Structural Coding: A Low-Cost Scheme to Protect CNNs from Large-Granularity Memory Faults

Ali Asgari Khoshouyeh
University of British Columbia
Vancouver, Canada
aasgarik@alum.ubc.ca

Florian Geissler
Intel Labs
Munich, Germany
florian.geissler@intel.com

Syed Qutub
Intel Labs
Munich, Germany
syed.qutub@intel.com

Michael Paulitsch
Intel Labs
Munich, Germany
michael.paulitsch@intel.com

Prashant J. Nair
University of British Columbia
Vancouver, Canada
prashantnair@ece.ubc.ca

Karthik Pattabiraman
University of British Columbia
Vancouver, Canada
karthikp@ece.ubc.ca

ABSTRACT

The advent of High-Performance Computing has led to the adoption of Convolutional Neural Networks (CNNs) in safety-critical applications such as autonomous vehicles. However, CNNs are vulnerable to DRAM errors corrupting their parameters, thereby degrading their accuracy. Existing techniques for protecting CNNs from DRAM errors are either expensive or fail to protect from large-granularity, multi-bit errors, which occur commonly in DRAMs.

We propose a software-implemented coding scheme, *Structural Coding (SC)* for protecting CNNs from large-granularity memory errors. SC achieves three orders of magnitude reduction in Silent Data Corruption (SDC) rates of CNNs compared to no protection. Its average error correction coverage is also *significantly* higher than other software techniques to protect CNNs from faults in the memory. Further, its average performance, memory, and energy overheads are respectively 3%, 15.71%, and 4.38%. These overheads are much lower than other software protection techniques.

CCS CONCEPTS

• **Computer systems organization → Embedded systems**; *Processors and memory architectures*; • **Hardware → Error detection and error correction**; **System-level fault tolerance**; • **Computing methodologies → Neural networks**.

KEYWORDS

Memory faults, Deep Neural Networks, Error Correction

ACM Reference Format:
Ali Asgari Khoshouyeh, Florian Geissler, Syed Qutub, Michael Paulitsch, Prashant J. Nair, and Karthik Pattabiraman. 2023. Structural Coding: A Low-Cost Scheme to Protect CNNs from Large-Granularity Memory Faults. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3581784.3607084

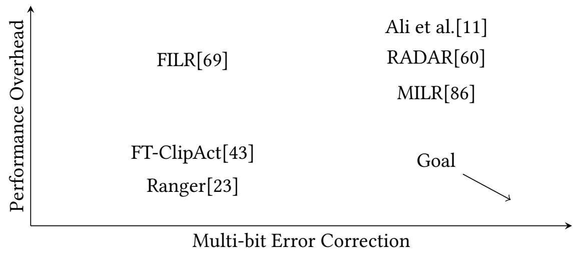


Figure 1: Existing software correction techniques with their performance overhead on the Y-axis and their correction coverage of multi-bit errors on the X-axis. We aim to design a software correction technique with low performance overhead and high multi-bit error correction capability.

1 INTRODUCTION

Among Machine Learning (ML) workloads in High-Performance Computing (HPC) [84], a wide range of safety-critical systems, such as Autonomous Vehicles (AVs), relies on the combination of Convolutional Neural Networks (CNNs) and Dynamic Random Access Memories (DRAMs)[4, 7]. However, DRAMs are susceptible to soft errors, leading to failures. Studies reveal that DRAM modules alone have a raw Failure In Time (FIT) rate [5, 58] significantly exceeding the recommended FIT rate of 10 (for the entire chip) by AV safety standards such as ISO 26262 [6]. For example, at the scale of a million operating AVs with a DRAM FIT rate of 50, the mean time to failure will be less than a day. These soft errors encompass both single-bit and multi-bit faults, with comparable occurrence frequencies and spanning multiple words [15, 58, 101, 102]. We call multi-bit faults spanning multiple words as *large-granularity faults. This paper targets soft errors that result in large-granularity faults.*

Traditionally, hardware-based Error Correction Coding (ECC) has been used to mitigate DRAM faults. However, the widely used ECC, known as SEC-DED (Single Error Correction, Double Error Detection), can only correct single-bit faults. Only variations of Chipkill [27] can correct both single-bit and multi-bit faults. Unfortunately, Chipkill is twice as expensive as SEC-DED[77], can lead to up to 4× higher memory bandwidth overheads [107], and imposes a significant energy burden [20, 49, 50]. Given the significance of metrics like cost, performance, and energy consumption, especially in domains like AV design [4, 111], Chipkill becomes undesirable.

In response, researchers have introduced software-based ECC approaches to mitigate DRAM faults. Fig. 1 shows a representative set of these techniques. Unfortunately, these proposals either neglect to address large-granularity faults [23, 43, 60, 69], or they incur substantial performance [11] or latency [86] overheads (as shown in Section 7). We aim to develop a software-based technique that corrects multi-bit faults in DRAM, including large-granularity faults. However, there are two challenges:

(1) As real-time applications use CNNs [92], the performance overhead of our technique must be low. Minimizing performance overheads is especially challenging when protecting the large body of CNN parameters stored in DRAM [97].
(2) The memory footprint and bandwidth overhead of our technique must be low, as they increase energy consumption and performance overhead. Keeping these overheads low is challenging for heterogeneous errors, ranging from those that spread across memory to localized clusters.

To address the above challenges, we leverage the property that the CNN parameter values are distributed around zero [61]. We propose a single floating-point summation as a checksum that provides effective and efficient detection (Section 5.2.7). The efficiency of the detection mitigates the performance overhead of the common case of having no faults. We design a coding scheme based on real-number codes [12] as they provide a flexible strategy with a trade-off between overhead and capability. Moreover, they have an adjustable symbol size, allowing for efficiently correcting bursts of multi-bit faults. We call our coding scheme *Structural Coding (SC)*, as it is based on the *structure* of the CNN.

To the best of our knowledge, SC is the first technique to provide strong error correction for CNN parameters against large-granularity multi-bit faults in DRAMs (a) without requiring any hardware support and (b) while incurring low performance and memory overheads. Additionally, existing CNN applications can use Structural Coding requiring neither significant developer effort, nor network retraining, thus making it practical.

We make four key contributions in this paper:

(1) We model a broad set of real-world memory failure modes, from small-granularity errors (bit-flips) to large-granularity errors (e.g., row failures), to study the effect of memory faults on ML applications. We consider a conservative, extrapolated bit-error rate in future DRAMs based on publicly reported memory error trends (Section 3).
(2) We propose a novel error correction technique, *Structural Coding* (SC), for CNNs that recovers from multi-bit memory errors (Section 4). We optimize our technique to keep the performance overhead low and the memory overhead sub-linear with the size of the CNN while providing robust protection (comparable to high-overhead techniques such as keeping three copies in the same memory device) (Section 4.4).
(3) We implement SC as an automated tool for CNNs developed with PyTorch [83], enabling seamless adoption of SC without requiring developer effort or retraining (released publicly[1]).
(4) We evaluate SC on a system with Intel(R) Core^{TM} i7 CPU. We measure SC's error-correction coverage, performance overhead, memory overhead, and energy overhead. We compare these metrics to four state-of-the-art software-based error correction techniques on six CNN classifiers and hardware-based Chipkill.

[1]https://github.com/DependableSystemsLab/structural-coding

From our experiments, we find that SC achieves *up to 1000× lower* Silent Data Corruption (SDC) rates of CNNs under a wide variety of fault models compared to an unprotected network. SC also provides a higher average coverage than prior techniques. Furthermore, based on experiments on a machine with Intel(R) Core^{TM} i7 CPU, SC's average memory footprint overhead is 15.71%, and its average energy consumption overhead is 4.38%. Additionally, the average performance overhead of SC is only 2.07% in the typical case of having no faults. These performance, energy, and memory overheads are lower than other protection techniques, despite SC providing *higher coverage* for additional fault types. Unlike prior techniques, we find that SC provides high coverage even at high bit-error rates. Finally, we find that the average energy overhead of SC is about 3.4x lower than commercial hardware Chipkill implementations.

2 BACKGROUND AND MOTIVATION

2.1 DRAM-based Memory Systems

Modern main-memory systems are primarily composed of *modules* placed on a *channel* [47, 48]. These modules may contain *multiple ranks* of *DRAM chips*. Each DRAM chip is internally split into several *banks*. These banks are organized into *rows* and *columns*. Each row contains many *words*, which consist of bits of DRAM cells. All DRAM chips operate in tandem, and the memory controller manages the channel.

Figure 2 shows the different components within a DRAM-based memory system and the types of multi-bit faults that could occur within a DRAM chip. These faults can occur across bits, words, rows, columns, and banks.

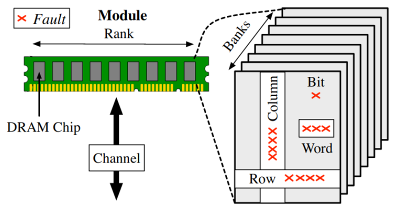


Figure 2: DRAM modules and fault modes - these can be Bit, Word, Row, Column, Bank, and Rank faults.

2.2 Memory Faults in DRAM systems

Memory faults (permanent or transient) harm CNN applications [13, 31, 67]. We consider errors in the CNN's trained parameters that remain in the DRAM and are used for multiple inferences, and cause misclassifications. These faults will *continue* to affect the accuracy of inference for ML models until the correct values overwrite these faulty values. Consider the example of an AV. Under a memory fault, the system may have a wrong steering angle for an extended time (until the faulty value is overwritten). Thus memory faults may cause AVs to drive haphazardly, which can be a safety hazard.

Memory systems encounter both single-bit and multi-bit faults [29]. While SEC-DED-based ECC memory can correct single-bit faults, they cannot correct multi-bit faults [14]. The granularity of multi-bit faults varies – they could be large clusters of faulty bits as words, rows, or columns of data. Multi-bit faults are as frequent as single-bit faults and will likely become even more frequent in future [96, 101]. Recently, Beigi et al. [15] reported that multi-bit faults have remained of comparable frequency to single-bit faults across DRAM generations.

Chipkill [27] is a *hardware* ECC technique that corrects single-chip errors and is effective on large-granularity faults [102]. Unfortunately, it incurs high memory bandwidth and high costs.

2.3 Syndrome-based Error Correction

One can use traditional Syndrome-based Error Correcting Codes (ECC) [73] to mitigate errors in the CNN models. To this end, Syndrome-based ECCs create symbols using a group of bits. To protect m symbols across a total of n symbols (also called a code word), we must choose k such that $n - k = m$, where n represents the number of symbols.

Anfinson et al. [12] proposed *weighted checksums-based correction* aka *real-number codes*. In this technique, similar to traditional Syndrome-Based ECC, k weighted sums of the data word (consisting of real numbers) are calculated and stored in the code word. This coding scheme detects up to k errors and corrects up to $\frac{k}{2}$ errors. If we know the location of errors in an erasure scenario, then this scheme allows us to correct up to k errors [21]. Unfortunately, as memory systems can have different granularities of multi-bit faults, naively applying real number codes to protect against these faults would incur high overheads. Therefore, we need to tailor error correction to the structure of CNNs to achieve low overheads.

2.4 Silent Data Corruption for CNN classifiers

This work focuses on CNN classification rather than other tasks. This approach aligns with prior work [11, 23, 43, 69, 86]. Memory errors in the parameters of CNNs can cause wrong classifications without any detectable side effects. These memory errors will typically *not* cause the CNN application to crash because the parameters are part of the program data. We call these misclassifications as Silent Data Corruptions (SDC), as done by prior work [23, 69]. We use SDC as a metric because the exact values do not matter in classifiers; only the correct classifications define the model's accuracy.

2.5 Real-time needs of Machine Learning

Because Machine Learning (ML) is actively used in real-time applications [35, 66, 95, 100], it is essential to protect the memory against errors in real-time. A trivial protection strategy of reloading parameters is not viable. This is because it can violate the real-time constraints of the application. For example, assume the constraint to be that the reaction time of an AV system to take an emergency brake should be within 300 ms – comparable to humans [92]. We use the modelling approach for the fastest reaction, as suggested by Rydzewski et al. [92], to measure the impact of reloading parameters on the reaction time. Fig. 3 shows the reaction time for different protection strategies. For *Reload*, we use the time that it takes to reload CNN parameters from a Solid State Drive (SSD) storage used in a real-world AV setup [4] shown in Table 1, averaged among the vision models. For the three techniques, RADAR, MLIR and SC, we use the average performance overhead that we obtained in Section 5. As can be seen, reloading the parameters or using the

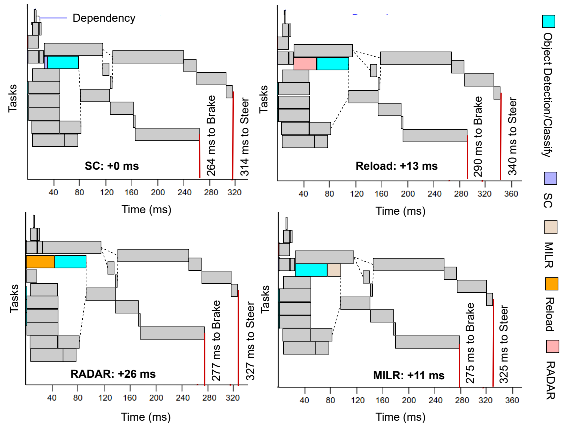


Figure 3: The reaction time to brake or steer for each technique (i.e., SC, Reloading, RADAR, and MLIR). The blocks show the time taken for an AV's operation or the overhead of the protection strategies. We highlight Object Detection/classification as the only operations affected by protection. The task durations are taken from Rydeski et al. [92].

RADAR or MLIR will increase the reaction time to above 300 ms and violate the constraint in this example.

Table 1: Dominance of weights in the ML model parameters, significance of parameters in Resident Set Size (RSS), latency of reloading them from SSDs, and kernel size range.

Model	Params (%)[a]		RSS size (MB)[a]		SSD Reload	Kernel Size
	Weights	*Other*	*Params*	*Activations*	Latency (ms)[a]	Range (KB)[a]
alexnet	99.98	0.02	245.31	14.04	**122.66**	0.35 - 9.0
squeezenet	99.68	0.32	5.64	28.50	2.82	0.02 - 0.56
mobilenet	99.33	0.67	11.57	15.56	5.79	0.01 - 1.0
googlenet	99.85	0.15	140.57	8.68	70.29	0.06 - 2.0
resnet50	99.79	0.21	103.65	57.22	51.83	0.06 - 4.5
shufflenet	99.35	0.65	6.79	12.92	3.40	0.01 - 1.0
mozafari	99.89	0.11	568.53	11.69	284.27	0.43 - 29.25
xvectors	99.69	0.31	33.75	13.17	16.88	0.12 - 2.93

[a]Rounded to 2 decimal places.

3 FAULT MODELLING

Fault Injection (FI): We use software-based FI to inject multi-bit faults into CNN applications. We use data from field studies [14, 34, 62, 79, 102] to inform our injection of multi-bit faults. A very recent study on DDR4 failures [15] also confirms these trends and finds that multi-bit faults are as frequent as single-bit faults in real-world DRAM systems and that they can span multiple words of DRAM.

We study the effect of errors only when they are manifested in the model parameters as opposed to activation inputs, or instructions, similar to what prior work [25, 69, 81] has done. We focus on CNN parameters due to three reasons:

(1) Parameters repeatedly work on multiple activation inputs. Thus, even transient faults in them can have long-term effects [59], and the contribution of model parameters to reliability dominates that of activation inputs.
(2) Parameters occupy a considerable amount of memory (e.g., 5.64 MB to 568.53 MB according to Table 1) which is much

larger than the instructions for a CNN model that typically include only several nested loops and can fit within few Kilobytes. Therefore, the contribution of model parameters to the overhead dominates that of instructions. Further, one can protect instructions by expensive techniques, such as Triple Data Redundancy (TDR), without prohibitive costs.
(3) Our technique is orthogonal to the protection methods for protecting other parts of the application memory.

Impact of Address Mapping To determine the distribution of multi-bit errors, we assume a random logical to physical page mapping [85]. Similarly, when deciding the number of corrupted bits within a memory word, we consider a random number of bits being corrupted. We assume bank interleaved mapping of pages. Address mappings tend to remain fixed. Specific address mappings of the memory controller are typically found in the official documentation or via reverse engineering [85, 114]. Further, we assume the memory organization is based on a commodity DRAM [1].

We evaluate our technique in the presence of three different multi-bit real-world failure modes [102] as follows:

- **word failures**: Flipped bits are within two consecutive bytes,
- **column failures**: word failure appears at the same 4KB random word index, on an average of 3% of 4KB pages,
- **row failures**: word errors with probability 30% occur in two randomly picked 4KB pages.

We use the numbers from Sridharan et al. [102] as detailed field studies of DRAM errors are sparse. However, our model is supported by a very recent field study [15].

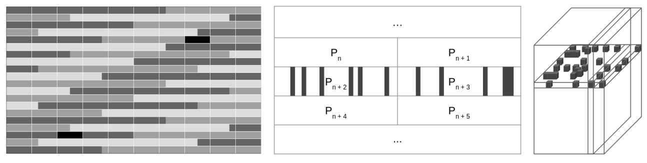


Figure 4: Row fault model, from DRAM to logical pages. Grey colours in logical pages are kernels of a convolutional layer. White borders surround logical pages. Errors in black.

For example, consider the row fault model in Fig. 4. Assuming that there are many erroneous words in a single row and there is only one rank in the DRAM module, these errors manifest in only two random pages of the model parameters, affecting two kernels.

Though our primary fault model is large-granularity memory faults, we also consider Raw Bit Error Rate (BER) to analyze the sensitivity of our technique to high bit error rates in harsh conditions. We evaluate at BERs observed at high temperatures reported in Matthew et al. [71], which can be as high as 10^{-5}. Similar to high temperatures, aging can increase BER by 50% [16, 33]. This fault model is also in line with prior work [86, 89, 109] and allows for a fair comparison. The BER fault model also captures larger granularity multi-bank and multi-rank faults [102], as prior work has shown that errors spread across a memory bank with a BER of 10^{-6}. Table 2 shows a summary of the fault models used.

Table 2: Fault models used and their justifications

Model	Description	Justification
Word	each bit of 2 consecutive bytes are corrupted with 50% probability	Assuming an x16 DRAM (Fig. 3). The number of corrupted bits has a sophisticated distribution across failure modes [102]. For simplicity, we corrupt each bit randomly with a 50% probability.
Column	~3% of 4KB pages have same word corrupted	Sridharan et al. [102] found up to 6% of rows got affected. Assuming 8KB logical banks (Fig. 3) and random logical to physical page mapping, each logical page is corrupted with probability 3%.
Row	~30% of two 4KB pages corrupted	Sridharan et al. [102] found up to 30% of columns got affected. Assuming 8KB logical banks (Fig. 3) and random logical to physical page mapping, with probability 30%, words of two logical pages get corrupted.
BER	Flip bits with uniform probability.	To test sensitivity of techniques to harsh conditions [71], aging [16, 33], multi-bank errors [102].

4 STRUCTURAL CODING (SC)

4.1 Challenges

Our goal is to design a technique for (1) protecting all the parameters, (2) catering to multi-bit memory errors, (3) and incurring low overheads. However, there are three challenges.

1) Keeping the memory overhead low while protecting all the parameters. We develop a coding technique to provide strong (with high probability) correction with low redundancy.

2) Recovering large clusters of affected bits, as traditional code words are affected by more than correction capability errors. We carefully select the granularity of CNN kernels for symbols, close to a logical page, to recover from both column and row failures.

3) Performance overhead due to our technique using one detection checksum per each symbol. Tolerating this overhead in fault-free execution is challenging. To decrease the overhead of our technique during fault-free execution, we separate the error detection and correction. We then remove from the execution part, the calculations that are only required during correction. That said, the correction is faster than reloading the value from secondary storage after every inference.

4.2 Overview of SC

We assume that for the naturally occurring errors, based on the data in Sridharan et al. [102], the portion of affected rows stays (with high probability) within bounds such as 6%. Depending on how the memory controller maps addresses, each row might affect two to eight (or even more if there are more channels and ranks) physical pages. Even so, there are upper bounds on the number of affected logical pages – we call this value l (Fig. 4 shows an example of row faults).

Each logical page contains a limited number of CNN kernels and fully connected layer matrix columns - we call these *parameter groups* (=symbols). For example, logical pages in AlexNet contain at most three parameter groups. Thus, we design the error correction to recover from errors in a certain number of parameter groups, each spanning one or a few pages (**Addressing Challenge 2**).

We propose a syndrome-based error correction similar to Reed Solomon (RS) codes. However, unlike normal RS coding [73], we use arithmetic multiplication and addition as in *real-number codes*

[22]. This allows us to recover the values at the software level, independent of the underlying representation of the real numbers (e.g., float64 and float32.)

4.3 Syndrome-based Error Correction

Similar to Anfinson et al. [12], we extend the RS coding scheme, but to *parameter groups* instead of single real numbers. To that end, as Fig. 5 shows, we add k redundant linear combinations of parameter groups. Given the location of faulty groups, we solve a system of linear equations to recover the original groups (see Fig. 5).

Note that the number of additional symbols is not as many as the number of protected symbols, but rather as many as the number of faulty symbols that we want to tolerate which is often much smaller than the number of protected symbols. Therefore, *the memory footprint overhead of the proposed technique is sub-linear with regard to the size of parameters.*

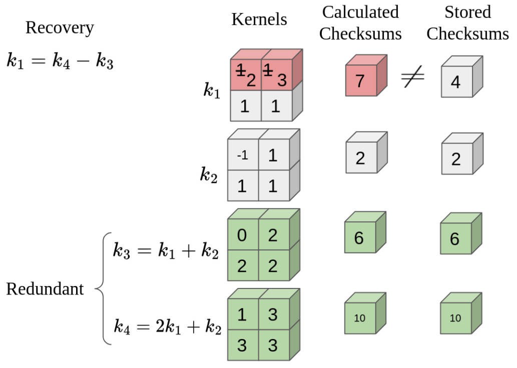


Figure 5: SC in the presence of faults, in a single kernel. Here $n = 4$ and $k = 2$. n is small for simplicity of the example and the memory overhead will be lower for larger values of n.

We must select the coefficients used to add the redundant groups in such a way that any correct subset of them would result in a system of linear equations with a single solution. There are a variety of ways to do so[18, 19, 21, 22], however, selecting random coefficients is often sufficient [22]. Therefore, we use a generator matrix with random values at the non-systematic rows. To avoid decoding during fault-free execution, we first initialize the generator matrix with the systematic rows. Then we add k random rows. Another advantage of using random redundant rows is that we can use a pseudo-random generator, leading to both reduced memory storage and bandwidth overhead **(Addressing Challenge 1)**.

In the case of a single error, localization of the error can be done efficiently in logarithmic time [98] based on the k redundant symbols. However, for multi-bit errors, the complexity of locating correct symbols increases exponentially with k. In the next section, we will address this challenge.

4.4 Adding Erasure Codes

To overcome the complexity of locating faulty parameter groups discussed in the prior section, we apply group-level checksums, by simply adding all the parameters in a group together (See Fig. 5 under *calculated checksums* and *stored checksums*). These checksums form an erasure code that detects the faults and eases the task of locating faulty parameter groups. We further simplify the detection by using a single checksum per CNN layer. These checksums are very small (less than 1%) compared to parameters and cause negligible memory bandwidth and storage overhead. Checksums based on the addition of values have been shown to be effective for detecting critical faults [81]. Together with the systematic property of our coding scheme, they obviate the need to do any decoding when no fault is detected. This is especially important as we need to protect each inference for real-time detection and recovery. *Therefore, we move the recovery logic off the critical path and perform only the detection for each layer* (**Addressing Challenge 3**).

Note that integrating the group level checksums, while needing to read the redundant values from memory, is typically quicker than reloading the weights (from secondary storage as shown in Table 1) as an alternative way of recovery.

4.5 Implementation

We have implemented SC as an automated transformation for PyTorch CNNs [83]. The user only needs to specify the 'n' and 'k' parameters (or choose the default ones provided in Section 5.1). Everything else is automated. *Thus, there is no developer effort or retraining needed to deploy our technique.* Our implementation transforms Pytorch modules to their protected versions with our protection activated on each forward pass. A challenge was to use appropriate Pytorch tensor operators so as not to cause unnecessary overhead. However, our coding scheme does not assume anything specific to PyTorch and can be ported to other frameworks.

5 EVALUATION

5.1 Experimental Setup

1. Models and Data Sets: We choose primarily image classification-based CNNs for our evaluation. We choose six CNN models for this task, GoogLeNet[104], ShuffleNet[68], AlexNet[56], SqueezeNet[46], ResNet50[39], and MobileNetV3[44]. For the FI experiments, we choose a random subset of 128 images out of more than $50,000$ validation images in ImageNet [28]. We had to limit the number of images to balance representativeness with time as FI experiments take a long time, and we performed $\sim$ 18 million FI experiments. Our images are diverse as they are from distinct object classes.

We also consider audio classification and text classification CNNs for evaluating SC's generality beyond image classification. For text classification, we use the dataset [26] and model from Mozafari et al. [74] that includes a transformer. For audio classification, we use X-Vectors [99] model for classifying Google Speech Commands [112] dataset.

2. Metrics and Research Questions: We use the following metrics for our experiments:

- Silent Data Corruption (SDC) Rate: For classification models, the SDC rate is the fraction of correct inferences turned into misclassifications by the CNN under a fault occurrence. For example, an SDC rate of 50% means that an accuracy of 80% in the fault-free case drops down to 40% due to the fault. For a model without protection (protection *None*), it shows how likely an error is to change the CNN classification outcome.
- Coverage: We define this as (1 - SDC rate). The higher the SDC rate, the lower the coverage (of the technique).

- Normalized MSE: Normalized Mean Squared Error (MSE) is the MSE of a regression model under a fault divided by the MSE of the fault-free pre-trained model. This ranges between 1 and $+\infty$ with 1 indicating the best protection.
- Performance overhead: Percentage increase in the average time of inference of the CNN protected with a technique.
- Correction overhead: Percentage increase in Floating Point Operations (FLOPs) as a hardware-independent measure for evaluating a technique's algorithmic optimizations.
- Memory overhead: Percentage change in memory footprint, bandwidth consumption, and latency for a technique.
- Energy overhead: Percentage change in energy consumption of CPU and DRAM for a technique.
- Undetected Rate: The fraction of faults on correct inferences that are not detected by a technique.
- Corrected Rate: Fraction of faults on correct inferences that remain correctly classified by a technique's protection.
- Miscorrected Rate: Fraction of faults turned into misclassifications after being detected and corrected by a technique

We answer the following research questions (RQs):
RQ1: What is SC's performance and memory overhead?
RQ2: What is the correction overhead of SC?
RQ3: What is SC's memory bandwidth and latency overhead?
RQ4: What is the coverage of SC in the presence of multi-bit memory errors compared to other techniques?
RQ5: How does the coverage of SC vary as Bit Error Rate (BER) increases compared to other techniques?
RQ6: How effective is SC for other data formats?
RQ7: How effectively does SC detect errors?
RQ8: What is the energy consumption overhead with SC?
3. Baselines: We compare the effectiveness of our technique to prior work that provides recovery for memory faults.

- MILR [86], which detects parameter errors using layer input and output checkpoints and recovers parameters by applying the inverse of the layer. We reimplemented MILR from the paper, as it was not available for PyTorch. To keep the correction overhead tractable, we keep a batch of *partial checkpoints*, and only correct kernels or columns corresponding to the erroneous outputs detected by the checkpoints. We report the running time overhead assuming that the detection was run at each inference.
- RADAR [60] as it also performs checksums and provides recovery by restoring the original values. We extend RADAR to floating point models. The *sign bit* and *exponent bits* are the most significant bits for faults [43], therefore, when we calculate the RADAR checksums, we only use the 8 most significant bits.
- *Range restriction approaches*: FT-Clip-Act (activation clipping) [43] and Ranger [23] truncate or reset intermediate layers' values if they fall outside a prescribed range, but do not restore the original values. We implement these for PyTorch.

We do not compare with Triple Modular Redundancy (TMR) because we assume only a single memory module is available, and hence we cannot have independent replicas. However, we compared it with a triple-data-replication (TDR) approach, where three copies of the data are stored in the *same* memory module, and voting is performed on a per-bit basis. *We found that while TDR protection had an SDC rate of nearly 0, it incurred performance overheads of more than 200%, and hence we did not consider it further.*
4. Software and Hardware: We use pre-trained model parameters from *TorchVision*[70] and SpeechBrain[88] for image and audio classification. For text classification, we reproduced the trained model from Mozafari et al [74].

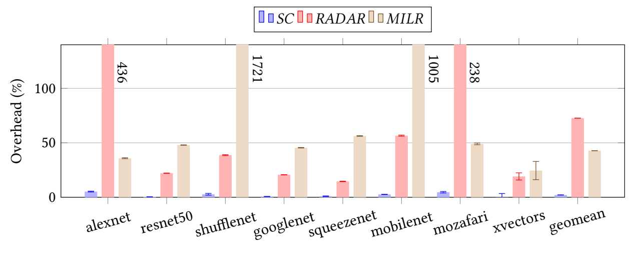
We perform FI and inference using *PyTorch* [83]. We generate random error patterns as per our fault model (Section 3) and inject them into the model parameters during inference (one fault/run).

To measure the performance overhead in FLOPs, we use PAPI to read performance counters [106]. We use *perf* [3] to measure the memory bandwidth and latency following the approach in Helm et al. [42], and energy consumption following the approach in Khan et al.[52]. We add the package and DRAM energy consumption for each technique per inference.

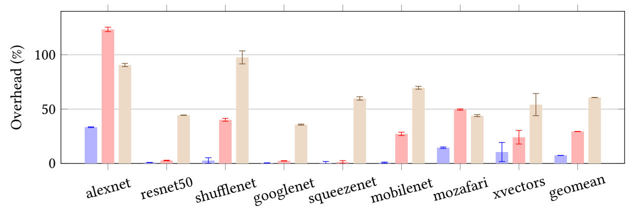
For the run-time and memory overhead measurements, we use a system running Ubuntu-20 with 32GB DDR3 DRAM and Intel(R) Core(TM) i7-4930K CPU @ 3.40GHz. For the energy overhead measurements, we used another system running Ubuntu-20 with 8GB DDR3 DRAM and Intel(R) Core(TM) i7-5500U CPU @ 2.40GHz that supported memory energy consumption measurement. For calculating the energy overhead of Chipkill, we use Micron's publicly available spreadsheet [9] and initiate it with the parameters of the latter system. We set the read/write utilization according to the size of activations and weights in Table 1. Then we change the chip density and width to get a Chipkill setup with 8GB capacity.
5. Execution Time and Memory Measurements: We calculate the median of 25 bursts of 5 inferences on random input examples by the model. We use Wilcoxon-Signed-Rank Test [90] to validate the statistical significance of the overhead measurement changes before and after applying a technique. This test does not assume the normality of measurements. However, it assumes symmetry in the difference scores for execution time, memory bandwidth and energy consumption, which is the case for us.
6. FI Experiments: We inject 400 faults per failure mode per model. These are sufficient for statistical significance in rejecting the null hypothesis that SC does not reduce the SDC rate more than the techniques used for comparison. *We report 95% confidence intervals for SDC rates in all experiments.* For each fault, we perform inference for all of the chosen images. If the inferred label of *any* of the images deviates from its correct label, we label the inference as an SDC.

When injecting faults, we exclude parameters that are weights of neither fully connected nor convolution layers of the networks. The percentage of these parameters with respect to the considered weights (Table 1) constitute less than 1% of the memory. Hence, they can be protected using TDR (i.e., keeping three copies of the data in memory), with low overheads. Table 1 also shows the size of the memory occupied by the parameters and activations. We find that model parameters, on average, occupy 58% of the Resident Set Size (RSS) occupied by the program data, which is quite sizeable.

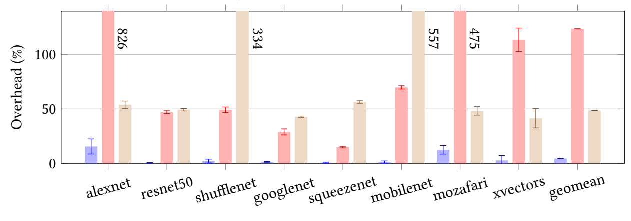
Note that we did not observe any crashes in our FI experiments, and hence we do not report crashes. This is because our fault injections are confined to the model parameters.
7. SC configuration To find the optimal configurations of SC with regard to memory footprint, we start at $k = 1$ and $n = 257$, and increase them, until the calculated probability of more than k erroneous kernels is less than 10%. We used an analytical model for



(a) Execution Time Overheads



(b) Bandwidth Overheads



(c) Energy Overheads

Figure 6: Execution time overheads, bandwidth overheads, and energy overheads for each CNN incurred by each technique. Lower is better. geomean = geometric mean. Error bars show quartiles. MILR outliers are not used for geomean as the long running time is due to normalization of grouped convolution.

this calculation. Based on the optimal values observed across all layers of all networks, we chose $n = 288, k = 32$.

5.2 Results

We organize the results by the RQs.

5.2.1 RQ1 – Performance and Memory Overheads. Fig 6a shows the detection performance overheads, across the models for SC and the other techniques. We find that the performance overhead of SC is 2.07% on average, which includes a summation over the parameters. AlexNet and Mozafari have the highest performance overheads (15% and 19%) as these have the biggest parameter sizes (Table 1). One reason for the low overhead is that the kernels of many layers are small enough to fit on the chip and get reused, and loading them for the checksum does not hurt performance. However, there are some cases where the kernels of a layer do not fit on-chip and will be streamed. In those cases, the overhead percentage is still low because calculating the checksum is much lighter than performing the total operations of the layer.

When comparing SC's overhead with other techniques, the range restriction [23, 43] techniques have a lower average overhead of 0.53%. However, as we show later, range restriction techniques have very low error coverage for most memory fault types (RQ4). Therefore, we exclude these techniques for further overhead comparison.

Fig. 6a shows the overheads of the remaining two techniques, MILR and RADAR. RADAR incurs very high performance overheads, with an average of 72.69%. This is because software implemented RADAR iterates over the parameters as well as the checksums, and performs many byte comparisons. Likewise, MILR incurs an average performance overhead of 42.78%, which is still high. This is because it duplicates almost every operation by checking the checkpoints at each layer for real-time detection. *SC has neither of these sources of overhead and therefore, the performance overhead of SC is much lower than RADAR and MILR.*

Table 3 shows the memory footprint and latency overheads, across the evaluated models for SC and the other techniques. The memory footprint overhead of SC is less than 27% across models, and is 15.71% on average. In comparison, the memory footprint overhead of RADAR is 25% (uniformly), while the memory footprint overhead of MILR depends on the relative size of layer outputs and parameters. On average, the additional memory used by our implementation of MILR is 71.81%. *Thus, the average memory footprint overhead of SC is lower than that of both RADAR and MLIR.*

Model	Memory footprint (%) SC	MILR	RADAR	Memory latency (%) SC	MILR	RADAR
alexnet	12.51	5.26	25	-14.94	-31.63	73.27
squeezenet	18.18	165.32	25	-	-6.02	-0.62
mobilenet	18.19	342.95	25	-	-16.41	3.81
googlenet	13.48	43.91	25	-0.68	-0.11	-0.66
resnet50	12.63	30.57	25	-0.37	10.39	1.65
shufflenet	26.26	122.47	25	-	2.45	6.86
mozafari	12.5	11.18	25	-19.45	0.66	38.53
xvectors	12.61	32.07	25	-	14.35	24.81
Avg[a]	**15.71%**	71.81	25	-9.26	-4.39	16.25

[a]We did not include the dashes and use geometric mean.

Table 3: Detection and Memory Overhead. The dashes mean there are no statistically significant changes in the measurements (p value < 0.05). Negative values indicate a reduction from the baseline.

5.2.2 RQ2 – Error correction overhead. Fig. 7 shows the *worst-case* FLOPs overhead of SC with regard to the number of parameter groups with faults across different models at the middle. On the right, it shows the FLOPs overhead of MILR for the same number of faults in the same parameter groups. We compare the correction overhead only to MILR, which requires compute for correction (others only set default values). We perform correction only in the (rare) case of an error, so the overhead is off the normal execution path. Up to eight erroneous symbols, we find that the overhead of SC is within 80%. If we exclude the two lightweight models (MobileNet and ShuffleNet), the overhead is within 6%. We did not show AlexNet for MILR as it has a correction overhead of about 400% for Alexnet. This is because AlexNet has larger, more complex kernels to calculate the inverse of convolution. Otherwise, the error correction overhead of MILR is comparable to SC.

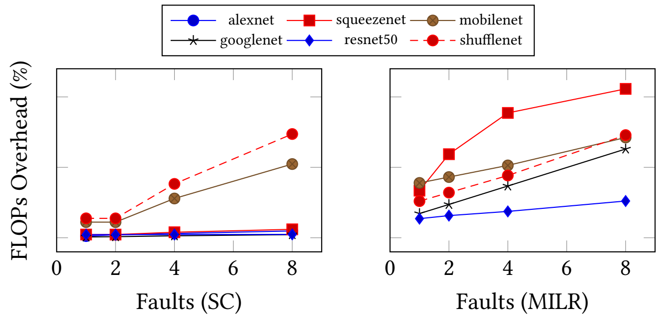


Figure 7: Worst-case correction overhead of SC (left) in FLOPs for the different CNNs per number of corrections, compared to MILR (right).

5.2.3 *RQ3 – Memory Bandwidth and Latency.* The bandwidth overheads are shown in Fig. 6b. The bandwidth overhead of SC is 7.46% on average. This overhead is dominated by the duplicate reads of the parameters for the purpose of calculating the checksums. Reading the (very small) detection checksums from the memory incurs negligible bandwidth overhead. Therefore, the highest overheads are incurred by the models with the biggest parameters (AlexNet and Mozafari). Comparing the bandwidth overhead with the other techniques, MILR and RADAR incur more overhead because of the additional need to read their checkpoints and checksums respectively. MILR incurs 60.66% and RADAR 29.44% average memory bandwidth overhead. Thus the memory bandwidth overhead of SC is lower than that of both MILR and RADAR.

The changes in average memory latency are shown in Table 3. SC leads to low latency across the different models, but the latencies of MILR and RADAR vary considerably and are neither homogeneously increased nor decreased. This is because the change in the memory latency is due to the interplay of many factors including (1) the number of memory requests and latency of the network itself, (2) the number of buffers accessed simultaneously for protection, (3) the relative size of activations and parameters, and (4) the size of the working set of the network compared to the size of the cache. SC has low latency as its detection is mostly sequentially accessing a single memory buffer and bank conflicts are rare.

5.2.4 *RQ4 – Coverage for different Fault Types.* Fig. 8 shows the SDC rate of SC compared with the other techniques, with regard to different models and under different fault types in our fault model (Section 3). As mentioned, we inject a single fault in each FI run for each technique and fault type.

We make two observations. First, as the granularity of errors grows larger (from word to column and then to row), the SDC rate increases (as expected). For example, ResNet50 has an SDC of 14% under word failure, 28% for column failure, and 100% under row failure. Second, the SDC rate of the model protected with SC is *three orders of magnitude lower than the SDC rate of the unprotected model.* For example, the SDC for AlexNet in word failures is 3.62×10^{-5} when it is protected with SC, while it is 5.69×10^{-2} with no protection at all.

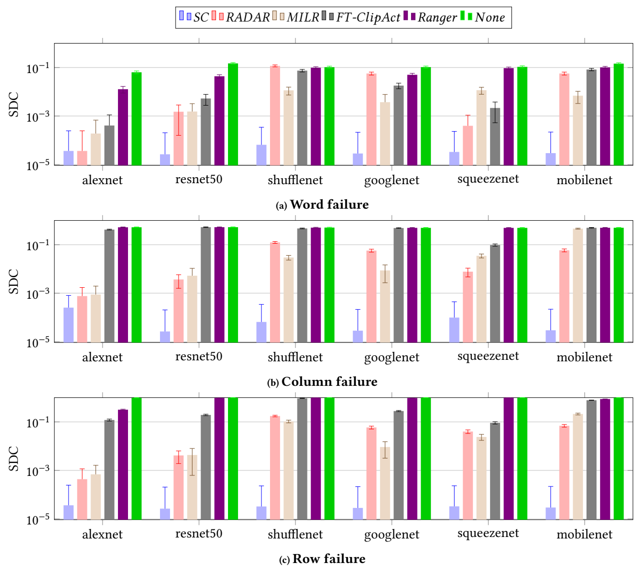


Figure 8: SDCs for different fault types (word, column, and row) for each CNN under different protections. Y-axis is in log scale. Lower is better.

Moreover, on average, SC's SDC rate is lower than MILR's SDC rate by 236× and lower than RADAR's SDC rate by 273×. Further, in all cases except for AlexNet for word and column failures, MILR and RADAR show higher SDCs than SC This is because RADAR's correction is similar to weight pruning [110] and AlexNet tolerates weight pruning [46]. MILR checkpoints use linear equations to recover the parameters. As erroneous parameters increase, the equations cannot determine the original values, as not all layers are invertible, and hence MILR cannot correct the error.

Finally, the techniques that have low performance and memory overhead (i.e., Ranger, FT-ClipAct). neither provide protection for all failure modes, nor for all models, e.g., FT-ClipAct's SDC rate is 19.37% under row failure for ResNet50.

5.2.5 *RQ5 – Coverage under different Bit Error Rates (BER).* Fig. 9 shows the SDC rate for different protection techniques under different BERs for different models. We consider three BER values, high (10^{-5}), medium (10^{-6}), and low (10^{-8}), to explore the range of behaviors. At each FI run, we choose a bit to flip with probability equal to BER, e.g., with 10^5 bits and BER = 10^{-4} around 10 bit-flips is expected on average.

We make two observations from the figure. First, we see that as the BER increases from 10^{-8} to 10^{-5}, in all cases the SDC rate increases (as expected). For example, the SDC for GoogLeNet increases from 5.24% to 61% and then to 100% as the BER increases. Second, SC provides the lowest SDC rate among all the techniques across all networks, and at all three BERs. *On average, the SDC rate of SC is two orders of magnitude lower than all the other techniques.*

In fact, RADAR in some cases, increases the SDC rate, such as for MobileNet at BER = 10^{-8} (SDC = 5.6%) compared to the case of no protection (SDC = 1.2%). This is because when RADAR encounters a bit-flip in the most significant bits of a parameter, it will erase the weight, which is sometimes worse than letting the error go undetected. For example, if the error leads to a smaller parameter

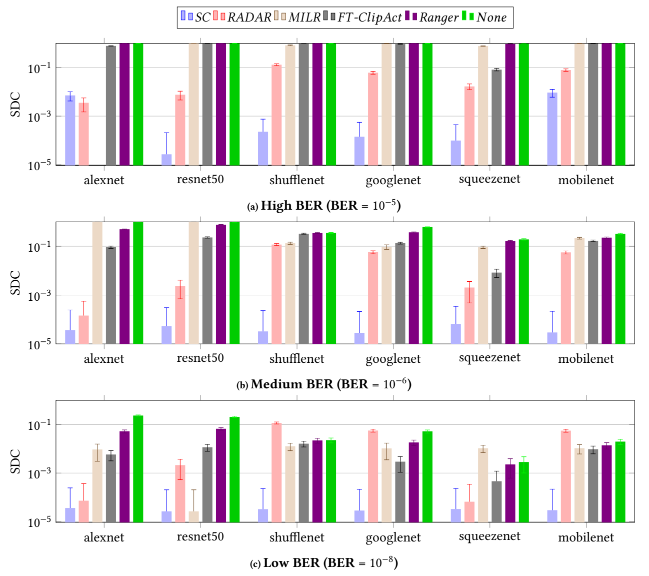


Figure 9: SDCs Vs. Bit Error Rate (BER) for each CNN under different protections[a]. Y-axis is in log scale. Lower is better.

[a] AlexNet data point is missing for MILR in which our implementation ran out of resources while running the FI experiments.

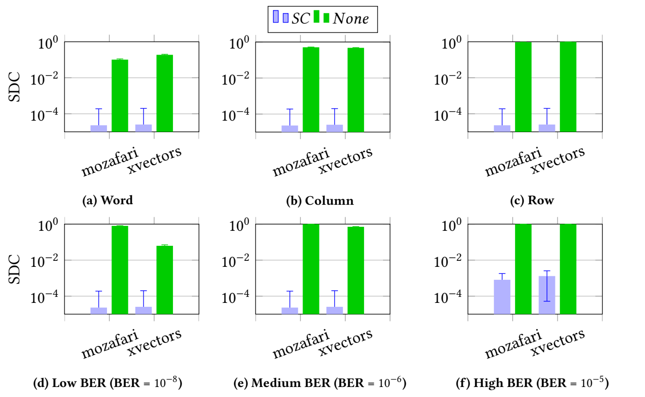


Figure 10: SDC of SC with other data formats. SDC of multi-bit and Bit Error Rate failure modes for each CNN with and without protection. Y-axis is in log scale. Lower is better.

value with the same sign, RADAR will likely make the error worse by erasing that parameter.

In summary, none of the other techniques provides as strong protection as SC. This is because, at high BER, they perform inaccurate corrections in many places of the network causing a large deviation of output values. The inaccuracy of MILR comes from non-invertible layers, and for the other techniques, it comes from correcting to default values.

Moreover, SC provides strong protection (SDC< 1%) even at the highest BERs, BER = 10^{-5}, stronger than all of the other techniques - except for the case of RADAR with SDC 0.3% for AlexNet at BER = 10^{-5} where SC's SDC rate is 0.7%.

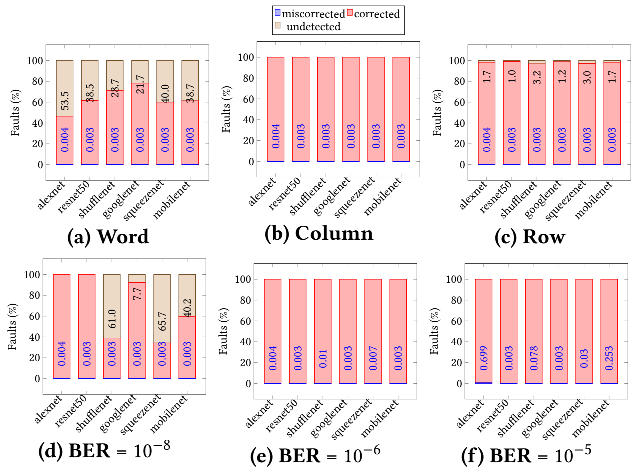


Figure 11: Breakdown of injections on the correctly classified inputs per model per fault type. Labels for corrected are not shown (= 100% − *undetected* − *miscorrected*).

5.2.6 RQ6 – Other data formats. In this RQ, we consider data formats other than images. Fig. 10, shows the SDC rate of the audio classifier (XVectors) and the text classifier (Mozafari) under different failure modes and BERs both with and without SC. We find that the worst SDC rate after protection is for XVectors under the High BER, which is 700x smaller than the SDC rate with no protection. Further, SC decreases the SDC rate by 9000× on average. Thus, SC's effectiveness also extends to audio and text inference.

5.2.7 RQ7 – Detection. Fig. 11 shows the breakdown of injections on the correctly classified inputs per model per fault type. All of the undetected faults were benign, and led to successful classification. This is because SC's detection, by using floating point summation instead of bit-wise operations, is sensitive to the errors in more significant bits and in bigger (more important [115]) weights. The checksum does not grow very large and remains sensitive to changes because the weights are distributed around zero. SC optimistically corrects the symbols with the most deviating checksums, leading to miscorrections when there are more than k erroneous symbols. However, the highest miscorrection percentage is for AlexNet under the high BER (10^{-5}) fault type occurring in only 0.699% of the injections. Thus, SC's detection is effective.

5.3 RQ8 – Energy consumption

Fig 6c shows the energy consumption overheads, across the evaluated models for SC and the other techniques. We find that the average energy overhead of SC is 4.38%. AlexNet and Mozafari have the highest energy overheads (15% and 12%) as they have the biggest parameter sizes in memory (Table 1).

When comparing SC's overhead with other techniques, we exclude range restriction [23, 43] techniques as they have negligible energy overheads (<2%) and as these techniques have very low error coverage for most fault types. This leaves only MILR and RADAR, and they are shown in Fig. 6c. RADAR incurs very high energy overheads, with an average of 123.77%. MILR also incurs an average energy overhead of 48.57%, which is still high. This is because the energy overhead is dominated by the run-time overhead.

Chipkill	Channel	Overheads (%)		
		Storage	Energy	Energy normalized by storage
x4				
Commercial dual-channel [77]	144-bit	12.5	33.71	32.37
Commercial single-channel [77]	72-bit	12.5	11.08	10.41
Virtual ECC [116]	128-bit	9.38	8.65	7.52
Bamboo [53]	72-bit	12.5	11.08	10.41
FrugalECC+Multi [54]	64-bit	7.25	8.65	7.16
FrugalECC+QPC [54]	64-bit	13.5	8.65	8.21
x8				
Virtual ECC [116]	144-bit	18.75	15.28	15.87
Virtual ECC [116]	128-bit	18.75	12.38	12.91
FrugalECC+OPC [54]	72-bit	26	2.22	3.32
FrugalECC+Multi [54]	64-bit	13.5	0.76	0.49
Structural Coding	any	15.71	4.38	4.38

Table 4: Chipkill ECC energy and storage overheads

When comparing the energy overhead of SC to Chipkill, we use Micron's energy spreadsheet[9] (Section 5.1) to extrapolate the energy consumption for our setup. We compare the energy overheads normalized to the storage overhead for a fair comparison. Table 4 shows the memory and energy overheads for the well-known Chipkill proposals. Most of the Chipkill proposals have a higher energy consumption than SC, because they activate at least twice the number of chips as SC. Non-commercial Chipkill proposals have higher energy consumption than SC; except for FrugalECC+OPC and FrugalECC+Multi for x8 DRAM. However, FrugalECC+OPC has higher storage overhead than SC and both x8 configurations of FrugalECC require hardware modification unlike SC. Commercial Chipkill's energy overhead on average is 10.4% and 32.4% for single-channel and dual-channel [77] respectively. *Therefore, the energy overhead of SC is lower than those of RADAR, MILR, commercial Chipkill and most non-commercial Chipkill proposals.*

6 DISCUSSION

Use cases: SC can be used in two classes of systems:

(1) *Without Hardware ECC*: SC can reduce the failure rate of CNNs by three orders of magnitude. Though many HPC systems have ECC, hardware ECC is not always available on commodity systems as the chips, DIMMs, and motherboards may not support it. For example, 40% of Intel chips do not support ECC [8].
(2) *SEC-DED*: As we found in Section 5.2.4, SC can correct multi-bit memory errors that SEC-DED cannot correct.

Furthermore, with hardware ECC one has to pay the cost even if errors are rare, while SC can be turned on/off as it is a software-based technique. That makes SC usable in a broad set of deployments, independent of the hardware platform. However, SC protects only the CNN parameters, while hardware ECC protects the whole memory.

Chipkill: Chipkill requires 18-36 DRAM devices [108]. Typically, this is ensured by using DRAM modules with 18-36 devices per channel or across multiple channels [50]. In contrast, a typical SECDED-based ECC uses traditional modules with 9 DRAM devices [75]. This has two key consequences: (1) As most systems use 64Byte cachelines, we can obtain 64Bytes of data and 8 bytes of ECC over modules that use only 9 DRAM devices. Thus, such a design efficiently uses memory bandwidth. However, if we use a module with 18-36 devices, each memory access is 128 or 256 Bytes of data and 8 or 16 Bytes of ECC respectively. This increases bandwidth overheads by 2×-4×, thereby reducing performance, and incurring power overheads[51]. (2) To reduce these bandwidth overheads, Chipkill-based systems use x4 devices. While this decreases the data that can be obtained from each device and tries to better utilize the bandwidth across a larger number of devices, it is typically prevalent in older DDR2-based systems. Modern DDR5-based memory systems [2], for instance, use x8 and x16 devices only – essentially increasing the data supplied by each DRAM device, thus making it even more inefficient to employ traditional Chipkill using these devices. Furthermore, Chipkill is more expensive than SEC-DED [77], and the current High Bandwidth Memory (HBM2) standard makes implementing Chipkill inefficient as every cache line comes from a single device [37].

Extension to GPUs: We chose CPUs for SC as they are commonly used for inference, which is our focus. To port SC to GPUs, we need to ensure a deterministic order of summation for robust detection (which is easily achievable in PyTorch [83]).

Optimizing correction overhead: While SC has low performance overheads, its overhead can be further reduced by optimizing its error correction at the cost of a slight increase in memory. Correcting k erroneous symbols among n symbols is equivalent to a matrix inversion and its known complexity is $O(n^2(n-k))$[63]. The Woodbury formula [113], by storing the pre-computed inverse of a square matrix of size $O(n^2)$, reduces the complexity of inverting any of its square sub-matrices of size $(n-k)\times(n-k)$ to $O(k^3+kn^2)$. Correcting k errors can be formulated as finding the inverse of such a sub-matrix. This is to the benefit of the performance overhead of the correction in our case because $k \ll n$. The additional memory footprint overhead will also be low as the size of the parameters is usually much larger than the size of the pre-computed inverse.

Unfortunately, the Woodbury formula [113], cannot be trivially applied, because our generator matrix is not a square matrix. Inspired by the effectiveness of random generator matrices [22], we pad the generator matrix with random numbers to obtain a square matrix while preserving its invertibility. This technique asymptotically reduces the overhead of error correction. We observed a similar error-correction coverage with and without this optimization, and a *2.7x reduction of the average correction overhead* in Fig. 7.

Limitations: Like most coding techniques, SC does not support targeted changes in values that result in the checksums not being violated. However, the probability of this occurring due to natural faults is very low. Also, very large-granularity memory failures of a bank, chip, rank or channel are not protected. However, such failures are more likely to arise due to permanent faults rather than transient faults [15]. We have also implemented this technique for quantized networks, but in those cases because of the smaller field of values, random generator matrices do not give full k error correction capability. An alternative is to use standard RS codes[105] or choose a higher value for k. Depending on the system setup, the correction due to SC may take more time than reloading safely stored parameters, in which case the correction part of SC does not help. Moreover, in applications where latency is not a constraint, reloading the values from storage, or scrubbing may suffice.

Finally, while there may be differences between the fault patterns we have used and those of newer technologies, the differences are likely to be small as SC relies on the locality of errors in a limited number of memory pages. The correspondence of memory rows and pages will likely remain in newer DRAM generations, and hence the fault patterns will be similar. We have also evaluated against a Bit Error Rate (BER) model without specific failure assumptions.

7 RELATED WORK

In this section, we classify the fault-tolerance techniques proposed for CNNs, into four categories as follows. (1) *High overhead* techniques despite their strong correction capability incur high latency, memory or performance overhead making them unsuitable in settings where the net cost matters. (2) *limited multi-bit protection* techniques do not address the large-granularity memory errors discussed in Section 2. (3) *Training or retraining required* techniques incur high costs, and (4) *Hardware-/Model-Dependent* ones are difficult to adapt for existing systems.

1. High overhead: TMR is used for error detection and recovery, but incurs high costs[111]. Ali et al. [11] use redundancy for convolutions, but they incur high execution overheads.

Zhao et al. [117] adapt ABFT for CNNs with multiple levels of checksums. Their technique relies on reloading model parameters for multiple parameter errors. However, this would incur high latency overheads, and would not be desirable for real-time operation. MILR[86] exploits the invertibility of DNN operations to recover the weights using the stored checkpoints of inference values. Consequently, the detection is not performed in real time. but only when the checkpoint inputs are run.

2. Limited Multi-bit protection: Mahmoud et al. [69] selectively protect only some of the kernels from computational faults. Other work selectively protects only parts of the network [10, 24, 65]. Unfortunately, selective protection does not provide strong protection against multi-bit faults. Qin et al. [87] propose a similar approach to RADAR as well as a slightly more robust weight representation. The robustness benefits of their representation are orthogonal to our approach. In *range restriction* techniques [23, 43, 80] the range of activation values is checked. When a fault is detected, the suspicious activation values are set to either zero [43, 80] or the profiled bound [23, 80]. However, these approaches scale only up to a few bits of multi-bit errors (Section 5.2.4).

Researchers have used algorithm-based checksums for detection [38, 55, 81]. However, they do not consider recovery. Recent work [55] focuses on optimizing the overhead of Algorithm Based Error Detection (ABED) by considering the memory and compute-bound operations. However, they do not address error correction, and limit the scope of errors to computation errors. Dos et al. [30] analyze the efficiency of ABFT for GPUs, but DRAM errors are out of their studied scope. Other work has addressed the problem of multi-bit faults in the stored parameters of the deep neural networks in memory, but they target the detection of these faults and do not provide any error correction [32, 41, 64, 66, 69, 72, 81].

3. Training or Retraining required: Training/Retraining the network has several disadvantages, such as (1) the training dataset might not be available, (2) it requires significant computational resources, and (3) the outcome is not guaranteed to provide the same accuracy. There are many approaches for protecting DNNs that require training/retraining [17, 36, 40, 78, 82, 93, 94], and hence share these disadvantages. Lee and Yang [57] fixed the limitation of Guan et al. [36], which is the need for retraining, but their approach can still cause an accuracy degradation and does not recover from more than two bits of error in a codeword.

4. Hardware-/Model-Dependent: Many researchers have proposed hardware modifications for fault detection and correction, which typically incur high development costs. For example, Goldstein et al. [35] proposed a new hardware design for multiplication and addition units for tolerating computation faults. However, they do not target memory faults.

For emerging memories used in CNN applications, Nguyen et al. [76] propose intra-block remapping for the parameters to prevent large accuracy degradation due to stuck-at faults. Similarly, Rios et al. [91] proposed a resilient in-memory computing architecture for CNNs. Unlike our technique, these techniques require hardware modification, thereby leading to high system costs.

Sullivan et al. [103] propose a hardware ECC technique to mitigate multi-bit DRAM errors in GPUs. Similarly, Kim et al. [53] propose Bamboo, a technique to inform ECC design by the DRAM errors observed. However, these approaches are not customized on a per-application basis, and hence incur high memory overhead.

Many papers have proposed to use coding to protect parameters of DNNs [45, 89, 109]. For binary neural networks, parity can be used for making classifications robust [89], however it is not compatible with other neural networks at software-level. Upadhyaya et al. [109] use analog codes to protect analog weights of hardware-implemented neural networks. Huang et al. [45] propose coding for hardware CNNs with their parameters stored in non-volatile memories. However, these are specific to the target hardware.

8 CONCLUSION

Multi-bit DRAM memory soft errors spanning multiple words (i.e., large-granularity faults) can have significant reliability consequences for Convolutional Neural Networks (CNNs). We propose SC, an automated software scheme to protect CNNs from large-granularity memory faults. SC is implemented as an automated transformation for PyTorch. We evaluate SC on six CNN classifiers, and find that it (1) incurs low performance and memory overheads, (2) reduces the Silent Data Corruption (SDC) rate by over three orders of magnitude compared to unprotected CNNs, (3) outperforms existing state-of-the-art software-based techniques in performance, energy and memory overheads, and coverage, and (4) incurs a lower energy overhead than most hardware-based Chipkill schemes.

ACKNOWLEDGMENT

This research was enabled in part by support provided by the Natural Sciences and Engineering Research Council of Canada (NSERC), BC DRI Group, the Digital Research Alliance of Canada (alliancecan.ca) and a research gift from Intel Corporation. We would like to acknowledge Sriram Sankar, Harish Dixit and Release To Production (RTP) team within Meta Infrastructure for their continued support in our resilience efforts. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 956123.

REFERENCES

[1] [n. d.]. DDR4 SDRAM EDY4016A - 256Mb x16. https://www.micron.com/-/media/client/global/documents/products/data-sheet/dram/ddr4/4gb_ddr4_dram_2e0d.pdf. Accessed: 2021-11-16.

[2] [n. d.]. Ddr5 Sdram. https://www.micron.com/products/dram/ddr5-sdram. Accessed: 2021-11-16.

[3] [n. d.]. Linux. Perf. https://perf.wiki.kernel.org/index.php/Main_Page. Accessed: 2022-07-15.

[4] 2004. Uprating Semiconductors for High-Temperature Applications. https://www.micron.com/about/blog/2022/february/mobileye-advances-automotive-autonomy. Accessed: 2022-11-22.

[5] 2004. Uprating Semiconductors for High-Temperature Applications. https://www.micron.com/-/media/client/global/documents/products/technical-note/dram/tn0018.pdf. Accessed: 2022-11-22.

[6] 2018. ISO 26262: Road vehicles — Functional safety. https://www.iso.org/standard/68383.html. Accessed: 2022-11-22.

[7] 2022. EyeQ®: The System-on-Chip for Automotive Applications. https://www.mobileye.com/technology/eyeq-chip/. Accessed: 2022-11-22.

[8] 2022. Intel Products Home. https://ark.intel.com/content/www/us/en/ark/search/featurefilter.html?productType=873&0_StatusCodeText1=3,4. Accessed: 2022-06-22.

[9] 2022. System Power Calculators. https://www.micron.com/support/tools-and-utilities/power-calc. Accessed: 2022-11-22.

[10] Khalid Adam, Izzeldin Ibrahim Mohamed, and Younis Ibrahim. 2021. A Selective Mitigation Technique of Soft Errors for DNN Models Used in Healthcare Applications: DenseNet201 Case Study. In *IEEE Access*, Vol. 9. 65803–65823. https://doi.org/10.1109/access.2021.3076716

[11] Muhammad Salman Ali, Md Tauhid Bin Iqbal, Kang Ho Lee, Abdul Muqeet, Seunghyun Lee, Lokwon Kim, and Sung Ho Bae. 2020. ErDNN: Error-resilient deep neural networks with a new error correction layer and piece-wise rectified linear unit. In *IEEE Access*, Vol. 8. https://doi.org/10.1109/access.2020.3017211

[12] Cynthia J. Anfinson and Franklin T. Luk. 1988. A linear algebraic model of algorithm-based fault tolerance. In *IEEE Transactions on Computers*, Vol. 37. Ieee, 1599–1604.

[13] Sanghyeon Baeg, Mirza Qasim, Junhyeong Kwon, Tan Li, Nilay Gupta, Shi-Jie Wen, and Satyadev Kolli. 2019. Correctable and uncorrectable errors using large scale DRAM DIMMs in replacement network servers. In *Microelectronics Reliability*, Vol. 99. 104–112. https://doi.org/10.1016/j.microrel.2019.05.008

[14] GeunYong Bak, Soonyoung Lee, Hosung Lee, KyungBae Park, Sanghyeon Baeg, ShiJie Wen, Richard Wong, and Charlie Slayman. 2015. Logic soft error study with 800-MHz DDR3 SDRAMs in 3x nm using proton and neutron beams. In *2015 IEEE International Reliability Physics Symposium*. Ieee, Se–3.

[15] Majed Valad Beigi, Yi Cao, Sudhanva Gurumurthi, Charles Recchia, Andrew Walton, and Vilas Sridharan. 2023. A Systematic Study of DDR4 DRAM Faults in the Field. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 991–1002.

[16] Md Kawser Bepary, Bashir Mohammad Sabquat Bahar Talukder, and Md Tauhidur Rahman. 2022. DRAM retention behavior with accelerated aging in commercial chips. *Applied Sciences* 12, 9 (2022), 4332.

[17] Michael Beyer, Christoph Schorn, Tagir Fabarisov, Andrey Morozov, and Klaus Janschek. 2021. Automated Hardening of Deep Neural Network Architectures. In *ASME International Mechanical Engineering Congress and Exposition*, Vol. 85697. American Society of Mechanical Engineers, V013t14a046.

[18] Daniel L Boley and Franklin T Luk. 1991. A well conditioned checksum scheme for algorithmic fault tolerance. *Integration* 12, 1 (1991), 21–32.

[19] Richard P Brent, Franklin T Luk, and Cynthia J Anfinson. 1989. Choosing small weights for multiple error detection. In *High speed computing II*, Vol. 1058. International Society for Optics and Photonics, 130–137.

[20] Hsing-Min Chen, Akhil Arunkumar, Carole-Jean Wu, Trevor Mudge, and Chaitali Chakrabarti. 2015. E-ecc: Low power erasure and error correction schemes for increasing reliability of commodity dram systems. In *Proceedings of the 2015 International Symposium on Memory Systems*. 60–70.

[21] Zizhong Chen. 2009. Optimal Real Number Codes for Fault Tolerant Matrix Operations. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (Portland, Oregon) *(Sc '09)*. Association for Computing Machinery, New York, NY, USA, Article 29, 10 pages. https://doi.org/10.1145/1654059.1654089

[22] Zizhong Chen and Jack Dongarra. 2005. Numerically stable real number codes based on random matrices. In *International Conference on Computational Science*. Springer, 115–122.

[23] Zitao Chen, Guanpeng Li, and Karthik Pattabiraman. 2021. A low-cost fault corrector for deep neural networks through range restriction. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 1–13.

[24] Wonseok Choi, Dongyeob Shin, Jongsun Park, and Swaroop Ghosh. 2019. Sensitivity based Error Resilient Techniques for Energy Efficient Deep Neural Network Accelerators. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*. 1–6.

[25] Charng da Lu and D.A. Reed. 2004. Assessing Fault Sensitivity in MPI Applications. In *SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*. 37–37. https://doi.org/10.1109/sc.2004.12

[26] Thomas Davidson, Dana Warmsley, Michael Macy, and Ingmar Weber. 2017. Automated Hate Speech Detection and the Problem of Offensive Language. In *Proceedings of the 11th International AAAI Conference on Web and Social Media* (Montreal, Canada) *(Icwsm '17)*. 512–515.

[27] Timothy J Dell. 1997. A white paper on the benefits of chipkill-correct ECC for PC server main memory. In *IBM Microelectronics division*, Vol. 11. 1–23.

[28] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.

[29] Catello Di Martino, Zbigniew Kalbarczyk, Ravishankar K Iyer, Fabio Baccanico, Joseph Fullop, and William Kramer. 2014. Lessons learned from the analysis of system failures at petascale: The case of blue waters. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. Ieee, 610–621.

[30] Fernando Fernandes dos Santos, Pedro Foletto Pimenta, Caio Lunardi, Lucas Draghetti, Luigi Carro, David Kaeli, and Paolo Rech. 2018. Analyzing and increasing the reliability of convolutional neural networks on GPUs. In *IEEE Transactions on Reliability*, Vol. 68. Ieee, 663–677.

[31] Xiaoming Du, Cong Li, Shen Zhou, Mao Ye, and Jing Li. 2020. Predicting Uncorrectable Memory Errors for Proactive Replacement: An Empirical Study on Large-Scale Field Data. In *2020 16th European Dependable Computing Conference (EDCC)*. 41–46. https://doi.org/10.1109/edcc51268.2020.00016

[32] Xianglong Feng, Mengmei Ye, Ke Xia, and Sheng Wei. 2021. Runtime Fault Injection Detection for FPGA-based DNN Execution Using Siamese Path Verification. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 786–789.

[33] Moritz Fieback. 2017. Dram reliability: Aging analysis and reliability prediction model. http://resolver.tudelft.nl/uuid:e36c2de7-a8d3-4dfa-9da1-ac5b7e18614b. Accessed: 2023-02-28.

[34] Pietro Frigo, Emanuele Vannacc, Hasan Hassan, Victor Van Der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2020. TRRespass: Exploiting the many sides of target row refresh. In *2020 IEEE Symposium on Security and Privacy (SP)*. Ieee, 747–762.

[35] Brunno F Goldstein, Victor C Ferreira, Sudarshan Srinivasan, Dipankar Das, Alexandre S Nery, Sandip Kundu, and Felipe MG França. 2021. A lightweight error-resiliency mechanism for deep neural networks. In *2021 22nd International Symposium on Quality Electronic Design (ISQED)*. IEEE, 311–316.

[36] Hui Guan, Lin Ning, Zhen Lin, Xipeng Shen, Huiyang Zhou, and Seung Hwan Lim. 2019. In-place zero-space memory protection for CNN. In *Advances in Neural Information Processing Systems*, Vol. 32.

[37] Sudhanva Gurumurthi, Kijun Lee, Munseon Jang, Vilas Sridharan, Aaron Nygren, Yesin Ryu, Kyomin Sohn, Taekyun Kim, and Hoeju Chung. 2021. HBM3 RAS: Enhancing Resilience at Scale. In *IEEE Computer Architecture Letters*, Vol. 20. Ieee, 158–161.

[38] Siva Kumar Sastry Hari, Michael Sullivan, Timothy Tsai, and Stephen W Keckler. 2021. Making Convolutions Resilient via Algorithm-Based Error Detection Techniques. In *IEEE Transactions on Dependable and Secure Computing*. 1–1. https://doi.org/10.1109/tdsc.2021.3063083

[39] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[40] Zhezhi He, Adnan Siraj Rakin, Jingtao Li, Chaitali Chakrabarti, and Deliang Fan. 2020. Defending and harnessing the bit-Flip based adversarial weight attack. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. https://doi.org/10.1109/cvpr42600.2020.01410

[41] Zecheng He, Tianwei Zhang, and Ruby Lee. 2019. Sensitive-sample fingerprinting of deep neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 4729–4737.

[42] Christian Helm and Kenjiro Taura. 2020. On the Correct Measurement of Application Memory Bandwidth and Memory Access Latency. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*. 131–141.

[43] Le-Ha Hoang, Muhammad Abdullah Hanif, and Muhammad Shafique. 2020. Ft-clipact: Resilience analysis of deep neural networks and improving their fault tolerance using clipped activation. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1241–1246.

[44] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. 2019. Searching for mobilenetv3. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 1314–1324.

[45] Kunping Huang, Netanel Raviv, Siddharth Jain, Pulakesh Upadhyaya, Jehoshua Bruck, Paul H. Siegel, and Anxiao Andrew Jiang. 2020. Improve Robustness of Deep Neural Networks by Coding. In *2020 Information Theory and Applications Workshop, ITA 2020*. https://doi.org/10.1109/ita50056.2020.9244998

[46] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. In *arXiv preprint arXiv:1602.07360*.

[47] JEDEC Standard. 2015. DDR3 Standard. In *Jesd79-3e*.

[48] JEDEC Standard. 2015. DDR4 Standard. In *Jesd79-4*.

[49] Xun Jian, Henry Duwe, John Sartori, Vilas Sridharan, and Rakesh Kumar. 2013. Low-power, low-storage-overhead chipkill correct via multi-line error correction. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–12.

[50] Xun Jian, Henry Duwe, John Sartori, Vilas Sridharan, and Rakesh Kumar. 2013. Low-power, low-storage-overhead chipkill correct via multi-line error correction. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–12.

[51] Xun Jian and Rakesh Kumar. 2013. Adaptive reliability chipkill correct (arcc). In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. Ieee, 270–281.

[52] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K Nurminen, and Zhonghong Ou. 2018. RAPL in Action: Experiences in Using RAPL for Power measurements. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)* 3, 2 (2018), 1–26.

[53] Jungrae Kim, Michael Sullivan, and Mattan Erez. 2015. Bamboo ECC: Strong, safe, and flexible codes for reliable computer memory. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. Ieee, 101–112.

[54] Jungrae Kim, Michael Sullivan, Seong-Lyong Gong, and Mattan Erez. 2015. Frugal ECC: efficient and versatile memory error protection through fine-grained compression. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12. https://doi.org/10.1145/2807591.2807659

[55] Jack Kosaian and KV Rashmi. 2021. Arithmetic-intensity-guided fault tolerance for neural network inference on gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(Sc '21)*. Association for Computing Machinery, New York, NY, USA, Article 79, 15 pages. https://doi.org/10.1145/3458817.3476184

[56] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.), Vol. 25. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[57] Seo-Seok Lee and Joon-Sung Yang. 2022. Value-aware parity insertion ECC for fault-tolerant deep neural network. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 724–729.

[58] Scott Levy, Kurt B Ferreira, Nathan DeBardeleben, Taniya Siddiqua, Vilas Sridharan, and Elisabeth Baseman. 2018. Lessons learned from memory errors observed over the lifetime of Cielo. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. Ieee, 554–565.

[59] Guanpeng Li, Siva Kumar Sastry Hari, Michael Sullivan, Timothy Tsai, Karthik Pattabiraman, Joel Emer, and Stephen W Keckler. 2017. Understanding error propagation in deep learning neural network (DNN) accelerators and applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[60] Jingtao Li, Adnan Siraj Rakin, Zhezhi He, Deliang Fan, and Chaitali Chakrabarti. 2021. Radar: Run-time adversarial weight attack detection and accuracy recovery. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 790–795.

[61] Jingtao Li, Adnan Siraj Rakin, Yan Xiong, Liangliang Chang, Zhezhi He, Deliang Fan, and Chaitali Chakrabarti. 2020. Defending bit-flip attack through DNN weight reconstruction. In *Proceedings - Design Automation Conference*, Vol. 2020-July. https://doi.org/10.1109/dac18072.2020.9218665

[62] Xin Li, Kai Shen, Michael C Huang, and Lingkun Chu. 2007. A Memory Soft Error Measurement on Production Systems.. In *USENIX Annual Technical Conference*. 275–280.

[63] Xiaocan Li, Shuo Wang, and Yinghao Cai. 2019. Tutorial: Complexity analysis of singular value decomposition and its variants. In *arXiv preprint arXiv:1906.12085*.

[64] Yu Li, Yannan Liu, Min Li, Ye Tian, Bo Luo, and Qiang Xu. 2019. D2nn: a fine-grained dual modular redundancy framework for deep neural networks. In *Proceedings of the 35th Annual Computer Security Applications Conference*. 138–147.

[65] Fabiano Libano, Brittany Wilson, J Anderson, Michael J Wirthlin, Carlo Cazzaniga, Christopher Frost, and Paolo Rech. 2018. Selective hardening for neural networks in FPGAs. In *IEEE Transactions on Nuclear Science*, Vol. 66. Ieee, 216–222.

[66] Qi Liu, Wujie Wen, and Yanzhi Wang. 2020. Concurrent weight encoding-based detection for bit-flip attack on neural network accelerators. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2020*.

[67] Xingxing Liu, Yongzhan He, Hongmei Liu, Jiajun Zhang, Bin Liu, Xiangyu Peng, Jialiang Xu, Jun Zhang, Alex Zhou, Paul Sun, Kunye Zhu, Ahuja Nishi, Dayi Zhu, and Ken Zhang. 2020. Smart Server Crash Prediction in Cloud Service Data Center. In *2020 19th IEEE Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems (ITherm)*. 1350–1355. https://doi.org/10.1109/ITherm45881.2020.9190321

[68] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. 2018. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *Proceedings of the European conference on computer vision (ECCV)*. 116–131.

[69] Abdulrahman Mahmoud, Siva Kumar Sastry Hari, Christopher W Fletcher, Sarita V Adve, Charbel Sakr, Naresh R Shanbhag, Pavlo Molchanov, Michael B Sullivan, Timothy Tsai, and Stephen W Keckler. 2021. Optimizing Selective Protection for CNN Resilience.. In *ISSRE*. 127–138.

[70] Sébastien Marcel and Yann Rodriguez. 2010. Torchvision the machine-vision package of torch. In *Proceedings of the 18th ACM international conference on Multimedia*. 1485–1488.

[71] Deepak M. Mathew, Martin Schultheis, Carl C. Rheinländer, Chirag Sudarshan, Christian Weis, Norbert Wehn, and Matthias Jung. 2018. An analysis on retention error behavior and power consumption of recent DDR4 DRAMs. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. 293–296. https://doi.org/10.23919/date.2018.8342023

[72] Fanruo Meng, Fateme S Hosseini, and Chengmo Yang. 2021. A self-test framework for detecting fault-induced accuracy drop in neural network accelerators. In *Proceedings of the 26th Asia and South Pacific Design Automation Conference*. 722–727.

[73] Jorge Castiñeira Moreira and Patrick Guy Farrell. 2006. *Essentials of error-control coding*. John Wiley & Sons.

[74] Marzieh Mozafari, Reza Farahbakhsh, and Noel Crespi. 2019. A BERT-based transfer learning approach for hate speech detection in online social media. In *International Conference on Complex Networks and Their Applications*. Springer, 928–940.

[75] Prashant J Nair, Vilas Sridharan, and Moinuddin K Qureshi. 2016. XED: Exposing on-die error detection information for strong memory reliability. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. Ieee, 341–353.

[76] Thai-Hoang Nguyen, Muhammad Imran, Jaehyuk Choi, and Joon-Sung Yang. 2021. Low-Cost and Effective Fault-Tolerance Enhancement Techniques for Emerging Memories-Based Deep Neural Networks. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 1075–1080. https://doi.org/10.1109/dac18074.2021.9586112

[77] Panagiota Nikolaou, Yiannakis Sazeides, Lorena Ndreu, and Marios Kleanthous. 2015. Modeling the implications of DRAM failures and protection techniques on datacenter TCO. In *Proceedings of the 48th International Symposium on Microarchitecture*. 572–584.

[78] Xuefei Ning, Guangjun Ge, Wenshuo Li, Zhenhua Zhu, Yin Zheng, Xiaoming Chen, Zhen Gao, Yu Wang, and Huazhong Yang. 2021. FTT-NAS: Discovering fault-tolerant convolutional neural architecture. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 26, 6 (2021), 1–24.

[79] Lois Orosa, Abdullah Giray Yaglikci, Haocong Luo, Ataberk Olgun, Jisung Park, Hasan Hassan, Minesh Patel, Jeremie S. Kim, and Onur Mutlu. 2021. A Deeper Look into RowHammer's Sensitivities:Experimental Analysis of Real DRAM Chipsand Implications on Future Attacks and Defenses. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(Micro '21)*. Association for Computing Machinery, New York, NY, USA, 1182–1197. https://doi.org/10.1145/3466752.3480069

[80] Elbruz Ozen and Alex Orailoglu. 2020. Just Say Zero: Containing Critical Bit-Error Propagation in Deep Neural Networks With Anomalous Feature Suppression. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 1–9.

[81] Elbruz Ozen and Alex Orailoglu. 2020. Low-Cost Error Detection in Deep Neural Network Accelerators with Linear Algorithmic Checksums. In *Journal of Electronic Testing: Theory and Applications (JETTA)*, Vol. 36. Issue 6. https://doi.org/10.1007/s10836-020-05920-2

[82] Elbruz Ozen and Alex Orailoglu. 2021. SNR: S queezing N umerical R ange Defuses Bit Error Vulnerability Surface in Deep Neural Networks. *ACM Transactions on Embedded Computing Systems (TECS)* 20, 5s (2021), 1–25.

[83] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[84] Arnab K Paul, Ahmad Maroof Karimi, and Feiyi Wang. 2021. Characterizing machine learning i/o workloads on leadership scale hpc systems. In *2021 29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 1–8.

[85] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM addressing for cross-cpu attacks. In *25th {USENIX$}$ security symposium (${$USENIX$}$ security 16)*. 565–581.

[86] Jonathan Ponader, Kyle Thomas, Sandip Kundu, and Yan Solihin. 2021. MILR: Mathematically induced layer recovery for plaintext space error correction of CNNs. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 75–87.

[87] Minghai Qin, Chao Sun, and Dejan Vucinic. 2017. Robustness of neural networks against storage media errors. *arXiv preprint arXiv:1709.06173* (2017).

[88] Mirco Ravanelli, Titouan Parcollet, Peter Plantinga, Aku Rouhe, Samuele Cornell, Loren Lugosch, Cem Subakan, Nauman Dawalatabad, Abdelwahab Heba, Jianyuan Zhong, Ju-Chieh Chou, Sung-Lin Yeh, Szu-Wei Fu, Chien-Feng Liao, Elena Rastorgueva, François Grondin, William Aris, Hwidong Na, Yan Gao, Renato De Mori, and Yoshua Bengio. 2021. SpeechBrain: A General-Purpose Speech Toolkit. arXiv:2106.04624 [eess.AS] arXiv:2106.04624.

[89] Netanel Raviv, Siddharth Jain, Pulakesh Upadhyaya, Jehoshua Bruck, and Anxiao Andrew Jiang. 2020. Codnn - robust neural networks from coded classification. In *2020 IEEE International Symposium on Information Theory (ISIT)*. IEEE, 2688–2693.

[90] Denise Rey and Markus Neuhäuser. 2011. *Wilcoxon-Signed-Rank Test*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1658–1659. https://doi.org/10.1007/978-3-642-04898-2_616

[91] Marco Rios, Flavio Ponzina, Giovanni Ansaloni, Alexandre Levisse, and David Atienza. 2022. Error Resilient In-Memory Computing Architecture for CNN Inference on the Edge. In *Proceedings of the Great Lakes Symposium on VLSI 2022*. 249–254.

[92] Aleksander Rydzewski and Pawel Czarnul. 2021. Human awareness versus Autonomous Vehicles view: comparison of reaction times during emergencies. In *2021 IEEE Intelligent Vehicles Symposium (IV)*. Ieee, 732–739.

[93] Christoph Schorn, Thomas Elsken, Sebastian Vogel, Armin Runge, Andre Guntoro, and Gerd Ascheid. 2020. Automated design of error-resilient and hardware-efficient deep neural networks. *Neural Computing and Applications* 32, 24 (2020), 18327–18345.

[94] Christoph Schorn, Andre Guntoro, and Gerd Ascheid. 2019. An Efficient Bit-Flip Resilience Optimization Method for Deep Neural Networks. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1507–1512. https://doi.org/10.23919/date.2019.8714885

[95] Art Sedighi and Milton Smith. 2019. Financial Market Risk. In *Fair Scheduling in High Performance Computing Environments*. Springer, 7–15.

[96] Taniya Siddiqua, Vilas Sridharan, Steven E. Raasch, Nathan DeBardeleben, Kurt B. Ferreira, Scott Levy, Elisabeth Baseman, and Qiang Guan. 2017. Lifetime memory reliability data from the field. In *2017 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. 1–6. https://doi.org/10.1109/dft.2017.8244428

[97] Kevin Siu, Dylan Malone Stuart, Mostafa Mahmoud, and Andreas Moshovos. 2018. Memory Requirements for Convolutional Neural Network Hardware Accelerators. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. 111–121. https://doi.org/10.1109/iiswc.2018.8573527

[98] Joseph Sloan, Rakesh Kumar, and Greg Bronevetsky. 2013. An algorithmic approach to error localization and partial recomputation for low-overhead fault tolerance. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 1–12. https://doi.org/10.1109/dsn.2013.6575309

[99] David Snyder, Daniel Garcia-Romero, Alan McCree, Gregory Sell, Daniel Povey, and Sanjeev Khudanpur. 2018. Spoken Language Recognition using X-vectors. In *Odyssey 2018*. 105–111.

[100] Chenchen Song. 2022. Design and Application of Financial Market Option Pricing System Based on High-Performance Computing and Deep Reinforcement Learning. *Scientific Programming* 2022 (03 Mar 2022), 8525361. https://doi.org/10.1155/2022/8525361

[101] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B. Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. 2015. Memory Errors in Modern Systems: The Good, The Bad, and The Ugly. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey) *(Asplos '15)*. Association for Computing Machinery, New York, NY, USA, 297–310. https://doi.org/10.1145/2694344.2694348

[102] Vilas Sridharan and Dean Liberty. 2012. A study of DRAM failures in the field. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. Ieee, 1–11.

[103] Michael B Sullivan, Nirmal Saxena, Mike O'Connor, Donghyuk Lee, Paul Racunas, Saurabh Hukerikar, Timothy Tsai, Siva Kumar Sastry Hari, and Stephen W Keckler. 2021. Characterizing and Mitigating Soft Errors in GPU DRAM. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 641–653.

[104] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 1–9. https://doi.org/10.1109/cvpr.2015.7298594

[105] Dahong Tang and Weimin Zhang. 2007. A Sub-matrix Method For Multiple Reed-solomon Erasure Coding.

[106] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. 2010. Collecting Performance Data with PAPI-C. In *Tools for High Performance Computing 2009*, Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 157–173.

[107] Aniruddha N Udipi, Naveen Muralimanohar, Rajeev Balsubramonian, Al Davis, and Norman P Jouppi. 2012. LOT-ECC: Localized and tiered reliability mechanisms for commodity memory systems. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 285–296.

[108] Aniruddha N. Udipi, Naveen Muralimanohar, Rajeev Balsubramonian, Al Davis, and Norman P. Jouppi. 2012. LOT-ECC: Localized and tiered reliability mechanisms for commodity memory systems. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. 285–296. https://doi.org/10.1109/isca.2012.6237025

[109] Pulakesh Upadhyaya, Xiaojing Yu, Jacob Mink, Jeffrey Cordero, Palash Parmar, and A Jiang. 2019. Error correction for hardware-implemented deep neural networks. In *Proc. Non-Volatile Memories Workshop*.

[110] Sunil Vadera and Salem Ameen. 2022. Methods for pruning deep neural networks. *IEEE Access* 10 (2022), 63280–63300.

[111] Jonathan Walker. 2016. Peak Car Ownership: The Market Opportunity Of Electric Automated Mobility Servicesthe Market Opportunity Of Electric Automated Mobility Services. https://rmi.org/insight/peak-car-ownership-report/. Accessed: 2021-11-01.

[112] P. Warden. 2018. Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition. In *ArXiv e-prints*. arXiv:1804.03209 [cs.CL] https://arxiv.org/abs/1804.03209

[113] Max A Woodbury. 1950. *Inverting modified matrices*. Statistical Research Group.

[114] Fan Yao, Adnan Siraj Rakin, and Deliang Fan. 2020. Deephammer: Depleting the intelligence of deep neural networks through targeted chain of bit flips. In *29th USENIX Security Symposium (USENIX Security 20)*. 1463–1480.

[115] Jianbo Ye, Xin Lu, Zhe Lin, and James Z Wang. 2018. Rethinking the smaller-norm-less-informative assumption in channel pruning of convolution layers. In *arXiv preprint arXiv:1802.00124*.

[116] Doe Hyun Yoon and Mattan Erez. 2010. Virtualized and flexible ECC for main memory. In *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems*. 397–408.

[117] Kai Zhao, Sheng Di, Sihuan Li, Xin Liang, Yujia Zhai, Jieyang Chen, Kaiming Ouyang, Franck Cappello, and Zizhong Chen. 2021. FT-CNN: Algorithm-Based Fault Tolerance for Convolutional Neural Networks. In *IEEE Transactions on Parallel and Distributed Systems*, Vol. 32. 1677–1689. https://doi.org/10.1109/tpds.2020.3043449

Appendix: Artifact Description/Artifact Evaluation

ARTIFACT DOI

https://doi.org/10.5281/zenodo.6463989

ARTIFACT IDENTIFICATION

Hereby, we present the software artifacts associated with the submission to Supercomputing 2023, *Structural Coding: A Low-Cost Scheme to Protect CNNs from Large-Granularity Memory Faults*. That includes the open source implementation of the method as well as its evaluation.

These artifacts help the reader to reproduce the functional results accurately. However, the performance results may vary in their scales depending on the deployment platform without changing the big picture of the comparison between the different techniques.

REPRODUCIBILITY OF EXPERIMENTS

1 DESCRIPTION

1.1 Check-list (artifact meta information)

- **Algorithm:** Structural Coding.
- **Program:** Python code.
- **Binary:** Docker/singularity image.
- **Data set:** ImageNet, Google Speech Commands, Mozafari et al., and our fault injection runs.
- **Run-time environment:** Slurm, Ubuntu.
- **Hardware:** Cedar cluster on Compute Canada (now called Alliance), PC.
- **Run-time state:** Run-time overhead controlled variables
- **Output:** Memory footprint overhead, run-time overhead, Silent Data Corruption (SDC) rates.
- **Experiment workflow:** Git clone, configure directory, run a script.
- **Experiment customization:** Yes, can be applied to convolutional neural networks.
- **Publicly available?:** Yes.

1.2 How software can be obtained

The source code is available at github:

https://github.com/DependableSystemsLab/structural-coding

The main components of the source code are described in Table 1.

Source[:Line]	Description
sc.py	Coding technique implementation
linearcode/fault.py	Memory fault model
injection.py:712	Protected convolution layer
injection.py:810	Protected linear layer
linearcode/map.py	Top-level fault injection script
linearcode/analyze.py	Result aggregation
linearcode/*_overhead.py	Overhead measurements
DDR3/*.xlsx	DRAM energy models

Table 1: Source code components

2 EXPERIMENT WORKFLOW

The results are obtained through two steps. In the first step, we run the experiments and log the measurements. We expect that to take 240 CPU days. If this become problematic on Chameleon, they can skip to the latter step and the authors are able to provide a dataset containing the fault injection experiment raw data. In the second step we aggregate those logs and write the digested results. The rest of the instructions here assume that you are working on an Ubuntu 20.04 machine with SSH access to a slurm cluster and with docker and singularity installed.

Create a directory on the cluster and specify its SCP address in the following script (addresses to files are relative to the repository root):

```
/build_singularity.sh
```

Run the script. This script will create a singularity image from the docker image and copies it to the cluster. It will also copy the batch job submission script. Then you need to go to the specified directory in the cluster and run the following command to spawn the fault injection experiments:

```
sbatch script.sh
```

It will take $\tilde{2}40$ CPU days to run the experiments. Once the experiments are done bring the results from the results folder back to your local machine. Create two directories named *home* and *data* along the *results* and run the following command to aggregate the results:

```
docker run -v home/:/root/ \
 -v results/:/code/linearcode/results/ \
 -v data/:/code/thesis/data/ \
 --env SHARD=ad dsn2022paper165/sc \
 ./pseudo_slurm_reduce.sh 12
```

The digest results will be written to the *data* directory. To run and aggregate the overhead results on your local machine run the following command:

```
./ad_overhead.sh
```

Again, the digest results will be written to the *data* directory. The SDC rate files have two columns, first one showing the SDC rate and the second showing confidence intervals. The name of the files are indicative of the model and fault model.

3 EVALUATION AND EXPECTED RESULT

The randomness in the SDC rate results is controlled via a fixed seed. So you will observer the same SDC rates as in the graphs. The results for each combination of fault model and protection will be in the *data* folder.

The memory footprint overhead results are deterministic and you will observe the same results as in the paper. However, the time overhead results may vary based on the configuration of your local machine. You will be able to confirm the overall order between the average of results for RADAR, MILR, and SC regarding the run-time.

4 EXPERIMENT CUSTOMIZATION

Applying structural coding on a typical CNN with near zero effort:

```
# linearcode/autonomy.py
from torchvision.models import resnet50
from linearcode.protection import apply_sc_automatically

model = resnet50(pretrained=True)
print(model)

model = apply_sc_automatically(model, n=256, k=32)
print(model)
```

5 NOTES

5.1 Run-time overhead controlled variables

We try to minimize the variability of the time measurements by doing the following:

- Closing all other applications.
- Flushing the disk cache.
- Running the inference 10 times.

5.2 Energy overhead models

The DRAM energy consumption values in *linearcode/analyze.py* come from Micron energy models that can be found under the *DDR3* for each neural network.

AUTHOR-CREATED OR MODIFIED ARTIFACTS:

Artifact 1

Persistent ID: `https://github.com/DependableSystemsLab/structural-coding`

Artifact name: Source Code

Citation of artifact: Ali Asgari. (2023). DependableSystemsLab/structural-coding: Artifacts Functional (Version sc2023functional). Zenodo. https://doi.org/10.5281/zenodo.6463989

Artifact 2

Persistent ID: `https://github.com/DependableSystemsLab/structural-coding/releases/download/sc2022functional/datasets.zip`

Artifact name: Datasets

Citation of artifact: Ali Asgari. (2023). DependableSystemsLab/structural-coding: Artifacts Functional (Version sc2023functional). Zenodo. https://doi.org/10.5281/zenodo.6463989

Reproduction of the artifact with container: The artifact is also available as a docker container at https://hub.docker.com/repository/docker/dsn2022paper165/sc. The usage is explained in the artifact description.

ARTIFACT DEPENDENCIES REQUIREMENTS

5.3 Experiment Hardware

For the fault injection experiments we used Cedar high performance computing cluster. We submit fault injections as batch jobs allocating a single core and 16GB of memory. We limit the execution time of the jobs to 18 hours. The cluster uses *slurm* to orchestrate the jobs. More details about Cedar can be found in the following link: https://docs.alliancecan.ca/wiki/Cedar

For the overhead measurements we used a Desktop PC. The relevant details of this machine is reported in Table 2.

Aspect	Specification
CPU	Intel(R) Core(TM) i7-4930K CPU @ 3.40GHz
Memory size	16 GB
Memory bandwidth	6400 MiB/s (128 MiB array benchmark)
Disk bandwidth	167.33 MiB/s (502 MB benchmark)
Disk read latency	5.41 milliseconds (iostat -x)
Operating system	Ubuntu 20.04.3 LTS

Table 2: Desktop PC hardware details

Python Dependency	Version
torchvision	0.10.1
matplotlib	3.5.1
python-papi	5.5.1.5
galois	0.0.24

Table 3: Python dependencies

5.4 Required Hardware

Fault injection experiment code requires a single core of CPU and 16 GB of memory. Performance measurements require a CPU with performance counters for floating point and DRAM operations.

5.5 Software Requirements

The experiments need Ubuntu 20.04.3 LTS or a similar operating system. If you choose to run the our docker image, you will not need to set up the software dependencies and Docker itself will be a requirement. Otherwise, the experiments require Python version 3 and the packages as listed in Table 3.

The workflow of the experiments as described, require a Slurm cluster. For the purpose of Artifact Evaluation, the reviewer should reserve and spawn 64 (recommended) single core machines from Chameleon cloud (240 CPU days in total are required) with 16 GB memory and a shared file system. Then, they should set up a Slurm cluster on the spawned nodes. From there, their workflow will be the same as what we did to conduct our experiments as described earlier.

5.6 Datasets

We have chosen the datasets so as they cover multiple media formats, are widely used in the research community, and are easily accessible.

All the dataset inputs used for evaluation are contained in the docker image. They are also available at https://github.com/DependableSystemsLab/structural-coding/releases/download/sc2023functional/datasets.zip . All the datasets are from public sources available for download.

ARTIFACT INSTALLATION DEPLOYMENT PROCESS

6 INSTALLATION AND SMALL SCALE DEMO

Please note that lines that are starting with a $ are showing the commands to enter. Those that did not fit within one line are extended to the next line with backslash. The rest show the expected output that you need to compare the *accuracy* or *correct* with the ones reported here.

6.1 Running Demo via Docker

Running the *resnet50* network without protection with no fault:

```
$ docker run \
    --env CONSTRAINTS="{'dataset': 'imagenet', 'model': 'resnet50', \
 'sampler': 'tiny', 'flips': 0, 'protection': 'none'}" \
    --env PRINT_STAT=1 dsn2022paper165/sc python map.py
Downloading: "https://download.pytorch.org/models/resnet50-19c8e357.pth"
 to /root/.cache/torch/hub/checkpoints/resnet50-19c8e357.pth
100.0%
Done with batch 0 after injection
{'injection': 0, 'model': 'resnet50',
 'quantization': False, 'sampler': 'tiny', 'dataset': 'imagenet',
'flips': 0, 'protection': 'none'}
accuracy 0.75 correct 12 all 16
loss 0.060732901096343994
```

Running the *resnet50* network without protection with 'row' fault model:

```
$ docker run \
    --env CONSTRAINTS="{'dataset': 'imagenet', 'model': 'resnet50', \
 'sampler': 'tiny', 'flips': 'row', 'protection': 'none'}" \
    --env PRINT_STAT=1 dsn2022paper165/sc python map.py
Downloading: "https://download.pytorch.org/models/resnet50-19c8e357.pth"
 to /root/.cache/torch/hub/checkpoints/resnet50-19c8e357.pth
100.0%
Done with batch 0 after injection
{'injection': 0, 'model': 'resnet50',
 'quantization': False, 'sampler': 'tiny', 'dataset': 'imagenet',
'flips': 'row', 'protection': 'none'}
Injecting 1193 faults at granularity 16
accuracy 0.0 correct 0 all 16
loss 4708924.5
```

Running the *resnet50* network with 'sc' (Structural Coding) protection with 'row' fault model:

```
$ docker run \
    --env CONSTRAINTS="{'dataset': 'imagenet', 'model': 'resnet50', \
 'sampler': 'tiny', 'flips': 'row', 'protection': 'sc'}" \
    --env PRINT_STAT=1 dsn2022paper165/sc python map.py
Downloading: "https://download.pytorch.org/models/resnet50-19c8e357.pth"
 to /root/.cache/torch/hub/checkpoints/resnet50-19c8e357.pth
100.0%
Done with batch 0 after injection
{'injection': 0, 'model': 'resnet50',
 'quantization': False, 'sampler': 'tiny', 'dataset': 'imagenet',
'flips': 'row', 'protection': 'sc'}
Injecting 1193 faults at granularity 16
accuracy 0.75 correct 12 all 16
loss 0.061373598873615265
```

Running the *resnet50* network with 'milr' (MILR) protection with 'row' fault model:

```
$ docker run \
    --env CONSTRAINTS="{'dataset': 'imagenet', 'model': 'resnet50', \
 'sampler': 'tiny', 'flips': 'row', 'protection': 'milr'}" \
    --env PRINT_STAT=1 dsn2022paper165/sc python map.py
Downloading: "https://download.pytorch.org/models/resnet50-19c8e357.pth"
 to /root/.cache/torch/hub/checkpoints/resnet50-19c8e357.pth
100.0%

Done with batch 0 after injection
{'injection': 0, 'model': 'resnet50',
 'quantization': False, 'sampler': 'tiny', 'dataset': 'imagenet',
'flips': 'row', 'protection': 'milr'}
Injecting 1193 faults at granularity 16
accuracy 0.75 correct 12 all 16
loss 0.06021653115749359
```

6.2 Run using Python virtual environment

Assuming ubuntu linux, create a virtual environment and activate it:

```
python3.8 -m venv venv
source venv/bin/activate
pip install --upgrade pip
```

Within the root directory of the code, install the project requirements:

```
pip install -r requirements.txt
```

Set the PYTHONPATH environment variable:

```
export PYTHONPATH=`pwd`
```

Navigate to the experiment subdirectory:

```
cd linearcode
```

You can run the example experiment commands:

```
$ export CONSTRAINTS="{'dataset': 'imagenet', 'model': 'resnet50', \
 'sampler': 'tiny', 'flips': 'row', 'protection': 'milr'}"
$ export PRINT_STAT=1; python map.py
{'injection': 0, 'model': 'resnet50',
 'quantization': False, 'sampler': 'tiny', 'dataset': 'imagenet',
'flips': 'row', 'protection': 'milr'}

Injecting 1193 faults at granularity 16
Done with batch 0 after injection
accuracy 0.75 correct 12 all 16
loss 0.06021653115749359
```

OTHER NOTES

The authors are still trying to validate the described experiment workflow on Chameleon cloud and the described commands may need minor changes. If the fault injection experiments turn out to take a lot of time due to low availability of resources, the authors are able to publish a dataset containing the fault injection raw data and the reviewers can reproduce the results from that dataset.