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 90% 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 applications, 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].

The 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 has motivated a significant amount of research on algorithm-based fault tolerance, where algorithm-specific methods are used to detect errors in intermediate data and repair them [2]–[5]. The main disadvantage of these methods is that they are algorithm-specific and require knowledge of the properties of the numerical method used.

Different designs have been proposed for generic error detectors that work for a large family of iterative algorithms [6]–[9]. 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. One drawback is that they need to compute an impact error bound as a threshold for each application, which requires prior and expensive experimentation. More importantly, these detectors need to be run at each iteration, which significantly reduces their usefulness. Furthermore, as illustrated in Figure 1a, existing work expects that errors occur immediately before detection. However, in practice, errors can happen at any time, as in Figure 1b. Applications may smooth errors within an iteration, making their detection more difficult for current approaches.

In this paper, we consider several questions: How do silent errors behave in different applications? Will they propagate and finally become undetectable as the computation continues? Do we really need to run the detector at each iteration? Can we instead run the detector every \( k \) iterations (e.g. before each checkpoint) if the error is still detectable, in order to reduce overhead?

To answer these questions, we first carefully study the behavior of silent errors in different types of HPC applications. We then propose a neural network-based SDC detector that mitigates the aforementioned drawbacks. The major contributions 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 90%+ 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 [10], [11]. 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 [12]. Some corruptions can be ignored as they are attenuated by the algorithm, and successful convergence to a valid output still occurs. In other cases, they can lead to unacceptable errors in the final result. A portion of this type of errors lead to crashes or obvious faulty results. We are interested in detecting the ones that have an impact on the results and do not lead to obviously erroneous results.

The impact of SDC is problem 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).

B. Bit flips in floating point

IEEE Standard 754 [14] 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). The mantissa, also known as the significand, represents the significant digits of the number. 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 easier to detect since the difference introduced by multiple bit-flips is larger than one bit-flip.

<table>
<thead>
<tr>
<th>Field</th>
<th>Single precision</th>
<th>Double precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fraction</td>
<td>[31]</td>
<td>52[51-00]</td>
</tr>
<tr>
<td>Sign</td>
<td>1[63]</td>
<td>52[51-00]</td>
</tr>
</tbody>
</table>

- **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 \( \sqrt{2T_f/T_c} \), where \( T_f \) is the mean time between failures (MTBF) and \( T_c \) is checkpoint time [13]; when this interval is used, the fraction of the total time spent on checkpoint/restart is \( \sqrt{2T_f/T_c} \). 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 creating the detector is application-independent.

### Table I: IEEE 754 floating point layout.

<table>
<thead>
<tr>
<th>Precision</th>
<th>Sign</th>
<th>Exponent</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double</td>
<td>1[63]</td>
<td>11[62-52]</td>
<td>52[51-00]</td>
</tr>
</tbody>
</table>
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 [12]. In this research, we only focus on those errors that lead to silent corruptions that are neither detected by hardware nor by applications.

Detection is normally more effective when errors are contained and affect only part of 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 different types of applications with arbitrary error injections.

We perform experiments on a representative subset of the applications we evaluate later (see Table VI for an overview). Results and movies of more applications can be found on this website.

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, 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 the ones using shock hydro solvers, 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 based detector that learns to recognize patterns of SDC propagation. One immediate benefit is that its no longer necessary to run the detector at every iteration: we can run it every few iterations (e.g., before each checkpoint) to reduce the overhead.

IV. METHOD

We treat the problem of detecting SDCs as a binary classification problem and train a convolutional neural network [15], [16] 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 [8], [9], which uses curve-fitting schemes to detect anomalous data points. The resounding success of modern convolutional neural networks on image classification problems (e.g., [17]–[19]), 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 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. As we will show in Section V-D, these errors may either be smoothed by the application solver, or result in negligible

---

1 http://chenw5.web.engr.illinois.edu/flash.html
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.
changes to the final result. In this paper, we focus on protecting only the first 21 bits out of a 64 bits floating value.

B. Network architecture

Our neural network is a LeNet- and AlexNet-inspired [20], [21] architecture with some modern improvements. We use blocks of convolution, batch normalization [22], and ReLU activations, some of which are followed by max pooling, three fully-connected layers with dropout [23], 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 $60 \times 60$), 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 [24].

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.

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 better learn 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 $k = 0, \ldots$. 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 intuition is that while the $0$-propagation dataset will allow the network to detect an error immediately after it is injected, the $k$-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 for five epochs using the Adam optimizer [25]. The learning rate is $0.0001$ and the mini-batch size is 64 samples. Training is done in the PyTorch framework [26]. 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 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 VI shows the applications we use in our evaluation from FLASH4.4 [27] and Mantevo [28] 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 × 480, and we split it into 60 × 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 flipping 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.

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

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.

<table>
<thead>
<tr>
<th>App</th>
<th>bits 1-20</th>
<th>bits 21-40</th>
<th>bits 41-63</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sedov</td>
<td>16.62% (591877764)</td>
<td>0.36% (5.13e-4)</td>
<td>0.36% (5.13e-4)</td>
</tr>
<tr>
<td>Sod</td>
<td>20.4% (0.0333)</td>
<td>0.05% (7.3791e-8)</td>
<td>0.05% (7.3791e-8)</td>
</tr>
<tr>
<td>BrioWu</td>
<td>nan</td>
<td>0.04% (2.443e-6)</td>
<td>0.04% (2.443e-6)</td>
</tr>
<tr>
<td>BlastBS</td>
<td>nan</td>
<td>0.20% (2.6585e-5)</td>
<td>0.20% (2.6585e-5)</td>
</tr>
<tr>
<td>DMReflection</td>
<td>0.35% (0.1763)</td>
<td>0.10% (4.137e-3)</td>
<td>0.10% (4.137e-3)</td>
</tr>
<tr>
<td>OrszagTang</td>
<td>nan</td>
<td>0.20% (4.855e-5)</td>
<td>0.20% (4.855e-5)</td>
</tr>
</tbody>
</table>

Each application is run for 1000 iterations. Tables IV shows the 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. Therefore, 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 [8]. 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 their 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. In our evaluation, some of which may be smaller than their impact error bound.

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 90% 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...
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.

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

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 $480 \times 480$, which results in 121 windows ($60 \times 60$ with a 20-pixel overlap) per variable.

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

<table>
<thead>
<tr>
<th>App</th>
<th>Running time for one iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sedov</td>
<td>0.183</td>
</tr>
<tr>
<td>Sod</td>
<td>0.276</td>
</tr>
<tr>
<td>BrioWu</td>
<td>0.543</td>
</tr>
<tr>
<td>BlastBS</td>
<td>0.544</td>
</tr>
<tr>
<td>DMReflection</td>
<td>0.176</td>
</tr>
<tr>
<td>Orszag-Tang</td>
<td>0.233</td>
</tr>
</tbody>
</table>

on CPU is much more expensive than on GPU. The average overhead is $0.11 \times$ on GPU and $29.51 \times$ 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 $0.011 \times$ on GPU and $2.951 \times$ 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 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. Additionally, standard techniques such as model compression and distillation (e.g. [29], [30]) can be applied to the neural network to decrease overhead.

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 [2]–[5] 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 [8] (e.g. sparse linear algebra [3]). While efficient, they are specific to particular applications thus cannot be used in arbitrary HPC programs.

Another type of detector uses a temporal based prediction scheme. [6]–[8], [37] 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 [8] 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, work as a threshold, is required to be calculated for each application.
beforehand.

Subasi et al. [9], [38] 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.

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 and developed a neural network based silent error detector to detect such errors. Future work will further test this hypothesis by checking whether errors that are not detected by our detector are smoothed over following iterations and do not corrupt the final result. We also expect that further performance improvements are feasible with the use of more carefully selected neural network architecture.

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


