Improving Compute In-Memory ECC Reliability with Successive Correction

Brian Crafton[1], Zishen Wan[1], Samuel Spetalnick[1], Jong-Hyeok Yoon[2], Wei Wu[3], Carlos Tokunaga[3], Vivek De[3], and Arijit Raychowdhury[1]

[1]Georgia Institute of Technology [2]Daegu Gyeongbuk Institute of Science and Technology [3]Intel Labs

***Abstract*—Compute in-memory (CIM) is an exciting technique that minimizes data transport, maximizes memory throughput, and performs computation on the bitline of memory sub-arrays. This is especially interesting for machine learning applications, where increased memory bandwidth and analog domain computation offer improved area and energy efficiency. Unfortunately, CIM faces new challenges traditional CMOS architectures have avoided. In this work, we explore the impact of device variation (calibrated with measured data on foundry RRAM arrays) and propose a new class of error correcting codes (ECC) for hard and soft errors in CIM. We demonstrate single, double, and triple error correction offering over 16,000× reduction in bit error rate over a design without ECC and over 427× over prior work, while consuming only 29.1% area and 26.3% power overhead.**

I. Introduction

The ever growing performance gap between main memory and logic coupled with trends towards data-centric applications poses a signficant challenge for modern computing systems. These applications demand higher memory capacity and bandwidth, while also lacking signficant data re-use that on-chip SRAM cache has historically exploited to reduce bandwidth requirements. To further this challenge, applications of machine learning are trending towards more memory intensive operations [1] on smaller, resource constrained devices [2]. These challenges have motivated widespread adoption of hardware accelerators and high bandwidth memory (HBM) to maximize the compute throughput under modern memory constraints. Despite strong improvements, we face limitations that have inspired new memory technologies and techniques to enable future workloads on future computing systems.

CIM is one such research thread that reads and accumulates multiple memory cells onto the same bitline (BL). This increases memory bandwidth and performs (binary) multiplication and addition without the use of CMOS logic. At the same time, various eNVM such as RRAM are being actively developed to enable high density non-volatile storage while being both logic and process compatible [3]. Despite these benefits, both CIM and eNVM face several challenges not before faced by traditional CMOS designs. First, because CIM accumulates several cells on the same bitline, it increases total noise and reduces sensing margin for each state. Conventional digital memory such as SRAM overcomes this challenge using differential sensing and a large ratio between the '0' and '1' states. However, when reading multiple memory cells at the same time with an analog-to-digital converter (ADC), high variation between resistive states results in sum-of-products errors accumulated on the bitline. Therefore, CIM will inherently have a higher bit error rate (BER) than traditional memory arrays which read a single wordline (WL) at a time.

Recent work has attempted to mitigate the impact of these errors in several different ways. Iterative write-verify methods have been proposed to reduce the cell-to-cell variation [4]. These methods write the device until the resistance falls inside a specified range. However these methods require high write energy and latency, and greatly reduce the endurance of the device. Another method is variation-aware training [5]. This method trains models while simulating errors that may occur during inference. Statistical methods have been used to maximize CIM performance (active wordlines) under error constraints [6]. Lastly, several recent ECC techniques have been proposed for CIM [7]–[9]. These techniques make use of new encodings and device properties to enable ECC.

In this work, we present a new class of ECC schemes for hard and soft CIM errors based off of traditional single error correction, double error detection (SECDED) used in DRAM and SRAM today. We achieve this with a few key observations which we demonstrate using experimental data collected from a 40nm foundry RRAM test-chip array, whose circuit details appear in [10]. We build off of prior work that demonstrated SECDED for CIM [9], and propose a new technique called *successive correction* which can be used to correct soft errors in CIM. Successive correction enables single, double, and triple error correction offering over 16,000× BER reduction, while consuming only 29.1% area and 26.3% power overhead.

II. Background and Motivation

A. Compute In-Memory (CIM)

To implement VMM ($\vec{y} = W\vec{x}$), CIM systems encode the input vector $\vec{x}$ as wordline voltages and the weight matrix W as conductance states in a memory cell. The current through each cell is proportional to the product of the programmed conductance (W_{ij}) and applied voltage ($\vec{x}_i$). The resulting currents that are summed along the columns of the crossbar are proportional to the product of the matrix and vector, ($\vec{y}$).

Typically, each memory cell only stores 1 or 2 bits and thus weights (W_{ij}) requiring higher precision are encoded as several cells on the same wordline. Similarly, input voltages are limited to 1 bit and vector values ($\vec{x}_i$) are input to the wordline over several cycles. Therefore, to implement 8-bit VMM with 1-bit cells and 1-bit voltages, we must use 8 adjacent cells for each weight (Fig. 1) and 8 voltage pulses for

DAC '22, July 10–14, 2022, San Francisco, CA, USA

ACM ISBN 978-1-4503-9142-9/22/07…
https://doi.org/10.1145/3489517.3530526

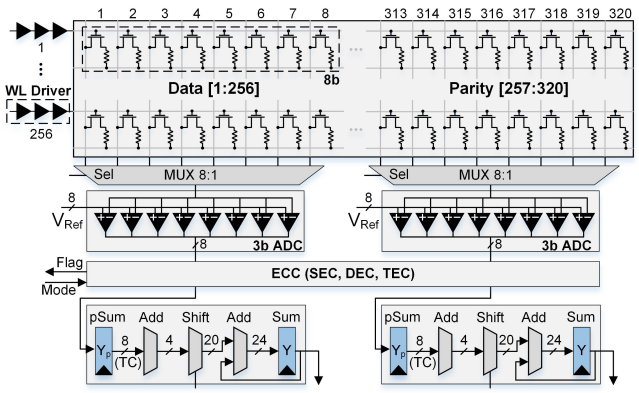

Fig. 1. 256×320 RRAM array architecture. 8 adjacent cells form an 8-bit weight and share a 3-bit ADC through a 8-to-1 multiplexer. Shift and add logic accumulate partial sums and apply corresponding magnitude.

each input. This encoding scheme results in 64 (8 cells × 8 cycles) partial sums that we must shift and sum to generate the full 8-bit VMM. In Figure 1, we illustrate our ECC-equipped RRAM macro design. This macro contains 256×320 RRAM cells, 3b ADCs, shift-and-add logic, and an ECC decoder. The additional 64 RRAM cells per WL (256+64 = 320) are check bits required for the ECC schemes we implement in this work.

B. Characterizing Device Variation in RRAM

CIM seeks to read and accumulate several states on a bitline at once, and therefore a key obstacle to enabling CIM is cell-to-cell variation. These variations are typically normally distributed [4], [10] and measured as the standard deviation (σ) from the mean (μ) resistance value. For binary (2-level) cells, a digital '0' is encoded as the high resistance state (HRS) and a digital '1' is encoded as the low resistance state (LRS). Ideally, current from reading a cell in the HRS could be ignored if the difference between the HRS and LRS (on/off ratio) were several orders of magnitude. Unfortunately, this is not the case as recent RRAM [10] and PCM [4] demonstrate an on/off ratio between 10× to 100×. To quantify the variation and on/off ratio, we measure the resistance values of RRAM cells from a recent RRAM test-chip prototype on a 40nm foundry RRAM array [10]. The array contains 256×256 RRAM cells (64Kb). The details of the read and write circuit of the array are beyond the scope of this paper and interested readers are pointed to [10] for further discussions.

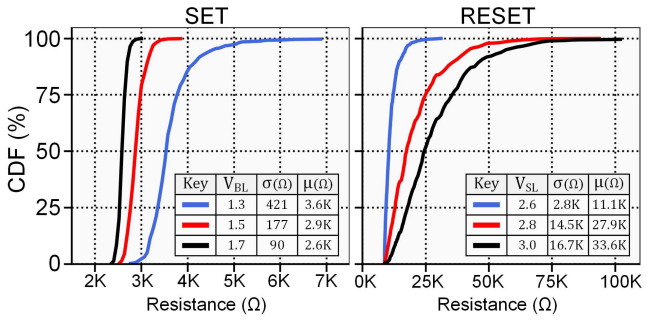

Fig. 2. CDF of measured resistance values for various write voltages for (A) set operation and (B) reset operation [10].

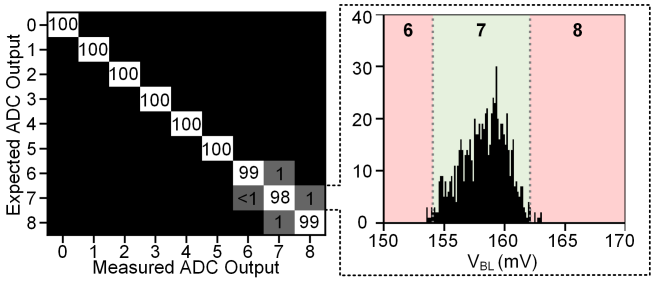

Fig. 3. Confusion matrix between measured and expected ADC outputs during CIM for measured data using 1.7V.

In Figure 2, we show measured resistance distributions for both the set (LRS) and reset (HRS) operations. We use 3 different write voltages for both set and reset, and find that higher write voltage decreases LRS variation, increases HRS variation, and increases on/off ratio. However, it should be noted that higher write voltage lowers device endurance and increases power. For LRS, we find that between 3.5% (σ/μ) and 11.7% cell-to-cell variation can be achieved depending on write voltage. For HRS, cell-to-cell variation is between 25.2% and 49.7% depending on write voltage. For on/off ratio, we observe between 3.1× and 12.9× depending on both set and reset voltages. We use these parameters as the basis for our simulations in Section V.

To understand how these parameters yield errors when performing CIM, we randomly program the cells such that a uniform distribution of values is achieved (0-8 LRS) and perform multi-row read. Figure 3 shows this result in the form of a confusion matrix where the expected ADC output code is on y-axis and the measured ADC output code is on the x-axis. It should be noted that we adjust ADC reference voltages to account for IR-drop along the bitlines of the array. Each bin shows the percent of measured ADC output codes were obtained for the expected ADC output code. When the number of LRS states is low (<5) the result is always correct for the experiment's sample size (8192 total). This behavior occurs because LRS variation ($\sigma_{\text{LRS}}/\mu_{\text{LRS}}$) has more impact than HRS variation. On the right, we show a histogram of samples measured for the 7 LRS states. The colored regions show where references are set for 6, 7, and 8 LRS states.

C. Stuck-At Faults in RRAM

Stuck-at faults (SAF) are present in all memories used today, and occur when a memory cell is stuck in the '0' or '1' state and cannot be overwritten. RRAM is no exception, and several works have studied and characterized the SAF for RRAM [11]. The SAF is categorized as a hard error because it cannot be corrected by tuning the cell. This is fundamentally different from soft faults observed during CIM that occur as sum-of-product errors due to device variation. In RRAM, SAF occur for a variety of reasons. First, process variation lowers yield and can cause some cells to be incorrectly fabricated. Next, RRAM suffers from relatively low endurance [11]. Lastly, RRAM cells require a forming step. During this forming step, a large voltage is applied across the device to create the

filament that serves as the memory. Prior work [11] observes that the high voltage applied during forming can cause SAF for some of RRAM cells.

D. ECC for CIM

In addition to IWV and variance-aware training methods, several works have proposed new ECC schemes for CIM [7]–[9]. In [7], arithmetic codes are applied to CIM for ECC. Arithmetic codes are a special class of ECC that protect arithmetic operations from faults. The main difference between arithmetic codes and standard ECC is the type of errors they detect and correct. Arithmetic codes protect against additive syndromes rather than bit flips. When applied to CIM, arithmetic codes correct single errors after summation, but cannot localize where an error occurs. For this reason, they cannot be applied to arbitrary blocks of data like traditional ECC. Instead, they must be applied to the operands of the operation they protect. So for 8-bit matrix multiplication (standard in DNN inference), 5-bits of check bits are padded to the weights (62.5% overhead).

In [9], the traditional SECDED ECC applied to commercial memory systems today is extended to support CIM. It is well known that SECDED cannot protect CIM because errors in the multi-level outputs cannot be localized or corrected. However, [9] observes that only ±1 errors occur in CIM. These ±1 errors are equivalent to bit flips ($0\rightarrow1$, $1\rightarrow0$) in traditional memory systems and allow localization to be performed. We re-create this behavior on our test macro in Figure 3, where all errors occur only ±1 from the correct result. To correct the ±1 error, [9] also requires a 2-bit checksum to compute the sign of the error. Fortunately, the double error detection bit serves as 1-bit of this checksum so that the modified SECDED requires only 1 additional bit.

III. Error Correction & Detection for CIM

A. Successive Correction by Detection

As we have seen in Section II, soft errors occur in CIM due to accumulated device variations that ultimately lead to a ±1 error. The simplest way to control accumulated variation-induced errors is to enable just 1 WL at time. Of course, this will unfortunately compromise the performance and energy efficiency of using CIM. One way to improve this trade-off is through error detection. If we can detect an error during CIM, we can re-read the WLs that led to the error at fewer WLs per cycle. This idea would allow us to maintain high performance by reading several WLs per cycle, but also reduce sum-of-product errors by reading fewer WLs upon error detection.

Fortunately, single error detection (SED) is inexpensive to implement. To illustrate SED, we break down the implementation steps required in Figure 4 using an example problem. First, we can pad each word in the array with a parity bit. This bit is '0' if the sum of the bits in the word is even, and '1' if odd. During CIM, this parity bit is read and accumulated just like the other bits on the wordline. After the CIM result is latched by the ADCs, detection is performed. In our example, an error occurs in the 2nd data bit because there exists 2 LRS cells but our ADC reads only 1. We detect this error by comparing the parity bit result with the XOR (LSB only) of the ADC outputs.

Upon detecting the error, correction is performed by re-reading the WLs. For each error that is detected, we divide the target WLs into 2 sets. Thus, in our example we read WLs 1 and 2 in step 1 and WLs 3 and 4 in step 2. Because another error occurs during step 1, we again breakdown the WLs into 2 more sets. This time performing only a single WL at a time, we read WL 1 in step 3 and WL 2 in step 4. Because step 2 did not result in error, the process ends not requiring any additional steps. For this small example, we could have read each WL serially and achieved the same performance (4 cycles) as recursively breaking down the WLs in 2 sets after each error. However, soft errors are typically corrected after the first recursion. Furthermore, if a hard error occurs, then it is quickly localized along one branch and also minimizes WLs re-reading time. This technique is especially powerful when many WLs are enabled ($\geq$32) which is demonstrated in Section V.

It should be noted that this technique relies on a relatively low BER. If errors are detected frequently, CIM performance will reduce towards 1 WL/cycle. Furthermore, this technique cannot correct SAF errors. To enhance the correction capability, successive correction can be paired with CIM-SECDED [9]. This combination can enable both double error correction (DEC) and triple error correction (TEC) after detection. This idea is the basis of our work, and in the following sections we discuss the implementation of DEC and TEC.

B. Double Error Correction

So far we have demonstrated how error detection can be used to correct soft errors in CIM. Furthermore, we have shown how a single parity bit can be used to achieve error detection in CIM. Using this idea, we can extend SECDED [9] to enable DEC. By definition, SECDED corrects single errors and detects double errors. The only purpose of double error detection is to prevent 2-error syndromes from being

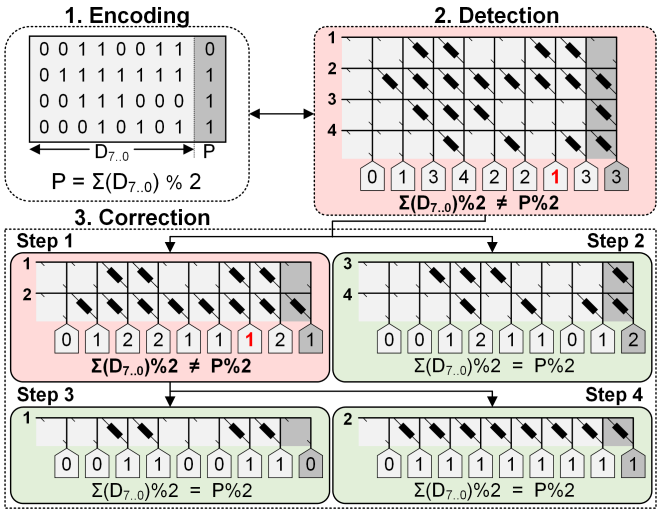

Fig. 4. Example of successive correction applied to 8b data with 1b parity

erroneously interpreted during SECDED. However, 2-error syndromes can be corrected using successive correction. So during CIM, SECDED is performed the same as [9]. But upon double error detection, successive correction is performed. At each recursive step of successive correction, SECDED is performed and the process continues until only 1 WL is enabled.

Using successive correction, DEC can be implemented using the same encoding and decoding logic as SECDED, the only difference is how double error detection is handled. This scheme can perform DEC for all 2-error syndromes, but for many >2-error syndromes, it can correct significantly more hard or soft errors. For example, consider a scenario where a 4-error syndrome occurs during CIM and DED is asserted. Successive correction will break down the WLs into 2 sets and re-try CIM. As this process continues, successive correction corrects all 4 errors. It should be noted that not all 4-error syndromes will be corrected by successive correction. Many will be misinterpreted as 1-error syndromes and erroneously "corrected" using SECDED. However, this reveals a powerful observation. By enabling more detection capability, successive correction can provide greater correction capability. Next, we discuss how TED can be used to implement TEC.

C. Triple Error Correction

To perform SECDED, we require an ECC code with a hamming distance of 4. Because SECDED has a hamming distance of 4, it can also be used perform triple error detection. For most practical applications, triple error detection has no utility, but using successive correction this code can be used to correct 3 (hard or soft) errors. To do so, we use our SECDED code for TED during CIM. Instead of decoding syndromes as 1-error or 2-error like SECDED, TED simply detects any non-zero syndrome as an error. So during CIM, our SECDED code is used to perform TED. Upon any error detection, successive correction is performed. At each recursive step of successive correction, TED is again performed and the process continues until only 1 WL is enabled. If only 1 WL is enabled, then SECDED is performed so that errors can be corrected.

In Figure 5, we present a schematic that serves as the decoder for both DEC and TEC. There are several XOR trees that detect the syndrome and perform traditional DED. The output from these modules are then used to perform TED and DED. TED is asserted when any non-zero syndrome occurs and DED is asserted when a non-zero syndrome occurs and

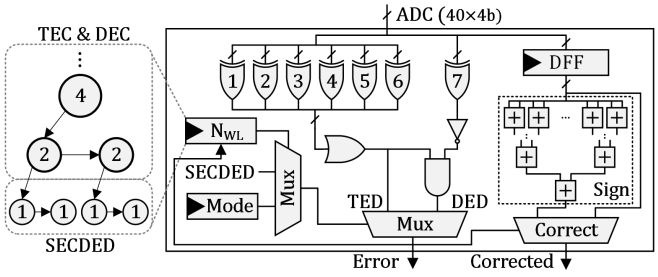

Fig. 5. Digital logic schematic of DEC & TEC decoder module.

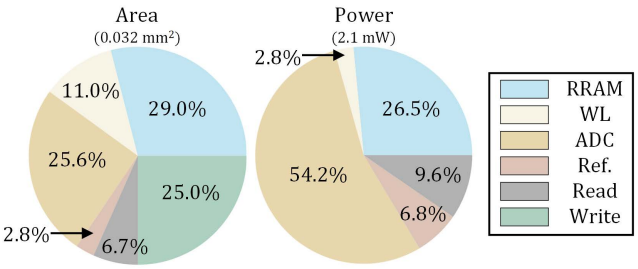

Fig. 6. Area and power breakdown by component in RRAM macro.

the XOR of all inputs is zero. The result of detection controls the next state of the N_{WL} state machine. This is used to keep track of state in successive correction. When N_{WL}=1, we perform SECDED because there is no utility in using successive correction.

IV. Hardware Implementation

To architect each of our ECC permutations, we consider a typical 256×256 RRAM macro based on the design we have measured data for. A schematic for this macro is given in Figure 1. Built around the RRAM array are necessary peripheral circuits. There is 1 3-bit ADC for every 8 columns where a single column is pitch-matched with a comparator. In standard CIM implementations [12], the ADCs feed into shift and add logic. However, for each ECC scheme we evaluate (besides arithmetic codes) the ADCs feed into the ECC decoder. After which shift-and-add is performed to implement matrix multiplication. It should be noted that for arithmetic codes [7], the decoder follows the shift-and-add logic.

To evaluate the area and power overhead of each ECC scheme, we first measure the area and power breakdown of our test macro [10]. The result of this is shown in Figure 6 as pie charts. Area is split across the 256×256 RRAM array and several peripheral circuits. For power, we observe a similar breakdown with ADCs consuming the majority of power during CIM operation. A major difference is that the write drivers account for zero power during CIM operation, but still account for a large fraction of area.

Using this measured data as baseline, we evaluate each ECC scheme by synthesizing and simulating RTL (Verilog) implementation. The area in μm^2 and percentage (%) of total macro area is given in Table 1. SECDED and DEC have the same implementation, so their overheads are identical. TEC requires slightly more area and slightly lower average power because correction logic is disabled during detection. Arithmetic codes require a 70.9% area overhead because the ratio between data bits and check bits is high for a 8-bit operation. Lastly, SEC has the lowest overhead (3.3%) because only 1 check bit and an XOR tree for detection is required. For power, we see similar results. However, the power overhead of the decoders is much smaller than their area overhead. It is important to note that for SEC, DEC, and TEC the energy consumed per correction would be higher than the other schemes because of sequential readout. This cost would be dependent on the number of errors detected during run time.

For our simulation experiments we adopted a similar architecture to previous work [12]. Our basic processing element (PE) contains 32 256×256 RRAM arrays. The activation inputs to the RRAM sub-arrays are stored in on-chip SRAM, while the input images are read in from external DRAM. Matrix multiplication is performed by the PEs, while custom vector units are used to perform vector-wise accumulation, bias addition, quantization, and ReLU. Because all inputs in ResNet18 are positive valued, we can use two's complement representation for our weights and avoid offset format.

V. Results

To benchmark our three ECC schemes (SEC, DEC, TEC), we compare against two prior works [7], [9] and a design without any ECC. We empirically evaluate performance, BER, and accuracy degradation on ImageNet using ResNet18. To perform these evaluations we construct a simulator capable of executing tensor operations using CIM with non-idealities (variation, SAF) based on our measured data. The ResNet18 model uses 8-bit weights and activations.

A. Simulation Framework

Our simulator performs cycle-accurate implementations of convolutional and fully connected layers. It is based in Python, but runs array level operations in C for faster evaluation. We model components in the design in object oriented fashion, iterating through all components in all PEs each cycle. Each ECC is implemented within the simulator by padding the check bits to the matrices encoded in the array. After each CIM operation, the ECC localization and correction is run on the result. We embed performance counters in our ADC and sub-array objects to track cycles, errors, and ECC events. As output, the simulator produces a table with performance counters and all intermediate layer activations that are verified against a TensorFlow implementation for correctness.

B. Simulation Results

To evaluate our ECC schemes, we sweep over expected ranges for both soft errors and hard errors. For soft errors, we consider LRS variation ($\sigma_{\text{LRS}}/\mu_{\text{LRS}}$), HRS variation ($\sigma_{\text{HRS}}/\mu_{\text{HRS}}$), and on/off ratio ($\mu_{\text{HRS}}/\mu_{\text{HRS}}$) from our measured devices. For hard errors, we sweep over a wide range of SAF rates because existing data is sparse and likely not consistent with state-of-the-art devices [11]. For our simulations we assume 8 WLs are enabled per cycle (except Figure 9) and zero-skipping is utilized to exploit sparsity.

ECC	Arithmetic	SEC	SECDED & DEC	TEC
Data Bits	8	32	32	32
Check Bits	5	1	8	8
Data Area (μm^2)	6750	27000	27000	27000
Check Area (μm^2)	4219	845	6750	6750
Decoder Area (μm^2)	570	49	1087	1097
Area Penalty (%)	70.9	3.3	29.0	29.1
Data Power (μW)	475	1900	1900	1900
Check Power (μW)	297	60	475	475
Decoder Power (μW)	17.2	1.1	25.4	23.5
Power Penalty (%)	66.1	3.2	26.3	26.2

Table 1. Area and power overhead using various ECC schemes.

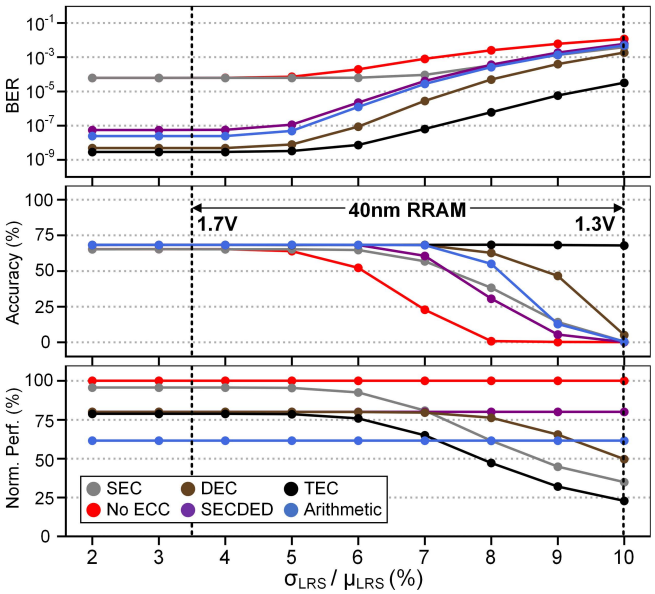

Fig. 7. Classification accuracy, BER, and normalized performance versus cell-to-cell variation ($\sigma_{\text{LRS}}/\mu_{\text{LRS}}$) for ResNet18 on ImageNet.

C. Soft Errors

Starting with soft errors, we simulate ImageNet & ResNet18 over the range of measured LRS variation and plot the result in Figure 7. We consider LRS variation because the measured range of LRS variation has significantly more impact on BER than the measured range of HRS variation or on/off ratio. We fix HRS variation at 35% and on/off ratio at 12. We also consider a fixed SAF rate of 10^{-5}. In Figure 7, we display BER, accuracy, and performance. For performance, we show normalized performance against the baseline implementation without ECC. To normalize the ECC implementations, we also consider their area overhead from check bits. For example, SEC requires 1 parity bit and a decoder characterized in Table 1. Therefore, we normalize performance by 3.1% to account for the reduced computational efficiency in a scaled design.

DEC and TEC yield the lowest BER for all LRS variation cases. For lower LRS variation, they yield nearly the same performance as SECDED because they seldom need to re-read the WLs. However, for higher LRS variation, performance greatly decreases because errors occur and WLs are frequently re-read. Interestingly, arithmetic codes yield lower performance and higher BER than both DEC and TEC. This is because of the high overhead required for arithmetic codes on 8-bit operations. At 4% LRS variation (1.7V in Fig. 2), TEC achieves 34.5× and 14.9× BER reduction over CIM-SECDED and arithmetic codes. TEC yields a 1.8% slowdown over CIM-SECDED and a 27.8% speedup over arithmetic codes at 4% LRS variation. At 6% LRS variation (1.5V in Fig. 2), TEC achieves 636.0× and 427.1× BER reduction over CIM-SECDED and arithmetic codes. TEC yields a 5.4% slowdown over CIM-SECDED and a 23.2% speedup over

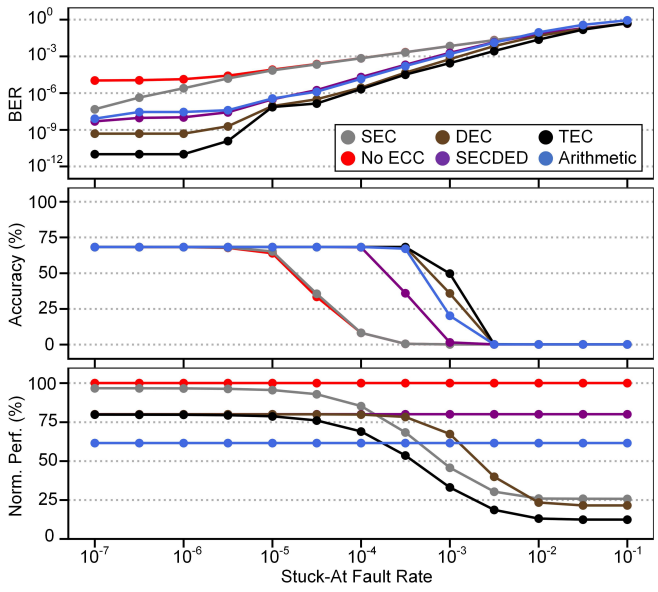

Fig. 8. Classification accuracy, BER, and normalized performance versus SAF rate for ResNet18 on ImageNet.

arithmetic codes at 6% LRS variation. Lastly, TEC enables full accuracy on ImageNet up to 10% device variation despite coming at a severe performance penalty.

D. Hard Errors

Next, we perform the same experiments while sweeping over SAF rate and plot the results in Figure 8. We fix LRS variation at 4%, HRS variation at 35% and on/off ratio at 12. We observe similar results to those in Figure 7 with a few major differences. First, SEC yields nearly the same BER as the design without any ECC. This is because it cannot correct hard errors by reading 1 WL per cycle. Next, DEC and TEC also produce nearly the same BER. This is again because hard errors cannot be corrected by reading 1 WL per cycle. For higher SAF rates, we find that all ECC schemes produce very poor results. However, for lower SAF rates DEC and TEC yield significantly better BER than prior work while achieving same or better performance.

E. BER vs Performance Tradeoff

Because the proposed ECC requires overhead in terms of both check bits and re-read latency, it will naturally have lower performance than a design without ECC. However, ECC can greatly reduce BER and thus we can enable more parallel WLs while achieving the same BER. To understand how our ECC scales to more parallel WLs, we perform soft error experiments operating at various numbers of WLs. We run these experiments at 8, 16, and 32 parallel WLs for designs with TEC ECC and designs without any ECC. The result of this experiment is shown in Figure 9. We find that TEC can enable 32 parallel WLs at lower BER than 8 parallel WLs with no ECC for all experiments. At 3.5% variation, we observe a 2.32× speedup and greater than 200× reduction in BER. Thus TEC ECC can achieve higher peak performance for a given

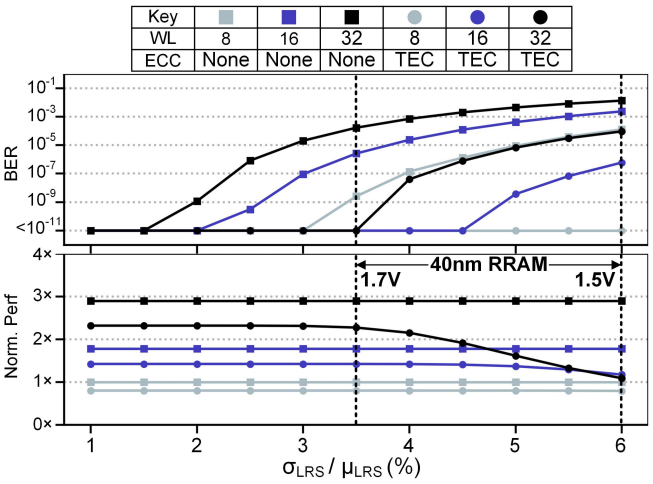

Fig. 9. BER and normalized performance for increasing # parallel WL.

BER over the range of measured device variation on our 40nm foundary RRAM array.

VI. Acknowledgement

This work was funded by the U.S. Department of Defense's Multidisciplinary University Research Initiatives (MURI) Program under grant number FOA: N00014-16-R-FO05 and the Semiconductor Research Corporation under C-BRIC.

References

[1] U. Gupta *et al.*, "The architectural implications of facebook's dnn-based personalized recommendation," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 488–501, IEEE, 2020.

[2] V. Sze *et al.*, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, pp. 2295–2329, 2017.

[3] S. Yu *et al.*, "Compute-in-memory chips for deep learning: Recent trends and prospects," *IEEE Circuits and Systems Magazine*, vol. 21, no. 3, pp. 31–56, 2021.

[4] J. Wu *et al.*, "A 40nm low-power logic compatible phase change memory technology," in *2018 IEEE International Electron Devices Meeting (IEDM)*, pp. 27–6, IEEE, 2018.

[5] Y. Long *et al.*, "Design of reliable dnn accelerator with un-reliable reram," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1769–1774, IEEE, 2019.

[6] B. Crafton *et al.*, "Statistical optimization of compute in-memory performance under device variation," in *2021 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 1–6, IEEE, 2021.

[7] B. Feinberg *et al.*, "Making memristive neural network accelerators reliable," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 52–65, IEEE, 2018.

[8] Q. Lou *et al.*, "Embedding error correction into crossbars for reliable matrix vector multiplication using emerging devices," in *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, pp. 139–144, 2020.

[9] B. Crafton *et al.*, "Cim-secded: A 40nm 64kb compute in-memory rram macro with ecc enabling reliable operation," in *2021 IEEE Asian Solid-State Circuits Conference (A-SSCC)*, pp. 1–2, IEEE, 2021.

[10] J.-H. Yoon *et al.*, "29.1 a 40nm 64kb 56.67 tops/w read-disturb-tolerant compute-in-memory/digital rram macro with active-feedback-based read and in-situ write verification," in *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 64, pp. 404–406, IEEE, 2021.

[11] 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.

[12] A. Shafiee *et al.*, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 14–26, 2016.