On Improving Fault Tolerance of Memristor Crossbar Based Neural Network Designs by Target Sparsifying

Song Jin
North China Electric Power University
Baoding, P. R. China
jinsong@ncepu.edu.cn

Songwei Pei
Beijing University of Posts and Telecommunications
Beijing, P. R. China
peisongwei@bupt.edu.cn

Yu Wang
North China Electric Power University
Baoding, P. R. China
wangyu@ncepu.edu.cn

***Abstract*— Memristor based crossbar (MBC) can execute neural network computations in an extremely energy efficient manner. However, stuck-at faults make memristors cannot represent network weight correctly, thus degrading classification accuracy of the network deployed on the MBC significantly. By carefully analyzing all the possible fault combinations in a pair of differential crossbars, we found that most of the stuck-at faults can be accommodated perfectly by mapping a zero value weight onto the memristors. Based on such observation, in this paper we propose a target sparsifying based fault tolerant scheme for the MBC which executes neural network applications. We first exploit a heuristic algorithm to map weight matrix onto the MBC, aiming at minimizing weight variations in the presence of stuck-at faults. After that, some weights mapped onto the faulty memristors which still have large variations will be purposefully forced to zero value. Network retraining is then performed to recover classification accuracy. For a 4-layer CNN designed for MNIST digit recognition, experimental results demonstrate that our scheme can achieve almost no accuracy loss when 10% of memristors in the MBC are faulty. As the faulty memristors increasing to 20%, accuracy loss is only within 3%.**

***Index Terms*—Memristor based crossbar, Neural network, Fault tolerance, Target sparsifying, Weight mapping**

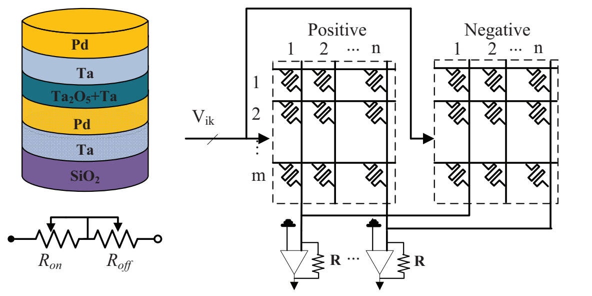
I. Introduction

In recent years, memristor based crossbar (MBC) has been attracted great concerns as a new generation of the hardware computing platform [1]. MBC can implement matrix-vector multiplication in an extremely energy efficient manner which make it a promising solution for neural network computations [2]-[3]. The existing work has demonstrated that MBC can improve the energy efficiency by over $100\times$ compared with the traditional CMOS ASIC and GPU solutions [4]-[5].

However, similar to the other nano-scale devices, memristor also suffers from severe process variations and hardware defects due to the immature fabrication processes [6]-[9]. Unlike parameter variations which can be calibrated by high precision memristor programming [10], hardware defects are unrecoverable. As shown in [7], the over-forming or the short defects result in the memristor stuck at lowest resistance state (LRS), manifesting as stuck-at 1 (SA1) fault. While the open defects lead to the memristor stuck at highest resistance state (HRS) , transforming into stuck-at 0 (SA0) fault electrically. With these defect induced faults, memristors in MBC cannot represent the mapped weights correctly, thus degrading classification accuracy when neural networks are deployed on the MBC. Although neural networks generally have some fault tolerant capabilities, the high proportion of faulty memristors in MBC still has become the main obstacle in executing neural network applications efficiently [11].

Some prior works have proposed various schemes to tolerate faults in MBC. Hardware level strategies include switching off access transistor to isolate faulty memristor or utilizing redundant columns of memristor to rich weight mapping choice [12]-[13]. However, hardware level optimization introduces large area overhead and complicates interface design. Software level optimizations often combine fault aware network retraining with weight mapping [14]-[15]. The core idea is to compensate fault induced weight variation by adjusting the weights mapped onto the faulty free memristors. Nevertheless, simply treating the weights mapped onto the faulty memristor as unadjustable restricts fault tolerant space, thus cannot obtain the optimal solution. A few of existing schemes proposed to utilize zero value weights in sparse network to tolerate SA0 faults in MBC. However, sparsifying technologies used in these schemes are designed originally for network compression, thus are fault blinding. Moreover, besides SA0 faults, there are large amount of SA1 faults which cannot be accommodated well by their sparsifying schemes.

In this paper, we propose a target sparsifying based fault tolerant scheme for the MBC executing neural network applications. To represent any signed weights in the neural network, a MBC is usually organized as a differential pair of positive and negative crossbars [1] [3]. By carefully analyzing all the possible fault combinations in the differential crossbars, we found that most of the stuck-at faults (SA0 and SA1) can be perfectly accommodated by mapping a zero value weight onto the memristors. Such observation motivates us to purposefully force some weights in the network to be zero value to tolerate faults. We first propose a heuristic algorithm to map the original weight matrix onto the MBC, aiming at minimizing weight variations in the presence of the stuck-at faults. After that, some weights mapped onto the faulty memristors which still have large variations are identified and forced to be zero value. Network retraining is then performed to recover the



(a) Memristor and its equivalent circuit (b) Memristor based crossbar which uses differential pair of memristors in positive and negative crossbars

Fig. 1. Memristor and MBC containing differential crossbars

classification accuracy.

Our scheme conducts on the software level. Above mentioned optimization processes are performed off-line. After determining the final weights by network retraining, weight matrices are then mapped onto the MBC practically. We apply the proposed scheme to a 4-layer CNN deployed on MBC which is designed for MNIST digit recognition. Experimental results demonstrate that our scheme can achieve almost no accuracy loss when 10% of memristors are faulty. When the faulty memristors increase to 20%, accuracy loss is only within 3%.

The rest of the paper is organized as follows. Section II introduces background knowledge and related work. Section III presents the proposed fault tolerance scheme. Experimental results are presented in Section IV and finally, we conclude in Section V.

II. Preliminaries and Related Work

A. *Memristor Crossbar based Neural Network Designs*

Fig.1(a) illustrates a memristor and its equivalent circuit [16]. Resistance of the memristor can be varied by adjusting the magnitude and pulse width of programming current or voltage. Fig.1(b) illustrates a MBC containing a differential pair of positive and negative crossbars. A pair of memristors at the same location in the differential crossbars are combined together to represent one signed weight. In each crossbar, there is a memristor lying on every crossing point between the horizontal word lines and the vertical bit lines. Input of the memristor crossbar is the digital or analog voltage applied on every word line while its output is the current flowing on every bit line. When the MBC executes a neural network, the connected weight matrix can be mapped onto the memristor crossbar by programming the memristors into the corresponding resistance states. When the input neuron vector are applied, calculation of the output neurons can be finished in one time step. The computation complexity of vector-matrix multiplication reduces from $O(n^2)$ to $O(1)$.

B. *Related Work*

Some researchers have proposed to model and detect the defect induced faults in MBC. Kannan et al. developed fault models for memristor and proposed efficient testing scheme to detect these faults [6]. Chen et al. proposed a march algorithm to cover defect induced faults in Resistive Random Access Memory (RRAM) [7]. Patrick et al. analyzed impacts of stuck-at faults on accuracy of a sparse coding network [17].

Some prior works have also proposed various schemes to tolerate faults in MBC. In hardware level, Manem et al. proposed switching off access transistor to isolate faulty memristors [12]. However, for the large MBC, controlling individual access transistor introduces tremendous routing and area overheads. Liu et al. proposed to utilize redundant columns of memristors as a substitution for some columns of memristors which have more failures [13]. The drawback is that utilizing redundant columns of the memristors not only introduces nontrivial area overhead, but also increases design complexity of the interface between MBC and peripheral devices like DACs, ADCs, etc.

Unlike hardware level schemes, software level optimizations commonly combine fault aware network retraining with weight mapping. For example, Chen et al. proposed a fault aware neural network training method [15]. Then, the bi-partite optimal matching algorithm was used to perform weight mapping. Besides, a few of prior works proposed to utilize zero value weights in sparse network to tolerate SA0 faults in MBC [14]. However, they use the traditional sparsifying technologies which are originally designed for network compression purpose. Moreover, besides SA0 faults, there are also many SA1 faults which cannot be accommodated well by their sparsifying schemes.

III. Target Sparsifying based Fault Tolerance

A. *Motivations*

As discussed above, a pair of differential crossbars in MBC are combined together to represent any signed weights in the neural network [3][10]. Due to the random distribution characteristic of the hardware defect, one or both memristors in the differential pair may be faulty. We carefully analyze all the possible fault combinations in the differential pair of memristors and identify the weight that can be mapped. The analyzing results are as shown in Table I.

C1 in the first row in Table I denotes the case that a pair of memristors are both faulty free, therefore the weights with any value can be mapped onto this pair of memristors. Cases C2 to C9 denote 8 fault combinations due to one or both memristors are faulty. In consequence, weights that can be mapped onto the faulty memristors are different. For example, C2 in Table I denotes that for a pair of the memristors, the one in the positive crossbar is faulty free while the one in the negative crossbar has SA0 fault. In such case, we can program the memristor in the positive crossbar from 0 to any positive values. As a result, the weight whose value is larger than or equal to 0 can be mapped on this pair of memristors. Cases C3 to C9 show the other fault combinations and the weight that can be mapped given each faulty case.

More importantly, from Table I we can see that 6 out of 8 faulty cases can be accommodated perfectly by mapping a zero value weight onto the differential pair of memristors

TABLE I
FAULT COMBINATIONS AND WEIGHTS THAT CAN BE MAPPED

Case	+Memristor	-Memristor	Weight that can be mapped
C1	normal	normal	any including 0 value weight
C2	normal	SA0	$0 \sim +W_{max}$
C3	normal	SA1	$-W_{max} \sim 0$
C4	SA0	normal	$-W_{max} \sim 0$
C5	SA1	normal	$0 \sim +W_{max}$
C6	SA0	SA0	0
C7	SA1	SA1	0
C8	SA0	SA1	$-W_{max}$
C9	SA1	SA0	$+W_{max}$

$+W_{max}/-W_{max}$: maximum positive/negative weights

if an appropriate weight is not available. As we all know, deep neural networks are usually over-fitted. In fact, a large percentage of the weights can be pruned without degrading classification accuracy of the network [18] [19]. These pruned weights are treated as zero value during the network test phase. This motivates us to perform target sparsifying which purposefully forces some weights to be zero value to tolerant faults and then perform network retraining to recover the classification accuracy. Note that our network sparsifying is for fault tolerance instead of network compression. Hence the way that choosing a weight to be pruned (i.e., set to zero value) is different from the existing network sparsifying schemes. The details of the proposed scheme are as below.

B. Overview

Fig.2 illustrates the proposed target sparsifying based fault tolerant scheme. We first perform a virtual weight mapping process to map the dense weight matrix onto the MBC, aiming at minimizing weight variations in the presence of stuck-at faults. The mapping process considers significance of the weight and fault induced variation. After the mapping, we identify the weights which still have large variations due to faulty memristors and force them to be zero value in a batched manner. Network retraining is then performed to recovery classification accuracy of the network. Above mentioned sparsifying and retraining processes conduct alternatively until the required accuracy is achieved. At last, we map the sparsified network onto the MBC practically.

C. Heuristic Algorithm for Weight Mapping

We first perform a heuristic algorithm to map the dense weight matrix onto MBC virtually, aiming at minimizing weight variations in the presence of stuck-at faults.

1) Significance of weights: Besides fault induced weight variations, the proposed mapping algorithm also considers significance of the weights. As the prior work stated [13], some connected weights have more prominent impact on classification accuracy of the network than the others. A little change in these significant weights may result in large degradation on the network accuracy. Hence, we should try to avoid to map a significant weight onto the faulty memristors if it is unfit to the faulty case (see Table I).

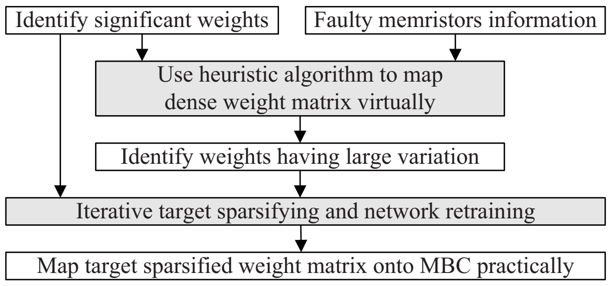

Identify significant weights
Faulty memristors information
Use heuristic algorithm to map dense weight matrix virtually
Identify weights having large variation
Iterative target sparsifying and network retraining
Map target sparsified weight matrix onto MBC practically

Fig. 2. Overview of the proposed fault tolerant scheme

We quantify significance of the weights by utilizing back propagation in the network training. In the back propagation process, errors are propagated from the network outputs to the inputs and weights are updated according to the received errors at the neurons. Based on gradient decent algorithm, weight update can be expressed as:

$$\triangle w_{ij} = \eta \frac{\partial E}{\partial w_{ij}} \tag{1}$$

where E denotes the global error. η denotes the learning rate. $\partial E/\partial w_{ij}$ is significance of w_{ij} which actually denotes the sensitivity of w_{ij} to the global error E [13].

2) Mapping algorithm: Assume that a $m \times n$ weight matrix is to be mapped onto the differential crossbars. We formulate the weight mapping as an optimizing problem:

$$min(\sum_{i}^{m}\sum_{j}^{n} \lambda_{ij}|(w_{ideal(ij)} - w_{mapped(ij)})|) \tag{2}$$

where λ_{ij} is equal to $\partial E/\partial w_{ij}$ in Eq.1, denoting significance of the weight w_{ij}. $w_{ideal(ij)}$ and $w_{mapped(ij)}$ denote the ideal weight and the practically mapped weight, respectively. Obviously, our purpose is to minimize the total amount of the mismatch of the ideal weights and the mapped weights.

We map the weight matrix onto the MBC in the row granularity, i.e., each time we map a whole row of weights onto a row of memristors in the MBC. Moreover, since it is hard or even impossible to change hardware connections in MBC after fabrication, a row of weights should be mapped onto the same rows in the two differential crossbars. To reduce the mapping complexity, we use a feature matrix to uniformly represent the two differential crossbars. Each element in the feature matrix denotes a pair of memristors in the differential crossbars that lie in the same locations. There are 6 kinds of value for the elements in the feature matrix, as illustrated in Table II. For example, feature *normal* with value 0 corresponds to case C1 in Table I. It means that a pair of memristors are both faulty-free, hence any weights can be mapped onto them. While feature *faulty-1* with value 1 corresponds to the cases of C2 and C5, thus only zero and positive weights can be mapped. The meaning of the other features in Table II can be deduced in a similar way.

The pseudo code of the proposed mapping algorithm is illustrated in Fig.3. The mapping process is divided into two

TABLE II
ILLUSTRATION OF ELEMENT VALUE IN FEATURE MATRIX

Feature	Value	Case	Weight that can be mapped
normal	0	C1	any weights
faulty-1	1	C2, C5	$0 \sim +W_{max}$
faulty-2	2	C3, C4	$-W_{max} \sim 0$
faulty-3	3	C6, C7	0 only
faulty-4	4	C9	$+W_{max}$ only
faulty-5	5	C8	$-W_{max}$ only

phases. In the first phase (lines 4-7), we make a statistic of the number of significant weights in each row in W. Then the rows in W are sorted in a descendent order. The rows with more significant weights have the priority to be mapped because weight variations in these rows have larger impact on the network accuracy. In the second phase, we start the mapping process in the row granularity. Each time we choose the current unmapped weight row in W which has the most significant weights and calculate the variations assuming that this row is mapped onto every row in M (lines 10-16). Then this row is mapped onto a row in M with the smallest variation meanwhile we flag the row in M as occupied (lines 18-19). Such mapping process conducts iteratively until all the rows in W have been mapped onto M. Finally, the algorithm outputs the mapping relationship and the weight variations.

D. Target Sparsifying and Network Retraining

After weight mapping, large weight variations may still exist due to we cannot find the appropriate weights to be mapped onto the faulty memristors. Fortunately, from Table I we know that zero value weight can be mapped to accommodate stuck-at faults perfectly. We hence perform a target sparsifying on the dense neural network, purposefully forcing some mapped weights which have large variations to be zero value. Network retraining is then conducted to recover classification accuracy from the network sparsification.

As same as in Algorithm 1, the target sparsifying is performed by combined considering weight variation and significance of the weight. Hence, we define a priority metric P as the product of weight significance and variation:

$$P_{ij} = S_{ij} \times V_{ij} \tag{3}$$

where S_{ij} and V_{ij} denote significance value and variation of the weight w_{ij}. During the target sparsifying, each time we choose a certain number of weights which have the large P and force them to be zero value.

Since some weights are set to zero through target sparsifying, network retraining should be conducted to recover the classification accuracy. During the network retraining, the weights having been set to zero should be fixed. We hence modify the training algorithm and add a mask matrix in each layer of the network to prevent zero value weights from being updated. The mask matrix is a binary matrix in which each element corresponds to an element in the weight matrix of a layer. Elements in the mask matrix take value of 0 or 1 only.

```
Algorithm 1 – Weight Mapping
1  //Input: weight matrix W; MBC feature matrix M; Significance list of weights;
2  //Output: Preliminary mapping decision; Weight variations after mapping
3  //Phase 1: Sort weight rows in W given the number of significant weights
4  For i=1 to m
5    Make statistic of the number of significant weights in each row in W;
6    Sort these rows in a descendent order;
7  End For
8  //Phase 2: Map weight rows in W onto the MBC feature matrix M
9  For i=1 to m
10   Fetch row i in W with the most significant weights that hasn't been mapped;
11   For j=1 to m
12     If(row j in M has not been occupied by the mapped weight row)
13       Calculate variation assuming that row i in W is mapped onto row j in M;
14     Else
15       Skip row j;
16     End If
17   End For
18   Map row i in W onto a row in M which produces the smallest variation;
19   Flag this row in M as occupied;
20 End For
21 Output the mapping relationship between W and M and weight variations;
```

Fig. 3. Mapping algorithm

We attach the mask matrices to the training algorithm. As a result, weight updating in Eq.1 should be reformulated as:

$$\triangle w_{ij} = \eta \frac{\partial E}{\partial w_{ij}} wm_{ij} \tag{4}$$

where wm_{ij} is the element in the mask matrix which takes binary value. Obviously, if wm_{ij} takes 0, $\triangle w_{ij}$ is also 0 which actually prevents w_{ij} from being updated by the training algorithm.

Fig.4 illustrates pseudo code of target sparsifying and network retraining algorithm. Algorithm conducts in an iterative manner. Each time the first n weights in weight matrix corresponding to the first n elements in priority list P which have the largest priority are pick out. At the same time, the corresponding elements in mask matrix WM are set to zero. Then network retraining is performed to update weights. If the desirable classification accuracy has not been reached after the retraining, next n weights will be set to zero and network retraining conducts again. Algorithm terminates until the desirable classification accuracy has been reached or the iteration number reaches the designated value.

IV. EXPERIMENTS

A. Experimental Setup

We construct a multi-layer CNN in Caffe for identifying MNIST handwriting digits set. The CNN mainly includes 3 convolutional layers and 1 full-connected layer. Classification accuracy of the original network is about 95.6%. Configurations of the CNN are listed in Table III.

We simulate deploying the CNN onto 4 MBCs in MATLAB. Each MBC has a pair of differential crossbars with the same

Algorithm 2 – Target Sparsifying and Network Retraining

```
1  //Input: weight matrix W; Mask matrix WM; Priority list of the weights P;
2  //Output: Sparsified network
3  While iteration has not been finished or accuracy still can be promoted do
4    Sort elements in P in a descendent order;
5    While P is not empty do //Start target sparsifying
6      Pick out n weights from W corresponding to the first n elements in P;
7      Set corresponding elements in WM and weights in W as zero;
8      For j=1 to b //b is the input batch size in network retraining
9        Error back propagated; //Network retraining
10       Weight updating;
11     End For
12     If accuracy has been reached
13       Break;
14     End If
15   End While
16   Remove the first n elements in P;
17 End While
18 Output the sparsified network;
```

Fig. 4. Algorithm of target sparsifying and network retraining

TABLE III
CONFIGURATIONS OF CNN AND MBCS

Layer	Filter	Channel	Batch	MBC size
Conv1	3×3	1	8	9×8
Conv2	3×3	8	16	72×16
Conv3	3×3	16	32	144×32
FC1				1568×10

Conv: convolutional layer; FC: full-connected layer
MBC: memristor based crossbar

size as the network layer. Configurations of the MBCs (only one crossbar in each MBC is shown) are also listed in Table III.We adopt the conversion scheme proposed in [2] to convert the weight value of the matrix into memristor conductance.

As for fault simulation, we randomly generate stuck-at faults throughout the MBCs. The percentages of the faulty memristors in each crossbar are set to 5%, 10%, 15% and 20%, respectively. Under each fault ratio, SA0 faults of 8% and SA1 faults of 2% are generated randomly [7].

B. Results and Analysis

We evaluate the classification accuracy of the CNN under different fault ratios after applying the proposed scheme. For comparison purpose, we also implement the other two schemes. The first is the one proposed in [15] which targets dense networks to perform network retraining and bi-partite weight mapping. We call this scheme as DRBM for short. The other scheme proposed in [14] utilizes sparse network to tolerate SA0 faults and also conducts weight mapping. We call this scheme as SWAM for short.

We randomly generate 100 different fault distributions in MBCs. For each fault distribution case, above mentioned three schemes are applied and the corresponding classification accuracies of the network are evaluated. Statistics of the evaluated accuracies among the 100 fault distribution cases are listed in Table IV.

As shown in Table IV, under the lower fault ratio, all the three schemes can achieve desirable fault tolerant effect. The average accuracies among the 100 fault distributions are 95.1%, 95.2% and 95.2% , respectively. However, SWAM needs to prune weights of 55% to achieve the desirable accuracy while our scheme only needs to prune (i.e., set to zero value) weights of 7%, as shown in Table V. The reason is that SWAM uses sparsifying technology which is originally conducted for network compression purpose. On the contrary, our scheme performs target sparsifying which prunes the weights according to the faulty cases.

At fault ratio of 10%, our scheme still achieves satisfied fault tolerance in most of the fault distribution cases. As shown in Table IV, the minimal and maximal accuracies are 93.3% and 95.2%. On average our scheme reaches accuracy of 94.1%. While accuracies of SWAM and DRBM degrade. The minimal and maximal accuracies of SWAM / DRBM are 90.2% /89.9% and 93.7% / 93.8%, respectively. The average accuracies are 92.8% and 92.0%. The reason is that SWAM only considers to utilize zero value weights to tolerate SA0 faults. However, there are still many SA1 faults existed in MBC. Moreover, the percentage of pruned weights are about 70% which has almost reached the limit on the network sparsifying if we want to maintain the classification accuracy.

As fault ratio increasing continuously, our scheme still has the best fault tolerant capabilities. On average, our scheme can achieve accuracies of 93.1% and 92.8% at fault ratios of 15% and 20%, respectively. The accuracy losses are within 3%, compared to the faulty free case. However, accuracies of DRBM decrease rapidly, as shown in Table IV. The reason is that DRBM uses dense neural networks that cannot utilize the zero weights to tolerate stuck-at faults. Accuracies of SWAM also start to degrade. The reason is that SWAM has been reached the pruning limit on the network sparsifying. As the fault ratio increasing further, aimless network sparsifying is inefficient on fault tolerance.

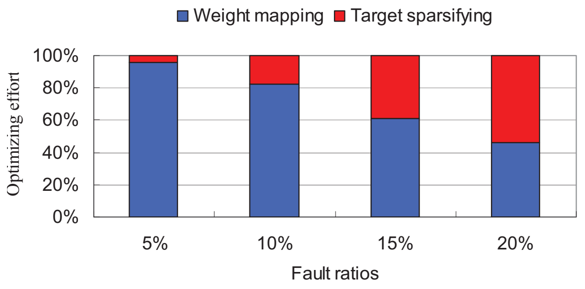


Fig. 5. Contribution of target sparsifying to the fault tolerance

Fig.5 illustrates the percentage of optimization effects contributed from weight mapping and target sparsifying, respectively. It is obvious that contribution of target sparsifying to fault tolerance promotes as the fault ratios increasing. For the high fault ratio (20%), contribution of target sparsifying to

TABLE IV
STATISTICS OF CLASSIFICATION ACCURACIES UNDER DIFFERENT FAULT RATIOS

Fault ratio (%)	**Accuracy breakdown**								
	SWAM [14]			**DRBM** [15]			**Ours**		
	MIN (%)	**MAX (%)**	**AVE (%)**	**MIN (%)**	**MAX (%)**	**AVE (%)**	**MIN (%)**	**MAX (%)**	**AVE (%)**
5	94.5	95.6	95.1	94.8	95.6	95.2	94.9	95.6	95.2
10	90.2	93.7	92.8	89.9	93.8	92.0	93.3	95.2	94.1
15	90.4	93.5	92.1	87.6	93.5	91.6	91.9	94.3	93.1
20	81.0	90.8	86.6	80.5	90.2	86.3	90.2	93.9	92.8

MIN: minimal accuracy; MAX: maximal accuracy; AVE: average accuracy

fault tolerance is more than 50%.

TABLE V
STATISTICS OF PRUNING EFFORTS UNDER DIFFERENT FAULT RATIOS

Fault ratio (%)	**Pruning ratios %**	
	SWAM	**Ours**
5	55.2	7.0
10	70.1	13.6
15	79.6	32.2
20	79.6	52.9

V. CONCLUSIONS

In this paper, we proposed a target sparsifying based fault tolerant scheme targeted MBC executing the neural network applications. We found that most of the stuck-at faults can be accommodated perfectly by mapping a zero value weight onto the differential crossbars. Based on such observation, we first exploit a heuristic algorithm to map weight matrix onto the MBC, aiming at minimizing weight variations in the presence of stuck-at faults. After that, some weights mapped onto the faulty memristors which still have large variations will be purposefully forced to zero value. Network retraining is then performed to recover classification accuracy. Experimental results demonstrate that the effectiveness of the proposed scheme.

ACKNOWLEDGMENT

This work was supported in part by National Natural Science Foundation of China (NSFC) under Grant No.61772061, in part by Natural Science Foundation of Hebei province of China under Grant No. F2017502043, in part by the Fundamental Research Funds for the Central Universities under Grants No. (2017MS114, 2019RC41), in part by Open Subject of State Key Laboratory of Computer Architecture under Grant No. CARCH201802.

REFERENCES

[1] M. Prezioso, F. Merrikh-Bayat, B. D. Hoskins et al. Training and operation of an integrated neuromorphic network based on metal-oxide memristors. Nature, Vol.521, pp.61-64, 2015.
[2] M. Hu, J. P. Strachan, Z. Li et al. Dot-product engine for neuromorphic computing: programming 1T1M crossbar to accelerate matrix-vector multiplication. ACM/IEEE Design Automation Conference (DAC), pp.1-6, 2016.
[3] M. Hu, H. Li, Y. Chen et al. BSB training scheme implementation on memristorbased circuit. IEEE Symposium on Computational Intelligence for Security and Defense Applications (CISDA), pp.80-87, 2013.
[4] A. Shafiee, A. Nag, N. Muralimanohar et al. ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. ACM/IEEE International Symposium on Computer Architecture (ISCA), pp.1-13, 2016.
[5] Z. Zhu, H. Sun, Y. Lin et al. A configurable multi-precision CNN computing framework based on single bit RRAM. ACM/IEEE Design Automation Conference (DAC), pp.1-6, 2019.
[6] S. Kannan, N. Karimi, R. Karri et al. Modeling, detection, and diagnosis of faults in multilevel memristor memories. IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems (TCAD), Vol.34, No.5, pp.822-834, 2015.
[7] C. Chen, H. Shih, C. Wu et al. RRAM defect modeling and failure analysis based on march test and a novel squeeze-search scheme. Transactions on Computers, Vol.64, No.1, pp.180-190, 2015.
[8] D. Niu, Y. Chen, C. Xu et al. Impact of process variations on emerging memristor. ACM/IEEE Design Automation Conference (DAC), pp.1-6, 2010.
[9] T. D. Dongalea, K. P. Patila, S. B. Mullanib et al. Investigation of process parameter variation in the memristor based resistive random access memory (RRAM): effect of device size variations. Materials Science in Semiconductor Processing, Vol.35, pp.174-180, 2015.
[10] G. Merced, J. Emmanuelle, N. Dvila et al. Repeatable, accurate, and high speed multi-level programming of memristor 1T1R arrays for power efficient analog computing applications. Nanotechnology, Vol.27, pp.1-11, 2016.
[11] B. Yan, Y. Chen and H. Li. Challenges of memristor based neuromorphic computing system. Sciece China. Information Sciences, Vol.61, No.6, pp.1-3, 2018.
[12] H. Manem, S. G. Rose, X. He et al. Design considerations for variation tolerant multilevel cmos/nano memristor memory. IEEE Symposium on Great Lakes Symposium on VLSI, pp.287-292, 2010.
[13] C. Liu, M. Hu, J. P. Strachan et al. Rescuing memristor-based neuromorphic design with high defects. ACM/IEEE Design Automation Conference (DAC), pp.1-7, 2017.
[14] L, Xia, M. Liu, X. Ning et al. Fault-Tolerant training with on-line fault detection for RRAM-based neural computing systems. ACM/IEEE Design Automation Conference (DAC), pp.1-6, 2017.
[15] L. Chen, J. Li, Y. Chen et al. Accelerator-friendly neural-network training: learning variations and defects in RRAM crossbar. ACM/IEEE Design, Automation and Test in Europe (DATE), pp. 19-24, 2017.
[16] B. S. Dmitri, S. S. Gregory, R. S. Duncan et al. The missing memristor found. Nature Letters, Vol.453, pp.80-84, 2008.
[17] S. Patrick, W. D. Lu. Defect consideratons for robust sparse coding using memristor arrays. IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH), pp.137-138, 2015.
[18] S. HanH. Mao, J. W. Dally. Deep Compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. International Conference of Learning Representations (ICLR), pp.1-14, 2016.
[19] F. N. Iandola, S. Han, M. W. Moskewicz et al. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and < 0.5MB model size. International Conference of Learning Representations (ICLR), pp.1-9, 2017.