Neural network based silent error detector

Chen Wang
Department of Computer Science
University of Illinois at Urbana-Champaign
chenw5@illinois.edu

Nikoli Dryden
Department of Computer Science
University of Illinois at Urbana-Champaign
dryden2@illinois.edu

Franck Cappello
MCS Division
Argonne National Laboratory
cappello@anl.gov

Marc Snir
Department of Computer Science
University of Illinois at Urbana-Champaign
snir@illinois.edu

***Abstract*—As we move toward exascale platforms, silent data corruptions (SDC) are likely to occur more frequently. Such errors can lead to incorrect results. Attempts have been made to use generic algorithms to detect such errors. Such detectors have demonstrated high precision and recall for detecting errors, but only if they run immediately after an error has been injected. In this paper, we propose a neural network detector that can detect SDCs even multiple iterations after they were injected. We have evaluated our detector with 6 FLASH applications and 2 Mantevo mini-apps. Experiments show that our detector can detect more than 89% of SDCs with a false positive rate of less than 2%.**

***Index Terms*—silent data corruption, fault tolerance, exascale computing**

I. Introduction

In exascale HPC computations, silent soft hardware errors can no longer be ignored as they become more frequent for a variety of reasons: larger number of components; higher vulnerability of smaller transistors; the cost of error detection logic; and the possible use of sub-threshold logic, in order to reduce energy consumption. These soft errors, if not detected, can lead to incorrect final results [1]. While statistics on the frequency of Silent Data Corruption (SDC) are rare and hard to obtain, there is strong evidence that SDC has affected HPC systems in the past [2] and experimental data to suggest the problem will get worse [3].

A naive way to ensure a correct final result is to simply run the same application multiple times. However, for large scale applications, it is expensive to do so due to the huge consumption of computational resources. This problem has also motivated a significant amount of research on Algorithm-Based Fault Tolerance (ABFT), where algorithm-specific methods are used to detect errors in intermediate data and repair them [4]–[7]. The main disadvantage of these methods is that they need to be developed separately for each algorithm; ABFT methods are known only for a subset of important numerical computations.

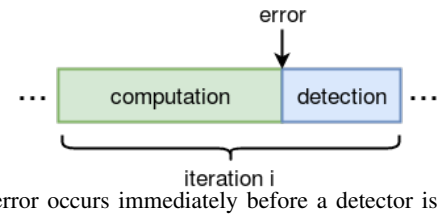
Transient undetected hardware errors can manifest themselves in multiple ways: (1) They may result in a detected software error, such as a segmentation fault; (2) they may stay undetected and cause a wrong answer; or, (3) they may be "benign" and lead to an acceptable answer. The latter is frequent with iterative numerical algorithms: Any answer that is within an error bound off from the exact answer is acceptable; errors in least significant bits may not propagate above the error bound; and iterative algorithms tend to smooth local perturbations [8]. Recent work on the use of compression or low-precision arithmetic in iterative algorithms leverage these phenomena [9]–[11]. Error detectors need to focus only on the second type of errors.

Moreover, different designs have been proposed for generic error detectors that work for a large family of iterative algorithms – algorithms where values at a point are repeatedly updated using values at neighboring points [12]–[15]. They use prediction-based techniques such as curve fitting or autoregressive-moving-average models and rely on the fact that an error normally manifests itself as a large gap between a point value and the value of neighbors or between current and previous point value. These detectors ignore small errors that are unlikely to lead to an incorrect answer and focus on detecting large errors. "Small" is defined as an application specific threshold, the *impact error bound*, which requires prior, expensive experimentation. More importantly, these detectors need to be run at each iteration, which significantly increases their overhead. Furthermore, as illustrated in Figure 1a, these error detectors have been tested by applying them immediately after an error is injected. However, in practice, errors can happen at any time, as in Figure 1b. Applications may smooth errors within an iteration, so current tests may not reflect the effectiveness of these detectors even when they are run once at each iteration.

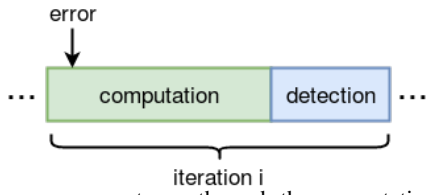
The weaknesses of current methods lead us to seek error detectors with the following properties:

- They can detect errors in an iterative algorithm many iterations after the error occurred. For example, they could be run just before a checkpoint is taken, to ensure the checkpoint is correct or trigger a restart.
- They are robust with respect to the actual choice of the impact error bound.
- They can be application-specific, but no knowledge of the specific application is needed in order to develop the detector.
- They are efficient and provide good coverage.

Fig. 1: The error occurs during or after one iteration.



(a) An error occurs immediately before a detector is run. This common assumption simplifies error detection.



(b) An error occurs part-way through the computation within a single iteration. The application may smooth the error to some extent, making detection more difficult.

In this paper, we first study how errors persist over multiple iterations. We then propose a neural network-based SDC detector that satsifes the requirements above. The intuition behind our approach is that, even as the magnitude of a point error may decrease over successive iterations, the error propagates to nearby mesh points. A detector that integrates information from a neighborhood of the point where the error occurred can identify it even after it was "smeared".

The major contributions of this paper are summarized as follows:

- We study and categorize the behavior of silent errors in different HPC applications. We show that for certain types of applications, large silent errors persist even after hundreds of iterations, which gives us the chance to detect them without running the detector at every iteration.
- Our approach does not rely on the impact error bound, and the detection algorithm is local, which means multiple detectors can be run concurrently on subsets of the data and no communication is required.
- We evaluate our algorithm with 6 FLASH applications and 2 Mantevo mini-apps. Experiments show that our detector achieves 89%+ recall in all applications with a false positive rate of less than 2%.
- We also demonstrate that our detector is able to detect errors many iterations after occurrence.
- We compare our detector to existing approaches and show that ours performs significantly better.

The rest of the paper is organized as follows. In Section II, we briefly review silent data corruptions along with the impact of bit flips in floating point values. In Section III, we present experimental results of the behavior of silent errors in different kinds of HPC applications. Then we describe our detection algorithm in Section IV. The evaluation is presented in Section V. The related work is discussed in Section VI followed by the conclusion in Section VII.

II. Background

A. Silent data corruptions

Energetic particles from cosmic radiation can invert the state of transistors [16], [17]. Manufacturing defects can lead to the same effect. One consequence is that these faults produce soft errors that can cause a silent data corruption (SDC)—i.e., an undetected erroneous deviation in system/application state [18]. Some corruptions can be ignored as they are attenuated by the algorithm, and successful convergence to a valid output still occurs. Some will cause software exceptions and will be detected. In other cases, they can lead to unacceptable errors in the final result. We are interested in detecting the errors that belong to the last category.

The impact of SDC is algorithm dependent. As an example, a small error introduced in a heat diffusion program will be smoothed as the algorithm iterates. However, the error may persist if it was injected in a shock hydrodynamics application. We will discuss more details about SDC propagation in Section III.

Ideally, an SDC detector should have these properties:

- *Low false negatives:* A false negative is an SDC that was not detected and led to a wrong result. Detecting an SDC that does not corrupt the final result should be considered a false positive.
- *Low false positives:* A false positive is an event that is wrongly detected as an SDC. These are less critical than false negatives, as they only affect performance (e.g. by requiring an unnecessary restart from a checkpoint).
- *Low overhead:* The detector should not significantly increase the application runtime and should have negligible memory footprint.
- *Ability to detect errors many cycles after its occurrence:* This is critical for the purpose of lowering overheads.

The most common error recovery mechanism in HPC is checkpoint-restart. Checkpoint-restart works only if errors occurring during a checkpoint interval are detected before the next checkpoint is taken. Thus, an ideal SDC detector should be able to run at the end of a checkpoint interval and detect errors that occurred during this interval; its running time should be low compared to checkpoint time.

The optimal checkpoint interval is approximately equal to $\sqrt{2T_fT_c}$, where T_f is the mean time between failures (MTBF) and T_c is checkpoint time [19]; when this interval is used, the fraction of the total time spent on checkpoint/restart is $\sqrt{2T_c/T_f}$. Thus, for practical systems, the checkpoint interval will be much smaller than the MTBF, and multiple errors in an interval will be rare. Our detectors use only data in a small window and the probability of multiple errors within the same window is even lower. Therefore, we focus on single error detection.

We assume that the same code will be run many times, for different input values. Thus, it is acceptable to train an SDC detector offline for a particular code and use the trained detector during actual runs of that code. Note that, while each SDC detector is application-specific, our methodology for

TABLE I: IEEE 754 floating point layout

	Sign	Exponent	Fraction
Single precision	1[31]	8[30-23]	23[22-00]
Double precision	1[63]	11[62-52]	52[51-00]

TABLE II: The impact of bit flips in different locations on the value 1.0 in single-precision floating point.

Flipped bit	Value	Deviation
-	1.0	0
31	-1.0	2
30	Infinity	Infinity
29	5.421011E-20	≈ 1
27	1.5258789E-5	0.99998474121
25	0.0625	0.9375
22	1.5	0.5
20	1.25	0.25
18	1.03125	0.03125
10	1.0004883	4.883e-4
5	1.0000038	3.8e-6
0	1.0000001	1e-7

creating the detector is application-independent. We therefore pay a one-time cost for training a detector for an application, which will be amortized over many runs of the application. In particular, since many of the applications most concerned with SDCs are used for large, long-running production runs, this cost should be minor.

B. Bit flips in floating point

IEEE Standard 754 [20] floating point is the most common representation today for real numbers on computers. IEEE floating point numbers have three basic components: the sign, the exponent, and the mantissa. The mantissa is composed of the fraction and an implicit leading digit. The exponent base (2) is implicit and need not be stored.

Table I shows the layout for single (32-bit) and double (64-bit) precision floating-point values. The table shows the number of bits for each field (bit ranges are in square brackets, 00 is the least-significant bit). Compared to the exponent, a bit flip in the mantissa introduces a relatively small change compared to the original value. Table II shows an example of how a single precision floating point (1.0) changes with different bit flips. In general, we could say that flips occurring in higher bits do more damage than flips in lower bits.

In this paper, we only consider the case of one bit-flip at a time. We hypothesize that more bit-flips could be no harder to detect since the difference introduced by multiple bit-flips is larger than one bit-flip.

III. SDC Propagation

When an error is introduced, the corruption can sometimes be detected by the application itself due to programmatic or algorithmic properties, e.g. a system detectable event like a segmentation fault, or an invalid data value such as a negative speed of sound. Those errors can cause the application to crash and are not silent anymore. On the other hand, modern hardware supports redundancy and techniques such as error correcting codes (ECC) that are able to detect some soft errors and prevent them from affecting the computation state [18]. In this research, we only focus on those errors that lead to silent corruptions that are neither detected by hardware nor by applications.

TABLE III: Error positions

App	Error bit position	Value changes
Sod	12	From: 0.99999999999999989 To: 0.74999999999999989
BrioWU	11	From: 1.0 To: 0.5
BlastBS	16	From: 0.91620054602679901 To: 0.93182554602679901
OrszagTang	8	From: 0.69874154473510708 To: 0.002729459159121512

Detection is normally more effective when errors are contained and affect only part of a computation's state, but the impact and propagation of SDC are algorithm/solver dependent. To get a better understanding of how SDC propagates in HPC applications and how silent errors affect the final results, we perform several experiments on a representative subset of the applications we evaluate later (see Table VII for an overview). Results and movies of more applications can be found on this website[1].

We set the data size to 480×480 for all applications. Except the change of mesh sizes, the initial conditions and configurations for all applications are unchanged. Each application runs 100 iterations after the error injection. We randomly flip one bit in the double precision variable density for all applications. Exact error positions are given in Table III along with the corresponding value changes. The bit position is indexed from MSB(0) to LSB(63). Because some errors (especially the ones that only introduce a small deviation) are difficult to see, in Figure 2 we show the difference between the error free simulation and the corrupted simulation for all applications.

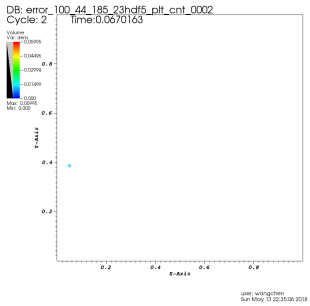
As we can see, errors tend to propagate locally as the computation goes on. One exception is OrszagTang [21], where the error affected much of the data. This observation shows that detectors can typically be run locally, and do not need to view the entire dataset. This also illustrates a simple way to achieve concurrency in detection: We run multiple identical detectors concurrently, each on a subset of the entire data.

Another important observation is that for most applications considered in this paper, especially for shock hydrodynamics applications, errors can persist for a long time (still visible after hundreds of iterations). This long-lasting property of SDC shows the potential to detect errors many iterations after they occur. Based on this observation, we are encouraged to design a neural network detector that learns to recognize patterns of SDC propagation.

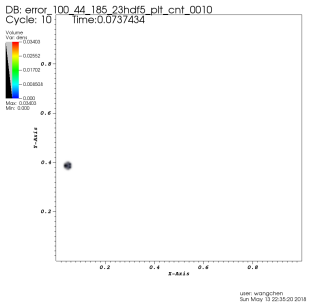
IV. Method

We treat the problem of detecting SDCs as a binary classification problem and train a convolutional neural network [22],

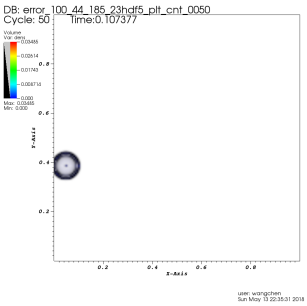
[1] http://chenw5.web.engr.illinois.edu/flash.html



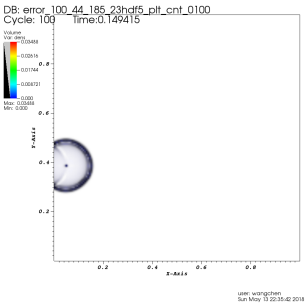
(a) Sod. 2 iterations after the error injection



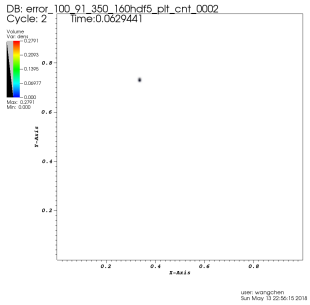
(b) Sod. 10 iterations after the error injection



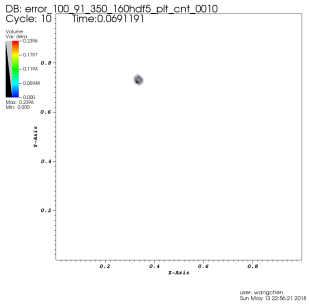
(c) Sod. 50 iterations after the error injection



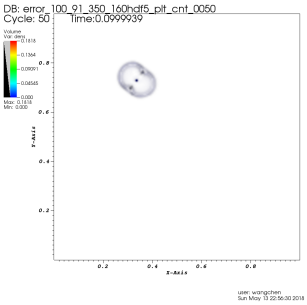
(d) Sod. 100 iterations after the error injection



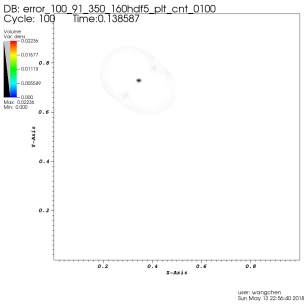
(e) BrioWu. 2 iterations after the error injection



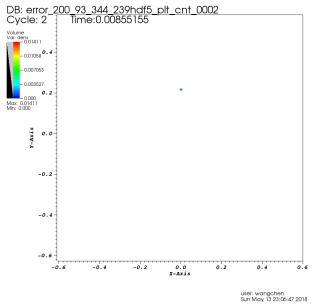
(f) BrioWu. 10 iterations after the error injection



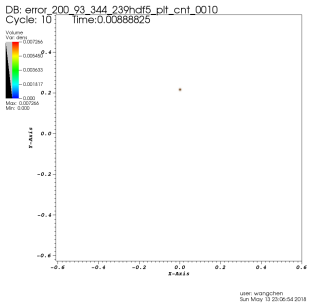
(g) BrioWu. 50 iterations after the error injection



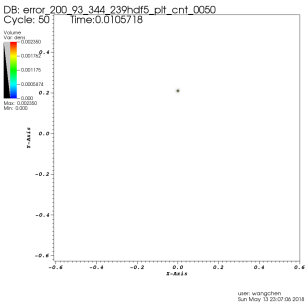
(h) BrioWu. 100 iterations after the error injection



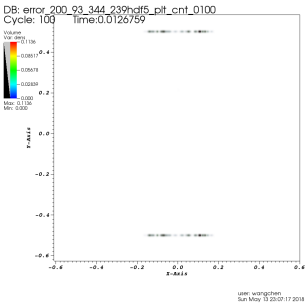
(i) BlastBS. 2 iterations after the error injection



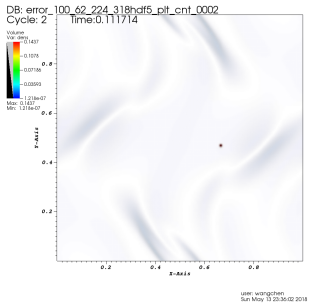
(j) BlastBS. 10 iterations after the error injection



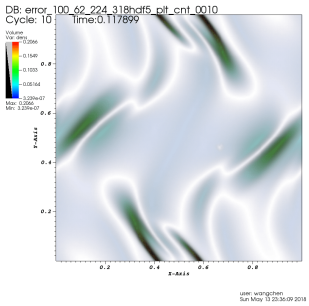
(k) BlastBS. 50 iterations after the error injection



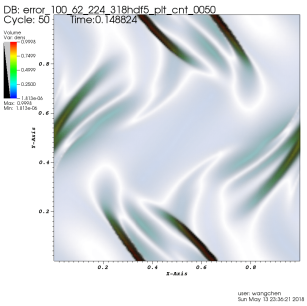
(l) BlastBS. 100 iterations after the error injection



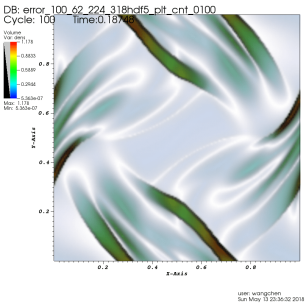
(m) OrszagTang. 2 iterations after the error injection



(n) OrszagTang. 10 iterations after the error injection



(o) OrszagTang. 50 iterations after the error injection



(p) OrszagTang. 100 iterations after the error injection

Fig. 2: SDC propagation in Sod (a-d), BrioWu (e-h), Blast (i-l) and OrszagTang (m-p). Because the errors are hard to see along with the simulation background, we show the difference between the faulty simulation and fault free simulation. Errors are still visually detectable many iterations after injection, and typically (except OrszagTang) expand fairly slowly throughout the domain.

[23] to solve it. The input data to the network is the numerical state of the simulation, which we think of as an "image" where each channel is a state variable (e.g. density, energy). Our network is then trained to determine whether, for a given input application state, an error is present in the data. This is in contrast to most recent work [14], [15], which uses curve-fitting schemes to detect anomalous data points. The resounding success of modern convolutional neural networks on image classification problems (e.g. [24]–[26]), combined with the visual distinctiveness of the errors shown in Section III, motivates our choice to use them here.

In this Section, we first discuss the error model, i.e., the problem that our detector is designed to address. Then we describe our neural network architecture and the training process.

A. Error Model

We assume soft errors that lead to SDC involve bit-flips in simulation data (e.g. density, energy), which is stored in a floating-point format. Only a single bit-flip is considered in most experiments, since the probability of multiple bit errors occurring simultaneously is low, and the magnitude of the error is dominated by the most significant corrupted bit, in general. We ignore bit-flips that result in NaNs, since these can be detected with standard hardware features (e.g. by enabling floating point exceptions).

Bit flips in low-order bits result in small perturbations of data. Experiments described in Section V-D indicate that errors in the the last 43 bits usually result in deviations to the final result that are within the error bound of the method used. Therefore, we focus on detecting errors that affect only the first 21 bits out of a 64 bits floating value.

B. Network architecture

Our neural network is a LeNet- and AlexNet-inspired [27], [28] architecture with some modern improvements. We use blocks of convolution, batch normalization [29], and ReLU activations, some of which are followed by max pooling, three fully-connected layers with dropout [30], and a sigmoid output for prediction. In total there are eight learned layers (excluding batch normalization). The convolutional layers learn to extract relevant features from the simulation data, while the fully-connected layers learn to classify the features. The pooling layers are overlapping, and we do downsampling with strided convolution.

We use this architecture for all our experiments, but a separate network is trained for each application as described below.

Due to the fully-connected layers, this architecture must be trained on fixed-sized inputs. However, we do not want our network to be limited to simulations of a fixed size. To overcome this, we split the input into fixed-sized windows (currently 60×60), and train on these data. These windows are slightly overlapped to avoid boundary effects from convolution. We treat an error as being present if the network reports an error in any window for data from a given iteration. This decomposition is natural for many HPC applications, where the data is spatially partitioned over many compute nodes, and it would be expensive to transfer all the data to a single node to detect errors. It is also natural in that errors in iterative codes often propagate slowly and affect only one window. Note that checking each window can be done independently, and this requires no communication. If necessary, we could also adapt standard techniques from computer vision to process arbitrarily-sized windows with the same network, such as fully-convolutional networks [31].

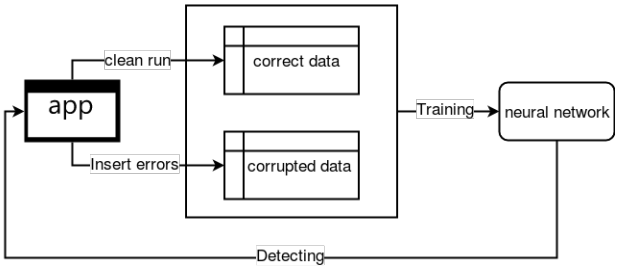


Fig. 3: System workflow: The application is used to generate training samples that are either clean or have artificial errors injected. A neural net is then trained on this dataset, and can then be used to detect errors in the application.

C. Training

An overview of our training workflow is given in Figure 3. Here we describe how we collect our training data, and give some details on the training process. Training is done once per application, and its cost is then amortized over many runs of the application.

1) Data: Our training dataset consists of two classes of data: clean and corrupted. Each sample consists of the state variables of the application, e.g. floating point vectors of physical quantities. The choice of which variables to protect is made by the user; however, using more variables may enable the neural network to use correlations between different quantities to better detect errors.

The clean data is easy to collect. We can simply run an error-free simulation and save the needed variables at each iteration by leveraging the application's checkpointing mechanisms. In order to achieve sufficient diversity in the data, we use many random initial conditions for the simulation problem, selected within a pre-determined realistic range of values.

The corrupted data can be collected similarly, but we augment the application with a mechanism to inject errors. We use this to inject errors at random iterations, into random locations in random state variables, by flipping a random bit within a range. We then run the application as usual and collect the data as the simulation runs on the corrupted state. This is typically a simple modification and imposes almost no overhead. An alternative approach to injecting errors is to corrupt checkpoint data and then restart the application with it. This approach enables us to collect essentially unlimited training data from an application and is very easy to automate.

Our testing dataset is collected similarly, except that we use different initial conditions for this data so that training data and testing data are distinct.

We can further subdivide our datasets based on how many iterations have passed since an error was injected. We call the dataset containing clean data and corrupted data from up to k iterations after the error was injected the *k-propagation dataset*. For example, the 0-propagation dataset contains clean data and corrupted data that had an error injected at that particular iteration (and hence has not propagated at all). Our expectation is that training on k-propagation datasets ($k > 0$) will help improve network accuracy when performing detection many iterations after an error is introduced.

2) Training details: The network uses a binary cross-entropy loss function and is trained with the Adam optimizer [32]. The learning rate is 0.00001 and the mini-batch size is 64 samples. Training is done in the PyTorch framework [33]. We use the same numerical precision for the network parameters as the input data from the application. This avoids rounding the application data in the detector, e.g. from double precision to single or half precision. We used these same settings, except number of epochs, for every application; doing hyperparameter tuning for individual applications is likely to result in improved performance over the results here.

V. Evaluation

A. Experimental Setup

We perform our experiments on Blue Waters, a Cray supercomputer managed by the National Center for Supercomputing Applications and supported by the National Science Foundation and the University of Illinois. Each compute node has 2 AMD 6276 Interlagos CPUs and 64 GB of RAM. The neural network is trained and evaluated on Nvidia DGX-1 at Argonne JLSE. The DGX-1 is equipped with 8 Tesla P100 GPUs.

Table VII shows the applications we use in our evaluation from FLASH4.4 [34] and Mantevo [35] package. We protect *state variables* such as density, pressure, velocity, etc. for each application.

B. Generating the training and testing datasets

1) Clean dataset: To gather the clean dataset, we run each application for 1000 iterations with 10 different cases (initial conditions). We output variables we want to protect at every 5 iterations. The mesh size is set to 480 $\times$ 480, and we split it into 60 $\times$ 60 windows with the overlapping of 20. So, in total, we collect 121 windows for each variable per iteration.

2) 0-propagation dataset: First, we make a copy of correct dataset and then we inject one error per window by randomly flipping one bit in a data point. The error positions in the window are also randomly picked.

3) k-propagation dataset: For FLASH applications, we utilize the checkpoint/restart mechanism to generate k-propagation dataset. We inject errors (similarly to the 0-propagation dataset) into many checkpoints. Then we restart from these corrupted checkpoints and save the variables of the following k iterations.

The testing datasets are generated in a similar manner. Note that the testing datasets are not used in the training process.

C. Metrics

The detection sensitivity (recall) is defined as the number of errors detected over the number of total errors. A time step is considered a false positive if the detector reported an error when no error is present. The false positive rate is then defined as the number of false positive iterations over the total number of iterations under evaluation.

D. Impact of different bit flips

As discussed in Section II, higher bit flips introduce large deviations that can lead to wrong results. In contrast, lower bit errors typically have only negligible impact on the final result. To show the impact caused by different bit errors, we perform the experiments on 6 FLASH applications with errors injected in the middle of the computation. According to the position of the flipped bit, we split errors into three sections, errors in 1-20 bits (MSB), errors in 21 to 40 bits, and errors in 41 to 63 bits (LSB). We did not include the sign bit in this experiment because sign bit errors almost always cause FLASH applications to crash, which means it is not a *silent* error anymore. The impact of SDC is defined as following, where $v_{correct}$ is the error free output and $v_{corrupted}$ is the corrupted output.

$$I = \frac{\text{sum}(\text{abs}(v_{correct} - v_{corrupted}))}{\text{sum}(v_{correct})}$$

Each application is run for 1000 iterations. Tables IV shows the average impact of errors on the final result. We also show mean square error (MSE) in brackets. The nan in BrioWu, BlastBS and OrszagTang is because errors in 1 to 20 bits sometimes lead to a totally abnormal output where some data points are nan. However, unlike the sign bit error which can cause crashes, errors of nan are not detected by the application, i.e. they are still silent errors. It is easy to see that for all six applications, errors in bits 0 to 20 result in a huge difference in the final output whereas errors in 21 to 40 and 41 to 63 bits only cause small deviations in the result. Note that the difference between the last two columns is very small - the numbers are identical in the first few digits. Therefore, in this paper, we focus only on flips in the top 21 bits of a 64-bit floating point value, as bit flips in the lower order bits will be benign.

E. Comparison

We compare our method with a state-of-the-art detector, AID [14]. There are two implementations mentioned in AID's paper: FP-adapted and FP-unadapted. The unadapted version has a better recall but also an unacceptably high false positive rate (up to 50%) for some applications. So in this section, we compare only to the adapted implementation of AID. All parameters for it are as specified in the paper. Note that in AID's paper, the authors first compute an impact error bound for each application, then inject errors that exceed this bound.

TABLE IV: The impact of bit flips in different locations on the final result of applications. Errors in the top bits are significantly worse than errors in lower bits.

App	bits 1-20	bits 21-40	bits 41-63
Sedov	16.62% (59187.704)	0.36% (5.13e-4)	0.36% (5.13e-4)
Sod	20.43% (0.0333)	0.05% (7.7391e-8)	0.05% (7.491e-8)
BrioWu	nan	0.04% (2.443e-6)	0.04% (2.443e-6)
BlastBS	nan	0.20% (2.685e-5)	0.20% (2.685e-5)
DMReflection	0.55% (0.1763)	0.10% (4.137e-3)	0.10% (4.137e-3)
OrszagTang	nan	0.20% (4.855e-5)	0.20% (4.855e-5)

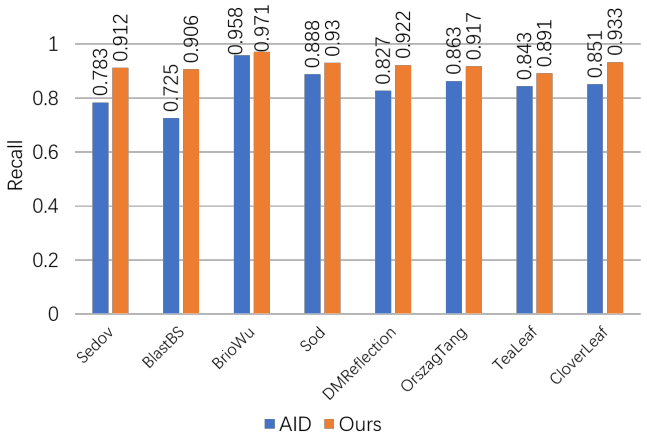


Fig. 4: Recall comparison with AID. Our detector achieves higher recall, outperforming AID.

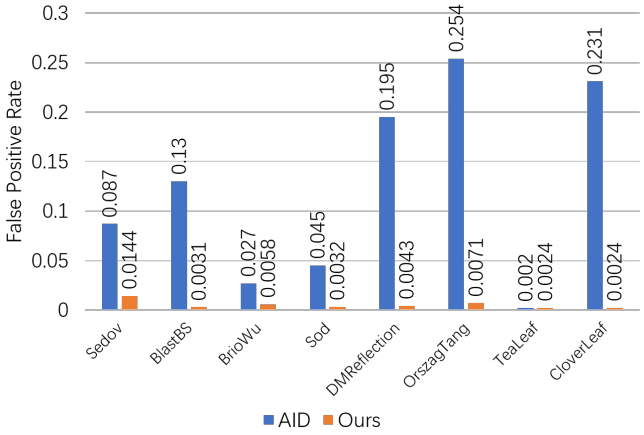


Fig. 5: False positive comparison with AID. Our detector has a significantly lower false positive rate.

First, we run both detectors at every iteration so an error can be identified right after it was injected. Figure 4 and Figure 5 show the detection sensitivity (recall) and false positive rate for all 8 applications. Our detector achieves more than 89% recall in all applications with a false positive rate less than 2%. It is clear that our detector outperforms AID both in recall and false positive rate for all applications.

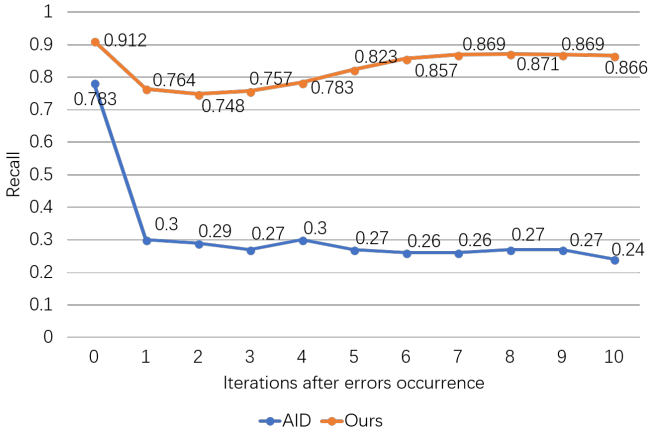
Next, we evaluate the efficiency of detecting errors with a certain delay. As shown in Table IV, errors in higher order bits in Sedov, Sod, BrioWu, BlastBS, and OrszagTang will lead to wrong final results, but for BrioWu, BlastBS and OrszagTang, errors can create nan in the data set, which makes them easy to detect. We use only Sedov and Sod in this evaluation to avoid that case. We run both detectors i iterations $(0 \leq i \leq 10)$ after the error injection. Figure 6 shows the recall for detecting errors with a delay of up to 10 iterations. AID is effective only if the detection process is performed right after the error injection. In contrast, our detector can detect more than 80% of errors even 10 iterations later. An interesting observation is that the recall of our detector drops quickly in the following several iterations after the injection but then it goes up again. One possible explanation is a common error pattern looks like a point surrounded by a ring as shown in Figure 2. At first it is difficult to differentiate the error point from the ring. But as the computation continues, the outer ring starts to expand so the error point can be identified. This behavior is algorithm dependent and is not always the case as we will show in the next experiment.

As we discussed previously, the 0-propagation dataset teaches the neural network to identify a single abnormal data point and it is very effective when running the detector at every iteration. However, by teaching the neural network to recognize error patterns, the k-propagation dataset can help improve accuracy when performing detection many iterations after an error is introduced. We use a 4-point stencil heat diffusion program to show how our detector performs when trained with the k-propagation dataset. We use this custom program because it is easy to understand and also easy to generate the k-propagation data. Figure 7 shows the result of running the detector 0 to 100 iterations after the error injection. We train the neural network with 0-propagation dataset and 5-propagation dataset separately. The accuracy of both detectors decrease as errors propagate, and they achieve similar results for 0 to 30 iterations after injection. However, the detector trained on the 5-propagation dataset remains more accurate for later iterations, and achieves a recall over 80% 100 iterations after an error was injected.

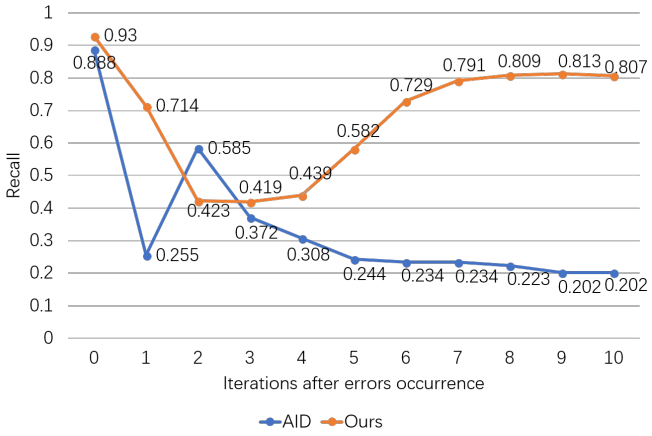
F. Overhead

To compute the overhead of our detector, we first measure the CPU running time of one iteration for each FLASH application. We run each application on a single compute node with 8 MPI ranks. The results are given in Table V. All applications take less than 1 second per iteration. Then we measure the detection time of our detector, which is application independent and only relies on batch sizes, i.e. how many windows we need to examine. In our evaluation, the mesh size of one variable is 480×480, which results in 121 windows (60×60 with 20-pixels overlap) per variable.

Figure 8 shows the detection time on CPU and GPU with different batch sizes. As expected, performing the detection on



(a) Sedov



(b) Sod

Fig. 6: Detecting errors several iterations after they were injected

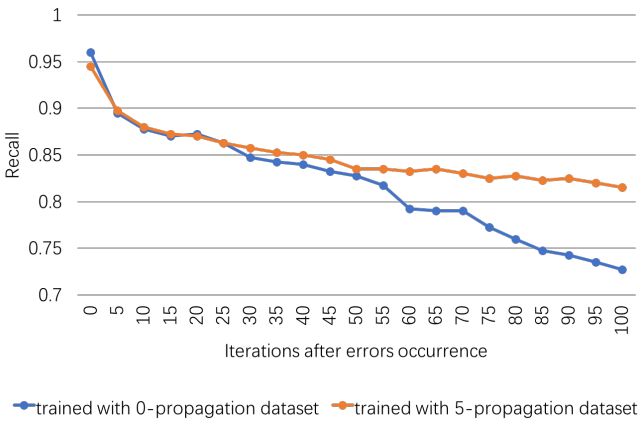


Fig. 7: Detecting errors in heat diffusion program with the neural network trained with 0- and 5-propagation dataset

TABLE V: Runtime for a single iteration of the FLASH applications. Our detector results in low overhead, especially when run on GPU or only every k iterations.

App	Running time for one iteration
Sedov	0.183
Sod	0.276
BrioWu	0.543
BlastBS	0.544
DMReflection	0.176
OrszagTang	0.233

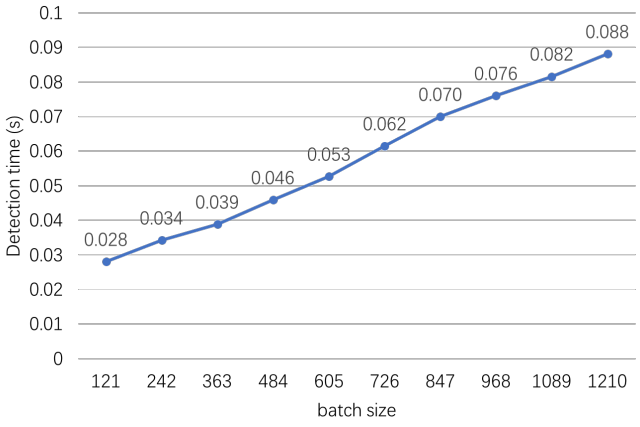
CPU is much more time consuming than on GPU. The average overhead is 0.11× on GPU and 29.51× on CPU. Considering that GPUs are available in most HPC environments, one can use the GPU to perform the detection if it exists on the same compute node that carries out the simulation. The overhead is small when the computation is done on CPUs and the GPUs are used for SDC detection. Further, because our detector is able to detect errors several iterations after errors occurrence, we can run the detector at every few iterations to reduce the overhead even more.

For example, performing the detection every 10 iterations reduces the overhead to 0.011× on GPU and 2.951× on CPU. Moreover, the detection time on both CPU and GPU increases linearly with the batch size. Thus we anticipate that the overhead of our detector will stay the same or decrease as the the mesh size increases, due to the fact that HPC applications may not always exhibit a linear scalability with problem sizes.

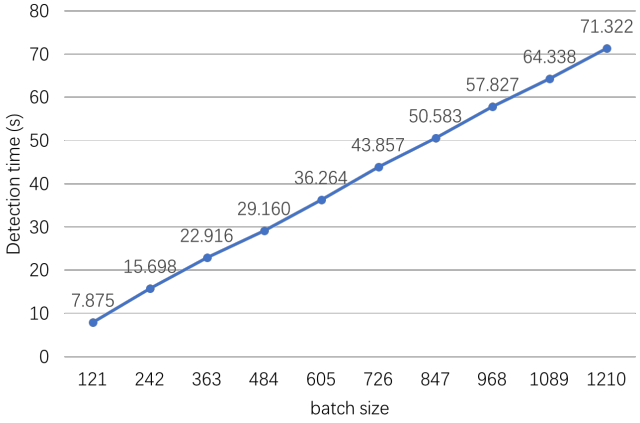
We made no effort to tune the detection code, while the simulation codes are well tuned; more tuning is likely to further decrease the overhead. There are many ways that a production detector could be further optimized to reduce overhead. A generic architecture was used for every application, and we did not engage in significant architecture exploration, which could lead to significantly more compact and efficient networks. Similarly, specialized networks for specific applications could achieve large improvements. Many techniques exist optimize neural networks for production deployment. This includes tuning steps such as optimized memory layouts, algorithm selection, and kernel fusion (e.g. [36]). Neural network quantization is commonly used to optimize inference speed and can enable the use of specialized hardware [37], [38]. Additionally, standard techniques such as model compression and distillation (e.g. [39], [40]) can be applied to the neural network to decrease overhead.

G. Training Time

We protect three variables for each application, leading to three channels in the input data. Note that this is simply due to the nature of the applications, and our method can protect an arbitrary number of variables. As mentioned in V-B, we run each application with 10 different initial conditions and collect the output of 200 iterations (every fifth step out of 1000 iterations) per run. Each iteration contributes 121 60×60 windows per variable, resulting in a total dataset size of 1,452,000 windows (clean and corrupted) per application. Currently this dataset is relatively small, and training with additional data is likely to further improve the performance of our detector.



(a) Detection time on GPU



(b) Detection time on CPU

Fig. 8: Detection time on CPU and GPU

TABLE VI: Training time for each application.

App	Epochs	Training time (hours)
Sedov	22	1.83
Sod	17	1.42
BrioWu	27	2.25
BlastBS	35	2.92
DMReflection	30	2.5
OrszagTang	30	2.5
CloverLeaf	28	2.33
TeaLeaf	30	2.5

The training time for each application is shown in Table VI, where the second column shows how many epochs we train for a specific application. Because applications solve different problems, the number of epochs required to learn to detect SDCs accurately varies. In every case, training time is at most a few hours, which is short compared to a production run of a large-scale simulation. Further, we need only train the detector once per application, amortizing the training time over every application run.

VI. Related Work

Silent errors detection methods have been extensively studied for years. In this Section, we briefly discuss some related work.

Specialized detection techniques [4]–[7] like Algorithm Based Fault Tolerance (ABFT) are designed for specific numerical algorithms. They exploit certain properties of a targeted class of applications. These methods are usually based on the fundamental analysis of linear algebra/matrix operations [14] (e.g. sparse linear algebra [5]). While efficient, they are specific to particular applications thus can not be used in arbitrary HPC programs.

Another type of detector uses a temporal based prediction scheme. [12]–[14], [46] propose different prediction models such as linear curve fitting, quadratic curve fitting, and autoregressive-moving-average. These methods first make a prediction for each data point and then compare it with the observed value. If the difference exceeds a certain threshold then it will be considered an error. Among the prediction based methods, AID [14] provides the best overall results. It combines several curve-fitting models and adaptively chooses the best fitting model to make the prediction. It maintains at least four recent data values for each data point that requires protection, which means 400% extra memory usage in terms of memory overhead. Moreover, the impact error bound, works as a threshold, is required to be calculated for each application beforehand.

Subasi et al. [15], [47] have proposed several spatial prediction based methods. Such detectors use spatial features (i.e. neighboring data values for each data point) to train the model and thus introduce only a small memory overhead. However, one major limitation of their work is that they assume multiple bits are flipped at one time, which makes the error much easier to detect. Further, the accuracy of such detectors is typically worse than temporal based methods.

Detecting silent errors can be thought of as anomaly detection. Deep neural networks have been used for anomaly detection in other fields such as network traffic inspection [48], particle physics [49], EEG waveforms [50], and videos [51]. In addition to applying deep networks to detect silent errors, our architecture is a simple CNN and we phrase the problem as supervised binary classification. Other work on anomaly detection with DNNs has typically used unsupervised autoencoders [52], [53] or LSTM-style networks for classification [54], [55].

VII. Conclusion and Future Work

The basic hypothesis underlying our work is that, with very high probability, an error that is large enough to corrupt the final result is also large enough to be detected long after it occurred. We presented in this paper evidence that this hypothesis holds true for many iterative algorithms and developed a neural network-based silent error detector to detect such errors. The results are preliminary; in particular, the decision to consider only errors in the top 21 bits is somewhat ad-hoc. A more accurate (but also more resource intensive) experiment would inject errors at arbitrary locations and run the computation after error injection to completion, in order to precisely identify benign and malign errors. More

TABLE VII: Applications from the FLASH package

Domain	App	Package	Description
HD	Sod [41]	FLASH [34]	Sodshock tube for testing compressible codes ability with shocks & contact discontinuities
	Sedov [42]	FLASH	Hydrodynamical test code involving strong shocks and non-planar symmetry
	DMReflection [43]	FLASH	Double Mach reflection: an evolution of an unsteady planar shock on an oblique surface
	CloverLeaf	Mantevo [35]	A hydrodynamics mini-app to solve the compressible Euler equations in 2D
MHD	BrioWu [44]	FLASH	Coplanar magneto-hydrodynamic counterpart of hydrodynamic Sod problem
	BlastBS [45]	FLASH	3D version of the MHD spherical blast wave problem
	OrszagTang [21]	FLASH	Simple 2D problem that has become a classic test for MHD codes
DIFF	TeaLeaf	Mantevo	A mini-app that solves the linear heat conduction equation on a using a 5 point stencil

work is needed to provide improved statistical estimates of the frequency of false positives and false negatives, with a feasible amount of experimentation. We have focused on iterative computations where values at a point are updated using values at neighboring points. We plan to consider other iterative applications, e.g. particle codes; and detection rates after more iterations. We expect that further performance improvements are feasible with the use of more carefully selected neural network architectures.

ACKNOWLEDGMENT

This research was supported by NSF SHF award number:1617488 and by the U.S. Department of Energy, DOE Office of Science under contract number DE-AC02-06CH11357. It used compute resources at ALCF and NCSA.

We thank Dr. Anshu Dubey and Dr. Sheng Di for their gracious help.

REFERENCES

[1] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson *et al.*, "Addressing failures in exascale computing," *The International Journal of High Performance Computing Applications*, vol. 28, no. 2, pp. 129–173, 2014.

[2] A. Geist, "Supercomputing's monster in the closet," *IEEE Spectrum*, vol. 53, no. 3, pp. 30–35, March 2016.

[3] D. Tiwari, S. Gupta, J. Rogers, D. Maxwell, P. Rech, S. Vazhkudai, D. Oliveira, D. Londo, N. DeBardeleben, P. Navaux, L. Carro, and A. Bland, "Understanding gpu errors on large-scale hpc systems and the implications for system design and operation," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2015, pp. 331–342.

[4] E. Ciocca, I. Koren, Z. Koren, C. M. Krishna, and D. S. Katz, "Application-level fault tolerance in the orbital thermal imaging spectrometer," in *Dependable Computing, 2004. Proceedings. 10th IEEE Pacific Rim International Symposium on*. IEEE, 2004, pp. 43–48.

[5] J. Sloan, R. Kumar, and G. Bronevetsky, "Algorithmic approaches to low overhead fault detection for sparse linear algebra," in *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*. IEEE, 2012, pp. 1–12.

[6] D. Tao, S. L. Song, S. Krishnamoorthy, P. Wu, X. Liang, E. Z. Zhang, D. Kerbyson, and Z. Chen, "New-sum: A novel online abft scheme for general iterative methods," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2016, pp. 43–55.

[7] M. Turmon, R. Granat, D. S. Katz, and J. Z. Lou, "Tests and tolerances for high-performance software-implemehted fault detection," *IEEE Transactions on Computers*, vol. 52, no. 5, pp. 579–591, 2003.

[8] J. Calhoun, L. Olson, M. Snir, and W. D. Gropp, "Towards a more fault resilient multigrid solver," in *Proceedings of the Symposium on High Performance Computing*. Society for Computer Simulation International, 2015, pp. 1–8.

[9] J. Calhoun, F. Cappello, L. N. Olson, M. Snir, and W. D. Gropp, "Exploring the feasibility of lossy compression for pde simulations," *The International Journal of High Performance Computing Applications*, p. 1094342018762036, 2018.

[10] R. Strzodka and D. Göddeke, "Mixed precision methods for convergent iterative schemes," *EDGE*, vol. 6, pp. 23–24, 2006.

[11] H. Anzt, V. Heuveline, and B. Rocker, "Mixed precision iterative refinement methods for linear systems: Convergence analysis based on krylov subspace methods," in *International Workshop on Applied Parallel Computing*. Springer, 2010, pp. 237–247.

[12] E. Berrocal, L. Bautista-Gomez, S. Di, Z. Lan, and F. Cappello, "Lightweight silent data corruption detection based on runtime data analysis for hpc applications," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2015, pp. 275–278.

[13] S. Di, E. Berrocal, and F. Cappello, "An efficient silent data corruption detection method with error-feedback control and even sampling for hpc applications," in *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*. IEEE, 2015, pp. 271–280.

[14] S. Di and F. Cappello, "Adaptive impact-driven detection of silent data corruption for hpc applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 10, pp. 2809–2823, 2016.

[15] O. Subasi, S. Di, L. Bautista-Gomez, P. Balaprakash, O. Unsal, J. Labarta, A. Cristal, and F. Cappello, "Spatial support vector regression to detect silent errors in the exascale era," in *Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on*. IEEE, 2016, pp. 413–424.

[16] T. C. May and M. H. Woods, "Alpha-particle-induced soft errors in dynamic memories," *IEEE Transactions on Electron Devices*, vol. 26, no. 1, pp. 2–9, 1979.

[17] R. C. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Transactions on Device and materials reliability*, vol. 5, no. 3, pp. 305–316, 2005.

[18] J. Calhoun, M. Snir, L. N. Olson, and W. D. Gropp, "Towards a more complete understanding of sdc propagation," in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2017, pp. 131–142.

[19] J. W. Young, "A first order approximation to the optimum checkpoint interval," *Commun. ACM*, vol. 17, no. 9, pp. 530–531, Sep. 1974. [Online]. Available: http://doi.acm.org/10.1145/361147.361115

[20] D. Zuras, M. Cowlishaw, A. Aiken, M. Applegate, D. Bailey, S. Bass, D. Bhandarkar, M. Bhat, D. Bindel, S. Boldo *et al.*, "Ieee standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1–70, 2008.

[21] S. A. Orszag and C.-M. Tang, "Small-scale structure of two-dimensional magnetohydrodynamic turbulence," *Journal of Fluid Mechanics*, vol. 90, no. 1, pp. 129–143, 1979.

[22] Y. LeCun, B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, and L. D. Jackel, "Handwritten digit recognition with a back-propagation network," in *Advances in neural information processing systems*, 1990, pp. 396–404.

[23] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015.

[24] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," *arXiv preprint arXiv:1707.07012*, 2017.

[25] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[26] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning." in *AAAI*, vol. 4, 2017, p. 12.

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

[28] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[29] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[30] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[31] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 3431–3440.

[32] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[33] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[34] B. Fryxell, K. Olson, P. Ricker, F. Timmes, M. Zingale, D. Lamb, P. MacNeice, R. Rosner, J. Truran, and H. Tufo, "Flash: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes," *The Astrophysical Journal Supplement Series*, vol. 131, no. 1, p. 273, 2000.

[35] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving Performance via Mini-applications," Sandia National Laboratories, Tech. Rep. SAND2009-5574, 2009.

[36] P. Team, "The road to 1.0: production ready pytorch," https://pytorch.org/2018/05/02/road-to-1.0.html, 2018.

[37] D. Lin, S. Talathi, and S. Annapureddy, "Fixed point quantization of deep convolutional networks," in *International Conference on Machine Learning*, 2016, pp. 2849–2858.

[38] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *Journal of Machine Learning Research*, vol. 18, no. 187, pp. 1–30, 2018. [Online]. Available: http://jmlr.org/papers/v18/16-456.html

[39] A. Polino, R. Pascanu, and D. Alistarh, "Model compression via distillation and quantization," *arXiv preprint arXiv:1802.05668*, 2018.

[40] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

[41] G. A. Sod, "A survey of several finite difference methods for systems of nonlinear hyperbolic conservation laws," *Journal of computational physics*, vol. 27, no. 1, pp. 1–31, 1978.

[42] L. I. Sedov, *Similarity and dimensional methods in mechanics*. CRC press, 1993.

[43] P. Colella and P. R. Woodward, "The piecewise parabolic method (ppm) for gas-dynamical simulations," *Journal of computational physics*, vol. 54, no. 1, pp. 174–201, 1984.

[44] M. Brio and C. C. Wu, "An upwind differencing scheme for the equations of ideal magnetohydrodynamics," *Journal of computational physics*, vol. 75, no. 2, pp. 400–422, 1988.

[45] A. L. Zachary, A. Malagoli, and P. Colella, "A higher-order godunov method for multidimensional ideal magnetohydrodynamics," *SIAM Journal on Scientific Computing*, vol. 15, no. 2, pp. 263–284, 1994.

[46] S. Di, E. Berrocal, L. Bautista-Gomez, K. Heisey, R. Guptal, and F. Cappello, "Toward effective detection of silent data corruptions for hpc applications," in *Proceedings of the 28th ACM international conference on supercomputing, SC*, vol. 14, 2014.

[47] O. Subasi and S. Krishnamoorthy, "A gaussian process approach for effective soft error detection," in *Cluster Computing (CLUSTER), 2017 IEEE International Conference on*. IEEE, 2017, pp. 608–612.

[48] U. Fiore, F. Palmieri, A. Castiglione, and A. De Santis, "Network anomaly detection with the restricted boltzmann machine," *Neurocomputing*, vol. 122, pp. 13–23, 2013.

[49] J. H. Collins, K. Howe, and B. Nachman, "Cwola hunting: Extending the bump hunt with machine learning," *arXiv preprint arXiv:1805.02664*, 2018.

[50] D. Wulsin, J. Blanco, R. Mani, and B. Litt, "Semi-supervised anomaly detection for eeg waveforms using deep belief nets," in *Machine Learning and Applications (ICMLA), 2010 Ninth International Conference on*. IEEE, 2010, pp. 436–441.

[51] B. R. Kiran, D. M. Thomas, and R. Parakkal, "An overview of deep learning based methods for unsupervised and semi-supervised anomaly detection in videos," *Journal of Imaging*, vol. 4, no. 2, p. 36, 2018.

[52] M. Sölch, J. Bayer, M. Ludersdorfer, and P. van der Smagt, "Variational inference for on-line anomaly detection in high-dimensional time series," *arXiv preprint arXiv:1602.07109*, 2016.

[53] S. Zhai, Y. Cheng, W. Lu, and Z. Zhang, "Deep structured energy based models for anomaly detection," *arXiv preprint arXiv:1605.07717*, 2016.

[54] P. Malhotra, L. Vig, G. Shroff, and P. Agarwal, "Long short term memory networks for anomaly detection in time series," in *Proceedings*. Presses universitaires de Louvain, 2015, p. 89.

[55] P. Malhotra, A. Ramakrishnan, G. Anand, L. Vig, P. Agarwal, and G. Shroff, "Lstm-based encoder-decoder for multi-sensor anomaly detection," *arXiv preprint arXiv:1607.00148*, 2016.