Ground-Truth Prediction to Accelerate Soft-Error Impact Analysis for Iterative Methods

Burcu O. Mutlu[†‡], Gokcen Kestor[†], Adrian Cristal[‡], Osman Unsal[*], Sriram Krishnamoorthy[†]
[†]Pacific Northwest National Laboratory, Richland, WA, USA
{burcu.mutlu,gokcen.kestor,sriram}@pnnl.gov
[‡]Polytechnic University of Catalonia, Barcelona, Spain
adrian.cristal@bsc.es
[*]Barcelona Supercomputing Center, Barcelona, Spain
osman.unsal@bsc.es

***Abstract*—Understanding the impact of soft errors on applications can be expensive. Often, it requires an extensive error injection campaign involving numerous runs of the full application in the presence of errors. In this paper, we present a novel approach to arrive at the ground truth–the true impact of an error on the final output–for iterative methods by observing a small number of iterations to learn deviations between normal and error-impacted execution. We develop a machine learning based predictor for three iterative methods to generate ground-truth results without running them to completion for every error injected. We demonstrate that this approach achieves greater accuracy than alternative prediction strategies, including three existing soft error detection strategies. We demonstrate the effectiveness of the ground truth prediction model in evaluating vulnerability and the effectiveness of soft error detection strategies in the context of iterative methods.**

I. Introduction

Soft errors are transient errors caused by environmental conditions, typically manifesting in the form of bit flips. With semiconductor device scaling and the need to limit power consumption to achieve exascale efficiency, soft errors are becoming a significant concern. Specifically, undetected soft errors can lead to application/system crash or even to silent data corruption [1].

Several techniques have been proposed to understand, prevent, and mitigate the effects of soft errors. A typically used technique involves injecting errors into applications runs and observing the outcomes. This approach does not rely on application-specific properties, enabling an understanding of application level error masking and data-dependent computations. However, these error injection campaigns can incur considerable overheads as they require a large number of application runs.

In this paper, we present a novel approach to reduce the cost of such error injection campaigns, without sacrificing their benefits. We present a machine learning based approach to observe a small window of execution past the point at which an error is injected to predict the *ground truth*–impact of the error on the output at the end of the application's execution. While being inherently less precise as compared to actual execution, we demonstrate that the machine learning based predictor is sufficiently accurate to enable meaningful analysis of application vulnerability and detection strategies.

We design our soft error impact strategy in the context of iterative methods used to solve systems of linear equations. These methods can mathematically converge to the correct result from an arbitrary initial guess, lending themselves to significantly application-level error tolerance. However, soft error impact analysis has demonstrated that errors can have a wide variety of outcomes [2]. This has motivated the continued development of novel strategies that observe the execution progress of iterative methods to identify anomalies in their execution [1].

Error detection strategies need to identify the absence of soft errors and the impact of soft errors. The need to correctly identify the more common scenario–the absence of errors or errors that masked at a lower-level–biases existing strategies to focus more on this scenario. We observe that vulnerability analysis, unlike error detection, involves understanding application behavior in the *presence* of errors. We empirically demonstrate that common soft error detection strategies for iterative methods are not equally adept at predicting the ground truth. We develop a prediction strategy that exploits the knowledge of the presence of the error injection (and its magnitude) to produce improved ground truth predictors.

We develop and empirically evaluate our approach in the context of three iterative methods (CG, BiCG, and CGS) and 15 datasets from the University of Florida Sparse Matrix Collection [3]. The primary contributions of this paper are:

- The observation that the error injection information and the execution deviation in a small window of iterations can be used to predict the ground truth for iterative solvers
- A novel machine learning based ground-truth predictor for iterative methods
- Empirical demonstration that the novel ground-truth predictor outperforms alternative strategies, including those based on existing soft error detection strategies
- Analysis of the feature selection and training set requirements for the ground-truth predictor

 DOI 10.1109/HiPC.2019.00048

- An analysis of the cost savings for error injection campaigns in terms of number of iterations of the iterative methods

The rest of the paper is organized as follows. Section 2 describes the iterative methods and datasets, and Section 3 details the proposed methodology along with the machine learning methods used in the study. Section 4 discusses the performance of the method. Section 5 summarizes the related work. Section 6 concludes the paper.

II. Background

A. Iterative Methods

Iterative methods are widely used to solve the linear system of equations:

$$A \cdot \vec{x} = \vec{b}, \tag{1}$$

where A is a sparse matrix, $\vec{x}$ and $\vec{b}$ are vectors. These methods solve the system of equations in an iterative manner. Each iteration computes a guess of the solution $\vec{x}$ that approximates the true solution. An evaluation of the error in the guess is used to determine the next guess. The method returns the computed $\vec{x}$ when the solution error, the residual norm, calculated as $|\vec{r}| = |\vec{b} - A \cdot \vec{x}|$, is within a predefined error threshold. The methods differ in their approach to exploring the solution space. Importantly, the residual norm does not monotonically decrease and the solution vector (and other vectors used in their algorithms) can vary widely between iterations.

We selected three widely used iterative solvers for this study. Below are some details for each method [4]:

- CG: Conjugate gradient method utilizing the LU preconditioner
- BICG: BIConjugate gradient method that does not use a preconditioner
- CGS: Conjugate gradient squared method

Conjugate gradient method can only solve symmetric positive-definite matrices. BICG method is a modified version of CG method to make it handle non-symmetric and non-definite systems. CGS method has better performance compared to BICG with more stability. These methods belong to the class of non-stationary iterative methods. Each considered method differs from other methods in a small yet significant way and testing them would give us insight into how these differences affect error behavior and error prediction. We use the implementations of the methods available as part of the Iterative Methods Library (IML++) v1.2a [5].

B. Datasets

The three iterative methods (discussed in Section II-A) are evaluated using real life matrices taken from the SuiteSparse matrix collection of University of Florida [3]. We select symmetric and positive definite matrices for A so that we can evaluate all iterative solvers considered for this study.

This matrix collection currently has more than 200 symmetric positive definite matrices. After a preliminary analysis, we discard matrices that are not representative—ones that either converge too quickly (in under a second) or do not converge after 35000 number of iterations. After we eliminate those datasets, we randomly select fifteen of the remaining datasets for this study. Table I details the size of the matrices (Rows), non-zero percentage (NZ%) and the number of iterations each solver needed to converge to an acceptable $\vec{x}$.

Matrix	Rows	NZ%	Number of Iterations		
			CG	BICG	CGS
bcsstk13	2003	2.1	928	928	1185
bcsstk14	1806	1.95	195	195	108
bcsstk15	3948	0.8	453	453	207
bcsstk24	3562	1.3	451	451	374
bcsstk28	4410	1.12	4344	4344	7381
bcsstk38	8032	0.55	426	426	1218
ex13	2568	1.15	146	146	104
ex15	6867	0.21	96	96	82
Pres_Poisson	14822	0.33	662	662	669
s1rmq4m1	5489	0.87	612	612	639
s1rmt3m1	5489	0.72	695	695	683
s2rmt3m1	5489	0.72	1787	1787	2085
s3rmq4m1	5489	0.87	2969	2969	2530
s3rmt3m1	5489	0.72	4497	4497	4514
s3rmt3m3	5357	0.72	8538	8538	12097

TABLE I
CHARACTERISTICS OF THE MATRICES SELECTED FROM SUITESPARSE AND THE NUMBER OF ITERATIONS PERFORMED BY SOLVER FOR THAT DATASET

Vector $\vec{b}$ in equation 1 is calculated as $A \cdot \vec{1}$, making the expected value of $\vec{x}$ is a vector of all ones. This practice is similar to one used in other prior studies [6], [2]. We run the iterative methods until residual norm converges to within 10^{-6} error threshold.

C. Detectors vs Ground Truth for Error Injection Experiments

The execution of the iterative methods is data-dependent with a wide variation in the number of iterations taken and changes in the solution space explored. Therefore, barring some work on numerical analysis of the algorithms, error detection strategies for these methods have focused more on observing dynamic program information rather than static analysis. In particular, several soft error detectors have been developed for iterative methods. Each of these detectors attempts to observe specific characteristics of an iterative method's execution to distinguish error-free or error-masking executions from those that can lead to silent data corruption. Given errors are rare, detectors place significant emphasis on reducing false positives, i.e., ensuring that the normal execution is not erroneously flagged as leading to data corruption.

Ground truth predictors to determine the outcome of an error injection can potentially exploit the knowledge of the error injection and a guaranteed non-erroneous execution to achieve improved accuracy.

III. Ground Truth Prediction

In this section, we describe our approach to ground truth prediction. The approach involves comparing the values encountered in variables used in the iterative solvers (specific vectors) between error-injected and error-free runs. Detectors based on redundant execution involve duplicated executions

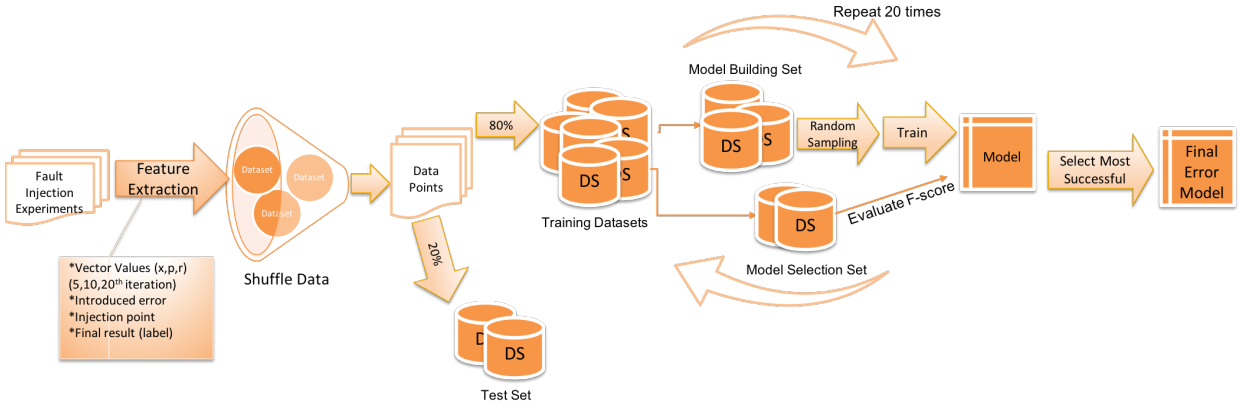


Fig. 1. Our overall approach to construct a ground-truth predictor using machine learning

that can be compared for deviations. These approaches typically identify errors that escape a particular architecture or abstraction level (e.g., errors that escape micro-architecture state or registers) rather than the final application output. We employ a strategy similar, in spirit, to redundant execution where an execution with no error injection is compared with one with error injections to predict outcomes. However, only one error-free execution is used to check numerous error-injected executions. In addition, we focus on predicting ground truth rather than just errors escaping architecture state.

A. Machine-learning based Prediction

Figure 1 illustrates our approach to construct the machine learning based predictor. The training data is constructed using a small number of error injection experiments. For each error injection, we monitor the execution for a small number of iterations. In this work, we limited ourselves to 20 iterations past the error injection to observe the error propagate and manifest itself in the variables monitored. During these 20 iterations after injection, we extract our feature data from the execution. We use $\vec{x}$, $\vec{p}$, and $\vec{r}$ vectors' value at the 5^{th}, 10^{th}, and 20^{th} iterations after the injections. We also observe the magnitude of introduced error and the injection point relative to the execution duration (calculated as injection iteration over the expected number of iterations). So the features we collect from the execution are:

- $\vec{x}$, $\vec{p}$, and $\vec{r}$ vectors' value at the 5^{th} iteration after injection
- $\vec{x}$, $\vec{p}$, and $\vec{r}$ vectors' value at the 10^{th} iteration after injection
- $\vec{x}$, $\vec{p}$, and $\vec{r}$ vectors' value at the 20^{th} iteration after injection
- Iteration percentage, calculated as

$$\frac{\textit{Injected Iteration}}{\textit{Expected \# of Iterations}} \quad (2)$$

- Magnitude of introduced error, calculated as the $\ell 1$ norm between the injected $\vec{v'}$ and the original $\vec{v}$ at the moment of injection

Features: Once the features are extracted, the error-injected run is allowed to proceed to completion unimpeded to determine the outcome. We classify the outcome into two categories. We label each error injection run `MASKED` or `NON-MASKED`:

- `MASKED`: When the solver returns a correct value for $\vec{x}$, and the number of iterations it takes to find a solution is within 5% of the expected number of iterations (i.e., number of iterations it takes when no error was introduced).
- `NON-MASKED`: When the solver either converges to a wrong solution, or takes an unexpected amount of iterations to find the solution.

To learn the deviations from error-free execution that result in a non-masked outcome, we evaluated various machine learning (ML) techniques available in the SciKit Learn package [7]. Specifically, we explored decision tree, support-vector machine, AdaBoost, Random Forest, Naive Bayes, and AdaBoost regression with Decision Trees. Preliminary analysis demonstrated that AdaBoost regression achieved the best results for the methods considered. Therefore, we build our predictor and present results with this ML technique.

For each vector value observed from a fault injected execution, we compare it to the error-free execution to understand how much the injected execution diverged from the expected values. For each iteration, we compare the observed value to a range of iteration values of the correct execution. That is, for the n^{th} iteration, we compare the injected vector value with the vectors at the {n-20 .. n+20} iterations. We compare the injected vector with the corresponding healthy vector's values over the course of the iteration window, as any change introduced by the error can hinder the convergence, but also,

by chance, it can help moving the execution in the right direction [2]. We calculated $\ell 1$ norms between each vector and the correct execution's corresponding vector range. We used the minimum $\ell 1$ value as the difference of the vector at the given iteration.

Figure 1 depicts our overall workflow to construct the ground truth prediction model using machine learning. We train separate models for each solver. For each model, 20% of the datasets are randomly selected to be used as a test set, and the rest of the datasets form the training set. Rather than build a model on the entire training set, we randomly select subsets of datasets to build models. These models are tested on the remaining datasets (ones not used to build the model). We build 20 models for each configuration—the number of datasets used to build the model and the number of samples. Among the many models built, we pick the model that best generalizes to handling datasets not used in building them. This way, we train the model using the most representative subset of datasets and test the final model on previously unseen data.

B. Error Injection Mechanism

There are several injection methods that can be considered for an error injection study, from low circuit level to high software level injections. Each error injection has its shortcomings and strengths. Lower level injection campaigns can provide low overhead injections with precision, but provide less control over temporal aspects of the error. On the other end of the spectrum, higher level injection mechanisms provide the user with more control over the error manifestation being less accurate in emulating the natural occurrence of the error (hardware bit flip.)

To understand and model the soft error behavior of iterative solvers, we lean on the side of increased control over the injection procedure. We follow an application-level error injection methodology that enables us to control the temporal and spatial aspects of the error. We instrument the iterative method implementation so that errors can be injected during the execution to any of the vectors, at any statement of the algorithm, during any of the iterations.

Iterative methods use vectors, two-dimensional matrices, and scalars for their calculations. Matrices in the algorithm are read-only and can be protected from soft errors with ease using established techniques [8], [9]. Scalars in the algorithm are relatively small compared to other data structures in the algorithm. Hence they are less likely to be impacted by an error and less likely to impact the program state even if they are hit. Therefore we focus our injection strategy on the vectors in the algorithm.

As all these iterative solvers solve the same equation, and as they are part of the same class of algorithms, they share some vectors in their algorithms. We selected 3 vectors $(\vec{x}, \vec{p}, \vec{r})$ crucial to all three solvers. Other vectors in the algorithms are either temporary variables or they are calculated using these three vectors. Therefore, we can limit our injections to these vectors and still accomplish a meaningful coverage of the error behavior for the iterative solver.

```
1  struct ErrorConfing {
     int eiteration;
3    ErrorInfo einfo;
   };
5  auto ErrorCampaign(Solver solver, vector<
     ErrorConfing> configs) {
     int it=0;
7    solver.init();
     vector<Outcome> outcomes; //masked or non-
       masked outcomes
9    for(auto ec: configs){
       while(!converged and it++ < ec.eiteration) {
11        solver.iteration_no_error();
       }
13     if (converged) break;
       auto ckpt = solver.checkpoint();
15     solver.iteration_with_error(ec.einfo);
       vector<Feature> features;
17     for(int i=0; i<20 and !terminated; i++) {
          solver.iteration_no_error();
19        features.push_back(solver.get_features());
       }
21     auto pred_ground_truth = classify(features);
       outcomes.push_back(pred_ground_truth);
23     solver.restore(ckpt);
     }
25   return outcomes;
   }
```

Fig. 2. Algorithm for an error-injection campaign based on ground truth prediction. The algorithm is executed for a given iterative solver, data set, and ordered list of error injection configurations.

Our error injection framework instruments the source code to simulate an error. The framework decides on where to inject the error based on the inputs provided by the user (similar to our previous works [1], [2]). These inputs are: iteration number, statement number, vector name, position in the vector, list of bit positions to flip in the vector element. We don't assume any previous knowledge on different vulnerabilities of the iterations and vectors. Therefore, we select the iteration number and vector position in which we inject an error uniformly at random.

We run random injections for each vector-statement pair, for our selected vectors, on the statements that use them. This way, we make sure our injection won't be overwritten and can have a chance to affect the program state. Error behavior including such overwriting can be calculated mathematically from this set of experiments without additional experiments as shown in prior work [2]. We inject uniform random 1-bit, 2-bit, and 4-bit errors. We collected more than 450 data points for each solver-dataset pair, a total of more than 32500 runs.

C. Overall Algorithm: Error Injection with Ground Truth Prediction

Figure 2 shows the overall algorithm for error injection campaigns based on the ground predictor built as described in the preceding sections. The algorithm takes as input a list of error injection points (`configs`) ordered in time. Until an injection point, execution proceeds without any error injection

(denoted by `solver.iteration_no_error()`). Before an error is injected, the solver state is checkpointed (line 14). This is followed by an iteration in which the error is injected (lined 15). The details of the actual error injection—bit flips in vectors at specific statements—are not shown for simplicity. After the error-injected iteration, execution proceeds for 20 more iterations (or termination if it happens sooner) without further errors (line 18). During these iterations, the key features described in the preceding section are captured (line 19). These features are used to predict ground truth using the ML model and the outcome is saved. Once predicted, execution is rolled back to the last saved checkpoint. The execution proceeds error-free until the next iteration in which an error is to be injected.

Note that even though multiple errors are injected into the single execution of the iterative method, the checkpoint-restart enables us to treat each error injection in isolation. In other words, each error injection scenario only analyzes the impact of one single-bit or multi-bit error impacting the execution. Depending on the number of error injection samples and their desired distribution, an initial error-free execution might be performed to compute the number of iterations in the absence of errors. This procedure is repeated for every pair of iterative method and data set of interest in the error injection campaign.

IV. Evaluation

In this section, we explain our experimental setup and evaluate our proposed method. We first evaluate the accuracy of the model in predicting a soft error profile for a subject program. Later we demonstrate the usage of the prediction model by leveraging the prediction to accelerate SDC detector analysis. We also show the method is cost effective compared to exhaustive fault injection studies.

We evaluate the performance of the method using precision, recall, F-score, and masked instance ratio. To recap, $precision$ is the number of MASKED instances correctly labeled, divided by the total number of instances labeled MASKED. $precision$ gives us a measure of the fraction of instances MASKED labels that are indeed MASKED. $recall$ is calculated as the number of MASKED instances correctly labeled, divided by the number of instances that are in fact MASKED. This metric gives us the sensitivity of the prediction in maximally capturing the MASKED instances. F-score combines precision and recall as:

$$\textit{F-score} = 2 \times \frac{precision \cdot recall}{precision + recall} \tag{3}$$

A. Ground Truth Predictor: Model Building and Selection

In building the predictor, we use 80% of the datasets (12) for training and use the remaining (3) for testing. We are interested in building a model that can effectively generalize to the test dataset. To this end, we perform our training in two stages. We observe that some data sets might better capture features from a larger data set than others. Also, to avoid just fitting the model to the training set, we build models using different subsets of the training data. The models are then used to predict the ground truth for the rest of the data sets. We illustrate the approach using an example.

Illustration: Consider the building of a model based on three data sets and 100 error injection experiments among a training data from five data sets. We randomly select 3 of the 12 datasets. Two samples from this selection are $\{1,3,5\}$ and $\{1,5,6\}$. For the first sample, among all error injections involving data sets 1, 3, and 5, 100 are chosen. A model is built using these error injections and is tested using error injections involving datasets 2 and 4 to compute its F-score. This process is repeated for the second sample $\{1,5,6\}$ and other samples. The sample and model that gives the best F-score is selected as the final model for the configuration involving three datasets and 100 error injections.

This procedure is repeated for different numbers of data sets used for model building and the total number of error injections used for training. Table II shows the results of our model building analysis. We build a model using 3, 6, 9, and 11 of the training datasets. Total number of error injections used to build the model are varied between 100, 200, 400, 1000, and all samples involved the dataset. This procedure is repeated for each solver. For all three solvers considered, we observe that the best model constructed using 400 error injection experiments chosen from 3 data sets achieves among the greatest F-scores with a relatively small number of error injection experiments. We use these models for the subsequent evaluation using the test set.

B. Evaluating Solver Vulnerability

We use the ground truth predictor model described above for each solver (best model involving 3 data and 400 points) in terms of its ability to predict application vulnerability. Vulnerability is measured as the average fraction of error injections that result in a non-masked outcome. To be consistent in the rest of the section, we equivalently looked at the masked ratio, which is the fraction of error injections that result in a masked outcome. Note that these are complementary and one can directly derive one from the other[1].

Figure 3 plots the masked ratio computed using ground truth (y-axis) versus the masked ratio computed using various prediction strategies. We present one data point for each combination of solver and data set, for a total of 9 data points for each prediction method. For each predictor, we also present a linear fit trendline to the data points and associated R^2 values. An ideal predictor would result in a fitted trendline from (0,0) to (1,1), indicating an exact match between vulnerabilities from actual and predicted ground truths.

We consider several candidates. The approach based on predicted ground truths are labeled `ML`. We also consider three detectors as potential predictors:

- Adaptive Impact-driven Detection (AID) [10] introduces an "impact error bound" that is used to pinpoint influen-

[1]Computing vulnerability from our injection data involves injecting at all candidate sites. However, this can be directly derived from the injection on "live" sites we consider as shown in prior work [2]

	3	6	9	11
100	0.91	0.89	0.86	0.84
200	0.91	0.90	0.86	0.84
400	**0.95**	0.90	0.87	0.87
1000	0.98	0.89	0.91	0.87
all	0.99	0.93	0.91	0.89

(a) CG

	3	6	9	11
100	0.88	0.84	0.83	0.83
200	0.91	0.85	0.84	0.84
400	**0.95**	0.89	0.86	0.85
1000	0.98	0.92	0.88	0.89
all	1.00	0.91	0.86	0.85

(b) BICG

	3	6	9	11
100	0.86	0.82	0.77	0.77
200	0.88	0.84	0.81	0.78
400	**0.93**	0.86	0.81	0.78
1000	0.98	0.86	0.87	0.80
all	0.98	0.87	0.82	0.83

(c) CGS

TABLE II

DESIGN SPACE EXPLORATION TO TRAIN THE GROUND-TRUTH PREDICTOR FOR (A) CG, (B) BICG, AND (C) CGS. THE ROWS CORRESPOND TO 100, 200, 400, 1000, AND ALL AVAILABLE ERROR INJECTION EXPERIMENTS USED FOR TRAINING. THE COLUMNS CORRESPOND TO 3, 6, 9, AND 11 DATA SETS USED TO BUILD THE MODEL. IN EACH INSTANCE, THE DATA SETS NOT USED TO BUILD THE MODEL ARE USED TO EVALUATE THE MODEL'S EFFECTIVENESS. EACH CELL SHOWS THE BEST F-SCORE ACHIEVED AMONG THE MODELS GENERATED FROM 20 RANDOM SAMPLES.

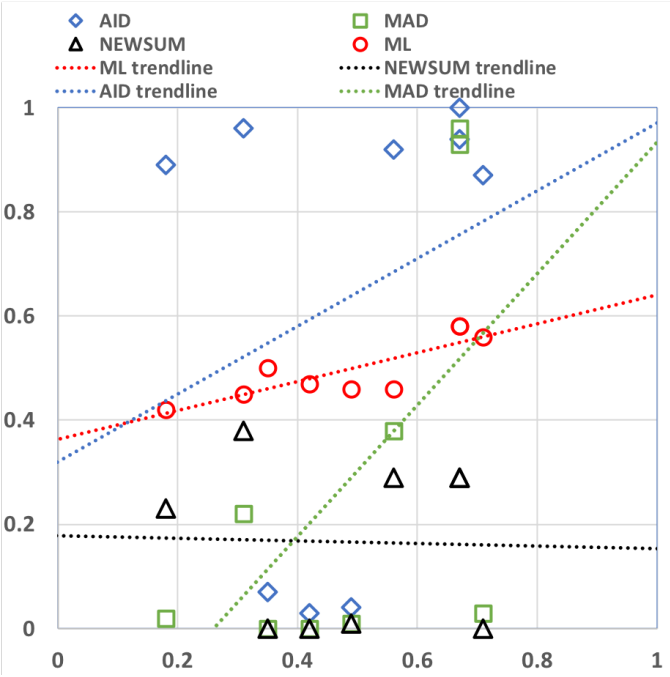


Fig. 3. Predicted MASKED ratio plotted against actual MASKED ratio. x-axis: MASKED ratio predicted from each candidate predictor. y-axis: MASKED ratio computed using ground truth from error injection experiments. ML denotes our approach. Each dot represents a solver-dataset pair. Trendlines for each detection method is also provided, R^2 values for each trendline is AID: 0.0729, MAD:0.3428, NEWSUM: 0.0022, and ML: 0.7073. An R^2 value closer to 1 denotes less error closer match between the trendline and the fitted data.

tial soft errors. They use dynamic curve fitting to detect an influential soft error.

- Checksums for matrix-vector multiplication (NEWSUM) Tao et al. [11] proposes a checksum encoding approach to detect soft errors in matrix-vector and vector-linear operations.
- Moving Average Detector (MAD) observes that residual norms in an iterative method shows a decreasing trend over time. They proposed a moving average schema to detect irregular increases in consecutive time periods, which points to an unexpected behavior, hence detection of a soft error [12].

We used these detectors with their suggested threshold values, AID was run with 0.00078125, NEWSUM with 10^{-10} and MAD with 0.1.

We observe that using detectors as ground truth predictors does not result in a consistent match with the masked ratio computed using the ground truth. All three detectors suffer from both over-estimation and under-estimation of the masked ratio. We observe a strong linear relationship (with a high R^2 fit) between the masked ratio computed using the ground truth and our approach. This shows that the actual masked ratio can be easily determined from the predicted masked ratio using our approach.

A predictor might miss out on matching the actual ground truth on a large number of samples but accidentally predict the overall masked ratio. This might be a challenge from only a subset of the scenarios considered are of interest (say to design a detector for a subset of the error injection points). Table III evaluates this per-prediction accuracy in terms of precision and recall of candidate prediction strategies. This includes three soft error detectors from prior work (AID, MAD, NEWSUM) and our approach (ML). In addition, we consider two random detectors to ensure that the candidate predictors do not succeed by chance. The FAIR COIN detector predicts the outcome to be masked or non-masked with equal probability at every prediction. The BIASED COIN predictor makes the same decisions in ration proportional to the masked ratio in the training data used by our approach.

We observe that our approach achieves the best or near-best precision and recall. Unlike the alternatives, it is always significantly better than the two random predictors. While NEWSUM achieves the best precision and recall for BICG, our approach is not far behind. Also, NEWSUM exhibits significantly lower accuracy for the other two solvers.

C. Evaluation of Detector Accuracy

Another important use of error injection experiments and their outcomes is to evaluate the design and evaluation of soft error detection strategies. Here, we evaluate the effectiveness of our approach to aid in such an evaluation. Table IV shows the precision and recall evaluated for the three detectors described earlier (AID, MAS, and NEWSUM) using actual

ground truths, ground truth predicted by our model, and two random (coin-toss) baseline strategies explained above. We observe that precision and recall determined for the detectors using our approach closely match those computed using the actual ground truth from error injection experiments. The largest deviation between the two is 0.04, clearly demonstrating the usefulness of our approach in evaluating soft error detectors for iterative solves.

In some of the scenarios considered, the random solutions seem to perform quite well as compared to the ground truth. This is just an artifact of the actual ground truth matching the metrics resulting from the random strategies, which are usually around 0.5.

D. Right Answers for the Right Reasons

Detectors often attempt to identify specific portions of an application state or computation space that can be efficiently protected. Depending on the detection strategy being explored, different portions of the error injection space might be of interest. Figure 4 shows the classification of the outcomes into cases where a detector and the predictor agree and where they don't. The decisions made in these cases will only be valid if the predictor matches the actual ground truth. Without such a match, the classification might be correct, but it will not be for the right reasons. We observe that, in this specific case, the predictor is nearly equally effective in identifying, for the right reasons, when the NEWSUM detector performs a correct versus incorrect determination.

Figure 5 shows a scatter plot depicting the fraction of the scenarios in which our approach correctly labels the detector behavior across solver-dataset pairs. x-axis denotes fraction represented by the left green node out of its parent node in the binary tree in Figure 4, across solver-dataset pairs. In this scenario, using our predictor results in the correct decision. Along the y-axis, we depict the fraction represented by the right green node out of its parent node in the binary tree in Figure 4, across solver-dataset pairs. Here the detector is flagged as being in error, and correctly so. With an ideal predictor, all data points plotted will be at (1,1), denoting a perfect match between prediction and actual ground truth for both positive and negative evaluation of the detector. In general, we observe good clustering of the data points around (1,1). We observe a bias in the positive versus negative detector evaluation, with correct detector behavior identified more accurately than incorrect detector behavior.

E. Reduction in Error Injection Campaign Costs

To assess the gain our model will provide against a traditional fault injection campaign for detection performance, we calculated the number of iterations our approach would save the user. In a traditional approach, one will let the execution run to the end, or will stop it when the expected number of iterations/time exceeded deciding there is an anomalous effect of the injection and labels the run accordingly. For our calculations, we assumed the user decides on an anomalous run after 105% of the expected iterations (5% flexibility

Method	CG		BICG		CGS	
	Prec.	Recall	Prec.	Recall	Prec.	Recall
AID	0.50	0.50	0.64	0.54	0.41	0.49
MAD	0.47	0.47	0.68	0.62	0.29	0.50
NEWSUM	0.69	0.66	**0.82**	**0.81**	0.29	0.50
ML	**0.90**	**0.89**	0.81	0.78	**0.80**	**0.80**
FAIR COIN	0.50	0.50	0.50	0.50	0.49	0.49
BIASED COIN	0.51	0.51	0.51	0.51	0.51	0.51

TABLE III
PRECISION AND RECALL OF ESTIMATION OF MASKED RATIO USING VARIOUS CANDIDATE PREDICTORS. ML DENOTES OUR APPROACH. AN IDEAL DETECTOR WILL HAVE PRECISION AND RECALL CLOSE TO 1. THE BEST CANDIDATE FOR EACH SOLVER IS SHOWN IN BOLD.

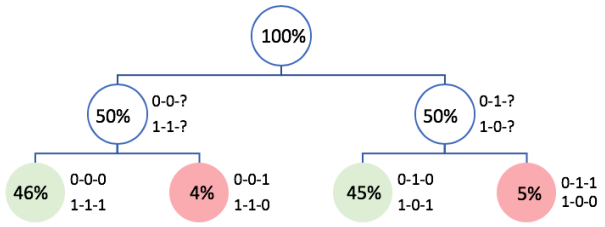


Fig. 4. Classification of scenarios for the CG solver with the NEWSUM detector. The labels are of the form a-b-c, where a is the prediction outcome, b is the detector's judgement, and c is the ground truth. Ideally, the red circles (where we judge a detector based on the wrong prediction) will be 0.

on the number of iterations of an error-less run). On the other hand, our approach can stop the execution whenever a detection flag is raised by a detector (after a minimum of 20 iterations from injection). We injected errors uniform randomly on the iteration space and counted the number of iterations we avoided for the experiments.

For the detection results reported, we saved 240 iterations per CG run, 653 iterations on average per BICG run and 697 iterations on average per CGS run. This corresponds to 21% of the average expected execution for CG, 25% of the average expected execution for BICG and 53% for CGS. The changes in the amount we saved can be explained by the number of masked instances and false positives. As in our method, a detection is awaited to halt the execution, when there is no detection, both the prediction method and the traditional method waits for the end of execution. So when a detector (correctly or not) does not detect any anomalies, our provided gain is on the smaller side. However, when detectors have many detection flags raised, the benefit of our approach magnifies.

F. Overhead Analysis

The cost of the method can be broken down to storing 9 vectors, calculating the $\ell 1$ norms for each vector, and calling the model to get the prediction. This method cuts the cost of the injection study by stopping the execution 20 iterations after the injection, not waiting until the end of the execution.

When solvers are run with our method, with the predictor running total time is around 160% of normal run time. As we leveraged `Python` libraries and `C++` to `Python` connection

		AID		MAD		NEWSUM	
	Against	Precision	Recall	Precision	Recall	Precision	Recall
CG	GT	0.50	0.50	0.47	0.47	0.69	0.66
	Prediction	0.54	0.51	0.49	0.49	0.67	0.65
	Fair Coin	0.56	0.51	0.50	0.50	0.52	0.52
	Biased Coin	0.52	0.50	0.52	0.52	0.52	0.51
BICG	GT	0.41	0.49	0.29	0.50	0.29	0.50
	Prediction	0.39	0.49	0.25	0.50	0.25	0.50
	Fair Coin	0.51	0.50	0.58	0.50	0.41	0.50
	Biased Coin	0.52	0.50	0.44	0.50	0.77	0.50
CGS	GT	0.64	0.54	0.68	0.62	0.82	0.81
	Prediction	0.64	0.54	0.66	0.60	0.85	0.83
	Fair Coin	0.48	0.50	0.50	0.50	0.52	0.52
	Biased Coin	0.48	0.49	0.50	0.50	0.51	0.51

TABLE IV
DETECTOR PRECISION AND RECALL WHEN CALCULATED WITH ACTUAL GROUND TRUTH OF THE EXECUTIONS, AND COMPARED WITH PREDICTED GROUND TRUTHS USING OUR APPROACH.

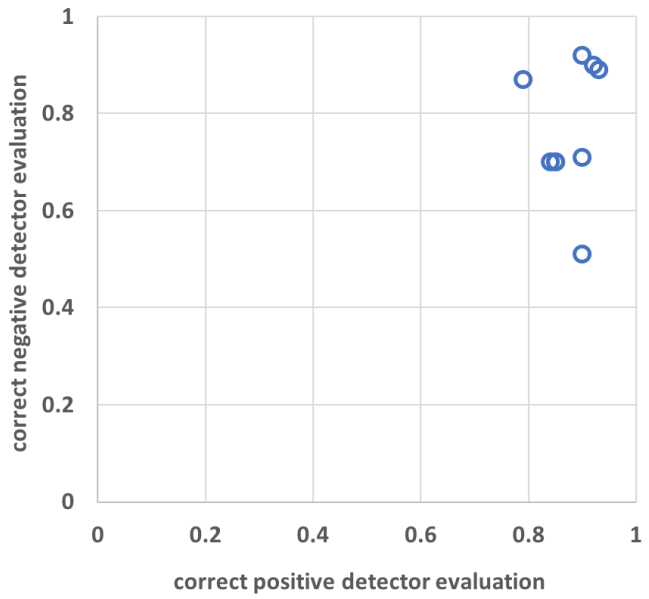


Fig. 5. Predictor accuracy is evaluating positive and negative detector outcomes. x-axis: fraction of all cases where predictor and detector match (marking the detector as being correct), where the ground truth also matches. y-axis: fraction of all cases where predictor and detector differ (flagging the detector as being incorrect), where the detector differs from the ground-truth.

in our tests, the majority of the time is consumed during the `Python` connection, which is known to be slower when handling files. This is not by any means a crucial part of our method. One can easily substitute `Python` with `C++` machine learning approaches to bypass this cost easily.

Even with `Python` costs, there is still an argument to be made for the effectiveness of this method. When an injection is introduced to the solver, the majority of the time, the effect is longer execution times (more iterations). We calculated average total iterations after an error injection from Iterative Method Injection Collection (IMIC) [2], for CG 13.9 times the expected iterations, for BICG 25, and for CGS 16.9 times the expected iterations were performed on average when under the effect of a soft error. These numbers possibly can go higher as in that work, executions are stopped after 35000 iterations, and labeled as 35000.

So, once a model is trained for a solver; for a fault injection study where

- N : Number of fault injections
- I : Number of iterations in normal application run
- I': Number of iterations in fault injected application run
- I_P: Average iteration before injection in proposed method
- I_{mntr}: Number of monitoring iterations after the injection
- P : Overhead cost for the proposed technique

$$\frac{COST_{proposed}}{COST_{traditional}} = \frac{N \times (I_P + I_{mntr}) + P}{N \times I'} \tag{4}$$

We set our monitoring iteration to 20 which corresponds to 1% of the average iteration count:

$$I_{mntr} \approx 0.01 \times I. \tag{5}$$

We set I' as 15 times I–based on the average iteration times from the IMIC database:

$$I' = 15 \times I \tag{6}$$

(they in fact range from $13.9 \times I$ to $25 \times I$ for our set of solvers). With uniform random distribution of injections, we can say on average half of the iterations will be performed before an injection, so proposed iteration cost is:

$$COST_{proposed} = (N \times (I/2 + (0.01 \times I))) \tag{7}$$

Therefore our approach's iteration cost is around 3.5% of the traditional cost of iterations:

$$\frac{N \times (I/2 + (0.01 \times I))}{N \times 15 \times I} \tag{8}$$

Assuming, we collect all the data and call the model/prediction once in the end for efficiency; when the constant model call and prediction costs are added, the overall $COST_{proposed}$ would be around 10% of $COST_{traditional}$; provided P is $1 \times I$ (around 60% for one run and more than 95% of that cost comes from model load).

Method	CG		BICG		CGS	
	Prec.	Recall	Prec.	Recall	Prec.	Recall
CG Model	0.90	0.89	0.74	0.65	0.78	0.73
BICG Model	0.69	0.71	0.81	0.78	0.63	0.62
CGS Model	0.76	0.80	0.75	0.75	0.80	0.80

TABLE V
PRECISION (PREC.) AND RECALL OF ESTIMATION OF MASKED RATIO USING THE MODELS THAT WERE TRAINED USING ANOTHER SOLVER'S DATA. AN IDEAL DETECTOR WILL HAVE PRECISION AND RECALL CLOSE TO 1 FOR ALL SOLVERS.

G. Transferability of the Models

While developing ground truth prediction models trained on the error injections from each solver will give us most accurate predictions, quick evaluation of a novel scenario (e.g., another iterative method) using pre-built models can help identify promising strategies and generate hypothesis before detailing analysis and evaluation. To determine the potential for such a transferability of the models we build in a new context, we determine the effectiveness of the model built using data from one solver in determining the error-impacted behavior of other two solvers. Table V shows the results of this evaluation. We observe that, while predicting outcomes for an iterative method using the model developed for that method performs best, models developed for other solvers are still useful in practice and perform better than alternatives evaluated in Table III.

H. Alternative Training Configurations

In our analysis and evaluation, we considered an injection `MASKED` when the solver returns a correct value for $\vec{x}$, and the number of iterations it takes to find a solution is within 5% of the expected number of iterations. In Figure 6 and Tables VI, VII, and VIII we analyzed how the scenery would change when we change this tolerance amount. We demonstrated the results where no tolerance (0%), 10% tolerance, and 20% tolerance is applied to the injection experiments.

Figure 6 shows slight differences compared to Figure 3, but in all tolerance configurations, we can still observe the machine learning approach showing the strongest linear relationship (R^2 closest to 1) between the masked ratio computed using the ground truth and our approach.

We also evaluated the per-prediction accuracy in terms of precision and recall of candidate prediction strategies when our definition of `MASKED` changed using different tolerance levels for the number of iterations taken. Tables VI, VII, and VIII shows that adjusting the `MASKED` has slightly affected the performance, nevertheless machine learning approach shows best precision and recall. In some cases even better than our selected configuration.

We also demonstrated the effect of splitting the data differently. For our main strategy, we considered a traditional 80%-20% split of the datasets (12 datasets - 3 datasets) into training and testing. Then we used a subset of the 80% (12 dataset) to find a representative subset to train a model. To further analyze the effects of configuration deviations, we also evaluated different train-test splitting methods for our approach. Figure 7 gives F-score distribution box-plots for each configuration considered for each solver. The first box in each figure is our selected approach. The variations we observe tells us there is not one golden train-test split that will work for every solver. Our selected configuration worked best for some settings, whereas it was outperformed for some. We deduce that even though most split methods show good performance, achieving optimal split for best performance requires a comprehensive study of several configurations.

Method	CG		BICG		CGS	
	Prec.	Recall	Prec.	Recall	Prec.	Recall
AID	0.46	0.49	0.54	0.53	0.51	0.50
MAD	0.50	0.51	0.69	0.82	0.44	0.50
NEWSUM	0.70	0.68	0.61	0.80	0.44	0.50
ML	**0.82**	**0.82**	**0.71**	**0.84**	**0.81**	**0.83**
FAIR COIN	0.50	0.50	0.50	0.50	0.49	0.49
BIASED COIN	0.51	0.51	0.51	0.51	0.51	0.51

TABLE VI
PRECISION AND RECALL OF ESTIMATION OF MASKED INSTANCES **(0 % TOLERANCE)** USING VARIOUS CANDIDATE PREDICTORS. ML DENOTES OUR APPROACH. AN IDEAL DETECTOR WILL HAVE PRECISION AND RECALL CLOSE TO 1. THE BEST CANDIDATE FOR EACH SOLVER IS SHOWN IN BOLD.

Method	CG		BICG		CGS	
	Prec.	Recall	Prec.	Recall	Prec.	Recall
AID	0.46	0.49	0.54	0.54	0.51	0.50
MAD	0.51	0.51	0.68	0.83	0.44	0.50
NEWSUM	0.70	0.69	0.61	0.80	0.44	0.50
ML	**0.86**	**0.88**	**0.81**	**0.81**	**0.78**	**0.78**
FAIR COIN	0.50	0.50	0.50	0.50	0.49	0.49
BIASED COIN	0.51	0.51	0.51	0.51	0.51	0.51

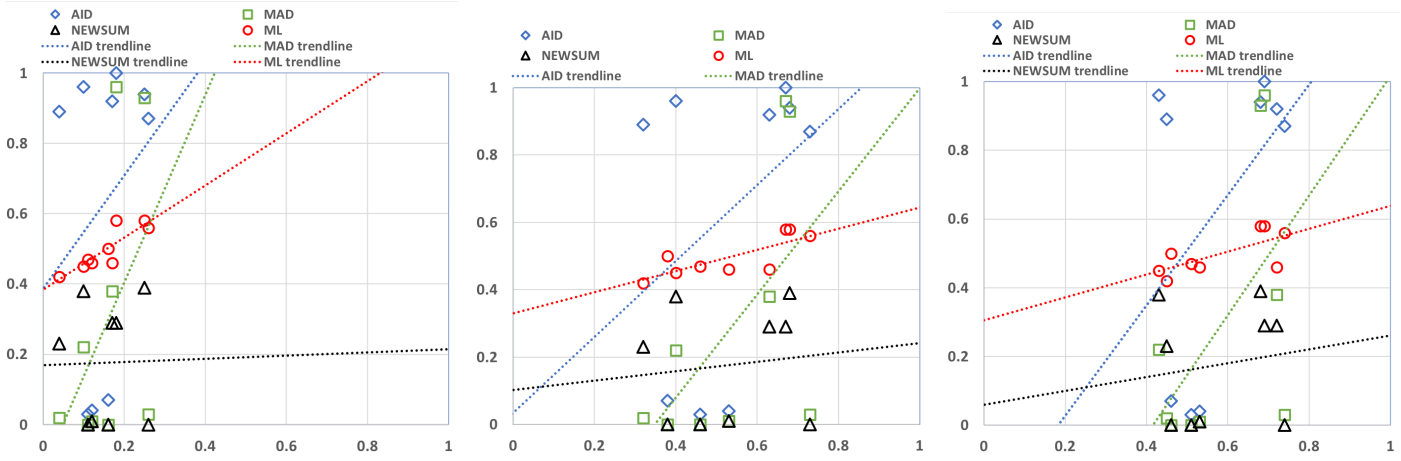
TABLE VII
PRECISION AND RECALL OF ESTIMATION OF MASKED INSTANCES **(10 % TOLERANCE)** USING VARIOUS CANDIDATE PREDICTORS. ML DENOTES OUR APPROACH. AN IDEAL DETECTOR WILL HAVE PRECISION AND RECALL CLOSE TO 1. THE BEST CANDIDATE FOR EACH SOLVER IS SHOWN IN BOLD.

Method	CG		BICG		CGS	
	Prec.	Recall	Prec.	Recall	Prec.	Recall
AID	0.46	0.49	0.54	0.53	0.51	0.50
MAD	0.51	0.51	0.68	0.83	0.44	0.50
NEWSUM	0.70	0.68	0.61	0.80	0.44	0.50
ML	**0.85**	**0.87**	**0.79**	**0.80**	**0.81**	**0.81**
FAIR COIN	0.50	0.50	0.50	0.50	0.49	0.49
BIASED COIN	0.51	0.51	0.51	0.51	0.51	0.51

TABLE VIII
PRECISION AND RECALL OF ESTIMATION OF MASKED INSTANCES **(20 % TOLERANCE)** USING VARIOUS CANDIDATE PREDICTORS. ML DENOTES OUR APPROACH. AN IDEAL DETECTOR WILL HAVE PRECISION AND RECALL CLOSE TO 1. THE BEST CANDIDATE FOR EACH SOLVER IS SHOWN IN BOLD.

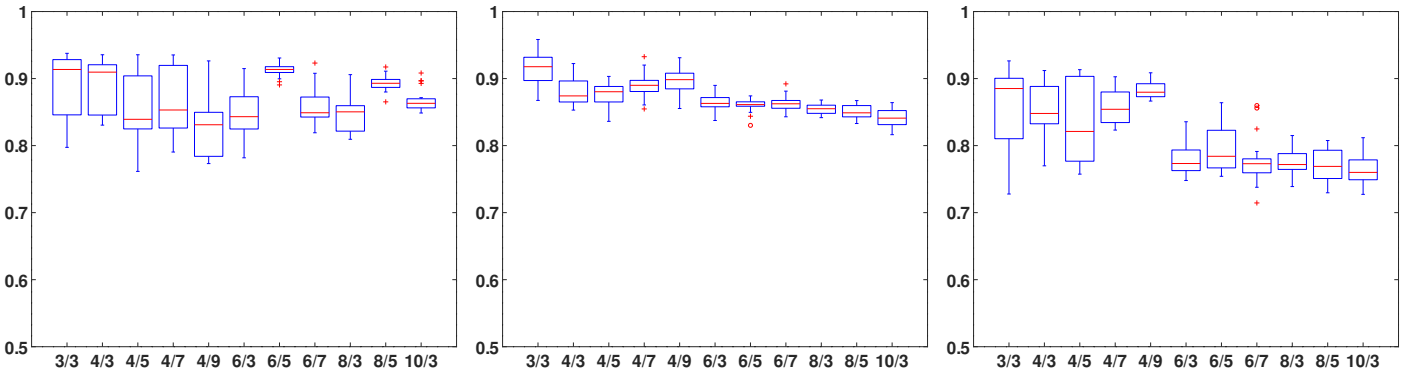
V. RELATED WORK

Many resilience studies based on fault injection campaigns use random fault injection [13], [14], [15], [16], [17]. Random fault injection enables statistical coverage of an ample space with a relatively smaller number of experiments and is employed when the user cannot or does not make assumptions about architecture or application vulnerability. We employ random fault injection on a subset of the application state—the



(a) Tolerance: 0% (b) Tolerance: 10% (c) Tolerance: 20%

Fig. 6. x-axis: MASKED ratio predicted from each candidate predictor. y-axis: MASKED ratio computed using ground truth from error injection experiments (20% flexibility on iteration amount). ML denotes our approach. Each dot represents a solver-dataset pair. Trendlines for each detection method is also provided. R^2 values for each trendline in (a) are AID: 0.0666, MAD:0.2288, NEWSUM: 0.0004, and ML: 0.7579. R^2 values for each trendline in (b) are AID: 0.0147, MAD:0.3371, NEWSUM: 0.1462, and ML: 0.6064. R^2 values for each trendline in (c) are AID: 0.2103, MAD:0.3085, NEWSUM: 0.0220, and ML: 0.4852. An R^2 value closer to 1 denotes less error and closer match between the trendline and the fitted data.



(a) CG (b) BiCG (c) CGS

Fig. 7. F-score performance using different train/test cutoffs for each solver. Label X/Y shows, Y datasets used for testing, from the remaining (15-Y) datasets, random X of them were used for training a model. For each X/Y pair, 20 different random splits were performed and their F-score box plots are shown. On each box, the red line indicates the median, and the bottom and top edges of the box indicate the 25^th and 75^th percentiles respectively. The whiskers extend to the most extreme data points not considered outliers, and the outliers are plotted individually using the red '+' symbol.

vectors—to focus our efforts on the key data structures that are modified in iterative methods.

Fault injection studies can be performed at many levels from lower-level to higher-level injection studies [18], [19]. Lower level injections like Register Transfer Level (RTL) fault injection [17] and architecture level injections [20], [21], [22], [23] tend to be more accurate [15], but the detailed characterization is more challenging.

Software-based error injection can be performed using binary instrumentation [16], compile-time transformations [24], [25], [26], [27], or operating system level injection [28]. There are also studies using emulation and virtualization to provide an injection set-up that requires minimal modification to the system or application [29], [30].

Each technique stresses distinct aspects of an application's footprint (e.g., architectural registers vs. intermediate representation, the specific compiler passes, etc.). We use application-level injection to understand application vulnerability in terms of program elements, analogous to program or data vulnerability factors [31], [32] (as compared to the architecture vulnerability factor [33]). Our injection approach complements the one presented by Xu and Li [34] and can be used in conjunction when an iterative method is used in the context of a larger application.

Recently, there have been many efforts to utilize machine learning [13], [35], [36], [37] to address resilience problems. IPAS [37] uses machine learning to decide on instructions that will likely to lead to corruption and duplicates them. Desh [35] uses systems logs and neural networks to predict node failures.

On [38], the authors present a Machine Learning approach to predict innocuous cases (minimal or no change in convergence behavior) of certain applications in the presence of

silent data corruption. The paper uses NWChem, LULESH, and SVM as their cases. Authors of [39] employed support vector machines to create an online soft error vulnerability prediction mechanism for memory arrays.

Farahani et al. leverages the architecture vulnerability factor to create an online reliability prediction mechanism for transient faults [40]. Several efforts focused on vulnerability factors for modeling error resilience of programs. In [31], authors practice fault modeling on program level. They suggest Program Vulnerability Factor for assessing the vulnerability of a software resource. Yu et al proposes Data Vulnerability Factor [41] which models the vulnerability of individual data structures in an application relying on access patterns. Architecture Vulnerability Factor [42] on the other hand, models the probability of an error happening when a fault happens in that hardware component.

Machine learning techniques have been used in the past to detect soft errors (e.g., [43] and [44]). These approaches attempt to identify executions impacted by soft errors and heading toward anomalous outcomes as compared to benign errors and runs that have not been affected by errors. To improve their accuracy, these methods have to correctly classify not just error-impacted runs but also error-free runs. Because error-free execution is the more common scenario, false positives can overwhelm any benefits from accurate error detection. We bring a novel perspective to soft error impact analysis by observing that this problem begins with an injected error. We observe that machine learning leads to a very accurate classification of benign versus harmful errors when we are given that the computation has been impacted by an error. We believe we are the first to explore machine learning based outcome classification with this prior knowledge. Also, we believe this is the first work to use machine learning techniques to accelerate analysis of the impact of soft errors.

VI. Conclusions and Discussion

In this work, we proposed a method to predict a program's resiliency against soft errors. We evaluated our method on iterative solvers and showed by monitoring only a portion of the execution we can have an acceptable fault profile of the subject program.

We show that not running the execution to completion gives us efficiency in fault injection tests. We demonstrate the use of the method by using it to assess SDC detectors' performances. Our tests reveal that this trained model is successful in assessing detector performance and significantly cuts costs depending on the solver and detector characteristics.

We will open source the data and framework used in this paper to enable reproduction of the results and the design of novel soft error detection strategies. Specifically, we will release the following:

- The framework to train the models from error injection data
- Runtime feature collection framework to predict ground truths
- Database of ground truths predicted by the model and detector estimates.

The database will be released as an extension to the IMIC database [45].

Future work: We demonstrated the effectiveness of machine learning in soft error impact analysis for iterative methods. However, as evidenced in Section IV-H, these models can be further improved. We consider devising the most effective training strategy to improve accuracy as future work. We have observed that the notion of evaluating the impact of soft errors as being distinct from designing a soft error detector. While our work has focused on iterative methods, we believe this observation can benefit other applications. Demonstrating the effectiveness of this approach in the context of other applications constitutes future research.

Acknowledgments

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under Award Number 66905, program manager Lucy Nowell. Pacific Northwest National Laboratory is operated by Battelle for DOE under Contract DE-AC05-76RL01830.

References

[1] G. Kestor, B. O. Mutlu, J. Manzano, O. Subasi, O. Unsal, and S. Krishnamoorthy, "Comparative analysis of soft-error detection strategies: A case study with iterative methods," in *CF '18*, 2018.

[2] B. O. Mutlu, G. Kestor, J. Manzano, O. Unsal, S. Chatterjee, and S. Krishnamoorthy, "Characterization of the impact of soft errors on iterative methods," in *HiPC '18*, 2018, pp. 203–214.

[3] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, 2011.

[4] G. H. Golub and C. F. Van Loan, *Matrix Computations (3rd Ed.)*. Baltimore, MD, USA: Johns Hopkins University Press, 1996.

[5] J. Dongarra, A. Lumsdaine, R. Pozo, and K. Remington, "IML++ v. 1.2 iterative methods library reference guide," Univ. of Tennessee, Tech. Rep. CS-95-303, 1995.

[6] G. Bronevetsky and B. de Supinski, "Soft error vulnerability of iterative linear algebra methods," in *ICS*, 2008, pp. 155–164.

[7] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[8] N. Ali, S. Krishnamoorthy, M. Halappanavar, and J. Daily, "Tolerating correlated failures for generalized cartesian distributions via bipartite matching," in *CF '11*, 2011, pp. 36:1–36:10.

[9] ——, "Multi-fault tolerance for Cartesian data distributions," *International Journal of Parallel Programming*, vol. 41, no. 3, pp. 469–493, 2013.

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

[11] D. Tao, S. L. Song, S. Krishnamoorthy, P. Wu, X. Liang, E. Z. Zhang, D. Kerbyson, and Z. Chen, "New-Sum: A novel online abft scheme for general iterative methods," in *HPDC '16*. New York, NY, USA: ACM, 2016, pp. 43–55.

[12] J. Liu, M. C. Kurt, and G. Agrawal, "A practical approach for handling soft errors in iterative applications," in *Cluster '15*, 2015, pp. 158–161.

[13] R. A. Ashraf, R. Gioiosa, G. Kestor, R. F. DeMara, C.-Y. Cher, and P. Bose, "Understanding the propagation of transient errors in hpc applications," in *SC '15*, 2015, pp. 72:1–72:12.

[14] M. Casas, B. R. de Supinski, G. Bronevetsky, and M. Schulz, "Fault resilience of the algebraic multi-grid solver," in *ICS '12*, 2012, pp. 91–100.

[15] H. Cho, S. Mirkhani, C.-Y. Cher, J. Abraham, and S. Mitra, "Quantitative evaluation of soft error injection techniques for robust system design," in *DAC*, 2013, pp. 1–10.

[16] D. Li, J. S. Vetter, and W. Yu, "Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool," in *SC '12*, 2012, pp. 57:1–57:11.

[17] M. Maniatakos, N. Karimi, C. Tirumurti, A. Jas, and Y. Makris, "Instruction-level impact analysis of low-level faults in a modern microprocessor controller," *IEEE Transactions on Computers*, vol. 60, no. 9, pp. 1260–1273, 2011.

[18] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[19] H. Ziade, R. A. Ayoubi, and R. Velazco, "A survey on fault injection techniques," *Int. Arab J. Inf. Technol.*, vol. 1, pp. 171–186, 2004.

[20] S. Bhm and C. Engelmann, "xSim: The extreme-scale simulator," in *HPCS '11*, 2011, pp. 280–286.

[21] A. Cristal, M. Valero, G. Yalcin, and O. S. Unsal, "Fimsim: A fault injection infrastructure for microarchitectural simulators," in *ICCD '11*, vol. 00, 2011, pp. 431–432.

[22] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: Probabilistic soft error reliability on the cheap," in *ASPLOS '10*, 2010, pp. 385–396.

[23] M.-L. Li, P. Ramachandran, U. Karpuzcu, S. Hari, and S. Adve, "Accurate microarchitecture-level fault modeling for studying hardware faults," in *HPCA '09*, 2009, pp. 105–116.

[24] J. Calhoun, L. Olson, and M. Snir, "FlipIt: An LLVM based fault injector for HPC," in *Euro-Par 2014*, ser. Lecture Notes in Computer Science. Springer International Publishing, 2014, vol. 8805, pp. 547–558.

[25] V. C. Sharma, A. Haran, Z. Rakamarić, and G. Gopalakrishnan, "Towards formal approaches to system resilience," in *PRDC '13*, 2013, pp. 41–50.

[26] D. Skarin, R. Barbosa, and J. Karlsson, "GOOFI-2: A tool for experimental dependability assessment," in *DSN '10*, 2010, pp. 557–562.

[27] J. Wei, A. Thomas, G. Li, and K. Pattabiraman, "Quantifying the accuracy of high-level fault injection techniques for hardware faults," in *DSN '14*, 2014, pp. 375–382.

[28] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "Ferrari: A flexible software-based fault and error injection system," *IEEE Trans. Comput.*, vol. 44, no. 2, pp. 248–260, 1995.

[29] Q. Guan, N. Debardeleben, S. Blanchard, and S. Fu, "F-SEFI: A fine-grained soft error fault injection tool for profiling application vulnerability," in *IPDPS '14*, 2014, pp. 1245–1254.

[30] S. Levy, M. G. Dosanjh, P. G. Bridges, and K. B. Ferreira, "Using unreliable virtual hardware to inject errors in extreme-scale systems," in *FTXS '13*, 2013, pp. 21–26.

[31] V. Sridharan and D. R. Kaeli, "Quantifying software vulnerability," in *Proceedings of the 2008 Workshop on Radiation Effects and Fault Tolerance in Nanometer Technologies*, ser. WREFT '08, 2008, pp. 323–328.

[39] A. Vijayan, A. Koneru, M. Ebrahimit, K. Chakrabarty, and M. B. Tahoori, "Online soft-error vulnerability estimation for memory arrays," in *2016 IEEE 34th VLSI Test Symposium (VTS)*, April 2016, pp. 1–6.

[32] L. Yu, D. Li, S. Mittal, and J. S. Vetter, "Quantitatively modeling application resilience with the data vulnerability factor," in *SC '14*, 2014, pp. 695–706.

[33] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *MICRO '03*, 2003, pp. 29–.

[34] X. Xu and M.-L. Li, "Understanding soft error propagation using efficient vulnerability-driven fault injection," in *DSN '12*, 2012, pp. 1–12.

[35] A. Das, F. Mueller, C. Siegel, and A. Vishnu, "Desh: Deep learning for system health prediction of lead times to failure in HPC," in *HPDC '18*, 2018, pp. 40–51.

[36] C. Kalra, F. Previlon, X. Li, N. Rubin, and D. Kaeli, "PRISM: Predicting resilience of gpu applications using statistical methods," in *SC '18*, 2018, pp. 69:1–69:14.

[37] I. Laguna, M. Schulz, D. F. Richards, J. Calhoun, and L. Olson, "IPAS: Intelligent protection against silent output corruption in scientific applications," in *CGO '16*, 2016, pp. 227–238.

[38] A. Vishnu, H. V. Dam, N. R. Tallent, D. J. Kerbyson, and A. Hoisie, "Fault modeling of extreme scale applications using machine learning," in *IEEE International Parallel and Distributed Processing Symposium, IPDPS*, 2016, pp. 222–231.

[40] B. Farahani and S. Safari, "A cross-layer approach to online adaptive reliability prediction of transient faults," in *2015 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*, Oct 2015, pp. 215–220.

[41] L. Yu, D. Li, S. Mittal, and J. S. Vetter, "Quantitatively modeling application resilience with the data vulnerability factor," in *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2014, pp. 695–706.

[42] S. S. Mukherjee, C. T. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "Measuring architectural vulnerability factors," *IEEE Micro*, vol. 23, no. 6, pp. 70–75, Nov 2003.

[43] O. Subasi, S. Di, L. Bautista-Gomez, P. Balaprakash, O. Unsal, J. Labarta, A. Cristal, S. Krishnamoorthy, and F. Cappello, "Exploring the capabilities of support vector machines in detecting silent data corruptions," *Sustainable Computing: Informatics and Systems*, vol. 19, pp. 277 – 290, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2210537917300896

[44] O. Subasi, S. Di, P. Balaprakash, O. Unsal, J. Labarta, A. Cristal, S. Krishnamoorthy, and F. Cappello, "Macord: Online adaptive machine learning framework for silent error detection," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2017, pp. 717–724.

[45] https://github.com/pnnl/IMIC, iMIC : Iterative Method Fault Injection Collection.