Dynamic Task Remapping for Reliable CNN Training on ReRAM Crossbars

Chung-Hsuan Tung*, Biresh Kumar Joardar†, Partha Pratim Pande‡, Janardhan Rao Doppa‡, Hai (Helen) Li* and Krishnendu Chakrabarty§
*Department of ECE, Duke University, Durham, NC 27708, USA. {chunghsuan.tung, hai.li}@duke.edu
†Department of ECE, University of Houston, Houston, TX 77204, USA. bjoardar@central.uh.edu
‡School of EECS, Washington State University, Pullman, WA 99164, USA. {pande, jana.doppa}@wsu.edu
§School of ECEE, Arizona State University, Tempe, AZ 85287, USA. krishnendu.chakrabarty@asu.edu

***Abstract*—A ReRAM crossbar-based computing system (RCS) can accelerate CNN training. However, hardware faults due to manufacturing defects and limited endurance impede the widespread adoption of RCS. We propose a dynamic task remapping-based technique for reliable CNN training on faulty RCS. Experimental results demonstrate that the proposed low-overhead method incurs only 0.85% accuracy loss on average while training popular CNNs such as VGGs, ResNets, and SqueezeNet with the CIFAR-10, CIFAR-100, and SVHN datasets in the presence of faults.**

I. Introduction

Deep learning (DL) techniques such as convolution neural networks (CNNs) have been deployed in many domains [1], [2]. Resistive random-access memory (ReRAM)-based crossbars can be used to perform high-throughput and energy-efficient CNN training [3]. However, CNN training on ReRAM crossbars must consider reliability challenges. ReRAM cells can be faulty due to fabrication defects; we refer to the faults arising from these defects as "pre-deployment faults". Even if a cell passes manufacturing tests, it can fail during operation due to the limited write endurance of ReRAMs [4]. We refer to these faults as "post-deployment faults." In the presence of pre- and post-deployment faults, ReRAM cells can get stuck at a fixed resistance, resulting in stuck-at-faults (SAFs). SAF impedes model training, leading to a significant accuracy drop. For instance, ResNet-18 trained on a ReRAM crossbar-based computing system (RCS) has a ~76% accuracy drop when 0.1% of the cells are faulty [5].

A number of techniques have been presented in the literature to detect and mitigate the effect of faults in RCS for DL applications [6]–[9]. These include error correction code (ECC), weight remapping, and retraining [9]–[12]. However, these techniques incur high area and performance overheads. For instance, the AN code-based ECC method introduces 6.3% area overhead [10], requires prior profiling of the application to develop the correction table, and is effective only if the number of faults is low [5]. In addition, new (post-deployment) faults can appear during operation. The AN code-based method must therefore update its correction table periodically to address new faults; this can lead to additional performance overhead. Also, existing solutions often do not consider the non-uniform spatial distribution of faults, which can lead to accuracy drop, as we discuss in Section III.A.

In this paper, we present a new fault tolerance solution based on *dynamic task remapping* that can address both pre- and post-deployment faults in RCS. We define "task" as the computations associated with a CNN layer which are executed on a ReRAM crossbar and "task remapping" as the remapping of the stored weights from one crossbar to another. The proposed solution utilizes a built-in self-test (BIST) technique to detect SAFs in each crossbar. Utilizing this knowledge of the fault occurrence, we propose a dynamic task remapping mechanism to judiciously move the computation from one crossbar to another. We refer to the proposed solution as "Remap-D" where "D" represents the dynamic nature of the remapping. Unlike existing remapping-based schemes (e.g., [8], [12]), Remap-D leverages the inherent fault tolerance of CNNs during training. We show that different portions of CNN computations have different degree of fault tolerance. By considering both the ReRAM fault density (defined as the percentage of ReRAM cells that are faulty in the RCS) and the CNN fault tolerance, Remap-D outperforms existing methods in terms of accuracy and performance/area overheads.

The contributions of this paper are as follows. (1) We show that existing methods are ineffective in protecting CNN training from a realistic non-uniform spatial distribution of pre- and post-deployment faults. (2) We propose a BIST-enabled fault-tolerant CNN training technique for RCS (i.e., Remap-D) that relies on dynamic task remapping. (3) Experimental evaluation shows that Remap-D achieves near-ideal accuracy in the presence of both pre- and post-deployment faults. Remap-D achieves 12.5% better accuracy on average compared to the AN code-based ECC.

The rest of the paper is organized as follows. Section II reviews related prior work. Section III presents the proposed method. Experimental results are presented in Section IV. Finally, Section V concludes the paper.

II. Related Prior Work

ReRAM crossbar arrays are natural matrix-vector multipliers [3], [13], and there is a rich body of work on ReRAM-based accelerators for CNN training and inferencing [3], [13], [14]. However, these architectures assume ideal ReRAM behavior. Prior work has shown that CNN models incur significant accuracy degradation when they are trained using faulty ReRAM devices [5], [8], [10]–[12], [15]. Fault detection methods such as the March test [16] and the sneak-path test [6] can detect pre-deployment faults but they introduce high overhead for detecting post-deployment faults. A low-overhead, online testing method based on changepoint detection has been proposed [7]. However, this solution is limited to CNN inferencing and is not applicable for training.

Common fault-mitigation methods include retraining, remapping, and error correction [8], [11], [12]. However, retraining requires an already trained model [12] whereas our goal is to train from scratch. Remapping requires redundant

C.H. Tung, H. Li, and K. Chakrabarty were supported in part by the National Science Foundation (NSF) under grant number CNS-1955196. P.P. Pande and J.R. Doppa were supported in part by the NSF under grant number CNS-1955353. J.R. Doppa was also supported in part by the NSF under grant number OAC-1910213. B.K. Joardar was supported in part by NSF Grant # 2030859 to the Computing Research Association for the CIFellows Project.

978-3-9819263-7-8/DATE23/© 2023 EDAA

hardware, which leads to higher area overhead [12]. ECC-based techniques require encoding and repeated decoding of weights, resulting in high area and performance overheads. A "free parameter tuning" method to compensate for the impact of SAF is proposed [11]. However, this method requires prior profiling to identify "ideal behavior," which can be costly and is often not feasible due to difference between CNN workloads. In [8], the authors proposed an online quiescent-voltage comparison, followed by neuron reordering. However, neuron reordering requires solving an NP-hard problem, which is impractical to implement using ReRAMs.

III. Fault-tolerant Training on RCS

A. Faults in RCS

In this paper, we focus on permanent faults in ReRAM cells, which can be modeled as stuck-at-1 (SA1) or stuck-at-0 (SA0) faults and result in write failures. Permanent faults lead to poor accuracy for both CNN training and inferencing [17]. Permanent faults can arise due to manufacturing defects (pre-deployment faults). Faults can also appear over time as the crossbar is utilized [4]. These (post-deployment) faults can be attributed to the limited write endurance of ReRAMs [4]. Unlike inferencing, CNN training requires multiple weight updates and repeatedly storing intermediate data such as activations, both of which involve frequent write operations. This can result in new faults during the training process. Overall, both pre- and post-deployment faults can result in poor accuracy of the trained model [5], [12], [15].

In addition, the distribution of faulty cells is also important. The majority of the faults tend to be clustered [16]. For instance, almost two-thirds of the faulty cells are clustered in a given area after fabrication, due to the unstable power supply during the forming process [16]. Post-deployment faults can also result in non-uniform fault distribution. During training, not all crossbars experience an equal number of writes. Hence, ReRAM crossbars which are written to more frequently will have more faulty cells than the rest of the RCS. Existing fault-tolerance methods do not consider such non-uniform fault distribution scenarios and may not be effective under these conditions, as we show later in Section IV.C. Hence, a fault-tolerant solution for CNN training on ReRAMs must be applicable under non-uniform fault distribution.

B. Fault-aware CNN Training

1) Overall RCS Architecture

Fig. 1 shows the architecture of the ReRAM-based CNN accelerator adopted in this work [3], [13]. The basic computation unit is the 128 × 128 ReRAM crossbar array, which can perform matrix-vector multiplications (MVMs) [3], [7], [13]. Multiple crossbars are grouped to form an *in-situ* multiply-accumulate (IMA) unit. Each ReRAM IMA also includes a BIST module, input registers, input DACs, sample-and-hold (S&H) circuits, ADCs, shift-and-add (S&A) circuits, and output registers. A tile contains multiple IMAs, along with

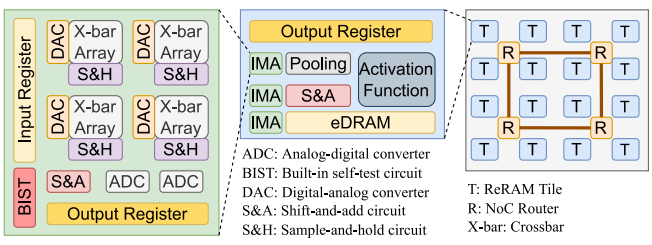


Fig. 1. Illustration of the target RCS architecture.

an eDRAM buffer and other functional units such as circuits to implement pooling layers and activation functions.

It is well-known that CNN training/inferencing generates a lot of traffic and requires multicast support [5]. The output of one crossbar is sent as input to multiple other crossbars, resulting in the multicast traffic. Moreover, the proposed remapping strategy (discussed later in Section III.B.4) also relies on broadcast traffic. Hence, the ReRAM tiles must be connected by a suitable network-on-chip (NoC) that can support broadcast/multicast traffic. Mesh NoCs can handle multicast traffic efficiently [5] and they are easy to design. However, it has been shown that the hop count and energy can be further improved in an RCS by using a concentrated mesh (c-mesh) NoC [13]. A c-mesh NoC connects multiple ReRAM tiles to a single router. The routers are then connected to each other in a mesh configuration. This reduces the overall number of routers in the NoC. Moreover, c-mesh is also able to support efficient multicast, which is necessary for CNN training. Hence, we consider a c-mesh NoC in this work.

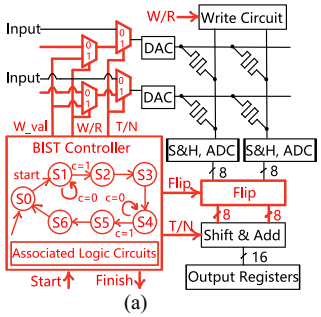
2) CNN Fault Tolerance

CNN models exhibit some inherent fault tolerance [5]. However, different components of a CNN exhibit varying levels of fault tolerance. Remap-D considers this information as a criterion to choose pairs of crossbars for remapping, as described in Section III.B.4. To identify which tasks of a CNN are more fault-tolerant, we inject faults in the ReRAM crossbars associated with specific parts of CNNs and note the accuracy obtained after training. We investigate the relative fault tolerance between the forward and backward phases of CNN training. Our experiments indicate that the backward phase is consistently less tolerant to faults than the forward phase. We found that this observation to be true for all the CNNs and datasets considered in this work. Hence, we need not repeat these experiments for all CNNs. From these experiments, we conclude that the effect of faults on the backward phase is more critical than that on the forward phase. This is because the backward phase is responsible for updating the CNN weights through gradient calculation. Faults in the backward phase result in incorrect gradients, which get accumulated after each weight update, resulting in poor accuracy. Our remapping method uses this information to prioritize the remapping of tasks from crossbars that are associated with these less fault-tolerant portions of the CNNs.

3) Online Fault Detection with BIST

The fault density information is necessary for Remap-D. To determine the fault density of a crossbar during CNN training, we incorporate a low-cost BIST module in each ReRAM IMA. Existing BIST architectures aim to identify the type and location of faults for each ReRAM cell, which can be expensive. However, our remapping solution requires only the overall fault density information; we do not need precise information about each cell. This makes our BIST solution simpler and more area-efficient than conventional BIST. The BIST module is activated after each epoch to determine the fault density of each crossbar. Fig. 2(a) shows the components of the BIST module. The BIST workflow is controlled by a finite-state machine (BIST controller) and associated BIST peripherals. Fig. 2(b) lists the states in the BIST controller.

The BIST controller includes an idle state (S0), three states for testing SA1 faults (S1, S2, S3), and three states for testing SA0 faults (S4, S5, S6). The logic block contains the state-transition logic, the output generation logic, and a counter for controlling the state transitions. To determine the number of



(a)

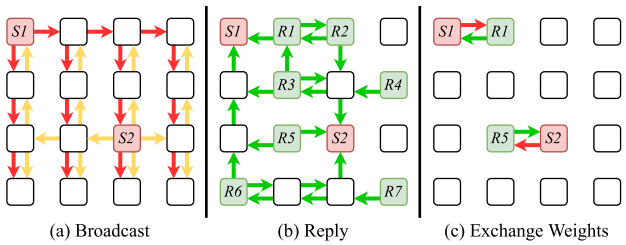
State	BIST Action
S0	Idle state
S1	Write logic "0" to ReRAM cells
S2	Apply read voltage
S3	Read SA1 information
S4	Write logic "1" to ReRAM cells
S5	Apply read voltage
S6	Read SA0 information

(b)

Fig. 2. (a) The BIST module used in the target RCS. The BIST controller and associated peripherals are marked in red and thick lines. This figure is for the illustration purpose only. The signal 'c' is the output of a counter that controls the state transition timings. (b) The states of the BIST controller.

SA1 faults in a crossbar array, the BIST controller goes from state S0 (idle) to state S1 and sets the appropriate values for "W/R" (write/read) and "T/N" (test/normal); "W_val" (write value) is set to write logic "0" to all the ReRAM cells. ReRAM cells are written in a row-by-row manner [18]. Hence, for a $c \times c$ crossbar it will take c cycles to finish writing the entire crossbar. In our case, the row-by-row write to a ReRAM crossbar array in the WR_ZERO state takes 128 ReRAM cycles (since we assume a 128×128 sized crossbar) [18]. Next, the BIST controller switches from state S1 to state S2. In state S2, an input voltage (logic "1") is applied to the crossbar input. Since all the cells in the crossbar are set to logic "0" at this point, the expected fault-free output is also logic "0" (the output of a ReRAM crossbar is the product between the input and stored logic values). However, SA1 cells will result in a non-zero output. The output current reflects the number of SA1 faults in a column. Read operations for all the columns in the crossbar are performed in parallel. Hence, the state machine stays in state S2 for one ReRAM cycle. Next, for obtaining the fault information, the BIST controller switches from state S2 to state S3. In state S3, the peripherals (such as ADC and S&A circuits) process the output to determine the local fault density; this step takes one ReRAM cycle as the outputs of the crossbar columns can be processed with CMOS-based logic and fit in one ReRAM cycle [13]. Overall, SA1 detection takes a total of 130 ReRAM cycles: 128 ReRAM cycles for writing logic "0" to all ReRAM cells, one ReRAM cycle for applying read voltage, and one ReRAM cycle for processing crossbar outputs to calculate the local SA1 fault density. Note that the ReRAM crossbars in the RCS operate at 10 MHz while the CMOS peripherals operate at 1.2 GHz [13], [18]; hence, one "ReRAM cycle" is 100 ns.

The SA0 fault density of a ReRAM column can be determined following a similar approach to detecting SA1. The only difference is the use of the "flip" logic to perform 1's complement. After BIST is completed, the BIST controller returns to state S0 and sets the finish flag. Like detecting SA1, SA0 detection also requires 130 ReRAM cycles. Overall, the entire process takes 260 ReRAM cycles, which corresponds to a negligible performance overhead of 0.13% for CNN training considering full system evaluation [3], [14]. Note that although the BIST procedure introduces two additional writes, they are negligible compared to the total number of writes due to weight updates in one epoch. Hence, the write operations due to BIST have minimal effect on cell write endurance. Also, we have assumed that the CMOS-based peripherals (such as the BIST module itself, ADC, etc.) are fault-free because CMOS manufacturing is a mature process, the devices are likely to be tested thoroughly before being deployed, and aging is of less concern at lower operating frequencies (as is the case here).



(a) Broadcast (b) Reply (c) Exchange Weights

Fig. 3. Illustration of the dynamic remapping method proposed in this work (S: Sender, R: Receiver). Here, we show the three different steps in the remapping procedure: (a) the faulty tiles (senders) broadcast remapping requests to all other tiles, (b) other tiles respond to the senders for potential remapping, and (c) each sender chooses its nearest responding tile as the receiver and remaps the weights.

4) Dynamic Remapping in Remap-D

Existing remapping-based methods either rely on complex remapping algorithms (e.g., a genetic algorithm for solving Knapsack problems [8]) that require additional digital co-processor(s) [8], or they require additional fault-free ReRAM crossbars [12], which may not be always available. In contrast, Remap-D does not require additional redundant hardware or the need to solve NP-hard problems [8]. In Remap-D, we remap the weights of a faulty crossbar (the corresponding tile is referred to as "sender"; recall that a tile has multiple crossbars) with that of another crossbar (its corresponding tile is referred to as "receiver"). The remapping happens if: (a) the "receiver" crossbar has a lower fault density than the "sender" crossbar, and (b) the task mapped on the "receiver" crossbar is more fault-tolerant than the task on the "sender" crossbar. To prevent frequent remapping, we carry out remapping only if the fault density of the sender crossbar exceeds a given threshold. This threshold can be determined by the user depending on the application's accuracy requirement. If the number of faults in a crossbar is less than the threshold, the weights mapped on the crossbar will not be remapped.

The remapping procedure is triggered at the end of each epoch when no computations are being performed and hence this procedure does not lead to pipeline bubbles/stalls. Before the weights are updated for the next epoch, the BIST module for each crossbar array is activated to initiate remapping. Next, each crossbar determines whether to request remapping based on the local fault density and the fault tolerance of the mapped task. The corresponding tiles (i.e., senders) then send a remapping request. Since the sender tile is unaware of the status of other tiles in the RCS, the remapping request is broadcasted to all the remaining tiles. The purpose of this request is to find a potential receiver tile to remap and alleviate the impact of faults on model accuracy. If a tile is not a sender, then it will be categorized as a potential receiver. The receiver tile will reply to the remapping request if the necessary conditions for remapping are satisfied. Each sender may receive multiple responses from potential receiver tiles. The sender then chooses the receiver based on proximity (i.e., NoC hop count) for remapping. This is done to avoid long-range traffic in the NoC, and thereby reduce performance overheads.

Fig. 3 illustrates the overall procedure with a 4×4 mesh NoC: request by broadcast (Fig. 3(a)), responses from multiple potential receivers (Fig. 3(b)), and the weight remapping (Fig. 3(c)). We use the XY tree multicast technique with dimension-ordered routing for efficient broadcast on a c-mesh based NoC [5]. As shown in Fig. 3(a), multiple senders (e.g., *S1* and *S2*) broadcast remapping requests to all other on-chip tiles. The broadcast paths are shown in red and yellow arrows for the two senders, respectively. Next, due to the non-uniform fault distribution, multiple tiles may reply to the

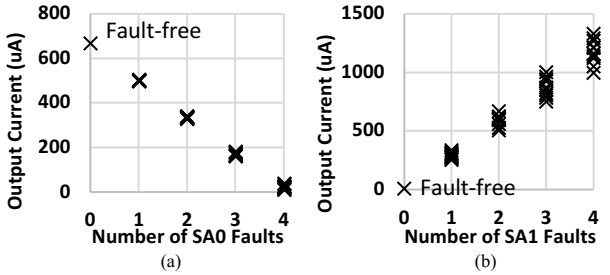


Fig. 4. The magnitude of the crossbar output currents during testing with respect to the number of (a) SA0, and (b) SA1 faults in a ReRAM column.

requests as potential receivers (e.g., *R1* to *R7*). This unicast-based communication is marked with green arrows. Finally, the senders (*S1* and *S2*) choose the corresponding receivers (*R1* and *R5*) based on the physical proximity (measured as hop count) to exchange their weights. Note that the use of an NoC enables us to perform multiple remappings in parallel if the communication paths do not overlap. In Section IV.C, we will show that the performance overhead of remapping is low.

IV. Experimental Results

A. Experimental Setup

Simulation setup: We use PytorX [15] to evaluate CNN training on the RCS. The RCS simulated on PytorX includes 128×128 sized crossbars along with peripherals, such as ADC, DAC, etc. In PytorX, the faults are simulated by making the ReRAM cell resistance to be stuck at high or low values. The PytorX simulations are run on an Nvidia GeForce RTX 3090 GPU with 24 GB memory. To simulate the communication along the c-mesh NoC, we use the BookSim simulator [19]. The message injection rates from the ReRAM crossbars are calculated from the PytorX simulations and are used as input to the Booksim to inject packets. We modified Booksim to implement the proposed remapping method (Fig. 3) to obtain the performance overhead introduced by Remap-D. To calculate the area of the proposed BIST hardware, we use the NeuroSim simulator [14].

Fault density: We consider both pre- and post-deployment faults. The fault distribution is non-uniform due to the clustering of manufacturing defects [16] and unequal wear out during use. To model such a non-uniform fault distribution, we assume that 20% of crossbars have a high density of pre-deployment faults (fault density of 0.4-1%), while the remaining 80% of crossbars have a low fault density (0-0.4%) as an example. Note that high pre-deployment fault densities (beyond 1%) are not realistic from the perspective of the manufactured chip since the faulty chip can be tested offline and discarded [6], [16]. The SA0 to SA1 fault density ratio for pre-deployment faults in RCS is typically 9:1 [11]. For the post-deployment faults, we assume that new faults occur every epoch. Note that new faults may not appear after every epoch; our assumption represents a worst-case scenario. We assume that n% of the crossbars have m% new faulty cells after each training epoch. We vary m from 0.1% to 1% and n from 0.1% to 2% to simulate different fault scenarios.

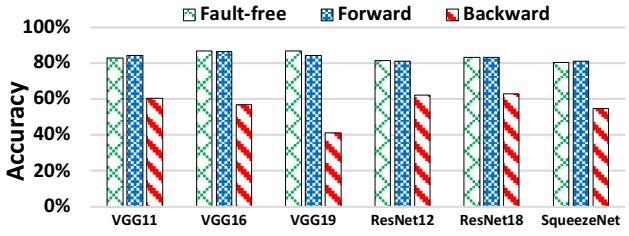


Fig. 5. Accuracy achieved after training the five CNN models in the presence of 2% fault density in the forward and backward phases.

CNN models and datasets: We use VGG-11/16/19 [2], ResNet-12/18 [1], and SqueezeNet [20] as CNN models to demonstrate the effectiveness of the proposed method. We create ResNet-12 by removing 6 convolution layers from ResNet-18. In all cases, we train these CNNs *from scratch* in the presence of different fault configurations. We train each of these models for 50 epochs using the CIFAR-10 [21] dataset. We also demonstrate the scalability and effectiveness of our solution with SVHN [22] and the CIFAR-100 [21] datasets.

Baseline methods: We consider two baseline methods for comparison. The first method uses the AN code [10]. The second baseline remaps the weights to fault-free crossbars based on weight significance [12] (we refer to this solution as "Remap-WS"; WS indicates "weight significance"). This method requires an analysis of pre-trained weights and the availability of redundant ReRAM cells for remapping. In contrast, Remap-D allows training from scratch without the need for additional fault-free crossbars.

B. Determining Fault Tolerance and Fault Density

We first evaluate our BIST module using HSpice simulations. The ReRAM cell behavior in the presence of faults is modeled following [4]. Fig. 4 shows the output current after we apply the test inputs to detect SA0 and SA1 faults. Recall that for detecting SA0 faults, the BIST controller: (a) writes logic "1" to all the ReRAM cells (state S4 in Fig. 2), followed by (b) applying input voltage (logic "1") to all crossbar rows (state S5 in Fig. 2), and finally (c) reads the crossbar output to determine the SA0 fault density (state S6 in Fig. 2). SA1 faults are detected using a similar procedure. Fig. 4 shows how the crossbar output current varies when the number of faulty ReRAM cells in a column is varied. For illustration, here we have considered a 4×4 sized crossbar, but the variation of current in the presence of faults is observed for larger crossbars as well. Fig. 4 shows a positive correlation between the output current and the number of faults. For instance, in Fig. 4(a), less current flows to the output as more cells are stuck at "0" (i.e., open circuit). Hence, through a calibration step, we can determine the number of faulty cells in a ReRAM crossbar by observing the output current after the test inputs are applied by the BIST controller. We follow a similar BIST procedure for detecting SA1 faults.

ReRAM cells are prone to variations [4]. For instance, the SA1 resistance of a ReRAM cell can vary between 1.5 KΩ to 3 KΩ while the SA0 resistance can vary between 0.8 MΩ to 3 MΩ [4]. Therefore, we assess the effectiveness of our BIST module in the presence of these variations. Fig. 4 also shows the output current as a function of the number of faulty cells for different SA0 and SA1 resistances. Here, we have randomly varied the SA0 resistance between 0.8 MΩ to 3 MΩ and the SA1 resistance between 1.5 KΩ to 2 KΩ, respectively, to model the ReRAM fault behavior. Fig. 4 clearly shows that the output current is a reliable indicator of fault density despite the variation in SA0/SA1 resistance of the ReRAM cell.

Fig. 5 shows the accuracy of the models trained with CIFAR-10 in the presence of faults. For this experiment, we inject faults on the crossbars that implement the computation associated with the forward phase tasks (referred to as "forward" in Fig. 5) and the backward phase tasks (referred to as "backward" in Fig. 5). From Fig. 5, we can see that faults in the backward phase lead to more accuracy loss (up to 45%) after training. In contrast, faults in the forward phase have a

very small impact on the training accuracy. These results show that the forward phase is more tolerant to faults than the backward phase. The backward phase is responsible for generating weight updates through gradient calculation. Faults in the backward phase result in incorrect gradients, which eventually causes training to fail. This observation was consistent across all the considered datasets. Hence, we use this fault tolerance information to guide our task-remapping policy. We also performed similar experiments with respect to the type of layer (convolution versus fully-connected) or the position of the layer (first few layers versus last few layers) to identify more inherently fault-tolerant portions of a CNN. However, we did not observe a consistent trend in fault tolerance for these cases across all CNNs and datasets. Hence, we do not consider these cases for Remap-D.

C. *Dynamic Remapping-based Fault Tolerance*

We next demonstrate the effectiveness of Remap-D in the presence of both pre- and post-deployment faults. We assume that 0.5% new post-deployment faults appear on 1% of the crossbars after every training epoch. Fig. 6 shows the accuracy achieved by different CNNs in the presence of both pre- and post-deployment faults using five different fault-tolerant solutions. As shown in Fig. 6, the AN code-based solution incurs a 13.41% accuracy loss on average compared to the fault-free case in the presence of both pre- and post-deployment faults. This happens because the AN code is unable to correct erroneous outputs at higher fault densities [5]. As explained in Section IV.A, the faults are distributed non-uniformly. Hence, some of the crossbars have a significantly higher number of faulty cells than the rest. As a result, the AN code is unable to correct the erroneous outputs of the crossbars with high fault density, resulting in a substantial accuracy drop. In addition, the AN code-based solution incurs 6.3% area overhead [10].

Similar to AN code, static mapping also fails to recover the lost accuracy in the presence of both pre- and post-deployment faults as shown in Fig. 6. This happens as the mapping is static in nature and is performed at $t = 0$. Hence, static mapping cannot mitigate the effects of post-deployment faults. This shows the necessity of dynamic remapping strategies. Remap-WS [12] remaps the top-5% most-significant weights *with faults* to fault-free ReRAM columns to preserve the inferencing accuracy. However, Remap-WS is targeted for inference, where a set of pre-trained weights are available. This assumption does not hold true for CNN training. Fig. 6 shows that Remap-WS leads to a significant accuracy drop. This happens because it only protects 5% of the faulty weights, leaving 95% of the faults unaddressed. Some of these remaining faults can be associated with the less fault-tolerant phases, which impact training accuracy. Moreover, the implementation of Remap-WS is associated

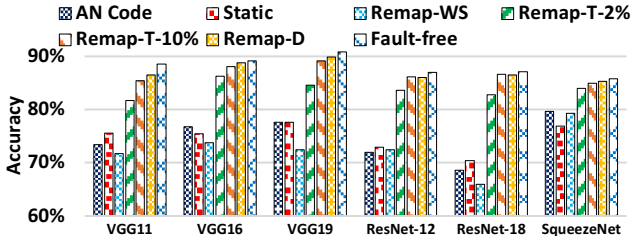


Fig. 6. Accuracy achieved after training the different CNN models in the presence of both pre- and post-deployment faults with different fault-tolerant solutions. "Remap-WS" denotes the remapping method presented in [12]. Remap-T-n% remaps the top n% weights to fault-free crossbars. Our proposed method is denoted as "Remap-D." "Static" denotes the cases that the fault-tolerant mapping is done once at $t = 0$.

with two major challenges: (a) it remaps the data from faulty crossbars to fault-free crossbars; this requires additional spare hardware (hence, high area overhead), and (b) it requires a weight significance classifier to analyze the weights, which can induce considerable overhead because it must be applied repeatedly during training.

For a thorough analysis, we also consider a remapping solution (referred to as Remap-T-n% in Fig. 6) that remaps the topmost n% important weights to fault-free crossbars. Unlike Remap-WS, Remap-T-n% preemptively remaps the top n% important weights in every epoch, irrespective of whether these weights are mapped to faulty cells or not. We define weights to be "important" based on the absolute value of gradients associated with them. As shown in Fig. 6, Remap-T-n% can recover some of the lost accuracies. However, this kind of remapping needs at least n% additional spare hardware. For instance, Remap-T-10% achieves near-ideal accuracy with 10% area overhead. Hence, such a remapping strategy is also not practical.

Similar to Remap-T-10%, the proposed remapping-based solution ("Remap-D" in Fig. 6) is able to achieve near-ideal accuracy. The weights of the faulty crossbars are remapped to other crossbars based on the local fault density and fault tolerance of the phases involved. This prevents significant accuracy loss after training. We observe only 0.91% accuracy drop on average for all the CNNs considered using Remap-D. However, unlike Remap-T-n%, Remap-D does not require spare fault-free crossbars. It utilizes fault-density and fault-tolerance information to remap tasks to the already available crossbars (which may or may not be fault-free); hence, it does not require additional crossbars. Moreover, Remap-D does not require the *a priori* analysis of the weights. Overall, the proposed solution provides a better accuracy-performance-area trade-off compared to Remap-T-n%. Remap-D achieves near-ideal accuracy with relatively lower overhead.

Next, we present the effectiveness of Remap-D under different post-deployment fault scenarios. Since the occurrence of post-deployment faults is workload-dependent and hence unpredictable in general, it is important to study the effectiveness of Remap-D by varying the number and location of faults. Fig. 7 shows the results of this experiment. For this experiment, we inject *m*% new faults on *n*% of the available crossbars after each epoch. In Fig. 7, we examine the effectiveness of the proposed dynamic remapping method by varying the values of *m* and *n* from 0.1% to 1% and from 0.1% to 2%, respectively. Here, we consider up to 1% new faults per epoch as higher post-deployment fault densities are not realistic. If 1% new faults appear after each epoch, then after 50 epochs of training, we may have a maximum of 50% faulty ReRAM cells in some crossbars. This fault density is already too high. If a crossbar has 50% defective cells, the user is likely to discard the product/crossbar. Hence, we do not consider more than 1% new faults per epoch.

Fig. 7 shows the accuracy achieved after VGG19 and ResNet-12 are trained with different fault configurations using the proposed solution. We observe similar trends with other CNNs. As shown in Fig. 7, the accuracy drop using Remap-D is negligible compared to the ideal case. At higher *m* and *n*, the drop in accuracy is higher. However, even in the worst case where 2% of the crossbars have 1% new faults after each epoch, Remap-D experiences an accuracy loss of only 2.48% after 50 epochs of training. Recall that this is a worst-case scenario that is unlikely to occur. In practice, post-deployment

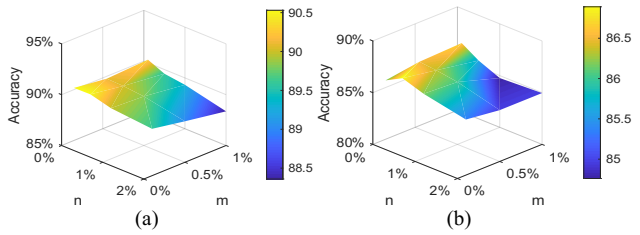


Fig. 7. Accuracy of (a) VGG19 and (b) ResNet-12 under different post-deployment fault scenarios. The parameter *n* represents the percentage of new faulty crossbars in the RCS while *m* represents the percentage of new faulty cells in these crossbars per epoch.

faults will not occur every epoch; hence, the accuracy drop will be considerably lower than what we report here.

Next, we analyze both the area and timing overheads introduced by Remap-D. To analyze the performance impact of Remap-D, we use Booksim [19] to simulate the on-chip traffic. We modified Booksim to incorporate the different steps of the remapping process (Fig. 3). We applied the Monte Carlo method to study the timing overhead for different fault situations. We did 50 rounds of simulation; in each round, we injected faults at different locations of the RCS. Our experiment shows that Remap-D introduces only 0.22% performance overhead on average and 0.36% in the worst-case scenario in our experiments. This happens because the use of an NoC enables multiple sets of communication to happen in parallel. As a result, we can remap the weights from multiple faulty crossbars parallelly. To evaluate the area overhead, we use NeuroSim [14]. NeuroSim provides analytical area models for a typical RCS; these models have been calibrated using circuit layouts. We implement the BIST circuit in NeuroSim. The simulation results show that the BIST module introduces an area overhead of only 0.61%. Overall, Remap-D achieves near-ideal accuracy at a negligible area cost of 0.61% while AN code and Remap-T-10% require 6.3% and 10% area overhead [10], respectively.

D. Scalability

To demonstrate the scalability of Remap-D, Fig. 8 shows the accuracy with larger and more complex datasets, such as SVHN and CIFAR-100. Compared to the CIFAR-10 dataset, which has 10 classes, CIFAR-100 has 100 classes and is more challenging to learn. The SVHN dataset has more images than the CIFAR-10 dataset. SVHN represents a significantly harder, real-world problem (recognizing digits and numbers in natural scene images). Fig. 8 shows the accuracy achieved after training on a faulty RCS using Remap-D. Here, we assume both pre- and post-deployment faults. As shown in Fig. 8, Remap-D has an average accuracy loss of 1.32% on the CIFAR-100 dataset, while the models without a fault-tolerance solution experience an accuracy drop of 33.42% on average. For the SVHN dataset, Remap-D has an accuracy drop of at most 0.45% compared to the fault-free case. The results in Fig. 8 demonstrate that the proposed method is applicable to more complex CNN datasets as well.

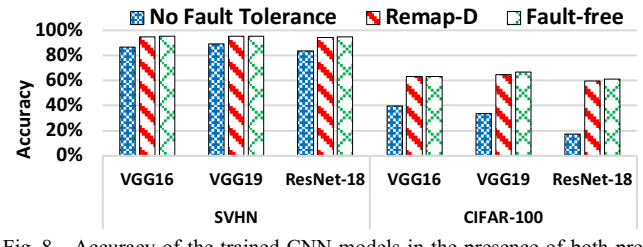


Fig. 8. Accuracy of the trained CNN models in the presence of both pre- and post-deployment faults with different datasets. The pre- and post-deployment fault configurations are the same as that of Fig. 6.

V. Conclusion

We have investigated the inherent fault tolerance in different CNN components and leveraged this to develop a new remapping-based solution for training. We have also developed a BIST-based fault detection method to determine the fault density of crossbars to guide the dynamic remapping technique. By remapping the less fault-tolerant tasks from high fault-density crossbars to less faulty ReRAM crossbars, we can train VGGs/ResNets/SqueezeNet from scratch, which results in only 0.85% accuracy drop on average. The timing overheads during training for fault detection and additional on-chip traffic are 0.13% and 0.35%, respectively. Moreover, the BIST circuit introduces only 0.61% area overehead; the additional traffic introduces less than 0.5% power overhead. The proposed solution enables fault-tolerant CNN training with negligible area, timing, and power overheads.

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in IEEE CVPR, 2016.

[2] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," in ICLR, 2015.

[3] L. Song et al., "PipeLayer: A Pipelined ReRAM-Based Accelerator for Deep Learning," in IEEE HPCA, 2017.

[4] A. Grossi et al., "Resistive RAM Endurance: Array-Level Characterization and Correction Techniques Targeting Deep Learning Applications," IEEE T-ED, vol. 66, pp. 1281–1288, 2019.

[5] B. K. Joardar et al., "Learning to Train CNNs on Faulty ReRAM-based Manycore Accelerators," ACM TECS, vol. 20, 2021.

[6] S. Kannan et al., "Sneak-Path Testing of Crossbar-Based Nonvolatile Random Access Memories," IEEE Trans Nanotechnol, vol. 12, pp. 413–426, 2013.

[7] M. Liu and K. Chakrabarty, "Online Fault Detection in ReRAM-Based Computing Systems for Inferencing," IEEE TVLSI, pp. 1–14, 2022.

[8] L. Xia et al., "Fault-Tolerant Training with On-Line Fault Detection for RRAM-Based Neural Computing Systems," in Proc. DAC, 2017.

[9] I. Yeo et al., "Stuck-at-Fault Tolerant Schemes for Memristor Crossbar Array-Based Neural Networks," IEEE TED, pp. 2937–2945, 2019.

[10] B. Feinberg et al., "Making Memristive Neural Network Accelerators Reliable," in IEEE HPCA, 2018.

[11] G. Jung et al., "Cost- and Dataset-free Stuck-at Fault Mitigation for ReRAM-based Deep Learning Accelerators," in DATE, 2021.

[12] C. Liu et al., "Rescuing Memristor-based Neuromorphic Design with High Defects," in Proc. DAC, 2017.

[13] A. Shafiee et al., "ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars," in ACM/IEEE ISCA, 2016.

[14] P.-Y. Chen et al., "NeuroSim: A Circuit-Level Macro Model for Benchmarking Neuro-Inspired Architectures in Online Learning," IEEE TCAD, vol. 37, pp. 3067–3080, 2018.

[15] Z. He et al., "Noise Injection Adaption: End-to-End ReRAM Crossbar Non-ideal Effect Adaption for Neural Network Mapping," in Proc. DAC, 2019.

[16] C.-Y. Chen et al., "RRAM Defect Modeling and Failure Analysis Based on March Test and a Novel Squeeze-Search Scheme," IEEE Trans. Comput., vol. 64, pp. 180–190, 2015.

[17] L. Xia et al., "Stuck-at Fault Tolerance in RRAM Computing Systems," IEEE JETC., vol. 8, pp. 102–115, 2018.

[18] C. Xu et al., "Design Implications of Memristor-Based RRAM Cross-Point Structures," in DATE, 2011.

[19] N. Jiang et al., "A Detailed and Flexible Cycle-Accurate Network-on-Chip Simulator," in IEEE ISPASS, 2013.

[20] F. N. Iandola et al., "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size." arXiv, 2016.

[21] A. Krizhevsky nd G. Hinton, "Learning Multiple Layers of Features from Tiny Images," technical report, Univ. of Toronto, 2009.

[22] Y. Netzer et al., "Reading Digits in Natural Images with Unsupervised Feature Learning," in NIPS Workshop, 2011.