tasks [5]. To support the increasing complexity of the applications, the size of DNN models has been continually increasing, resulting in exorbitant energy and storage requirements for their deployment on hardware platforms [21]. Managing the enormous hardware demands of DNN accelerators is particularly critical for edge computing systems, which are severely resource-constrained.

One approach to reduce the resource requirements of DNNs is to reduce the bit precision of their weights and activations through quantization. Lowering the bit precision decreases storage, computation and communication energy, as well as the inference latency of DNNs without losing much accuracy. For highly energy-constrained platforms, aggressive quantization strategies involving ultra-low precisions (ULP) have also been proposed, where the bit precision of the DNN parameters is reduced to binary or ternary precision [8] [3]. Binary quantization offers the largest reduction in DNN resource requirements, while ternary quantization trades off some of this reduction for a higher inference accuracy.

Although quantization to binary/ternary precisions is very promising to reduce the computational and storage demands, the von-Neumann architecture-based DNN accelerators still suffer from latency-and energy-expensive memory processing transactions due to separate compute and memory units. A popular hardware-centric way of alleviating this issue is to use in-memory computing (IMC) [28]. IMC-based AI accelerators perform a majority of the DNN computations (specifically, vector-matrix multiplications or VMMs) within the memory array storing the weights. Thus, IMC minimizes the data movement between the memory and the compute unit, leading to significant energy savings and speed-up.

Since IMC and ULP quantization are complementary strategies, various works have explored combining the two for the design of IMC-based ULP edge artificial intelligence (AI) accelerators. Multiple IMC-based binary neural network (IMC-BNN) and IMC-based ternary neural network (IMC-TNN) accelerators have been proposed, demonstrating significant energy, latency, and area benefits [9, 22, 23, 27].

ICCAD '24, October 27–31, 2024, New York, NY, USA

ACM ISBN 979-8-4007-1077-3/24/10.
https://doi.org/10.1145/3676536.3697134

To achieve further enhancements in the energy efficiency and performance, the use of emerging non-volatile memories (NVM) like resistive random access memories (ReRAMs) and ferroelectric field effect transistors (FeFETs) has been explored for the IMC-BNN/TNN designs. While the state-of-the-art memories such as static random access memories (SRAMs) offer technological maturity and high programming efficiency, they suffer from leakage issues and large memory footprint. NVMs, on the other hand, counter these problems by virtue of their non-volatility (which leads to zero stand-by leakage) and high integration density [10, 20]. These are particularly appealing to enhance the storage capacity of edge AI platforms to enable deployment of larger DNN models.

Although IMC-BNNs and IMC-TNNs are suitable options for resource constrained edge AI accelerators, their deployment must consider the impact of manufacturing defects on DNN performance. As an example, stuck-at faults (SAFs) are irreversible hard faults that corrupt some of the memory bits in an array. Since the memory in IMC-BNN/TNNs stores the weights, SAFs cause the weight values to be incorrectly programmed into the memory, leading to erroneous computations and potentially, accuracy degradation. This issue is further aggravated for IMC-BNN/TNNs designed with emerging memories, since their fabrication processes are relatively immature, resulting in a higher probability of SAFs.

While one can pursue the trivial approach of utilizing only those chips with SAFs within a certain limit, the resultant reduction in yield and increase in cost would limit the applicability of a technology, which may be otherwise promising for DNN acceleration. Hence, to counter this, various design solutions have been proposed that aim to mitigate the impact of the SAFs on the DNN accuracy. However, most of these solutions are not compatible with IMC [25], require additional DNN training [6], and/or have a large resource overhead [13]. Also, some of these solutions are designed for platforms with full precision parameters and may have reduced efficacy for ULP DNN acclerators.

In this paper, we present an overview of two of our techniques, TFix [15] and BNN-Flip [16], designed to enhance the tolerance of ULP IMC-based DNNs to SAFs. Both techniques are IMC compatible and do not require any additional training. TFix is based on leveraging the inherent redundancy present in IMC-TNN memory cells along with the high weight sparsity of TNNs to enhance their inference accuracy in the presence of faults [15]. BNN-Flip is a weight transformation technique based on modifying the IMC-BNN weight matrices in a way that minimizes the impact of SAFs on the inference accuracy, while preserving the computational correctness of IMC-based VMM [16]. The key aspects of fault-tolerant IMC for ULP DNNs discussed in this paper are summarized below:

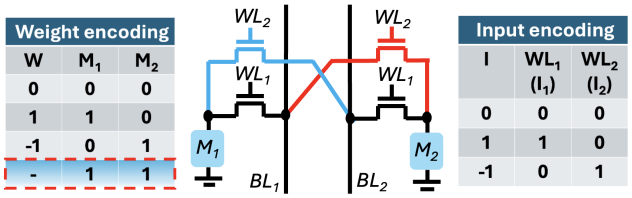


Figure 1: IMC-TNN memory cell with its input and weight encoding. The boxed state in the weight encoding is an unused state.

- We present IMC-compatible, training-free and low overhead fault-tolerant techniques viz. TFix and BNN-Flip for TNNs and BNNs, respectively. We discuss how these techniques utilize the unique properties of the TNNs/BNNs to achieve fault tolerance.
- We evaluate TFix on ResNet-18 TNNs and BNN-Flip on ResNet-18 BNNs, showing that these techniques significantly improve accuracy in the presence of SAFs.
- We examine the hardware overhead of these techniques, showing that TFix and BNN-Flip achieve SAF tolerance with negligible energy/latency/area overheads.

2 BACKGROUND AND RELATED WORKS

2.1 IMC-based ULP DNN accelerators

By combining the benefits of IMC and ULP quantization, IMC-based ULP DNN accelerators offer immense promise for edge AI. Binary and ternary quantization are two popular ULP options. In IMC-BNNs, binary quantization restricts the weights and activations of the DNN to two values: -1 and +1. In comparison, ternary quantization uses three values: -1,0 and +1, for the weights and activations. Due to the signed binary nature of the weights and activations in IMC-BNNs, their scalar multiplication is equivalent to the XNOR operation. Similarly, the scalar multiplication of ternary weights and activations can be thought of as a gated-XNOR operation; i.e. the operation is XNOR if both the weight *and* activation are non-zero; else, the output is 0.

Implementing XNOR functionality for scalar products in IMC-BNNs often involves custom memory cell design. Previous research has explored using custom 12T-SRAM [27] for IMC-BNNs. While these designs facilitate XNOR-based IMC, they come at the cost of lower cell density and higher energy usage. An alternative method, particularly for emerging memories, involves storing each weight across two memory elements [22]. However, this approach nearly doubles the area requirements, as it uses 2 bits per signed weight.

Recently, works like NAND-Net [11] have shown that by applying linear transformations to the activations and weights, IMC-BNNs can be enabled using standard memories, such as 8T-SRAM, 1T-1RRAM and 1FeFET. The linear transformations map the '-1' value of the weight/activation to '0' (high resistance state or HRS of the memory) and the

'1' value to '1' (low resistance state or LRS). This leads to the scalar product computation mapping to the AND (NAND) operation (instead of XOR). Following the in-memory AND operation and accumulation of the scalar products, some inexpensive post-processing yields the correct VMM result. This method offers lower area, energy consumption, and latency compared to approaches that utilize custom designs. Although BNN-Flip can be applied to a variety of IMC-BNN memory cells, in this paper, we focus on IMC-BNNs implemented in the NAND-Net style due to the above-mentioned benefits.

IMC-TNN memory cells also require custom design. Fig. 1 shows a common memory cell topology used for IMC-TNNs [23]. Since a single ternary weight can have three possible states, two binary memory elements are required to store it. In this paper, we restrict our discussion to memory cell design with binary memory elements, due to their relative maturity compared to multi-bit memory elements. To keep the discussion technology-agnostic, we have used M_1 and M_2 in Fig. 1 to depict the binary memory elements in the memory cells, which could be based on SRAMs or NVMs. $M_1/M_2 = 0$ (1) signifies the high (low) resistance state or HRS (LRS) of the memory cell. The weight and the activation encoding can also be seen in Fig. 1. Note, the access transistors controlled by WL_1 connect M_1 to BL_1 and M_2 to BL_2. On the other hand, the access transistors controlled by WL_2 connect M_1 to BL_2 and M_2 to BL_1, thus, cross-coupling the memory elements to the opposite bitlines. This enables scalar multiplication in the signed ternary regime, as explained subsequently. Since two binary elements can store up to four states, whereas storing a ternary weight needs only three states, we see that $M_1M_2 = 11$ state is unused. In other words, IMC-TNN memory cells have one *natural* redundant state present.

In-memory scalar multiplication $I * W$ in IMC-TNNs is performed as follows: First, the bitlines BL_1 and BL_2 are precharged to the bitline voltage V_{BL}. Then WL_1 and WL_2 are asserted based on the I value for that row (using the encoding in Fig. 1). I_1 (I_2) represents the logic value corresponding to the voltage on WL_1 (WL_2), with '1' representing a voltage of V_{WL} and '0' representing a voltage of 0. If there is a low resistance path from BL_1/BL_2 to ground, that bitline discharges by Δ. A Δ discharge on BL_1 (BL_2) corresponds to a scalar multiplication output of $+1(-1)$. Mathematically, this can be expressed as:

$$Out = \frac{Discharge(BL_1) - Discharge(BL_2)}{\Delta} \quad (1)$$

Out represents the scalar product. To calculate the dot product $\sum I * W$, multiple rows of the memory array are activated simultaneously and the discharges from the memory cells accumulate on the bitlines. Analog to digital converters (ADCs) are connected to the bitlines to obtain the digital values corresponding to their voltages.

BL_1/BL_2 discharges when an access transistor that is connected to it is ON *and* the memory element connected to that access transistor is in LRS (stores 1). Thus,

$$Discharge(BL_1) = \Delta * (I_1 * M_1 + I_2 * M_2) \quad (2)$$

$$Discharge(BL_2) = \Delta * (I_2 * M_1 + I_1 * M_2) \quad (3)$$

and,

$$Out = \frac{Discharge(BL_1) - Discharge(BL_2)}{\Delta} = (I_1 - I_2) * (M_1 - M_2) \quad (4)$$

For example, if $W = 1$ and $I = -1$, $M_1M_2 = 10$ and $I_1I_2 = 01$, thus $Out = -1$. From the memory cell perspective, we can see that BL_2 has a low resistance path through M_1, and therefore discharges through that path by Δ to produce $Out = -1$.

To understand the behaviour of the redundant state $M_1M_2 = 11$, we can substitute it into Eq. 4. We see that $M_1M_2 = 11$ always leads to $Out = 0$. Thus, the state $M_1M_2 = 11$ is another way to represent the weight value of 0.

Finally, let us look at how weights are distributed in BNNs and TNNs. In BNNs, the weight distribution has been observed to be even, with 50% 1's and -1's [4]. However, in TNNs, the percentage of zero weights is observed to be higher than the percentage of +1/-1 weights [17]. The pronounced sparsity arises because the weights in DNNs usually follow a normal distribution centered around zero. Several previous works have highlighted the sparse nature of TNNs [17][31].

2.2 Stuck-at faults (SAFs) in DNN accelerators

SAFs in DNN accelerators pose a critical computational robustness concern. SAFs occur when memory elements get irreversibly fixed at either '0'/HRS (SA0) or '1'/LRS (SA1). In DNN accelerators, they corrupt DNN weights, leading to incorrect computations and reduced DNN inference accuracy. SAFs can be unmasked or masked, depending on the type of SAF and the intended data value. Unmasked faults occur when the stored value and the fault state differ (e.g., a SA1 fault in a location storing '0'). In contrast, masked faults occur when the stored value matches the fault state. It is easy to see that the unmasked faults can potentially lead to errors and accuracy degradation, while masked faults are innocuous.

Multiple techniques have been proposed in recent times for alleviating the impact of SAFs on IMC-based DNN accelerators. From the algorithmic side, fault-aware training/fine-tuning has shown to be effective in increasing the accuracy of SAF-afflicted accelerators to near fault-free values [6]. However, training-based solutions are not only computationally expensive but also require access to training data, which may not always be feasible. From the hardware perspective, SAF

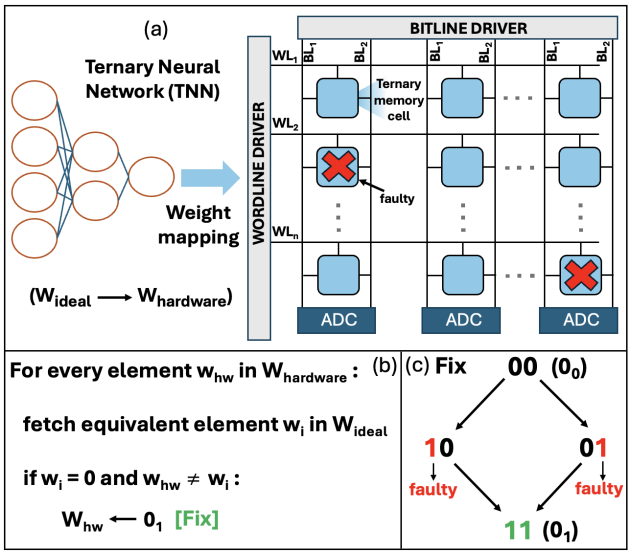


Figure 2: Overview of TFix: (a) weight mapping step, where the ideal TNN weights are mapped to the memory. (b) diagnosis step, where the SAFs are diagnosed and the hardware weights are compared to their ideal counterparts (c) fix step, wherein, when an ideal zero weight (0_0), is erroneously stored as as -1 or $+1$ due to a SAF, it is fixed by reprogramming the memory cell to 0_1.

tolerance enhancement techniques broadly fall in two categories: adding redundancy and weight remapping. The work in [13] designs a custom 2T-2ReRAM memory cell for IMC-BNNs, such that the unused states present in the memory cell can be used for increasing the fault tolerance. However, such an approach compared to the NAND-Net design requires custom memory cell design with two memory elements storing a binary weight, and therefore, incurs area and energy overheads. The work in [29] maps the least significance bits (LSBs) to the faulty locations in the memory and thus protects the most significant bits (MSBs) from being corrupted. However, this method suffers from the hardware overhead associated with tracking the position changes. Furthermore, this method is not applicable to BNNs/TNNs, where there is no MSB or LSB. In this paper, we discuss fault tolerant techniques that cater specifically to BNNs and TNNs and do not require fault-aware training. The discussed techniques exploit the unique properties of BNNs/TNNs to mitigate the impact of SAFs on inference accuracy.

3 FAULT TOLERANCE ENHANCEMENT TECHNIQUES

In this section, we present TFix and BNN-Flip, two fault tolerance enhancement techniques designed for IMC-TNNs and IMC-BNNs, respectively. TFix uses the natural redundant state and high weight sparsity in IMC-TNNs (see Section 2) to increase the inference accuracy in the presence of SAFs. BNN-Flip transforms the BNN weight matrices before they are mapped on the accelerator memory in a way that converts error-inducing unmasked faults to innocuous masked faults.

3.1 TFix

TFix exploits the fact that there are two unique ways to store the weight value '0' in IMC-TNN memory cells, as discussed in Section 2. In this paper, we will refer to the two '0' states as 0_0 and 0_1. 0_0 corresponds to when $M_1M_2 = 00$ and 0_1, which is the *unused* state in a ternary cell (as proposed in [23]), corresponds to $M_1M_2 = 11$. TFix consists of three stages: weight mapping, diagnosis and fix, as shown in Fig 2. In the weight mapping stage, the trained TNN weights, which we refer to as W_{ideal}, are programmed into the IMC-TNN memory arrays. These memory arrays may have certain memory elements affected by SAFs, causing incorrect weight values to be stored in the IMC-TNN. We refer to the hardware-mapped weights as $W_{hardware}$. The next stage is diagnosis, in which we use standard fault diagnosis techniques to determine the location and nature of the SAFs in the IMC-TNN memory [7]. Using the fault diagnosis information, we can decipher which SAFs in the memory are unmasked, causing errors in weight storage. Thus, we determine all the TNN weights w_{hw} in $W_{hardware}$ that are different from their equivalent weights w_i in W_{ideal}. Now, there are three scenarios in the event that $w_{hw} \neq w_i$:

(1) When both memory elements M_1 and M_2 are faulty: If both M_1 and M_2 in a single memory cell are affected by SAFs, the value of $w_{hardware}$ cannot be corrected by TFix. However, having both memory elements faulty in a single memory cell is fairly improbable, considering a random distribution of SAFs in the IMC-TNN memory.
(2) When w_i is -1 (Intended $M_1M_2 = 10$) or 1 (Intended $M_1M_2 = 01$): A single SAF would lead w_{hw} to have the value 0_0 ($M_1M_2 = 00$) or 0_1 ($M_1M_2 = 11$). TFix cannot correct this weight; hence, $w_{hardware}$ will have the incorrect weight value.
(3) When w_i is 0 (Intended $M_1M_2 = 00$): A single SAF would cause the w_{hw} value to be either -1 ($M_1M_2 = 01$) or 1 ($M_1M_2 = 10$), as shown in Fig 2(c). Since the 0_1 state is available for storing 0, the stored weight can be corrected by changing the value of the fault-free memory element. For example, if M_1 has a SA1 fault, 0_0 will be incorrectly programmed as 1 ($M_1M_2 = 10$). To correct this, M_2, which is fault-free, can be reprogrammed to 1, changing the stored weight value to 0_1 and eliminating the impact of the SAF on IMC.

Thus, TFix can cancel out the impact of SAFs on zero-valued TNN weights. The efficacy of TFix is further boosted by the fact that TNNs have substantial weight sparsity (discussed in Section 2). Therefore, TFix is able to neutralize the impact of SAFs on majority of the weights (storing 0).

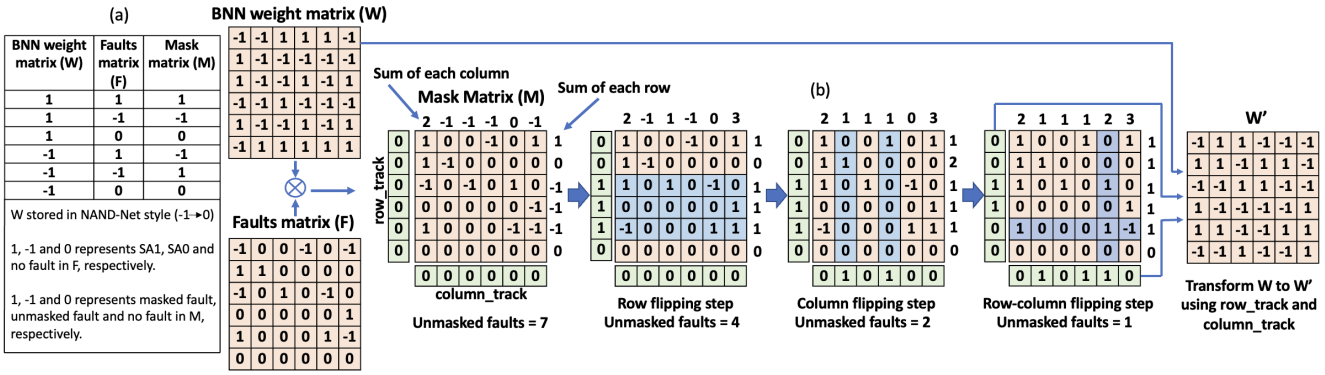


Figure 3: (a) Value of a mask matrix (M) element for corresponding BNN weight matrix (W) and fault matrix (F) elements. (b) The working of the BNN-Flip algorithm. Using row and column flips, BNN-Flip converts W to W', minimizing the number of unmasked faults for each IMC-BNN memory array.

Since TFix utilizes the natural redundant state in IMC-TNN memory cells, it incurs no area overhead. While TFix requires memory array fault diagnosis and a few additional write operations, these are one-time costs that are amortized over many inference cycles. Also, since TFix converts faulty weights from 0_0 to 0_1, it decreases the bitwise sparsity of the memory array, leading to higher bitline charging and discharging, on an average. Thus, TFix results in a mild increase in energy consumption (more details in Section 4).

3.2 BNN-Flip

BNN-Flip is based on transformation of the weight matrix in such a way that the impact of SAFs on IMC-BNNs is mitigated. This technique relies on two types of linear transformations: row flip and column flip, which form the basis of the overall matrix transformation. BNN-Flip is applied to the BNN weights before they are programmed into the IMC-BNN hardware. Recall that BNN weights and activations have two possible values: -1 and 1. However, when they are mapped on to NAND-Net based IMC-BNNs, the -1 (1) values are mapped as 0 (1). Thus, the row/column flips are applied to the original weight matrix with weights being either -1 or 1; while the SAFs (SA0 and SA1) are considered are in the context of logic '0' (HRS) and logic '1' (LRS) of the memory cell.

A row flip is defined as multiplying a row of the BNN weights and the corresponding input by '-1'. Since $I * W = (-I) * (-W)$, a row flip has no impact on the VMM of the BNN weights and activations. A column flip is defined as multiplying a column of the BNN weights by '-1'. To obtain the correct VMM value, the column output of a 'flipped' column must also be multiplied by '-1'. This is because $\sum I * W = -1 * \sum I * (-W)$. Note that a row/column in this case refers to a row/column of an IMC-based memory array in which BNN weights are to be mapped.

To preserve the accuracy of the VMM computations while modifying the BNN weights via row and column flips, it is essential to keep track of which rows and columns have been flipped. To achieve this, we introduce two tracking vectors: *row_track* and *column_track*. For an *nxm* memory array, *row_track* and *column_track* are bit vectors of size n and m, respectively. When a flip is applied to the i^{th} row (or column) of the array, the corresponding i^{th} entry in *row_track* (or *column_track*) is toggled to reflect the change. The *row_track* and *column_track* vectors are initialized to zero. If a row or column is flipped from its original state, its corresponding entry in *row_track* or *column_track* is set to 1. If that particular row or column is flipped the second time, the *row_track* or *column_track* entry is reset to 0. The understanding here is that an even number of consecutive flips on the same row or column cancel themselves out. However, if these flips are not consecutive (e.g. if two row-flips on the same row are separated by a column flip), then this cancellation is partial, which the *row_track* and the *column_track* together are able to keep track of.

BNN-Flip calculates the optimal row and column flip strategy to transform the BNN weights with the aim of minimizing the error-causing unmasked faults by transforming them to innocuous masked faults. BNN-Flip is applied to multiple weight sub-matrices, with each sub-matrix corresponding to the IMC-enabled memory array on which it is mapped. Fig. 3(b) depicts the BNN-Flip algorithm applied on a single weight matrix W with an example. First, the faults matrix F associated with the memory array is determined using conventional fault diagnosis techniques [7]. F contains information about the nature (SA1 or SA0) and location of faults in the array. Next, the mask matrix M, which contains information about where the masked and unmasked faults are located, is calculated using W and F. Fig. 3(a) shows the values of an element of M for various combinations of W and F elements. For example, a SA0 fault on a location that stores the weight value of '-1' will have M value of 1, denoting masked fault (recall '-1' weight is stored as '0' in a NAND-Net IMC-BNN design). However, a SA1 fault on

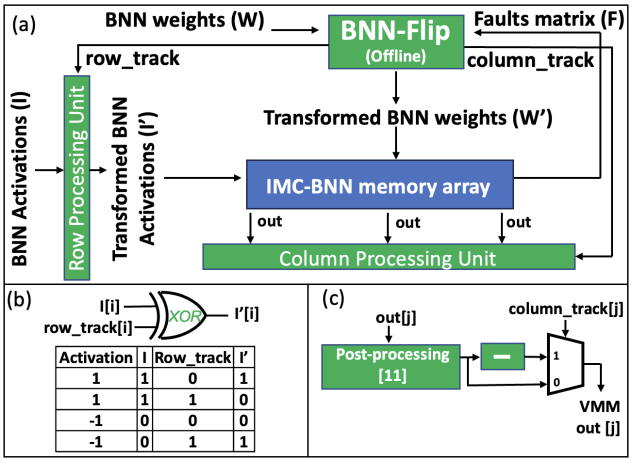


Figure 4: (a) An overview of an IMC-BNN memory array using BNN-Flip. Once the optimal values of *row_track*, *column_track* and W' are determined, they are mapped on to the IMC-BNN. W' is stored in the memory array, while the calculated *row_track* and *column_track* vectors are stored in near-memory registers for the necessary pre- and post-processing. (b) Flipping activation I for a single row in the row processing unit and (c) Flipping the VMM output for a single column in the column processing unit.

the same location will cause an unmasked fault, leading to $M = -1$. All locations that are free of SAFs have a value of 0 in M. From Fig. 3(a), we observe that M can be calculated by simply performing the element-wise multiplication of W and F (Hadamard product).

Once M is calculated, the next step is to minimize the number of unmasked faults ($M = -1$) in M. This is achieved by performing row and column flips on M in an iterative manner, until the number of unmasked faults cannot be reduced further. Every row/column flip is tracked by the *row_track*/*column_track* vectors, which are then used to convert W to W'. Note that although *row_track*/*column_track* vectors are eventually applied on W to convert it to W', they are calculated by minimizing the unmasked faults in M. This is possible because F is fixed (SAFs are irreversible) post the diagnosis step. And since M is the element-wise product of F and W , any row/column flip in M is equivalent to a row/column flip in W. This observation enables us to avoid recomputing M from W and F after every row/column flip.

Finding the optimal *row_track*/*column_track* vectors is akin to tackling a well-known NP-hard 'Shortest Vector in a Lattice,' problem [2]. Hence, we employ a *heuristic* algorithm to obtain these vectors, a strategy also adopted by other studies encountering this NP-hard problem [30].

The algorithm begins by initializing all entries in the *row_track* and *column_track* vectors to zero. We then calculate the sum of the elements in each row of matrix M. If a row's sum is less than zero, it indicates that the number of unmasked faults ($M = -1$) exceeds the number of masked faults ($M = 1$). In this case, flipping the row would reduce the unmasked faults, as this operation flips the weight values, converting unmasked faults to masked ones and vice versa. Therefore, we flip all rows with a sum less than zero and update the corresponding *row_track* values. This process is illustrated in the row flipping step in Fig. 3 (b), where the highlighted rows have been flipped.

We then perform a similar operation on the columns, calculating the sum of elements in each column, flipping columns where the sum is less than zero, and updating the corresponding *column_track* values (see Fig. 3 (b)). These two steps are repeated iteratively until no row or column has a sum of elements less than zero.

Next, we examine all possible row-column pairs, checking if they satisfy the following condition:

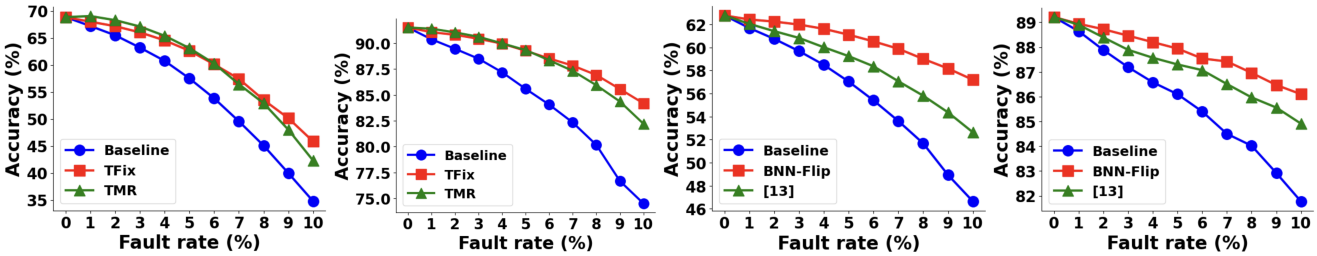
$$\sum_{j=1}^{m} M[i][j] + \sum_{i=1}^{n} M[i][j] - 2M[i][j] < 0 \qquad (5)$$

In this equation, for an *nxm* memory array, the first two terms represent the sums of the i^{th} row and j^{th} column, respectively, while the third term is twice the value of the common element at the intersection of that row and column. The rationale behind equation 5 is that when both the row and column are flipped, the common element remains unchanged. Therefore, if the sum of the elements in the row and column, excluding the common element, is less than zero, flipping the row-column pair would reduce the number of unmasked faults. For all row-column pairs meeting this condition, we simultaneously flip the corresponding rows and columns along with toggling their *row_track* and *column_track* values. This step is shown in Fig. 3(b), where the highlighted row-column pairs are flipped together.

These three steps are iteratively repeated until no further flips can reduce the number of unmasked faults.

Finally, the computed *row_track* and *column_track* vectors are used to derive W' from W. This is achieved by flipping (i.e. multiplying by '-1') each row and column of W for which the corresponding *row_track*/*column_track* entry is '1'. This step is shown in Fig. 3(b) as the final step.

W', *row_track* and *column_track* are then mapped onto the IMC-BNN. Note that the original BNN weight matrix W does not need to be stored in the memory. Some additional hardware is required to maintain the correctness of the in-memory computations when using W'. To address this need, a row and column processing unit is integrated into the memory array. Fig. 4(a) shows the IMC-BNN array augmented with the row and column processing units. The row processing unit for a single row is simply an XOR gate, which is shown in Fig. 4(b). It flips the activation value I for that row, if the corresponding *row_track* value is '1' (recall that the '-1' activations/weights are mapped to '0' in NAND-Net). I remains unchanged when the *row_track* value for the row is 0. Similarly, the column processing unit, shown in Fig.4(c)),



(a) TFix - CIFAR-100 **(b)** TFix - CIFAR-10 **(c)** BNN-Flip - CIFAR-100 **(d)** BNN-Flip - CIFAR-10

Figure 5: Inference accuracy versus fault rate for ResNet-18 IMC-TNNs and IMC-BNNs. Both TFix and BNN-Flip provide substantial improvement in inference accuracy in the presence of SAFs over the baseline.

adjusts the VMM output for a column by multiplying the post-processed output by '-1' if the *column_track* value for that column is '1'. This is achieved by computing the 2's complement of the post-processed VMM output, (effectively achieved by adding '1' to the 1's complement of the output). As discussed in Section 2, NAND-Net requires some post-processing to transform the column output from the [0,1] domain to the [-1,1] domain. Thus, the post-processing block depicted in Fig. 4(c) is part of the peripheral circuitry required by the NAND-Net design [11] and does not contribute to the overhead of BNN-Flip. We will further explore the hardware overheads of BNN-Flip quantitatively in Section 4. It is important to note here that the BNN-Flip algorithm, which determines the optimal *row_track* and *column_track* vectors, is executed offline before the weights are programmed into the IMC-BNN memory. Thus, it does not add to the inference overhead.

4 RESULTS

In this section, we quantitatively analyze the improvements in inference accuracy in the presence of SAFs achieved by TFix for IMC-TNNs and BNN-Flip for IMC-BNNs. We also discuss the hardware overhead of these techniques.

4.1 Accuracy Enhancement

For our accuracy analysis, we consider ResNet-18 based BNNs and TNNs, trained on the CIFAR-100 and CIFAR-10 datasets. To maintain competitive accuracy, the first and last layers of each model are kept at full precision, following the approach used in prior studies [18]. We observe that the weight sparsity of the TNNs is 64.8% and 74.1% when trained on CIFAR-100 and CIFAR-10, respectively. The weight sparsity for the BNNs is observed to be $\sim$ 50% for both the datasets. SAFs are randomly and uniformly injected throughout the TNNs/BNNs, consistent with methods in related research [26]. We conduct 100 Monte Carlo-based fault injection experiments, and report the mean inference accuracy for each fault rate (FR - ranging from 0% to 10%).

For the IMC-TNNs, we compare accuracy for three designs: i) the baseline design with no fault tolerance applied, ii) TFix, and iii) Triple Modular Redundancy (TMR) [14]. For TMR, three identical copies of the IMC-TNNs are deployed for inference and their outputs are averaged. This approach is commonly used in DNN hardware, as averaging the outputs from three copies effectively reduces the impact of random faults in each individual copy [19]. While TMR introduces a significant area overhead of $\sim$ 200%, it remains a robust fault tolerance strategy compatible with IMC.

For the IMC-BNNs, we compare the accuracy achieved with BNN-Flip against the baseline accuracy (with no fault tolerant technique applied) and the fault tolerant design proposed in [13]. The method in [13] leverages the unused memory states to improve fault tolerance of IMC-BNNs, where each weight is represented by two memory elements (refer to Section 2.2). Although this technique is effective, it is limited to a subset of IMC-BNNs and results in reduced area efficiency compared to NAND-Net (which we have used for BNN-Flip in our study).

From Fig. 5, we observe that both TFix and BNN-Flip outperform their respective baselines in reducing the accuracy degradation due to SAFs over FR ranging from 1% - 10%. For FR of 5%, TFix improves accuracy by 5.11% on CIFAR-100 and 3.68% on CIFAR-10. BNN-Flip increases accuracy by 4.07% on CIFAR-100 and 1.84% on CIFAR-10.

We also observe that TFix provides comparable accuracy improvements to TMR. It is even able to outperform TMR at higher FRs, surpassing TMR accuracy by 3.54% and 1.97% for CIFAR-100 and CIFAR-10, respectively, for FR of 10%. Similarly, BNN-Flip outperforms the work in [13] for almost all FRs, with an accuracy improvement of up to 4.56%.

4.2 Hardware Overhead

To evaluate the hardware overhead, we design IMC-TNN and IMC-BNN memory arrays along with the peripheral circuits (including the extra circuits needed by BNN-Flip). For TFix, we design IMC-TNN memory arrays of size 256x256 using three types of memories: Ferroelectric FETs (FeFETs), resistive random access memories (ReRAMs) and 8T SRAMs. We use [20] and [10] as the experimentally calibrated models for FeFETs and ReRAMs, respectively. We use a V_{BL} and V_{WL}

of 0.7V for IMC-based inference, Δ of 100mV and utilize the PTM 7nm technology node for our simulations [1]. We use 3-bit flash ADCs for our simulations and activate multiple rows simultaneously, following the methodology used in [23]. Our evaluation focuses on the memory macro, which consists of the memory array and the ADCs.

Since TFix exploits the natural redundancy present in ternary memory cells, it incurs no area overhead. TMR, on the other hand, suffers from an area overhead of 200%. The accuracy results mentioned earlier reflect the full implementation of TMR, but even partial versions of TMR, where only critical hardware components are triplicated, still carry an area overhead ranging from 30.4% to 73.3%. TFix is, however, accompanied with an energy overhead. TFix mitigates faults by converting weights intended to be 0_0 into 0_1, which reduces the bitwise sparsity of the memory array. This reduction in sparsity leads to increased charging and discharging of the bitlines, resulting in higher energy consumption. As TFix transforms more weights from 0_0 to 0_1 with rising fault rates, we assess the energy overhead at a fault rate of 10% to evaluate the worst-case scenario. Table 1 presents the energy overhead of TFix for IMC-TNNs designed with various memory technologies. The energy overhead introduced by TFix ranges from approximately 3% to 6% compared to the baseline (where no fault mitigation technique is applied). The energy overhead is minimal since bitline charging forms only a small portion of the total array level energy consumption, which is generally dominated by the ADCs [12].

For BNN-Flip, we utilize the same design parameters as above, except for the array size (=64x64). For enabling BNN-Flip, the IMC-BNN memory array has to be augmented with two extra registers for storing the *row_track* and *column_track* vectors, 1 XOR gate per row (to flip the activations), and 2's complement circuitry and multiplexers to flip the column outputs (details in Section 3.2). These additions lead to energy, latency and area overheads, which are quantified in Table 1. We observe that BNN-Flip results in a modest increase in energy consumption, ranging from 2.5% to 2.9%, and a latency increase of 1.6% to 1.8% across the three memory technologies. These overheads are minimal and have a negligible impact on the overall energy and latency of IMC macros, where the majority of the energy consumption is attributed to ADCs [12]. We also estimate the area overheads of BNN-Flip for the memory macro using layouts based on [24]. BNN-Flip results in an area increase of 2.3% to 6.3% over the baseline for the three memory types. The bulk of this modest overhead comes from the near-memory registers used to store the row and column track vectors. This area overhead is significantly lower than that of [13], which demands approximately 100% additional area.

Table 1: Hardware overheads of BNN-Flip and TFix for different memory technologies

	BNN-Flip			TFix	
	Energy	Latency	Area	Energy	Latency/Area
SRAM	2.5%	1.6%	2.3%	5.1%	no impact
ReRAM	2.9%	1.8%	5.8%	3.9%	no impact
FeFET	2.9%	1.8%	6.7%	3.4%	no impact

5 SUMMARY

In this paper, we present two fault-tolerant design techniques, TFix and BNN-Flip, to mitigate the impact of SAFs on the inference accuracy of IMC-based ULP DNN accelerators. TFix, which is designed for IMC-TNNs, leverages the natural redundancy and high weight sparsity present in IMC-TNNs to achieve SAF fault tolerance. BNN-Flip, which is targeted for IMC-BNNs, utilizes row and column flips to convert error-inducing unmasked faults to benign masked faults, reducing the impact of SAFs on the inference accuracy. We evaluate both TFix and BNN-Flip on ResNet-18 TNN and BNN models, respectively, trained on CIFAR-10 and CIFAR-100. Our experiments show that TFix can regain the inference accuracy by up to 11.07%, with an energy overhead of < 6%; and BNN-Flip can recoup the inference accuracy by 10.55% with an energy overhead of < 3%.

It is interesting to note certain similarities and differences between TFix and BNN-Flip. Both techniques are IMC-compatible, memory technology agnostic, require a one time fault diagnosis and do not require additional training. The first key difference lies in the type of faulty weights that these techniques are fundamentally designed to fix. While TFix is targeted towards redressing zero weights (since the redundancy that TFix exploits is associated with zero weights), BNN-flip is amenable to remedying the impact of SAFs on non-zero weights (as the row/column flip, at the algorithm level, involves the change in sign, which cannot be used for a zero weight). Thus, while TFix cannot be used for NAND-Net based BNNs (which do not have redundancy), the row/column flips can potentially be utilized to target the non-zero weights in TNNs. This presents an opportunity to utilize TFix and flip techniques in TNNs for potentially enhanced fault tolerance, which we are currently exploring. The second difference between the techniques lies in their hardware requirements. TFix does not need any additional circuitry, whereas BNN-Flip incurs a mild hardware overhead.

6 ACKNOWLEDGEMENT

This work is supported by SRC/DARPA-funded JUMP 2.0 center CoCoSys (Center for the Co-Design of Cognitive Systems). The authors thank Chunguang Wang from Purdue University for his help with hardware evaluation.

REFERENCES

[1] [n. d.]. . https://asap.asu.edu/ Accessed: August 2024.

[2] Miklós Ajtai. 1998. The Shortest Vector Problem in L2 is NP-Hard for Randomized Reductions (Extended Abstract). In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*. 10–19.

[3] Hande Alemdar et al. 2017. Ternary neural networks for resource-efficient AI applications. In *2017 International Joint Conference on Neural Networks (IJCNN)*. 2547–2554. https://doi.org/10.1109/IJCNN.2017.7966166

[4] Milad Alizadeh et al. 2019. A Systematic Study of Binary Neural Networks' Optimisation. In *International Conference on Learning Representations*. https://openreview.net/forum?id=rJfUCoR5KX

[5] Tom Brown et al. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 1877–1901. https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[6] Gouranga Charan et al. 2020. Accurate Inference With Inaccurate RRAM Devices: A Joint Algorithm-Design Solution. *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits* 6, 1 (2020), 27–35. https://doi.org/10.1109/JXCDC.2020.2987605

[7] Ching-Yi Chen et al. 2015. RRAM Defect Modeling and Failure Analysis Based on March Test and a Novel Squeeze-Search Scheme. *IEEE Trans. Comput.* 64, 1 (2015), 180–190. https://doi.org/10.1109/TC.2014.12

[8] Itay Hubara et al. 2016. Binarized Neural Networks. In *Advances in Neural Information Processing Systems*, Vol. 29.

[9] Shubham Jain et al. 2020. TiM-DNN: Ternary In-Memory Accelerator for Deep Neural Networks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28, 7 (2020), 1567–1577. https://doi.org/10.1109/TVLSI.2020.2993045

[10] Zizhen Jiang et al. 2016. A Compact Model for Metal–Oxide Resistive Random Access Memory With Experiment Verification. *IEEE Transactions on Electron Devices* 63, 5 (2016), 1884–1892. https://doi.org/10.1109/TED.2016.2545412

[11] Hyeonuk Kim et al. 2019. NAND-Net: Minimizing Computational Complexity of In-Memory Processing for Binary Neural Networks. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 661–673. https://doi.org/10.1109/HPCA.2019.00017

[12] Sangyeob Kim et al. 2023. Neuro-CIM: ADC-Less Neuromorphic Computing-in-Memory Processor With Operation Gating/Stopping and Digital–Analog Networks. *IEEE Journal of Solid-State Circuits* 58, 10 (2023), 2931–2945. https://doi.org/10.1109/JSSC.2023.3273238

[13] Shamik Kundu et al. 2022. RIBoNN: Designing Robust In-Memory Binary Neural Network Accelerators. In *2022 IEEE International Test Conference (ITC)*. 504–508. https://doi.org/10.1109/ITC50671.2022.00061

[14] R. E. Lyons et al. 1962. The Use of Triple-Modular Redundancy to Improve Computer Reliability. *IBM Journal of Research and Development* 6, 2 (1962), 200–209. https://doi.org/10.1147/rd.62.0200

[15] Akul Malhotra et al. 2023. TFix: Exploiting the Natural Redundancy of Ternary Neural Networks for Fault Tolerant In-Memory Vector Matrix Multiplication. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1109/DAC56929.2023.10247835

[16] Akul Malhotra et al. 2024. BNN-Flip: Enhancing the Fault Tolerance and Security of Compute-in-Memory Enabled Binary Neural Network Accelerators. In *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 146–152. https://doi.org/10.1109/ASP-DAC58780.2024.10473947

[17] Arturo Marban, Daniel Becking, Simon Wiedemann, and Wojciech Samek. 2020. Learning Sparse & Ternary Neural Networks With Entropy-Constrained Trained Ternarization (EC2T). In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*.

[18] Mohammad Rastegari et al. 2016. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. In *ECCV*.

[19] Annachiara Ruospo et al. 2022. Selective Hardening of Critical Neurons in Deep Neural Networks. In *2022 25th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*. 136–141. https://doi.org/10.1109/DDECS54261.2022.9770168

[20] Atanu K. Saha and Sumeet K. Gupta. 2018. Modeling and Comparative Analysis of Hysteretic Ferroelectric and Anti-ferroelectric FETs. In *2018 76th Device Research Conference (DRC)*. 1–2. https://doi.org/10.1109/DRC.2018.8442136

[21] Siddharth Samsi et al. 2023. From Words to Watts: Benchmarking the Energy Costs of Large Language Model Inference. arXiv:2310.03003 [cs.CL]

[22] Xiaoyu Sun et al. 2018. XNOR-RRAM: A scalable and parallel resistive synaptic architecture for binary neural networks. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 1423–1428. https://doi.org/10.23919/DATE.2018.8342235

[23] Sandeep Krishna Thirumala et al. 2020. Ternary Compute-Enabled Memory using Ferroelectric Transistors for Accelerating Deep Neural Networks. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 31–36. https://doi.org/10.23919/DATE48585.2020.9116495

[24] Chunguang Wang et al. 2023. Design Space Exploration and Comparative Evaluation of Memory Technologies for Synaptic Crossbar Arrays: Device-Circuit Non-Idealities and System Accuracy. arXiv:2307.04261 [cs.ET] https://arxiv.org/abs/2307.04261

[25] Rachmad Vidya Wicaksana Putra et al. 2021. ReSpawn: Energy-Efficient Fault-Tolerance for Spiking Neural Networks considering Unreliable Memories. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)* (Munich, Germany). IEEE Press, 1–9. https://doi.org/10.1109/ICCAD51958.2021.9643524

[26] Lixue Xia et al. 2018. Stuck-at Fault Tolerance in RRAM Computing Systems. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 8, 1 (2018), 102–115. https://doi.org/10.1109/JETCAS.2017.2776980

[27] Shihui Yin et al. 2020. XNOR-SRAM: In-Memory Computing SRAM Macro for Binary/Ternary Deep Neural Networks. *IEEE Journal of Solid-State Circuits* 55, 6 (2020), 1733–1743. https://doi.org/10.1109/JSSC.2019.2963616

[28] Jintao Zhang et al. 2017. In-Memory Computation of a Machine-Learning Classifier in a Standard 6T SRAM Array. *IEEE Journal of Solid-State Circuits* 52, 4 (2017), 915–924. https://doi.org/10.1109/JSSC.2016.2642198

[29] Jiangwei Zhang et al. 2022. WESCO: Weight-encoded Reliability and Security Co-design for In-memory Computing Systems. In *2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 296–301. https://doi.org/10.1109/ISVLSI54635.2022.00065

[30] Lei Zhao et al. 2021. Flipping Bits to Share Crossbars in ReRAM-Based DNN Accelerator. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*. 17–24. https://doi.org/10.1109/ICCD53106.2021.00016

[31] Shien Zhu et al. 2023. FAT: An In-Memory Accelerator With Fast Addition for Ternary Weight Neural Networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 3 (2023), 781–794. https://doi.org/10.1109/TCAD.2022.3184276