Value-aware Parity Insertion ECC for Fault-tolerant Deep Neural Network

Seo-Seok Lee
Sungkyunkwan University, Suwon, Korea
System LSI Division, Samsung Electronics, Korea

Joon-Sung Yang
School of Electrical and Electronic Engineering,
and Department of System Semiconductor Engineering
Yonsei University, Seoul, Korea
js.yang@yonsei.ac.kr

***Abstract*—Deep neural networks (DNNs) are deployed on hardware devices and are widely used in various fields to perform inference from inputs. Unfortunately, hardware devices can become unreliable by incidents such as unintended process, voltage and temperature variations, and this can introduce the occurrence of erroneous weights. Prior study reports that the erroneous weights can cause a significant accuracy degradation. In safety-critical applications such as autonomous driving, it can bring catastrophic results. Retraining or fine-tuning can be used to adjust corrupted weights to prevent the accuracy degradation. However, training-based approaches would incur a significant computational overhead due to a massive size of training datasets and intensive training operations. Thus, this paper proposes a value-aware parity insertion error correction code (ECC) to recover erroneous weights with a reduced parity storage overhead and no additional training processes. Previous ECC-based reliability improvement methods, *Weight Nulling* and *In-place Zero-space ECC*, are compared with the proposed method. Experimental results demonstrate that DNNs with the value-aware parity insertion ECC can perform inference without the accuracy degradation, on average, in $122.5\times$ and $15.1\times$ higher bit error rate conditions over *Weight Nulling* and *In-place Zero-space ECC*, respectively.**

***Index Terms*—Value-aware Parity Insertion ECC, Deep Neural Network, Fault-tolerance, Error Correction Code**

I. Introduction

The state-of-the-art deep neural networks (DNNs) are widely adopted in various fields such as image classification and autonomous driving. DNN models are deployed on hardware devices and perform inference from inputs by using trained weight parameters [1]–[3]. Unfortunately, semiconductor hardware devices could be unreliable due to unintended process, voltage and temperature variations. [4] reports that a fault rate could become 10^{-4} in process technology below 20nm. In addition, memory reliability [5]–[8] can directly influence the reliability of DNN. Especially, for safety-critical applications such as autonomous driving, there would be catastrophic sequences if erroneous weights degrade the classification accuracy. Therefore, it is essential to enhance the reliability of DNN by preserving weights from errors.

When DNN performs inference, activations are calculated by weights and inputs. Thus, both activations and weights can affect the reliability of DNN. [9] reports that erroneous weights can cause a significant accuracy degradation and the weights are less tolerant to faults than activations. While the activations are continuously changed by new inputs, the weights are reused many times. In this paper, we focus on correcting corrupted weights.

There has been some research done to prevent an accuracy degradation of DNN. One approach is to exploit retraining or fine-tuning for updating erroneous weights [10]–[12]. However, due to a massive size of training datasets and computationally intensive training operations, these methods would introduce significant overheads. For instance, the size of ILSVRC-2012 [13] training dataset is over 100GB. Furthermore, retraining and fine-tuning require a considerable time for iterative computations to adjust weight values.

Another approach is to employ error correction code (ECC) [14] to recover erroneous weights of DNN. ECCs correct bit error(s) by redundant information bits called parity bits (also referred as check bits). *Weight Nulling* [15] and *In-place Zero-space ECC* [16] are introduced to recover corrupted weights of DNN by ECC. These methods apply ECC with parity insertion techniques because the state-of-the-art DNN models can incur a significant parity storage overhead by millions of weights. *Weight Nulling* uses a last bit of each weight as a single parity bit and replaces erroneous weight values with zero when the parity bit detects the odd number of bit errors in each weight. However, as the last bit of each weight is just replaced as a parity bit without considering the influence on weight values, *Weight Nulling* may experience the classification accuracy degradation of DNN models. Furthermore, *Weight Nulling* is not able to perform error correction. *In-place Zero-space ECC* makes weight values reside in a certain value range by modified training and inserts parity bits into weights. A single bit error for eight 8-bit weights can be corrected by the inserted parity bits. *In-place Zero-space ECC* might encounter a computational overhead before DNN deployment due to additional modified training. In addition, it has a limited error correction capability of single error correction (SEC).

This paper proposes a value-aware parity insertion ECC to make weights of DNN models more fault-tolerant to perform inference in unintended higher bit error conditions. The proposed ECC conducts double error correction (DEC) for 8-bit quantized weights with a negligible storage overhead and without additional training. The major contributions of this paper are summarized as follows:

- This paper introduces a value-aware parity insertion ECC for DNN to enhance the error correction capability from

978-3-9819263-6-1/DATE22/©2022 EDAA

single to double error correction with a reduced parity storage overhead and no additional training processes.

- To mitigate the parity storage overhead, this paper proposes a novel parity insertion technique to replace less important data bits of the 8-bit quantized weights with ECC parity bits by weight value comparison.
- The experiments report that DNN models with the value-aware parity insertion ECC can perform inference in higher bit error rate conditions by up to $350\times$ and $40\times$ than *Weight Nulling* and *In-place Zero-space ECC*, respectively.

Rest of the paper is organized as follows. Section II introduces related works. Section III gives an observation and Section IV describes the value-aware parity insertion ECC in detail. The experimental results are given in Section V and Section VI concludes the paper.

II. Related Works

This paper aims for an ECC-based approach to recover erroneous weights of DNN. Recent related works, *Weight Nulling* [15] and *In-place Zero-space ECC* [16], exploit ECC to prevent an accuracy degradation from corrupted weights with parity insertion techniques. Details of *Weight Nulling* and *In-place Zero-space ECC* are explained in the following subsections.

A. Weight Nulling

Weight Nulling [15] uses the last bit of each weight as a single parity bit. For 16-bit weights ($b_{15} \sim b_0$), the higher-order 15 bits ($b_{15} \sim b_1$) are used to represent weight values and the last bit (b_0) is replaced with a parity bit. For example, when even parity is used, a parity bit is generated and appended to the 15 bits, and this constructs the 16-bit weight. Assume that a single bit error occurs on the weight. The single bit error in the weight would make the modulo 2 of the sum of 16 bits be non-zero due to the odd number of '1's. Then, the corrupted weight value is set to 0 after.

Weight Nulling can enhance the DNN reliability by the erroneous weight zeroing technique. However, *Weight Nulling* is not able to perform error correction, but detect the odd number of bit error occurrence for each weight. While corrupted weights with the odd number of bit error are detected, the erroneous bits cannot be recovered. Furthermore, this causes information loss because the erroneous weights are replaced to zeros. Thus, the reliability enhancement of *Weight Nulling* could be limited. To overcome this limitation by the absence of error correction capability and the information loss, the proposed method exploits ECC to correct bit errors of weights.

B. In-place Zero-space ECC

In-place Zero-space ECC [16] enhances the fault-tolerance of DNN with a modified training and single error correction-double error detection (SEC-DED)-ECC. The modified training performs quantization-aware training (QAT) and throttling to set the weight values in a specific value range. QAT performs inference with quantized weight parameters and calculates 32-bit floating point gradients. As updating the quantized weights by the 32-bit gradients, 32-bit floating point weights are generated. Then, the 32-bit weights are converted into new quantized weights. After QAT, throttling intentionally sets the value range of 8-bit weights to $[-64, 63]$, even though 8-bit weights can represent the weight values between $[-128, 127]$. When iterations of QAT and throttling are done, 8-bit weights are in $[-64, 63]$. Thus, the second higher-order bit (b_6) is always the same as the sign bit (b_7) (i.e., MSB). *In-place Zero-space ECC* utilizes b_6 as a parity bit and restores it by the sign bit.

In-place Zero-space ECC exploits a SEC-DED (64, 57) code for a 64-bit data block including eight 8-bit weights. SEC-DED (64, 57) code uses 7 parity bits to correct a single bit error and to detect double bit errors. For seven out of eight weights in 64-bit codeword, the second higher-order bits (b_6) are replaced with the parities. After error correction, original weight values are easily restored by copying the sign bit (b_7) to b_6 in seven weights.

To set 8-bit weights in a specific value range, *In-place Zero-space ECC* requires a large number of modified training iterations for DNN models before deployment. It could incur a significant overheads due to the size of training datasets and intensive computing operations. In addition, *In-place Zero-space ECC* cannot handle double bit errors for a 64-dit data block which consists of eight 8-bit weights. Therefore, to overcome these concerns about additional modified training and limited correction capability, this paper proposes a value-aware parity insertion ECC to improve the correction capability from single error to double error correction without additional training process.

III. Observation for Parity Insertion Technique

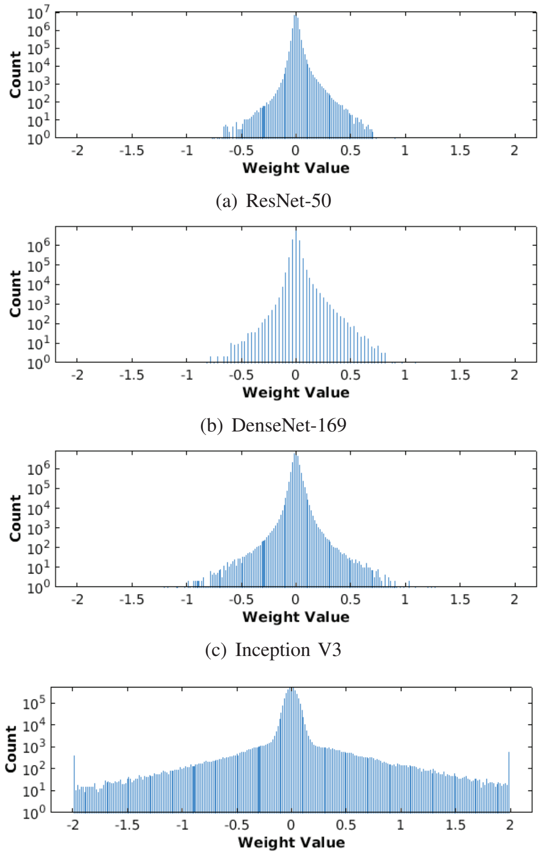
ECC-based fault-tolerant methods could introduce a parity storage overhead due to millions of weight parameters in recent DNN models. To reduce the parity overhead, we propose a novel parity insertion technique based on the analysis on weight value distributions and a value representation format, sign-magnitude representation. The observation for the parity insertion technique is described in Section III-A and III-B.

A. Weight Value Distributions of Various DNN models

Figure 1 presents distributions of weight values for different DNN models. The weight values are near-symmetrically distributed around zero, showing a similar number of positive and negative weights. In addition, most of weight values are found between -0.5 and 0.5. This paper exploits the fact that, as the most weights reside in a certain range, there would be a possible flexibility of utilizing few bit fields in weights for parity bits.

As considering the magnitude of values, in binary value representation for 8-bit weight (2, 6)[1], the lower-order bits are used to represent values less than $|0.5|$. This means that the lower-order bits mainly represent weight values as the

[1] (x, y) denotes x integer and y fraction bits. The integer bit field includes a most significant bit (MSB) used to represent a sign bit.



(a) ResNet-50

(b) DenseNet-169

(c) Inception V3

(d) MobileNet V2

Fig. 1. Weight value distributions in various DNN models.

dominant values are found near zero. Compared to the lower-order bits, there would be some higher-order bits which do not contribute to representing weight values because the higher-order bits are rarely used for value representation when there are not many weight values greater than 0.5 or less than -0.5.

B. Sign-magnitude Representation

In computing system, two's complement is the most commonly used binary representation format due to simple handling of negative values. However, considering the symmetric weight value distributions in DNN models, this paper pays attention to sign-magnitude representation. In sign-magnitude representation, binary expressions of the positive and negative values are the same except for the sign bit when their absolute values are equal. Table I shows binary expression examples by two's complement and sign-magnitude representation. For the same absolute values, sign-magnitude expressions are the same except for the sign bit while two's complement gives different binary expressions for them.

Table II gives examples of how the positive and negative values are represented by sign-magnitude format for 8-bit weight (2, 6). As can be seen, in sign-magnitude expression, the positive and negative values have the same binary values for $b_6 \sim b_0$ except for b_7 which is the sign bit. With symmetric weight value distributions in Figure 1, sign-magnitude representation would be an attractive option to deal with both positive and negative values. In addition, if any absolute value of weight ($|W_{value}|$) is less than 0.5, b_6b_5 is always guaranteed to be 0 in sign-magnitude representation. This tells that, for weights with absolute values less than 0.5, b_6b_5 can be used to store parity information because the original weight value can be easily rebuilt by overwriting the parity bits at b_6b_5 with 0s. It should be noted that the sign-magnitude format is only considered to represent weight values not to perform arithmetic operations. Sign-magnitude representation is converted to two's complement for easier arithmetic operations before inference starts.

TABLE I
8-BIT BINARY EXPRESSIONS (2, 6) FOR TWO'S COMPLEMENT AND SIGN-MAGNITUDE BINARY FORMAT ($b_7\_b_6b_5\_b_4b_3b_2b_1b_0$).

Value	Two's complement	Sign-magnitude
0.125	0_00_01000	0_00_01000
-0.125	1_11_11000	1_00_01000
0.5625	0_01_00100	0_01_00100
-0.5625	1_10_11100	1_01_00100

TABLE II
SIGN-MAGNITUDE BINARY EXPRESSIONS FOR DIFFERENT VALUES OF 8-BIT WEIGHT (2, 6).

Value		Sign-magnitude Binary Expression ($b_7\_b_6b_5\_b_4b_3b_2b_1b_0$)
0.0625	(-0.0625)	0_*00*_00100 (1_*00*_00100)
0.125	(-0.125)	0_*00*_01000 (1_*00*_01000)
0.25	(-0.25)	0_*00*_10000 (1_*00*_10000)
0.5	(-0.5)	0_*01*_00000 (1_*01*_00000)
0.5625	(-0.5625)	0_*01*_00100 (1_*01*_00100)

IV. VALUE-AWARE PARITY INSERTION ECC

The proposed value-aware parity insertion ECC employs a DEC (64, 50) code to achieve an error correction capability of double error correction (while *In-place Zero-space ECC* [16] is SEC-DED) with a reduced parity storage overhead and no additional training requirements. The proposed ECC consists of two processes, encoding and decoding. In encoding process, parity bits are generated and inserted into weights by weight value comparison. Before inference starts, decoding process performs error correction and zero masking.

A. Encoding Process for 64-bit Data Block with Eight 8-bit Weights

In this work, we assume 8-bit weight quantized networks. The DEC (64, 50) code corrects up to double bit errors for a 64-bit data block by using 14 parity bits. Encoding process generates 14 parity bits of the DEC (64, 50) code and stores the parity bits in eight weights with sign-magnitude representation. This paper uses seven weights among eight weights to store 2-bit each for the 14 parity bits because it preserves one weight without being overwritten by the parity.

14 parity locations are decided by weight value comparison. Based on Section III, weight value comparison compares absolute values of weights with *threshold value* to store parity bits in higher-order bits (b_6b_5) of weights. This paper defines *threshold value* to secure the parity locations. We set *threshold value* as 0.5 for parity insertion as considering 8-bit weight (2,

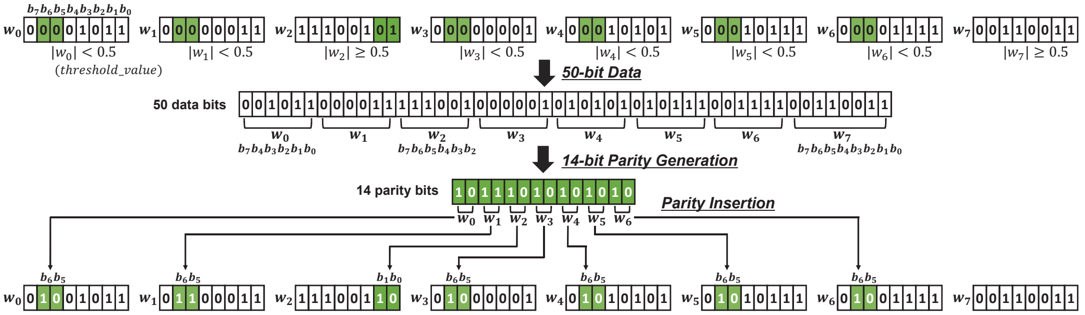


Fig. 2. Example of encoding process of the value-aware parity insertion ECC. Eight weights are expressed by sign-magnitude representation.

6)[2]. If one weight has an absolute value smaller than *threshold value*, higher-order bit locations ($b_6 b_5$) are overwritten by parity bits. In constrast, for weights larger than or equal to *threshold value*, because the higher-order bits have a more contribution in representing larger values than lower-order bits ($b_1 b_0$), the proposed method stores parity bits at the lower-order bits in weights. However, the original weight value may not be fully restored because the lower-order bits are not always guaranteed to be 0. This will be further discussed in Section V-C.

Figure 2 shows an example of encoding process in the value-aware parity insertion ECC. First, to form 64-bit data block, eight weights ($w_0 \sim w_7$) with sign-magnitude representation are concatenated. Then, for (64, 50) DEC code, seven weights are determined to store 14 parity bits by weight value comparison. To minimize the information loss by parity insertion, weights smaller than *threshold value* are preferentially chosen because they can be fully restored during decoding. This finds six weights (w_0, w_1, w_3, w_5, w_5, w_6) smaller than *threshold value*. Each can store two parity bits at b_6 and b_5, and the locations of 12 parity bits out of 14 for (64, 50) DEC code are determined. To store 2 more parity storage locations in weights, w_2 or w_7 needs to be chosen. In this example, the proposed method selects w_2 to keep the parity inserted weight value as close as its original value. As addressed, the weight whose absolute value is larger than *threshold value* uses b_1 and b_0 to store the parity information, and it cannot be guaranteed to restore its original weight value after decoding (as the decoding overwrites $b_1 b_0$ with *00*). Hence, to minimize the weight value difference, w_2 is selected to store 2 parity bits (if w_7, 001100_*11*, is chosen, the decoded w_7 becomes 001100_*00* while w_2, 111001_*01* would change to 111001_*00* after decoding). From 64-bit eight weights, 50-bit data is found only extracting bits not used to store parity bits. The 14-bit parity information is generated using the 50-bit data and this forms (64, 50) DEC code. Then, the generated parity bits are inserted into each weight. This shows how the proposed value-aware parity insertion ECC operates to minimize the parity storage overhead.

TABLE III

RATIO OF CODEWORD FORMAT AND STORAGE RATIO OF WEIGHT AND SPARSE MATRIX IN DNN MODELS WHEN *threshold value* IS 0.5.

Model	Ratio of codeword format (**HHHH_HHHN**)	Storage ratio of weight	Storage ratio of sparse matrix
ResNet-50	99.9%	99.998%	0.002%
DenseNet-169	99.9%	99.994%	0.006%
Inception V3	99.9%	99.995%	0.005%
MobileNet V2	98.4%	99.923%	0.077%

B. Sparse Matrix for Parity Identification

After encoding process, it is essential to mark the information on parity locations to perform error correction in decoding process because the parity bits are placed at different bit fields ($b_6 b_5$ or $b_1 b_0$). In 64-bit data blocks, seven weights are used to store parity bits in the higher-order or lower-order field according to the comparison result between the absolute values of weights and *threshold value*.

There are possibly 1024 formats of parity locations ($1024 = \binom{8}{1} \times 2^7$) in one 64-bit data block. If there are various codeword formats, the information for parity identification could require a large storage of additional flag bits to indicate each codeword format. However, as shown in Figure 1, the weight values in [-0.5, 0.5] are dominant. Thus, parity bits might be mostly stored in the higher-order bit field ($b_6 b_5$) because weights are mostly less than *threshold value*.

A ratio of codeword formats is evaluated. **L**, **H** and **N** denote for the lower-order parity, the higher-order parity, and no parity information in a weight, respectively. Table III indicates that most of codewords in DNN models have the codeword format of **HHHH_HHHN**. Thus, we exploit sparse matrix to record parity locations in each data block after parity generation and insertion are done. The sparse matrix only stores information for which codewords are not **HHHH_HHHN**. Table III shows the ratio of storage volume between weight and sparse matrix. In all DNN models, less than 0.1% of storage is occupied by the sparse matrix when each element of the sparse matrix is 10-bit size due to 1024 formats of parity locations. Therefore, a storage overhead of the sparse matrix is negligible.

C. Decoding Process for Error Correction and Zero-masking

Decoding process of the proposed ECC performs error correction and zero-masking before inference starts. Figure 3 shows a decoding process of the proposed ECC. Assume that

[2] It should be noted that *threshold value* can be determined by a weight value distribution of DNN model. For instance, when weights represent a range of $[-4, 4)$ with 8-bit (3, 5), *threshold value* is set as 1 because $b_6 b_5$ of weights less than $|1|$ is *00*.

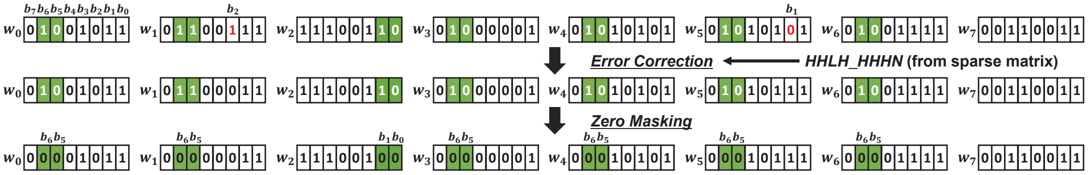

$b_7b_6b_5b_4b_3b_2b_1b_0$
b_2
b_1
w_0 01001011 w_1 01100111 w_2 11100110 w_3 01000001 w_4 01010101 w_5 01010101 w_6 01001111 w_7 00110011
Error Correction
HHLH_HHHN (from sparse matrix)
w_0 01001011 w_1 01100011 w_2 11100110 w_3 01000001 w_4 01010101 w_5 01010111 w_6 01001111 w_7 00110011
Zero Masking
b_6b_5
b_1b_0
w_0 00001011 w_1 00000011 w_2 11100100 w_3 00000001 w_4 00010101 w_5 00010111 w_6 00001111 w_7 00110011

Fig. 3. Example of decoding process of the value-aware parity insertion ECC.

there are two error occurrences at b_2 in w_1 and b_1 in w_5. To correct the errors, the locations of 14 bit parity bits are identified with the sparse matrix information (**HHLH_HHHN**). Then, the errors are corrected by the parity bits. After error correction, the parity bits in weights are no longer required, and the original weights need to be restored. The weights can be simply reconstructed by masking the parity bits with 0s. Figure 3 shows that the color-highlighted parities are masked with *0*s in zero-masking step. As can be seen, the weights smaller than *threshold value* are completely restored and the weights bigger than *threshold value* are closely restored to their original weights after decoding.

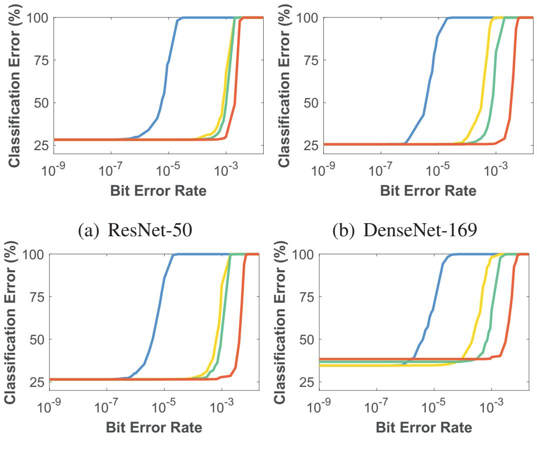
Sign-magnitude representation is only used to handle parity bits for mitigating a parity storage overhead. Thus, after zero masking, binary expressions are converted to two's complement for arithmetic operations of inference phase. It should be noted that, the sign-magnitude to two's complement conversion is only required for negative weights as positive ones are represented the same. After weights are converted to two's complement format, inference begins.

V. Experiment

The value-aware parity insertion ECC is evaluated with various DNN models (ResNet-50 [17], DenseNet-169 [18], Inception V3 [19] and MobileNet V2 [20]) which have different topologies with quantized 8-bit weights. All DNN models on validation dataset of ILSVRC-2012 [13] are implemented by Keras framework [21].

A. Random Fault-injection into DNN models

For fault-injection, this paper considers hard faults which affect hardware devices persistently. The hard faults (stuck-at-0 or stuck-at-1) are injected into weights of DNN models by sampling random uniform distribution before the DNN models start inference. After a fault-injection stage, the error correction is performed according to error correction schemes, *Weight Nulling*, *In-place Zero-space ECC*, and the value-aware parity insertion ECC. Various bit error rates (BERs) are considered and hundreds of iterations are run for each BER. The fault-tolerance is evaluated by measuring top-1 average accuracy of the DNN models.



(a) ResNet-50

(b) DenseNet-169

(c) Inception V3

(d) MobileNet V2

Fig. 4. Classification error results for various BERs in weights from models with No ECC (blue line), *Weight Nulling* (yellow line), *In-place Zero-space ECC* scheme (green line) and Value-aware Parity Insertion ECC (red line).

B. Fault-tolerance Analysis

Figure 4 shows the results of random bit error simulations for DNN models. The X-axis denotes BER from 10^{-9} to 10^{-2} and the Y-axis represents the corresponding average classification error. As shown, the classification error is maintained up to a certain BER and it starts to increase sharply as the BER further increases. In this paper, the BER making the model accuracy lower than the accuracy from error-free condition is referred as the critical BER. For *Weight Nulling* [15], *In-place Zero-space ECC* [16] and the proposed value-aware parity insertion ECC, Table IV shows that the critical BERs are found much higher than DNN models without ECC protection. The last column of Table IV indicates an improvement ratio of critical BERs for [15], [16] and the proposed ECC normalized by the critical BER of DNN model without ECC protection.

1) Comparison with Weight Nulling: Even though both *Weight Nulling* and the proposed ECC achieve a higher reliability for DNN models than the no ECC scheme, the improvement ratio of critical BER by the proposed ECC is $122.5\times$ higher than *Weight Nulling*, on average. The proposed ECC provides a DEC capability while *Weight Nulling* sets erroneous weights to zero. Furthermore, a replacement of corrupted weight value by 0s in *Weight Nulling* could incur more information loss. Hence, the proposed fault-tolerant DNN method is more effective in enhancing the reliability of DNN than *Weight Nulling*.

2) Comparison with In-place Zero-space ECC: As correction capability is improved to DEC, DNN models with the proposed ECC have a $15.1\times$ higher improvement ratio of critical BERs than the models with *In-place Zero-space ECC* , on average. In addition, the proposed method does not require any additional training process for error correction using a (64, 50) DEC code. In *In-place Zero-space ECC*, to employ a DEC (64, 50) code, it is necessary to apply strong throttling to weights to insert more parity bits into the weights. This would require more numbers of modified training iterations of *In-place Zero-space ECC* than SEC-DED (64, 57) code

TABLE IV
CRITICAL BERS OF VARIOUS DNN MODELS.

Model	Error Correction Scheme	Critical BER	Improvement Ratio
ResNet-50	No ECC	2×10^{-7}	
	Weight Nulling	2×10^{-5}	×100
	In-place Zero-space ECC	8×10^{-5}	×400
	Value-aware Parity Insertion ECC	3×10^{-4}	×1500
DenseNet-169	No ECC	2×10^{-7}	
	Weight Nulling	2×10^{-6}	×10
	In-place Zero-space ECC	1×10^{-5}	×50
	Value-aware Parity Insertion ECC	9×10^{-5}	×450
Inception V3	No ECC	2×10^{-7}	
	Weight Nulling	5×10^{-6}	×25
	In-place Zero-space ECC	1×10^{-5}	×50
	Value-aware Parity Insertion ECC	4×10^{-4}	×2000
MobileNet V2	No ECC	4×10^{-7}	
	Weight Nulling	2×10^{-6}	×5
	In-place Zero-space ECC	9×10^{-5}	×225
	Value-aware Parity Insertion ECC	7×10^{-4}	×1750

TABLE V
IMPACT ON THE BASELINE CLASSIFICATION ACCURACY BY THE PROPOSED VALUE-AWARE PARITY INSERTION ECC.

Model	Accuracy (Baseline)	Accuracy (Proposed method)
ResNet-50	71.7%	71.7%
DenseNet-169	74.4%	74.4%
Inception V3	73.5%	73.5%
MobileNet V2	65.4%	61.6%

deployment. Thus, the value-aware parity insertion ECC would be an attractive solution for fault-tolerant DNN architecture.

C. Impact on Baseline Accuracy

The proposed parity insertion technique sacrifices few bits in the weights for parity bits. While the weights less than *threshold value* can restore their original values, the weights greater than or equal to *threshold value* would not be fully restored as the lower-order bits ($b_1 b_0$) are masked to 0s. Hence, it is critical to analyze how much the baseline accuracy is affected by the proposed parity insertion technique.

Table V shows the accuracy difference between conventional DNN models (i.e., no weight protection (Baseline)) and the proposed method in an error-free condition. For ResNet-50, DenseNet-169 and Inception V3, the proposed method maintains its baseline accuracy. However, in MobileNet V2, 3.8% of accuracy loss is observed by the proposed method. It can be explained by investigating Figure 1(d). Unlike other models, the distribution of weight values in MobileNet V2 is more stretched and the number of weights belonging to distribution tails is relatively larger. This means that, for this model, the number of weights greater than or equal to *threshold value* is higher when compared to other models. As the lower-order bits are used in many weights, the meaningful lower-order bits being replaced by the parity bits introduce the accuracy loss. However, in the presence of errors, the proposed method effectively maintains the reliability of MobileNet V2 with higher BERs while the conventional model itself drastically loses the accuracy as the BER increases. It should be noted that if a significant accuracy degradation is a concern, the value-aware parity insertion ECC can be enabled if DNN becomes unreliable by aging, environmental or other mechanisms responsible for errors in the weights.

VI. CONCLUSION

This paper presents a value-aware parity insertion ECC that makes the weights fault-tolerant with negligible storage overhead and no additional training requirements. It finds parity locations in weights by weight value comparison and stores parity bits in the weights to mitigate a parity storage overhead. Most of weights are fully recovered and the baseline accuracy is kept from all models, except MobileNet V2. Compared to the critical BERs of *Weight Nulling* and *In-place Zero-space ECC*, DNN with the proposed ECC can perform inference without the accuracy degradation in $122.5\times$ and $15.1\times$ higher BER conditions on average, respectively. This proves that the value-aware parity insertion ECC would be an elegant solution for fault-tolerant DNN development as it almost removes a storage overhead and there is no need for retraining or fine-tuning.

ACKNOWLEDGMENT

This work was supported by Samsung Electronics Co., Ltd. under project number IO201208-07834-01.

REFERENCES

[1] S. Han *et al.*, “EIE: Efficient Inference Engine on Compressed Deep Neural Network,” in *ISCA*, June 2016.
[2] Y. Chen *et al.*, “Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks,” *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, Jan 2017.
[3] I. Ullah *et al.*, “Factored Radix-8 Systolic Array for Tensor Processing,” in *DAC*, 2020.
[4] S. Cha *et al.*, “Defect Analysis and Cost-Effective Resilience Architecture for Future DRAM Devices,” in *HPCA*, Feb 2017.
[5] J.-S. Kim *et al.*, “DRIS-3: Deep Neural Network Reliability Improvement Scheme in 3D Die-Stacked Memory based on Fault Analysis,” in *DAC*, 2019.
[6] J. M. You *et al.*, “MRLoc: Mitigating Row-hammering based on memory Locality,” in *DAC*, 2019.
[7] M. Imran *et al.*, “Flipcy: Efficient Pattern Redistribution for Enhancing MLC PCM Reliability and Storage Density,” in *ICCAD*, 2019.
[8] S. Jeong *et al.*, “PAIR: Pin-aligned In-DRAM ECC architecture using expandability of Reed-Solomon code,” in *DAC*, 2020.
[9] B. Reagen *et al.*, “Ares: A framework for quantifying the resilience of deep neural networks,” in *DAC*, June 2018.
[10] Z. Du *et al.*, “Leveraging the Error Resilience of Machine-Learning Applications for Designing Highly Energy Efficient Accelerators,” in *ASP-DAC*, 2014.
[11] J. Deng *et al.*, “Retraining-based timing error mitigation for hardware neural networks,” in *DATE*, March 2015.
[12] C. Schorn *et al.*, “An Efficient Bit-Flip Resilience Optimization Method for Deep Neural Networks,” in *DATE*, March 2019.
[13] O. Russakovsky *et al.*, “ImageNet Large Scale Visual Recognition Challenge,” *IJCV*, vol. 115, no. 3, pp. 211–252, 2015.
[14] W. W. Peterson and J. E. J. Weldon, *Error-Correcting Codes*, 2nd ed. MIT Press, 1972.
[15] M. Qin *et al.*, “Robustness of Neural Networks against Storage Media Errors,” *arXiv preprint arXiv:1709.06173*, 2017.
[16] H. Guan *et al.*, “In-Place Zero-Space Memory Protection for CNN,” in *NIPS*, 2019.
[17] K. He *et al.*, “Deep Residual Learning for Image Recognition,” in *CVPR*, 2016.
[18] G. Huang *et al.*, “Densely Connected Convolutional Networks,” in *CVPR*, 2017.
[19] C. Szegedy *et al.*, “Rethinking the Inception Architecture for Computer Vision,” in *CVPR*, 2016.
[20] M. Sandler *et al.*, “MobileNetV2: Inverted Residuals and Linear Bottlenecks,” in *CVPR*, 2018.
[21] “Keras: the Python deep learning API,” https://keras.io, 2021.