ATT: A Fault-Tolerant ReRAM Accelerator for Attention-based Neural Networks

Haoqiang Guo, Lu Peng, Jian Zhang, Qing Chen, Travis D LeCompte
School of Electrical Engineering and Computer Science
Louisiana State University
Baton Rouge, LA, 70803, USA
{ghaoqi1, lpeng, jz, qchen11, tlecom3}@lsu.edu

***Abstract*—Crossbar-based resistive RAM has been widely used in deep learning accelerator designs because it largely eliminates weight movement between memory and processing units. The high-density storage and low leakage power make it a good fit for edge/IoT devices. However, existing ReRAM designs for traditional neural networks cannot support Attention-based Neural Networks, which are stacked with encoders and decoders instead of convolutional layers or fully connected layers. In addition to matrix-matrix multiplications in traditional neural networks, an encoder or a decoder also includes the attention mechanism, the layer normalization and the gaussian error linear unit. These new characteristics make the data flow far more complicated than that of a convolutional layer. Faulty ReRAM devices are additional obstacles when mapping weights that severely degrade computation accuracy. Existing hardware redundancy strategies that are unaware of application characteristics usually result in inefficient designs.**

In this work, we analyze the data flow of these attention-based neural networks and propose a ReRAM-based accelerator with a dedicated pipeline design for Attention-based Neural Networks. When considering cells with hard faults in crossbars, we further propose NuXG, a non-uniform redundancy strategy, to meet accuracy requirements and save energy consumption by decreasing the redundancy ratio. Finally, we evaluate results and demonstrate that the proposed can achieve more than two times improved performance over existing redundancy schemes in both power efficiency and throughput for Attention-based Neural Networks. Moreover, it also significantly outperforms an NVIDIA GPU.

***Index Terms*—Attention Neural Networks, Fault tolerance, ReRAM accelerator**

I. Introduction

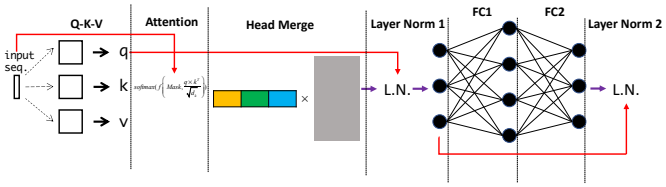
Processing-in-Memory (PIM) platforms are more and more popular in accelerating neural network applications due to fewer data movements compared to FPGA and ASIC implementations [1]–[3]. Essentially, computations in both convolutional layers and fully-connected layers can be transformed to matrix-matrix and matrix-vector multiplications. These linear algebra operations can be mapped to crossbars to achieve excellent performance [4], [5]. Weight pruning on PIM is seldom studied because mapping irregular computation patterns to crossbars is challenging.

Attention-based Neural Networks (AttNNs) have been proven to significantly outperform convolutional neural networks (CNNs) and recurrent neural networks (RNNs) in the wide variety of Natural Language Processing (NLP) tasks [6]–[9]. In general, it is impractical to deploy a deep learning network to an accelerator designed for a different network. Specifically for AttNNs, the gaussian error linear unit (gelu) activation function [10] is unsupported on pre-existing platforms. Additionally, it includes the multiplication between intermediate matrices, shortcut-based layer normalization and vector concatenation. Instead of simply concentrating on the attention module [11], we analyze the AttNNs by identifying the performance bottlenecks and then comprehensively designing a hardware accelerator for AttNNs. We propose an architecture (ATT) for AttNNs which explores intra-layer pipelining to fully utilize the on-chip hardware. Our pipeline design addresses hardware hazards and includes modules for AttNN-specific operations including attention-based mechanisms and gelu. Resistive RAM (ReRAM) crossbars are leveraged to implement weight stationary data flow (the multiplication between the neuron matrix and the weight matrix). ATT goes a step beyond previous in-situ computation works by taking hard faults into consideration.

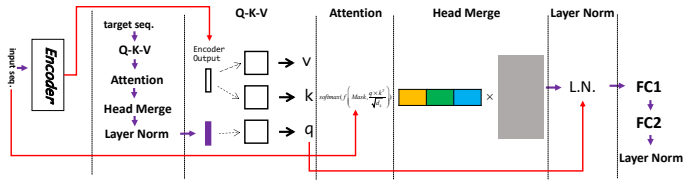
Hard faults in crossbars that are caused by current immature fabrication and process variation have become a issue for ReRAM accelerators. The most common hard faults are Stuck-At-Faults (SAF) [12], which include Stuck-At-Zero (SA0) and Stuck-At-One (SA1). Once a cell is identified as SAF, its resistance cannot be changed. Hence, weights cannot be programmed into the cell. A crossbar with the incomplete matrix will definitely leads to accuracy degradation because the inference accuracy is sensitive to the number of defective memristors [13]. Some existing works have attempted to recover the application accuracy through various methods: software solutions [14] either explore re-training models or other techniques to make the inference reliable; hardware solutions [14] attempt to search for a trade-off between the hardware overhead and accuracy requirements. While the hardware solutions have better performance compared with software approaches, they are still not efficient enough since they do not consider characteristics of algorithms. In this work, we design a heuristic hardware redundancy algorithm to improve fault-tolerance of ATT by taking the algorithm's properties into account.

The contributions of this work can be listed as follows:

- A pipelined accelerator ATT for AttNNs is proposed. Hardware hazards and AttNN-specific module designs have been considered.



(a) An encoder



(b) A decoder

Fig. 1: The data-flow of attention-based blocks

- We design a heuristic redundancy algorithm that takes layer-wise sparsity into account and makes ATT fault tolerant.
- We evaluate the performance and the power efficiency of ATT and compare it to GPU and other redundancy work.

II. Background

A. AttNN Algorithms

Deep neural networks are characterized by stacked layers, including convolutional and fully connected layers. AttNNs are stacked with two types of basic blocks: the encoder and the decoder. They are briefly illustrated in Fig.1. Roughly, both the encoder and the decoder consist of five sections: the Q-K-V, the Attention, the Head Merge, the Layer Normalization and the Fully Connected Layer sections.

$$\begin{cases} q^i_{h_d} = W^{q_i}_{h_d} \times x_{i-1} \\ k^i_{h_d} = W^{k_i}_{h_d} \times x_{i-1} \\ v^i_{h_d} = W^{v_i}_{h_d} \times x_{i-1} \end{cases} \quad (1)$$

The Q-K-V section computes q,k,v vectors for each head according to trained model weight matrices $W^{q_i}_{h_d}$, $W^{k_i}_{h_d}$ and $W^{v_i}_{h_d}$, where $h_d \in (1, D)$ and D is a hyper parameter that represents the number of heads. The specific computation is in equation (1). The computation of q,k,v vectors in one head and across different heads are independent. In this notation, x_{i-1} denotes the inputs of the (i-1)-th Xcoder [1], while $q^i_{h_d}$, $k^i_{h_d}$ and $v^i_{h_d}$ denote the q,k,v vectors of the h_d-th head belonging to the i-th Xcoder.

The Attention section scores each word of the input sentence against other words in the same sentence. The scores are computed by softmax normalizing the dot products of the q vectors and the k vectors, then the scores are multiplied by the v vectors. As can be seen in equation (2), the matrices involved in this step are intermediate results, which is one of the different computation patterns from previous basic blocks.

$$Vin^i_{h_d} = Softmax(\frac{q^i_{h_d} \times (k^i_{h_d})^T}{\sqrt{D}}) \times v^i_{h_d} \quad (2)$$

The Head Merge section first concatenates output vectors from different heads. Next, the concatenation vector is multiplied by a trained weight matrix W^i_{hm}. Equation (3) defines the operations in this step.

$$Xin^i = (Vin^i_{h_1}, Vin^i_{h_2}, \cdots, Vin^i_{h_D}) \times W^i_{hm} \quad (3)$$

The Fully connected layer is the same as previous SOTA networks. There is one hidden layer in this step, so two weight matrices W^i_1 and W^i_2 are required as shown in equation (4). Here i denotes the i-th Xcoder. We should note that the activation function is the gelu function [10], instead of the sigmoid or ReLU functions used in traditional neural networks. The gelu function is defined in equation (5).

$$x^i = f(W^i_2 \times f(W^i_1 \times Xin^i + b^i_1) + b^i_2) \quad (4)$$

$$gelu(x) = \frac{x}{2}(1 + \frac{e^{\sqrt{\frac{2}{\pi}}(x+0.044715x^3)} - e^{\sqrt{\frac{2}{\pi}}(x+0.044715x^3)}}{e^{\sqrt{\frac{2}{\pi}}(x+0.044715x^3)} + e^{\sqrt{\frac{2}{\pi}}(x+0.044715x^3)}}) \quad (5)$$

.

As shown in equation (6), the Layer Norm performs layer normalization on each elements of the matrix. For a 3D matrix (batch_size×seq_length×model_size), layer normalization performs

$$y = \frac{x - E[x]}{\sqrt{Var[x] + \epsilon}}\beta + \gamma \quad (6)$$

Popular AttNNs such as Transformer [6], Bert [7], XLNet [8], XLM [9] and GPT2 are stacked with Xcoders. Transformer is composed of encoder layers and decoder layers, while Bert, XLNet and XLM include only encoder layers. GPT2 only contains decoder layers.

B. In-ReRAM Computation and Hard Faults

A ReRAM array consists of interconnected filamentary ReRAM cells whose states can be switched between a High Resistant State (HRS or OFF-state) and Low Resistant State (LRS or ON-state) by the peripheral read/write circuit. Since the state value can be seen as a matrix element, a matrix can easily programmed into the ReRAM array. An input vector (each component is 16 bits) can be converted to analog signals bit by bit via 1-bit Digital-to-Analog Converters (DACs). The voltage signals are applied to each word line, then the current of each bit line can be regarded as the dot-product between the voltage vector and the resistance vector. The current is converted to digital signals by Analog-to-Digital Converters (ADCs), then shifted and accumulated in the Shift&Add unit to get the the final dot-product. For simplification, we use a single crossbar model [15] to demonstrate the proposed redundancy algorithm in this work. ReRAM can be designed

[1]We use the Xcoder to denote the encoder or the decoder in following sections.

as either a crossbar architecture or a grid architecture. Our work focuses on the former, because it can overcome the sneak current issue when a dedicated selector is inserted in each cell.

Limited by current immature fabrication and process variation, faulty devices and cells frequently appear. A filamentary ReRAM cell structure is essentially a metal–insulator–metal (MIM) structure, which is simply an oxide material sandwiched between two metal electrodes. Immature fabrication causes variations in the oxide thickness of cells. Cells are then initialized by applying a high voltage for a certain period of time, known as forming. Over Forming (OF) defects or reset failures lead to the Stuck-At-Zero (SA0) fault, whose resistance is fixed at LRS. Stuck-At-One (SA1) faults instead have the resistance fixed at HRS. However, it is important to note that faulty cells with SAFs are still readable.

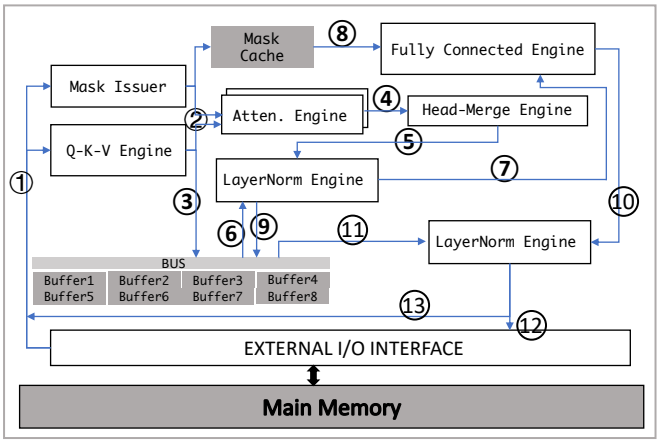


Fig. 2: Overall architecture

III. ATT Architecture Design

A series of words (a sentence) are first converted to a matrix (this process is called embedding in the machine learning community) on the CPU. Each word in the sentence accounts for a row (each component is 16 bits), whose length depends on the embedding algorithm. Hence, the width of the matrix is the embedding size, while the height equals the length of the sentence. For a batch of sentences, we can get a tensor with three dimensions of which the third dimension is the batch size. This tensor is stored in main memory, and our accelerator takes it as input and outputs the results to main memory. The CPU then translates the outputs to the objective sentence.

A. Pipeline Architecture

Fig.2 illustrates the architecture of the accelerator, which fetches data from main memory through an external I/O interface. The accelerator is equipped with eight on-chip buffers to speed-up the access to different types of data from different pipeline stages. These buffers are divided into two sets (Buffer1∼4 and Buffer5∼8). The buffers in each set are read and written in a round-robin manner to enable the pipeline to avoid stalls incurred by buffer access conflict.

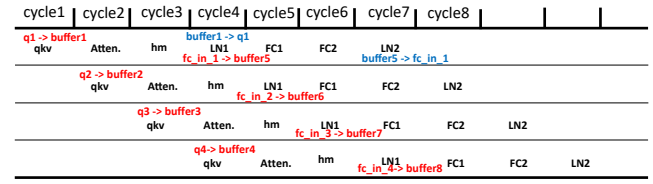
Q-K-V engine. This engine employs crossbar-based ReRAM to perform the matrix-vector multiplication introduced in equation (1). Weights $W_{h_d}^{q_i}$, $W_{h_d}^{k_i}$ and $W_{h_d}^{v_i}$ in equation (1) are programmed in crossbars before inference. An input matrix is fetched from main memory by the Q-K-V engine ①. Each row (with 16-bit components) is converted to analog signals by Digital-to-analog converters (DAC) bit by bit. These analog signals are applied to word lines of crossbars. The currents flowing out from each bit line are the partial sums of the expected inner product. Analog-to-digital converters (ADC) convert these analog signals back to digital signals. DACs, ADCs and Shift&Add units are integrated into the Q-K-V engine to enable the matrix-vector multiplication to be calculated in the analog domain. At the end of this stage, the q,k and v vectors are forwarded to the Attention engine ②, and the q vectors are stored to buffers ③. This stage is represented as **qkv** in the following discussion.

Mask issuer. The Mask issuer computes masks according to the matrix fetched by the Q-K-V engine. Two different masks are output from this module. One is forwarded to the Attention engine, and the other is stored in the Mask cache to filter inputs and outputs of the Fully Connected engine. the Mask issuer work in parallel with the Q-K-V engine, so this stage can be seen as a parallel stage with **qkv**.

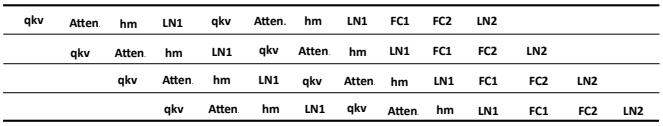
Attention engine. This engine performs operations shown in equation (2). As can be seen, the matrices involved in the computation are all intermediate results, which are generated by the previous **qkv** stage: Q-K-V engine. The v vectors received from **qkv** are stored in a local cache. It is time consuming to program matrices into the crossbar to execute matrix related multiplication during the inference process. Additionally, the matrices involved are very small. Hence, we tailor a matrix-matrix multiplication engine to implement it. There are three sub-stages in this engine. The first is computing the inner-product between q vectors and the transposition of k vectors. The second is to softmax the previous inner-product results. The third is multiplying the softmax results by the v vectors. We denote this stage as **atten** in the following discussion. The outputs of this stage are forwarded to the Head-Merge engine ④. Two Attention engines are deployed to avoid structural hazards. Details will be discussed in the next section.

Head-Merge engine. This engine executes operations shown in equation (3). Weight matrix W_{hm}^{i} in equation (3) is programmed into crossbars prior to inference. Each head generates one $Vin_{h_d}^{i}$. These $Vin_{h_d}^{i}$ vectors are forwarded to the Head-Merge engine (④) and concatenated as one vector first, then the components of this vector are truncated to 16 bits. Finally, they are multiplied by W_{hm}^{i} in the same way as the Q-K-V engine. The results are forwarded to the LayerNorm engine ⑤. We call the stage in the Head-Merge engine **hm** hereafter.

LayerNorm engine. There are two LayerNorm engines embedded on the accelerator. The first one works following the Head-Merge engine. One input to this LayerNorm engine is the output of the Head-Merge engine ⑤, and the other input is



(a) The encoder pipeline



(b) The decoder pipeline

Fig. 3: Pipeline analysis

the q vectors produced two stages before (**qkv**). The q vectors are fetched from on-chip buffers ⑥. One copy of outputs of are forwarded to the Fully connected engine ⑦. Another copy of outputs are stored in buffers ⑨. We use **LN1** to denote this first LayerNorm engine. The second LayerNorm engine carries out the layer normalization the after fully connected layers. It also takes in two inputs. One input is the output of the first LayerNorm engine. This input must be loaded from buffers ⑪. The other input is the outputs of the Fully Connected engine ⑩. **LN2** is used to represent this step in the following sections. Where the outputs of the second LayerNorm engine flow is determined by whether or not there has an additional layer next.

Fully Connected engine. There are two pipeline stages in this engine. One stage is the fully connected layer corresponding to the weight matrix W_1^i in equation (4). The other stage is the fully connected layer characterized by weight matrix W_2^i in equation (4). They are represented as **FC1** and **FC2** respectively hereafter. The engine takes in masks from the Mask cache ⑧ and outputs from the LayerNorm engine ⑦. The activation function in this engine is gelu. The mechanism of matrix-vector multiplication is the same as the Q-K-V engine and the Head-Merge engine.

A specific pipeline of an encoder layer with four input sentences is illustrated in Fig.3a. Fig.3b depicts a decoder layer with four input sentences.

B. Hazard Analysis

Data hazards and structural hazards were taken into account when we designed the pipeline. More specifically, Write after Read hazards (WAR), the Attention engine conflict and the LayerNorm engine conflict are considered in this section.

Data hazards. There are eight buffers deployed on the chip. These buffers can be divided into two sets. The first set include buffer1, buffer2, buffer3 and buffer4. The remaining four buffers belong to the second buffer set. Let's take the encoder pipeline in Fig.3a for example. In the first cycle, the q vector (represented as q1 in red) is produced by the Q-K-V engine for the first input sentence, and is stored to buffer1 (③ in Fig.2). The q vector will be used in the fourth cycle. The size of the q vectors depends on models and is proportional to the batch size, so deploying another three buffers (buffer2, buffer3 and buffer4) for the following three input sentences can eliminate overwrite issues. The q vectors produced in the second, the third and the fourth cycle are stored to buffer2, buffer3 and buffer4 respectively. In the fourth cycle, the LayerNorm engine reads q1 from buffer1 (⑥ in Fig.2), and writes its results (represented as fc_in_1 in red) to buffer5 (⑨ in Fig.2). In the fifth cycle, the q vector produced by the Q-K-V engine for the fifth input sentence is able to be written to buffer1 because there are no WAR hazards in buffer1. Similarly, the second set of buffers (buffer5∼8) work in the same way with the first buffer set.

To make the above idea work, data are arranged as a First-In-First-Out (FIFO) queue in each buffer. For buffers in the same set, they are accessed in a round-robin manner.

Structural hazards. Structural hazards appeared when more than one encoder layers or decoder layers are stacked to a network. Bubbling the pipeline once can not address this issue. Deploying too many hardware modules is energy inefficient. Through analysis, we find that two LayerNorm engines (one for LN1 and the other for LN2) and two Attention engines are good enough to eliminate structural hazards if we insert a few stall cycles, such as 15 stall cycles for Transformer (6 encoders and 6 decoders).

C. Module Designs

For the Q-K-V engine, the Head-Merge engine and the Fully connected engine, we feed crossbars one bit at a time for the 16-bit truncated input neurons. A 1-bit DAC is equipped to each row of crossbars to transform digital signals to analog signals. Each crossbar cell is four bits wide, so two adjacent bit lines are used to represent weight values. The current in each bit line is transformed back to digital signals by an 8-bit ADC, then shifted and accumulated in Shift&Add units to get the final dot product.

The layer normalization in equation (6) is applied to all the elements of the sum matrix of the two input matrices. First, the mean value $E[x]$ and the variance $Var[x]$ is calculated along the sentence axis (which means the mean and the variance are different across sentences). The computation of the mean value $E[x]$ is essentially a sum reduction, and the variance $Var[x]$ can be computed according to $Var[x] = E[x^2] - (E[x])^2$. A sum reduction tree is implemented in this unit. Partial sums are accumulated in a register file. $E[x^2]$ and $E[x]$ are computed in in the same way. When both the mean and the variance are ready, we can then perform the normalization for each element. A set of parallel subtractors and dividers perform normalization after the mean computation and the variance computation.

From the equation (5), $\sqrt{\frac{2}{\pi}}$ is a constant, so the basic component in the gelu circuit is the exponent generator, which we adapt from existing literature [16].

The softmax function is

$$softmax(x_i) = \frac{e^{x_i}}{\Sigma_{j=0} e^{x_j}} \tag{7}$$

A sum reduction tree is implemented to compute the denominator, and a set of dividers work in parallel afterwards. To meet accuracy requirements, we use the following formula

$$\frac{e^{x_i}}{\Sigma_{j=0}e^{x_j}} = \frac{e^{x_i - x_{MAX}}}{\Sigma_{j=0}e^{x_j - x_{MAX}}} \tag{8}$$

to compute the softmax results. This idea is borrowed from existing hardware literature [39].

IV. Fault tolerant Strategy for ReRAM Engines

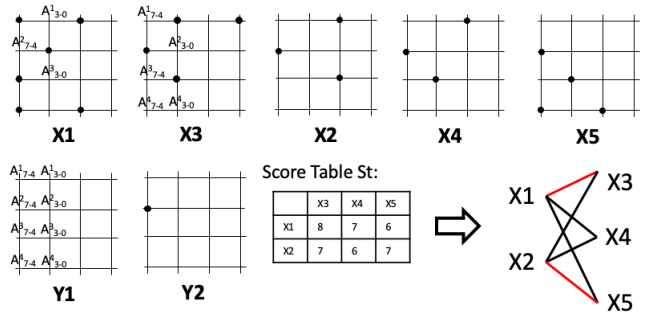


Fig. 4: For simplicity, we use 8 bits to represent input neurons in this example. The last cell in the 2nd bit-line of X1 can be set to an extremely low conductance.

The Q-K-V engine, the Head-Merge engine and the Fully Connected engine each play key roles in ATT. Accuracy loss within these engines is significant if ATT does not consider hard faults in ReRAM. However, vanilla redundancy algorithms lead to energy-inefficient performance. This section aims to propose a heuristic redundancy strategy: Non-uniform Xbar Grouping algorithm (NuXG).

A. Preliminaries and Subroutines

As accuracy is limited by resistance precision, typically more than one cell is used to store a weight. The cell resolution of the ReRAM we utilized is 4 bits, the same as with PRIME [1]. Hence, two adjacent cells on the same word line are leveraged to store high 4 bits and low 4 bits of a 8-bit weight. These cells can only be used to store weights if they are both fault free. Without loss of generality, we use 4×4 crossbars (X1∼X5, black dots represent faulty cells) in Fig.4.

Three terms need to be defined first. **Virtual crossbar (Vc):** it refers to a virtual crossbar, which is made up of cells at the best condition in that position, for a group of crossbars. For example, the virtual crossbar of $\{X1, X3\}$ is Y1. Similarly, Y2 is the virtual crossbar of $\{X2, X5\}$. **Storage capacity (Sc):** it refers to the maximum number of weights a crossbar or a group of crossbars can store. As presented in Fig.4, the storage capacity of X5 is 4, because at most four weights can be stored in it. For $\{X2, X5\}$, the storage capacity of this crossbar group equals that of its virtual crossbar, which is 7 for Y2. **Subgroup:** it refers to a group of physical crossbars corresponding to a Vc. For $\{\{X1, X3\}, \{X2, X5\}\}$, both $\{X1, X3\}$ and $\{X2, X5\}$ are subgroups.

With the constraint of accuracy requirements, the weight sparsity differs across layers (a layer is an Xcoder in AttNNs). Weights of AttNNs refers to weight matrices ($W^{q_i}_{h_d}$,$W^{k_i}_{h_d}$,$W^{v_i}_{h_d}$ in equation(1), W^i_{hm} in equation(3) and W^i_1, W^i_2 in equation (4)) in an Xcoder, instead of weights between two neighboring layers. If these weight matrices of an Xcoder are sparse, it is unnecessary for the crossbars that store them to have large storage capacity. On the other hand, for layers with dense weight matrices, larger storage capacity results in less accuracy loss. For simplicity, we group layers according to their sparsity. The **Low** group includes layers having low sparsity, which means they are more sensitive to pruning than other layers. Layers of which most weights can be pruned without degrading the accuracy are grouped into the **High** group.

Algorithm 1: SeedSearch Subroutine

```
while exist G_i s.t. |G_i| < gi do
    if the Sc of G_i's last subgroup ≥ ri then
        select the Xbar X* with the largest Sc from
          Res, and append to G_i as the last subgroup;
        remove X* from Res;
        update the Vc of G_i;
update St by computing capacities between each Vc
  and each Xbar in Res
```

The rest of the layers are grouped into the **Medium** group. There are two things that should be noted here. First, the sparsity value thresholds (or split points) for these groups are configured manually. The configuration of thresholds affects layer group results. Second, the configuration of split points can be different from networks. One can view each individual layer as a group. However, this strategy brings no accuracy improvement and increases the complexity of the proposed grouping algorithm. We have shown that three groups are enough to meet the accuracy budget.

The redundancy process can be modeled as maximum-weight matching in a bipartite graph. Suppose we have five crossbars in Fig.4, namely X1,X2,X3,X4,and X5. If we need to find redundancy crossbars for X1 and X2 respectively, a score table (St) is constructed first. The content of the table denotes the **Sc** when the corresponding two crossbars are combined one group. For example, the first row of the score table in Fig.4 stands for the **Sc** (8,7 and 6) when X3,X4 and X5 is selected as the redundancy for X1 in a line. The objective is to maximize $\Sigma score(Xi, Xj)$ ($i \in \{1, 2\}$ and $j \in \{3, 4, 5\}$), where Xj can only appear once. Next, the problem is converted to find the maximum sum of elements in different rows and different columns in the score table. X1, X2 can be seen as a set of points, and X3, X4, x5 is another set of points. The capacity value in the score table represents the edge weight between the two corresponding points in the above two sets. Fortunately, this problem can be addressed using a Hungarian matching algorithm (the Kuhn-Munkres algorithm) [18]. In Fig.4, the outputs of Hungarian matching algorithm are edges (X1,X3) and (X2,X5), so the redundancy pairing results are

as follows: X3 acts as the redundancy for X1, and X5 acts as the redundancy for X2. NuXG in the next section iteratively call this subroutine to find an optimized grouping solution. This subroutine is marked as **MaximumWeightMatchings()** in line 7 of algorithm 2.

Another subroutine that will be used is **SeedSearch()**. The example in Fig.4 can also be used to illustrate when this subroutine is needed. After the above **MaximumWeight-Matchings()**, two groups are identified: G1 = $\{\{X1, X3\}\}$, G2 = $\{\{X2, X5\}\}$, and the crossbar $X4$ is left. The capacity of each subgroup is 8 and 7 respectively. Assuming G2 needs another crossbar or crossbar group with **Sc** larger than 6, we are supposed to add X4 to G2 and find redundancy crossbars for X4 till they can meet the capacity requirement (larger than 6). This process is demonstrated in algorithm 1. Line 1 finds

Algorithm 2: NuXG algorithm

Input: Xbar matrices: X = $\{X1, X2, \cdots, XN\}$; $\{(g1, r1), (g2, r2), \cdots, (gM, rM)\}$, gi refers to the required # for the Vc of the i-th group, and ri is the minimum Sc of the Vc in the i-th group

Output: Xbar-group sets: G = $\{G_1, \cdots, G_M\}$

```
1  Res = X;
2  Sort crossbar matrices in X by the Sc;
3  Add crossbars with the Sc ≥ ri to G_i;
4  call SeedSearch();
5  while |Res| > 0 & ∃ |G_k| < gk do
6      /*Tp = {tp_1, ··· , tp_r}, tp_i = (V_{G_i}, X^i), X^i ∈ Res*/
7      Tp = MaximumWeightMatchings(St[][]);
8      for each tp_i ∈ Tp do
9          Add X^i into the last subgroup of G_k ;
10         remove X^i from Res;
11         update the Vc of G_k;
12     call SeedSearch();
13 return: G;
```

that G2 requires an additional crossbar or crossbar group. Line 2 finds the **Sc** of $\{X2, X5\}$ is larger than 6, so a new subgroup will be built for G2. Line 3 is to make sure the current choice is at least locally optimal. Line 4 updates Res, which is the set of crossbars that have not been grouped.

B. Heuristic Redundancy Algorithm: NuXG

Given a set of M group specifications $\{(g1, r1), (g2, r2), \cdots, (gM, rM)\}$ and a collection of physical crossbars X = $\{X1, X2, \cdots, XN\}$, find a subset of X with minimum cardinality (minimum number of physical crossbars) and construct a set of virtual crossbars using the physical ones in this subset to meet the M group specifications. Each physical crossbar in this subset will be used in exactly one constructed virtual crossbar.

Algorithm 2 takes in the required # for the **Vc** and the minimum **Sc** of the **Vc** in each group. Given a set of crossbar matrices $X = \{X1, X2, ..., XN\}$, algorithm 2 first sorts them according to their capacities in descending order in line 2. Then, line 3 and 4 call **SeedSearch()** to initialize the score table and virtual crossbars for each group. Lines 5∼12 are the main part of the proposed algorithm, which terminates when no residual crossbars are left or each group of G meets the corresponding required #. Line 6 calls **MaximumWeightMatchings()** to find the local optimal grouping strategy according to the current score table. Lines 8∼11 update the set of remaining crossbars and current grouping results G. Line 11 updates the **Vc** for each group. The final step of each iteration is to update the score table St in line 12 (inside SeedSeach()).

TABLE I: ATT Configuration

ATT		#	Area(mm^2)	Power(W)
Q-K-V Engine	Crossbar	3456	0.0864	1.0368
	ADC	3456	4.1472	6.912
	DAC	3456	0.07344	1.728
Atten. Engine	MM engine (C_L1_F)	2	1.527	0.0006
	Softmax	2	0.388	0.00068
Head Merge	Crossbar	864	0.0216	0.2592
	ADC	864	1.0368	1.728
	DAC	864	0.01836	0.432
L.N.	LayerNorm	2	0.0065	0.0124
Fully Connected Engine	Crossbar	6912	0.1728	2.0736
	ADC	6912	8.2944	13.824
	DAC	6912	0.14688	3.456
	gelu	2	0.015	0.0001
Mask Cache (8k)		1	0.0074	0.011
Buffer(64k)		8	0.835	0.249
eDRAM Bus		1	0.09	0.007
External I/O Interface		1	15.7	0.013
Total			32.57	31.74

TABLE II: Benchmark Configurations

Benchmarks							
	b_s	d_m	d_i	d_k	n_h	n_l	s_l
Transformer [6]	64	512	2048	64	8	6	20∼40
Bert(base) [7]	8	768	3072	64	12	12	128
XLNet [8]	8	768	3072	64	12	12	128
XLM [9]	8	1024	4096	128	8	6	128

V. METHODOLOGY

Power and Area Models. We use CACTI 7.0 [19] at 32 nm to model the power and area of the SRAM buffer and the Mask Cache. The area and power for memristor-based crossbars are adapted from ISAAC. The area and power of DAC and ADC units are modeled from the analysis in [21]. One 1-bit DAC is used for each word line of the 128×128 crossbar. The power and area of the Shift&Add unit are so small that we do not list them in Table I. For the power and area of the MM units, Softmax, and LayerNorm Engines, we scaled the results of existing literature ([22] for MM unit and [23] for a 16-bit truncated fixed-point multiplier and an adder respectively) to 32nm according to a recently proposed model [24]. We employ a Hyper Transport serial link model as off-chip links. Roughly, the area and power breakdown of the proposed accelerator is listed in Table I. The time consumption of GPU implementations is measured using NVVP.

Benchmarks. The benchmarks selected from SOTA works are shown in Table II. Note that all the parameters (except s_l of Transformer) listed in the table are independent from data sets. In the table, b_s, d_m, d_i, d_k, n_h, n_l and s_l denote the batch size, the embedding size, the inner layer size of fully connected layers, the q,k,v vector length, the number of heads, the number of layers and the sequence length respectively. The data set for Transformer is the WMT'16 Multi modal Translation Task [25]. Transformer is implemented as an open source tool using PyTorch [26]. Other baselines come from an open source repository implemented using PyTorch: PyTorch-Transformers-1.1.0 [27]. The data set for Bert, XLNet and XLM is MRPC from GLUE data [28].

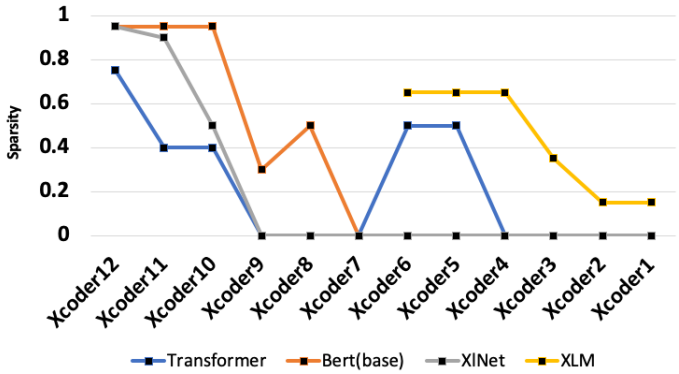


Fig. 5: Sparsity across layers with inference accuracy less than 2%

Performance Models. We developed an in-house simulator based on data from NVSim [29] to model the forward propagation process of attention-based neural networks. The cycle time in the proposed accelerator is 50.88ns, which is consistent with many existing PIM accelerators developed from NVSim. We compare the simulated performance result with the same neural networks running on a real GPU. The GPU platform is an NVIDIA GTX 1080 Ti, and we use CUDA 10.1 to compile the benchmarks. For energy saving, the metric we use is power efficiency (PE, the number of 16-bit operations performed per watt, $GOPs/W$).

Reference Schemes. To verify the efficiency of the proposed algorithm, we use two schemes as the reference: Ideal and RX [14]. Ideal refers to crossbars without any faulty cells, and it can be considered as the theoretical ideal case. The RX scheme is selected for the reference due to the minimal hardware overhead over other schemes in the paper. The system-level redundancy ratio for the RX is 3 under 20% SAFs because this configuration brings about the minimal classification error.

VI. Evaluation Results

A. Benchmark Profiling

We first profile the benchmarks of SOTA AttNNs in the community using an NVIDIA GTX 1080 Ti GPU to understand the performance bottlenecks. qkv, Heads Merge and FC (Fully connected layers, FC1 and FC2 in Fig.1) account for 81.5%, 75.79%, 74.2% and 83.69% of the computation time for Transformer [6], Bert [7], XLNet [8] and XLM [9] respectively. These modules are absolutely the bottlenecks and need to be focused on. By contrast, the attn consumes at most 20% for XLNet and around 10% for other benchmarks.

TABLE III: Layer Grouping Configurations

	High(<0.35)	Medium(0.35,0.5)	Low(>0.5)
Transformer	1∼4,7∼9	5,6,10,11	12
Bert_base	1∼7,9	8	11∼13
XLNet	1∼9	10	11,12
XLM	1,2	3	4∼6

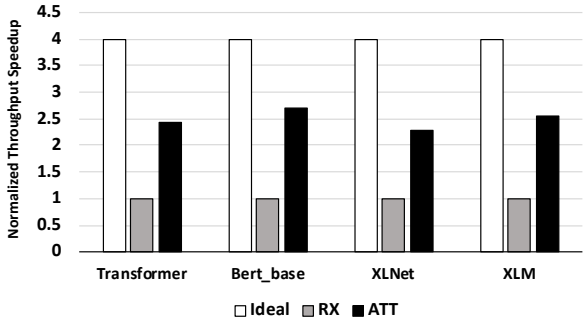


Fig. 6: Throughput improvement

Fig. 5 shows weight sparsity across layers. The horizontal axis denotes the layers. The vertical axis represents the percentage of pruned weights due to sparsity. From Fig.5, we can observe that the layers close to the inputs are less sparse than the remaining layers. For bert and XLNet, the sparsity of the last few layers is nearly 100%. The largest accuracy loss is 1.9% which comes from Transformer.

B. Xbar Grouping Benefits

We should note that the layer sparsity configuration of the four benchmarks in Fig.5 may be not the optimal configuration, but layer-wise sparsity of benchmarks generally exists. Finding the optimal sparsity configuration is not the focus of this work.

According to Fig.5, we group the layers as shown in Table III. We use 0.35 and 0.5 as sparsity thresholds to divide the groups. Different split points impact the following crossbar grouping results. The class describes how high or low the redundancy requirements are for a given group of layers. The storage capacity we use is 0.9,0.95 and 0.99 for the Low, Medium and High groups respectively. For example, 0.9 means that we can meet the accuracy requirements in the Low group only if at least 90% of cells are fault free. As expected, the redundancy ratio for the Low group will be smaller than that of the other two groups.

To provide a fair comparison, we set the number of crossbars for the Ideal and ATT schemes to be equal to the RX. The performance improvement showed in Fig.6 verifies the benefits of the proposed algorithm, and as such only reflect

the results from bottlenecks. The throughput for RX is only a quarter of that of the Ideal. For the fixed crossbars, the RX scheme can only store one copy of weight matrices, while the ideal scheme has four times the storage capacity for all benchmarks. Compared to RX, ATT can store around 2.5 times the weight matrices. Therefore, ATT boosts the throughput by about 2.5 times over RX for all benchmarks. We see the least improvement with XLNet because its Q-K-V engine must store an additional weight matrix that the others do not.

For the power efficiency improvement, the values for the Ideal case are theoretical maximums. The RX achieves only 20%∼25% of the Ideal values. In contrast, ATT largely improves the performance and reaches around 60% of the theoretical values for all benchmarks.

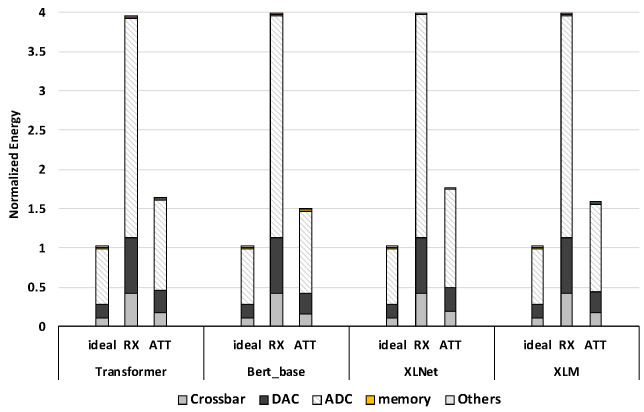


Fig. 7: Energy breakdown

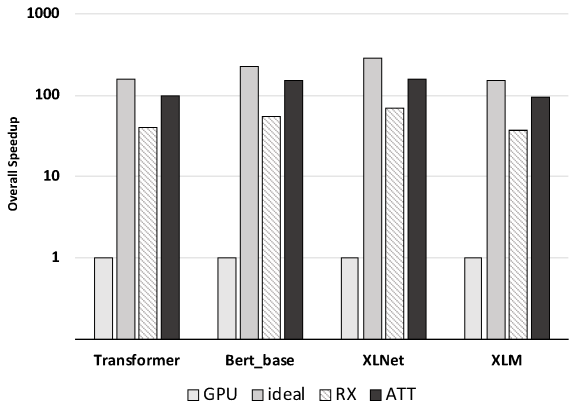


Fig. 8: Overall speedup

C. Overall Performance Improvement

Fig.7 compares energy consumption including a breakdown into components. Although ADCs and DACs are still high-energy-consuming components for all three schemes, we find the total energy of ATT is significantly reduced compared to RX. This is because our proposed algorithm decreases the redundancy ratio, thus reducing the number of ADCs and DACs. Overall, ATT consumes just around 1.5 times more energy than the ideal case to achieve almost the same inference accuracy for all benchmarks.

Fig. 8 provides a comparison of the speedup of both ATT and GPU implementations to both baselines. In general, crossbar based accelerators perform tens of times faster than GPU. ATT achieves better performance than the RX, but still worse than the Ideal. The average speedup of the Ideal with respect to the GPU is 202.78×, whereas the RX can only achieve 50.69×. This is because more than half of the crossbars are used for redundancy purposes. The average speedup of ATT reaches 125.28×. XLM gets the smallest speedup because it has the largest q,k,v vector size, which imposes pressure on the Attention and LayerNorm engines.

VII. Related Work

Hardware accelerator for AttNNs has been recently studied by researchers [11]. However, they only concentrate on the attention module that doesn't dominate the inference time. Additionally, their proposed cannot adapt to the latest AttNNs, such as XLNet, GPT and XLM. One of the reasons is that the gelu function is not supported by the aforementioned work.

There are previous works [1]–[3], [30] developing traditional neural networks including CNNs and RNNs. The unique characteristics of AttNNs make these accelerators cannot tailored to AttNNs. Among these works, Processing-In-Memory architectures [1]–[3] achieve better performance than FPGA and ASIC.

Fault tolerance has been intensively investigated for soft errors [31]–[35] and hard errors [36]–[38] in the field of conventional processors. On the other hand, there are only a few related studies on ReRAM accelerators. Chen et al. [12] first propose Stuck-At-Fault issues and studies their characteristics (percentage and distribution). Xia et al. [14] imply that hard faults can only be solved by redundancy in section II.B. Meanwhile, Zhang et al. [13], [41] also use similar methods to handle this issue. Seong. [40] focus on the SAFs of PCM, which has a different memristor cell structure from ReRAM. Li et al. [39] tackle soft faults in ReRAM.

VIII. Conclusion

PIM accelerators for deep learning regardless of hard faults result in significant accuracy loss. Hardware redundancy strategies that are unaware of application characteristics usually lead to inefficient architectures. The observation that weight sparsity varies with layers offers us an opportunity to design a heuristic redundancy algorithm to deploy crossbars. The proposed algorithm can be adapted to PIM accelerators for CNNs. In this work, we apply this algorithm to an accelerator for AttNNs, achieving more than 60% of the theoretical peak performance and more than two times as much throughput as a SOTA redundancy work. Additionally, an intra-layer pipeline is designed for the accelerator.

Acknowledgement

We appreciate the insightful feedback and suggestions from anonymous reviewers who help us finalize the paper. This

work is supported in part by NSF Grants 1422408, 1527318, 1946626, and 2020446. Travis LeCompte is supported by a Louisiana Board of Regent Fellowship.

REFERENCES

[1] P. Chi, et al. "PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory," ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), pp. 27-39, 2016.

[2] A. Shafiee, et al. "ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars," ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), pp. 14-26, 2016.

[3] Hao Yan, et al. "CELIA: A Device and Architecture Co-Design Framework for STT-MRAM-Based Deep Learning Acceleration," In Proceedings of the 2018 International Conference on Supercomputing (ICS '18), pp. 149–159. 2018.

[4] Haoqiang Guo, et al. "Hardware Accelerator for Adversarial Attacks on Deep Learning Neural Networks," In 2019 Tenth International Green and Sustainable Computing Conference (IGSC), pp. 1-8, 2019.

[5] Haoqiang Guo, et al. "Fooling AI with AI: An Accelerator for Adversarial Attacks on Deep Learning Visual Classification," In 2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP),Vol. 2160, pp. 136-136, 2019 Jul 15.

[6] Vaswani, Ashish, et al. "Attention is all you need," NeurIPS, 2017, pp. 5998-6008.

[7] Devlin J. et al. "Bert: Pre-training of deep bidirectional transformers for language understanding," arXiv preprint arXiv:1810.04805. 2018 Oct 11.

[8] Z. Yang, et al. "Xlnet: Generalized auto regressive pretraining for language understanding," In Advances in neural information processing systems (NeurIPS), pp. 5754–5764, 2019.

[9] G. Lample and A. Conneau, "Cross-lingual language model pretraining," In Advances in neural information processing systems (NeurIPS), pp. 7057-7067, 2019.

[10] Hendrycks, Dan, and Kevin Gimpel. "Gaussian error linear units (gelus)," arXiv:1606.08415, 2016.

[11] Ham, Tae Jun, et al. "A^3: Accelerating Attention Mechanisms in Neural Networks with Approximation," In 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 328-341, 2020.

[12] C.Y. Chen, 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, 2014.

[13] B. Zhang, et al. "Handling stuck-at-faults in memristor crossbar arrays using matrix transformations," In Proceedings of the 24th Asia and South Pacific Design Automation Conference (ASP-DAC '19), pp. 438–443, 2019.

[14] L. Xia, 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.

[15] S. N. Truong and K.-S. Min, "New memristor-based crossbar array architecture with 50-% area reduction and 48-% power saving for matrix-vector multiplication of analog neuromorphic computing," JSTS: Journal of Semiconductor Technology and Science, vol. 14,no. 3, pp. 356–363, 2014.

[16] P. Nilsson, et al. "Hardware implementation of the exponential function using taylor series," in 2014 NORCHIP, IEEE, pp. 1–4, 2014.

[17] G. Du, et al. "Efficient softmax hardware architecture for deep neural networks," GLVLSI, In Proceedings of the 2019 on Great Lakes Symposium on VLSI (GLSVLSI'19), pp. 75–80, 2019.

[18] H. W. Kuhn, "Variants of the hungarian method for assignment problems," Naval Research Logistics Quarterly, 1956, vol. 3, no. 4, pp.253–258.

[19] R. Balasubramonian, et al. "Cacti 7: New tools for interconnect exploration in innovative off-chip memories," ACM Trans. Archit. Code Optim.(TACO), vol. 14, no. 2, pp. 1–25, 2017.

[20] L. Kull, et al. "A 3.1 mw 8b1.2 gs/s single-channel asynchronous sar adc with alternate comparators for enhanced speed in 32 nm digital soi cmos," IEEE Journal of Solid-State Circuits, vol. 48, no. 12, pp. 3049–3058, 2013.

[21] M. Saberi, et al. "Analysis of power consumption and linearity in capacitive digital-to-analog converters used in successive approximation adcs," IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 58, no.8, pp.1736–1748, 2011.

[22] R. P. Rajput and M. S. Swamy, "Super scalar pipelined inner product computation unit for signed unsigned number," Perspectives in Science, vol. 8, pp. 606–610, 2016.

[23] M. Aguirre-Hernandez and M. Linares-Aranda, "Cmos full-adders for energy-efficient arithmetic applications," IEEE transactions on VLSI systems, vol. 19, no. 4, pp. 718–721, 2010.

[24] A. Stillmaker and B. Baas, "Scaling equations for the accurate prediction of cmos device performance from 180 nm to 7 nm," Integration, vol. 58, pp. 74–81, 2017.

[25] D. Elliott, et al. "Multi30k:Multilingual english-german image descriptions," 2016. pp. 70–74.

[26] https://github.com/jadore801120/attention-is-all-you-need-pytorch

[27] https://github.com/huggingface/transformers

[28] A. Wang, et al. "Glue: A multi-task benchmark and analysis platform for natural language understanding," arXiv:1804.07461, 2018.

[29] X. Dong, et al. "Nvsim: A circuit-level performance, energy, and area model for emerging non volatile memory," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 31, no. 7, pp. 994–1007, 2012.

[30] S. Angizi, et al. "MRIMA: An MRAM-Based In-Memory Accelerator," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 39, no. 5, pp. 1123-1136, May 2020, doi: 10.1109/TCAD.2019.2907886.

[31] L. Duan, et al. "Versatile Prediction and Fast Estimation of Architectural Vulnerability Factor from Processor Performance Metrics," In Proceedings of the 15th IEEE International Symposium on High-Performance Computer Architecture (HPCA-15), Raleigh, NC, Feb. 2009.

[32] L. Duan, et al. "Universal Rules Guided Design Parameter Selection for Soft Error Resilient Processors," In Proceedings of The 2011 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Austin, TX, Apr. 2011.

[33] T. LeCompte, et al. "Soft Error Resilience of Big Data Kernels through Algorithmic Approaches," Springer Journal of Supercomputing, Vol. 73, pp. 4739–4772, Nov. 2017.

[34] L. Duan, et al. "Comprehensive and Efficient Design Parameter Selection for Soft Error Resilient Processors via Universal Rules," In IEEE Transactions on Computers, Volume 63, Issue 9, pages 2201 – 2214, Sep. 2014.

[35] L. Duan, et al. "Predicting Architectural Vulnerability on Multi-Threaded Processors under Resource Contention and Sharing," In IEEE Transactions on Dependable and Secure Computing, Vol. 10(2), pages 114-127, Mar.-Apr. 2013.

[36] Y. Zhang, et al. "Optimal Microarchitectural Design Configuration Selection for Processor Hard-Error Reliability," In Proceedings of The 13th IEEE International Symposium on Quality Electronic Design (ISQED), Santa Clara, CA, Mar. 2012.

[37] Y. Zhang, et al. "Design Configuration Selection for Hard-error Reliable Processors via Statistical Rules", In Journal of Microprocessors and Microsystems, Volume 38, Issue 1, pages 22–30, Feb. 2014.

[38] M. Alwadi,et al. "Phoenix: Towards Ultra-Low Overhead, Recoverable, and Persistently Secure NVM," in IEEE Transactions on Dependable and Secure Computing, doi: 10.1109/TDSC.2020.3020085.

[39] W. Li, et al. "RRAMedy: Protecting ReRAM-Based Neural Network from Permanent and Soft Faults During Its Lifetime," 2019 IEEE 37th International Conference on Computer Design (ICCD), pp. 91-99, 2019.

[40] N. H. Seong, et al. "Safer: Stuck-at-fault error recovery for memories," in Proceeding of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 115–124, 2010.

[41] B. Zhang, et al. "Handling stuck-at-fault defects using matrix transformation for robust inference of dnns,"IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TACD), 2019.