Improving Self-Fault-Tolerance Capability of Memristor Crossbar Using a Weight-Sharing Approach

Dev Narayan Yadav
NIT Rourkela, India
Email: yadavd@nitrkl.ac.in

Phrangboklang Lyngton Thangkhiew
IIT Guwahati, India
Email: phrangboklang@iitg.ac.in

F Lalchhandama
JNU Delhi, India
Email: fcdama@mail.jnu.ac.in

Kamalika Datta
University of Bremen/DFKI, Germany
Email: kdatta@uni-bremen.de

Rolf Drechsler
University of Bremen/DFKI, Germany
Email: drechsler@uni-bremen.de

Indranil Sengupta
IIT Kharagpur, India
Email: isg@iitkgp.ac.in

***Abstract*—The ability of resistive memory (ReRAM) to naturally conduct vector-matrix multiplication (VMM), the primary operation carried out in neural networks, has caught the interest of researchers. The memristor crossbar is a suitable architecture to perform VMM and additionally offers benefits like in-memory computation (IMC), low power, and high density. Memristor-based neural networks are typically trained using a mechanism where weight computations are carried out on a host machine and downloaded into the crossbar. However, due to faulty memristors in the crossbar, a cell may not be able to store the exact weight values, which may lead to inference errors. In this paper, we propose a weight-sharing method to improve the self-fault-tolerance capability of memristor crossbar. In order to reduce the impact of faulty memristors, the weights are shared among different layers of memristors in a 3D crossbar. Simulation analyses show considerable improvements in the fault-tolerance capability of the crossbar.**

***Index Terms*—Fault tolerance, Memristor crossbar, Neural network, Stuck-at-faults, Weight-sharing**

I. Introduction

Neuromorphic computing has become popular in various applications and often demands low power and high performance. It is observed that vector-matrix multiplication (VMM) is the core computation carried out in a neural network, which is computationally expensive and has a direct impact on higher power consumption and latency when performed on conventional architectures.

ReRAM technologies such as memristor crossbars support in-memory computation (IMC) [1] and are able to overcome the processor-memory bottleneck issues during computation. Also, VMM operations execute much faster on ReRAM crossbars as compared to conventional systems. In the crossbar, the weights are stored in the form of resistance/conductance values in the cells. Among the various competing technologies, memristors are considered one of the most desirable candidates for this kind of application due to their non-volatile nature, IMC capability, low power consumption, dense layout, and high density [2].

The inference quality of a crossbar-based neural network depends on how accurately the weights can be stored in the cells. In such systems, training can be performed in two ways: (a) direct-downloading, and (b) chip-on-the-loop approaches [3]. In both methods, an external system (host) carries out all necessary weight calculations, which are downloaded onto the crossbar. In the first approach, training is done completely offline, whereas in the second approach, the crossbar interacts with the host system in the calculation of network error. However, in the presence of faulty memristors, the weights may not be programmed accurately, which necessitates an analysis of fault-tolerance capabilities of the crossbar. The fault-tolerance approaches available in the literature can handle up to 10% of faults and are dependent on the fault types and their positions. Also, they incur latency, area and energy overheads. However, it has been observed that a memristor crossbar can tolerate a limited number of faults [4], [5]. This paper proposes a weight-sharing approach to improve the self-fault-tolerance capability of memristive crossbars. To achieve this, the weights of the network are shared among layers of memristors. To implement this, a three-dimensional (3-D) crossbar structure is proposed, implemented as a cascade of several 2-D crossbars. The effectiveness of the approach is analyzed using different fault patterns and datasets.

The rest of the paper is organized as follows: Section II presents the background and related works. Section III discusses the proposed mapping scheme, and Section IV evaluates the same using various fault patterns and datasets. A comparative study is presented in Section V followed by concluding remarks in Section VI.

II. Preliminary and Related Works

A. Memristor

A memristor is a passive circuit element capable of remembering the total amount of charge passed through it [6]. The first TiO_2-based memristor device was fabricated by HP Lab in 2008 [7]. By applying a suitable voltage across the device, the resistance of the device changes in a non-volatile manner. In binary logic applications, the resistance of the device can be set to one of two different states, viz. *high resistance state (R_{HRS})* and *low resistance state (R_{LRS})*.

Memristors are often fabricated in the form of crossbar [1], with vertical and horizontal nanowires on two planes that run perpendicular to each other, with the devices fabricated at every junction. Many works have been reported discussing the implementation of neural networks using conventional transistors [8], [9]. It has been observed that memristor crossbars can perform VMM operations naturally and can significantly accelerate overall computations in neural network applications.

The VMM operation in a crossbar can be carried out by applying suitable voltages to the horizontal wires and measuring the currents generated along the columns. Fig. 1(a) shows a crossbar structure that can be used to perform the operation $I = V \times M$. The voltages applied along the rows can be represented as the vector $V = \{V_0, V_1, \ldots, V_{n-1}\}$ and the currents flowing along the columns as the vector $I = \{I_0, I_1, \ldots, I_{m-1}\}$. The crossbar M can be expressed as an $n \times m$ matrix, where the $(i,j)^{th}$ co-efficient denotes the conductance of the device in row R_i and column C_j.

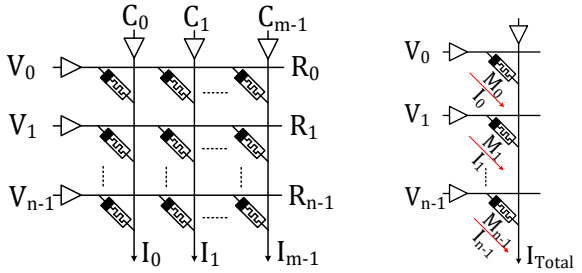


Fig. 1: VMM and dot-product operation in a crossbar

The functionality of a single neuron is similar to the dot product operation, which can be performed in a crossbar column as shown in Fig. 1(b). If the conductance of the memristor M_i is G_i, the current flowing through M_i will be $I_i = V_i * G_i$, for $0 \leq i \leq (n-1)$. The total current will be:

$$I_{total} = \sum_{i=0}^{n-1} (V_i * G_i) \tag{1}$$

Eqn. (1) represents the dot product of the two vectors $\{V_0, V_1, \ldots, V_{n-1}\}$ and $\{G_0, G_1, \ldots, G_{n-1}\}$. Various works are available in the literature that use memristive technology in neuromorphic applications [10], [11].

B. Defects in Crossbar

Circuit defects are anomalies that cause undesirable variations between the design and the implemented hardware. Memristor crossbars suffer from various defects caused by fabrication anomalies [12]. In general, the defects can be classified as: (i) *Imperfect cross-sections* – these occur due to imperfect cross-section of the devices, and can cause slow/fast write, deep faults and stuck-at-faults; (ii) *Variable oxide thickness* – this affects the normal resistive switching behaviour and can cause non-programmable deep faults; (iii) *Open defects* – these can break the crossbar structure, thereby causing access issues of a single cell, a row or multiple rows; (iv) *Short defects* – these cause rows/columns to merge, giving rise to coupling faults; (v) *Operational defects* – these occur during operation. The presence of faults can degrade the performance of VMM operations and, hence, the overall accuracy of a neuromorphic application.

C. Fault-Tolerance Approach

The fault-tolerance approaches for memristive crossbars [13]–[16] can be broadly classified into three approaches.

i) *Fault Aware Training*: The authors in [13] proposed a training approach by considering faults that can occur in the crossbar. However, this approach does not deal with actual faults in the chip, and so the actual performance may deviate from expected results.

ii) *Remapping*: The authors in [14], [15] proposed a row-interchanging method to reduce the overall error. The location of faulty memristors is first identified, and then a bipartite matching approach is used such that the overall error is minimized.

iii) *Redundant Neurons*: The authors in [16], [17] proposed a solution to use additional memristors to overcome the effect of faulty memristors.

Recent works have reported that to incorporate fault tolerance in crossbars, 30-50% energy overhead and 50-100% area overhead are incurred [13]–[17]. Most of these works are capable of tolerating the effects of only up to 8–10% faulty memristors.

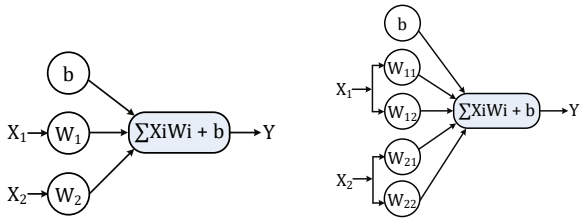
III. The Proposed weight-sharing Approach for Improved Self-Fault-Tolerance

It has been observed that neural networks implemented using memristor crossbars can tolerate a limited number of faults [4], [5]. However, performance can significantly degrade in the presence of a higher number of faults. In this section, we discuss how the self-fault-tolerance capability of the crossbar can be further improved.

A. General Analysis on weight-sharing

To illustrate the effectiveness of the weight-sharing approach, we consider a perceptron implementation of an AND gate as shown in Fig 2(a). Here X_1 and X_2 are the inputs, b is the bias, and W_1 and W_2 are weights.

The output of the network shown in Fig 2(a) is given by:



(a) (b)

Fig. 2: Perceptron model of AND gate: (a) without weight-sharing; (b) with shared weights

$$Y = \begin{cases} 0, & \text{if } \sum_{i=0}^{n-1} (W_i * X_i) + \text{b} \leq 0 \\ 1, & \text{if } \sum_{i=0}^{n-1} (W_i * X_i) + \text{b} > 0 \end{cases} \quad (2)$$

The corresponding truth table considering inputs, non-faulty and faulty weight patterns is shown in Table I. We observe that, for $b = -1$, $W_1 = 1$ and $W_2 = 1$, the network behaves as an AND gate. However, if one of the weight storage units (say, W_2) is faulty with stuck-at-0 fault, it will not perform as an AND gate.

TABLE I: Truth table of AND gate without weight-sharing.

Inputs		Ideal Network		Faulty Network	
X_1	X_2	(b=-1, W_1=1, W_2=1)		(b=-1, W_1=1, W_2=0)	
		$\sum$	Y	$\sum$	Y
0	0	-1	0	-1	0
0	1	0	0	-1	0
1	0	0	0	0	0
1	1	1	1	0	0

Now consider a modified implementation of an AND gate as shown in Fig 2(b), where each weight is shared between two cells. We assume for the time being that the weights are distributed equally among the cells. The corresponding truth table is shown in Table II. We observe that with the values $b = -1$, $W_{11} = 0.5$, $W_{12} = 0.5$, $W_{21} = 0.5$ and $W_{22} = 0.5$, the network behaves as an AND gate. In the presence of a stuck-at-0 fault on W_{22}, the behaviour of the network does not change (i.e., the fault can be tolerated). However, with multiple faults, the functionality of the network might change.

TABLE II: Truth table of AND gate with weight-sharing.

Inputs		Ideal Network		Faulty Network	
X_1	X_2	(b=-1, W_{11}=0.5, W_{12}=0.5) (W_{21}=0.5, W_{22}=0.5)		(b=-1, W_{11}=0.5, W_{12}=0.5) (W_{21}=0.5, W_{22}=0)	
		$\sum$	Y	$\sum$	Y
0	0	-1	0	-1	0
0	1	0	0	-0.5	0
1	0	0	0	0	0
1	1	1	1	0.5	1

In the next subsection, we discuss how this functionality can be implemented in a crossbar.

B. Implementation of weight-sharing in Memristor Crossbar

We now discuss the architecture and approach that can be used to implement the weight-sharing network.

1) Suitable Architecture: We propose to implement the weight-sharing network in a 3-D crossbar. A 3-D crossbar can be constructed as a cascade of several 2-D crossbars, as shown in Fig 3.

Suppose the crossbar consists of k layers $\{L_1, L_2, \ldots, L_k\}$, and the memristor at location (i, j) and layer L_x is represented as $M_{ij,x}$. A weight W will be shared among the memristors available at $M_{ij,x}$ for $x = 1, 2, \ldots, k$. However, instead of the actual weight-sharing across layers, we can modify the magnitude of voltage used during VMM operation that will have an equivalent effect of weight-sharing.

For example, let the conductance of a memristor M_i be G_i. If we apply a voltage V, then the total current injected through M_i in the respective crossbar column will be:

$$I_i = V \times G_i \quad (3)$$

Now consider memristors M_{i1} and M_{i2} with same conductance value G_i. If we reduce the voltage to $V/2$, the current injected in the column through both the memristors will be:

$$I_{i_{new}} = \frac{V}{2} \times G_i + \frac{V}{2} \times G_i = V \times G_i \quad (4)$$

In other words, I_i = $I_{i_{new}}$.

This will reduce the latency as unnecessary weight setup can be avoided, and the VMM operation can be performed in parallel across different layers of the crossbar.

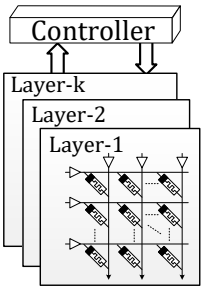


Fig. 3: 3D Crossbar using cascaded 2D crossbars

The outputs of the 3-D crossbar will be the sum of the currents in corresponding columns of all the layers. For instance, for a weight that is shared across the first columns of all layers, the output will be the sum of currents generated by the first columns of all layers:

$$I_c = \sum_{i=1}^{k} I_{ci} \quad (5)$$

2) How to Share Weights: The weights can be shared among the memristors in various ways.

i) *Equal weight-sharing:* The simplest approach is to share the weight equally in memristors $M_{ij,x}$ across all layers x. Also, we reduce the V_{read} voltage to perform VMM operation; thus, for 2 layers the read voltage will be V_{read}/2.

ii) *Weighted weight-sharing:* Here we apply unequal weight values across layers. Instead of applying V_{read} equally,

we divide it in a ratio such that the weight value is maintained. Thus, for 2 layers, the read voltage can be $V_{read} \times 0.7$ for the first layer and $V_{read} \times 0.3$ for the second layer.

iii) *Memristors with different R_{on}/R_{off}:* Here we use memristors with different R_{on}, R_{off} values in the different layers and the same V_{read} in all layers. However, this method is difficult to implement in practice.

3) Fault Proportionality Effect: It may be noted that with the increase in layers, the number of faults may increase. However, the proposed weight-sharing approach can provide improved fault tolerance even in the presence of larger number of faulty cells.

To understand the fault proportionality effect, we consider an example that uses an equal weight-sharing approach. If we use a single cell, the computed weights will have to be stored completely (100%), or else we shall lose the weight value. Thus, there is a 50% chance of not being able to store the weight correctly. However, if we store the weights in two different cells, if both are non-faulty then we will be able to store the learned weight correctly. If one of the cells is faulty, we will be able to store 50% of the learned weight, and if both are faulty, we will not have any information about the weight. Thus, the probability that we can store either all or part of the weight is 75%. Similarly, if we have three cells to store the weight, then the probability that we can store all or part of the learned weight will be 87.5%.

In general, the probability that we can store complete or part of the learned weight with n crossbar layers will be:

$$p_{non-zero} = \frac{n-1}{2^n} \tag{6}$$

and the probability that we lose the weight value will be:

$$p_{zero} = \frac{1}{2^n} \tag{7}$$

Clearly, with an increase in the number of layers, the probability of losing the complete weight will decrease, which basically increases the fault-tolerance capability. However, this will incur area and energy overheads.

IV. Simulation Evaluation

The proposed weight-sharing approach has been implemented in Python and runs on an i7-based desktop with a 2.6 GHz clock and 16 GB of RAM running Ubuntu. We have used the simple forward weight update algorithm to train the applications [18]. The Stanford memristor model [19] is used to simulate the crossbar using the Cadence Virtuoso environment. For simulation, we have used the *MNIST* [20], *Extended-MNIST* [21], *Fashion-MNIST* [22] and *CIFAR-10* [23] datasets. The direct-downloading training approach [3] is used for offline training and the generation of the weights.

All the chosen datasets are trained using fully connected neural networks, with crossbar sizes of $N_a \times N_c$, where N_a and N_c respectively denote the number of attributes and classes, respectively. For example, to train the dataset for MNIST handwritten digits, a 784×10 crossbar is used on which VMM operations are performed. To implement weight-sharing, multiple such crossbars are used in a 3-D configuration.

Various types of traditional and unique faults are possible in the crossbar. The HfO_2-based memristor crossbars [12] show 33% faulty cells, with $\approx$ 12% being stuck-at-faults (SAF). The occurrence of stuck-at-1 (SA1) is observed to be higher ($\approx 80\%$), as compared to stuck-at-0 (SA0) ($\leq 20\%$). It is also observed that the occurrence of similar types of faults in the same row is higher and occurs with higher frequency in some blocks. About 66% of total faults were found to occur in a few blocks only. In the present work, we have used SAF only.

We consider different types of crossbars as follows:

i) *Type-1 (C1):* The crossbar consists of 8:2 SA1 and SA0 faults, where the faults are limited to a few blocks, rows, and columns.

ii) *Type-2 (C2):* The crossbar consists of 8:2 SA1 and SA0 faults, where the faults occur at random positions.

iii) *Type-3 (C3):* The crossbar consists of an equal ratio of SA1 and SA0 faults, where the faults are limited to a few blocks, rows, or columns.

iv) *Type-4 (C4):* The crossbar consists of an equal ratio of SA1 and SA0 faults, where the faults occur at random positions.

A. Analysis of Self-Fault-Tolerance

To analyze the fault-tolerance capability of the proposed approach, we use the framework as proposed in [5]. Training is first performed considering an ideal crossbar; faults are then added, and the network is evaluated against the added faults. If the accuracy is found to be $\leq \pm 1$ as compared to the previous accuracy, then the fault percentage is increased and the process is repeated. For an ideal memristor crossbar with no faults, the network incurs 5.88% of inference errors for the MNIST dataset. Fig. 4 shows the impact of faults in the network if the weights are stored in a single memristor or shared among multiple memristors for the different crossbar types for the MNIST dataset [20].

Fig. 5 shows the fault-tolerance threshold (FTT) for the MNIST dataset for non-shared and shared-weight cases. FTT is defined as the threshold up to which fault percentage does not degrade by $\leq \pm 1$ as compared to the ideal crossbar accuracy. In the figure, the y-axis shows variation in accuracy with respect to the ideal case (as represented by line 0). If the accuracy of the faulty crossbar reduces by more than 1 (lies outside the green lines), then the fault % is returned as the FTT (F_{th}) for the application. In Fig. 5 average FTT of both approaches is shown.

From Fig. 4 and 5 we observe that for up to 20% faults, the accuracy can degrade by more than 40%. Furthermore, both weight-sharing approaches show significant improvements in self-fault-tolerance capability as compared to the conventional crossbar, where a single memristor is used to store weights. With each additional layer, the fault tolerance capability increases by $\approx$2%. A similar result has been observed for other datasets as well that are reported in Table III.

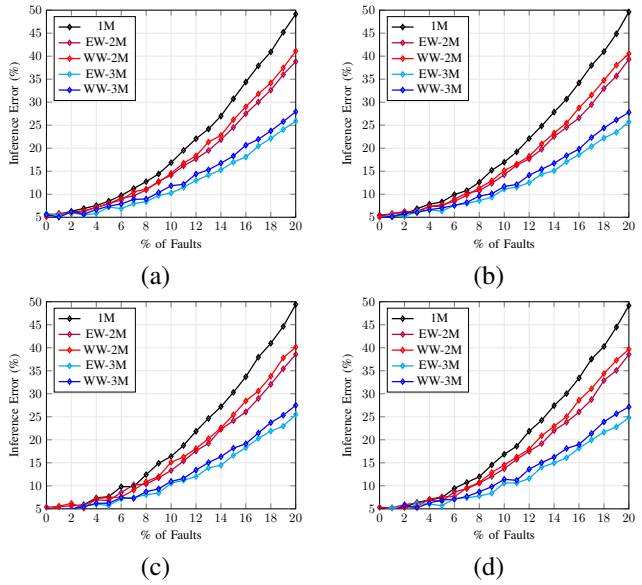


Fig. 4: Self-fault-tolerance capability of crossbar for MNIST dataset: (a) Type C1, (b) Type C2, (c) Type C3; (d) Type C4 (1M: single-memristor based network; 2M: 2-layer crossbar; 3M: 3-layer crossbar; EQ: equal weight-sharing, WW: weighted weight-sharing).

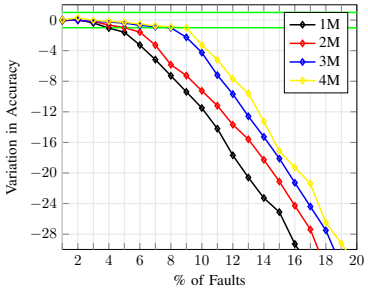


Fig. 5: Average variation in accuracy with faults for MNIST dataset for shared and non-shared network

B. Energy, Cycle and Area Overhead

In this work, training is done in offline mode on the host system, and the final weights calculated are downloaded into the crossbar. In *direct-downloading approach*, the crossbar does not interact with the host, and so there is no energy overhead during training. However, if we use the *chip-on-the-Loop* learning approach, the errors are generated by the crossbar chip itself. This increases the energy consumption as weight updates will take place for the memristors of all available layers.

Table IV shows the energy overheads caused by the proposed scheme. In direct-downloading, the weight update operation is required only once (the final weight writing); thus, we have reported the energy overhead for writing the weights in crossbar. During the inference phase, the VMM operation will be performed across all layers that also incur energy overhead; thus, the average energy consumption required for a single inference is reported. We have ignored the energy overheads due to the peripheral components like ADC, DAC, etc. in the calculation.

TABLE III: Percentage of faults that can be tolerated.

Data-set	No of Shared Memristors (Layers) used in Network						
	1	2		3		4	
		EQ	WW	EQ	WW	EQ	WW
MNIST [20]	≤3	≤5.1	≤5.2	≤7.9	≤8	≤9.6	≤ 9.5
EMNIST [21]	≤2	≤4.3	≤4.5	≤6.2	≤6.1	≤7.6	≤7.5
FMNIST [22]	≤2	≤4.2	≤4.1	≤6.3	≤6.2	≤7.2	≤7.3
CIFAR-10 [23]	≤2	≤4.1	≤4.2	≤6.1	≤6.2	≤7.3	≤7.2

TABLE IV: Energy overhead for MNIST dataset.

Approaches		Write (e)	Read (e)
Ideal		162 J	1.33 J
EQ	2M	405 J	2.45 J
	3M	608 J	3.97 J
WW	2M	425 J	2.94 J
	3M	616 J	4.05 J

The approach does not cause any cycle overhead during weight update and inference. This is because all memristors need to be set with similar weights as in the conventional network and can be done in the same number of cycles and also in parallel across the layers. The approach causes an area overhead with a multiple of 100% for each layer, as the same size crossbar is used in every additional layer. Similar energy overhead is reported for other datasets as well.

V. Discussion

In the literature, there is no prior work that improves the self-fault-tolerance capability of a crossbar. The proposed approach tries to reduce the effect of faulty memristors by sharing the weights across multiple memristors. However, various fault-tolerance approaches exist that try to reduce the impact of faulty cells in a crossbar for neural network applications. We provide a brief discussion as to how the proposed work can be beneficial as compared to other fault-tolerance approaches.

i) *Fault Aware Training*: In a retraining-based approach, training is carried out considering possible variations and faults that can occur in the crossbar. To achieve this, during the training process, a variation or fault model is validated with training samples. As a result, re-training-based approaches become highly dependent on the variation/fault model considered during training. Here, the actual performance may deviate from the expected results, as the actual faults in the crossbar are not considered. Furthermore, about 20% more training cycles are required. This approach also requires weight pruning; for up to 10% faults, 70% of the weights need to be pruned. It has been observed that the performance of retraining methods degrades for a large number of faults [13]. The proposed approach, however, deals with actual faults in the crossbar, does not require weight pruning, and can handle faults up to 7-10% without retraining.

ii) *Remapping*: In Remapping based approach row interchanging method is used to minimize the overall error [14], [15]. First, the locations of faulty memristors are identified, and then some matrix matching approach (e.g., bipartite matching) is used such that the overall error is minimized. The diagnosis of faulty memristor cells itself requires a high number of read/write operations (usually 2 writes and 2 reads for each cell) that cause high energy and cycle overheads. Furthermore, this requires complex routers to map the rows for efficient performance. On the other hand, the proposed approach does not need to identify the faulty cell locations.

iii) *Redundant Neurons*: Here, additional rows or crossbars are used as in the proposed approach [16], [17]. However, this approach necessitates the diagnosis of faulty cells and requires a complex router and controller to route the input of the defective cells to additional crossbar or crossbar rows. Furthermore, we need to initialize the weights for the additional memristors; as the locations of these additional memristors are not regular, the update can be done in a serial or semi-parallel way. On the other hand, in the proposed approach, all the weights can be initialized in parallel across the layers and do not require fault diagnosis as well.

Many works in the literature do not follow a single strategy but rather use a mixed approach. For example, the work in [15] uses retraining, remapping, and redundant rows (neurons). Most of the work can recover from the effect of up to 10% faulty memristors with 30-50% energy and cycle overheads and 50-100% area overhead with complex circuits. In contrast, the proposed approach gives similar fault-tolerance performance without using any extra circuit; rather, it uses additional crossbar layers and allows parallel execution among layers.

VI. Conclusion

Resistive RAM crossbars have drawn the attention of researchers in neuromorphic computing due to their capability of low-power VMM operation. The proposed work uses additional crossbar layers to improve the crossbars self-fault-tolerance capability. The weights are shared among layers to reduce the impact of faults. The proposed approach allows parallel operation among layers, which does not cause any latency overhead but rather achieves similar fault tolerance capability with additional crossbar layers. It is possible that by applying the state-of-the-art fault tolerance approach in the proposed weight-sharing architecture, the fault tolerance capability can be further improved. This can lead to a high fault tolerance memristor crossbar-based architecture, which can be taken up as future work.

Acknowledgement

This work is partly supported by the Indo-German project funded by Department of Science and Technology (DST), India, Federal Ministry of Education and Research (BMBF) and German Academic Exchange Service (DAAD), Germany (DST TPN No. 86669, DAAD No. 57682048) and by the German Research Foundation (DFG) within the Project PLiM (DR 287/35-1, DR 287/35-2).

References

[1] K. Akarvardar and H. S. P. Wong, "Ultralow voltage crossbar nonvolatile memory based on energy-reversible NEM switches," *IEEE Electron Device Letters*, vol. 30, no. 6, pp. 626–628, 2009.

[2] O. Kavehei, *Memristive devices and circuits for computing, memory, and neuromorphic applications.* PhD thesis, The University of Adelaide, Australia, 2012.

[3] S. M. Tam, B. Gupta, *et al.*, "Learning on an analog VLSI neural network chip," in *IEEE Intl. Conf. on Systems, Man, and Cybernetics*, pp. 701–703, November 1990.

[4] D. N. Yadav, K. Datta, and I. Sengupta, "Analyzing fault tolerance behaviour in memristor-based crossbar for neuromorphic applications," in *2020 IEEE International Test Conference India*, pp. 1–9, 2020.

[5] D. N. Yadav, P. L. Thangkhiew, *et al.*, "Famcrona: Fault analysis in memristive crossbars for neuromorphic applications," *Journal of Electronic Testing*, vol. 38, no. 2, pp. 145–163, 2022.

[6] L. O. Chua, "Memristor-the missing circuit element," *IEEE Transactions on Circuit Theory*, vol. 18, pp. 507–519, September 1971.

[7] D. B. Strukov, G. S. Snider, *et al.*, "The missing memristor found," *nature*, vol. 453, no. 7191, pp. 80–83, 2008.

[8] A. Sengupta, A. Banerjee, *et al.*, "Hybrid spintronic-cmos spiking neural network with on-chip learning: Devices, circuits, and systems," *Physical Review Applied*, vol. 6, no. 6, p. 064003, 2016.

[9] W. Shan, M. Yang, *et al.*, "14.1 a 510nw 0.41 v low-memory low-computation keyword-spotting chip using serial fft-based mfcc and binarized depthwise separable convolutional neural network in 28nm cmos," in *2020 IEEE International Solid-State Circuits Conference*, pp. 230–232, IEEE, 2020.

[10] F. M. Bayat, M. Prezioso, *et al.*, "Implementation of multilayer perceptron network with highly uniform passive memristive crossbar circuits," *Nature communications*, vol. 9, no. 1, pp. 1–7, 2018.

[11] C. Li, D. Belkin, *et al.*, "Efficient and self-adaptive in-situ learning in multilayer memristor neural networks," *Nature communications*, vol. 9, no. 1, pp. 1–8, 2018.

[12] C.-Y. Chen, H.-C. Shih, *et al.*, "Rram defect modeling and failure analysis based on march test and a novel squeeze-search scheme," *IEEE Transactions on Computers*, vol. 64, no. 1, pp. 180–190, 2015.

[13] L. Xia, M. Liu, *et al.*, "Fault-tolerant training with on-line fault detection for rram-based neural computing systems," in *54th ACM/EDAC/IEEE Design Automation Conference*, pp. 1–6, 2017.

[14] B. Zhang, N. Uysal, *et al.*, "Handling stuck-at-fault defects using matrix transformation for robust inference of dnns," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 2448–2460, 2020.

[15] Y. Xu, S. Jin, *et al.*, "Aggressive fault tolerance for memristor crossbar-based neural network accelerators by operational unit level weight mapping," *IEEE Access*, vol. 9, pp. 102828–102834, 2021.

[16] L. Xia, W. Huangfu, *et al.*, "Stuck-at fault tolerance in rram computing systems," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 8, no. 1, pp. 102–115, 2018.

[17] W. Huangfu, L. Xia, *et al.*, "Computation-oriented fault-tolerance schemes for rram computing systems," in *22nd Asia and South Pacific Design Automation Conference*, pp. 794–799, 2017.

[18] D. N. Yadav, P. L. Thangkhiew, *et al.*, "Feed-forward learning algorithm for resistive memories," *Journal of Systems Architecture*, vol. 131, p. 102730, 2022.

[19] Z. Jiang, Y. Wu, *et al.*, "A compact model for metal–oxide resistive random access memory with experiment verification," *IEEE Trans. on Electron Devices*, vol. 63, no. 5, pp. 1884–1892, 2016.

[20] Y. LeCun, "The MNIST database of handwritten digits," *http://yann.lecun.com/exdb/mnist/*, 1998.

[21] G. Cohen, S. Afshar, *et al.*, "EMNIST: Extending mnist to handwritten letters," in *International Joint Conference on Neural Networks*, pp. 2921–2926, IEEE, 2017.

[22] H. Xiao, K. Rasul, *et al.*, "Fashion-MNIST: a novel image dataset for benchmarking machine learning algorithms," *http://yann.lecun.com/exdb/mnist/*, 2017.

[23] A. Krizhevsky, V. Nair, *et al.*, "CIFAR-10," *http://www.cs.toronto.edu/kriz/cifar.html*, vol. 5, p. 4, 2009.