Understanding a Program's Resiliency Through Error Propagation

Zhimin Li
University of Utah
zhimin@sci.utah.edu

Harshitha Menon
Lawrence Livermore National Laboratory
gopalakrishn1@llnl.gov

Kathryn Mohror
Lawrence Livermore National Laboratory
mohror1@llnl.gov

Peer-Timo Bremer
Lawrence Livermore National Laboratory
bremer5@llnl.gov

Yarden Livant
University of Utah
yarden@sci.utah.edu

Valerio Pascucci
University of Utah
pascucci@sci.utah.edu

Abstract

Aggressive technology scaling trends have worsened the transient fault problem in high-performance computing (HPC) systems. Some faults are benign, but others can lead to silent data corruption (SDC), which represents a serious problem; a fault introducing an error that is not readily detected nto an HPC simulation . Due to the insidious nature of SDCs, researchers have worked to understand their impact on applications. Previous studies have relied on expensive fault injection campaigns with uniform sampling to provide overall SDC rates, but this solution does not provide any feedback on the code regions without samples.

In this research, we develop a method to systematically analyze all fault injection sites in an application with a low number of fault injection experiments. We use fault propagation data from a fault injection experiment to predict the resiliency of other untested fault sites and obtain an approximate fault tolerance threshold value for each site, which represents the largest error that can be introduced at the site without incurring incorrect simulation results. We define the collection of threshold values over all fault sites in the program as a fault tolerance boundary and propose a simple but efficient method to approximate the boundary. In our experiments, we show our method reduces the number of fault injection samples required to understand a program's resiliency by several orders of magnitude when compared with a traditional fault injection study.

PPoPP '21, February 27-March 3, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8294-6/21/02…$15.00
https://doi.org/10.1145/3437801.3441589

CCS Concepts • **Hardware → Failure prediction**; • **Computer systems organization → Reliability**; • **Software and its engineering → Software fault tolerance**.

Keywords Silent Data Corruption, Natural Resilience, Fault Tolerance, Error Propagation

1 Introduction

Scaling trends in architecture design have led to smaller hardware features and more dense hardware components, exacerbating the transient fault problem in current high performance computing (HPC) systems. Transient faults caused by random events, such as device noise or cosmic radiation, can lead to bit flip events during computation, which can corrupt HPC simulation results. The problem is generally recognized as silent data corruption (SDC), and it threatens the reliability of scientific simulations since SDC can introduce undetectable errors into the simulation output.

To improve an application's resiliency and protect it from SDC, techniques such as instruction duplication [24] and triple modular redundancy [21] are often deployed. However, these techniques introduce significant computation overhead and degrade the system throughput. Previous studies have found that a small fraction of static instructions contribute to the majority of SDC events [10, 12]. Therefore, understanding a program's resiliency and finding the vulnerable program instructions are critical for designing an economic and efficient solution to SDC. The traditional method for understanding a program's resiliency is with a fault injection campaign [13]. In a fault injection campaign, a fault is injected during an application run and the result is recorded. For most benchmarks, an exhaustive fault injection campaign, which tests all possible instructions of an application, will require billions or trillions of fault injection runs [12, 14], which is clearly infeasible (e.g., a fast Fourier transform algorithm with a 512x512 matrix needs 48 billion fault injection runs.). To address this problem, researchers conduct a randomized fault injection campaign with a smaller sample size to get the overall SDC rate of an

application [18]; however, this statistical approach does not guarantee full coverage and does not provide information on code regions with no samples.

A previous study [20] modeled and visualized how an error propagates through a program's computation to infer the program's resiliency. This methodology requires a lower number of fault injection experiments and is more scalable compared with a fault injection campaign. However, an error corrupting an instruction will propagate through the program's dependency graph, and extracting an accurate program dependency graph is not trivial [19]. Thus, interpreting the behavior of error propagation in a program to infer a program's resiliency is still in the exploration stage.

In this paper, we present a method to understand a program's resiliency through error propagation without extracting a program's dependency graph. We define a fault tolerance threshold boundary, in which each dynamic instruction has a maximum value Δe such that with Δe or less error in that particular instruction, the program will still generate an acceptable output. We show that such a boundary can be constructed in practice and propose an economical method that uses the propagated error to infer the Δe of each dynamic instruction. The method is built on the intuition that if an injected error is benign and the program produces an acceptable outcome, this program is highly likely to tolerate the propagation of the error to subsequent dynamic instructions individually. The collection of maximum tolerable error over all dynamic instructions of a program represents its threshold boundary. The construction of such a boundary can help application programmers understand the resiliency of their programs to SDC by predicting the vulnerability of dynamic instructions to injected and propagated error. We show that the uncertainty of the approximated boundary can be verified easily and that our method significantly reduces the number of samples needed to understand a program's resiliency. Our main contributions are:

- An inference method to interpret the error propagation in an application and determine the resiliency of program with respect to fault injection (Section 3.3).
- A new concept of the *fault tolerance boundary* to help diagnose a program's resiliency and a verifiable method to quantify the uncertainty of the approximated boundary (Section 3.2, 3.6).
- An adaptive sampling method to efficiently approximate a program's fault tolerance boundary (Section 3.4).
- An evaluation of our approach with common high performance computation kernels. (Sections 4).

Overall, we find that our fault tolerance boundary method has a prediction precision of over 98% and provides predictions similar to those of an exhaustive fault injection campaign. Additionally, our experiments show that our method reduces the number of fault injection samples needed to understand a program's full-resolution resiliency profile, by up to four orders of magnitude compared with an exhaustive fault injection campaign.

2 Background

In this section, we provide background information on fault injection experiments, models of error propagation, and the metrics used to understand and evaluate the impact of SDC.

2.1 Fault Injection Model

The single bit flip event is the default assumption to study a program's resiliency [1, 4, 8, 13, 19, 30] and in the work we deploy the single bit flip model to study the impact of SDC in scientific applications. In this model, a transient fault is simulated as a single bit flip in one of the data elements of a dynamic instruction. The dynamic instruction here is a single injection site where the result is corruptible. In reality, transient faults can cause bit flips to occur in data in any architectural component, including memory, cache, functional units, and registers, and can impact instructions, control flow variables, and pointer variables as well as data elements. However, bit flips that corrupt instructions of control and pointer variables are relatively easy to detect, because they often cause program exceptions or result in other obvious errors. In contrast, an error that corrupts a data element is challenging to detect, because, in this case, the program flow will most likely continue normally, but can produce an incorrect final output, which may or may not be detectable by the user. Because corruption in application data elements represents the biggest challenge with respect to SDC, we focus our efforts on studying the impact of bit flips on data elements.

We classify the outcomes of an error-corrupted program into three categories:

- **Masked**. In this case, a bit flip occurs but the error is mitigated during the execution. At the end, the program generates an acceptable program output. Here, the acceptable output might not be bitwise reproducible from the error-free run, in which not error occur during the computation, and the program generates a correct output, but it is within an acceptable tolerance level defined by the domain user.
- **Silent Data Corruption (SDC)**. For SDC, a bit flip does not cause any obvious symptoms, e.g., abnormal termination, yet the program produces an incorrect final result.
- **Crash**. Here, a bit flip causes an abnormal termination of the program. For example, a variable value could be corrupted such that it causes a NaN exception.

In this work, we use these standard classifications to describe application outcomes. Further, to quantify the error, we use

the $L\infty$ norm between outputs, although any other metric could be used as well.

We measure the vulnerability of a program using a metric called the SDC ratio, defined as the number of experiments that result in an SDC outcome over the total number of fault injection experiments:

$$SDC_{ratio} = \frac{n_{sdc}}{N},$$

where n_{sdc} is the number of SDC outcomes over the fault injection experiments, and N is the total number of experiments. Previous researchers have often reported their analysis results with a single SDC ratio that represents the entire program execution. In this study, we predict the SDC ratio of each individual instruction and compare the result with the ground truth. We use this metric to measure the vulnerability of each dynamic instruction over the entire program and to evaluate the performance of our method.

2.2 Error Propagation Model

Fault tolerance researchers have developed an intuitive method for understanding the effects of SDC on program output by inspecting error propagation at the source code level, where the result of the analysis can be interpreted directly by the application programmer [20]. In this method, the propagation of an error introduced in a fault injection experiment is monitored by tracking the data variables of a program execution during load/store operations. The error at each dynamic instruction is computed by comparing the difference between the experimental run and the golden run.The error Δx_i in the i-th dynamic instruction is defined as the absolute output difference between the fault-injected run and error-free run. $\Delta x_i = |x_i - x_i'|$ and x_i is the error-free run's value, x_i' is the fault-injected run's value. As discussed earlier, a bit flip may alter the flow of a program by corrupting a data variable. In this case, the method tracks the error propagation over dynamic instructions before the computation diverges, since without the same computation sequence, defining an error represents a fundamental challenge. We utilize this error propagation in this work to develop an application's fault tolerance boundary.

3 Methodology

In this section, we introduce the concept of our fault tolerance boundary, describe our methodology for computing the boundary efficiently, and discuss our approach for determining the accuracy of the boundary.

3.1 Comparison to Fault Injection Campaign

Before we delve into details, first we provide an intuition of the benefits of our fault tolerance boundary method with a comparison to a traditional fault injection campaign in Figure 1. In the figure, each circle represents the outcome of a bit flip of a fault injection experiment (the position of each circle is simplified into a regular grid for this example). The x-axis represents the dynamic instruction index, and the y-axis is the error injected at each sample. To study the resiliency of the application, the traditional fault injection method uses a Monte Carlo simulation, which uniformly samples a subset of dynamic instructions and calculates the SDC ratio to indicate the overall resiliency of a program. As Figure 1 (left) shows, the Monte Carlo simulation gives an overall approximation of the SDC ratio without inspecting most of the dynamic instructions' vulnerability information. In contrast, our fault tolerance boundary method samples a small number of dynamic instructions and collects the error propagation information for each sample to approximate a fault tolerance boundary of the program. Our method uses the boundary to predict the impact of a bit flips and create a full-resolution picture of the resiliency of these instructions.

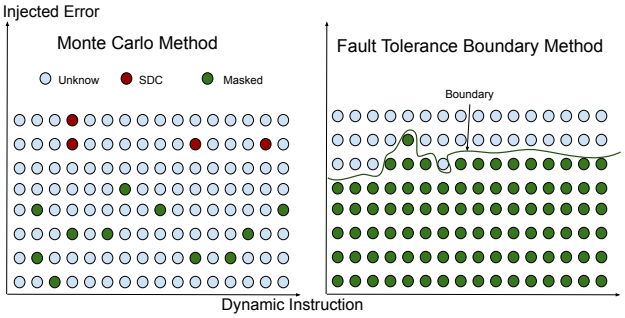

Figure 1. A fault injection campaign (left) randomly samples dynamic instructions to approximate the SDC ratio of a program, but the outcome of many instructions is unknown. The fault tolerance boundary method (right) uses sampled dynamic instructions and their propagation data to produce a fault tolerance boundary that approximates a full picture of the resilience of all dynamic instructions.

3.2 Fault Tolerance Boundary

A program's fault tolerance boundary is a collection of the fault tolerance threshold values of each dynamic instruction. The fault tolerance threshold value of a dynamic instruction is the maximum error value that can be injected such that a lesser or equal amount of error will result in an acceptable program output. The smallest possible threshold value for a dynamic instruction is zero, where the dynamic instruction is extremely sensitive to any level of perturbation, and the largest threshold value is infinity, where the dynamic instruction does not affect the final output calculation. We can model the relationship between the injected error and a program's output error as a function. Assume T is the maximum error a program can tolerate in its output, ϵ is the value of the injected error, and $f_i(\epsilon)$ is a function that describes how much error will be introduced into the final output by

injected error ϵ at fault injection site i. We define the fault tolerance threshold as follows:

Definition. *The fault tolerance threshold at fault injection site i is the threshold value* $\epsilon_{max} \in R^+$ *such that* $\epsilon \leq \epsilon_{max}$ *and* $f_i(\pm\epsilon) \leq T, \forall\epsilon \in R^+$.

Finding the ϵ_{max} value at every fault injection site of a program is challenging. For a single fault injection site, the search space of ϵ is in $[0, \infty)$. Attempting to find such a value in a single location could require a prohibitive amount of testing. However, in reality, the sample space is discrete due to the nature of IEEE standard floating point representations [11], and the number of possible fault injection experiments is limited (e.g., 32 or 64). Therefore, it is possible to approximate the fault tolerance boundary with a large number of fault injection experiments. If a fault injection site is a 64-bit floating point data variable, then the total number of possible experiments at that location is 64. To find the fault injection boundary, one could devise an algorithm to iterate through all 64 experiments to find the minimum bit flip error α that results in $f(\alpha) > T$, and then the threshold value is the maximum value $\epsilon < \alpha$ such that $f(\epsilon) \leq T$.

3.3 Fault Tolerance Threshold Inference

Using an exhaustive fault injection campaign to approximate the fault tolerance boundary as described in the previous section would be computationally expensive. To make our approach economical, we infer the threshold value at each location using error-propagation information from a fault injection experiment, in which the final output is masked.

We illustrate our approach in Figure 2. Here, in experiment A, we inject an error into the dynamic instruction i and the error propagates to the rest of the program, but the final output is acceptable. We compare the error-corrupted run with the error-free run and calculate the perturbation values at each subsequent dynamic instruction. The curve in Figure 2 shows the error at each dynamic instruction of a error-corrupted run. Once the error propagates to dynamic instruction k and causes Δe perturbation from the ground truth, we infer that dynamic instruction k can tolerate less than or equal to Δe error with high probability. To explain the inference, assume we conduct a new experiment B where we inject error into dynamic instruction k with error $\leq \Delta e$. Comparing the two experiments, the error in experiment A propagates to the subsequent dynamic instructions and causes Δe amount of perturbation in dynamic instruction k. In experiment B, there is no error between instruction i and k, and the error in k is less or equal to Δe. The error corruption condition in experiment A is the same or worse than in experiment B. Because the error in experiment A is masked during the computation, we infer that the error in experiment B can also be masked with high probability.

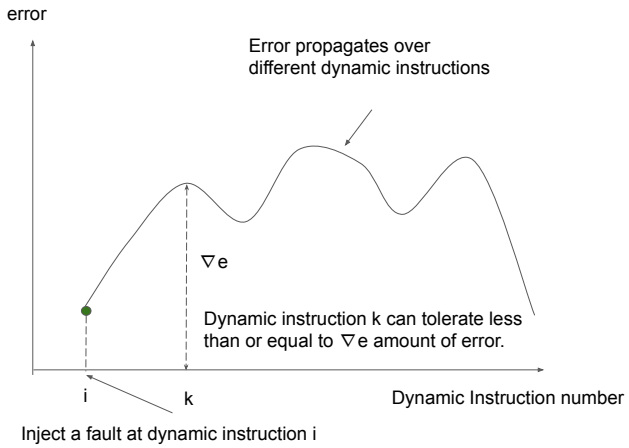

Figure 2. In an experiment, error is injected into dynamic instruction i, which results in a masked final outcome. The error propagates to subsequent dynamic instructions and causes perturbation. We infer that the perturbation values represent the likely amount of error that each dynamic instruction can tolerate in an individual fault injection experiment. In dynamic instruction k, error propagation causes the value to deviate Δe from the ground truth; thus we infer that the dynamic instruction k can tolerate at least Δe amount of injected error and still result in an acceptable program output.

Algorithm 1: Fault tolerance threshold approximation

```
Result: [Δe_1, Δe_2,..., Δe_n] is the fault tolerance
        threshold value over different dynamic
        instructions
S is the sample space and s ⊂ S;
for each s_i ∈ s do
    if s_i is Masked then
        for j ← 0 to n do
            Δe_j = max(Δe_j, s_i[j]);
        end
    end
end
```

Assuming that each masked propagation experiment indicates the minimum error the corrupted subsequent instructions can tolerate, the fault tolerance boundary can be approximated as the aggregation of multiple masked experiments' propagation errors over all the dynamic instructions. Algorithm 1 provides a detailed description of the algorithm. In the algorithm, S represents the complete sample space, and s is the subset of samples chose from S. For the selective masked experiment propagation data, we take the maximum value as the threshold value of a dynamic instruction.

3.4 Adaptive Sampling Method

Our default methodology uses samples selected uniformly at random; however, the experimental evaluation in Section 4.2 shows that dynamic instructions that have more fault injection and propagation information often have better prediction results, and dynamic instructions that have less information may overestimate the SDC ratio. In order to address the shortage of propagation information and to improve the prediction accuracy for instructions with fewer samples, we will bias our sample selection to locations that have less fault injection and propagation information.

We design a bias term that is used to shift the sample density over the dynamic instructions of the program. $p_i = \frac{1}{Z}(\frac{1}{S_i})$, where p_i is the probability for the instruction I and S_i is the amount of information used for approximating the threshold value at instruction i. Z is the normalization constant equal to $\sum(\frac{1}{S_i})$. Meanwhile, instead of randomly sampling a certain amount of samples at once, the sampling process can be constructed progressively. It can select a small amount of samples (e.g., 0.1% or 1000 samples) to approximate a boundary and use the boundary to filter out many masked samples and shrink the potential sample space. The next round of samples will be drawn from the new sample space. The sampling process continues progressively until it does not find any new masked cases or only a small number of masked cases (e.g., 95%, 99% of the new samples are SDC). In the following progressive sampling experiment, we use 0.1% samples in each iteration and use 95% as the stop criteria.

3.5 Improvements to Inference Method

The non-monotonic behavior of the error corrupted program may cause a few errors in the propagation data to degrade the performance of the inference method to approximate the fault tolerance boundary. To improve the fault tolerance boundary approximation, the SDC cases can be used to filter the masked propagation data and build a more accurate boundary. We introduce a *filter operation* based on the condition that if the masked propagation error is greater than the injected error of any known SDC cases, the masked propagation data will not be used to approximate the fault tolerance boundary.

3.6 Validation and Uncertainty

In the machine learning community,precision and recall are used to evaluate the ability of machine learning models to correctly classify data [9]. We evaluate our approximation of the fault tolerance boundary using these metrics, where we note that our approximation process is similar to training a classifier model with the samples resulting in masked outcomes as the training set, and the rest of the sample space as the testing set. We apply these metrics to our fault tolerance boundary method to evaluate its ability to classify the outcome of errors in dynamic instructions.

Precision is defined as the portion of relevant items with respect to the total retrieved items, and recall is defined as the ratio of relevant items to the total relevant items. To utilize these metrics for our fault tolerance boundary, we define the prediction precision and prediction recall based on the number of samples under the fault tolerance boundary, where the boundary predicts whether a case is masked.

$$Precision = \frac{M_{positive}}{M_{predict}}, Recall = \frac{M_{positive}}{M_{total}}$$

In the formula, $M_{positive}$ is the number of predicted positive masked, $M_{predict}$ is the total number of predicted cases, and M_{total} is the total number of masked cases.

The probabilistic nature of our inference method introduces uncertainty to the boundary. Our method uses experiments that result in masked outcomes to approximate the fault tolerance boundary, but information about experiments that result in SDC is not used. The model's prediction over the selected sample data set (both masked and SDC) can indicate the prediction precision over the sample space and reveal the global uncertainty of the boundary. With the uncertainty metric, the application programmer does not need an exhaustive fault injection campaign information to verify the performance of the approximated boundary. We define the uncertainty as:

$$Uncertainty = \frac{Ms_{positive}}{Ms_{predict}},$$

where $Ms_{positive}$ is the number of predicted positive masked in the selected samples, and $Ms_{predict}$ is the total number of predicted cases in the selected samples.

4 Evaluation

Here, we evaluate the efficacy of our fault tolerance boundary for understanding the vulnerability of programs to SDC. First, we explore the ability of the boundary to predict overall SDC using an exhaustive fault injection campaign. Next, we evaluate our boundary inference method from Section 3.3 by inspecting the predicted SDC ratio for every dynamic instruction. Finally, we quantify the uncertainty of our inference method using techniques from machine learning.

We use three common HPC kernels to evaluate our method: LU decomposition [31] is a factorization algorithm that factors a matrix into a low triangle matrix and an upper triangle matrix; conjugate gradient (CG) is a common linear solver for linear equations; and fast Fourier transform (FFT) [31] is a six-step Fourier algorithm in a 1-D domain.

4.1 Exhaustive Campaign Approach

To show that a reasonable fault tolerance boundary exists and is useful, we use an exhaustive fault injection campaign

Name	Benchmark	$Golden_{SDC}$	$Approx_{SDC}$	Size
CG	MiniFE	8.2%	8.92%	47360
LU	splash2	35.89%	36.06%	754176
FFT	splash2	8.33%	8.33%	1064960

Table 1. Comparison of the known true SDC ratio with the approximated SDC ratio from the fault tolerance boundary constructed using an exhaustive fault injection campaign. The approximated SDC from the fault tolerance boundary is very close to the ground truth for all three benchmarks.

to search for the threshold value of each dynamic instruction and build the boundary. The threshold value for a dynamic instruction is chosen such that it is the maximum value that results in a masked outcome, but is also less than the minimum value that results in SDC. To evaluate the performance of the brute-force method, we use the boundary to predict the SDC ratio and compare it with known true values from the exhaustive fault injection campaign where every bit is flipped. The result is presented as a ΔSDC metric in each dynamic instruction and summarized as a histogram over the benchmarks. ΔSDC is equal to $Golden_{SDC} - Approx_{SDC}$, where $Golden_{SDC}$ is the known true SDC ratio, and $Approx_{SDC}$ is the approximate SDC ratio derived from the boundary.

In Figure 3, we present the results for the three benchmarks. The y-axis is the number of fault injection sites, and the x-axis is the value of ΔSDC. We find that the boundary correctly predicts the majority of the dynamic instructions' SDC ratio. In LU and CG, 10.7% and 9.3% of dynamic instructions demonstrate non-monotonic behavior, where a fault injection value e causes SDC, but an error larger than e causes a masked outcome. The SDC ratio of most of these non-monotonic cases is overestimated by 1.5%, with a small number having 3% to 11% overestimation. In Table 1, we show that the aggregate SDC ratio approximated from the fault tolerance boundary is very close to the ground truth for all three benchmarks.

Overall, the fault tolerance threshold boundary constructed from an exhaustive fault injection campaign can provide an accurate vulnerability analysis of the dynamic instructions. The fault injection results of the three benchmarks have a similar pattern in that most dynamic instructions exhibit monotonic behavior with respect to error, and errors larger than an e that causes SDC will almost always cause SDC.

4.2 Inference Approach

To avoid the expense of using an exhaustive fault injection campaign to construct the fault tolerance boundary, we developed an inference approach in Section 3.3 that uses error propagation data of masked experiments to approximate the fault tolerance threshold value of subsequent dynamic instructions. To evaluate the performance of the inference approach, we use Monte Carlo sampling to randomly select 1% of dynamic instructions to approximate the boundary. We use the boundary to predict the SDC ratio over subsequent dynamic instructions in the execution and compare the predicted result with the known true values. Instead of presenting the final result as an overall SDC ratio, we compare the SDC ratio of each dynamic instruction over the execution. To ease the visualization of millions of data points in Figure 4, we group the data of multiple consecutive dynamic instructions (8 dynamic instructions in CG, 147 in LU, and 208 in FFT) and present the mean SDC ratio of each group of instructions.

In the first row of Figure 4, we show a comparison of the known true SDC ratio for dynamic instructions (blue) against the SDC ratio predicted using our inference method with only 1% of the dynamic instructions (orange). For CG, the first 80 dynamic instructions initialize floating point variables to zero. Therefore, the impact of SDC is minor in those instructions. In a 32-bit float-point variable with a value of zero, a maximum perturbation of 2 occurs when there is a flip in the highest exponent bit. Perturbation in the remaining 31 bits causes only small errors, with a maximum value of $1.08 * 10^{-19}$, and such small perturbations will often be masked during floating point computations. In dynamic instructions 80 to 200, the CG benchmark executes initialization instructions. This section of code is executed only once at the beginning of the program, so errors incurred elsewhere in the execution do not propagate to these instructions. As a result, our predictions for instructions 80 to 200 are not very accurate compared to the known true values.

We observe similar results in the first row of Figure 4 for LU and FFT. In LU, the algorithm uses a 16x16 block size and factorizes a 32x32 matrix, so we see that our predicted SDC ratio for dynamic instructions has four regions where our prediction differs from the known true values. In each region, a new loop is started to process a block of the matrix, and the error from prior dynamic instructions is not propagated from region to region. For FFT, the early dynamic instructions transpose a $n1 \times n2$ matrix into $n2 \times n1$ and perform the first $n2$ point 1D FFT on the resulting $n2 \times n1$ matrix. Most of the data elements in instructions 0 to 9000 are accessed only a few times, so errors introduced in this region do not propagate readily and our prediction for this region is relatively inaccurate.

The second row in Figure 4 shows the potential impact corruption in dynamic instructions has on our predictions due to their ability to propagate errors. The y-axis is a measure of the potential impact — the sum of how often the group of dynamic instructions represented by each dot was injected with significant error (relative error greater than 10^{-8}) and how often corrupted data was propagated to those instructions. In CG, instructions 0 to 200 have much less potential to influence our prediction than the subsequent instructions since they are not very likely to have injected error with a sampling rate of 1% and do not propagate errors injected in other regions of the code. As the computation continues

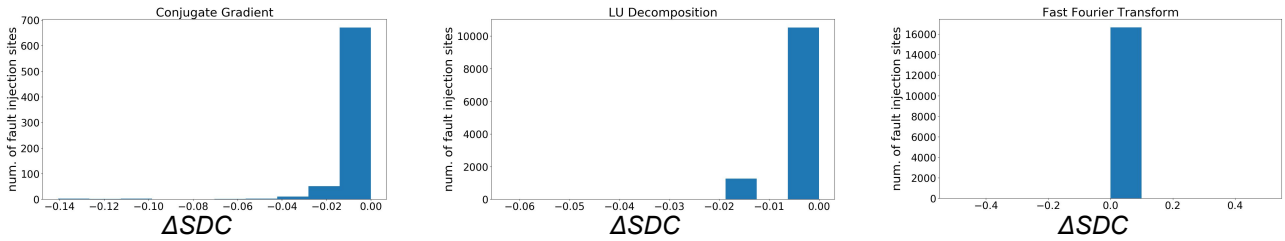

Figure 3. The histograms summarize values of ΔSDC, which is the difference between the known true SDC ratio and the predicted SDC ratio using the fault tolerance boundary constructed from an exhaustive fault injection campaign. Over all dynamic instructions, the FFT benchmark has the same SDC ratio compared with the ground truth. 10.3% of the dynamic instructions in the LU decomposition are overestimated the SDC ratio by 1.5%. 9.3% of the dynamic instructions in conjugate gradient benchmark are overestimated the SDC ratio around 1.5% with a small portion are overestimated around 3% to 14%.

in CG, later instructions have more potential to propagate error and influence our prediction. This observation explains the prediction error pattern of the top row, where the regions with more propagation information often have better predictions, and the regions with less information may overestimate the SDC ratio. We observe similar results for LU and FFT in the second row of Figure 4. In LU, for each of the four regions corresponding to new loops processing blocks of the matrix, the potential to propagate error drops at the start of each region corresponding to an overestimation of our predicted SDC in the top row. Similarly, for FFT, the first 9000 instructions have a very low potential impact on our prediction due to the limited propagation of error from these instructions.

4.3 Performance of Inference Approach

To verify the performance of our method, we quantify the precision, recall, and uncertainty of the predictions (see Section 3.6) of the inference method using 1% sampling of the dynamic instructions in each benchmark.

In Table 2, we present the experiment's prediction precision, recall, and uncertainty and their standard deviations over 10 trails. The prediction precision values are 98.6%, 99.9%, and 100% for CG, LU, and FFT respectively. We also find that the 1% sample approximation boundary gives recall values of 94.31% in CG, 84.58% in LU, and 77.7% in FFT. These results reveal that our method can identify the majority of masked cases without fault injection with an extremely low sampling rate of 1%. Additionally, the standard deviation in each metric over multiple experiment trials is small, which indicates the prediction stability of our method. The precision values are close to the uncertainty values, which suggests that the precision in the training set is close to the precision in the testing set, and that the uncertainty metric can verify whether the approximated boundary will perform well in the complete sample space without the exhaustive fault injection campaign data.

Name	Precision	Recall	Uncertainty
CG	98.64% ± 0.2%	94.31% ± 1.6%	98.4% ± 0.8%
LU	99.9% ± 0.01%	84.58% ± 0.9%	99.9 ± 0.05%
FFT	100%	77.2% ± 0.19%	100%

Table 2. We evaluate the performance of inference method using a 1% sampling rate to approximate the fault tolerance boundary. We see that the precision of our method is very high for all three benchmarks. Additionally, the precision values are similar to the uncertainty values, which suggests that our method can verify the uncertainty of the approximation without the exhaust sampling.

4.4 Performance and Sample Size

The results in Section 4.2 and 4.3 show that the inferred fault tolerance boundary can predict the LU, FFT, and CG benchmark masked cases accurately, and we can quantify the uncertainty of the boundary. However, because we assume the outcome of unknown sample cases as SDC, the overall SDC ratio is overestimated in the three benchmarks when we use a low sampling rate, as shown in the top row of Figure 4. To mitigate this problem, we need to increase the number of samples or design a better way to select samples to approximate the boundary.

One way to improve the performance of the approximated fault tolerance boundary is to increase the number of samples. To study the relationship between the number of samples and the efficiency of the approximated boundary, we use uniform random sampling to select 0.1%, 0.5%, 1%, 5%, 10%, and 50% of the dynamic instructions in each benchmark to approximate the boundary. During the prediction, if all possible error conditions are injected into a dynamic instruction, we simply use the correct boundary value for the instruction instead of prediction. For each benchmark, we perform 10 trails for each sample size and report mean values as the results.

The top row of the Figure 5 shows the precision and recall results for boundary prediction with increasing sample size. In each graph, the x-axis is the percentage of selected samples over the complete sample space. The blue line is the

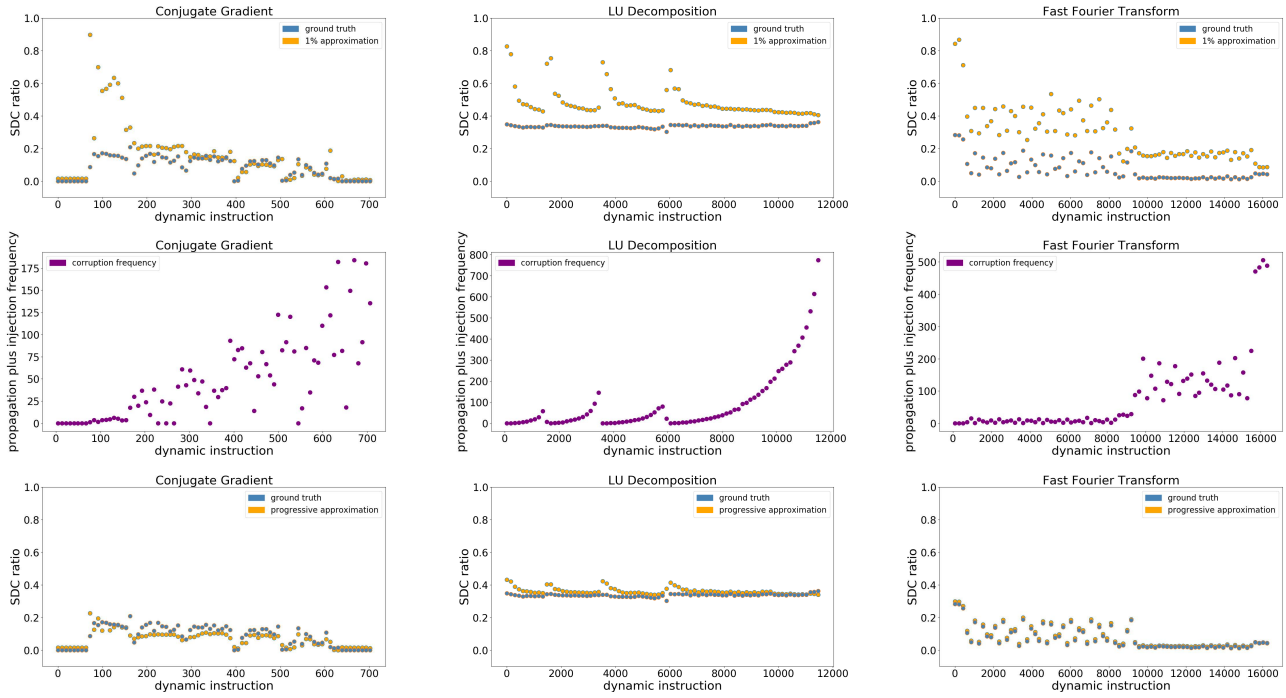

Figure 4. This figure shows the predictive capability of our fault tolerance boundary inference method. The first row shows the predicted SDC ratio for dynamic instructions using a sampling rate of 1%. The second row shows the potential impact of each dynamic instruction on our prediction using a 1% sampling rate. The third row shows our predictions using sampling rates of 1.09% for CG, 4.7% for LU, and 11.2% for FFT. A comparison between the first and second rows shows the reason for our method overestimated SDC ratio for several dynamic instruction regions, where some regions have less potential to propagate error information. We implement a progressive sampling technique to select more samples from those regions with less potential and create a new boundary (row three), which results in highly accurate predictions of SDC.

the prediction recall, and the orange line is the prediction precision. The relationship between the number of selected samples and the prediction recall has a similar pattern in the three benchmarks. At smaller sample sizes, the prediction recall increases exponentially with the number of selected samples, but begins to level out at about 80% to 90%, after which the recall converges slowly to 100% . One outcome of increasing the number of samples is that a larger number of masked samples may cause the prediction precision to drop because of the non-monotonic behavior of the error corrupted program. In the CG, the increasing number of samples to approximate the boundary will cause the prediction precision to drop at the beginning and go back to 100% slowly. In the bottom row of the Figure 5, with the filter operation, which uses SDC cases to filter unqualified propagation data, the prediction precision is always close to 100%, but the prediction recall increases more slower compared with the without filtering experiment in the CG benchmark.

4.5 Performance of Adaptive Sampling Method

In this section, we evaluate the performance of Adaptive sampling method (see section 3.4) at mitigating the shortage of information at certain dynamic instructions and improving their prediction accuracies. The following experiment uses the method to approximate the boundary 10 times and reports the mean value and standard deviation of each benchmark. Each experiment progressively selects a new sample to improve the boundary. For each iteration, we sample 0.1% of the samples. The final result is reported in Table 3. The predicted ratio is close to the golden SDC ratio. The number of samples used in FFT is $10.2\% \pm 0.04\%$ and $4.82\% \pm 0.4\%$ in the LU decomposition. The conjugate gradient uses $1.09\% \pm 0.2\%$ samples. The SDC ratios over different dynamic instructions are reported in the third row of Figure 4. The prediction over the conjugate gradient has a close approximation compared to the ground truth, and the approximation also reflects that the early iteration of the program is more vulnerable to the bit flip error compared to the later computation [20]. In the LU decomposition and the FFT benchmark, the approximated boundary predicts an almost identical SDC ratio compared with the ground truth.

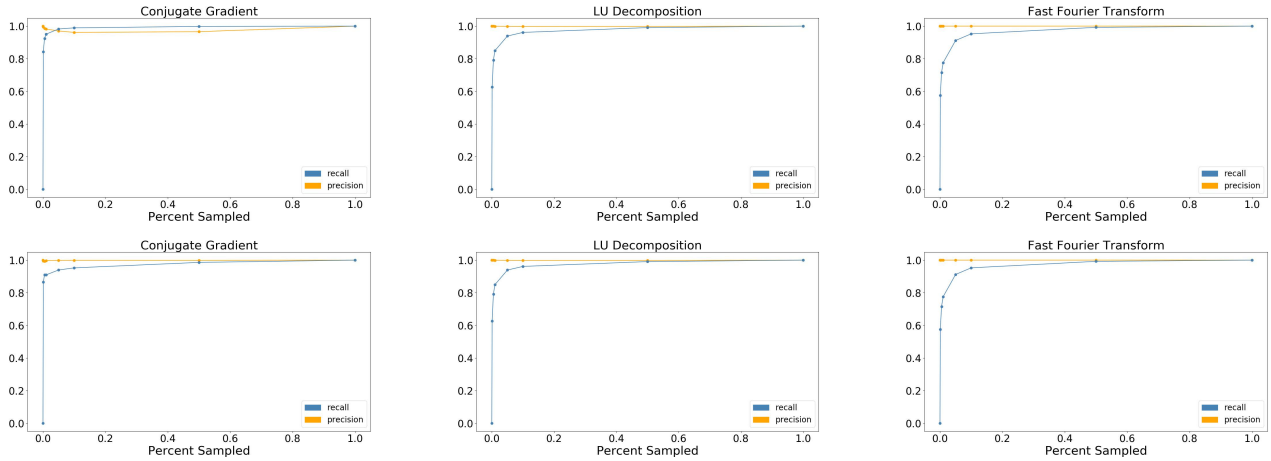

Figure 5. The increasing number of samples also exponentially increases the prediction recall and slows down when the recall reaches around 80% to 90%. Over the three benchmarks, a boundary constructed from a small number of samples can predict the majority of the positive masked samples. Without the filtering operation (top), the prediction precision drops in the conjugate gradient benchmark as the amount of propagation information used to construct the boundary increases. With the filtering operation (bottom), the prediction precision in each benchmark is close to 100%.

Name	SDC Ratio	Sample Size	Predict SDC Ratio
CG	8.2%	1.09% ± 0.2%	5.3% ± 0.7%
LU	35.89%	4.82% ± 0.4%	36.1% ± 0.1%
FFT	7.83%	10.2% ± 0.04%	9.2% ± 0.08%

Table 3. Using the adaptive sampling method to approximate the fault tolerance boundary will reduce the number of samples used to understand a program's resiliency. In LU and FFT, the method ends up with 1 order of magnitude fewer samples to understand a program's resiliency. In the conjugate gradient benchmark, the method ends up with an average 1.09% of total samples, which is 2 orders of magnitude fewer samples to approximate the boundary.

4.6 Scalability in the Iterative Method

The increasing input size can also increase the portion of execution dynamic instructions that are frequently propagated by the error. As an example, a small matrix has the initialization instructions account for $\frac{1}{6}$ of the total samples, but for a large matrix, the instructions accounts for $\frac{1}{22}$. The boundary approximation method has better approximation at the locations that are frequently propagated by the error and the increasing number of execution instructions will reduce the number to understand a program's vulnerability.

To verify the hypothesis, we use the approximate boundary to predict the resiliency of the conjugate gradient with a 20x20 matrix and 100x100 matrix, and compare it with the ground truth. For each input, we randomly select 1000 samples to approximate the boundary and predict the SDC ratio over different dynamic instructions. The experiment will perform 10 trails and the summary of the results are presented in Table 4. The numbers of dynamic instructions in each input are 254784 and 16789952. The predicted SDC ratio is similar to the golden ground truth and the prediction precision around 98.27% and 97.64% with 0.05% and 0.03% standard deviation. The prediction recall is above 96%. Overall, the fault tolerance boundary, which is approximated by 1000 samples (0.4% and 0.006%), can be used to understand the resiliency of a conjugate gradient iterative method with large amount of dynamic instructions well.

5 Discussion and Future Work

The main benefit of our approach is that we can obtain fine-grain resiliency information of an application without needing to perform exhaustive fault injection campaigns. Although our approach is shown to provide good prediction for the applications evaluated, here we discuss the limitations of our current approach and possible future directions to pursue.

Assumption of monotonic nature of error: One potential problem with using the error propagation model to analyze a program's resiliency is that an error does not propagate to certain locations. In the initialization stage of the computation, the errors may not propagate to one another, and the propagation information will not help to understand the vulnerability property of the computation component. A monotonic reaction of an application's output error to the injected fault can help explain the behavior described in section 3.3 and also lead to easily interpretable behavior of the error-corrupted application. The fault injection site i is monotonic if $\epsilon <= \epsilon'$ and $f_i(\epsilon) <= f_i(\epsilon')$ for $\epsilon, \epsilon' \in R^+$. $f_i(\epsilon)$ is the function that describes how much an error in

Input	SDC ratio	predict SDC ratio	precision	uncertainty	recall	num. of samples
20x20	4.5%	6.65% ± 0.9%	98.27% ± 0.05%	98.1% ± 0.5%	96.28% ± 0.01%	254784
100x100	5.0%	6.1% ± 1.2%	97.64% ± 0.03%	97.87% ± 0.5%	96.7% ± 1.2%	16789952

Table 4. To compute the fault tolerance boundary, we use 1000 samples from runs of CG with a 20×20 matrix and a 100×100 matrix, which represents sampling 0.4% and 0.006% of the total samples. The prediction recall is around 96% which indicates that the fault tolerance boundary cover majority of the masked samples and the final precision is around 98%.

injection site i will contribute to the final output. The existence of the fault tolerance boundary does not rely on the monotonic assumption. However, a fault injection site that has a monotonic reaction, guarantees that an application will have an SDC or crash output if the injection error is above the fault tolerance threshold value. Some basic HPC computation kernels such as stencil computation and sparse or dense matrix multiplication can be proven to have such a property.

2D stencil computation computes the value of a center grid by averaging its value and the value of the neighbor grid cells, which can be described as $s(x_{i,j}) = 0.2 \times (x_{i,j} + x_{i+1,j} + x_{i,j+1} + x_{i-1,j} + x_{i,j-1})$. If an error ϵ is injected into one of the elements during the computation, the error term in the initial corrupted location and the four nearby elements is rephrased as $\Delta s(x_{i,j}) = |0.2 \times \epsilon|$. As the computation continues, the error will propagate to the nearby elements. If we use the L_2 norm to measure the error by comparing the ground truth matrix with the error output matrix, the error function is $f(\epsilon) = \sqrt{5 \times (0.2 \times \epsilon)^2} = \sqrt{0.2}\epsilon$. If the computation continues, the error function can be described as $f(\epsilon) = C\epsilon$, and C is a positive constant that depends on the number of computation iterations. The error function is a monotonic function with respect to ϵ. Similarly, one element in the output of a matrix vector multiplication is $\sum_{i,j=1}^{N} x_{ij} v_j$. If an error corrupts a vector element v_k, the output error function is $f(\epsilon) = \sqrt{\sum_{i=0,k} x_{ik}^2 \times \epsilon^2} = C\epsilon$, where C is a constant, which is also the case if an error corrupts a matrix element. For a benchmark without the monotonic reaction to the injected error, our approximation approach provides a trade-off between the number of fault injection experiments and the analysis accuracy, which may sacrifice the accuracy of the analysis but significantly reduce the time to conduct fault injection experiments. Moreover, our approach is self-verified, which tells users whether our approximation result is reliable or needs more samples. In the future, we are interested in formalizing and extending the above analysis process to more complex HPC kernels. On the other hand, we would also like to investigate whether an application that satisfies the method proposed in section 3.3 can indicate that the application output error is monotonic to the corruption error.

Overhead: Our approach does not require exhaustive fault injection runs, but we do need to store the dynamic state of the golden run (run without any faults injected). Therefore, the scalability of our approach is dependent on the size of the golden run against which we compare. Currently, we load the entire state into the memory for calculating the fault tolerance boundary, but that can result in substantial memory overhead for a large-scale application. In future work, we plan to address scalability and memory consumption concerns. One potential solution would be to use the computation duplication to track the error propagation.

6 Related Work

A rich history of research has been dedicated to understanding a program's robustness. The static analysis approach [2, 23, 27] is a low-overhead approach to study a program's robustness. Li et al. designed Trident[19], a three-level model to predict the SDC probability over a program without running any fault injection campaign. Feng et al. [10] proposed a shoestring framework, which leverage the overhead and protection during the compiler time to improve program's resiliency. Static analysis can approximate the overall SDC rate of the application but verifying how accurately it detects fault injection sites is difficult. In comparing with the static method, our approach is self-verifying, which gives users the prediction accuracy of our approximation.

Some works are dedicated to online detection of silent error corruption during program execution. Di et al.[7, 28] assume that adjunct time steps of scientific simulations have auto-correlation and design a time series linear model to predict the values in the current time step based on the value of the previous k time steps' value. They use the predicted value to detect potential corruption with a tolerance value range. Chen [6] designed an approach to detect the occurrence of silent error corruption of the iterative method using the relationship between each iteration step (e.g.conjugate gradient's walk steps are orthonormal to each other). Huang and Jacob [15] designed an online detection matrix multiplication algorithm to detect the silent error corruption during the matrix multiplication. However, these approaches works only for algorithms with certain features. Laguna et al. [17] applied machine learning techniques to a set of selected features and used the trained model to decide whether a program should protect a specific instruction, but the feature selection process is a not trivial.

Replication technique and triple redundancy methods are often deployed in mission-critical applications to prevent SDC. However, the overhead of these approaches is too expensive for most HPC systems. The alternative approach is using partial replication, which selects only the vulnerable

components and protects them. To determine which components are vulnerable, an expensive exhaustive fault injection campaigns are often deployed to study a program's resiliency. Many studies have been done to reduce the computation resource for conducting an exhaustive campaign. Hari et al. [13, 25] proposed Relyzer, an analysis tool that contains a set of fault pruning heuristics to reduce the number of potential fault injection sites. They used a set of carefully selected pilot instructions as representatives of a group of dynamic instructions to reduce the number of tests, based on the assumption that instructions that have similar propagation paths with limited depth share the same vulnerability. A fault injection experiment can be terminated early if a fault injection experiment has similar intermediate states compared with the previous fault injection experiments, and the experiment will be predicted to have a similar computation output as the previous fault injection experiment. Venkatagiri et al. [29, 30] proposed an analysis tool balance the trade-off between the output qualify and computation performance. Kaliorakis et al. [16] designed a method to group different execution intervals based on the program's static feature and found that the execution intervals belonging to a similar group have the same impact on the program's execution. Instead of grouping multiple instructions and picking one dynamic instruction's resiliency to represent all the instructions' resiliency, our approach uses the propagation data to predict the resiliency of all fault injection sites of a program. Each sample is able to cover many more fault injection sites than previous approach. Furthermore, Our analysis approach does not conflict with the previous heuristic approach, and the two approaches can be combined to further reduce the number of samples.

Other researchers also devoted time to designing theory to better model the fault tolerance analysis process. Chaudhuri et al. [5] proposed an auto-verification framework to show an application is robust if it satisfies the Lipschitz continuous property, which bounds the impact of a small perturbation in the program. Menon [22] designed an automatic differentiation method to analyze a program's sensitivity to SDC and to identify variables vulnerable to SDC. Even though iterative linear solvers have a natural resiliency to errors that corrupt computation, Bronevetsky [3] has shown that these solvers are still vulnerable to soft errors. Shantharam et al. [26] found that soft errors may significantly degrade a linear solver's performance and showed that the error in a series of sparse matrix vector multiplication computations grows nonlinear.

7 Conclusion

In this research, we design a method that uses a fault injection experiment's error propagation data to infer the resiliency of a program's instructions without fault injection testing. We propose the fault tolerance threshold boundary, a novel concept, which gives the maximum tolerated error of a program's each dynamic instruction and uses the proposed inference method to approximate the boundary. We test our methodology in common HPC computation kernels and demonstrate that our method accurately predicts a program's resiliency with the prediction precision above 98%. Compared with the exhaust fault injection campaign, our method can reduce the number of samples up to several orders of magnitude.

8 Acknowledgements

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-764021).

References

[1] Rizwan A Ashraf, Roberto Gioiosa, Gokcen Kestor, Ronald F DeMara, Chen-Yong Cher, and Pradip Bose. 2015. Understanding the propagation of transient errors in HPC applications. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, USA, 1–12.

[2] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, and L. Taghaferri. 2003. Data criticality estimation in software applications. In *International Test Conference, 2003. Proceedings. ITC 2003.*, Vol. 1. IEEE, USA, 802–810. https://doi.org/10.1109/TEST.2003.1270912

[3] Greg Bronevetsky and Bronis de Supinski. 2008. Soft Error Vulnerability of Iterative Linear Algebra Methods. In *Proceedings of the 22nd Annual International Conference on Supercomputing (ICS '08)*. Association for Computing Machinery, New York, NY, USA, 155–164. https://doi.org/10.1145/1375527.1375552

[4] Marc Casas, Bronis R. de Supinski, Greg Bronevetsky, and Martin Schulz. 2012. Fault Resilience of the Algebraic Multi-Grid Solver. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS '12)*. Association for Computing Machinery, New York, NY, USA, 91–100. https://doi.org/10.1145/2304576.2304590

[5] Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman. 2012. Continuity and robustness of programs. *Commun. ACM* 55, 8 (2012), 107–115.

[6] Zizhong Chen. 2013. Online-ABFT: An online algorithm based fault tolerance scheme for soft error detection in iterative methods. *ACM SIGPLAN Notices* 48, 8 (2013), 167–176.

[7] Sheng Di, Eduardo Berrocal, and Franck Cappello. 2015. An efficient silent data corruption detection method with error-feedback control and even sampling for HPC applications. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, IEEE, USA, 271–280.

[8] James Elliott, Mark Hoemmen, and Frank Mueller. 2014. Evaluating the impact of SDC on the GMRES iterative solver. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, IEEE, USA, 1193–1202.

[9] Tom Fawcett. 2006. An introduction to ROC analysis. *Pattern recognition letters* 27, 8 (2006), 861–874.

[10] Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. 2010. Shoestring: probabilistic soft error reliability on the cheap. *ACM SIGARCH Computer Architecture News* 38, 1 (2010), 385–396.

[11] David Goldberg. 1991. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)* 23, 1 (1991), 5–48.

[12] Siva Kumar Sastry Hari, Sarita V Adve, and Helia Naeimi. 2012. Low-cost program-level detectors for reducing silent data corruptions. In

IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012). IEEE, IEEE, USA, 1–12.

[13] Siva Kumar Sastry Hari, Sarita V. Adve, Helia Naeimi, and Pradeep Ramachandran. 2012. Relyzer: Exploiting Application-Level Fault Equivalence to Analyze Application Resiliency to Transient Faults. *SIGPLAN Not.* 47, 4 (March 2012), 123–134. https://doi.org/10.1145/2248487.2150990

[14] Martin Hiller, Arshad Jhumka, and Neeraj Suri. 2002. On the placement of software mechanisms for detection of data errors. In *Proceedings International Conference on Dependable Systems and Networks*. IEEE, IEEE, USA, 135–144.

[15] Kuang-Hua Huang and Jacob A Abraham. 1984. Algorithm-based fault tolerance for matrix operations. *IEEE transactions on computers* 100, 6 (1984), 518–528.

[16] Manolis Kaliorakis, Dimitris Gizopoulos, Ramon Canal, and Antonio Gonzalez. 2017. Merlin: Exploiting dynamic instruction behavior for fast and accurate microarchitecture level reliability assessment. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. IEEE, USA, 241–254.

[17] Ignacio Laguna, Martin Schulz, David F. Richards, Jon Calhoun, and Luke Olson. 2016. IPAS: Intelligent Protection against Silent Output Corruption in Scientific Applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO '16)*. Association for Computing Machinery, New York, NY, USA, 227–238. https://doi.org/10.1145/2854038.2854059

[18] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert. 2009. Statistical Fault Injection: Quantified Error and Confidence. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '09)*. European Design and Automation Association, Leuven, BEL, 502–506.

[19] Guanpeng Li, Karthik Pattabiraman, Siva Kumar Sastry Hari, Michael Sullivan, and Timothy Tsai. 2018. Modeling soft-error propagation in programs. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, IEEE, USA, 27–38.

[20] Z. Li, H. Menon, D. Maljovec, Y. Livnat, S. Liu, K. Mohror, P. Bremer, and V. Pascucci. 5555. SpotSDC: Revealing the Silent Data Corruption Propagation in High-performance Computing Systems. *IEEE Transactions on Visualization and Computer Graphics* 0, 01 (may 5555), 1–1. https://doi.org/10.1109/TVCG.2020.2994954

[21] Robert E Lyons and Wouter Vanderkulk. 1962. The use of triple-modular redundancy to improve computer reliability. *IBM journal of research and development* 6, 2 (1962), 200–209.

[22] Harshitha Menon and Kathryn Mohror. 2018. Discvar: Discovering critical variables using algorithmic differentiation for transient faults. *ACM SIGPLAN Notices* 53, 1 (2018), 195–206.

[23] Karthik Pattabiraman, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. 2005. Application-Based Metrics for Strategic Placement of Detectors. In *Proceedings of the 11th Pacific Rim International Symposium on Dependable Computing (PRDC '05)*. IEEE Computer Society, USA, 75–82. https://doi.org/10.1109/PRDC.2005.19

[24] George A Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, David I August, and Shubhendu S Mukherjee. 2005. Software-controlled fault tolerance. *ACM Transactions on Architecture and Code Optimization (TACO)* 2, 4 (2005), 366–396.

[25] Siva Kumar Sastry Hari, Radha Venkatagiri, Sarita V Adve, and Helia Naeimi. 2014. GangES: Gang error simulation for hardware resiliency evaluation. *ACM SIGARCH Computer Architecture News* 42, 3 (2014), 61–72.

[26] Manu Shantharam, Sowmyalatha Srinivasmurthy, and Padma Raghavan. 2011. Characterizing the Impact of Soft Errors on Iterative Methods in Scientific Computing. In *Proceedings of the International Conference on Supercomputing (ICS '11)*. Association for Computing Machinery, New York, NY, USA, 152–161. https://doi.org/10.1145/1995896.1995922

[27] Vilas Sridharan and David R Kaeli. 2009. Eliminating microarchitectural dependency from architectural vulnerability. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. IEEE, USA, 117–128.

[28] Omer Subasi, Sheng Di, Leonardo Bautista-Gomez, Prasanna Balaprakash, Osman Unsal, Jesus Labarta, Adrian Cristal, Sriram Krishnamoorthy, and Franck Cappello. 2018. Exploring the capabilities of support vector machines in detecting silent data corruptions. *Sustainable Computing: Informatics and Systems* 19 (2018), 277–290.

[29] Radha Venkatagiri, Khalique Ahmed, Abdulrahman Mahmoud, Sasa Misailovic, Darko Marinov, Christopher W Fletcher, and Sarita V Adve. 2019. Gem5-approxilyzer: An open-source tool for application-level soft error analysis. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, Portland, OR, USA, 214–221.

[30] Radha Venkatagiri, Abdulrahman Mahmoud, Siva Kumar Sastry Hari, and Sarita V. Adve. 2016. Approxilyzer: Towards a Systematic Framework for Instruction-Level Approximate Computing and Its Application to Hardware Resiliency. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*. IEEE Press, USA, Article 42, 14 pages.

[31] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 programs: Characterization and methodological considerations. *ACM SIGARCH computer architecture news* 23, 2 (1995), 24–36.