2018 IEEE International Symposium on High Performance Computer Architecture

Making Memristive Neural Network Accelerators Reliable

Ben Feinberg*, Shibo Wang†, and Engin Ipek*†
*Department of Electrical and Computer Engineering
†Department of Computer Science
University of Rochester
Rochester, NY 14627 USA
*bfeinber@ece.rochester.edu †{swang, ipek}@cs.rochester.edu

***Abstract*—Deep neural networks (DNNs) have attracted substantial interest in recent years due to their superior performance on many classification and regression tasks as compared to other supervised learning models. DNNs often require a large amount of data movement, resulting in performance and energy overheads. One promising way to address this problem is to design an accelerator based on *in-situ* analog computing that leverages the fundamental electrical properties of memristive circuits to perform matrix-vector multiplication. Recent work on analog neural network accelerators has shown great potential in improving both the system performance and the energy efficiency. However, detecting and correcting the errors that occur during in-memory analog computation remains largely unexplored. The same electrical properties that provide the performance and energy improvements make these systems especially susceptible to errors, which can severely hurt the accuracy of the neural network accelerators.**

This paper examines a new error correction scheme for analog neural network accelerators based on arithmetic codes. The proposed scheme encodes the data through multiplication by an integer, which preserves addition operations through the distributive property. Error detection and correction are performed through a modulus operation and a correction table lookup. This basic scheme is further improved by data-aware encoding to exploit the state dependence of the errors, and by knowledge of how critical each portion of the computation is to overall system accuracy. By leveraging the observation that a physical row that contains fewer 1s is less susceptible to an error, the proposed scheme increases the effective error correction capability with less than 4.5% area and less than 4.7% energy overheads. When applied to a memristive DNN accelerator performing inference on the MNIST and ILSVRC-2012 datasets, the proposed technique reduces the respective misclassification rates by 1.5x and 1.1x.

***Keywords*-Computer architecture; Accelerator architectures**

I. INTRODUCTION

In recent years, analog *in-situ* computing has received significant attention due to its potential to reduce the energy cost of data movement. At the same time, machine learning has attracted a great deal of interest with potential applications to many problems. These two trends have lead to accelerator proposals that exploit the fundamental electrical properties of memristive devices to perform matrix vector multiplication (MVM)—a dominant component of neural network computation—in the analog domain [1]–[3].

Analog MVM is not a new idea, as various implementations of analog MVM circuits have been developed over the past 25 years [4]–[8]. Advances in dense, CMOS compatible resistive memories now make these architectures even more compelling. While analog neural network accelerators such as the memristive Boltzmann machine [1], ISAAC [9], PRIME [10], and PipeLayer [11] exhibit significant potential to improve the energy efficiency, error correction for these accelerators remains unaddressed. The same fundamental electrical properties that allow for the speedup and energy efficiency improvements also make these systems much more sensitive to errors. **Even if no one cell is in error such that it would be read improperly in a traditional memory array, the sum of the errors when computing a dot product operation that spans an entire row can produce incorrect results.** Moreover, since *in-situ* computation relies on reading the result of a dot product operation performed in the analog domain without reading the individual data blocks from memory, it is not immediately clear whether conventional ECC techniques can correct faults in *in-situ* memristive accelerators.

Despite the tolerance of neural networks to limited precision and data errors, we find that the errors in a memristive network substantially impact the overall system accuracy, increasing the respective misclassification rates on the well known MNIST and ILSVRC-2012 datasets [12], [13] from 1% to 3%, and from 43% to 48%. To place the ILSVRC results in context, a 1% improvement in accuracy was the difference between GoogleNet [14] and VGG [15], respectively the winner and the runner up in the ILSVRC-2014 competition [13]. Likewise, the increase in the MNIST misclassification rate is comparable to the improvements achieved by data-skewing to increase the training set size, doubling the number of hidden units, or using a convolutional neural network (CNN) rather than a multi-layer perceptron (MLP) [12], [16]. Furthermore, these accuracy losses increase with multi-bit cells, limiting the scalability of memristive accelerators to higher numbers of bits per cell.

Memristive hardware accelerators require new error correction schemes tailored to their unique requirements. Unlike the traditional error correction techniques used in memory arrays in which a collection of data elements are corrected

2378-203X/18/$31.00 ©2018 IEEE
DOI 10.1109/HPCA.2018.00015

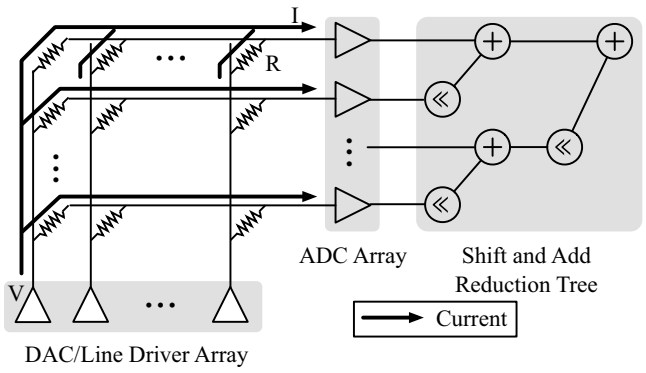

Figure 1. Memristive crossbar for matrix vector multiplication.

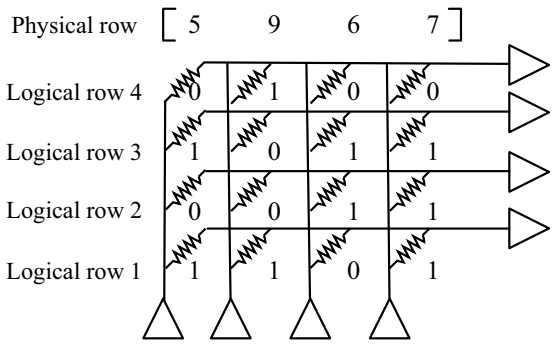

Figure 2. Illustrative example of the bit slicing operation.

prior to computation, here the correction must be performed post-computation. This paper presents the first work on error correction schemes for these accelerators. We propose an error correction scheme based on AN codes [17]–[19] that can correct the errors that occur during computation. We then present a new form of data-aware AN codes that increase the error correction capability by leveraging the properties of the resistive networks, and the importance of the error to overall system accuracy within a DNN computation. As a result, we can greatly restore the accuracy of the DNN models running on memristive hardware accelerators, enabling reliable, fast, and energy-efficient computation for an important class of machine learning workloads.

II. Background

The proposed techniques build upon prior work on memristive neural network accelerators, failure models of memristors, and arithmetic codes.

A. Digital Accelerators for Neural Networks

The growing interest in neural networks has made them a prime candidate for special purpose accelerators in recent years. DaDianNao [20] proposes an accelerator for convolutional and deep neural networks, utilizing eDRAM to store large network layers and intermediate values. TABLA [21] provides a template system and model definition syntax for the creation of FPGA based neural networks. In addition to this academic work, Google has deployed their custom Tensor Processing Unit neural network accelerator in their datacenters [22].

B. Memristive Accelerators for Neural Networks

Recent developments in memristive devices make it possible to build nanometer ICs with dense and programmable resistive networks. This capability has lead to significant interest in implementing the dot product in the analog domain, thereby improving the speed and energy efficiency of prediction with neural networks [1]–[3], [9]–[11].

Memristive dot product accelerators exploit the relationship between a dot product computation and the currents in a resistive mesh. Figure 1 shows a conceptual example of a dot product operation using a resistive network. The current through each resistor R_i is equal to V_i/R_i, and the total current through a bitline is $\sum V_i/R_i$. Hence, it is possible to implement a dot product $\mathbf{u} \cdot \mathbf{v}$ between two vectors $\mathbf{u}$ and $\mathbf{v}$ by programming the resistors in inverse proportion to the vector coefficients u_i, and setting the voltages V_i in proportion to the coefficients v_i. The dot product is then given by the current, I, as shown on the bitline in Figure 1.

Regrettably, this simple conceptual model relies on high precision memristor programming, analog to digital converters (ADCs) to read each row, and digital to analog converters to drive each column. To make memristive acceleration of dot product operations practical, recent proposals have employed various techniques that compensate for the limited precision with which these operations can be carried out. Without loss of generality, we focus on four of these recent proposals in the rest of this paper: 1) the Memristive Boltzmann Machine (MBM) [1], 2) ISAAC [9], 3) PRIME [10], and 4) PipeLayer [11].

1) Bit Slicing: To address the adverse effect of limited precision memristor programming and sensing on model accuracy, all four of the aforementioned accelerators employ a technique called *bit slicing*. The key idea is to map a single logical row of the matrix across multiple physical rows of an array. An example of bit slicing is shown in Figure 2, wherein a single logical row with four elements is bit sliced into four physical rows, one per bit position.

The full sum is computed through a shift-and-add reduction tree that aligns the sum bits from each row as shown on the right hand side of Figure 1. The bit slices can be organized in several different ways: grouping all of the bit slices for a single row within one array as in MBM [1], striping bit slices across multiple arrays so that each array contains a single bit slice for several different rows as in ISAAC [9], or a combination thereof.

In addition to the bit slicing of the matrix, the incoming

vector is bit sliced and applied to the memory array one or several bits at a time. A similar reduction operation is needed to reduce the resultant stream of bits into the full output.

2) Hierarchical Organization: To handle large matrices and to reduce the overhead of the peripheral circuitry through resource sharing, existing MVM accelerators use a hierarchical design. In MBM, the memory arrays are organized in a hierarchy of banks, subbanks, and mats. A large matrix is striped across multiple subbanks, and the results are aggregated via an intra-bank reduction network. In ISAAC, the system is organized into *in-situ* multiply accumulate (IMA) units, each of which includes multiple crossbar arrays, their peripheral circuits, and a reduction network capable of performing the shift and add operations. Larger reductions are performed within a tile that contains multiple IMAs, an internal data buffer, and global shift/add and sigmoid computation units. PRIME places its computational arrays within a standard memory chip, wherein each array performs a subset of the computation. In the case of large weight matrices, PRIME spreads a single matrix over multiple banks and performs reduction within another array.

C. Fault Models of RRAM

Significant research has been carried out on developing accurate fault models for RRAM. We focus primarily on noise sources that are directly relevant to memristive accelerators [23]: thermal noise, shot noise, random telegraph noise (RTN), and programming errors.

1) Thermal Noise: Thermal noise, also known as Johnson-Nyquist noise, is a property of all passive devices caused by the thermal agitation of carriers [24], [25]. Thermal noise can be modeled as a current source in parallel with a passive resistor. The magnitude of the current is modeled by a zero mean Gaussian distribution with a standard deviation of $\sqrt{\frac{4K_BTf}{R}}$, where K_B is the Boltzmann constant, T is temperature in Kelvins, f is the frequency of the signal, and R is the resistance of the device. Thermal noise is a fundamental property of resistive circuit elements, and cannot be mitigated unless the circuit is cooled or the operating frequency is reduced. Prior work by Soudry *et al.* [26] examined the impact of thermal noise on the analog training of memristive networks, finding that the noise defines a hard upper bound on the training accuracy.

2) Shot Noise: Shot noise affects the measurement accuracy of the current flowing into a detector. Shot noise is caused by the fluctuations in the number of electrons flowing through a cross-sectional area at a fixed point in time. Although such fluctuations can be averaged out over a sufficiently long measurement interval, filtering out shot noise over shorter measurement intervals is a significant challenge. Shot noise is modeled by a zero mean Gaussian with a standard deviation of $\sqrt{2qIf}$, where q is the charge of an electron, I is the current flowing into the device, and f is the frequency.

3) Random Telegraph Noise: Random telegraph noise (RTN) exists in both CMOS and memristive circuits, and constitutes a major cause of the errors in memristive devices [27]–[36]. RTN is caused by the electrons that temporarily become trapped within the device, thereby changing the effective resistance of the conductive filament. The result is a temporary and unexpected reduction in the resistance of a memristor at runtime. The trapping and untrapping of the electron follows a Poisson process and is unpredictable. When taking short measurements with a sampling frequency of a MHz or higher, there is no guarantee of the state that the device will occupy. Moreover, RTN is strongly state dependent. The resistance deviation ($\Delta R/R$) in the RTN error state varies from less than 1% to upwards of 40%.

While the exact mechanism of RTN in memristive devices is still the subject of significant research, we adopt a physically motivated model proposed by Ielmini *et al.* [30]. This model considers the resistive deviation as a function of the cross-sectional area of the resistive filament. In a low resistance state, the electron impacts only a small proportion of the conductive region, which results in a relatively small resistance deviation. However, as the resistance increases and the filament narrows, the electron impacts a greater proportion of the filament, until eventually the resistance deviation saturates as the electron impacts the entire filament. This model is consistent with results from measured devices of multiple material stacks, and with simulations [27], [28]. The Ielmini model determines the resistance deviation $\Delta R/R$ as a function of seven material parameters: 1) ε_r, the relative permittivity of the dielectric; 2) N_d, the concentration of dopant atoms; 3) ρ_0, the resistivity of the metallic nanowire; 4) th, the thickness of the dielectric; 5) α, the relative resistivity increase caused by the trapped electron; 6) T the temperature in Kelvins; and 7) the current resistance of the device. The RTN error probability is determined based on the time constants of switching, τ_{ON} and τ_{OFF} [28]. These time constants are based on the energy required for the state transition and are asymmetric, leading to different dwell times in each state. While the time constants are dependent on the material stack of the device, experimental results from multiple device stacks including TaO and HfO show asymmetric dwell times, with τ_{OFF} several times larger than τ_{ON} [27]–[29].

4) Programming Errors: Even in the absence of the aforementioned sources of noise in memristive circuits, process variations limit the precision with which a memristor can be programmed. Alibart *et al.* propose a simple, widely used scheme for accurate memristor programming; the key idea is to successively apply a series of short pulses to converge slowly on the target resistance [37]. This iterative programming technique may suffer from a long write latency; however, the memristive accelerators

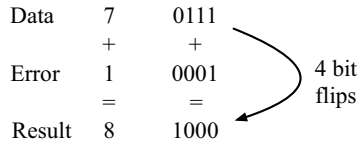

Figure 3. Arithmetic vs. Hamming distance.

discussed in this work are used for inference only, and do not require online updates. As a result, the relatively long programming times can be amortized over the entire lifetime of the learned neural network model. The short pulse programming technique has been shown to program resistance to within 1% of the target value; consequently, we permit a 1% deviation in the programmed resistances in the simulation experiments.

Programming errors are a major issue for the use of RRAM in both storage and *in-situ* computation, and several approaches to mitigate the problem have been explored. Niu *et al.* [38] study how the tradeoff between programming accuracy and duration—and by extension, energy and bandwidth—can be optimized by leveraging the traditional error correcting codes used in DRAM. They propose reducing the programming time of an array to reduce the write energy, and providing ECC support to correct the resultant errors. Xu *et al.* [39] examine programming optimizations for multi-level cells, including non-linear mapping of resistances, and two different programming schemes optimized for speed and long term data retention.

5) Manufacturing Defects: Device yield is a well known problem in CMOS design with a wide variety of architectural, circuit, and device level solutions. In DRAM, row and column sparing is used to increase the yield, wherein the rows or columns containing faulty cells are laser cut and replaced by a spare row or column. While this technique is effective, it incurs significant overheads even at error rates of less than a tenth of a percent [40]. The problem is exacerbated by the relative immaturity of RRAM technology, where two recent simulation studies have placed the cell error rates at greater than 1% [41], [42].

6) Endurance Failures: Memristor endurance varies widely based on the material properties and write mechanisms. An endurance range of 10^6 to 10^{12} writes has been reported [43], [44], after which the cell does not switch reliably and becomes stuck in one state. Managing device endurance has been widely explored in the context of PCM [45]–[48]. Existing memristive accelerators focus primarily on matrices that are written once during the problem setup, and then rewritten either for new problems or during network updates. Bojnordi *et al.* [1] compute a worst case system lifetime of 1.5 years for the Memristive Boltzmann Machine; for deep learning, however, this analysis does not consider the variability in device endurance. Even with a 1.5 year system lifetime, faults must be handled gracefully. Prior work by Xia *et al.* [49] proposed a scheme to map the weight matrices stored in memristive crossbars for computation around faults or endurance failures through a combination of neural network pruning and data remapping. This scheme increases the the life of the neural network accelerator, allowing it to be used for training; however, it does not address the transient faults caused by the analog summation of the error currents.

D. Arithmetic Codes

Arithmetic codes are a class of ECCs that conserve the result of a set of arithmetic operations when the operands are subject to noise. Formally, $f()$ is an arithmetic code over the operation $\circ$ if $f(x) \circ f(y) = f(x \circ y)$ [50]. Arithmetic codes have been applied in a variety of contexts to increase the reliability of systems built from unreliable hardware components [17], [18], [51]–[53].

An important difference between arithmetic and non-arithmetic codes is the type of error against which each code is designed to protect. In non-arithmetic codes, the errors are expected to be the flips of individual bits during data transmission or storage. In contrast, the expected errors in arithmetic codes manifest as an additive syndrome. The difference between these two error models can be seen by examining an example addition operation whose sum must equal 7 (Figure 3). An additive error of 1 can change the output of the computation from 0111 in binary representation to 1000. This change has a Hamming distance of four and resides outside the correction capability of the commonly used SECDED ECC [19], although from an arithmetic perspective, only a single error has occurred. Just as SECDEC ECC can protect against errors that occur due to a wide variety of physical phenomena that cause a bit flip, the arithmetic codes can protect against additive syndromes regardless of the source of the error.

We focus on a specific class of arithmetic codes called the AN-codes [52], [54], [55]. AN-codes are non-separable arithmetic codes that encode data through multiplication by an integer called A. These codes conserve addition operations through the distributive property, using the definition above: $Ax + Ay = A(x + y)$. Error detection in AN codes is performed through a simple modulus operation: if N is the output of a computation, the residue N%A must equal zero for a correct computation.

As the simplest case, consider an AN code where A = 3. This code carries an overhead of one bit, and can detect all additive syndromes of ± 1 or ± 2 at any bit position. This A = 3 code is comparable to single bit parity codes in non-arithmetic ECC, in that it is guaranteed to detect a single error. An A = 3 code cannot, however, correct errors because all residues of N are 0, 1, or 2, offering no information about the bit position of the additive error. To correct the errors, a code must have a sufficient number of unique residues for all possible syndromes, and the modulus of all syndromes

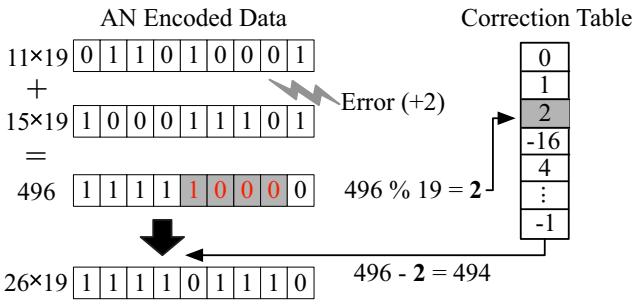

Figure 4. Example of error correction with AN codes using an A=19 code.

under A (syndromes%A) must be unique. This implies that:

$$A \geq \text{Number of Syndromes} \quad (1)$$

for an AN correcting code. Notably, this is a necessary, but not sufficient condition for identifying the permissible values of A.

Figure 4 provides an example of how AN codes can be used to correct the data after computation. An error at a single bit position can be viewed as the addition of a syndrome $\pm 2^i$. As mentioned above, each syndrome must map to a unique residual that can be stored in the correction table on the right. Once the residual is computed and used to index into the correction table, the table entry is subtracted from the result to restore the value. Note that the error in this example affects the values of multiple bit positions, even though the error itself is only a single bit.

The A = 19 code in Figure 4 is comparable to a SECDED Hamming code for 5-bit values. After the input data is multiplied by A, the resulting value is 9 bits wide, requiring 18 syndromes to correct all $\pm 2^i$ errors at each of the 9 bit positions. The mapping of the residuals to syndromes is stored in a correction table, shown on the right hand side of Figure 4. Since A = 19 has 19 possible residuals that indicate an error, an A = 19 code is the minimum code that can correct any single error. Similarly, for 32 bit values, an A = 79 code is used. A = 79 adds 7 bits for correction, resulting in 39 possible bit positions in error and 78 unique residuals; thus, A = 79 is the minimum A that can correct all single bit errors. A proof of this property for single error correcting AN codes can be found in [19].

Unlike a SECDED Hamming code, the A=79 AN code has no detection capability. Since all of the possible residues are used for correction, there is no excess capacity to indicate an uncorrectable error.

III. Can Memristive Dot-Product Accelerators be Protected with SECDED ECC?

One central question when attempting to design error correction schemes for memristive accelerators is whether traditional SECDED ECC can be relied on to proctect the result of the computation. Unlike the arithmetic codes

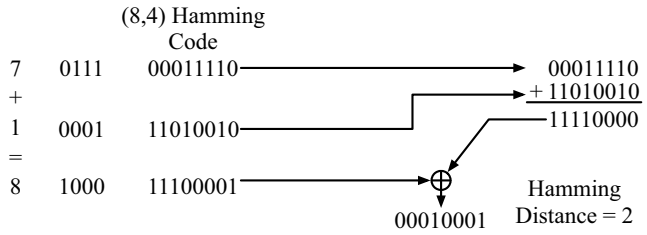

Figure 5. SECDED ECC does not conserve addition.

described in Section II-D, SECDED ECC does not conserve addition, and attempting to add two values encoded with SECDED ECC *in-situ* within a memristive crossbar is not guaranteed to be correct even in the error free case. This is shown in Figure 5 where we attempt to add two 4 bit values encoded with the (8,4) Hamming Code, without errors. On the left side of the figure, the addition operation is performed on the unencoded operands; the sum, 8, is then encoded with the (8,4) Hamming Code, $f(\mathbf{x} + \mathbf{y})$. On the right hand side, we show the addition of the two encoded 8 bit values, $f(\mathbf{x}) + f(\mathbf{y})$, using the previous notation. As shown, these two sums are not equal—in fact, they are separated by a Hamming distance of two so that no correction is possible with SECDED ECC. Since the computation is not guaranteed to work in the error-free case, there is also no guarantee that it would work in the presence of errors.

IV. Error Distributions in Memristive Accelerators

To examine the impact of errors on matrix-vector multiplication, we build a SPICE model of a single 128 entry row, and perform a transient analysis to study how the output current varies over 1 second of operation. The simulated circuit is shown in Figure 6. We adopt the model proposed by Hu *et al.* [23] with one important modification to the random telegraph noise. Hu *et al.* use a fixed $\Delta R/R$ of 10% across the entire resistance range and assume that the cells spend equal time in the RTN high and low states. These two assumptions lead to the error canceling effects that they note in large arrays. We use instead the Iemini model discussed in Section II-C3, where $\Delta R/R$ varies based on the memristor state and has an uneven state distribution that removes the error-canceling effects. This state dependent ΔR and the uneven state probabilities represent a better model of the RTN in a large number of memristive material stacks [27]–[36].

Since the detrimental effects of noise on the circuit increase with the row length, we model a single row driven by ideal voltage sources rather than a small crosspoint array. This model does not consider the reduced voltage seen by the rows that are farther from the drivers; however, the mapping function of Hu *et al.* [23] compensates for these effects. For simplicity, we model each memtistor as a resistor since we consider only the read process of each memristive device.

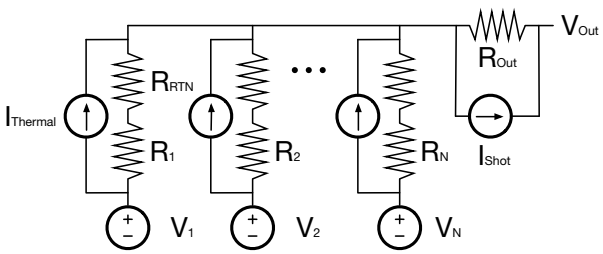

Figure 6. Circuit model of a single row.

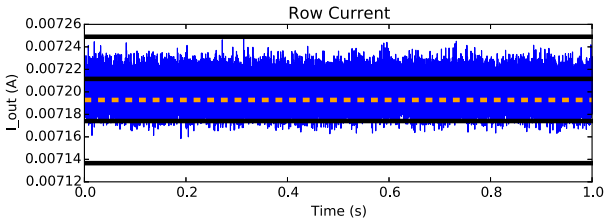

Figure 7. Current transient of a 128 element row with two bits per cell.

The current transient for a 128 element row, with two bits per cell and an equal number of elements in each state, is shown in Figure 7. The dotted orange line is the ideal error free current of the row with the black bars showing the ± 1 and ± 2 error thresholds. Since RTN errors do not have a zero-mean, we apply an *RTN offset* to the programmed resistance based on the $\Delta R/R$ of the target resistance and the probability of being in an RTN error state. This moves the overall average current seen closer to the error free current target, similar to the calibration scheme of Hu *et al.* [23] but without the series of calibration vectors. The overall error rate in this distribution is 14.5% (13.9% high errors, and 0.51% low errors). In practice, the error rate experienced may be different due to the dependence on both the input vector pattern and the row states. In particular, the case of all ones for the vector creates the worst case error probability since each cell contributes to the error distribution. This error dependence on the row state is discussed in the next section.

V. AN Codes for In-Situ Matrix Vector Multiplication

Since AN codes operate on full integer operands, they can be applied directly to an *in-situ* MVM operation by multiplying the matrix by the selected *A* value. The multiplication is performed before the data is bit sliced and written to the accelerator.

As discussed in Section II-D, AN codes are designed to correct errors that occur at a particular bit position. This is analogous to an error that occurs within a physical row such that the integer output of the ADC servicing that physical row differs from the actual quantized output. After the value has been reduced via a shift and add tree (Section 2.1.1), the final logical row output is corrected by the AN codes.

A. Naïve Use of AN Codes

We first consider AN codes that have been previously examined in other applications. The A = 19 code is shown in Figure 4, which corrects all single bit errors for 5-bit operands. For a 32-bit operand, an A = 79 AN adds 7 check bits for a final 39 bit encoded data value. This code can correct errors of $S = \{\pm 2^i; 0 \leq i \leq 39\}$; hence, it can correct an erroneous quantized output of one logical row if that error is affects at most a single bit. For greater error correction capability, we can consider burst error correction codes where the syndromes can be 2 bits. The burst error correction code for 2 bits can correct all errors of $S = \{\pm 2^i\} \cup \{\pm(2^i + 2^{i+1})\}$, up to a quantization error of 3 in one physical row per logical row. These larger AN codes, however, are generally not as efficient as the A = 79 and similar AN codes. While the A = 79 code uses every single residual in A, an AN code that can correct up to a 2 bit quantization error wastes approximately 15% of the residuals. As discussed by Mandelbaum [56], AN codes that can correct multiple, uncorrelated errors require A values that are impractically large.

Experiments with different A values reveal three limitations of the AN codes. First, larger codes can actually reduce reliability. Since each physical row is a potential source of error, there is a tradeoff between the correction capability and the increased error susceptibility of a logical row. This tradeoff exacerbates the decrease in residue efficiency as the number of syndromes increases, and creates an upper bound on the correction capability of the system. Second, as mentioned in Section II-D, the AN codes described here do not provide detection capability. If the code encounters a syndrome, it blindly attempts to correct the error, and in the process, may make the error even worse. Consider an A = 79 code where the value 1024 is encoded, which results in an encoded value of 80896. Assume that the encoded value suffers a single digit quantization error in the least and the fourth least significant bits for a syndrome of the digit 9. The decoded value will be -12249, which differs from the correct value by more than the uncorrected value. Third, even when using a distance 3 code such as A = 79 for 32 bit integers, the bit overhead of the code is larger than that of SECDED Hamming codes. An A = 79 code requires 7 bits of ECC to correct a single bit error, a 22% overhead, compared to the to the 12.5% overhead of a 64,72 Hamming code. Using the table of distance 3 codes in [57], we see that a distance 3 code for 64 bit operands would require 8 bits, while a 16 bit operand would require 6 check bits, for overheads of 12.5% and 37.5%, respectively. To remedy these problems, we propose *data-aware ABN Codes*.

B. Data-Aware ABN Codes

The AN codes discussed above implicitly assume that all of the physical rows within a logical row are equally susceptible to error, and equally important to the system

level accuracy of a DNN. This assumption is incorrect for the memristor error model discussed in Section II-C. In particular, RTN errors, which are the dominant source of error in memristive accelerators [23], have a strong state dependence. Moreover, since each row represents a single bit position, the data residing at the most significant physical row are much more important to the accuracy of the computation. By considering the properties of the underlying data, we can increase the effective correction capability of the system without an undue storage overhead.

1) Allocating Syndromes to Minimize the Error Probability: As discussed in Section II-D, an AN correction scheme can be defined in terms of an A value and a correction table that maps the residuals to syndromes. For each physical row, we compute the probability of that row being in error due to RTN effects, and select the worst case physical row from the array. These probabilities are then sorted and combined to form 2, 3, and 4 physical row combinations until the probability of a combination falls outside of the total number of available syndromes. Finally, each combination is weighted by the bit position of the most significant bit in the combination to form a final *error score*. This process, going from a matrix to the error list, is shown in Figure 8.

To generate the syndrome table, we iterate over the sorted list of possible error scores and compute the residual mod *A* of each syndrome. If the residual is unique, it is added to the syndrome table and the process continues until the list is exhausted or the syndrome table is full. The resulting table can correct both single and multi-bit errors based on the computed probability.

This approach naturally extends to yield or endurance failures. A cell stuck at the incorrect state has an error probability of 1 when that cell is active. This means that when forming the syndrome table, each RTN error is combined with the error caused by the stuck-at fault. However, errors that occur when the stuck cell is not being multiplied by the input vector may remain uncorrected. We therefore split the table into two halves, one that considers the stuck-at faults, and one that does not. This degrades the overall correction capability of the table; however, it also allows the array to operate in the presence of stuck-at faults.

2) Constant Overhead, Multi-Operand Correction: Due to the correction condition in Equation 1, the number of check bits required relative to the operand size falls as the operand size increases. To simplify the hardware, we would instead like constant overhead regardless of the underlying number of bits. Additionally, we would like large operand sizes to reduce the storage overhead of the code. To satisfy these requirements, we propose multiple-operand correction.

A coded operand AN' is defined by first concatenating the underlying operands, and then multiplying them by the

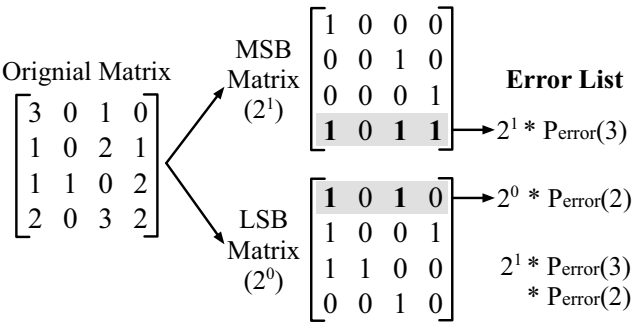

Figure 8. The error list for a 2 bit 4x4 matrix.

selected A value. Mathematically, this can be represented as

$$AN' = A \times \sum_{i=0}^{op} (2^{i \times b} * N_i) \quad (2)$$

where op is the number of operands in a coded operand, and b is the number of bits per individual operand. The multiplication by $2^{i \times b}$ is equivalent to shifting the operands so that they do not overlap.

It may appear as though the process of multiplying the multiple operand block by *A* could create errors due to operands that cause carries; however, this is not an issue so long as the correction and decoding is done on the full coded block and split into the underlying sums after the block has been decoded. For the purposes of syndrome allocation, we consider all of the physical rows within a coded group as making up one logical row; however, the bit weight is computed based on the bit position within the individual operands.

3) Detection Capability: In Section II-D, we noted that an *A*=3 code is comparable to a simple parity bit scheme since it can detect single bit position errors via a single check bit. Just as single error correcting Hamming codes can be augmented with an additional bit to form SECDED ECC, we can augment the AN codes with an additional *B* value to form *ABN codes*—a new family of codes similar to the bi and multiresidue codes proposed by Rao [54], [55]. In ABN codes, *B* is a small prime number that is multiplied by *A* to form the multiplier for the code. At decode time, the *A* value is first used to correct an error as discussed above. After the correction, the detection is performed on the corrected value using *B*.

4) Selecting A*:* As discussed in Section II-D, there is no known way to find *A* for a set of syndromes without searching over all candidate *A*s. Furthermore, the underlying bit counts in each physical row themselves rely on *A*, creating a circular dependence. Since existing schemes for memristive accelerators perform a moderate to substantial amount of matrix preprocessing to set up the data arrays and require a large number of memristors to be written with high accuracy [1], [9], [23], we search over the space of *A*s to determine which one minimizes the error probability

using the RTN error threshold. We define the set of candidate *A*s to be all odd numbers that can be represented by the number of check bits available. The maximum candidate *A* is divided by *B* so that the number of check bits required is not exceeded. We select the A that results in the table with the greatest error correction capability.

5) Predicting Row Error Rate: To predict the row error rate, we consider a simple model of a physical row as a series of parallel resistors. This simple model assumes that the input vector comprises all **1**s, which represents the worst case error susceptibility for a row. Using this model, we compute the error free current for a physical row state, defined by the number of resistors in each state. Due to the non-zero conductance of the devices in the R_{HI} state, the error free current will not be perfectly centered on the ideal current. Additionally, due to the RTN offset discussed in Section IV, the error free current will drop further below the ideal current. By comparing the error free current to the current boundaries for correct quantization, we determine the number of cells that must be in error to exceed the upper current boundary, and the number of cells not in error for the current to fall below the lower boundary. Given these counts, we can compute the error probability using a binomial CDF $B(n, k)$, where n is the total number of cells in the **1** state and k is the error threshold computed above.

Alternatively, the aforementioned process can be replaced by testing and characterization of rows in fabricated systems. Characterization may also allow the row error rate prediction to account for local device variation, increasing the prediction accuracy by capturing effects that are not included in the idealized model. The important point is the mapping of the row state to the error probabilities for data aware computation, rather than the exact method of determining the error rates.

VI. Implementation of AN Codes

One important goal of the proposed error correction scheme is transparent compatibility with existing memristive accelerators. As discussed in Section II-B2, prior work on *in-situ* MVM for neural networks splits the computation into blocks made up of multiple arrays, called submats by MBM, and IMAs by ISAAC. (For simplicity, we adopt the ISAAC terminology.) Each IMA contains an *error correction unit* (ECU) that consists of all of the hardware to implement the proposed AN code.

The error correction unit comprises three major components: two divide/residual units for the residual computation of *A* and *B* (one each), and a correction table that maps each residual to a syndrome. The output of the first divide/residual unit computes the integer division of the input by *A*, and outputs the residual along with the quotient. The residual is used to index into the correction table, and the value read from the correction table is added to the result. This value is then fed into the second divide/residual unit where it is

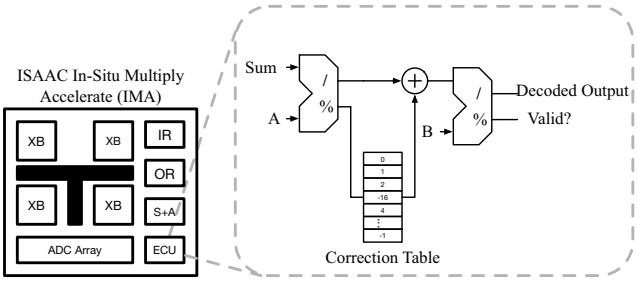

Figure 9. Error correction within an ISAAC IMA.

divided by *B*. The output of this unit is the final output of the error correction system, and includes a flag indicating if the computation was in error. The basic error correction unit is shown in Figure 9.

To reduce the overhead of the ECU, we employ three optimizations. First, we note that most of the benefit from the proposed data-aware ABN codes comes from the tables rather than the value of *A*. In fact, during the *A* search process, more than half of the IMAs select one of three A values. Instead of implementing a full divider (or a multipliplier for multiplication by division [58]), the division by *A* unit has a set of five constant *A* values that are used as the candidate *A*s (rather than all odd numbers as discussed in Section V-B4). Similarly, *B* values beyond 3 do not increase the error detection capability of the code until they become so large that they instead start significantly decreasing the correction capability. For this reason, the second divider purely divides by three, forming the equivalent of the parity bit used for SECDED ECC.

Second, if syndromes must be as long as the coded word, the correction table can become quite large. To remedy the problem, we exploit the sparsity of the syndromes, each of which contains at most four bits in error. These bit positions can be encoded as four indicies and expanded into the integer syndrome representation before being added for correction. Since each correction table is only accessed once per n cycles, where n is the number of operands per multiple operand group, this allows the table to be shared between n different IMAs with staggered accesses.

A. Handling Uncorrectable Errors

When the error correction unit outputs an uncorrectable error flag, several different steps can be taken to handle the error within the system. As discussed in Section V-A, in an uncorrectable error case, the corrected value can actually be further from the correct value than the uncorrected value. This is handled by storing a post-division-by-*B* syndrome in the correction table that can be added back to the result in the case of an uncorrectable error. This solution preserves the throughput of the system in the presence of uncorrectable errors at the possible cost of accuracy.

Another possibility is to retry the computation on an uncorrectable error. This requires modifications to the datapath

Table I
DEVICE PARAMETERS

Parameter	Value
R_{LO}	$2k\Omega$
R_{HI}	$5M\Omega$
Bits per Cell	$2-5$
V_{LO}	$0V$
V_{HI}	$0.3V$
Temperature	$350K$
Film Thickness (t_{NiO})	20nm
Film Resistivity (ρ_0)	$100\mu\Omega cm$
RTN Defect Resistivity Multiplier(α)	2
Permittivity of Film (ϵ_r)	$12\epsilon_0$
Failure Rate	0.1%

Table II
EVALUATED NEURAL NETWORKS.

Name	Network Topology	Source
MLP1	3 layer MLP 500 and 150 hidden units	[12]
MLP2	2 layer MLP 800 hidden units	[16]
CNN1	5 Layer CNN 6 5x5 feature maps 16 5x5 feature maps 120 and 84 node fully connected layers	[12]
AlexNet	8 Layer CNN 5 convolutional layers 3 fully connected layers	[64]

and comes at the cost of throughput. Retries must stall the ongoing array computation, which can disrupt the pipelining and bus scheduling. Retries also incur a substantial energy overhead since the full array must be active even if only a small number of logical rows are uncorrectable.

VII. EXPERIMENTAL SETUP

We evaluate accuracy using a custom simulator that performs Monte-Carlo simulation of *in-situ* MVM with errors.

A. Architecture

We evaluate a memristive accelerator similar to ISAAC [9], wherein each layer of the neural network is placed in one or more tiles. We use multibit memristors with 1 to 5 bits per cell, and 128x128 arrays to achieve a good balance between throughput and baseline reliability; 16 bit fixed point weights with coded operands of 128 bits; and 7 to 10 check bits. For weight matrices with more than 128 columns, we split the matrix evenly into chunks no larger than 128 columns. The data-aware AN codes consider each array separately for the purposes of selecting *A* and building the correction table.

B. Device Modeling

The simulator incorporates the error model described in Section II-C; the key parameters of the model are listed in Table I. The values of R_{LO}, R_{HI}, and V_{HI} were selected to be similar to Hu *et al.* [23] with an extended dynamic range to increase the noise margin, and by extension reduce the error susceptibility, for the baseline system. For RTN modeling, we use the parameters reported by Ielmini *et al.* [30] for a NiO memristor. Based on these parameters, we derive $\Delta R/R$ for R_{LO} and R_{HI} as 2.8% and 50%, respectively. Within the simulator, $\Delta R/R$ is calculated directly as a function of the resistance rather than using the derived $\Delta R/R$. To model the effect of fabrication defects and cell endurance failures, we include a failure rate term representing the probability that a cell will exhibit a stuck-at fault and thus cannot be programmed.

C. Circuit Modeling

We implement the error correction units in Verilog RTL and synthesize the units with the Synopsys Design Compiler [59] using FreePDK45 [60]. The results are scaled to 32nm in order to compare the hardware overhead directly to that reported by ISAAC, following the methodology published in earlier work by Stan *et al.* [61]. The error correction table is evaluated using CACTI 6.5 [62] at 32nm. For all other structures that are common with ISAAC, we use the parameters presented in the ISAAC paper.

D. Workloads

Using the TensorFlow framework [63], we train three neural networks, one CNN and two MLPs, on the MNIST data set [12]. We then extract the weight matrices for use in the developed simulator. The parameters of the neural networks are listed in Table II. The neural networks are trained on 50,000 data points from the MNIST training set, with 10,000 points used for validation. We use the MNIST dataset directly and do not apply distortions. The neural networks are trained using single precision floating point values and then converted to 16 bit fixed point integers for mapping. We use the negative value normalization and encoding from ISAAC [9] for the weights. We evaluate the accuracy of the neural networks using the developed simulator on 1000 randomly selected examples from the MNIST test set. Additionally, we evaluate the well known AlexNet CNN [64] using the pre-trained weights from the Caffe codebase [65]. Due to the performance limitations of our Monte-Carlo simulation, we evaluate AlexNet at a single 2 bit cell, 9 ECC bit design point rather than on the full spectrum of cell and ECC parameters.

VIII. EVALUATION

This section presents the the accuracy, energy, and area characteristics of the proposed ECC scheme.

A. Accuracy Evaluation

Figure 10 show the misclassification rate of an uncorrected memristive accelerator and a data-aware AN coded scheme using 7 to 10 bits of error correction as compared to the respective software implementation for MLP1, MLP2, and CNN1 over a range of bits per cell. We evaluate 7 schemes with different levels of overhead. The Static16, and Static128 codes are the naïve AN codes described in Section V-A for 16 and 128 bit (grouped) operands augmented with a *B*=3 check term. These codes are designed to correct an error at exactly one bit position; if the code detects an error post-correction using the B value, it reverts to the detected value. Data-aware ABN error correction schemes are specified as ABN-X, where X is the total

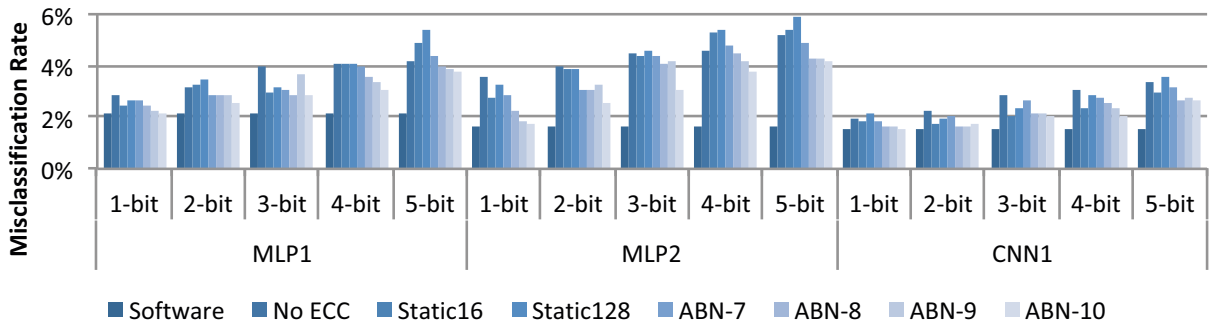

Figure 10. Misclassification rate of MLP1, MLP2, and CNN1 for different bit widths.

number of ECC bits. *A* is selected from a set of 5 candidates, and the contents of the correction table are determined for each array. *B* is 3 for all dynamic codes and reverts to the uncorrected value on a detected error.

These results show the limitations of the naïve application of AN codes. At lower numbers of bits per cell, the static AN codes achieve results that are competitive with dynamic codes. As the number of bits increases, the outputs begin to have multiple bit positions in error, and accuracy converges to that of the uncorrected baseline. Even at these lower bit levels, the Static16 code uses 48 check bits (6b per operand) as opposed to the 7 to 10 of the dynamic schemes. Attempting to avert the problem through multiple operand codes without applying the proposed data-aware scheme, shown as Static128, significantly increases the misclassification rate.

The errors increase the misclassification rates from 1-2% in the pure software case to 3-4% for the MLPs, and 2-3% for the CNN. It is important to place the accuracy differences seen here in context: **while a 1% reduction in the MNIST misclassification rate for a MLP may appear small, this is an improvement comparable to that achieved by data-skewing to increase the training set size, doubling the number of hidden units, or using a CNN rather than an MLP [12], [16]. Given the high accuracy of MNIST where a single layer linear classifier can achieve less than 8% misclassification rate [12] and current state of the art neural networks achieve misclassification rates less than 0.5%, misclassification rate improvements of 1% are considered highly significant [66].**

The results demonstrate the two possible uses for the proposed error correction scheme. First, the proposed scheme can be applied to reduce the raw error rate while holding the number of bits per cell constant. The data-aware AN schemes can fully eliminate the misclassifications due to array noise for 1-bit cells, and generally eliminate up to one half of the misclassifications caused by the array noise. The other possibility is to use the error correction scheme to bound the number of errors while aggressively increasing the number of bits per cell. For instance, in MLP1, a data-aware 9-bit code using 4-bit cells can provide accuracy comparable to an uncorrected array using 2-bit cells. This technique can be used to reduce the overall energy consumption and area of the system, as an eight operand group of 16 bit operands requires 35 bit slices at 4-bits per cell, compared to 64 bits for eight unprotected operands using 2-bits per cell.

Table III
ALEXNET ACCURACY.

	Software	Uncorrected	ABN-9
Top 1 Misclassification	42.96%	48.3%	43.9%
Top 5 Misclassification	19.74%	21.3%	20.1%

MLP2 shows reduced error resilience as compared to MLP1 despite the greater number of hidden units, and higher baseline accuracy. This reduction is due to the single hidden layer of MLP2 as compared to the multiple hidden layers of MLP1. The second hidden layer of MLP1 allows it to perform classification using features that are several steps removed from the incoming data, and are somewhat skew-invariant. An even more pronounced version of this effect can be seen in CNN1 where the misclassification increase caused by the array noise from errors is less than 1%. This is in part caused by the lower number of active elements in each row of the convolutional layers, as well as the features that are less directly tied to the input values.

1) Accuracy Evaluation with Cell Faults: Figure 11 show the same networks analyzed with the addition of stuck-at faults. We again see the limitations of the AN codes without data-aware mapping: even the Static16 code exhibits worse results than the data-aware variants across the bit range, despite the significantly higher overhead. Conceptually, each code has a certain error tolerance before its ability to correct falls off dramatically, and the misclassification rate converges toward the uncorrected baseline. The addition of cell faults reduces the tolerance threshold, and can be seen in the MLP1 results. In the simulation without stuck-at faults, MLP1 shows correction capability that generally scales with the number of correction bits as the number of bits per cell is increased. However, when moving from 4- to 5-bit cells with stuck-at faults, the ABN-8, ABN-9, and ABN-10 results all start to approach the uncorrected baseline.

2) AlexNet Evaluation: To demonstrate the scalability of the proposed scheme to larger networks, we evaluate the well known AlexNet eight layer CNN. The accuracy results

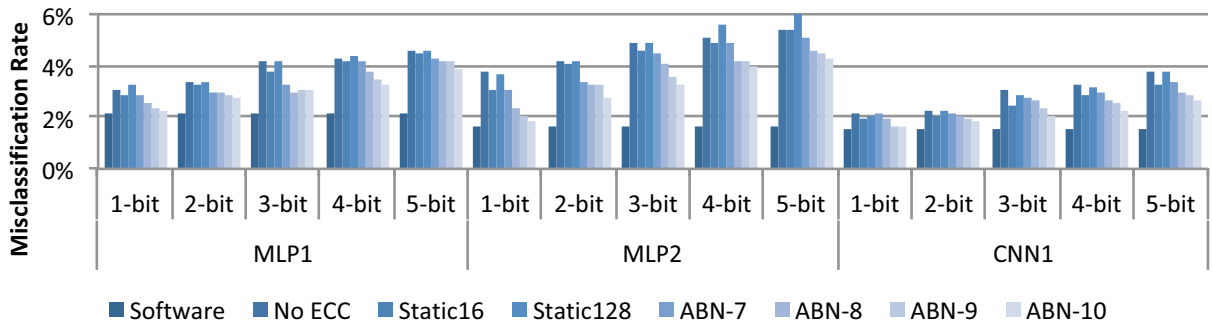

Figure 11. Misclassification rate of MLP1, MLP2, and CNN1 with cell faults for different bit widths.

Table IV
POWER AND AREA OF THE 9 BIT ERROR CORRECTION HARDWARE.

Component	Area	Power
Error Correction Unit (ECU)	0.0031 mm^2	1.42mW
Error Correction Table	0.0012 mm^2	0.51 mW

are summarized in Table III. The results show that errors are an even larger problem on more challenging neural network applications than on the small MNIST examples evaluated above, with over a 5% increase in misclassification rate for top 1 classifications. To place these results in context, **a 1% improvement in top 5 misclassification was the difference between GoogleNet [14] and VGG [15], respectively the winner and the runner up in the ILSVRC-2014 competition [13].** Further, the results demonstrate that the proposed data-aware ABN codes can be applied to networks with 60 million parameters, compared to the approximately 600K in the MNIST networks, and still provide significant accuracy improvements.

B. Area, Power, and Latency Overheads

This section evaluates the overhead of the proposed error correction scheme. Table IV shows the overhead of the proposed ECU. We compute full power and area overheads considering not only the ECU, but also the additional rows and the peripheral circuitry required by the check bits.

1) Area: The ECU alone requires a 3.4% overhead on top of an ISAAC tile designed to handle 16 bit operands with 2 bits per cell. The additional 9 bits per 128 adds an additional 7% overhead to the ADCs, DACs, and memristor arrays. Taken together, the error correction scheme requires a 6.3% area overhead per tile. This overhead corresponds to a 5.3% increase in the overall area of the memristive IC.

2) Power: The ECU requires a 2.1% power overhead on top of a tile. Handling the additional bits bring the total chip-wide power increase for the accelerator to 5.8%.

3) Throughput: Since the ECU is fully pipelined, it can be integrated directly into previously proposed accelerators without reducing the system throughput. ISAAC, for instance, asserts that there are no structural hazards and that the throughput is fully deterministic.

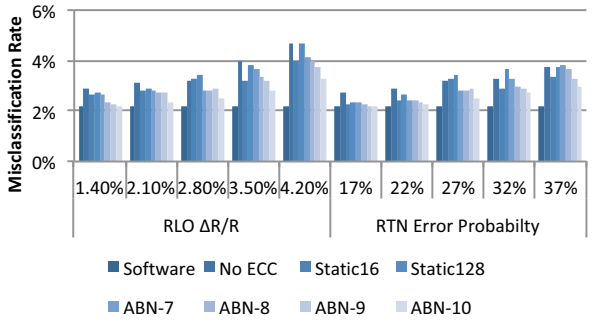

Figure 12. Misclassification rate of MLP1 with different R_{LO} $\Delta R/R$ and RTN error probability.

C. Sensitivity to RTN Parameters

We conduct sensitivity studies on the R_{LO} $\Delta R/R$ (R_{HI} $\Delta R/R$ is fixed at 50%), and the probability of a cell being in the RTN error state. The $\Delta R/R$ values considered here can be achieved through different combinations of device parameters. All of the tests are performed using 2b cells.

Figure 12 shows how R_{LO} $\Delta R/R$ (left) and RTN error probability (right) impact MLP1 accuracy. The misclassification rate is more sensitive to the $\Delta R/R$ rather than the error probability, suggesting directions for future device innovations. Furthermore, for cells with less aggressive RTN parameters, ABN-10 can fully restore the network accuracy loss caused by RTN errors with multibit cells.

IX. CONCLUSIONS

Memristive accelerators for MVM offer substantial performance and energy over conventional systems. Error correction has not been explored in these systems, and represents a major bottleneck even in neural networks, a relatively noise tolerant application. We present data aware AN-codes, the first error correction system for *in-situ* MVM that can largely restore the accuracy loss caused by the noise. Data aware AN-codes leverage the noise properties and data layout of *in-situ* MVM to increase the accuracy with a moderate overhead. The results point to important future directions in the design of memristive accelerators to tolerate errors.

ACKNOWLEDGMENTS

This work was supported in part by NSF grant CCF-1533762.

REFERENCES

[1] M. N. Bojnordi and E. Ipek, "Memristive boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning," in *Intl. Symp. on High Performance Computer Architecture (HPCA)*, March 2016.

[2] P. Sheridan, C. Du, and W. D. Lu, "Feature extraction using memristor networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 27, no. 11, pp. 2327–2336, Nov. 2015.

[3] P. Sheridan, W. Ma, and W. Lu, "Pattern recogntion with memristor networks," in *Intl. Symp. on Circuits and Systems (ISCAS)*, May 2014.

[4] B. E. Boser, E. Sackinger, J. Bromley, Y. Le Cun, and L. D. Jackel, "An analog neural network processor with programmable topology," *Journal of Solid-State Circuits*, 1991.

[5] R. Genov and G. Cauwenberghs, "Charge-mode parallel architecture for matrix-vector multiplication," in *Midwest Symposium on Circuits and Systems (MWSCAS)*, Aug. 2000.

[6] ——, "Kerneltron: Support vector "machine" in silicon," *IEEE Transactions on Neural Networks*, vol. 15, no. 5, pp. 1426–1434, Sept. 2003.

[7] F. Kub, K. Moon, I. Mack, and F. Long, "Programmable analog vector-matrix multipliers," *IEEE Journal of Solid-State Circuits*, vol. 25, no. 1, Feb 1990.

[8] K. K. Moon, F. J. Kub, and I. A. Mack, "Random address 32x32 programmable analog vector-matrix multiplier for artificial neural networks," in *IEEE Custom Integrated Circuits Conference*, May 1990, pp. 26.7/1–26.7/4.

[9] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *Intl. Symp. on Computer Architecture (ISCA)*, June 2016.

[10] P. Chi, S. Li, S. Li, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory," in *Intl. Symp. on High Performance Computer Architecture (HPCA)*, June 2016.

[11] L. Song, X. Qian, H. Li, and Y. Chen, "PipeLayer: A pipelined ReRAM-based accelerator for deep learning," in *Intl. Symp. on High Performance Computer Architecture (HPCA)*, Feb 2017.

[12] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov 1998.

[13] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[14] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Computer Vision and Pattern Recognition (CVPR)*, 2015.

[15] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014.

[16] P. Y. Simard, D. Steinkraus, and J. C. Platt, "Best practices for convolutional neural networks applied to visual document analysis," in *Intl. Conference on Document Analysis and Recognition (ICDAR)*, 2003.

[17] A. Avižienis, "A set of algorithms for a diagnosable arithmetic unit," Jet Propulsion Laboratory, California Institute of Technology, Tech. Rep. No. 32-546, March 1964.

[18] ——, "Design of fault-tolerant computers," in *AFIPS Fall Joint Computer Conference*, April 1967.

[19] W. W. Peterson and E. J. W. Jr., *Error-Correcting Codes*, 2nd ed. Cambridge, Massachusetts: MIT Press, 1972.

[20] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "DaDianNao: A machine-learning supercomputer," in *Intl. Symp. on Microarchitecture (MICRO)*, 2014.

[21] D. Mahajan, J. Park, E. Amaro, H. Sharma, A. Yazdanbakhsh, J. K. Kim, and H. Esmaeilzadeh, "TABLA: A unified template-based framework for accelerating statistical machine learning," in *Intl. Symp. on High Performance Computer Architecture (HPCA)*, March 2016.

[22] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *Intl. Symp. on Computer Architecture (ISCA)*, 2017.

[23] M. Hu, J. P. Strachan, Z. Li, E. M. Grafals, N. Davila, C. Graves, S. Lam, N. Ge, J. J. Yang, and R. S. Williams, "Dot-product engine for neuromorphic computing: Programming 1T1M crossbar to accelerate matrix-vector multiplication," in *Design Automation Conference (DAC)*, June 2016.

[24] J. B. Johnson, "Thermal agitation of electricity in conductors," *Phys. Rev.*, vol. 32, pp. 97–109, Jul 1928.

[25] H. Nyquist, "Thermal agitation of electric charge in conductors," *Phys. Rev.*, vol. 32, pp. 110–113, Jul 1928.

[26] D. Soudry, D. D. Castro, A. Gal, A. Kolodny, and S. Kvatinsky, "Memristor-based multilayer neural networks with online gradient descent training," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 26, no. 10, pp. 2408–2421, Oct 2015.

[27] S. Ambrogio, S. Balatti, A. Cubeta, A. Calderoni, N. Ramaswamy, and D. Ielmini, "Understanding switching variability and random telegraph noise in resistive RAM," in *International Electron Devices Meeting (IEDM)*, Dec 2013.

[28] ——, "Statistical fluctuations in HfOx resistive-switching memory: Part II—random telegraph noise," *IEEE Transactions on Electron Devices*, vol. 61, no. 8, pp. 2920–2927, Aug 2014.

[29] S. Choi, Y. Yang, and W. Lu, "Random telegraph noise and resistance switching analysis of oxide based resistive memory," *Nanoscale*, vol. 6, pp. 400–404, 2014.

[30] D. Ielmini, F. Nardi, and C. Cagli, "Resistance-dependent amplitude of random telegraph-signal noise in resistive switching memories," *Applied Physics Letters*, vol. 96, no. 5, 2010.

[31] D. Lee, J. Lee, M. Jo, J. Park, M. Siddik, and H. Hwang, "Noise-analysis-based model of filamentary switching ReRAM with ZrO_x/HfO_x stacks," *IEEE Electron Device Letters*, vol. 32, no. 7, pp. 964–966, July 2011.

[32] R. Soni, P. Meuffels, A. Petraru, M. Weides, C. Kgeler, R. Waser, and H. Kohlstedt, "Probing Cu doped Ge0.3Se0.7 based resistance switching memory devices with random telegraph noise," *Journal of Applied Physics*, vol. 107, no. 2, 2010.

[33] M. Terai, Y. Sakotsubo, Y. Saito, S. Kotsuji, and H. Hada, "Effect of bottom electrode of reram with Ta_2O_5/TiO_2 stack on RTN and retention," in *IEEE International Electron Devices Meeting (IEDM)*, Dec 2009, pp. 1–4.

[34] ——, "Memory-state dependence of random telegraph noise of Ta_2O_5/TiO_2 stack ReRAM," *IEEE Electron Device Letters*, vol. 31, no. 11, pp. 1302–1304, Nov 2010.

[35] Y. H. Tseng, W. C. Shen, C.-E. Huang, C. J. Lin, and Y.-C. King, "Electron trapping effect on the switching behavior of contact RRAM devices through random telegraph noise analysis," in *Intl. Electron Devices Meeting (IEDM)*, Dec 2010.

[36] D. Veksler, G. Bersuker, L. Vandelli, A. Padovani, L. Larcher, A. Muraviev, B. Chakrabarti, E. Vogel, D. C. Gilmer, and P. D. Kirsch, "Random telegraph noise (RTN) in scaled RRAM devices," in *Intl. Reliability Physics Symp. (IRPS)*, April 2013.

[37] F. Alibart, L. Gao, B. D. Hoskins, and D. B. Strukov, "High precision tuning of state for memristive devices by adaptable variation-tolerant algorithm," *Nanotechnology*, vol. 23, no. 7, Jan. 2012.

[38] D. Niu, Y. Xiao, and Y. Xie, "Low power memristor-based ReRAM design with error correcting code," in *Asia and South Pacific Design Automation Conference (ASPDAC)*, Jan 2012.

[39] C. Xu, D. Niu, N. Muralimanohar, N. P. Jouppi, and Y. Xie, "Understanding the trade-offs in multi-level cell ReRAM memory design," in *Design Automation Conference (DAC)*, May 2013.

[40] P. J. Nair, D.-H. Kim, and M. K. Qureshi, "ArchShield: Architectural framework for assisting DRAM scaling by tolerating high error rates," in *Intl. Symp. on Computer Architecture (ISCA)*, June 2013.

[41] M. Abdallah, H. Mostafa, and M. Fathy, "Yield maximization of TiO2 memristor-based memory arrays," in *Intl. Conference on Microelectronics (ICM)*, Dec 2014.

[42] X. Y. Xue, W. X. Jian, J. G. Yang, F. J. Xiao, G. Chen, X. L. Xu, Y. F. Xie, Y. Y. Lin, R. Huang, Q. T. Zhou, and J. G. Wu, "A 0.13 um 8Mb logic based CuxSiyO resistive memory with self-adaptive yield enhancement and operation power reduction," in *Symp. on VLSI Circuits*, June 2012.

[43] C. H. Cheng, A. Chin, and F. S. Yeh, "Novel ultra-low power RRAM with good endurance and retention," in *Symp. on VLSI Technology*, June 2010.

[44] C. W. Hsu, I. T. Wang, C. L. Lo, M. C. Chiang, W. Y. Jang, C. H. Lin, and T. H. Hou, "Self-rectifying bipolar TaOx/TiO2 RRAM with superior endurance over 1012 cycles for 3D high-density storage-class memory," in *Symp. on VLSI Technology*, June 2013.

[45] J. Fan, S. Jiang, J. Shu, Y. Zhang, and W. Zhen, "Aegis: Partitioning data block for efficient recovery of stuck-at-faults in phase change memory," in *Intl. Symp on Microarchitecture (MICRO)*, 2013.

[46] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable DRAM alternative," in *Intl. Symp. on Computer Architecture (ISCA)*, June 2009.

[47] R. Melhem, R. Maddah, and S. Cho, "RDIS: A recursively defined invertible set scheme to tolerate multiple stuck-at faults in resistive memory," in *Intl. Conference on Dependable Systems and Networks (DSN)*, June 2012.

[48] S. Schechter, G. H. Loh, K. Strauss, and D. Burger, "Use ECP, not ECC, for hard failures in resistive memories," in *Intl. Symp. on Computer Architecture (ISCA)*, June 2010.

[49] L. Xia, M. Liu, X. Ning, K. Chakrabarty, and Y. Wang, "Fault-tolerant training with on-line fault detection for RRAM-based neural computing systems," in *Design Automation Conference (DAC)*, June 2017.

[50] A. Avižienis, "Arithmetic error codes: Cost and effectiveness studies for application in digital system design," *IEEE Transactions on Computers*, vol. C-20, no. 11, pp. 1322–1331, Nov. 1971.

[51] C. Fetzer, U. Schiffel, and M. Süßkraut, "AN-encoding compiler: Building safety-critical systems with commodity hardware," in *Intl. Conference on Computer Safety, Reliability, and Security (SAFECOMP)*, Sept. 2009.

[52] U. Schiffel, "Hardware error detection using AN-codes," Ph.D. dissertation, Technische Universität Dresden, 2011.

[53] U. Schiffel, A. Schmitt, M. Süßkraut, and C. Fetzer, "ANB- and ANBDmem-encoding: detecting hardware errors in software," in *Intl. Conference on Computer Safety, Reliability, and Security (SAFECOMP)*, Sept. 20010.

[54] T. R. Rao, "Biresidue error-correcting codes for computer arithmetic," *IEEE Transactions on Computers*, vol. 19, no. 5, pp. 398–402, May 1970.

[55] T. R. Rao and O. N. Garcia, "Cyclic and mmltiresidue codes for arithmetic operations," *IEEE Transactions on Information Theory*, vol. 17, no. 1, pp. 85–91, Jan 1971.

[56] D. Mandelbaum, "Arithmetic codes with large distance," *IEEE Transactions on Information Theory*, vol. 13, no. 2, pp. 237–242, Apr. 1967.

[57] D. T. Brown, "Error detecting and correcting binary codes for arithmetic operations," *IRE Transactions on Electronic Computers*, vol. EC-9, no. 3, pp. 333–337, Sept 1960.

[58] J. Henry S. Warren, *Hacker's Delight*, 2nd ed. Upper Saddle River, New Jersey: Pearson Education, 2012.

[59] Synopsys, "Synopsys Design Compiler User Guide," http://www.synopsys.com/\Tools/Implementation/RTLSynthesis/DCUltra/Pages/.

[60] "Open-Access-based PDK for 45nm Technology Node," http://www.eda.ncsu.edu/wiki/FreePDK.

[61] M. R. Stan, K. Skadron, W. Huang, and K. Rajamani, "Scaling with design constraints: Predicting the future of big chips," *IEEE Micro*, vol. 31, no. 4, pp. 16–29, 2011.

[62] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A Tool to Model Large Caches," HP Laboratories, Tech. Rep. HPL-2009-85, 2009.

[63] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[64] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Intl. Conference on Neural Information Processing Systems (NIPS)*, Dec 2012.

[65] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," Nov 2014.

[66] L. Wan, M. Zeiler, S. Zhang, Y. Lecun, and R. Fergus, "Regularization of neural networks using dropconnect," in *Intl. Conference on Machine Learning (ICML)*, June 2013.