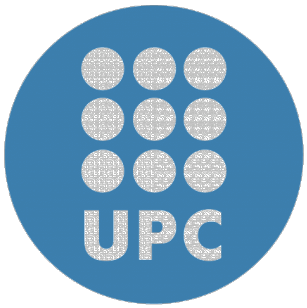
AN EXTENSIVE STUDY ON ITERATIVE SOLVER RESILIENCE: CHARACTERIZATION, DETECTION, AND PREDICTION



Burcu O. Mutlu

Department of Computer Architecture
Universitat Politècnica de Catalunya

This dissertation is submitted for the degree of
Philosophiæ Doctor (PhD)

September 2019

1st Advisor: Osman Ünsal
Barcelona Supercomputing Center, Barcelona, Spain

2nd Advisor: Gökçen Kestor
Pacific Northwest National Laboratory, WA, USA

Thesis Tutor / Internal Examiner: Adrian Cristal Kestelman
Universitat Politècnica de Catalunya, Barcelona, Spain

Declaration

I herewith declare that I have produced this document without the prohibited assistance of third parties and without making use of aids other than those specified; notions taken over directly or indirectly from other sources have been identified as such. This document has not previously been presented in identical or similar form to any other examination board.

The thesis work was conducted from November 2015 to March 2019 under the supervision of Osman Ünsal (Universitat Politècnica de Catalunya, Barcelona, Spain) and Gökcen Kestor (Pacific Northwest National Laboratory, WA, USA).

Burcu O. Mutlu
September 2019

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my thesis advisors, Osman Unsal and Gokcen Kestor for their guidance, patience and support throughout my Ph.D. studies. Their experience and wisdom added a lot to my graduate experience and inspired me to be a better researcher. I would also like to thank Adrian Cristal Kestelman for his support throughout my studies. I am particularly grateful to have a chance to work with Sriram Krishnamoorthy, whose vision and mentorship guided my time as an intern at Pacific Northwest National Laboratory.

I would also like to show gratitude to my committee, including Marc Casas, Leonardo Bautista-Gomez, Ramon Canal, Ferad Zyulkyarov and Roberto Gioisa for their insightful comments and encouragement. I would like to acknowledge Ryan Friese and Roberto Gioisa as the readers of this thesis, and I am gratefully indebted to their very valuable comments on this thesis.

I sincerely thank all my friends and colleagues from Barcelona Supercomputing Center, Pacific Northwest National Laboratory, Tri-Cities, and Istanbul. You gave me strength and energy as my studies carried me accross continents.

My special thanks are extended to my family, my parents and my sister, for their support thorughout this journey. Last but not least, I thank Erdal Mutlu who has been a constant source of support and encouragement during the challenges of graduate school and life. Your love and guidance made me push through the hardships.

This work is in parts supported by the following projects;

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.

European Union Mont-Blanc 2 Project (www.montblanc-project.eu), grant agreement no. 610402.

European Union's Horizon 2020 LEGaTO Project (www.legato-project.eu), grant agreement no 780681.

Abstract

Soft errors caused by transient bit flips have the potential to significantly impact an application's behavior. This has motivated the design of an array of techniques to detect, isolate, and correct soft errors using microarchitectural, architectural, compilation-based, or application-level techniques to minimize their impact on the executing application. The first step toward the design of good error detection/correction techniques involves an understanding of an application's vulnerability to soft errors. This work focuses on silent data corruption's effects on iterative solvers and efforts to mitigate those effects.

In this thesis, we first present the first comprehensive characterization of the impact of soft errors on the convergence characteristics of six iterative methods using application-level fault injection. We analyze the impact of soft errors in terms of the type of error (single- vs multi-bit), the distribution and location of bits affected, the data structure and statement impacted, and variation with time. We create a public access database with more than 1.5 million fault injection results. We then analyze the performance of soft error detection mechanisms and present the comparative results. Motivated by our observations, we evaluate a machine-learning based detector that takes as features that are the runtime features observed by the individual detectors to arrive at their conclusions. Our evaluation demonstrates improved results over individual detectors. We then propose a machine learning based method to predict a program's error behavior to make fault injection studies more efficient. We demonstrate this method on assessing the performance of soft error detectors. We show that our method maintains 84% accuracy on average with up to 53% less cost. We also show, once a model is trained further fault injection tests would cost 10% of the expected full fault injection runs.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

Exascale era is right around the corner, and with the vast capacity high performance computers will provide, they are of crucial importance to many scientific fields. With high performance computing, computations and simulations that were not possible to finish feasibly are computed in timely manners.

Correctness of the results are as important as the speed, since fast delivered wrong results wouldn't be useful, besides if undetected, could be very harmful. As the number of components multiply, HPC systems will become much more prone to faults. Mean time between failures is expected to become less than an hour (Table 1.1). When some scientific calculations are taking days, even weeks it is obvious that resilience is a key issue for HPC systems, and it will be even more important with the emergence of exascale systems in the near future. It comes as no surprise that resilience is reported as one of the top exascale research challenges by European Technology Platform for High Performance Computing (ETP4HPC) [30] and United States Department of Energy (DOE) [53].

System	Cores	MTBF/I
ASCI Q	8192	6.5 hours
ASCI White (2001)	8192	5 hours
ASCI White (2003)	8192	40 hours
PSC Lemieux	3016	9.7 hours
Google	15000	20 reboots a day
Exascale	more than 10^6	<1hr projection

Table 1.1 Reported and projected mean time between failure/interrupts (MTBF/I) for high performance systems. [40, 53]

When a transient bit-flip affects a hardware component, the application is said to be impacted by a soft error. When a soft error escapes the hardware detection and impacts

the application state, it can impact execution by leading to incorrect results or significantly impacting application execution times. Architectural trends such as near-threshold voltage operation and constrained power budgets exacerbate the frequency and impact of soft errors. Among the two main types of failure modes, stop-failures and Silent Data Corruptions (SDCs), SDCs are particularly hazardous and difficult to cope with since they are undetected by the underlying hardware. They silently corrupt the application data resulting in a soft error. When an execution takes days and hundreds of cores to finish, it is important to trust the outcome, as it is not feasible to do multiple runs for confirmations. Consequently, it is most important to analyze and understand the effects of SDCs in HPC applications.

Sparse matrix operations are crucial to current and future generation of high performance computing. They are at the core kernel of engineering fields such as physics based modeling and simulation, circuit simulation, mechanics of materials, geophysics and many other application fields. These scientific computations calculate solutions with structured or unstructured grid models, that leads to large sparse linear systems. Algorithms enforce the sparsity of the systems, in doing so computational and storage efficiency of sparse systems is leveraged.

Sparse matrix operations are so important for high performance computing that benchmarks for HPC systems use sparse matrix calculations. Dongarra et al [26] created the High Performance Linpack (HPL) benchmark to assess HPC system performances using floating point operations (FLOPS), back in 1979. However, although these dense matrix calculations are efficient to determine the speed and performance of an HPC system, by essentially calculating number of FLOPS it can sustain for a given time, HPL is not representative of an actual workload of high performance systems. Hence, Dongarra et al. proposed HPCG [25] that uses conjugate gradient iterative methods as a complementary benchmark to list HPC machines. Now, twice a year when Top500 list is announced for world's top HPC systems, they also announce a second list using HPCG benchmark[75].

Sparse matrices are extensively used to represent and solve linear system of equations in HPC applications. In principle, linear systems can be solved using direct methods like Gaussian elimination. However, direct methods require absence of round-off errors to reach a correct solution and they are highly expensive. Gaussian elimination would require around N^3 arithmetic operations for N linear equations, this becomes unattainable where scientific applications yield millions or billions of equations. Also usage of floating point numbers means numeric precision is not feasible for HPC systems. These shortcomings of direct methods encourages researchers to leverage iterative methods (iterative solvers). Iterative methods solve the linear system of equations with convergence analysis. They generate more refined solutions in each iteration using mainly matrix operations. Which are less costly

than direct methods in computation and storage. Hence analyzing the iterative solvers' error behavior is of high importance for HPC future as well as creating new methods of resilience as systems progress.

Current measures to mitigate the effect of soft errors include error correcting codes (ECC), checkpoint restart techniques and error detection mechanisms. While memory structures such as DRAM and caches can be protected by ECC mechanisms such as SECDED or Chipkill, memory structures in legacy GPUs and FPGAs are not protected by ECC. Even when ECC is available, it could be turned off for performance and energy savings [19]. Processor data-path structures are not commonly protected by ECC, and a single bit flip in the data-path may manifest itself as a single or multiple bit flip in application state [68]. Moreover, multiple bit flips have been observed in low-power DRAMs [5] and caches operating at close to threshold voltage [36]. Single-bit flip errors enable systematic exploration of the space of possible errors. As the systems keep scaling up; it is predicted that checkpoint/restart - by far the most popular technique to minimize fault's effects [35] - will not be able to scale. As error rates increase, it is expected that the time between failures will be too short to have checkpoints in time [53]. Hence, new detection and mitigation techniques are needed. For this, we need deep understanding of the underlying mechanisms' reaction to errors, we need new techniques to handle errors as well as new techniques to facilitate those who are developing mitigation techniques.

Understanding the behavior of iterative solvers in the presence of failures will shed a light on understanding the susceptibility of solvers' results to silent errors. It will help broaden the error handling techniques. This knowledge will also be useful in designing new methods that facilitates the development of mitigation techniques.

In this thesis we focused on the soft error vulnerability and behavior of iterative solvers. Our contribution to field can be itemized as follows;

- Providing the first comprehensive characterization of the impact of soft errors on the convergence characteristics of six iterative methods using application level fault injection
- Creating a publicly open database with millions of fault injection results for further analysis in the field
- Comparative evaluation of state of the art soft error detectors for iterative solvers
- Design and evaluation of a machine learning based detector trained on the features that are used by the analyzed detectors

- Developed a strategy to optimize soft error impact analysis using machine learning. Using the trajectory analysis of a fault, this method can predict the error behavior, get an error profile of the application and can be used to efficiently assess detector performance.

In consequence of above studies, we observed that error-induced behavior of solvers vary widely. Vectors in the algorithms are not equally impacted by errors, which can pave the way for selective protection mechanisms. Interestingly, having a pre-conditioner can have a noticeable impact on the error behavior of solvers. Given the diverse characteristics of solvers and datasets observed, we surmise that fault injection and detection evaluations needs to use several solvers and datasets. Secondly, we derive that even though the detectors we studied have their strengths, none of them were flawless. Hence, we hypothesized machine learning and combining the powers of those detectors could offer more performance, and show improved results using features monitored by the detectors as our machine learning model's features. Conducting these studies showed us adequate fault injection studies are costly and needed often by detection and error profiling studies. So we developed a method to model the error behavior of iterative solvers to speed up this process. Our machine learning error behavior profiling approach was able to assess detector performances with 84% accuracy on average with up to 53% less cost. We showed that, once a model is trained further fault injection tests would cost 10% of the expected full fault injection runs.

1.1 Thesis Structure

The remainder of this thesis is structured as follows.

Chapter 2: State of the Art This chapter gives an overview of the state of the art on several aspects of this study. Sections include related works about iterative solvers, software fault injection, fault detection and prediction, and using machine learning for fault mitigation.

Chapter 3: Background This chapter introduces the main components used in this study. Iterative solvers, datasets used to test these solvers, and SDC detectors. We give details of the iterative solvers used in these experiments, also we explain the setup for using the datasets. We also explain state of the art SDC detectors we employed in this study, and detail their setups.

Chapter 4: Solver Characterization Handling the errors in a system starts with understanding their nature. This chapter focuses on characterizing the soft error behavior of 6

iterative solvers. Results from an exhaustive injection campaign on six iterative solvers, 28 datasets, and several error manifestation strategies are reported in this chapter. Behavior of the solvers are characterized from several angles, and discussed.

Chapter 5: Detector Characterization Next step in this study is to characterize the state of the art error detection mechanisms for iterative solvers. This chapter shows the results from a comparative study of four detectors. Detection performances among solvers for the same detector and performances differences of detectors are studied. Reasons behind these differences are also discussed in this chapter.

Chapter 6: Machine Learning Based Detection This chapter explores the room for improvement in detection. We introduce a machine learning setup that uses features from all the detectors discussed and leverages all features to decide on an SDC. Several machine learning algorithms and techniques have been used and results are reported in this chapter.

Chapter 7: Soft Error Prediction This chapter proposes a trajectory analysis mechanism to relieve the cost of testing for fault error injection studies. Error injection is monitored for a short amount of time and using machine learning algorithms final outcome is predicted. This chapter reports the analysis of the prediction performance as well as experimental results for a detector performance study using the prediction.

Chapter 8: Future Work and Conclusions Final chapter of this thesis gives an outlook on the future work. Future work includes designing a composition strategy for SDC detectors to mitigate the fact that no one detector performs perfectly. We plan to explore the room for improvement when we make the detectors work in harmony.

Chapter 2

State of the art

2.1 Analyzing Iterative Methods

Elliott et al. [29] characterize the GMRES iterative solver in terms of the impact of single-bit soft errors using numerical analysis of the algorithm. Such analysis complements our approach as it considers multiple error-injection scenarios and data-set-specific characteristics. Another characterization section is presented in the [67] which apply "smart" sampling techniques to the ABFT based on the matrix structure for sparse computations. This uses preconditioned CG and IR to showcase the accuracy or "goodness" of their techniques. Our study is broader and we can use their sampling to improve our coverage. Through theoretical analysis, Shantharam et al. [61] observe that soft errors can significantly degrade convergence, sometimes taking 200 times more iterations, in CG and preconditioned CG. Bronevetsky and de Supinski [8] evaluate soft error vulnerability of iterative methods together with recovery techniques focusing solely on single bit flips.

There are also many previous efforts using iterative solvers to evaluate soft error detection strategies [8, 67, 76, 13, 23, 73, 52]. In their work [67], Sloan et al, focuses on matrix vector multiplication calculations and bases the work on arithmetic errors. Tao et al, also focuses on arithmetic erros on matrix vector calcualtions [76]. Authors of [13] use the mathematical nature of the iterative solvers to create a detection mechanism and demonstrates on Cholesky solver. In [54], Malkowski et al. uses cache vulnerability to analyze the soft error resilience of solvers.

2.2 Error Injection Strategies

Many resilience studies based on fault injection campaigns use random fault injection [16, 55, 49, 3, 12]. Random fault injection enables statistical coverage of a large 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 vectors—to focus our efforts on the key data structures that are modified in the iterative methods.

Cho et al. [16] observed that lower-level fault injection approaches are more accurate than higher-level injection studies. Hsueh et al. [41] and Ziade et al. [89] survey common fault injection tools, concluding that Register Transfer Level (RTL) fault injection [55] (examples include VERIFY[65] and MEFISTO-C[34]) as well as injecting faults into actual hardware using a particle accelerator are most accurate, however those methods are too low-level to make detailed characterization of realistic applications and data sets.

Architecture-level fault injection [50, 7, 32] addresses some of these challenges but can still be expensive to perform injection campaigns such as the one presented in this thesis. Moreover, it is very difficult to develop specialized architecture-level fault injectors for a particular class of algorithms such as iterative solvers. FERRARI (Fault And Error automatic real time injection)[45] uses traps to insert faults on CPU components (branches, registers, instructions), memory, or bus packets. They can be time or place triggered. FTAPE (Fault Tolerance and Performance Evaluator) [78] can inject bit flips on CPU modules, memory location, and disk subsystems using fault injection drivers attached to the OS and are inserted based on activity levels. FIAT (Fault Injection based automated testing) [60] is an environment that allows to develop several fault injection scenarios by allowing the experimenters to decide where, when and how faults are injected. It can inject faults in messages, as they can be corrupted, lost, or delayed; and on tasks, which can be delayed or aborted, as well as on timers. DOCTOR [38] allows injections into the CPU, memory and network messages. It is an important tool when simulating memory faults because the subtle and non-deterministic way that a memory fault might appear. It uses three triggering mechanisms: time-out for memory faults, traps for transient ones, and compilation based when dealing with permanent CPU faults.

Software-based error injection can be performed using binary instrumentation [49], compile-time transformations [63, 83, 10, 66], or operating system level injection [45]. KULFI [63] and VULFI [62] are LLVM-based fault injection tools that uses the LLVM infrastructure to simulate transient faults in CPU state elements. Compared to other approaches, it provides fine-grained error injection control and features to control where (on the control flow of the program) and how (register, load/store, branches, etc.) to insert the

fault. PDSFIS [42] uses the PIN Intel framework to inject faults without changing the source code or recompiling. It can target any software components that are visible to the PIN tools (including dynamic libraries) and can be used to do pattern based fault injection into specific software components.

Each technique stresses distinct aspects of an application's footprint (e.g., architectural registers vs intermediate representation, 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 [70, 85] (as compared to the architecture vulnerability factor [57]).

An important aspect of such studies is the coverage of the error injection experiments. Xu and Li [84] proposed statistical fault injection coverage and pruning of the testing space to increase the performance and coverage as compared to "blind" sampling. We prune the error injection space by studying the implementations of the iterative methods. Our approach complements the one presented by Xu and Li, and can be used in conjunction when an iterative method is used in the context of a larger application.

2.3 SDC Detection and Mitigation

Research on SDC detection can be categorized mainly into three different categories: algorithm-based fault tolerance (ABFT), runtime analysis techniques, and replication of computation techniques.

ABFT [79, 17, 67, 18] techniques are tailored solutions to specific numerical algorithms. Consequently, they are usually efficient. However, they fundamentally lack the ability to be applicable to algorithms other than the specific numerical or algebraic kernel they are designed for.

Runtime data analysis recently has gained attention in the HPC community. These techniques can be further classified into temporal or spatial method depending on the type of interpolation they perform. Moreover, these techniques can be also classified based on whether the training is done online or offline. Studies [6, 23, 22] investigate and compare different prediction methods such as linear curve fitting or autoregressive moving average (ARMA) models, to detect SDCs. These are online and temporal techniques. They investigate the evolution of the data over time, as a consequence these studies incorporate temporal aspects of data evolution. The main drawbacks of temporal data analytics are the memory overhead and the computation cost of maintaining snapshot data. In contrast, as a spatial and online technique, SSD [73] incurs low memory cost while having low computation overhead. On the offline side, the Sirius [77] is a neural network based offline SDC detection tool. It

generates application specific invariants to be checked at runtime. Techniques such as the Sirius are limited by the coverage of training datasets.

Replication-based schemes [33] can be deployed for mission-critical situations. In such contexts, double or triple redundancy of computation is performed in order to detect SDCs by comparing the results of replica computations. The inherent drawback of the replication is its high power/energy cost; e.g., with double redundancy, the cost is 100%. Partial replication [74] has been proposed to decrease costs while providing required level of reliability. Although partial replication is promising, it may not be applicable for certain HPC systems, mainly because errors may not be reproducible for some systems, such as heterogeneous systems.

As part of our studies, we have encountered several efforts on how to characterize the (or similar) problems that we tackled in this thesis. This is a collection of the most aligned efforts that the community is (or had) undertaken. In [29], the authors present a characterization of the GMRES iterative solver that is used to create detectors that bound the error introduced by faults. It provides the general intuition on the solver behavior that tell us that if the fault is introduced in the inner part of the solver, the solver will actually converge in all cases. This is similar to our solver analysis with the difference that we are broader on our scope (more solvers) but not as deep. Another characterization section is presented in [67] which apply "smart" sampling techniques to the ABFT based method on the matrix structure for sparse computations. This uses preconditioned CG and IR to showcase the accuracy or "goodness" of their techniques. Again, our study is broader and we can use their sampling to improve our coverage.

In [81], 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 test cases. This research explores another family of Machine Learning techniques that can be exploited in the Iterative solver space. In the case that we need to inject faults into a program, the actual methodology of injection is important. Although we chose a more controlled error injection methodology, there exists a substantial set of injection methodology and tools that we can exploit. Some examples are presented below.

FERRARI (Fault And Error automatic real time injection)[45] is an injector tool that uses traps to insert fault on CPU components (branches, registers, instructions), memory, or bus packets. They can be time or place triggered. FTAPE (Fault Tolerance and Performance Evaluator) [78] can inject bit flips on CPU modules, memory location and disk subsystems using fault injection drivers attached to the OS and are inserted based on activity levels. FIAT (Fault Injection based automated testing) [60] is an environment to develop several fault

injection scenarios by allowing the experimenters to decide where, when and how faults are injected. It can inject faults in messages, as they are being corrupted, lost, or delayed, on tasks, as they are delayed or aborted, and timers. DOCTOR [38] allows injections into the CPU, memory and network messages. It is an important tool when simulating memory faults because the subtle and non-deterministic way that a memory fault might appear. It uses three triggering mechanisms: time-out for memory faults, traps for transient ones, and compilation based when dealing with permanent CPU faults. KULFI [63] / VULFI [62] is an LLVM based fault injection tool that uses the LLVM infrastructure to simulate transient faults in CPU state elements. Compared to other approaches, it provides fine grained error injection control and it provides a large set of features to control where (on the control flow of the program) and how (register, load/store, branches, etc) to insert the fault.The PDSFIS fault injector [42] uses the PIN Intel framework to inject faults without changing the source code or recompiling. It can target any software components that are visible to the PIN tools (including dynamic libraries) and can be used to do pattern based fault injection into specific software components.

2.4 Machine Learning and SDC Prediction

Recently, there have been many efforts to utilize machine learning [3, 20, 44, 48] to address resilience problems. IPAS [48] uses machine learning to decide on instructions that will likely to lead to corruption and duplicates them. Desh [20] uses systems logs and neural networks to predict node failures.

In [81], 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 test cases. Authors of [80] employed support vector machines to create an online soft error vulnerability prediction mechanism for memory arrays.

Farahani et. al. leverages architecture vulnerability factor to create an online reliability prediction mechanism for transient faults [31]. Several efforts focused on vulnerability factors for modeling error resilience of programs. In [71], authors practice fault modeling on the program level. They suggest the Program Vulnerability Factor for assessing the vulnerability of a software resource. Yu et. al. proposes the Data Vulnerability Factor [86] which models the vulnerability of individual data structures in an application relying on access patterns. Architecture Vulnerability Factor [56] on the other hand, models the probability of an error happening when a fault occurs in that hardware component.

Chapter 3

Background

3.1 Iterative Solvers

We consider iterative methods to solve a system of linear equations

$$A \cdot \vec{x} = \vec{b}, \tag{3.1}$$

where A is a sparse matrix, $\vec{b}$ and $\vec{x}$ are vectors. At each iteration, the methods compute an approximate value of $\vec{x}$. Execution completes when the norm $|\vec{r}| = |\vec{b} - A \cdot \vec{x}|$, referred to as residual norm, is lesser than the required threshold. We consider six iterative methods [37]:

- CG: Conjugate gradient with incomplete LU preconditioner
- ICCG: CG with incomplete cholesky preconditioner
- BiCG: Unpreconditioned biconjugate gradient
- BiCGSTAB: Biconjugate gradient stabilized
- CGS: Conjugate gradient squared
- QMR: Quasi-Minimal Residual method

CG and ICCG require A to be a symmetric positive-definite matrix and differ primarily in the preconditioner used. BiCG generalizes CG and can handle non-symmetric and non-definite systems. BiCGSTAB is more numerically stable than BiCG, but computationally more expensive. CGS is more stable than BiCG but less expensive than BiCG. These methods belong to the class of non-stationary iterative methods. Each method considered differs from another method in a small yet significant way. These methods were chosen to investigate

Listing 3.1 The implementation used for CG and ICCG (CG with preconditioner) methods for solving the symmetric positive-definite system $A \cdot \vec{x} = \vec{b}$

```
  for (int i = 1; i <= max_iter; i++) {
      z = M.solve(r);
      rho(0) = dot(r, z);

      if (i == 1)
          p = z;
      else {
          beta(0) = rho(0) / rho_1(0);
          p = z + beta(0) * p;
      }
      q = A*p;
      alpha(0) = rho(0) / dot(p, q);
      x += alpha(0) * p;
      r -= alpha(0) * q;
      resid = norm(r) / normb;
      rho_1(0) = rho(0);

      if (resid <= tol) {
          tol = resid;
    max_iter = i;
    return 0;
      }
    }
    tol = resid;
    return 1;
  }
```

the impact of selected differences in a group of related algorithms. Identical behavior under error would indicate that a study of one of these solvers is representative of others. If not, any study of error behavior of iterative methods needs to consider multiple iterative methods.

For this characterization, we used the implementation of these methods in the Iterative Methods Library (IML++) v1.2a [24]. In this library, the implementation of the methods is done using high-level API similar to the algorithmic descriptions of the iterative methods.

Method Implementations

Pseudocode of all the solvers are provided to the reader in the appendices. CG solver's implementation is also provided here for the algorithms and how their statements are used.

3.2 Datasets

The methods were evaluated using the matrices in the SuiteSparse matrix collection [21]. To enable comparative evaluation, we selected the symmetric positive-definite matrices from the collection, which can be evaluated on the methods considered. Of these 31 matrices, three converged quickly ($<$10 iterations). Therefore, we focus on the remaining 28 matrices in this study. Table 3.1 shows the list of the datasets used. As shown in Table 3.1, the number of iterations performed by a solver varies from one dataset to another one. The matrices in the library are used to initialize A in equation 3.1. The vector $\vec{b}$ is determined as $A \cdot \vec{1}$, where $\vec{1}$ is a vector of all ones, as has been done in other efforts [8]. We executed the iterative methods until the residual norm is less than 10^{-6}.

As execution progresses, the residual norm does not always strictly decrease. Soft errors might also lead to non-monotonic changes in the residual norms, making it difficult to discriminate error-induced behavior from normal behavior. This makes it difficult to differentiate an iterative method's execution in the presence and absence of soft errors.

3.3 Detectors

We consider four online soft error detection algorithms.

Adaptive impact-driven detection (AID) Di and Cappello [23] observed that the impact of "influential" soft errors can be characterized by an *impact error bound*, defined as the maximum ratio of the data value change between adjacent time steps to the global value range for every data point in a snapshot. This observation is used to dynamically fit different curves—last-state, linear, and quadratic—to the temporal evolution of data based on their prediction error. An error is flagged if the observed value falls out of the impact error bound range.

Orthonormality detector (Ortho) Chen et al. [13] identify an intrinsic orthogonal relationship of specific vectors in several Krylov linear solvers. They construct a detector that periodically checks this relationship and flags an error if the orthogonality condition is violated (within a certain tolerance). Among the solvers considered, the orthogonality relationship only exists for CG, ICCG, and BiCG.

Checksums for matrix-vector multiplication (New-Sum) Tao et al. [76] presented a checksum encoding scheme for matrix-vector multiplication and vector linear operations. The

Matrix	Rows	NZ%	Number of iterations					
			CG	ICCG	BiCG	BiCG -STAB	CGS	QMR
af_shell3	504855	0.01	1117	572	1117	524	1214	890
af_shell4	504855	0.01	1117	572	1117	524	1214	890
af_shell7	504855	0.01	1118	571	1118	466	1327	891
af_shell8	504855	0.01	1118	571	1118	466	1327	891
bcsstk13	2003	2.1	928	330	928	462	1185	-1
bcsstk14	1806	1.95	195	101	195	108	108	6736
bcsstk15	3948	0.8	453	166	453	198	207	27371
bcsstk16	4884	1.2	148	49	148	95	94	333
bcsstk24	3562	1.3	451	727	451	711	374	-1
bcsstk27	1224	3.7	185	59	185	142	155	3844
bcsstk28	4410	1.12	4344	1309	4344	16226	7381	13763
bcsstk38	8032	0.55	426	119	426	156	1218	-1
ex3	1821	1.6	181	123	181	155	3625	790
ex9	3363	0.88	153	44	153	181	-1	5969
ex13	2568	1.15	146	34	146	101	104	11632
ex15	6867	0.21	96	33	96	72	82	-1
Kuu	7102	0.67	378	116	378	260	-1	570
msc04515	4515	0.48	2169	1176	2169	3580	-1	9394
NASA2146	2146	1.6	171	56	171	116	99	603
Pres_Poisson	14822	0.33	662	218	662	709	669	6335
sts4098	4098	0.43	244	96	244	158	219	-1
s1rmq4m1	5489	0.87	612	172	612	489	639	22828
s2rmq4m1	5489	0.87	1237	546	1237	422	-1	-1
s3rmq4m1	5489	0.87	2969	1274	2969	2096	2530	-1
s1rmt3m1	5489	0.72	695	205	695	576	683	22661
s2rmt3m1	5489	0.72	1787	657	1787	1733	2085	-1
s3rmt3m1	5489	0.72	4497	2111	4497	1937	4514	-1
s3rmt3m3	5357	0.72	8538	2487	8538	5037	12097	-1

Table 3.1 Sparse matrices selected from SuiteSparse and the number of iterations performed by solver for each dataset. -1 denotes that the solver does not converge to achieve the norm of the residual error below 10^{-6}. We exclude these cases from our analysis.

checksums are maintained by augmenting each solver operation with an efficient checksum operation. The separate checksum is recomputed from the current matrices periodically and compared to the one computed at each iteration step. The detector flags an error if the two checksums differ.

Moving average detector (MAD) Liu et al. [52] observe that the residual norms in iterative methods might not strictly decrease at every iteration but that the norms exhibit a decreasing trend over a longer period (multiple consecutive iterations). Rather than focusing on one iteration at the time, the authors employ a moving average of the residual norms over a sliding window. The detector flags an error if the moving average increases with respect to the previous window rather than decreasing, as expected.

Each detector also requires the user to choose a parameter that acts as a threshold to flag an error. For this study we used the default threshold values provided by the developers. Table 3.2 lists the detectors and shows the parameters used for each detector.

Table 3.2 shows the information about the parameters used in our evaluation.

Detector	Parameter	Parameter value
AID	Impact error bound	0.00078125
Ortho	Tolerance	10^{-10}
MAD	Fixed parameter	0.1
New-Sum	Tolerance	10^{-10}

Table 3.2 Parameters of detectors used in our evaluation.

3.4 Machine Learning Algorithms

In this work, we employ machine learning to combine soft error detectors and to model the fault behavior of iterative solvers. We leverage several well-established machine learning techniques. In this section, we provide a brief overview of the machine learning algorithms we employed. These are the well-established algorithms that cover different classification techniques we used from SciKit Learn [59].

- **Decision tree (DT)**: Decision tree is a tree-based model. The leaves represent class labels and the branches represent conjunctions of features that lead to these class labels.
- **Support vector machines (SVM)**: SVM is a supervised method that builds an hyper-plane between training instances and classifies samples according to their position with

respect to that hyperplane. This method has been proved to work well on non-linearly separable training sets and generally shows good precision and recall.

- **AdaBoost (AB)**: This an iterative machine learning algorithm that improves its precision by increasing the weights of mis-classified samples. The algorithm uses techniques that combines the outputs of weak learners (arbitrary learning algorithms) and formulates a weighted sum of the outputs as the final output.
- **AdaBoost Regression with Decision Tree**: This estimator uses Decision Trees as a base estimator and improves its original classifier by adjusting weights.
- **Stochastic gradient boosting (SGB)**: SGB constructs an additive model in a stage-wise fashion. This method effectively combines boosting with gradient descent algorithms.
- **Random forest (RF)**: Over-fitting is a common issue for several machine learning algorithms. Randomized forest addresses this issue by constructing a set of decision tree classifiers during the training stage and averaging them.
- **Extremely Randomized Trees (ET)**: This algorithm is a slightly different algorithm than the random forest. As in a random forest, a random subset of candidate features is used. However instead of searching for the most discriminative thresholds, thresholds are drawn at random for each candidate feature.
- **Bootstrap Aggregation Techniques (Bagging)**: Bagging is an ensemble method that takes a set of classifiers and aggregate their predictions by voting or averaging to form a final decision.
- **Naive Bayes (NB)**: This machine learning algorithm is a supervised method based on the Bayes' theorem. The method makes the "naïve" assumption that every feature is independent from the rest of the features.
- **Multilayer Perceptrons (MLP)**: MLP is a feedforward neural network algorithm which maps a set of inputs onto the set of corresponding outputs. MLPs consist of multiple layers of nodes in some directed graph. Except for the input nodes, each node is a neuron having a nonlinear activation function. MLPs use back-propagation for training the network.

3.5 Performance Metrics

This thesis deals with demonstrating the performance of classification tasks in both analyzing detection mechanisms, and machine learning algorithms. In a classification problem, there are established metrics to assess the performance of the results. Following are the performance metrics used for the classification tasks in this thesis.

True Positive (TP) An instance was correctly labeled as positive (i.e. execution was erroneous and labeled as such).

False Positive (FP) An instance was labeled as positive but it was in fact negative (i.e., an execution was labeled as erroneous but the error was masked).

True Negative (TN) The instance was correctly labeled as negative (i.e. execution was vorrect and labeled as such).

False Negative (FN) The instance was labeled as negative but it was in face positive. (i.e. an execution was labeled as correct but the execution is corrupted.)

A classification is considered *precise* if it can correctly identify all the SDC cases (true positives) and only the SDC cases (no false positives). We also define *recall* as the fraction of SDCs detected over all the cases in which an SDC occurred.

More formally:

$$Precision = \frac{TP}{TP+FP} \tag{3.2}$$

$$Recall = \frac{TP}{TP+FN} \tag{3.3}$$

The precision of a classification defines how accurately it can identify an SDC, i.e., a large number of false positives decreases the detector precision. Recall, instead, defines what fraction of SDC has been detected, i.e., the number of SDC detected over all SDC that should have been detected.

We also use F-Score which combines precision and recall:

$$F - Score = 2 \times \frac{precision \cdot recall}{precisoin + recall} \tag{3.4}$$

Precison, *Recall*, and *F-score* have values between 0 and 1, where 1 indicating a perfect score.

Chapter 4

Solver Characterization

4.1 Introduction

A broad array of techniques has been designed to understand application behavior under soft errors and to detect, isolate, and correct soft-error-impacted application state. The first step toward tolerating soft errors involves understanding an application's behavior under soft errors. This can help understand the need for error detection/correction techniques. An ideal error detection/correction strategy identifies all and only the errors that can materially impact application behavior. Detecting and recovering from errors that might be eventually masked by the application can unnecessarily increase the cost of soft error resilience. Different portions of the application state might be impacted differently by a soft error, enabling optimizations and data-structure-specific resilience techniques. Finally, evaluating the effectiveness of such techniques requires a systematic evaluation of their effectiveness in protecting various portions of the application state throughout the execution.

In this chapter, we systematically characterize the behavior of six iterative methods—CG, ILU-preconditioned CG (ICCG), BiCG, CGS, BiCGSTAB, and QMR—in the presence of soft errors. These methods are exemplar of an important class of methods used to solve systems of equations and constitute the core kernel in many large-scale scientific applications. These methods employ closely related approaches to solving a system of linear equations, enabling us to understand the impact of seemingly small, albeit significant, algorithmic changes on soft error behavior.

We employ a deterministic error injection strategy to systematically explore the space of possible error behaviors. We consider 1, 2, and 4 bit error injections under uniform and beta distribution of the bit positions affected by the error. We consider all statements and vectors in the iterative methods as candidates for error injection. To reduce fault injection overheads, we identify and prune error injections that will lead to masked errors and those that will lead

to the same outcomes as other error injection configurations. In sum, we performed a total of 1,744,800 error injection runs and collected more than 2.5TB data.

4.2 Error Injection Model

To study the behavior of iterative methods in the presence of soft errors, we inject errors during the execution of these methods. In particular, we study the impact of one error (single- or multi-bit) on the execution of iterative methods.

Error injection can be performed at various abstraction levels, from circuit to application level. Error injection at the circuit-level is considered the most accurate method but it requires sophisticated infrastructures, such as radiation-exposure to processor chips [15, 15], or processor RTL simulations [16, 55]. While bombarding real hardware certainly allows the user to run full-size applications, these techniques are considerably expensive and generally destroy the testbed. RTL simulations, instead, are very accurate but also quite slow. These approaches are expensive in terms of resources and time and are only practical for small benchmarks and limited error-injection campaigns.

Architecture-level simulators [7, 32] also have been used to study the impact of soft errors on applications and partially mitigate the issues with RTL level error injections. However the execution time overheads might still be too large to study multiple executions of large applications in the high-performance computing domain.

Software-based injection techniques can perform error injection at the application level in an accelerated fashion. Software fault-injection techniques are attractive because they don't require expensive hardware. Furthermore, they can be used to target applications and operating systems, which is difficult to do with hardware fault injection. Software-implemented error injection can be performed at different levels of abstraction: PinFI [83] and BIFIT [49] are dynamic binary instrumentation-based injectors, wherein an error is randomly injected into data-structures of an application. Other tools such as LLFI [83] and KULFI [63] are compiler-level injectors, which inject errors at register level. Each technique stresses distinct aspects of an application's footprint (e.g., architectural registers vs intermediate representation, specific compiler passes, etc.).

The aforementioned tools come with pros and cons: binary instrumentation based tools enable user-defined temporal and spatial injection but they might introduce considerable overhead that limits the number of error injection experiments that can be performed. On the other hand, compiler-based tools significantly reduce the injection overhead but do not allow the user to precisely explore the temporal aspect of injection. The lack of temporal aspect

makes it hard to study and understand the correlation between the outcome and the location of the injected errors.

Given these limitations, we focus on a controlled application-specific error injection methodology by instrumenting the source code. The main property of our injection methodology is to provide easy exploration of the temporal (when the error is injected) and spatial (in which data-structures the error is injected) aspects of the error-injection space. We use application-level injection to understand application vulnerability in terms of program elements, analogous to program or data vulnerability factors [70, 85].

Exploring the entire error-injection space is time consuming but does not necessarily bring additional knowledge. For example, injecting an error in a dead vector will result in the error being masked and correct results, which we can assess without actually performing the experiment. Instead, we opt for a methodology in which we perform the minimum set of experiments that still covers the meaningful part of the error-injection space. We prune the number of error-injection experiments to be performed by identifying live vectors at each step of the algorithm and thus avoiding injecting into dead vectors.

```
   Compute r^0 = b - Ax^0 given an inital guess x^0
   for i=1,2, ...
0.   solve Mz^(i-1) = r^(i-1)
1.   ρ^(i-1) = r^(i-1) * z^(i-1)
     if i == 1
2.       p^(1) = z^(0)
     else
         β^(i-1) = ρ^(i-1) / ρ^(i-2)
2.       p^(i) = z^(i-1) + β^(i-1) * p^(i-1)
     endif
3.   q^(i) = Ap^(i)
4.   α^(i) = p^(i-1) / p^(i) * q^(i)
5.   x^(i) = x^(i-1) + α^(i) * p^(i)
6.   r^(i) = r^(i-1) - α^(i) * q^(i)
7.   residual = r^(i) . r^(i)
```

Fig. 4.1 The conjugate gradient method for solving the symmetric positive-definite system $A \cdot \vec{x} = \vec{b}$

4.3 Error-injection sites

In iterative methods, the primary data structures involve two-dimensional matrices, multiple vectors, and scalars. The matrix representing the system of equations remains read-only throughout the execution. Read-only data structures can be protected efficiently through employing simple copy-and-compare or fingerprint techniques. Scalars represent a relatively small fraction of the overall application state and are least likely to be affected by a soft error. Therefore, we focus on the vectors used in iterative methods, which are modified every iteration.

In an exhaustive error-injection strategy, a random element of every vector can be considered as a candidate for error injection before every statement. The implementation of the algorithm is similar to the one in Figure 4.1 with one function call (or operator-overloaded) statement per algorithm operation. We only consider statements that involve vectors. All iterative methods considered involve a conditional statement to initialize a subset of vectors in the first iteration. In these statements, both branches access the same vectors. Therefore, from the perspective of our error-injection strategy, we can treat both branches as part of the same statement. The eight statements in the CG algorithm that are considered for error injection are numbered in Figure 4.1. Note that both branches of the conditional statement are labeled with line number 2.

While every vector needs to be considered for error injection before every statement to ensure coverage, many of the cases lead to identical outcomes. We consider two pruning steps to identify and eliminate such redundant error-injection experiments. First, an error injected into a vector will always be masked if that vector will be overwritten before its next use. In terms of data-flow analysis, such a vector is not considered "live". When computing the overall impact of an error, errors at these positions can be noted as masked.

Second, consider the injection of error into vector r at statements 2–5 in Figure 4.1. These statements neither define nor use r. All these injections result in the same outcome: impact on statement 6 as if the error were injected just before statement 6. Therefore the error injection experiments conducted for vector r just before statement 6 can be reused to characterize the impact of an error on vector r in statements 2–5.

We classify each statement-vector pair in the program according to the following rules:

- If the value in vector v will have no further uses during or after execution of statement s, but will be overwritten with a new value, (s, v) will be classified into the *Dead* set. In Figure 4.1, (0,q), (1,q), (2,q), and (3,q) belong to this set because q will be overwritten in statement 3 before subsequent uses.

- If the value in vector v is used in executing statement s, (s,v) is placed in the used set $Used$. (2,z) and (5,x) are examples of pairs in the injection set.
- If a vector v is live at statement s ($(s,v) \notin Dead$) and v's value is not used in s ($(s,v) \notin Used$), it is classified as being $Alive$.

Table 4.1 shows the classification of statement-vector pairs for the CG algorithm (Figure 4.1). Table 4.2 reports the effects of the pruning steps in reducing the number of statement-vector pairs to be considered for error injection. For example, with CG the two pruning steps reduce this candidate set from 40 to 13 injection points.

4.3.1 Reconstructing the error behavior under full coverage

Given a classification of statement-vector pairs into $Dead$, $Used$, and $Alive$ sets, we perform error-injection experiments only on the members of the $Used$ set. Given the outcomes of these error-injection experiments, we need to compute the distribution of outcomes as if we had considered all candidates. Given a function $Distribution(s,v)$ that returns the distribution of outcomes from error-injections on a vector v before statement s, the overall error behavior can be computed as;

$$\frac{\sum_s \sum_v Distribution(s,v)}{|s|.|v|.|FI|}$$

where $|s|$, $|v|$, and $|FI|$ denote the number of statements, vectors, and error injections per statement-vector pair in the program, respectively. The distribution of outcomes $Distribution(s,v)$ is determined as :

- If $(s,v) \in Used$, $Distribution(s,v)$ is obtained from the error-injection experiments for the statement pair.
- If $(s,v) \in Dead$, $Distribution(s,v) = \{MASKED = |FI|\}$.
- If $(s,v) \in Alive$, we find the statement s' that follows s such that $(s',v) \in Used$ and return $Distribution(s',v)$.

Reproducing the error-induced behavior requires us to account for the differences in the execution times of individual statements. For example, a highly resilient and computationally expensive statement can make an algorithm more resilient, while multiple vulnerable yet inexpensive statements might have negligible impact. The execution times of individual statements will depend on the actual execution times on a given platform. To avoid tying our analysis to a specific architecture and software stack, we associate each statement with an

abstract cost metric. The statements in iterative methods can be classified into three groups; (a) scalar operations, (b) vector-scalar and vector-vector operations, and (c) matrix-vector product and preconditioners. We assume scalar operations incur zero cost. Vector-vector and vector-scalar operations incur costs proportional to the length of the vectors N (all vectors in a given execution of an iterative method are of the same size). The cost of a matrix-vector product can be approximated by $N^2 \cdot nnz$ where N is the matrix dimension size and nnz is the fraction of non-zeroes in the sparse matrix. Estimating the cost of a preconditioner step is challenging as it can involve an arbitrary number of operations. We approximate it with the cost of a matrix-vector multiply ($N^2 \cdot nnz$). Introducing these statement weights, gives us the weighted distribution of outcomes as:

$$\frac{\sum_s w_s \sum_v Distribution(s,v)}{(\sum_s w_s) \cdot |v| \cdot |FI|}$$

We will use this weighted distribution of outcomes as our primary metric in analyzing the error injection outcomes.

	p	q	r	x	z
stmt-0	■	■	■	■	■
stmt-1	■	■	■	■	■
stmt-2	■	■	■	■	■
stmt-3	■	■	■	■	■
stmt-4	■	■	■	■	■
stmt-5	■	■	■	■	■
stmt-6	■	■	■	■	■
stmt-7	■	■	■	■	■

Table 4.1 Classification of each statement-vector into ■ alive, ■ dead, and ■ used sets for the CG method described in Figure 4.1.

4.3.2 Error-injection implementation

Our error-injection framework determines when and where to inject an error based on the following inputs:

- iteration number,
- statement number,

Method	#Statements	#Vectors	#Points	#Alive	#Used
CG	8	5	40	30	13
ICCG	8	5	40	30	13
CGS	12	10	120	71	19
BiCG	12	9	108	74	19
BiCGSTAB	13	9	117	83	22
QMR	22	15	330	217	32

Table 4.2 Classification of statements and vectors in each method

- vector name,
- position in the vector,
- list of bit positions to flip in the 64-bit vector element.

Since we assume no previous knowledge about what iteration and what position in vector are more vulnerable to Silent Data Corruptions (SDCs), we randomly determine both the iteration number and the vector position in which to inject an error. We use two independent random sequences and consider two distinct error models: single- and multi-bit errors. Single-bit flip errors result from alpha particle strikes that induce a transition in a bit. While memory structures such as DRAM and caches can be protected by ECC mechanisms such as SECDED or Chipkill, memory structures in legacy GPUs and FPGAs are not protected by ECC. Even when ECC is available, it could be turned off for performance and energy savings [19]. Processor datapath structures are not commonly protected by ECC, and a single bit flip in the datapath may manifest itself as a single or multiple bit flip in application state [68]. Moreover, multiple bit flips have been observed in low-power DRAMs [5] and caches operating at close to threshold voltage [36]. Single-bit flip errors enable systematic exploration of the space of possible errors. While single-bit flips are generally easier to detect (e.g., through parity code), those that escape hardware detection and generate an SDC are generally more difficult to detect by software detectors because they introduce a smaller perturbation compared to multi-bit flips. The latter are more difficult to detect in hardware but generally introduce a larger perturbation that it is easier to detect by a software detector.

In our experiments, we inject an error in a used vector in a given statement and at a given iteration. In this work, we analyze errors that induce 1, 2 or 4 bit flips. We determine the positions of the bit to flip based on two different probability distributions: uniform and beta 5-1. The Beta 5-1 is a distribution where the probability of error occurring increases as we get closer to the higher bit locations. The uniform distribution, instead, assumes a equally likely probability of errors among the bit locations.

We performed our experiments on a 128-node cluster equipped with two AMD Interlagos [9] 16-core sockets, for a total of 32 cores per node and 4096 cores per system. We employed the solver implementations in the Iterative Methods Library (IML++) v1.2a [24]. All solvers are compiled with GCC 4.7 with -O2 optimization. Error injection space includes the number of statements in a method, the number of used vectors at a given statement in a method, see Table 4.2, 28 data sets (24 for CGS, 18 for QMR), type of error distribution and the number of bits flipped (1,2,4 bit flips for uniform and beta error distributions).

We performed 100 experiments for each instance in error-injection space. Those injections were uniformly random with respect of iteration space and position in the vector. We performed a total of 1,744,800 error-injection experiments to study the impact of soft errors on six iterative methods, generating over 2.5TB of data.

4.3.3 Outcome classification

A soft error can result in a variety of outcomes. In this study, total number of iterations was used to classify the outcomes of the injection experiments;

- **MASKED**: The execution is masked if the error injection does not change the number of iterations taken by an iterative method to converge to the correct solution.
- **FAST**: The execution exits the convergence loop (with or without the correct solution) in fewer iterations than the corresponding error-free run.
- **SAME**: The execution exits the convergence loop after the same number of iterations as error-free execution. This does not necessarily mean it outputs the correct solution.
- **ANOMALY:** The execution is anomalous, i.e., the number of iterations to converge in the presence of errors differs from the one without errors. However, the impact is not severe, defined as being less than $2\times$ the number of iterations in the error-free run.
- **ADVERSE:** The execution suffers from severe slowdown, i.e., the execution with an error takes at least twice as long (in terms of number of iterations) to converge as the one without errors.

In our categorization, we also use two categories to further classify FAST, SAME, and ANOMALY outcomes. We use the execution with the convergence checkers as the baseline for defining the outcomes. In other words, when a method exits the convergence loop, the solution is checked. If the residual error is less than a given threshold (10^{-6} in our experiments), this execution is marked as *CONVERGED* to the correct solution; otherwise

as *NOT-CONVERGED*. Note that SAME and CONVERGED together is equivalent to MASKED.

4.4 Experiments

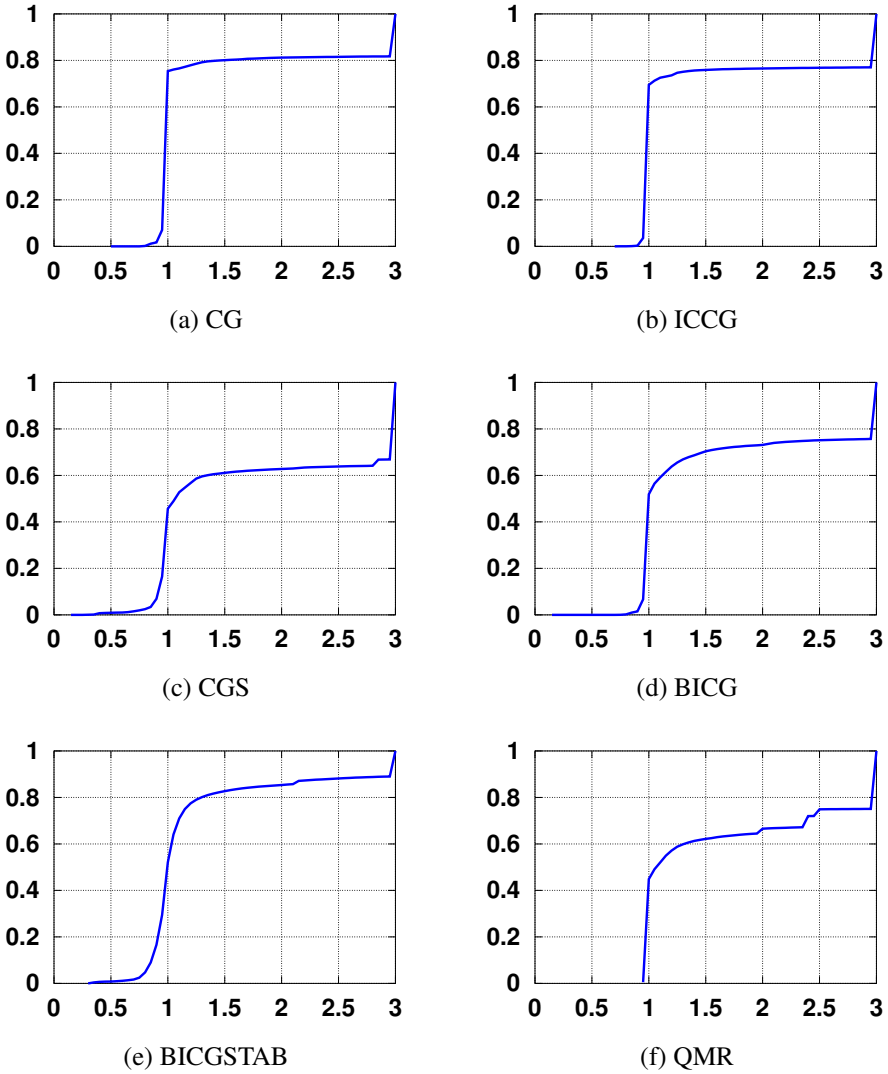
We present the key characteristics of the solvers as inferred from the error-injection experiments. We identify factors that can potentially influence the impact of soft errors to understand the distinguishing features of the behavior of each iterative method under soft error.

Impact of error on number of iterations executed

Figure 4.2 shows the impact of errors on the number of iterations executed. This is plotted as a ratio of the number of iterations executed in the presence of errors and the number in an error-free run. The figure is a cumulative distribution of the percentage of runs that led to a given ratio of iterations. A number less than 1 indicates a speedup while a number of greater than 1 indicates slowdown. A value close to 1 indicates no change. As the masked outcomes would have drowned out the other features of the graph, we do not include them in this figure. In general, we observe that a significant fraction of the non-masked outcomes result in a small change in the number of iterations, often making it faster. In the extreme, errors can result in a reduction in the number of iterations by more than 50%. Note that this could either be due to convergence or erroneously exiting the iterative loop. For all solvers, a small, but significant, fraction (up to 20% for CGS) of the non-masked errors lead to a more than 2.5$\times$ increase in the number of iterations. This shows that, in absence of timely detection, errors can have a significant impact on the overall application performance. In the rest of the discussion, we focus on the outcomes rather than the number of iterations.

Overall behavior

Figure 4.3 summarizes all the error-injection experiments for each solver. Figure 4.3a summarizes the data from the error-injection experiments without taking into account pruning as discussed in Section 4.3 We observe that different solvers result in different frequencies of masked outcomes. A small but significant fractions of runs resulted in adverse outcomes (i.e., $> 2\times$ slowdown). Interestingly, BiCGSTAB exhibited a large number of runs in which an error resulted in faster convergence. Also, CGS and QMR exhibit the largest fraction of adverse outcomes.



(a) CG (b) ICCG

(c) CGS (d) BICG

(e) BICGSTAB (f) QMR

Fig. 4.2 Cumulative distribution of the total number of iterations with error injection as compared to baseline iterations. Masked experiments not included in this histogram. x-axis: Ratio of the number of iterations with and without errors. y-axis: Cumulative distribution. For plotting clarity, all cases with more than three times baseline is associated $3\times$ slowdown.

Figure 4.3c reconstructs and weighs this data as explained in Section 4.3 to reconstruct the behavior anticipated in a more exhaustive error-injection experiment. Figure 4.3c is strikingly different from the one without weights, Figure 4.3a. The fraction of errors that are masked grows while the fraction of adverse outcomes shrinks. Importantly, the trends between solvers in terms of anomalous runs is not preserved when weights are applied.

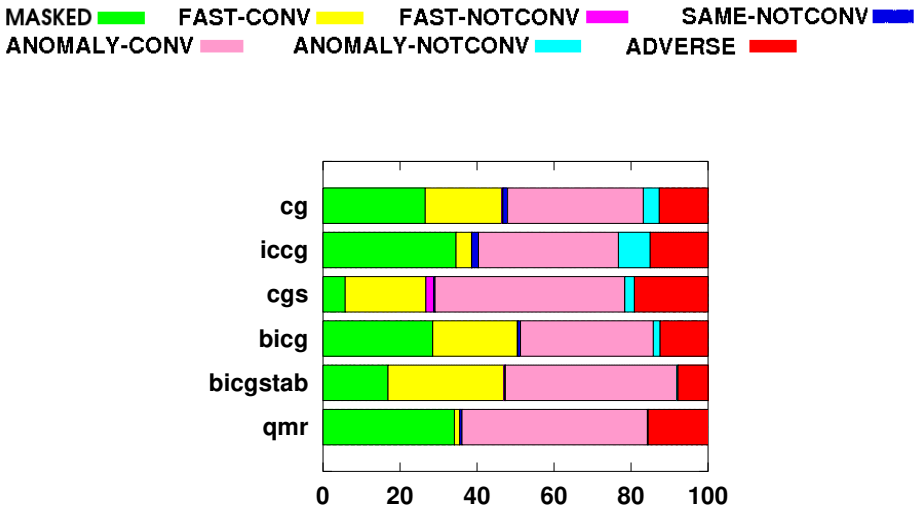
In summary, a large fraction of soft errors are masked by all six iterative methods. Therefore, detecting and recovering from all soft errors can be overly pessimistic. In addition, we find that soft errors have a non-trivial probability of leading to faster convergence, lending potential optimization opportunities.

Evaluation of soft error detection techniques needs to account for the differences due to weights associated with each statement. Even when a detection technique protects only a specific statement of a data structure, error injection with sufficient coverage is required to understand runtime behavior. We have employed an approximate weighting strategy. A more accurate weighting strategy, based on execution time analysis on a specific platform, can be combined with the presented error-injection data to derive a more precise distribution of outcomes.

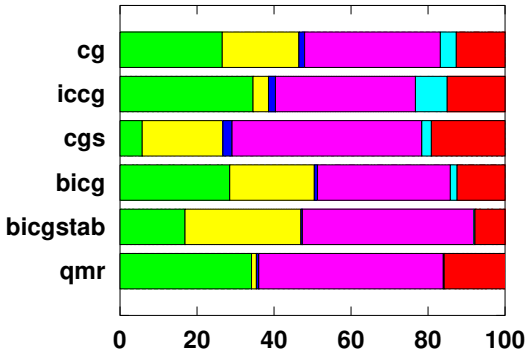
Soft error behavior for different data sets.

Figure 4.4 shows the error-induced behavior of the solvers for each data set. Blank spaces depict cases where an iterative method does not converge for a data set. We observe significant differences in the behavior of solvers between data sets. For example, CG incurs far greater masked outcomes with the `ex15` data set than with other data sets considered. Also, QMR shows clear differences in number of adverse outcomes between data sets. All solvers incur significant adverse outcomes with the `ex3` data set. A data set that is more vulnerable to soft errors when using one solver is not also vulnerable with a different solver. However, for some data sets, different solvers behave differently. Examples include CGS with the `s3rmt3m3` and CG with the `bcsstk24` data sets.

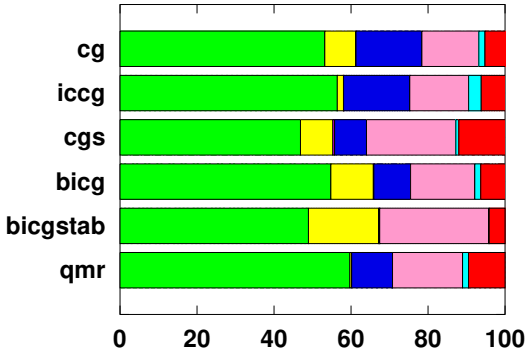
The relative behavior of the solvers can be markedly different depending on the data sets used. Therefore, choosing workloads representative of the application context of interest is crucial to meaningfully analyze the behavior of iterative methods in the presence of soft errors. Workload-independent analysis should consider as large a collection of data sets as feasible.



(a) Injection Results



(b) Weighted Injection Results



(c) Weighted & Reconstructed Results

Fig. 4.3 Overall performances of the solvers among all vectors, statements, iterations and error injections. For reconstructed graphs, expected population mean for each outcome are within (0.2%, 1.1%, 0.6%, 0.3%, 1.4%, 0.8%) of our sample mean at the maximum point with a confidence level of 95%.

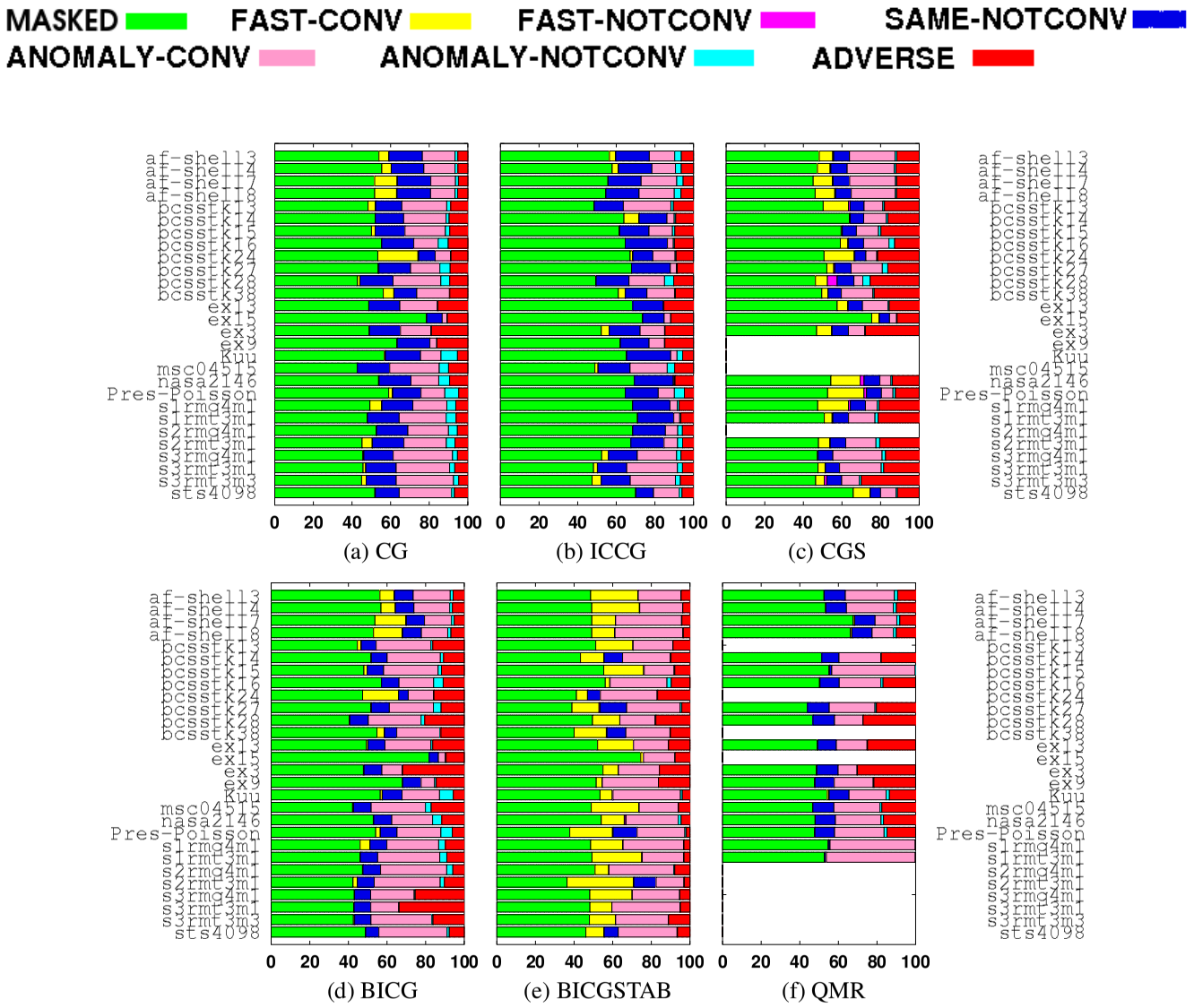


Fig. 4.4 Graphs representing the behaviour of each dataset for each solver. Expected population mean for each outcome are within (2.9%, 202.0%, 67.1%, 28.7%, 68.7%, 7.6%) of our sample mean at the maximum point with a confidence level of 95%.

Impact of error-injection strategy.

Figure 4.5 expands the summary presented in Figure 4.3c into 1, 2, and 4 bits flipped using uniform or beta distribution. In general, we observe that uniform distribution leads to a smaller fraction of adverse outcomes. In general, corrupting more bits increases the likelihood of an anomalous or adverse outcome. In the case of CGS and BiCGSTAB, the number of masked outcomes is less influenced by the error injection strategy. Interestingly, for CGS, with increase in number of bits corrupted, the fraction of adverse outcomes increases with a proportional decrease in the anomalous outcomes. In this case, the fraction of masked and fast outcomes stays the same. In summary, understanding the soft error behavior of solvers requires an understanding of the types of errors expected to affect the target environment.

Influence of the vector in which an error is injected.

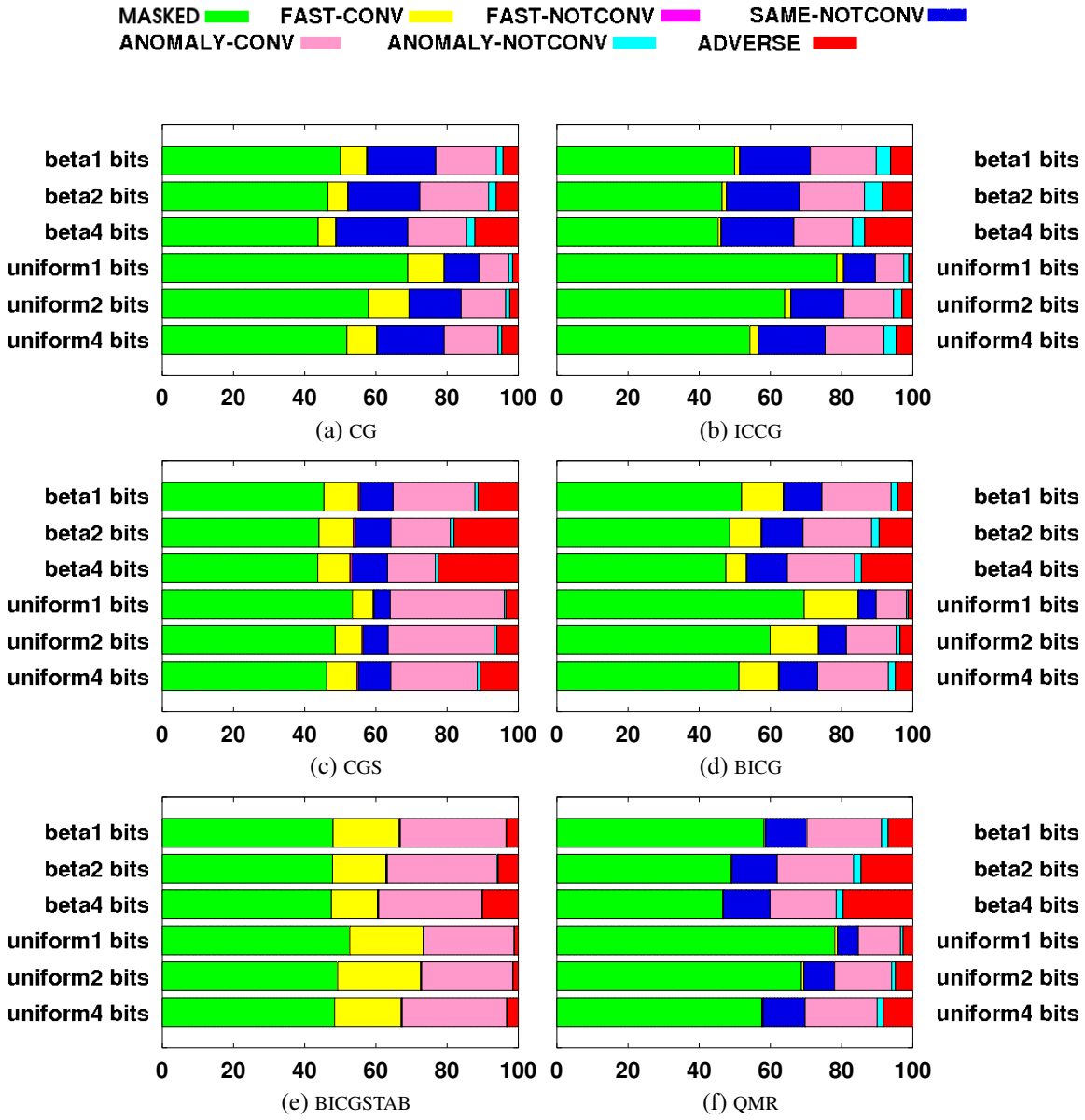
Figure 4.6 presents the distribution of outcomes in terms of the vector impacted by an error. We observe that behavior can be grouped into three broad classes. For all solvers considered (except BiCGSTAB), a significant fraction of errors in the solution vector (x) (almost 80%) can lead to incorrect early termination of the execution. This behavior is unique for the x vector for all solvers. Errors in the second group lead to significant fraction of non-masked outcomes. This group is exemplified by vectors p and r in all solvers, with additional vectors in some solvers. These are vectors that are live for a large fraction of the execution duration. In these categories, we observe that a significant fraction of errors in CG and BiCG lead to faster convergence. In the third class are the short-lived vectors, which are less vulnerable to soft errors. A resilience strategy can selectively focus on the solution vector and the vectors in the second category to maximize coverage at a given cost.

Influence of the statement in which an error is injected.

Figure 4.7 shows the influence of the statement in which an error in injected. In CG and ICCG, we observe a non-uniform distribution of anomalous not converged outcomes. Other than that, we see that solver behavior is not very sensitive to the exact statement being affected by a fault. In particular, CGS and QMR show almost no difference due to the choice of statement.

Impact of bit positions in which an error is injected.

Figure 4.8 plots the weighted outcomes of the error-injection experiments in terms of the location of the error within the double-precision number: sign-bit, exponent, or mantissa.



(a) CG (b) ICCG

(c) CGS (d) BICG

(e) BICGSTAB (f) QMR

Fig. 4.5 Solver behavior for different error injection scenarios. Expected population mean for each outcome are within (0.9%, 18.3%, 3.0%, 1.9%, 8.3%, 3.6%) of our sample mean at the maximum point with a confidence level of 95%.

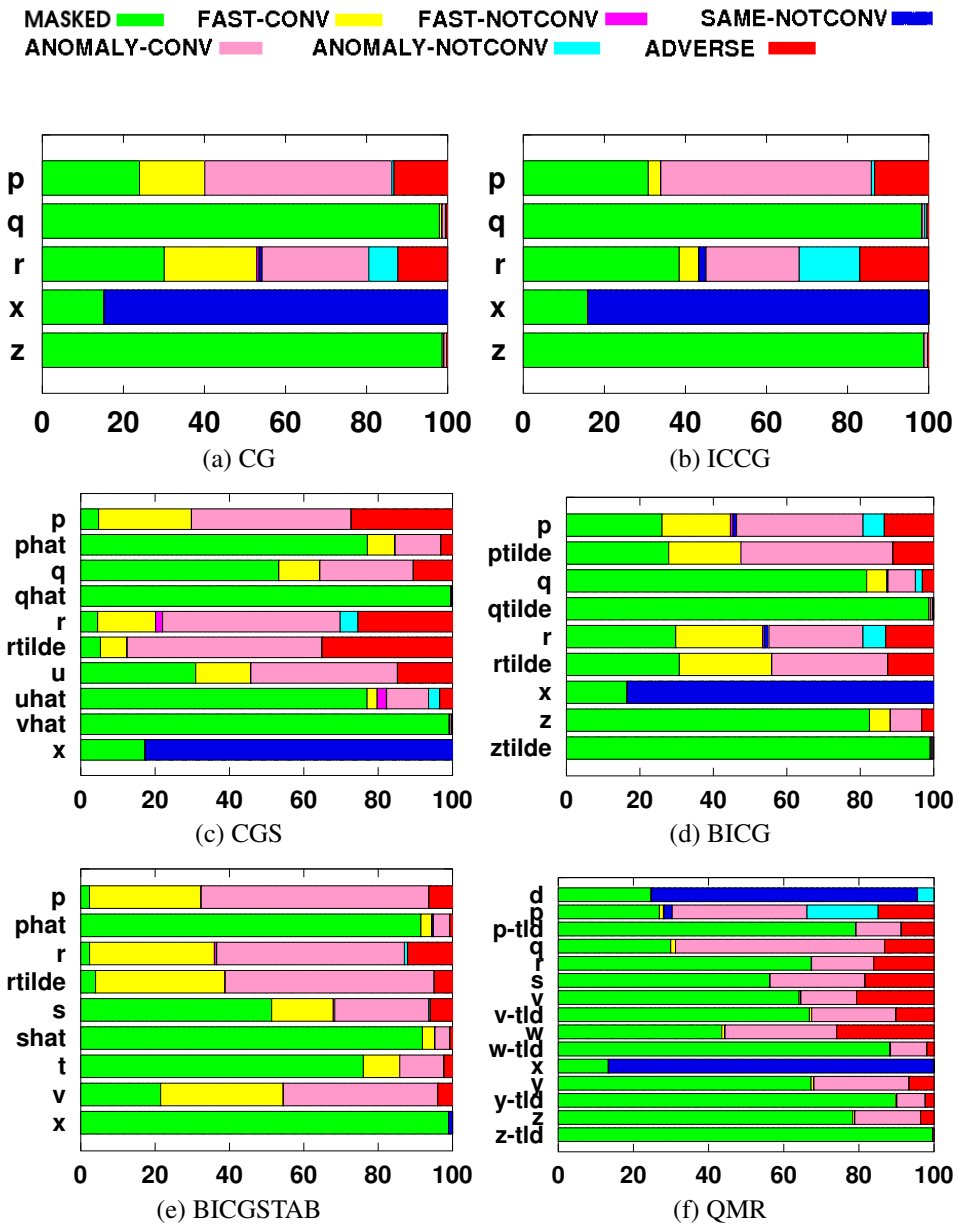


Fig. 4.6 Solver behavior when different vectors are injected with an error. Expected population mean for each outcome are within (2.0%, 14.8%, 64.0%, 3.7%, 112.3%, 29.8%) of our sample mean at the maximum point with a confidence level of 95%.

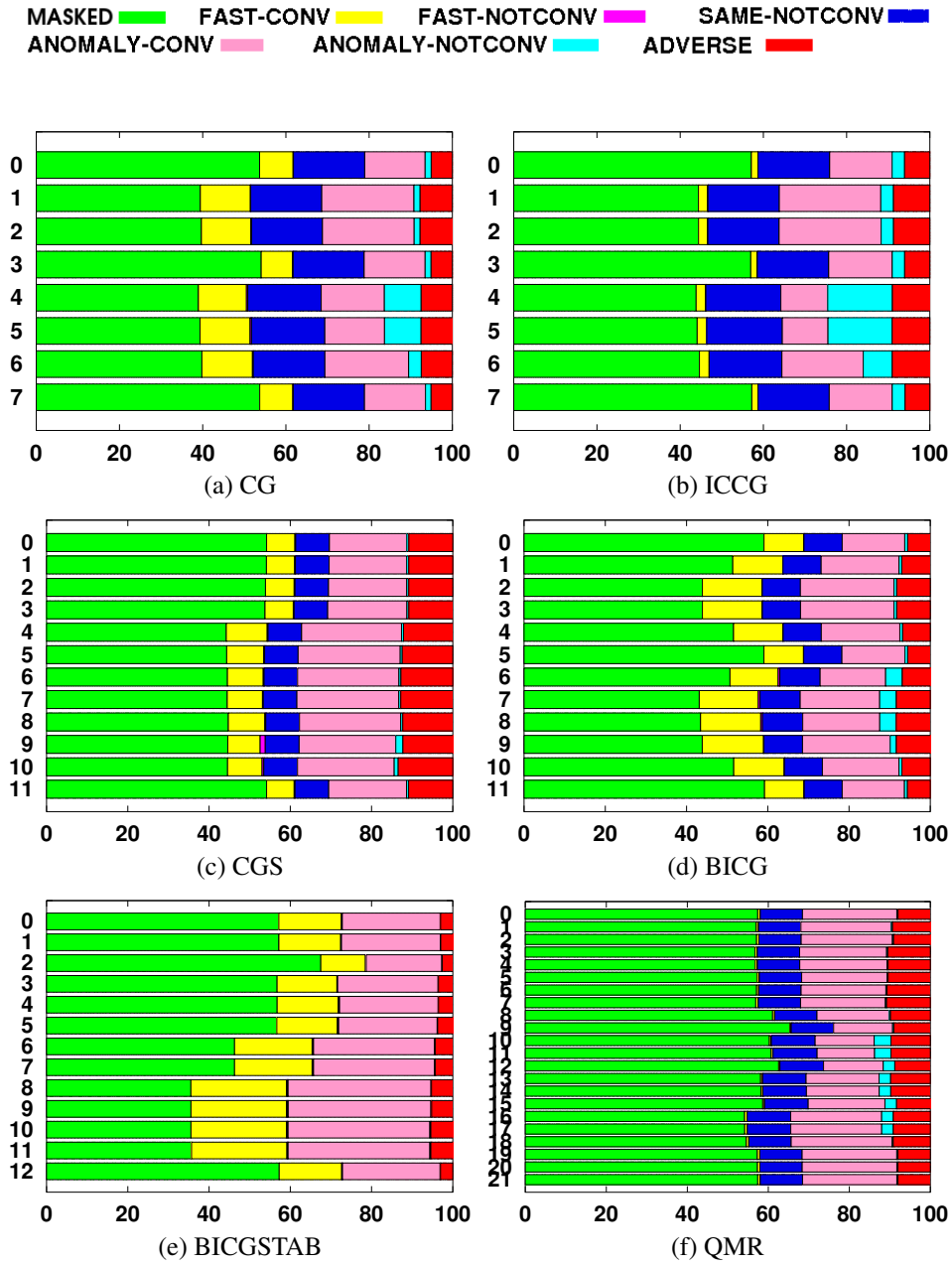


Fig. 4.7 Solver behavior when error is injected at different statements within the algorithm. Expected population mean for each outcome are within (1.3%, 11.0%, 3.8%, 2.2%, 14.7%, 2.7%) of our sample mean at the maximum point with a confidence level of 95%.

Because multi-bit errors can affect more than one location, we plot only the outcomes of single-bit error injections.

For all solvers considered, adverse outcomes are mostly restricted to errors affecting the exponent. Across all solvers considered, single-bit errors affecting the mantissa do not lead to any observable fraction of outcomes being adverse. A small but noticeable fraction of sign errors results in adverse outcomes. This suggests that protecting the exponent and, possibly, the sign, are more important than protecting the mantissa. Given that the mantissa covers the largest fraction of a double-precision number's storage, selective protection for the sign and exponent location might catch the most impactful soft errors.

To further analyze the significance of bit position within the double-precision number, we also plot the behavior for when each bit within the exponent is hit with an error. Figure 4.9 plots the weighted outcomes of the error-injection experiments in terms of the location of the error within the exponent of the double precision number. We observe that, 62^{th} bit, being the most significant bit, is always important. An error hitting the 62^{th} bit is highly likely to lead to adverse outcomes. Another observation we gather is generally, higher bits (56 - 62) are more important and the possibility of leading to SDCs multiplies after the 56^{th} bit. This behavior is also observed in literature [51].

We see that BiCG and BiCGSTAB are less precise and less sensitive to the errors in the exponent. CGS and QMR solvers again show higher bits in the exponent are more important for the calculations, with QMR having around 3% - 4% variations between adverse outcomes at the higher bits, whereas CGS showing a more regular distribution of the adverse outcomes on higher bits. We also observe CG and ICCG showing a higher sensitivity on 56^{th} bit. It shows that these solvers could benefit from stronger software based protection.

Error-injection at different points in the execution

Figure 4.10 shows the distribution of outcomes depending on the iteration in which errors are injected. The number of iterations executed in a run depends on the data set and the solver. As with analysis of the impact of vector position, we normalize the number of iterations across runs. In particular, for each injection run with solver s and data set d, we compute the ratio of the iteration in which the error was injected and the number of iterations executed by solver s on data set d in the absence of errors. Just as in the case of vector position, this ratio is binned into 20 bins (0–5%, 5–10%, etc.). We see that the iteration point affected has a greater influence that the vector position, with greater probability that an error later in the execution will be masked. Except towards the end of execution, CGS and BiCGSTAB seem least influenced by the iteration in which the error is injected.

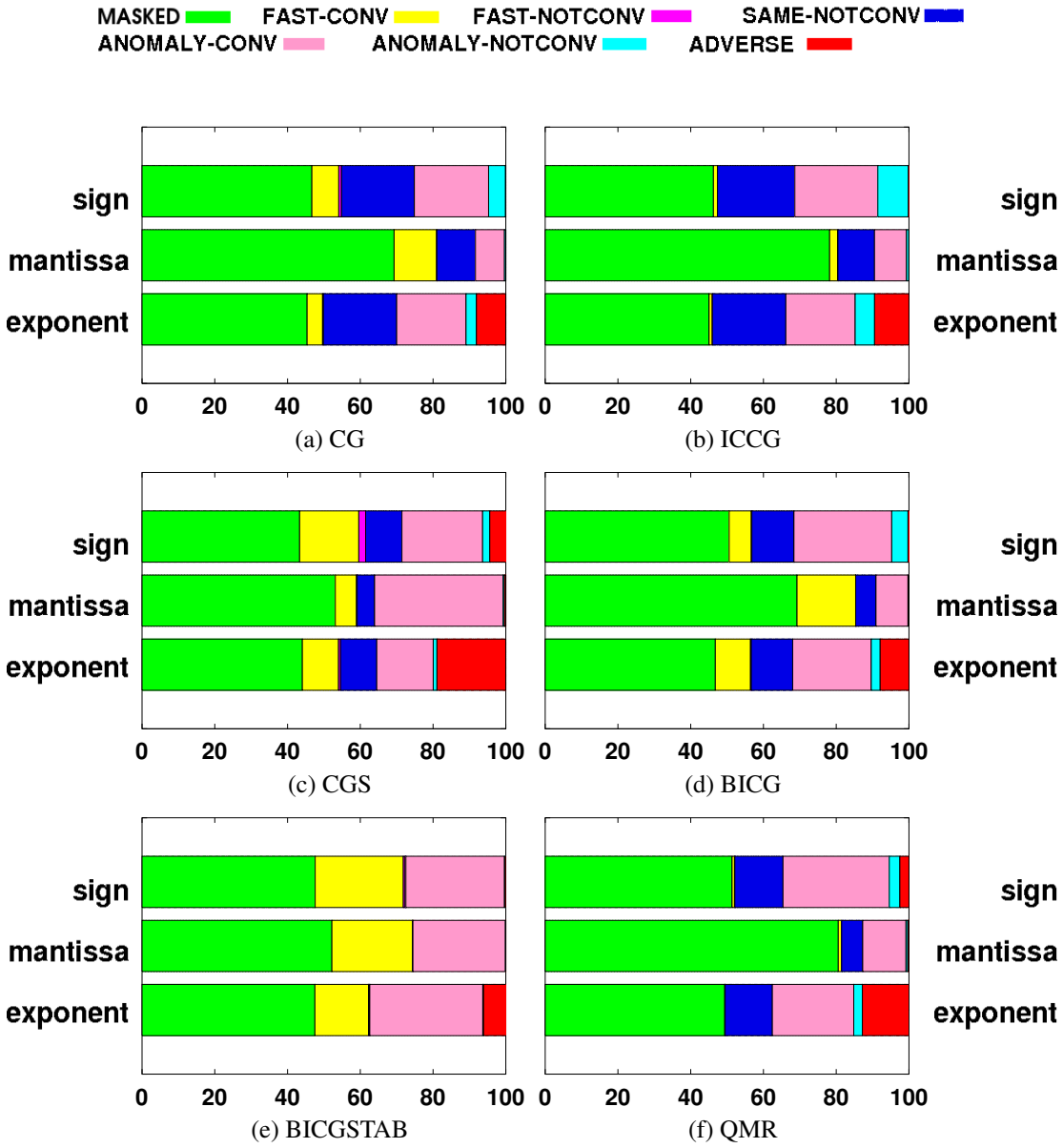
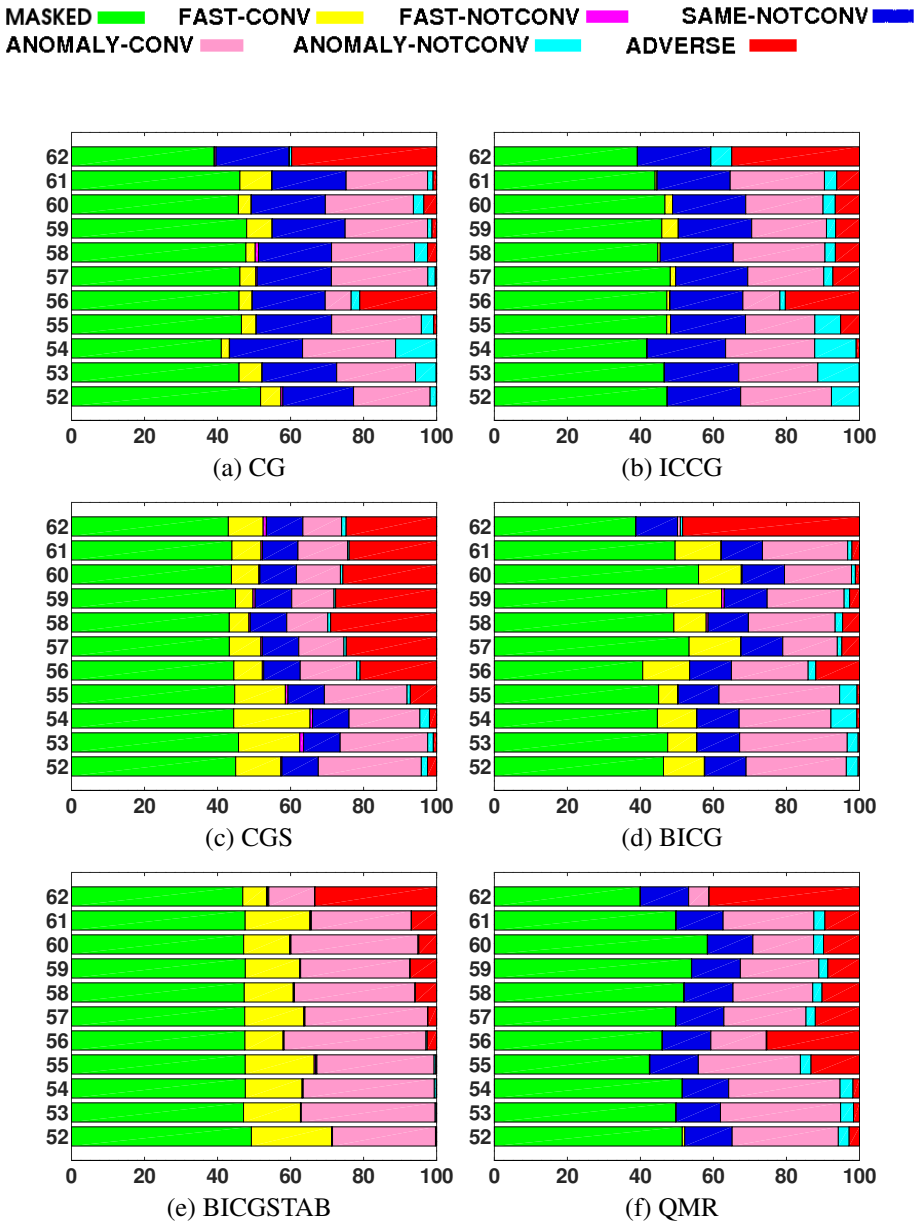


Fig. 4.8 Weighted solver behavior when a single bit error is injected at different points of the variable, results are combined from uniform and beta error distributions. Expected population mean for each outcome are within (9.1%, 21.3%, 8.7%, 9.9%, 14.0%, 13.1%) of our sample mean at the maximum point with a confidence level of 95%.

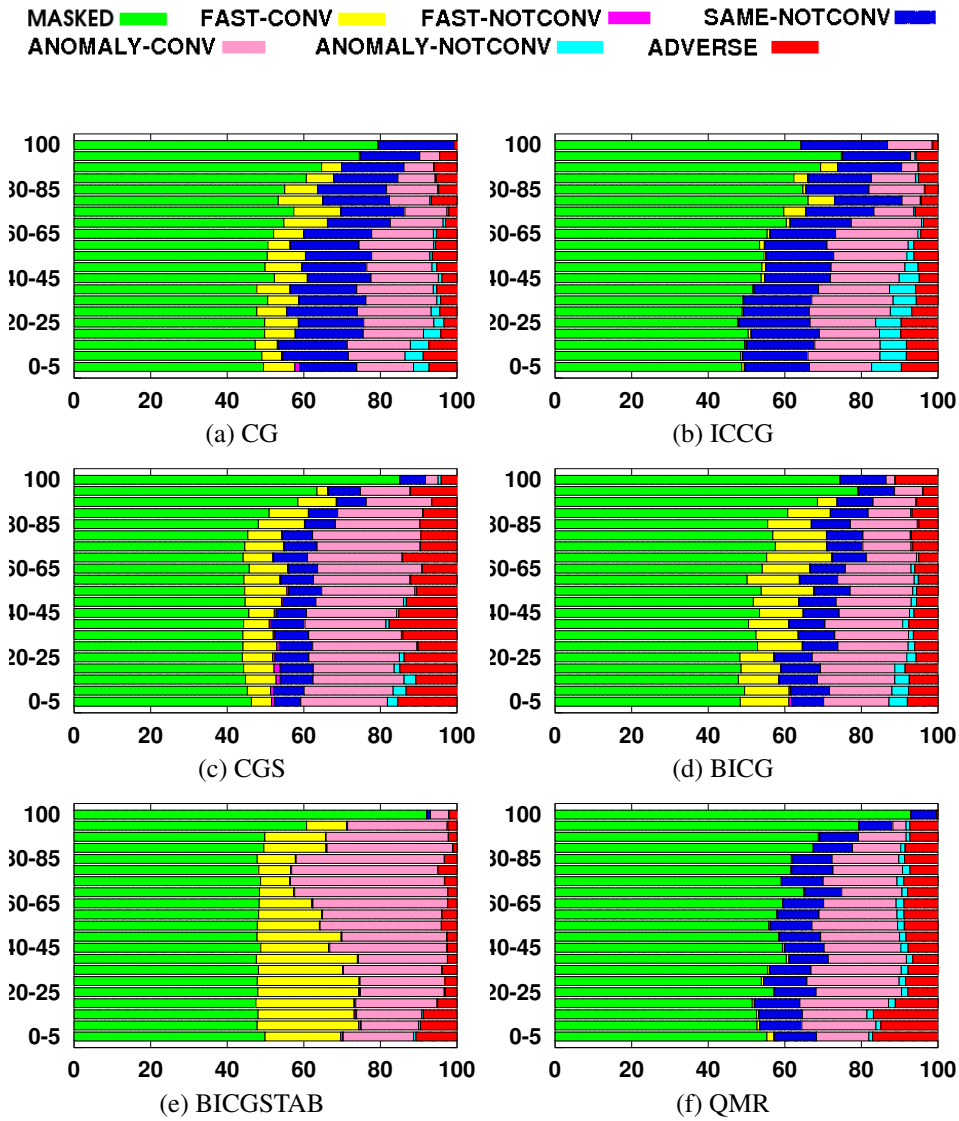


(a) CG (b) ICCG

(c) CGS (d) BICG

(e) BICGSTAB (f) QMR

Fig. 4.9 Weighted solver behavior when a single bit error is injected at different points of the exponent, results are combined from uniform and beta error distributions.



(a) CG (b) ICCG

(c) CGS (d) BICG

(e) BICGSTAB (f) QMR

Fig. 4.10 Solver behavior when injections made at different points of the execution in terms of iteration. Expected population mean for each outcome are within (2.4%, 10.9%, 4.7%, 3.4%, 9.2%, 3.8%) of our sample mean at the maximum point with a confidence level of 95%.

Iterative method	Statements involving matrix-vector operations
CG	0, 3
ICCG	0, 3
CGS	3, 4, 7, 9
BiCG	0, 1, 5, 6
BiCGSTAB	2, 3, 7, 8
QMR	5, 6, 9, 12, 14, 15

Table 4.3 Statement numbers for matrix-vector operations in each iterative method.

4.5 Posterior Probability Analysis

We analyze the outcome distributions obtained from the error injection experiments to understand the relative susceptibility of different statements and vectors in iterative methods. Specifically, we apply the Bayes' theorem to compute the probability of an error being in a particular statement or vector, given an anomalous or adverse outcome:

$$P(S|O) = \frac{P(O|S) * P(S)}{P(O)}$$

where $P(S|O)$ is the posterior probability of a statement being affected by an error (hypothesis) given a specific outcome (evidence or observation); $P(O|S)$ is the likelihood or probability of observing an outcome O given an error affects statement S; $P(S)$ is the prior or probability of an error affecting statement S; and $P(O)$ is the marginal likelihood of a given outcome O. Similarly, we compute $P(A|O)$, where A is the vector impacted by an error.

Figures 4.11 present $P(S|O)$ information for each iterative method. These graphs plot statement probabilities given there is anomalous (both converged and not converged), or adverse outcomes. We observe that statements performing matrix-vector operations consume a significantly larger fraction of the total time than other operations. Table 4.3 lists the statement numbers corresponding to matrix-vector operations for each iterative method. In general, we observe that matrix-vector operations are the most likely to be affected by an error, given anomalous or adverse outcomes. Intuitively, this implies that any protection scheme should focus on these operations. Interestingly, BiCGSTAB does not exhibit this behavior. In the case of BiCGSTAB, given an anomalous or adverse outcome, matrix-vector operations are no more likely than other operations to be affected by error. Therefore, for BiCGSTAB, protection schemes need to consider many more statements.

Figures 4.12 give us the posterior probabilities of error happening in a certain vector given the outcome is anomalous and adverse. In general, the specific vector being affected, with an anomalous or adverse outcome, depends on the solver. However, in many cases, vectors p

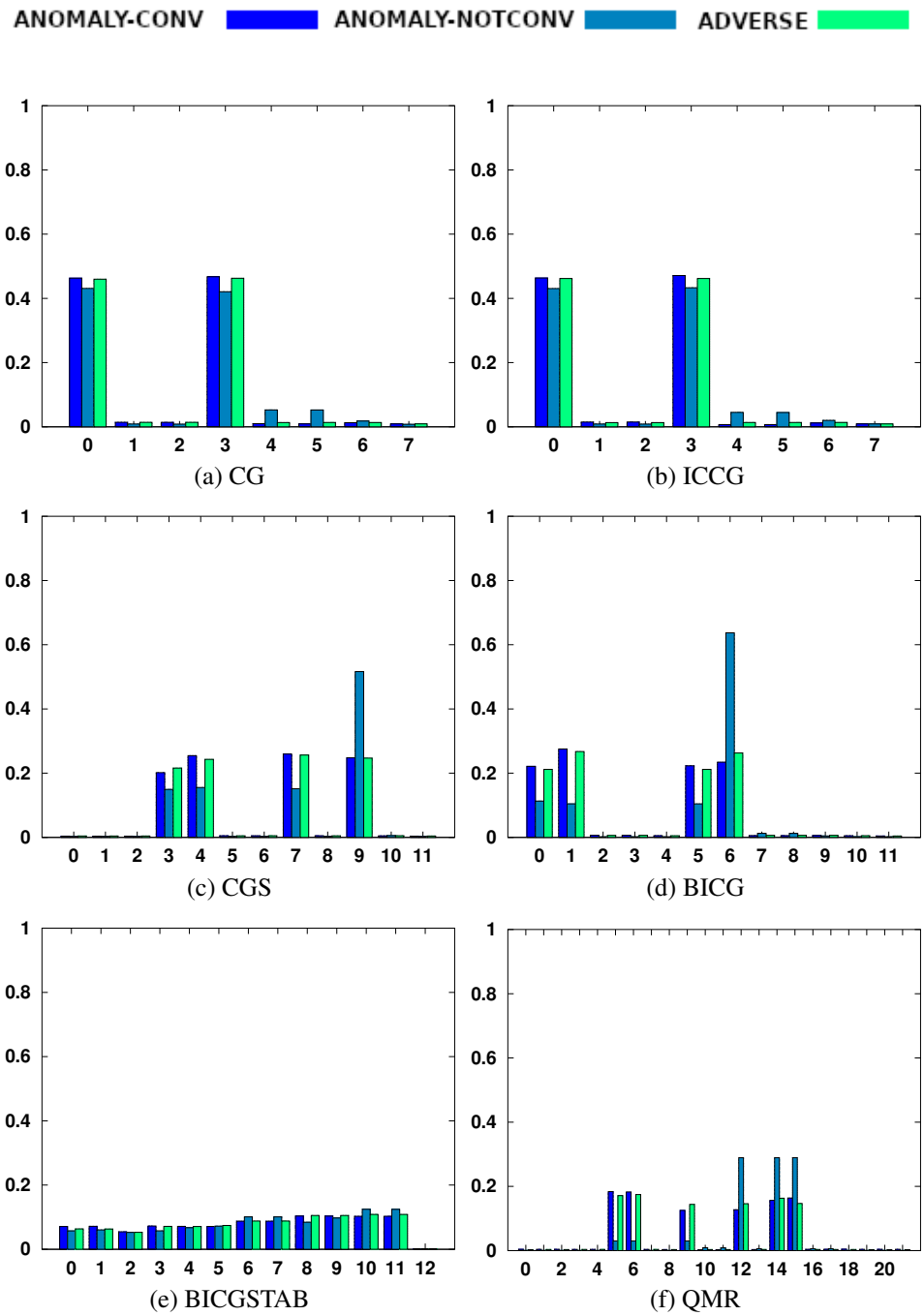


Fig. 4.11 Probability of an error affecting a statement given the outcome is anomalous or adverse, P(Statement|Outcome)={Anomaly-Conv or Anomaly-NotConv or Adverse}

and r are more susceptible than others. Especially if we focus on anomaly not-converged and adverse outcomes, which are the most harmful of all 7 outcome categories being considered, vector r (residual) is the most susceptible in all solvers (with the exception of QMR). Thus, it is the best candidate to be protected against silent data corruption. In the case of QMR, around 80% of the anomalous non-converging outcomes stem from an error occurring at the vector p, whereas adverse outcomes are more or less balanced among the different vectors. In all cases, we observe that vector x (from $A \cdot x = b$) is the most resilient, and errors affecting x do not yield harmful outcomes as much as errors affecting other vectors.

4.6 Summary Of Observations

To the best of our knowledge, this is the first comprehensive characterization of the behavior of iterative methods in the presence of soft errors. In addition to enabling a concrete understanding of soft error behavior for an important application class, this analysis has the potential to aid the design of soft error detectors and realistic evaluation strategies. Summarizing the data presented in the chapter, we observe the following:

- Error-induced solver behavior varies widely. In particular, CGS behaves differently from the other solvers considered.
- The comparative behavior of the solvers varies widely with the selection of data sets. Therefore, a large number of data sets should be chosen for meaningful analysis.
- As shown by CG versus ICCG, a change in the preconditioner can have a noticeable impact on the error behavior.
- Not all vectors are equally impacted by soft errors. In many cases, the solution vector x behaves noticeably differently.
- All solvers except CGS are more resilient to errors in the mantissa than other portions of the floating-point number.

Given the diverse characteristics of iterative solvers, soft error detection mechanisms need to be evaluated with multiple solvers. In addition, all potential injection sites must be accounted for in evaluating the usefulness of a soft error detection mechanism, not just a chosen subset of interest. In addition, due to the wide variations in error-induced behavior of solvers with changes in data sets, the effectiveness of any error detection strategy should be evaluated in the context of the application of interest.

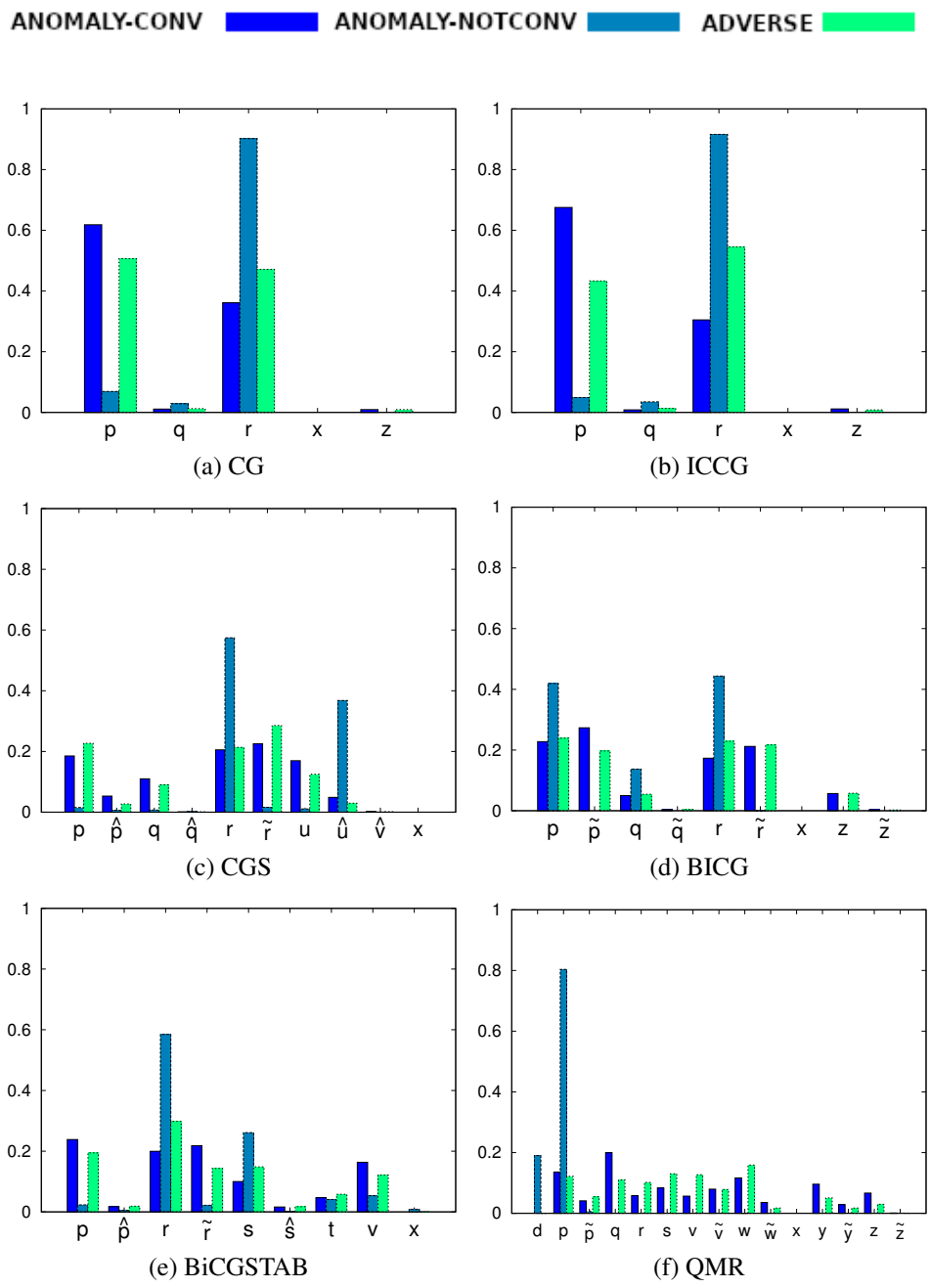


Fig. 4.12 Probability of an error affecting a certain vector given the outcome is anomalous or adverse, P(Vector|Outcome)={Anomaly-Conv or Anomaly-NotConv or Adverse}

4.7 Discussion: Using the Characterization Data

In this chapter, we have focused on detailing our extensive fault-injection experiments and summarizing our observations. Developing strategies to exploit this data to build resilience solutions is not this chapter's focus. In this section, we summarize some ways in which the presented data might be used.

Soft-error detector design An ideal soft-error detection strategy will detect and report all types of errors (high coverage), the moment the error impacts the application (low detection latency), with no performance penalty (low performance overhead). But for the most-trivial applications, such detectors do not exist. The best detector depends on the use case. This has motivated the design of multiple types of detectors for different classes of programs. Considering iterative solvers, the data in this chapter was used to comparatively evaluate four detection strategies: adaptive impact-driven detection (AID) [23], Orthonormality based detection [13], checksum-based detection [76], and moving average detector [52]. In addition, the characterization data was used to train machine-learning based detector that combined the features of these individual detectors [46] (Chapters 5 & 6).

Comparative solver evaluation The numerical characteristics of solvers have been extensively studied. In particular, choosing the right solver and preconditioner for a given problem can have a dramatic impact on the time to solution. However, the relative resilience behavior of solvers is not well understood or characterized. The overall resilience of a solver depends on the time it takes to solve a system of equations and its vulnerability to errors. While a fast invulnerable solver is desired, the relative performance and vulnerability of different solvers needs to be quantified to evaluate their overall performance in the presence of errors.

Selective resilience strategies With a detailed understanding of the impact of errors on a solver's runtime behavior, one can design tailored resilience solutions. For example, we observe that sign and exponent are most responsible for the adverse outcomes (Figure 4.10). Strategies to detect the most egregious changes in these parts of a floating point number might incur less overhead than techniques that detect errors affecting the mantissa. As another use case, we observe that vectors p and r are responsible for the most adverse outcomes in the CG solver (Figure 4.6). Studying the CG algorithm (Figure 4.1), we observe that both vectors are updated using vector operations, which are much cheaper than matrix-vector multiplication. Going further, no reads or writes to r involve matrix-vector multiplication. Therefore, r can be cheaply protected while almost halving the number of adverse outcomes for CG.

Design-space exploration for energy efficiency In trying to improve energy efficiency, various forms of less-than-exact execution have been considered. One such option is the use of unreliable memory (e.g., [11]). Data structures which, when affected by an error, have a lower impact on the application might be beneficially placed on such less reliable memory. In Figure 4.6, we clearly see that some vectors have negligible impact on application correctness with respect to soft errors (e.g., z and q for CG/ICCG, *qhat* and *vhat* for CGS, and $qtilde$ and $ztilde$ for BiCG). In addition to performing such static mapping of vectors to memory regions, the temporal information in the characterization data can be used to query for change in vulnerability as the execution progress. This can be used to device dynamic remapping strategies.

In general, we believe the data is a useful resource to quickly test hypothesis relating to the runtime behavior of iterative solvers, develop new solutions that exploit the observed behavior, and comparatively evaluate different strategies.

4.8 IMIC Database

As discussed in the previous section, the data can be leveraged for further anaylsis and can facilitate future studies. Therefore, at the end of this study, we created a publicly available database called Iterative Method Injection Collection at https://github.com/pnnl/IMIC [58]. This database has all 1.75 million injection results with traces collected during this study. By doing this, our aim was to make this vast data available for other scientists to apply their own analysis without having to run exhaustive injection campaigns.

We collected about 2.5TB of data from the fault-injection experiments. This data included the injection characteristics (location, time, number of bits, etc.), convergence result (Masked, Anomaly, etc.), as well as other monitored values such as the norm of the residual vector over the course of the execution. To ease public access, we extracted the injection and convergence information from all the fault injection experiments and made it available in a table-like format.

What's more, to enable reproducibility of our experiments, we also provided an installer for the experimental setup. We provided links to the libraries used and explained the modifications applied to generate the data.

Some approaches that can be applied using this database can be listed as follows;

- For the purpose of this study, we employed simplified weights, especially for preconditioners, to the statement costs. This can be tuned for specific architecture and software stack through trial executions to obtain exact statement execution costs.

- The analysis presented averages (arithmetic mean) of the error-injection experiments from different datasets by taking into account the statement execution costs. Depending on the target workload, the analysis across the data sets can be performed in different ways.

In general, we believe the data is a useful resource to quickly test hypothesis relating to the runtime behavior of iterative solvers, develop new solutions that exploit the observed behavior, and comparatively evaluate different strategies. This database is available to add to the knowledge and improve the field by facilitating further research by scientists around the world.

4.9 Conclusions

In this chapter, we presented a comprehensive characterization of the iterative method behavior under soft errors. We considered 6 solvers, 28 datasets, and multiple fault injection scenarios. We believe this data is a useful resource that can aid in testing runtime behavior of iterative solvers, comparative solver evaluation, error detection studies, and design space exploration. As an exemplar case study for using this data, we provided a joint work with another group at Pacific Northwest National Laboratory in the Appendix B. Results from this study is in the process of being submitted.

We employ a deterministic error injection strategy to systematically explore the space of possible error behaviors. We consider 1, 2, and 4 bit error injections under uniform and beta distribution of the bit positions affected by the error. We consider all statements and vectors in the iterative methods as candidates for error injection. To reduce fault injection overheads, we identify and prune error injections that will lead to masked errors and those that will lead to the same outcomes as other error injection configurations.

We analyzed the data to identify differences in soft-error-induced behavior stemming from the choice of data sets, choice of position and number of bits affected, the statement and vector affected, and the point in an execution time when an error is injected. In sum, we performed a total of 1,744,800 error injection runs and collected more than 2.5TB data which is made into the IMIC database, which is publicly available at https://github.com/pnnl/IMIC.

Chapter 5

Detector Characterization

5.1 Introduction

The challenges that stem from soft errors on iterative solvers motivate the design of soft error detectors that can detect the adverse impact of soft errors in a timely fashion. To mitigate the adverse impact of soft errors, techniques have been designed to efficiently and accurately detect the presence of soft errors and recover from them. These detectors employ a variety of techniques (curve fitting, machine learning, algorithm analysis, etc.) to flag observed behavior that deviates from predicted correct behavior as a potential error. These detectors have been developed and evaluated in diverse contexts, making a comparative analysis of their effectiveness difficult. In this chapter, we present a comparative evaluation of four state-of-the-art online soft error detectors in the context of iterative methods through extensive single-bit and multi-bit fault injection experiments. We track the evolution of the residual vector for fault-free method execution, fault injection experiments, and detector characterizations experiments, totaling several million runs. This should enable systematic design and evaluation of new detectors for iterative methods.

We performed extensive fault injection experiments involving 28 data sets, five iterative methods, single- and multi-bit errors, and uniform and normal error distributions, totaling over 1.4 million fault-injection experiments. All the detectors were evaluated using identical fault-injection configuration, enabling a direct and unbiased comparison of their behavior.

Given the extent of the fault-injection analysis involved and the burden on computing time, we performed this study on sequential implementations of the iterative methods. We verified that these methods are implemented using an abstraction (matrices, vectors, and operations on them) similar to PETSc [4]. For example, our implementation of New-Sum [76], one of the detectors considered, is operation-for-operation same as the one by New-Sum's authors in PETSc. The primary difference between sequential and parallel runs is the potential

differences in ordering of floating-point arithmetic. While rigorous analysis of the inter-play between soft errors and finite-precision arithmetic on realistic data sets is beyond the scope of this work, the lessons from the comparative analysis presented in this chapter is applicable to parallel execution of these methods.

The primary contributions of this chapter include:

- Extensive fault-injection evaluation of iterative solvers,
- Use of the fault-injection experiments to comparatively evaluate the four state-of-the-art soft-error detectors,

A soft error detector periodically observes the execution of an application to identify whether it is in a soft-error induced incorrect state. Not all soft errors lead to an adverse outcome. In particular, iterative methods can inherently mask some errors. Detecting such errors can lead to unnecessary re-execution. Therefore, we classify the impact of a given fault injection as leading to an adverse or benign outcome by executing the error-impacted run to completion.

Based on this classification of a fault-injection run, a detector's decision is characterized as:

True Positive (TP) The detector correctly labels an execution as erroneous.

False Positive (FP) The detector labels an execution as erroneous but the error was masked, i.e., execution completed with the correct results in the same number of iterations as an execution without error injection.

True Negative (TN) The detector correctly labels an execution as correct.

False Negative (FN) The detector labels an execution as correct but the execution is corrupted.

A detector is considered *precise* if it can correctly identify all the SDC cases (true positives) and only the SDC cases (no false positives). We also define *recall* as the fraction of SDC detected over all the cases in which an SDC occurred.

More formally:

$$Precision = \frac{TP}{TP + FP} \tag{5.1}$$

$$Recall = \frac{TP}{TP + FN} \tag{5.2}$$

The precision of a detector defines how accurately a detector can identify an SDC, i.e., a large number of false positives decreases the detector precision. Recall, instead, defines what fraction of SDC has been detected, i.e., the number of SDC detected over all SDC that should have been detected.

When designing a fault tolerant or resilient solution, it is important to minimize the number of false negatives, i.e., SDC that should have been detected but that pass unnoticed. False negatives, in fact, may hide incorrect results and lead scientists in the wrong direction, delay the execution of the application (longer time to converge), or induce application crashes. We thus consider recall as the primary metric when evaluating a detector for a certain solver. If two detectors present similar recall, we then consider their precision. A high precision implies that the number of unnecessary rollbacks is low (few false positives), thus there is no waste of energy or additional overhead when executing the solver. We remark that rolling back an application because of an incorrect labeling of an iteration (false positive) increases the execution overhead of the application but does not impact the application correctness. On the other hand, not detecting an SDC (false negative) might severely impact the correctness of the application.

Until the iteration at which an error is injected, the execution is considered fault-free. We distinguish these iterations from iterations past the fault-injection. In particular, we employ False Positive Rate (FP^*) [23, 73], which evaluates the number of false positive iterations when running a detector on fault-free execution of each solver. Ideally, a detector should not report any errors for these iterations. Note that this metric is distinct from, and reports on a disjoint set of iterations/executions than, the false positives due to a detector reporting errors that are masked. Separating these out helps us distinguish detector behavior for error-free versus masked runs.

5.2 Experiment Setup and Error Model

As we discussed in previous chapters, iterative methods solve a system of linear equations $A \cdot \vec{x} = \vec{b}$, where A is a sparse matrix, $\vec{b}$ and $\vec{x}$ are vectors by iteratively computing increasingly accurate approximations to the exact solution. At each iteration, the methods maintain a residual corresponding to the computation $\vec{r} = \vec{b} - A \cdot \vec{x}$. Iterations complete when the norm of the residual vector is below a user-specified threshold. Here is a list of iterative methods we target in this work.

- CG: Symmetric positive-definite, Conjugate gradient with incomplete LU preconditioner

- ICCG: Symmetric positive-definite, CG with incomplete cholesky preconditioner
- BiCG: Need not be self-adjoint, Unpreconditioned biconjugate gradient
- BiCGSTAB: Non-symmetric input, Biconjugate gradient stabilized
- CGS: Non-symmetric input, Conjugate gradient squared

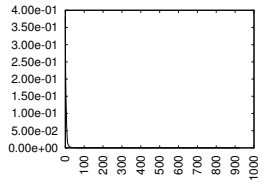
Related methods were chosen to evaluate the variations in their behavior to across similar methods. CG and ICCG differ in the preconditioner used. They require their input to be a positive definite symmetric matrix. BiCG is a generalization of CG and can handle non-symmetric and non-definite systems. However, it requires a multiplication of A with its conjugate transpose which makes BiCG numerically less stable. The BiCGSTAB is a more stable version of BiCG but involves greater computational cost per iteration. CGS has the same computational cost as BiCG but does not require the transpose of A.

The dataset matrices are used to initialize A. The vector $\vec{b}$, given as input to the method, is computed by initializing the solution vector to all ones and computing $\vec{b} = A \cdot \vec{x}$. This ensures that the system of equations has a valid solution. All computation is performed in double precision. We used the solver implementations in the Iterative Methods Library (IML++) v1.2a [24], compiled using GCC 4.7 with `-O2` optimization.

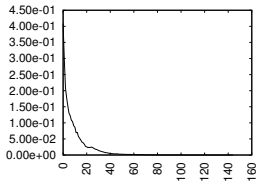
We select 28 symmetric positive definite matrices, which can be handled by all the methods considered, from the University of Florida Sparse Matrix database [21]. List of datasets and their properties are listed in Table 3.1.

While some of the data sets are relatively small, they represent realistic data sets from diverse application domains. The number of iterations performed by iterative methods ranges from 33 to 16226, depending on the solver and the dataset.

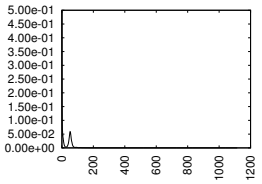
Application detectors attempt to identify anomalies during the execution of a solver that could be due to soft errors. Many of such detectors observe specific values and track their evolution during the execution. The observed trends are then compared to expected correct behavior and an error is flagged whenever the observed behavior differ from the expected one. In the case of the iterative methods, the residual error is often used as the observed value. In general, detection approaches perform better when the evolution has a discernible trend and gradually evolves. Effective design of this class of detectors can be aided by information on the convergence characteristics of iterative methods.



(a) BiCG: 5*,6,7,8,9,10,11,12, 18,21,23,26,24,27,28

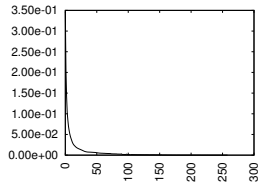


(b) BiCG: 15*,17,20,16,13, 19,22,25

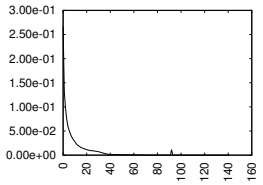


(c) BiCG: 2*,1,3,4,14

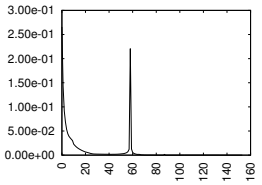
(d) BiCG: None



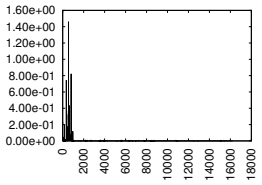
(e) BiCGSTAB: 17*,1,2,3,4,5, 6,8,9,14,15,23,24,26,27



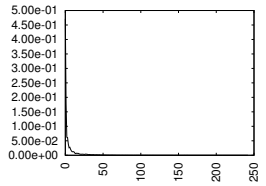
(f) BiCGSTAB: 10*,16,19,21,22



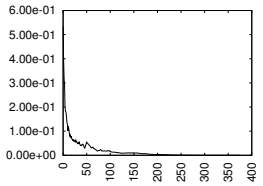
(g) BiCGSTAB: 13*,7,12,25,28



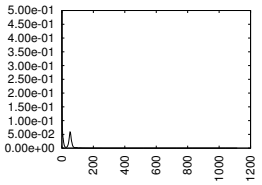
(h) BiCGSTAB: 11*,20



(i) CG: 21*,5,6,7,8,9,10,11, 12,15,18,23,24,26,27,28

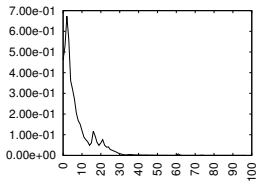


(j) CG: 17*,20,16,13,22,25



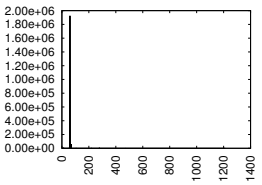
(k) CG: 1*,2,3,4,14,19

(l) CG: None

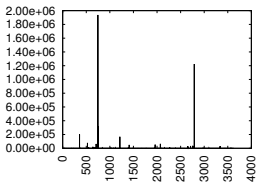


(m) CGS: None

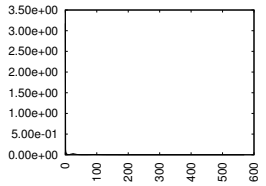
(n) CGS: 19*,6



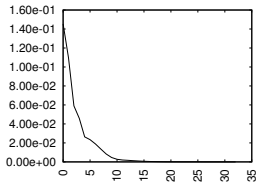
(o) CGS: 1*,2,8,16,25,27



(p) CGS: 13*,3,4,5,7,9,10,11, 12,13,14,17,18,20,21, 22,23,24,26,28



(q) ICCG: 4*,1,2,3,5,6,7,9,11,12, 16,18,23,24,26,27,28



(r) ICCG: 15*,8,10,13,14,17, 19,20,21,22,25

(s) ICCG: None

(t) ICCG: None

Fig. 5.1 Representative graphs of the evolution of residual norm with iteration count. Each graph plots residual norm evolution for one method on one data set (highlighted by a * in the caption). The other data sets that exhibit similar trends are listed in the caption. x-axis: iteration count; y-axis: norm of residual vector.

5.3 Convergence Characteristics

While the general convergence characteristics of methods are known, their behavior on specific data sets cannot be easily determined. Therefore, we evaluated the convergence characteristics of the five methods on the 28 data sets in terms of the evolution of the residual norm as the iterations progress. An execution converges when the residual error falls below 10^{-6}. Figure 5.1 groups the data sets based on their convergence trends, showing one representative graph for each group. Note that the grouping is presented for brevity and visual clarity to identify the key distinguishing features. Where possible, we have tried to group data sets within similar trends across methods in the same column. In general, while the residual error might monotonically decrease over a window of iterations, the trends show significant diversity. In the first group (first column in the figure), we observe a quick and sharp reduction in the residual errors, followed by steady reduction in residual error over several iterations. In the second group, the residual error decreases more gradually but exhibits similar trends as the first group. The third group shows scenarios with an occasional spike in the residual error. ICCG did not exhibit this trend. The final column shows trends with multiple spikes in the residual error evolution.

In summary, we observe that the evolution of residual errors shows significant variation even for the same method. Not all the solver-dataset pairs exhibit monotonic trends in the residual, which may lead to SDCs passing unnoticed. Identifying data set characteristics that lead to different convergence trends can aid the design of accurate detectors.

5.4 Soft Error Detection

In this section, we describe and evaluate the effectiveness of state-of-the-art error detectors while running the solvers and data sets described in the previous section.

5.4.1 State-of-the-art Soft Error Detectors

Adaptive impact-driven detection (AID) Di and Cappello [23] observed that the impact of "influential" soft errors can be characterized by an *impact error bound*, defined as the maximum ratio of the data value change between adjacent time steps to the global value range for every data point in a snapshot. This observation is used to dynamically fit different curves—last-state, linear, and quadratic—to the temporal evolution of data based on their prediction error. An error is flagged if the observed value falls out of the impact error bound range.

Orthonormality detector (Ortho) Chen et al. [13] identify an intrinsic orthogonal relationship of specific vectors in several Krylov linear solvers. They construct a detector that periodically checks this relationship and flags an error if the orthogonality condition is violated (within a certain tolerance). Among the solvers considered, the orthogonality relationship only exists for CG, ICCG, and BiCG.

Checksums for matrix-vector multiplication (New-Sum) Tao et al. [76] presented a checksum encoding scheme for matrix-vector multiplication and vector linear operations. The checksums are maintained by augmenting each solver operation with an efficient checksum operation. The separate checksum is recomputed from the current matrices periodically and compared to the one computed at each iteration step. The detector flags an error if the two checksum differ.

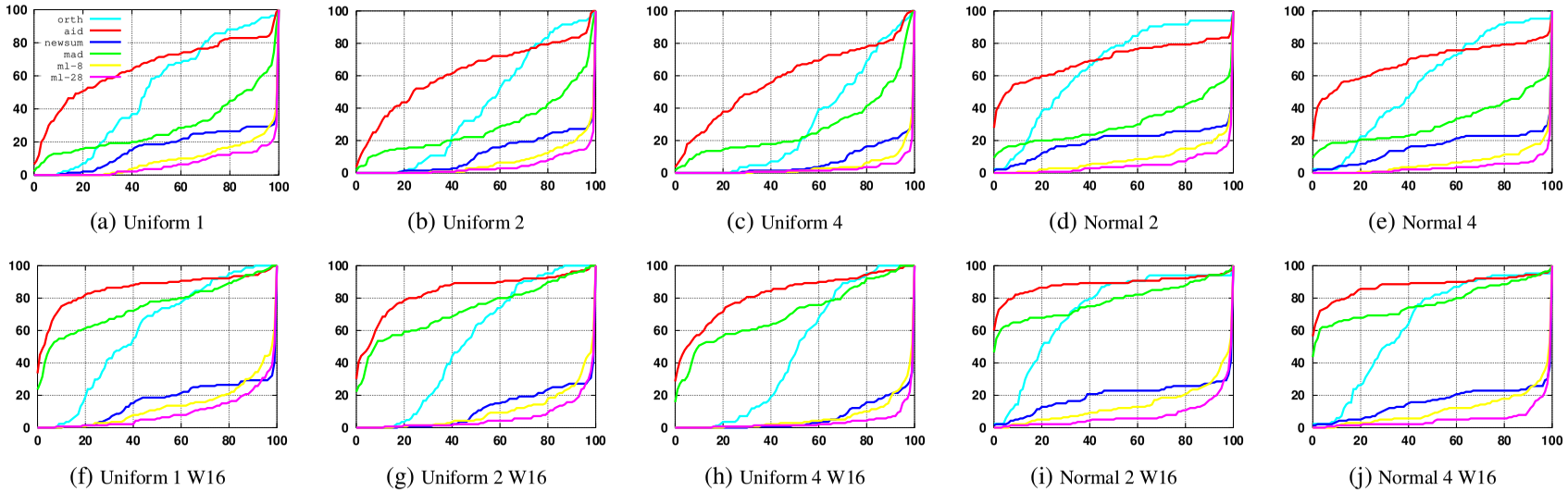
Moving average detector (MAD) Liu et al. [52] observe that the residual norms in iterative methods might not strictly decrease at every iteration but that the norms exhibit a decreasing trend over a longer period (multiple consecutive iterations). Rather than focusing on one iteration at the time, the authors employ a moving average of the residual norms over a sliding window. The detector flags an error if the moving average increases with respect to the previous window rather than decreasing, as expected.

To avoid over-reacting to minor perturbations, detectors also require the usage of a threshold before flagging an error. In this work, we use the default threshold values for each detector: 0.00078125 for *AID*, 10^{-10} for *Ortho* and *New-Sum*, and 0.1 for *MAD*.

It is important to remark that an error occurring during the execution might not necessarily affect the application's final results nor the convergence time. In fact, errors might be masked, for example because they impact a bit in a floating point value with low significance (thus, they are absorbed by the natural errors in floating point computation). Labeling an application run as incorrect because of an error detected in one iteration might be overly pessimistic and induce a high number of false positives, hence trigger unnecessarily recovery actions. To reduce the number of false positives, a more focused detector may flag an error only after it has observed a consistent trend (i.e., residual norm increasing over several consecutive iterations). We define an *adverse outcome* as an execution in which the number of iterations required to converge in the presence of errors is different from the number of iterations necessary to converge in an error-free execution.

Quantifying the effectiveness of a detector is not easy, as several non-functional parameters are involved. In this work we consider :

- accuracy in detecting adverse outcomes,



(a) Uniform 1 (b) Uniform 2 (c) Uniform 4 (d) Normal 2 (e) Normal 4

(f) Uniform 1 W16 (g) Uniform 2 W16 (h) Uniform 4 W16 (i) Normal 2 W16 (j) Normal 4 W16

Fig. 5.2 Cumulative function for recall for uniform and normal distribution with 1-bit, 2-bit, and 4-bit errors (denoted "Uniform 1", etc.) with immediate detection and with a detection window of 16 iterations (denoted W16). The x-axis represents the recall percentage, while the y-axis represents the percentage of the solver-dataset pairs. For an ideal detector, we expect recall values of 1, which corresponds to curves that are flat (value zero) for x<100% and then jumps to 100% for x=100%.

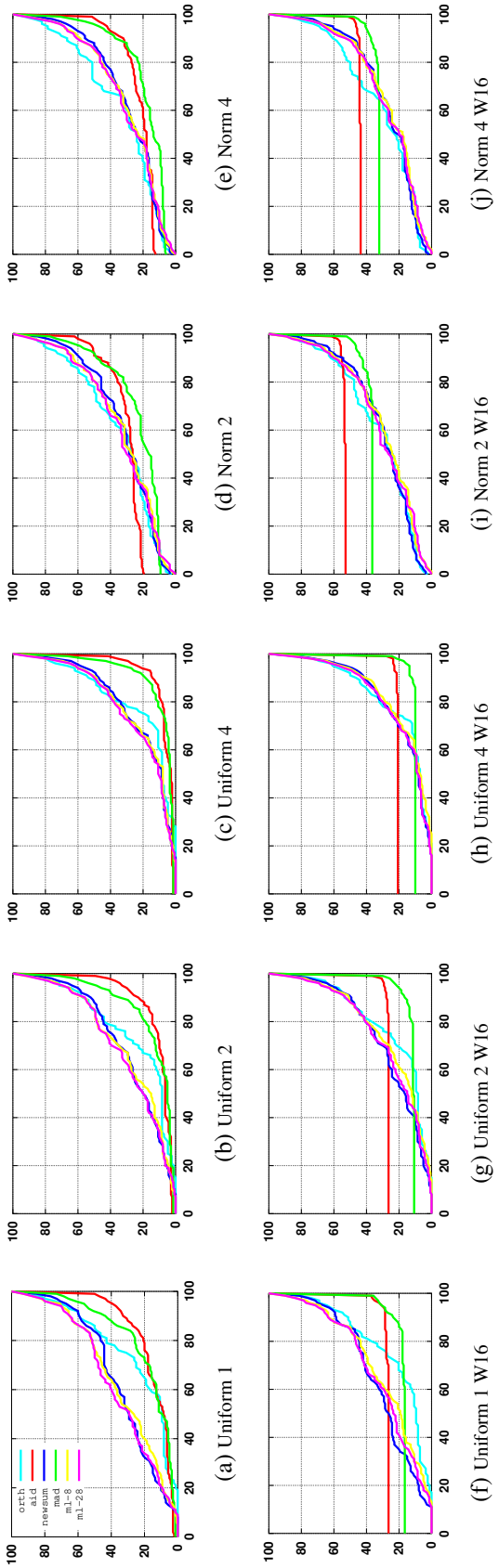
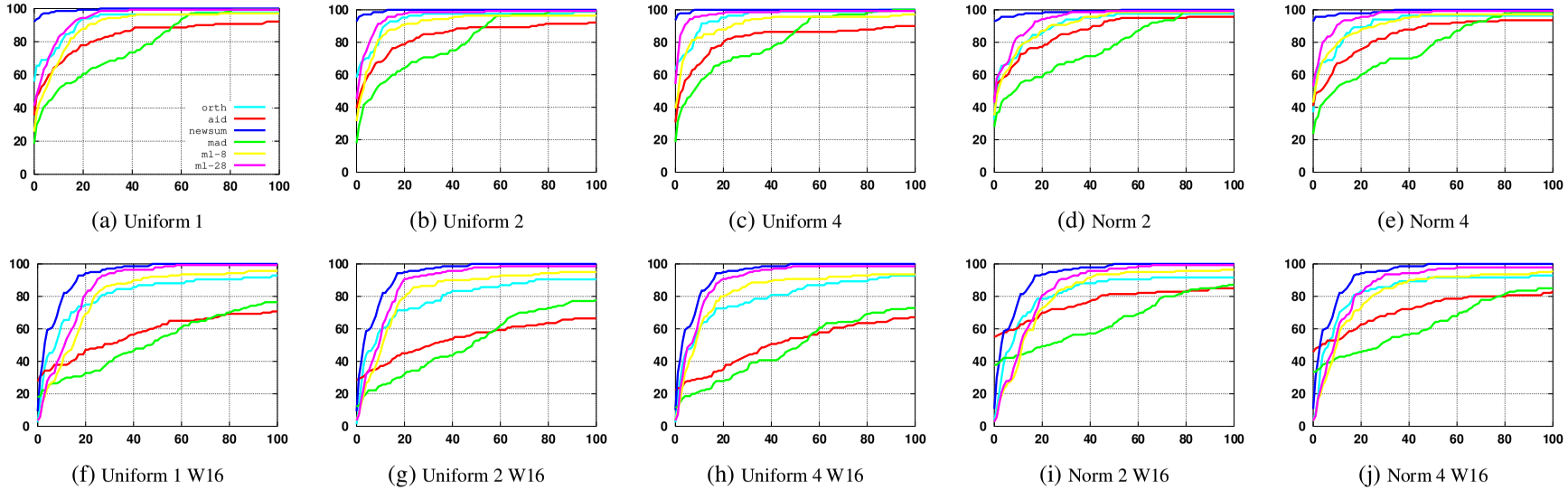


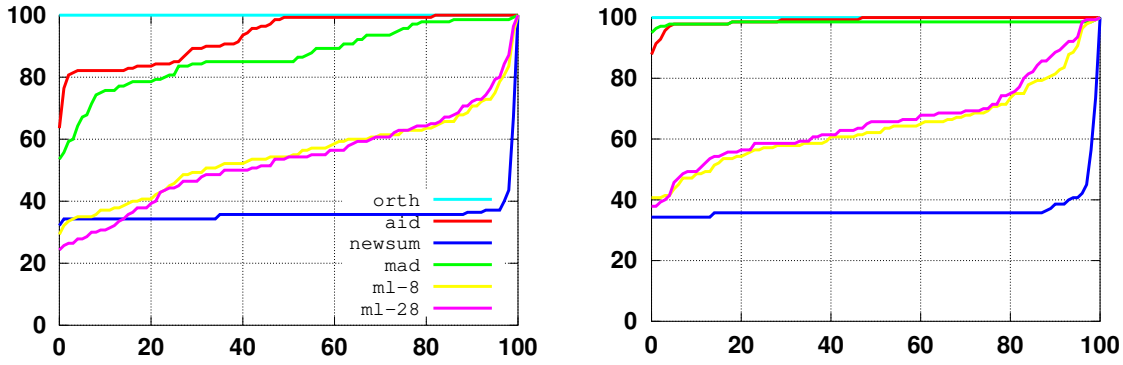
Fig. 5.3 Cumulative function for precision for uniform and normal distribution with 1-bit, 2-bit, and 4-bit errors (denoted "Uniform 1", etc.) with immediate detection and with a detection window of 16 iterations (denoted W16). The x-axis represents the precision percentage, while the y-axis represents the percentage of the solver-dataset pairs. For an ideal detector, we expect precision values of 1, which corresponds to curves that are flat (value zero) for x$<$100% and then jumps to 100% for x=100%.



(a) Uniform 1 (b) Uniform 2 (c) Uniform 4 (d) Norm 2 (e) Norm 4

(f) Uniform 1 W16 (g) Uniform 2 W16 (h) Uniform 4 W16 (i) Norm 2 W16 (j) Norm 4 W16

Fig. 5.4 Cumulative function for detection latency for uniform and normal distribution with 1-bit, 2-bit, and 4-bit errors (denoted "Uniform 1", etc.) with immediate detection and with a detection window of 16 iterations (denoted W16). The x-axis represents the detection latency as a percentage of number of iterations in an error-free execution, while the y-axis represents the percentage of the solver-dataset pairs. For an ideal detector, we expect the latency to be close to 0, corresponding to a flat line at y=100% from x =0 till x=100. Note that, in the presence of errors, the number of iterations executed and thus the detection latency can be greater than iterations executed without errors. While we only show the graphs with detection latencies of $\leq$100% for brevity, we do observe such long detection latencies (especially in the W16 case).



(a) Window 1 (b) Window 16

Fig. 5.5 Cumulative function for false positive rates. The x-axis represents the percentage of FP rates, while the y-axis represents the percentage of the solver-dataset pairs examined. An ideal detector will show a straight line at y=100% for all values of x>0.

- timeliness of detection,
- detection overhead and its impact on performance,
- generality across solvers and data sets.

An ideal detector is one that, for all solvers and data sets, detects all and only adverse outcomes (high precision and recall), detects an error immediately after it appears, and does not increase overall execution time. We provide a detailed analysis of the tested detectors in the next sections.

5.4.2 Detector Accuracy

Ortho detector exploits mathematical properties of the solvers and can, in principle, detect every injected error. Barring the handling of the preconditioner, this same property holds for *New-Sum*. However, this guarantee is provided in the context of solvers operating on real numbers. The use of finite-precision arithmetic leads to dramatic deviations from this anticipated behavior. This motivates the need for detector analysis using realistic data sets in the application domain of interest. We evaluate each detector in terms of precision, recall, and false positive rate, as defined in Section 5.1.

Our deterministic error injection strategy allows us to repeat any experiment X multiple times, every time injecting an error in the same iteration, residual vector location, and bits. Using this methodology we can perform an apple-to-apple comparison among all the detectors and compare their decisions (i.e., prediction of the final outcome) to the final result of the

application. Consider an experiment X^* in which we do not run any detector and inject an error in a certain residual vector element at a given iteration. We can deterministically inject the same error(s) in an equivalent experiment X_d, with $d \in \{AID, NewSum, MAD, Ortho\}$, where we run detector d. We then analyze whether the decision taken by detector d in the experiment X_d is consistent with the outcome of experiment X^*. We label the detector decision as true positive if d identifies an error in X_d and predict an adverse outcome that is observed in X^*. A false positive occurs when d detects an error and predicts an adverse outcome in X_d but we do not observe the adverse outcome in X^*. A true negative is a case in which neither X_d nor X^* show an adverse outcome. Finally, a false negative occur when d misses an adverse outcome that is observed in X^*.

Additionally, we investigate the false positive rate (FP-rate). We compare the detectors' decisions to the error-free executions of the solvers. In this scenario, we expect that the detectors will not identify any errors and report false positives in case they predict an adverse outcome. The FP-rate is computed as the number of false positives over the total number of iterations.

As discussed above, flagging an execution as incorrect when an error is detected might be overly pessimistic. An error detected in one iteration might be masked in later iterations and still result in a final correct outcome. To mitigate the impact of false positives we adopted two strategies for evaluating the detector decisions.

Point detection (W1): This is the basic detection method. If a detector observes an error at iteration i, we stop the execution and label the experiment as producing an adverse outcome, regardless of whether or not the error will be masked in later iterations.

Sliding window detection (W16): In this method, we label an experiment as producing an adverse outcome if a detector observes an error for n consecutive iterations. The reasoning behind this methodology is that if an error has not been masked and has been observed for n consecutive iterations, there is a high probability that it will never be masked and will result in an adverse outcome. Conversely, if the error observed at iteration i is masked at iteration $i+1$, the detector will not flag an error, hence reducing the number of false positives and increasing the confidence in the results. For this study we analyzed windows of size 2, 8 and 16 but, due to space limitations, we report the results for size 16, which has the lowest false positive rates.

Figure 5.2, 5.3 and 5.5 plot the cumulative distribution functions for the recall, precision, and false positive rates for the tested detectors, respectively. Each sub figure presents a single metric, window size, and error distribution tuple. The y-axis at each graph represents the

percentage of the solver-dataset pairs examined. The x-axis represent the percentage of recall, precision, and false positive rate.

Each point (x,y) on a plot line represents the percentage of the data points (y) that has a lower or equal performance value than (x). For an ideal detector, we expect recall and precision values of 1, which corresponds to curves that are flat at the value zero for x$<$100 and then jumps to 100 for x=100. For FP-rate on the other hand, an ideal detector will show a straight line at y=100 for all values of x. For example, Figure 5.5 shows that *Ortho* detector has a perfect 0 false positive rate, which means it never flags an error for error-free execution.

The graphs in Figure 5.2 and 5.3 show that each detector has its own pros and cons. We observe that *New-Sum* has the highest recall value and that *MAD* with W16 and *Ortho* have good false positive rates (close to zero for all of the data points). However, their recall plots show that they miss a considerable number of errors. *AID* shows low false positive rates with the sliding window detection method (W16), though its recall values are also low. Point detection method for *New-Sum* and *MAD* have promising performance values, though *New-Sum* shows the worst false positive rates among all the experiments.

5.4.3 Detection Latency and Overhead

Detecting adverse outcomes early is almost as important as detecting them correctly. For the cases in which the detectors correctly flag an error, Figure 5.4 shows the latency values for each detector, window size, and injection method. Latency values are computed as the distance (in terms of number of iterations) between the injection of the error and its detection. To get a latency percentage we divide this distance by the number of iterations executed in error-free runs.

We plot the cumulative distribution for latency values, where y axis represents the percentage of solver-dataset pairs and x axis shows the latency percentage. Similar to false positive rate, a perfect latency plot would be the line jumping to y=100 at x=1 and goes a straight line at the y=100 line. Some detectors never reached to 100%. These detectors identify the error after the total error-free iterations. For example, for point prediction and with 1 bit flip uniformly distributed, *AID* shows more than 100% latency for about 10% of the solver-dataset pairs on average. These detectors present a “long tail” latency distribution with a non-trivial fraction of errors detected thousands of iterations after the error is injected.

We observe that, irrespective of the error schemes and window sizes, *MAD* and *AID* have high latency values compared to the other detectors. *New-Sum* has slightly better latency performance than the rest, which are performing in a similar fashion to each other. With a window size 16, we observe that all detectors other than *AID* and *MAD* have similar latencies.

	Mean				Stdev				min				max			
	AID	NSM	MAD	ORTH	AID	NSM	MAD	ORTH	AID	NSM	MAD	ORTH	AID	NSM	MAD	ORTH
CG	1.4	3.1	1.0	2.7	0.5	1.7	0.0	0.3	1.0	1.1	0.9	2.3	3.2	5.4	1.1	2.8
ICCG	1.1	1.8	1.0	1.8	0.2	0.8	0.0	0.0	1.0	1.0	1.0	1.7	1.8	2.9	1.1	1.9
BiCG	1.1	1.6	1.0	1.5	0.1	0.5	0.0	0.0	1.0	1.0	1.0	1.4	1.5	2.3	1.0	1.5
BiCGST	1.1	1.8	1.0	NA	0.1	0.9	0.0	NA	1.0	1.1	1.0	NA	1.4	3.1	1.2	NA
CGS	1.3	2.1	1.2	NA	0.2	0.9	0.1	NA	1.0	1.1	0.9	NA	1.9	3.2	1.3	NA

Table 5.1 Mean, standard deviation, min, and max slowdown due to detectors as compared to baseline execution. Average over 10 runs.

Table 5.1 shows the detector overhead (geometric mean, minimum, and maximum) as the ratio of the execution time with and without detector instrumentation. *AID* and *MAD* present the lowest overheads. Among the solvers, CG has the lowest cost per iteration, thus the relative impact of the detectors is higher. Note that, in our evaluation, the detectors are used every iteration, presenting a worst-case usage pattern in terms of overheads. In practice, detector overhead is amortized by checking only every several iterations.

The runtime overhead can be divided in two phases: the first part is the extraction of the data to be used in the detection. *AID* and *MAD* use already available variables from the execution and have negligible extraction costs. *Ortho* and *New-Sum* extracts orthogonal relationship of specific vectors and checksum of matrices, respectively, which lead to the larger portion of their overhead. The second part of the overhead comes from the analysis. *AID* and *MAD* have higher analysis costs. *Ortho* and *New-Sum*, on the other hand, only requires a conditional check to decide on the error.

5.5 Conclusions

In this chapter, a comprehensive evaluation of the behavior of soft error detectors were presented. We consider five iterative methods, 28 data sets, and multiple fault-injection scenarios. We evaluated flagging an error based on detector behavior at a single iteration or over a sliding window of iterations. While each detector considered has been shown to be effective in a distinct context, extensive analysis of various configurations evaluated demonstrates that, in the context of iterative methods, they do not achieve perfect detection accuracy. Given the high false positive rates, which can lead to a large re-execution overhead, existing detection techniques might better serve as a component of a larger detection system. In addition, the detection latencies, in many cases, can be a substantial fraction of the total execution time.

We observe that each detector achieves a desirable performance, however none of them were perfect or close to. We theorized there is room to improve and explored a machine learning feature combination avenue to analyze this potential (Chapter 6).

Another observation from Chapter 4 and Chapter 5 is that fault injection experiments for error behavior profiling and detector performance analysis is costly. We devised a machine learning based prediction approach to tackle this problem, which we discussed at Chapter 7.

Chapter 6

Machine Learning Based Error Detection

6.1 Introduction

While individual detectors achieve high detection rates, none of the detectors considered in Chapter 5 achieve perfect detection. To understand the potential for improved detectors, we design an online detector based on offline *machine learning* methodology.

We present a machine learning based detector using the features of individual detectors to identify the potential for an improved accuracy based on these features. As discussed in the previous chapter, no detector satisfies all the requirements of an ideal detector (Section 5.4), but each detector has its own pros and cons. A natural approach towards an ideal detector consists of coalescing the information used by the four detectors into a new error detector. We explored this avenue but quickly realized that the space resulting from the fault injection campaign was too large to be analyzed manually. Instead, we opted for using machine learning approaches. We extracted application-independent features from the data collected during the fault injection campaign, namely, residual norm (*all detectors*, one feature), checksum values computed from the current matrices (*New-Sum*, three features), and orthogonality relationships (*Ortho*, two features). We then train the network using different machine learning algorithms and build models that predict the occurrence of adverse outcomes based on observations during the execution. Finally, we evaluate the resulting machine learning-based error detector on a disjoint testing set and compare it to state-of-the-art detectors.

While generating the dataset sets for the machine learning algorithms, we follow the following steps:

1) We randomly select 8, 16, and 28 (all) data sets and only collect features from the experiments belonging to those selected datasets using the normal error distribution.

2) We shuffle the resulting training set to remove any bias due to the ordering of fault injection experiments.

3) For the cases in which we use only a subset of the solver's data sets (8 and 16), we use the remaining data sets to build the testing set. For the case in which we use all solver's data set as training set, we split the shuffled data by randomly selecting 90% of the samples for training and 10% for testing sets.

4) Then, we further prune the training set to select equal number of innocuous and erroneous samples.

We use several supervised learning algorithms to create a model per iterative solver. Supervised learning methodology uses a training set with a label for each sample, either innocuous or erroneous. Each label presents the ground truth determined based on the comparison to the fault-free execution. Our training sets with ground truth are the collection of the fault-injected experiments performed while evaluating the selected state-of-the-art online soft error detectors.

6.2 Supervised Learning Algorithms

We used the following machine learning algorithms to build our error detector. These are the well-established algorithms that cover different classification techniques.
Decision tree (DT): Decision tree is a tree-based model. The leaves represent class labels and the branches represent conjunctions of features that lead to these class labels.
Support vector machines (SVM): SVM is a supervised method that builds a hyperplane between training instances and classifies samples according to their position with respect to that hyperplane. This method has been proved to work well on non-linearly separable training sets and generally shows good precision and recall.
AdaBoost (AB): This an iterative machine learning algorithm that improves its precision by increasing the weights of mis-classified samples. The algorithm uses techniques that combines the outputs of weak learners (arbitrary learning algorithms) and formulates a weighted sum of the outputs as the final output.
Stochastic gradient boosting (SGB): SGB constructs an additive model in a stage-wise fashion. This method effectively combines boosting with gradient descent algorithms.

Random forest (RF): Over-fitting is a common issue for several machine learning algorithms. Randomized forest addresses this issue by constructing a set of decision tree classifiers during the training stage and averaging them.
Extremely Randomized Trees (ET): This algorithm is a slightly different algorithm than the random forest. As in a random forest, a random subset of candidate features is used. However instead of searching for the most discriminative thresholds, thresholds are drawn at random for each candidate feature.
Bootstrap Aggregation Techniques (Bagging): Bagging is an ensemble method that takes a set of classifiers and aggregate their predictions by voting or averaging to form a final decision.
Naive Bayes (NB): This machine learning algorithm is a supervised method based on the Bayes' theorem. The method makes the "naïve" assumption that every feature is independent from the rest of the features.
Multilayer Perceptrons (MLP): MLP is a feedforward neural network algorithm which maps a set of inputs onto the set of corresponding outputs. MLPs consist of multiple layers of nodes in some directed graph. Except for the input nodes, each node is a neuron having a nonlinear activation function. MLPs use back-propagation for training the network.

	8/28		16/28		28/28	
Solver	Alg.	F-Score	Alg.	F-Score	Alg.	F-Score
CG	ET	84.7	Bagging	85.5	ET	90.9
CGS	ET	78.9	ET	77.9	ET	79.8
ICCG	RF	78.3	NB	77.8	ET	80.9
BICG	ET	82.9	Bagging	86.6	ET	87.2
BICGSTA	Bagging	78.5	ET	79.9	ET	85.4

Table 6.1 Best machine learning algorithms and their F-scores for each training set configuration.

Table 6.1 shows the best machine algorithm for each pair solver-training set (8/28, 16/28, and 28/28) and the corresponding F-Score for the relative testing sets. ExtraTrees is generally the best machine learning algorithm across all the solver/training set configurations, especially for the 28/28 case. The F-Score values range from 77.809 to 90.877. We observe that using the training set configuration 28/28 generally produce better models (higher F-Score) while there is not much difference between the 8/28 and the 16/28. Thus, for brevity, we omit the case 16/28 and focus on two approaches: ML-8 and ML-28.

6.2.1 Evaluating Machine Learning-Based Detectors

To evaluate our machine learning-based error detector we follow the same methodology used for the evaluation of the other detectors. The plots in Figure 6.1, 6.2, 6.3, and 6.4 show the recall, precision, latency, and false positive rate for our machine learning detector, respectively. We report results for detectors generated with training set configurations that use 8 and 28 datasets out of the 28 datasets available (yellow and purple line, respectively).

The plots show that the recall values for the machine learning detectors are generally the best among all the detectors. Only *New-Sum* performs similarly with a window of size 16 (W16). The machine learning detectors outperform *MAD* and are significantly better than *Ortho* and *AID*.

In terms of precision (Figure 6.2), the machine learning detectors behave similarly to *New-Sum* and slightly worse than *Ortho*. *MAD* and *AID* show higher precision but might miss some errors (low recall values). These two detectors are more "optimistic" than the others and tend not to flag errors unless they observe strong evidences. This means that the machine learning detectors may trigger some unnecessary recoveries but still the correctness of the applications are not affected. In fact, the high recall values indicate that almost all errors are detected (low false negatives).

The machine learning detectors are also capable of detecting faults very quickly, as the results in Figure 6.3 show. Only *New-Sum* is faster than the machine learning-based detectors, while *Ortho* is generally comparable and *AID* and *MAD* are slower. The overhead costs of the machine learning-based detectors (Table 5.1) consists of extraction of the features used by other detectors and analysis. Because the machine learning approaches use all the features extracted by the other detectors, including New-Sum and Ortho's high extraction costs, it is not surprising to see that the extraction cost is relatively high. The analysis cost, instead, introduces only negligible overhead. On average, the machine learning detector presents overhead close to the sum of New-Sum and Ortho.

Finally, Figure 6.4 shows false positive rates (FP^*). The graphs show that the false positive rates are not as good as *Ortho*, *AID*, or *MAD*, but better than *New-Sum*.

Overall, we conclude that our machine learning detectors are not ideal detectors because they still show a large number of false positives. However, they generally outperform the other individual detectors. The machine learning detectors show better recall values than *Ortho*, *AID*, and *MAD* (in the case of *AID* and *MAD*, the recall values are significantly higher), which indicates that the machine learning approaches do not miss errors. Also, error detection is much faster than *AID*, *MAD*, and *Ortho*. Compared to *New-Sum*, the machine learning detectors show similar recall values and error detection latencies, hence both approaches

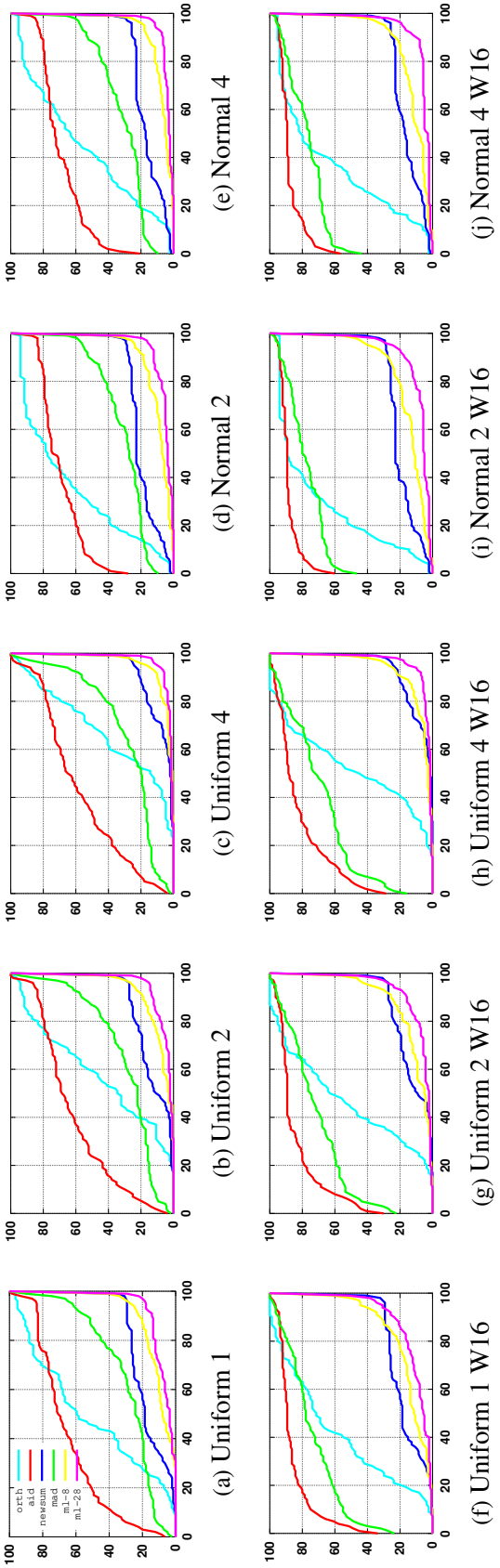
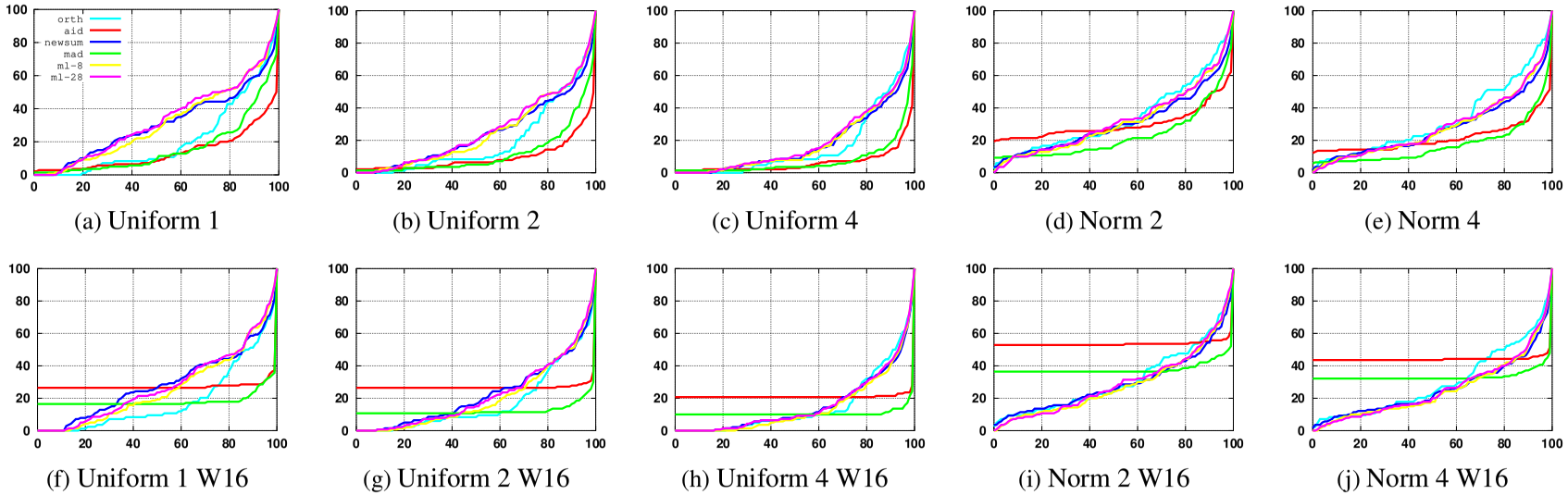


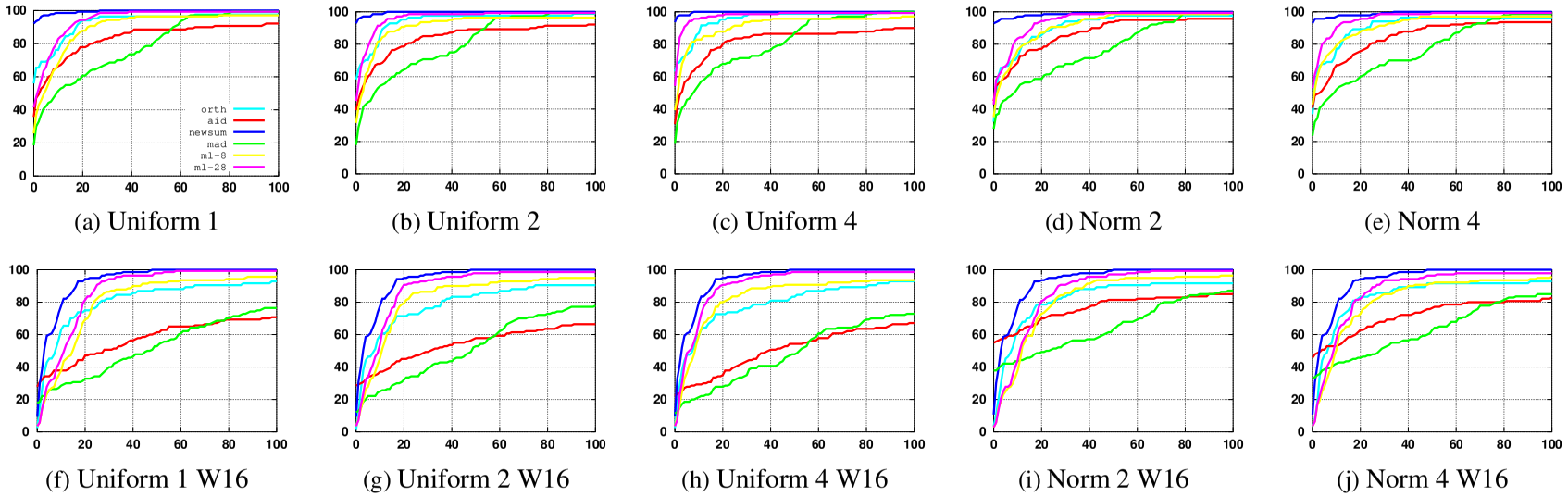
Fig. 6.1 Cumulative function for recall for uniform and normal distribution with 1-bit, 2-bit, and 4-bit errors (denoted "Uniform 1", etc.) with immediate detection and with a detection window of 16 iterations (denoted W16). The x-axis represents the recall percentage, while the y-axis represents the percentage of the solver-dataset pairs. For an ideal detector, we expect recall values of 1, which corresponds to curves that are flat (value zero) for x$<$100% and then jumps to 100% for x=100%.



(a) Uniform 1 (b) Uniform 2 (c) Uniform 4 (d) Norm 2 (e) Norm 4

(f) Uniform 1 W16 (g) Uniform 2 W16 (h) Uniform 4 W16 (i) Norm 2 W16 (j) Norm 4 W16

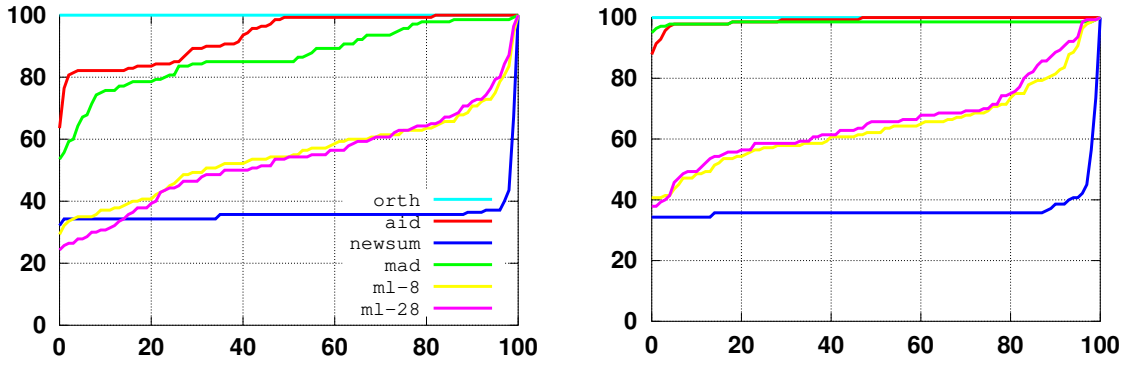
Fig. 6.2 Cumulative function for precision for uniform and normal distribution with 1-bit, 2-bit, and 4-bit errors (denoted "Uniform 1", etc.) with immediate detection and with a detection window of 16 iterations (denoted W16). The x-axis represents the precision percentage, while the y-axis represents the percentage of the solver-dataset pairs. For an ideal detector, we expect precision values of 1, which corresponds to curves that are flat (value zero) for x<100% and then jumps to 100% for x=100%.



(a) Uniform 1 (b) Uniform 2 (c) Uniform 4 (d) Norm 2 (e) Norm 4

(f) Uniform 1 W16 (g) Uniform 2 W16 (h) Uniform 4 W16 (i) Norm 2 W16 (j) Norm 4 W16

Fig. 6.3 Cumulative function for detection latency for uniform and normal distribution with 1-bit, 2-bit, and 4-bit errors (denoted "Uniform 1", etc.) with immediate detection and with a detection window of 16 iterations (denoted W16). The x-axis represents the detection latency as a percentage of number of iterations in an error-free execution, while the y-axis represents the percentage of the solver-dataset pairs. For an ideal detector, we expect the latency to be close to 0, corresponding to a flat line at y=100% from x =0 till x=100. Note that, in the presence of errors, the number of iterations executed and thus the detection latency can be greater than iterations executed without errors. While we only show the graphs with detection latencies of $\leq$100% for brevity, we do observe such long detection latencies (especially in the W16 case).



(a) Window 1 (b) Window 16

Fig. 6.4 Cumulative function for false positive rates. The x-axis represents the percentage of FP rates, while the y-axis represents the percentage of the solver-dataset pairs examined. An ideal detector will show a straight line at y=100% for all values of x>0.

effectively detect most of the errors. However, the number of false positives is lower for the machine learning approaches, thus they trigger fewer unnecessary recoveries than *New-Sum*.

6.3 Conclusions

In this chapter, to identify the potential for an improved accuracy based on the features used by the detectors evaluated, we presented a machine learning based detector using these features. While improved, the machine learning based detector is still far from perfect in terms of its accuracy. We believe, in addition to new methods, additional features need to be incorporated to improve detection accuracy.

We observe after this experiment, using machine learning and combining the powers of the detectors is an area that can offer more performance. First, we can get the decisions of the detectors for the same execution and train another model with the actual effect of the soft error. This model would work with the decisions of the individual detectors and decide on a final outcome.

On the other hand, for this study, we haven't considered spatial or temporal aspects of the error. We can analyze the injections and see if one detector works comparatively better on the earlier executions versus later ones, or if a detector has comparatively superior performance while checking some data structures but not as good working for others. Analyzing the injections and detector performance deeper will give more insight on the detectors success and will open new avenues for having detectors work together. Exploration of these ideas are discussed on Chapter 8.

Chapter 7

Soft Error Prediction

7.1 Introduction

Understanding the impact of soft errors on applications can be costly. Often, it requires an extensive error injection campaign involving numerous runs of the full application in the presence of errors.

In this chapter, 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 (See Chapter 4). This has motivated the continued development of novel strategies that observe the execution progress of iterative methods to identify anomalies in their execution (Chapters 5 & 6).

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 are 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 [21]. The primary contributions of this chapter 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
- An analysis of the cost savings for error injection campaigns in terms of number of iterations of the iterative methods

7.2 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 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 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.

7.2.1 Machine-learning based Prediction

Figure 7.1 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

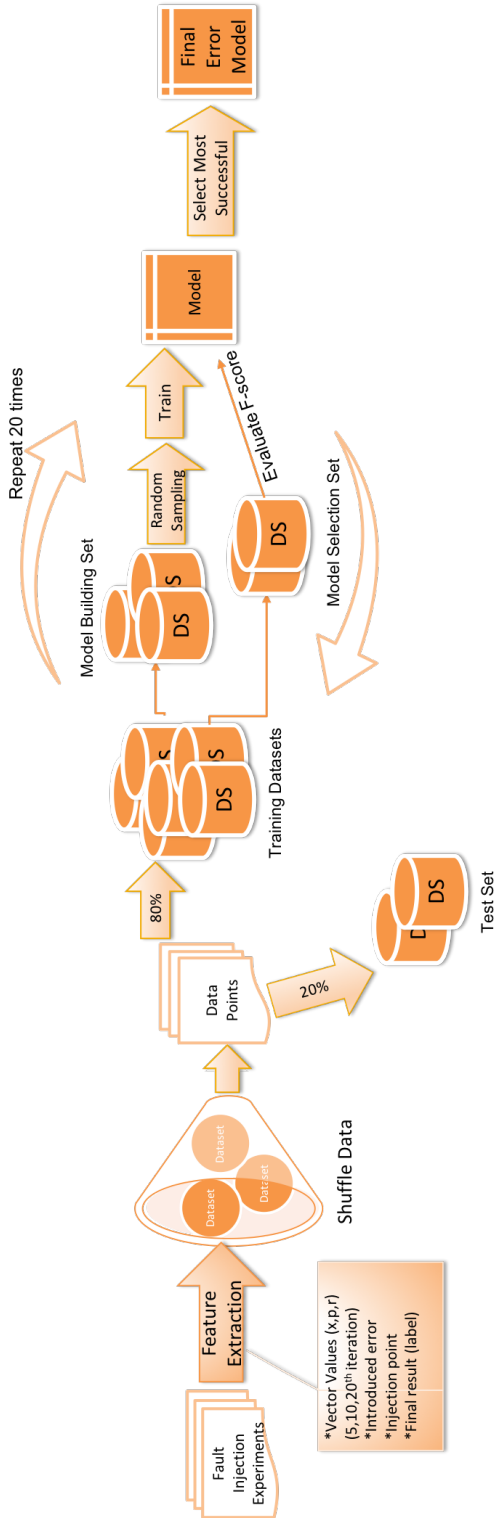


Fig. 7.1 Our overall approach to construct a ground-truth predictor using machine learning

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

$$\frac{\textit{Injected Iteration}}{\textit{Expected \# of Iterations}} \tag{7.1}$$

- 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 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 [59]. 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 iteration window, as any change introduced by the error can hinder the convergence, but also, by chance, it can help moving the execution to the right direction [58]. 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 7.1 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—number of datasets used to build the model and 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.

7.2.2 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 control the temporal and spatial aspect 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 [1, 2]. 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.

Our error injection framework instruments the source code to simulate an error. The framework decides in where to inject the error based on the inputs provided by the user (similar to our previous works [46, 58]). 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 [58]. 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.

7.2.3 Overall Algorithm: Error Injection with Ground Truth Prediction

Figure 7.2 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

```
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. 7.2 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.

(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.

7.3 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:

$$F\text{-}score = 2 \times \frac{precision \cdot recall}{precision + recall} \tag{7.2}$$

7.3.1 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.

	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

Fig. 7.3 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.

This procedure is repeated for different numbers of data sets used for model building and total number of error injections used for training. Figure 7.3 shows the results of our model building analysis. We build 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.

7.3.2 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 7.4 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 exact match between vulnerabilities from actual and predicted ground truths.

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

- Adaptive Impact-driven Detection (AID) [23] introduces an "impact error bound" that is used to pinpoint influential soft errors. They use dynamic curve fitting to detect an influential soft error.
- Checksums for matrix-vector multiplication (NEWSUM) Tao et al. [76] 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

[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 [58]

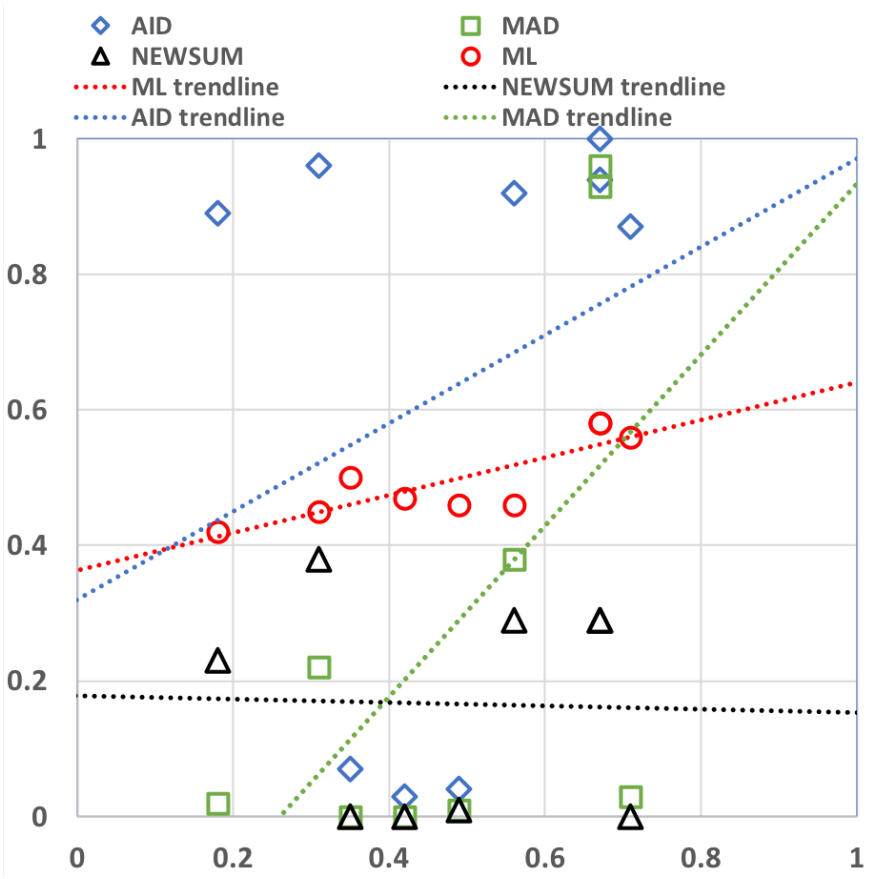


Fig. 7.4 Predicted MASKED ratio plotted against actual MASKED ratio. x-axis: MASKED ratio predicted from each candidate predictor. y-axis: MASKED ratio computed using ground truth from error injection experiments. ML denotes our approach. Each dot represents a solver-dataset pair. Trendlines for each 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.

irregular increases in consecutive time periods, which points to an unexpected behavior, hence detection of an soft error [52].

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 suffers 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 when only a subset of the scenarios considered are of interest (say to design a detector for a subset of the error injection points). Table 7.1 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 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.

7.3.3 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 evaluation. Table 7.2 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 matches 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.

7.3.4 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 7.5 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 7.6 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 7.5, 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 7.5, 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 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 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.

7.3.5 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 user decides on an anomalous run after 105% of the expected iterations (5% flexibility 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 prediction method and traditional method waits for the end of execution. So when a detector (correctly or not) does not detect any anomalies,

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 7.1 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.

		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 7.2 Detector precision and recall when calculated with actual ground truth of the executions, and compared with predicted ground truths using our approach.

our provided gain is on the smaller side. However, when detectors have many detection flags raised, the benefit of our approach magnifies.

7.3.6 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.

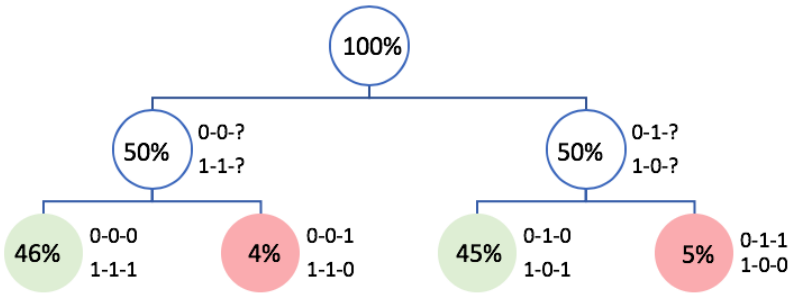


Fig. 7.5 Classification of scenarios for the CG solver with the NEWSUM detector. The labels are of the form a-b-c, where a is the prediction outcome, b is the detector's judgement, and c is the ground truth. Ideally, the red circles (where we judge a detector based on the wrong prediction) will be 0.

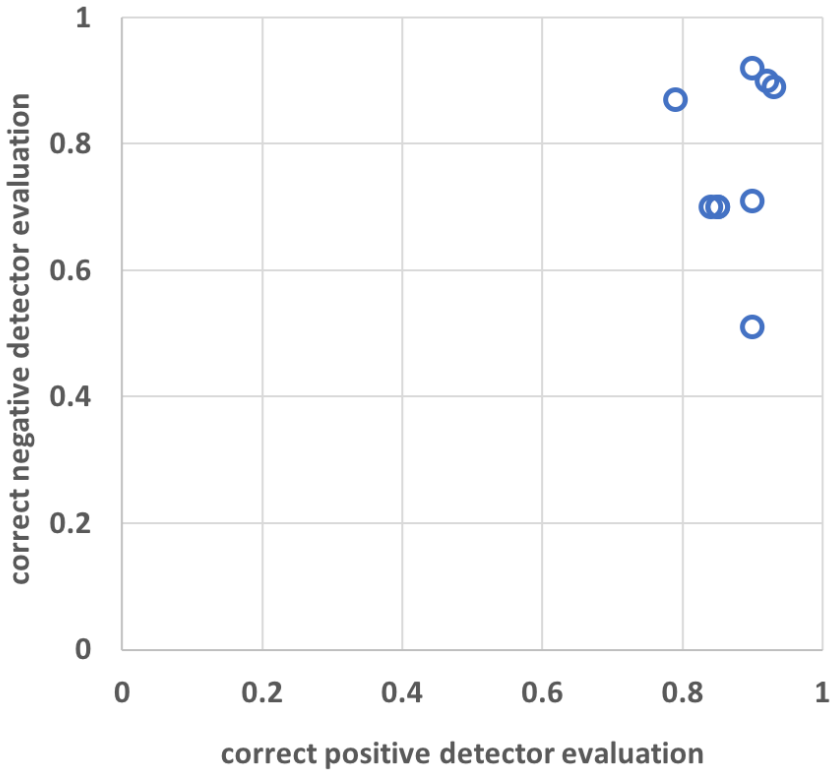


Fig. 7.6 Predictor accuracy is evaluating positive and negative detector outcomes. x-axis: fraction of all cases where predictor and detector match (marking the detector as being correct), where the ground truth also matches. y-axis: fraction of all cases where predictor and detector differ (flagging the detector as being incorrect), where the detector differs from the ground-truth.

When solvers are run with our method, with the predictor running, the total time is around 160% of a normal run time. As we leveraged `Python` libraries and `C++` to `Python` connections 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) [58], and saw 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

$$\frac{COST_{proposed}}{COST_{traditional}} = \frac{N \times (I_P + I_{mntr}) + P}{N \times I'} \tag{7.3}$$

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

$$I_{mntr} \approx 0.01 \times I. \tag{7.4}$$

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

$$I' = 15 \times I \tag{7.5}$$

(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

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 7.3 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.

before an injection, so the proposed iteration cost is:

$$COST_{proposed} = (N \times (I/2 + (0.01 \times I))) \tag{7.6}$$

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

$$\frac{N \times (I/2 + (0.01 \times I))}{N \times 15 \times I} \tag{7.7}$$

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).

7.3.7 Transferability of the Models

While developing ground truth prediction models trained on the error injections from each solver will give us the 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 7.3 shows the results from 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 7.1.

7.3.8 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 7.7 and Tables 7.4, 7.5, and 7.6 we analyzed how the results would change when we change this tolerance amount. We demonstrated the performance where no tolerance (0%), 10% tolerance, and 20% tolerance is applied to the injection experiments.

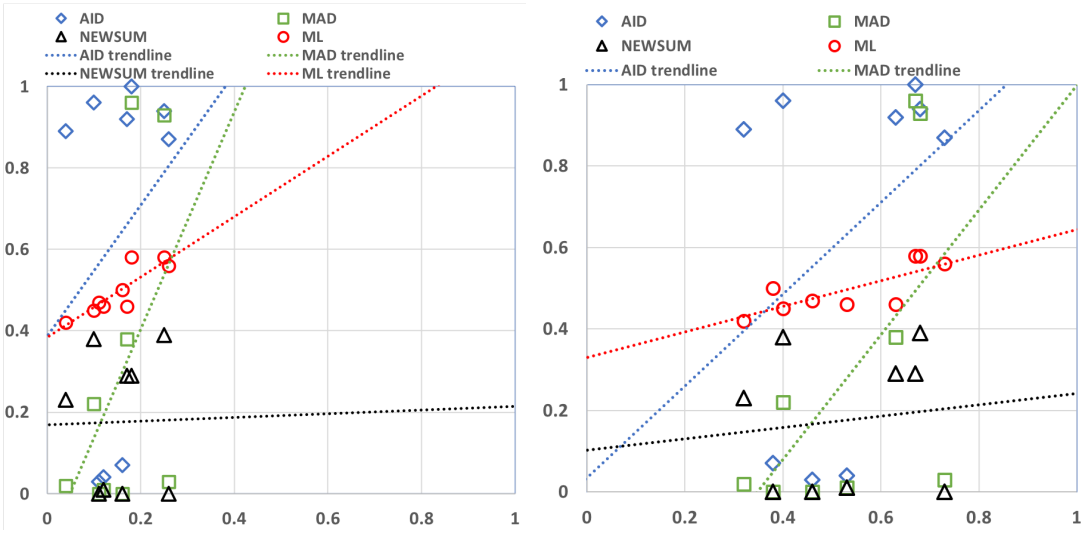
Figure 7.7 shows slight differences compared to Figure 7.4 but in all tolerance configurations we can still observe the machine learning approach showing the strongest linear relationship (R^2 closest to 1) between the masked ratio computed using the ground truth and our approach.

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 number of iterations taken. Tables 7.4, 7.5, and 7.6 shows that adjusting the MASKED has slightly affected the performance, nevertheless the machine learning approach shows best precision and recall. In some cases even better than our selected configuration.

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

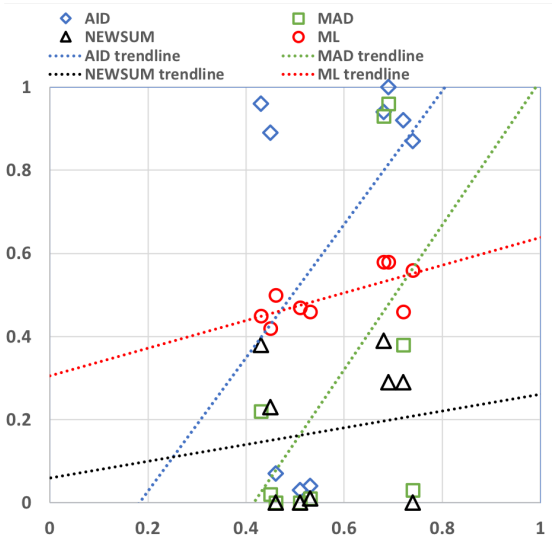
Table 7.4 Precision and recall of estimation of masked instances **(0 % tolerance)** using various candidate predictors. ML denotes our approach. An ideal detector will have precision and recall close to 1. The best candidate for each solver is shown in bold.

We also demonstrated the affect 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 affects of configuration deviations, we also evaluated different train-test splitting methods for our approach. Figure 7.8 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,



(a) Tolerance: 0%

(b) Tolerance: 10%



(c) Tolerance: 20%

Fig. 7.7 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 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.

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 7.5 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

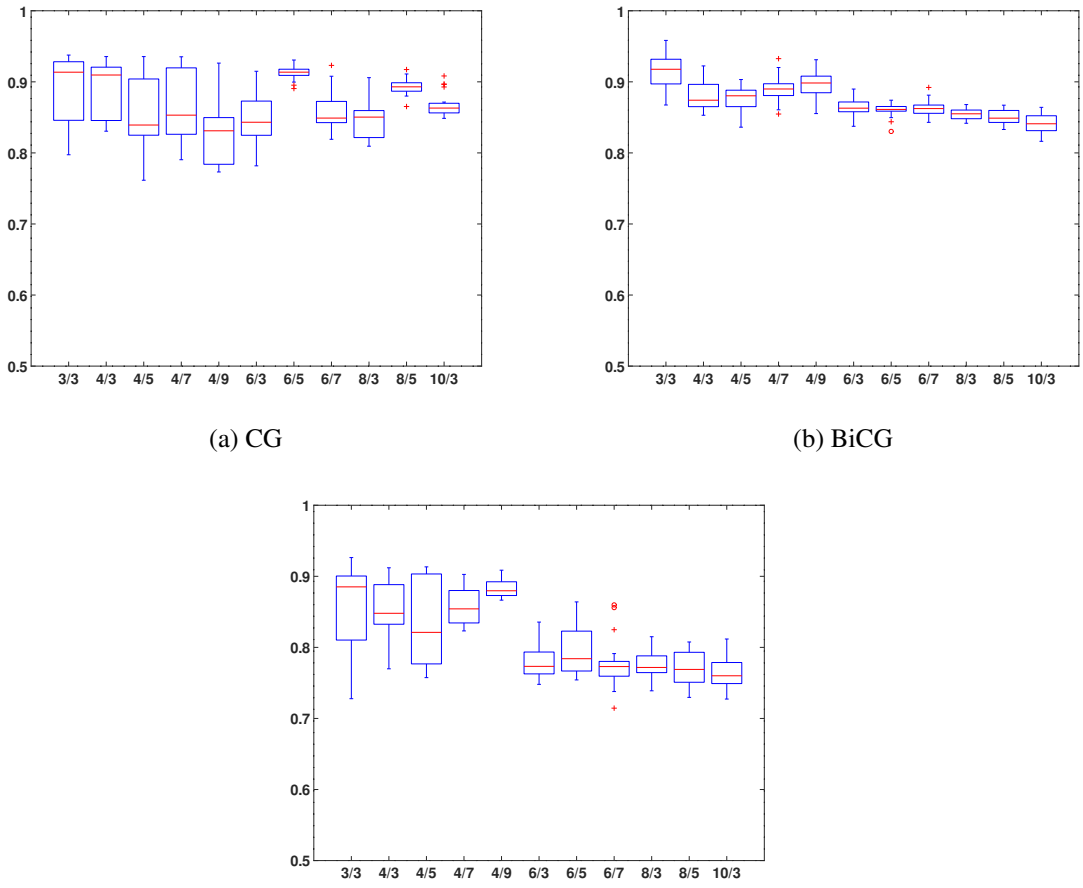
Table 7.6 Precision and recall of estimation of masked instances **(20 % tolerance)** using various candidate predictors. ML denotes our approach. An ideal detector will have precision and recall close to 1. The best candidate for each solver is shown in bold.

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.

7.4 Conclusion

In this chapter, 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 cuts costs by 21% - 53% depending on the solver and detector characteristics. We also



(a) CG

(b) BiCG

(c) CGS

Fig. 7.8 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.

analyze that, once we have a model trained, this method can provide acceptable results with 10% of the cost of exhaustive injection experiments.

Chapter 8

Conclusions and Future Work

8.1 Detector Composition

As discussed in Chapters 5 and 6; we observed that, even though each detector has their strengths, none of them performed perfectly. Our hypothesis for Chapter 6 was that there is room for improvement that we possibly can explore using the features of the detectors. This study showed us, while using machine learning nudged the detection performance in the right direction, it wasn't enough improvement, and more work needed to be put into the premise.

We can achieve having these detection methods work together in two ways;

- Feature Composition; monitor the features used by the detectors and make a decision using all of the info, as we explored in Chapter 6
- Detector Composition; as in having more than one detector work at the same time and decide on a final flag using these individual flags.

This work will focus on the second method. There are several avenues we plan to explore in this study.

First, we will run all the detectors together and trace the flags of each detection mechanism. We will get a set of flags for each injection run and use Machine Learning to have a viable model that can flag the actual behavior.

Then, we can use detectors according to their strengths. We will analyze the execution data we have for detectors temporally and spatially. We hope to find differences in detector performance for different times of the execution or different components of the algorithm. As shown in Figure 5.1, convergence characteristics of the algorithms are not linear, and beginnings of the executions are significantly different from the final part. Therefore, it is not

unreasonable to expect some detection methods perform differently at the initial iterations compared to later iterations. Hence, we can use the detector(s) that are stronger in the initial iterations first and then switch to the ones that work better on later iterations.

Spatially, we will see if some detectors are better on detecting the anomalies in certain data structures or during certain procedures in the algorithm (statements in the code), using this knowledge we can use detectors side by side each detecting an anomaly in different parts of the execution.

What's more, for this study we activated the detectors after every iteration in the execution. As the developers of the detectors suggest, we can find a frequency for the detectors to work in every n iteration. We can heuristically find an optimal period for the detectors and have them work in different intervals together. Then as before, we can use this string of flags in a window to decide on a final flag for the execution.

8.2 Conclusions

Architectural trends such as technology scaling and near-threshold voltage operation are expected to make soft error resilience an important consideration in performance-oriented and power-constrained environments. Soft errors –transient bit flips– impacting an application state, can lead to application crashes, slow down in execution, or silent data corruption.

A broad array of techniques has been designed to understand application behavior under soft errors and to detect, isolate, and correct soft-error-impacted application state. The first step toward tolerating soft errors involves understanding an application's behavior under soft errors. This can help understand the need for error detection/correction techniques. An ideal error detection/correction strategy identifies all and only the errors that can materially impact application behavior. Detecting and recovering from errors that might be eventually masked by the application can unnecessary increase the cost of soft error resilience. Different portions of the application state might be impacted differently by a soft error, enabling optimizations and data-structure-specific resilience techniques. Evaluating the effectiveness of such techniques requires a systematic evaluation of their effectiveness in protecting various portions of the application state throughout the execution. As systems keep scaling, new methods will be needed for detection/correction of these errors, new methods to optimize the process of testing is a compelling step towards resiliency.

In this thesis we contributed in this field in following ways;

- We presented a comprehensive characterization of the iterative method behavior under soft errors. We considered 6 solvers, 28 datasets, and multiple fault injection scenarios.

We believe this data is a useful resource that can aid in testing runtime behavior of iterative solvers, comparative solver evaluation, error detection studies, and design space exploration.

- To enable such analysis, we have publicly released the data from the error injection experiments at IMIC database (https://github.com/pnnl/IMIC).
- We presented a comprehensive evaluation of the behavior of soft error detectors. We consider five iterative methods, 28 data sets, and multiple fault-injection scenarios. We evaluated flagging an error based on detector behavior at a single iteration or over a sliding window of iterations. While each detector considered has been shown to be effective in a distinct context, extensive analysis of various configurations evaluated demonstrates that, in the context of iterative methods, they do not achieve perfect detection accuracy. Given the high false positive rates, which can lead to a large re-execution overhead, existing detection techniques might better serve as a component of a larger detection system.
- To identify the potential for an improved accuracy based on the features used by the detectors evaluated, we presented a machine learning based detector using these features. While improved, the machine learning based detector is still far from perfect in terms of its accuracy. We believe, in addition to new methods, additional features need to be incorporated to improve detection accuracy.
- 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.

This work can be beneficial for different areas in the field.

Domain scientists developers can benefit from the provided data and tools. Using this knowledge can help developers to design selective resilience strategies to only protect the most vulnerable components of the algorithms when resources are limited. For example, one can opt to protect certain parts of a floating point, or they can protect certain data structures (some vectors rather than all, etc.). Besides, in pursuit of energy efficiency, the components that are more resilient or that have a lower impact on the application state can be placed on less reliable memory. Additionally, detailed understanding of the vulnerabilities of iterative solvers will yield more resilient algorithm designs.

When focusing on making robust systems / algorithms against errors; **software scientists** can choose the right detection mechanism for their needs using the knowledge provided. We

showed there is room for improvement combining the detectors' knowledge and machine learning. Future detectors can be built keeping these in mind. When developing a new detection mechanism or getting the error profile of a software system, the proposed prediction mechanism will facilitate and speed up the process.

This study is also useful for **hardware scientists**. With the new developments of hardware, new error mitigation strategies need to be established. Novel techniques need a deep understanding of the vulnerabilities of expected workloads of the systems, so that they can protect the system efficiently. This study sheds light on the types of errors that iterative solvers are more prone to be affected by. This knowledge and the prediction techniques will facilitate new hardware component design and testing.

What's more, as the data from vast fault injection experiments is publicly available, **data scientists** can utilize this database for machine learning algorithm design and testing, or parameter fine-tuning studies. The data provided at the IMIC database can be a useful dataset for data scientists, as it has many dimensions (solvers, datasets, vectors, error distributions, outcome classes).

Chapter 9

Publications and Invited Talks

The work of the thesis has resulted in the following peer-reviewed publications.

- **Mutlu, B. O.**, Kestor, G., Manzano, J. B., Unsal, O. S., Chatterjee, S., and Krishnamoorthy, S. (2018). *Characterization of the Impact of Soft Errors on Iterative Methods*. In The 14th Workshop on Silicon Errors in Logic- System Effects, SELSE 2018, Boston, USA, April 3-4, 2018.

- Kestor, G., **Mutlu, B. O.**, Manzano, J. B., Subasi, O., Unsal, O. S., and Krishnamoorthy,S. (2018). *Comparative analysis of soft-error detection strategies: a case study with iterative methods.* In Proceedings of the 15th ACM International Conference on Computing Frontiers, CF 2018, Ischia, Italy, May 08-10, 2018, pages 173–182.

- **Mutlu, B. O.**, Kestor, G., Manzano, J. B., Unsal, O. S., Chatterjee, S., and Krishnamoorthy, S. (2018). *Characterization of the impact of soft errors on iterative methods*. In 25th IEEE International Conference on High Performance Computing, HiPC 2018, Bengaluru, India, December 17-20, 2018, pages 203–214.

- **Mutlu B. O.**, Kestor, G., and Krishnamoorthy, S. (2018). *IMIC Iterative Methods Injection Collection*. 2018. url: http://https://github.com/pnnl/ IMIC (visited on 12/20/2018).

- **Mutlu B. O.**, Kestor, G., Cristal, A., Unsal, O. S., and Krishnamoorthy, S. (2019) *Ground-Truth Prediction to Accelerate Soft-Error Impact Analysis for Iterative Methods*. *Submitted to* 26th IEEE International Conference on High Performance Computing, HiPC 2019, Hyderabad, India, December 17-20, 2019.

- Manzano, J. B., **Mutlu B. O.**, Kestor, G., Li, A., and Krishnamoorthy, S. *Exploring Deep Learning Models for Silent Data Corruption in Scientific Kernels: A Case Study*. *In process of submission.*

This work also resulted in following invited talks.

- **Mutlu, B. O.**, Kestor, G., Manzano, J. B., Chatterjee, S., Unsal, O. S., and Krishnamoorthy, S. (2018) *Impact Of Soft Errors On Iterative Methods And Analysis Of Detection Methods*. Presented at TechFest Computing at PNNL 2018 event. url: techfest2018.pnl.gov. TechFest 2018, Richland, WA, USA, June 7, 2018.

- **Mutlu, B. O.**, Kestor, G., Unsal, O. S., and Krishnamoorthy, S. (2018). *Characterization of the impact of soft errors on iterative methods*. Presented at Women in HPC: Diversifying the HPC Workforce workshop in conjunction with SC18, Dallas, TX, USA, November 11, 2018.

References

[1] Ali, N., Krishnamoorthy, S., Halappanavar, M., and Daily, J. (2011). Tolerating correlated failures for generalized cartesian distributions via bipartite matching. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, pages 36:1–36:10, New York, NY, USA. ACM.

[2] Ali, N., Krishnamoorthy, S., Halappanavar, M., and Daily, J. (2013). Multi-fault tolerance for cartesian data distributions. *International Journal of Parallel Programming*, 41(3):469–493.

[3] Ashraf, R. A., Gioiosa, R., Kestor, G., DeMara, R. F., Cher, C., and Bose, P. (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*, pages 1–12.

[4] Balay, S., Abhyankar, S., Adams, M. F., Brown, J., Brune, P., Buschelman, K., Dalcin, L., Eijkhout, V., Gropp, W. D., Kaushik, D., Knepley, M. G., May, D. A., McInnes, L. C., Rupp, K., Smith, B. F., Zampini, S., Zhang, H., and Zhang, H. (2017). PETSc Web page. http://www.mcs.anl.gov/petsc.

[5] Bautista-Gomez, L., Zyulkyarov, F., Unsal, O. S., and McIntosh-Smith, S. (2016). Unprotected computing: a large-scale study of DRAM raw error rate on a supercomputer. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016*, pages 645–655.

[6] Berrocal, E., Bautista-Gomez, L. A., Di, S., Lan, Z., and Cappello, F. (2015). Lightweight silent data corruption detection based on runtime data analysis for HPC applications. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2015, Portland, OR, USA, June 15-19, 2015*, pages 275–278.

[7] Böhm, S. and Engelmann, C. (2011). xSim: The extreme-scale simulator. In *2011 International Conference on High Performance Computing & Simulation, HPCS 2012, Istanbul, Turkey, July 4-8, 2011*, pages 280–286.

[8] Bronevetsky, G. and de Supinski, B. R. (2008). Soft error vulnerability of iterative linear algebra methods. In *Proceedings of the 22nd Annual International Conference on Supercomputing, ICS 2008, Island of Kos, Greece, June 7-12, 2008*, pages 155–164.

[9] Butler, M., Barnes, L., Sarma, D. D., and Gelinas, B. (2011). Bulldozer: An approach to multithreaded compute performance. *IEEE Micro*, 31(2):6–15.

[10] Calhoun, J., Olson, L., and Snir, M. (2014). Flipit: An LLVM based fault injector for HPC. In *Euro-Par 2014: Parallel Processing Workshops - Euro-Par 2014 International Workshops, Porto, Portugal, August 25-26, 2014, Revised Selected Papers, Part I*, pages 547–558.

[11] Carbin, M., Misailovic, S., and Rinard, M. C. (2013). Verifying quantitative reliability for programs that execute on unreliable hardware. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 33–52, New York, NY, USA. ACM.

[12] Casas-Guix, M., de Supinski, B. R., Bronevetsky, G., and Schulz, M. (2012). Fault resilience of the algebraic multi-grid solver. In *International Conference on Supercomputing, ICS'12, Venice, Italy, June 25-29, 2012*, pages 91–100.

[13] Chen, Z. (2013a). Online-ABFT: An online algorithm based fault tolerance scheme for soft error detection in iterative methods. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 167–176, New York, NY, USA. ACM.

[14] Chen, Z. (2013b). Online-abft: An online algorithm based fault tolerance scheme for soft error detection in iterative methods. In *PPoPP*, pages 167–176.

[15] Cher, C., Muller, K. P., Haring, R. A., Satterfield, D. L., Musta, T. E., Gooding, T., Davis, K. D., Dombrowa, M. B., Kopcsay, G. V., Senger, R. M., Sugawara, Y., and Sugavanam, K. (2014). Soft error resiliency characterization on IBM bluegene/q processor. In *19th Asia and South Pacific Design Automation Conference, ASP-DAC 2014, Singapore, January 20-23, 2014*, pages 385–387.

[16] Cho, H., Mirkhani, S., Cher, C., Abraham, J. A., and Mitra, S. (2013). Quantitative evaluation of soft error injection techniques for robust system design. In *The 50th Annual Design Automation Conference 2013, DAC '13, Austin, TX, USA, May 29 - June 07, 2013*, pages 101:1–101:10.

[17] Ciocca, E., Koren, I., Koren, Z., Krishna, C. M., and Katz, D. S. (2004). Application-level fault tolerance in the orbital thermal imaging spectrometer. In *10th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2004), 3-5 March 2004, Papeete, Tahiti*, pages 43–48.

[18] Cools, S., Vanroose, W., Yetkin, E. F., Agullo, E., and Giraud, L. (2016). On rounding error resilience, maximal attainable accuracy and parallel performance of the pipelined conjugate gradients method for large-scale linear systems in petsc. In *Proceedings of the Exascale Applications and Software Conference 2016, Stockholm, Sweden, April 26-29, 2016*, pages 3:1–3:10.

[19] Coplin, J. and Burtscher, M. (2015). Effects of source-code optimizations on GPU performance and energy consumption. In *Proceedings of the 8th Workshop on General Purpose Processing using GPUs, GPGPU@PPoPP 2015, San Francisco, CA, USA, February 7, 2015*, pages 48–58.

[20] Das, A., Mueller, F., Siegel, C., and Vishnu, A. (2018). Desh: Deep learning for system health prediction of lead times to failure in hpc. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '18, pages 40–51, New York, NY, USA. ACM.

[21] Davis, T. A. and Hu, Y. (2011). The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25.

[22] Di, S., Berrocal, E., and Cappello, F. (2015). An efficient silent data corruption detection method with error-feedback control and even sampling for HPC applications. In *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2015, Shenzhen, China, May 4-7, 2015*, pages 271–280.

[23] Di, S. and Cappello, F. (2016). Adaptive impact-driven detection of silent data corruption for HPC applications. *IEEE Transactions on Parallel and Distributed Systems*, 27(10):2809–2823.

[24] Dongarra, J., Lumsdaine, A., Pozo, R., and Remington, K. (1995). Iml++ v. 1.2 iterative methods library reference guide. Technical Report CS-95-303, University of Tennessee.

[25] Dongarra, J. J., Heroux, M. A., and Luszczek, P. (2016). High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems. *The International Journal of High Performance Computing Applications IJHPCA*, 30(1):3–10.

[26] Dongarra, J. J., Luszczek, P., and Petitet, A. (2003). The LINPACK benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820.

[27] Dozat, T. (2015). Incorporating nesterov momentum into adam.

[28] Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12:2121–2159.

[29] Elliott, J., Hoemmen, M., and Mueller, F. (2014). Evaluating the impact of SDC on the GMRES iterative solver. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*, pages 1193–1202.

[30] European Technology Platform for High Performance Computing (2017). Strategic research agenda (sra 3). Technical report.

[31] Farahani, B. and Safari, S. (2015). 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)*, pages 215–220.

[32] Feng, S., Gupta, S., Ansari, A., and Mahlke, S. A. (2010). Shoestring: probabilistic soft error reliability on the cheap. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2010, Pittsburgh, Pennsylvania, USA, March 13-17, 2010*, pages 385–396.

[33] Fiala, D., Mueller, F., Engelmann, C., Riesen, R., Ferreira, K. B., and Brightwell, R. (2012). Detection and correction of silent data corruption for large-scale high-performance computing. In *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA - November 11 - 15, 2012*, page 78.

[34] Folkesson, P., Svensson, S., and Karlsson, J. (1998). A comparison of simulation based and scan chain implemented fault injection. In *Digest of Papers: FTCS-28, The Twenty-Eigth Annual International Symposium on Fault-Tolerant Computing, Munich, Germany, June 23-25, 1998*, pages 284–293.

[35] Gainaru, A., Cappello, F., Snir, M., and Kramer, W. (2013). Failure prediction for HPC systems and applications: Current situation and open issues. *The International Journal of High Performance Computing Applications*, 27(3):273–282.

[36] Ganapathy, S., Kalamatianos, J., Kasprak, K., and Raasch, S. (2017). On characterizing near-threshold SRAM failures in finfet technology. In *Proceedings of the 54th Annual Design Automation Conference, DAC 2017, Austin, TX, USA, June 18-22, 2017*, pages 53:1–53:6.

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

[38] Han, S., Shin, K. G., and Rosenberg, H. A. (1995). DOCTOR: an integrated software fault injection environment for distributed real-time systems. In *IEEE International Computer Performance and Dependability Symposium*, pages 204–213.

[39] Hinton, G. et al. (2014). Neural networks for machine learning: Lecture 6a overview of mini-batch gradient descent.

[40] Hsu, C. and Feng, W. (2005). A power-aware run-time system for high-performance computing. In *Proceedings of the ACM/IEEE SC2005 Conference on High Performance Networking and Computing, November 12-18, 2005, Seattle, WA, USA, CD-Rom*, page 1.

[41] Hsueh, M., Tsai, T. K., and Iyer, R. K. (1997). Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82.

[42] Jin, A., Jiang, J., Hu, J., and Lou, J. (2008). A pin-based dynamic software fault injection system. In *Proceedings of the 9th International Conference for Young Computer Scientists, ICYCS 2008, Zhang Jia Jie, Hunan, China, November 18-21, 2008*, pages 2160–2167.

[43] Kaelbling, L. P., Littman, M. L., and Moore, A. P. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285.

[44] Kalra, C., Previlon, F., Li, X., Rubin, N., and Kaeli, D. R. (2018). PRISM: predicting resilience of GPU applications using statistical methods. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, Dallas, TX, USA, November 11-16, 2018*, pages 69:1–69:14.

[45] Kanawati, G. A., Kanawati, N. A., and Abraham, J. A. (1995). FERRARI: A flexible software-based fault and error injection system. *IEEE Trans. Computers*, 44(2):248–260.

[46] Kestor, G., Mutlu, B. O., Manzano, J. B., Subasi, O., Unsal, O. S., and Krishnamoorthy, S. (2018). Comparative analysis of soft-error detection strategies: a case study with iterative methods. In *Proceedings of the 15th ACM International Conference on Computing Frontiers, CF 2018, Ischia, Italy, May 08-10, 2018*, pages 173–182.

[47] Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980.

[48] Laguna, I., Schulz, M., Richards, D. F., Calhoun, J., and Olson, L. N. (2016). IPAS: intelligent protection against silent output corruption in scientific applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization, CGO 2016, Barcelona, Spain, March 12-18, 2016*, pages 227–238.

[49] Li, D., Vetter, J. S., and Yu, W. (2012). Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool. In *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA - November 11 - 15, 2012*, page 57.

[50] Li, M., Ramachandran, P., Karpuzcu, U. R., Hari, S. K. S., and Adve, S. V. (2009). Accurate microarchitecture-level fault modeling for studying hardware faults. In *15th International Conference on High-Performance Computer Architecture (HPCA-15 2009), 14-18 February 2009, Raleigh, North Carolina, USA*, pages 105–116.

[51] Liu, J. and Agrawal, G. (2016). Soft error detection for iterative applications using offline training. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*, pages 2–11.

[52] Liu, J., Kurt, M. C., and Agrawal, G. (2015). A practical approach for handling soft errors in iterative applications. In *2015 IEEE International Conference on Cluster Computing, CLUSTER 2015, Chicago, IL, USA, September 8-11, 2015*, pages 158–161.

[53] Lucas, R., Ang, J., Bergman, K., Borkar, S., Carlson, W., Carrington, L., Chiu, G., Colwell, R., Dally, W., Dongarra, J., Geist, A., Haring, R., Hittinger, J., Hoisie, A., Klein, D. M., Kogge, P., Lethin, R., Sarkar, V., Schreiber, R., Shalf, J., Sterling, T., Stevens, R., Bashor, J., Brightwell, R., Coteus, P., Debenedictus, E., Hiller, J., Kim, K. H., Langston, H., Murphy, R. M., Webster, C., Wild, S., Grider, G., Ross, R., Leyffer, S., and Laros III, J. (2014). DOE Advanced Scientific Computing Advisory Subcommittee (ASCAC) report: Top ten exascale research challenges.

[54] Malkowski, K., Raghavan, P., and Kandemir, M. T. (2010). Analyzing the soft error resilience of linear solvers on multicore multiprocessors. In *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings*, pages 1–12.

[55] Maniatakos, M., Karimi, N., Tirumurti, C., Jas, A., and Makris, Y. (2011). Instruction-level impact analysis of low-level faults in a modern microprocessor controller. *IEEE Trans. Computers*, 60(9):1260–1273.

[56] Mukherjee, S. S., Weaver, C. T., Emer, J., Reinhardt, S. K., and Austin, T. (2003). Measuring architectural vulnerability factors. *IEEE Micro*, 23(6):70–75.

[57] Mukherjee, S. S., Weaver, C. T., Emer, J. S., Reinhardt, S. K., and Austin, T. M. (2003). A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the 36th Annual International Symposium on Microarchitecture, San Diego, CA, USA, December 3-5, 2003*, pages 29–42.

[58] Mutlu, B. O., Kestor, G., Manzano, J. B., Unsal, O. S., Chatterjee, S., and Krishnamoorthy, S. (2018). Characterization of the impact of soft errors on iterative methods. In *25th IEEE International Conference on High Performance Computing, HiPC 2018, Bengaluru, India, December 17-20, 2018*, pages 203–214.

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

[60] Segall, Z., Vrsalovic, D. F., Siewiorek, D. P., Yaskin, D. A., Kownacki, J., Barton, J. H., Dancey, R., Robinson, A., and Lin, T. (1988). Fiat-fault injection based automated testing environment. In *Proceedings of the Eighteenth International Symposium on Fault-Tolerant Computing, FTCS 1988, Tokyo, Japan, 27-30 June, 1988*, pages 102–107.

[61] Shantharam, M., Srinivasmurthy, S., and Raghavan, P. (2011). Characterizing the impact of soft errors on iterative methods in scientific computing. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 152–161, New York, NY, USA. ACM.

[62] Sharma, V. C., Gopalakrishnan, G., and Krishnamoorthy, S. (2016). Towards resiliency evaluation of vector programs. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2016, Chicago, IL, USA, May 23-27, 2016*, pages 1319–1328.

[63] Sharma, V. C., Haran, A., Rakamaric, Z., and Gopalakrishnan, G. (2013). Towards formal approaches to system resilience. In *IEEE 19th Pacific Rim International Symposium on Dependable Computing, PRDC 2013, Vancouver, BC, Canada, December 2-4, 2013*, pages 41–50.

[64] Shewchuk, J. R. (1994). An introduction to the conjugate gradient method without the agonizing pain. Technical report, Pittsburgh, PA, USA.

[65] Sieh, V., Tschäche, O., and Balbach, F. (1997). VERIFY: evaluation of reliability using vhdl-models with embedded fault descriptions. In *Digest of Papers: FTCS-27, The Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing, Seattle, Washington, USA, June 24-27, 1997*, pages 32–36.

[66] Skarin, D., Barbosa, R., and Karlsson, J. (2010). GOOFI-2: A tool for experimental dependability assessment. In *Proceedings of the 2010 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2010, Chicago, IL, USA, June 28 - July 1 2010*, pages 557–562.

[67] Sloan, J., Kumar, R., and Bronevetsky, G. (2012). Algorithmic approaches to low overhead fault detection for sparse linear algebra. In *IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2012, Boston, MA, USA, June 25-28, 2012*, pages 1–12.

[68] Smolens, J. (2007). Fingerprinting: Hash-based error detection in microprocessors. *PhD Thesis, ECE/CMU*.

[69] Song, K., Liu, Y., Wang, R., Zhao, M., Hao, Z., and Qian, D. (2016). Restricted boltzmann machines and deep belief networks on sunway cluster. In *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 245–252.

[70] Sridharan, V. and Kaeli, D. R. (2008a). Quantifying software vulnerability. In *Proceedings of the 2008 Workshop on Radiation Effects and Fault Tolerance in Nanometer Technologies*, WREFT '08, pages 323–328, New York, NY, USA. ACM.

[71] Sridharan, V. and Kaeli, D. R. (2008b). Quantifying software vulnerability. In *Proceedings of the 2008 Workshop on Radiation Effects and Fault Tolerance in Nanometer Technologies*, WREFT '08, pages 323–328, New York, NY, USA. ACM.

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

[73] Subasi, O., Di, S., Bautista-Gomez, L., Balaprakash, P., Ünsal, O. S., Labarta, J., Cristal, A., and Cappello, F. (2016). Spatial support vector regression to detect silent errors in the exascale era. In *IEEE/ACM 16th International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2016, Cartagena, Colombia, May 16-19, 2016*, pages 413–424.

[74] Subasi, O., Moreno, J. A., Unsal, O. S., Labarta, J., and Cristal, A. (2015). Programmer-directed partial redundancy for resilient HPC. In *Proceedings of the 12th ACM International Conference on Computing Frontiers, CF'15, Ischia, Italy, May 18-21, 2015*, pages 47:1–47:2.

[75] Supercomputers, T. (2018). Top 500 Supercomputers using HPCG.

[76] Tao, D., Song, S. L., Krishnamoorthy, S., Wu, P., Liang, X., Zhang, E. Z., Kerbyson, D., and Chen, Z. (2016). New-Sum: A novel online ABFT scheme for general iterative methods. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '16, pages 43–55, New York, NY, USA. ACM.

[77] Thomas, T. E., Bhattad, A. J., Mitra, S., and Bagchi, S. (2016). Sirius: Neural network based probabilistic assertions for detecting silent data corruption in parallel programs. In *35th IEEE Symposium on Reliable Distributed Systems, SRDS 2016, Budapest, Hungary, September 26-29, 2016*, pages 41–50.

[78] Tsai, T. K. and Iyer, R. K. (1995). Measuring fault tolerance with the FTAPE fault injection tool. In *Quantitative Evaluation of Computing and Communication Systems, 8th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, Performance Tools '95, 8th GI/ITG Conference on Measuring, Modeling and Evaluating Computing and Communication Systems, MMB '95, Heidelberg, Germany, September 20-22, 1995, Proceedings*, pages 26–40.

[79] Turmon, M., Granat, R., Katz, D. S., and Z. Lou, J. (2003). Tests and tolerances for high-performance software-implemehted fault detection. *Computers, IEEE Transactions on*, 52:579 – 591.

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

[81] Vishnu, A., Dam, H. V., Tallent, N. R., Kerbyson, D. J., and Hoisie, A. (2016). Fault modeling of extreme scale applications using machine learning. In *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*, pages 222–231.

[82] Vishnu, A., Manzano, J. B., Siegel, C., and Daily, J. (2017). User-transparent distributed tensorflow. *CoRR*, abs/1704.04560.

[83] Wei, J., Thomas, A., Li, G., and Pattabiraman, K. (2014). Quantifying the accuracy of high-level fault injection techniques for hardware faults. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*, pages 375–382.

[84] Xu, X. and Li, M. (2012). Understanding soft error propagation using efficient vulnerability-driven fault injection. In *IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2012, Boston, MA, USA, June 25-28, 2012*, pages 1–12.

[85] Yu, L., Li, D., Mittal, S., and Vetter, J. S. (2014). Quantitatively modeling application resilience with the data vulnerability factor. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16-21, 2014*, pages 695–706.

[86] Yu, L., Li, D., Mittal, S., and Vetter, J. S. (2014). 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*, pages 695–706.

[87] Zeiler, M. D. (2012). ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701.

[88] Zekany, S., Rings, D., Harada, N., Laurenzano, M. A., Tang, L., and Mars, J. (2016). Crystalball: Statically analyzing runtime behavior via deep sequence learning. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12.

[89] Ziade, H., Ayoubi, R. A., and Velazco, R. (2004). A survey on fault injection techniques. *Int. Arab J. Inf. Technol.*, 1(2):171–186.

Appendix A

Algorithm Implementations

In this appendix, pseudocode of the implementations of the solvers are provided to the reader for understanding the algorithms and how their statements used. In our experiments, following implementations of CG, ICCG, CGS, BICG, BICGSTAB and QMR solvers are used.

```
  for (int i = 1; i <= max_iter; i++) {
2   z = M.solve(r);
    rho(0) = dot(r, z);
4
    if (i == 1)
6     p = z;
    else {
8     beta(0) = rho(0) / rho_1(0);
      p = z + beta(0) * p;
10  }
    q = A * p;
12  alpha(0) = rho(0) / dot(p, q);
    x += alpha(0) * p;
14  r -= alpha(0) * q;
    resid = norm(r) / normb;
16  rho_1(0) = rho(0);

18  if (resid <= tol) {
      tol = resid;
20    max_iter = i;
      return 0;
22  }

24  tol = resid;
    return 1;
26 }
```

Listing A.1 The implementation used for CG and ICCG (CG with preconditioner) methods for solving the symmetric positive-definite system $A \cdot \vec{x} = \vec{b}$

```
  for (int i = 1; i <= max_iter; i++) {
    z = M.solve(r);
    rho(0) = dot(r, z);

    if (i == 1)
      p = z;
    else {
      beta(0) = rho(0) / rho_1(0);
      p = z + beta(0) * p;
    }
    q = A * p;
    alpha(0) = rho(0) / dot(p, q);
    x += alpha(0) * p;
    r -= alpha(0) * q;
    resid = norm(r) / normb;
    rho_1(0) = rho(0);

    if (resid <= tol) {
      tol = resid;
      max_iter = i;
      return 0;
    }

    tol = resid;
    return 1;
  }
```

Listing A.2 The implementation used for CG and ICCG (CG with preconditioner) methods for solving the symmetric positive-definite system $A \cdot \vec{x} = \vec{b}$

```
  for (int i = 1; i <= max_iter; i++) {
    rho_1(0) = dot(rtilde, r);

    if (i == 1) {
      u = r;
      p = u;
    } else {
      beta(0) = rho_1(0) / rho_2(0);
      u = r + beta(0) * q;
      p = u + beta(0) * (q + beta(0) * p);
    }

    phat = M.solve(p);
    vhat = A * phat;
    alpha(0) = rho_1(0) / dot(rtilde, vhat);
    q = u - alpha(0) * vhat;
    uhat = M.solve(u + q);
    x += alpha(0) * uhat;
    qhat = A * uhat;
    r -= alpha(0) * qhat;
    rho_2(0) = rho_1(0);

    resid = norm(r) / normb;
    if (resid <= tol) {
      tol = resid;
      return 0;
    }

    tol = resid;
    return 1;
  }
```

Listing A.3 The CGS method for solving the symmetric positive-definite system $A \cdot \vec{x} = \vec{b}$

```
  for (int i = 1; i <= max_iter; i++) {
    z = M.solve(r);
    ztilde = M.trans_solve(rtilde);
    rho_1(0) = dot(z, rtilde);
    if (rho_1(0) == 0) {
      tol = norm(r) / normb;
      max_iter = i;
      return 2;
    }
    if (i == 1) {
      p = z;
      ptilde = ztilde;
    } else {
      beta(0) = rho_1(0) / rho_2(0);
      p = z + beta(0) * p;
      ptilde = ztilde + beta(0) * ptilde;
    }
    q = A * p;
    qtilde = A.trans_mult(ptilde);
    alpha(0) = rho_1(0) / dot(ptilde, q);
    x += alpha(0) * p;
    r -= alpha(0) * q;
    rtilde -= alpha(0) * qtilde;
    rho_2(0) = rho_1(0);
    resid = norm(r) / normb;
    if (resid <= tol) {
      tol = resid;
      return 0;
    }

    tol = resid;
    return 1;
  }
```

Listing A.4 The BICG method for solving the symmetric positive-definite system $A \cdot \vec{x} = \vec{b}$

```
1   for (int i = 1; i <= max_iter; i++) {
      rho_1(0) = dot(rtilde, r);
3     if (i == 1)
        p = r;
5     else {
        beta(0) = (rho_1(0) / rho_2(0)) * (alpha(0) / omega(0));
7       p = r + beta(0) * (p - omega(0) * v);
      }
9     phat = M.solve(p);
      v = A * phat;
11    alpha(0) = rho_1(0) / dot(rtilde, v);
      s = r - alpha(0) * v;
13    if ((resid = norm(s) / normb) < tol) {
        x += alpha(0) * phat;
15      tol = resid;
        return 0;
17    }
      shat = M.solve(s);
19    t = A * shat;
      omega = dot(t, s) / dot(t, t);
21    x += alpha(0) * phat + omega(0) * shat;
      r = s - omega(0) * t;
23    rho_2(0) = rho_1(0);
      resid = norm(r) / normb;
25    if (resid <= tol) {
        tol = resid;
27      return 0;
      }
29    if (omega(0) == 0) {
        tol = norm(r) / normb;
31      return 3;
      }
33    tol = resid;
      return 1;
35  }
```

Listing A.5 The BICGSTAB method for solving the symmetric positive-definite system $A \cdot \vec{x} = \vec{b}$

```
for (int i = 1; i <= max_iter; i++) {
  if (rho(0) == 0.0)
    return 2;
  if (xi(0) == 0.0)
    return 7;

  v = (1. / rho(0)) * v_tld;
  y = (1. / rho(0)) * y;
  w = (1. / xi(0)) * w_tld;
  z = (1. / xi(0)) * z;

  delta(0) = dot(z, y);
  if (delta(0) == 0.0)
    return 5;

  y_tld = M2.solve(y);
  z_tld = M1.trans_solve(z);

  if (i > 1) {
    p = y_tld - (xi(0) * delta(0) / ep(0)) * p;
    q = z_tld - (rho(0) * delta(0) / ep(0)) * q;
  } else {
    p = y_tld;
    q = z_tld;
  }

  p_tld = A * p;
  ep(0) = dot(q, p_tld);
  if (ep(0) == 0.0)
    return 6;

  beta(0) = ep(0) / delta(0);
  if (beta(0) == 0.0)
    return 3;

  v_tld = p_tld - beta(0) * v;
  y = M1.solve(v_tld);
  rho_1(0) = rho(0);
  rho(0) = norm(y);
  w_tld = A.trans_mult(q) - beta(0) * w;
  z = M2.trans_solve(w_tld);
  xi(0) = norm(z);

```

```
    gamma_1(0) = gamma(0);
45  theta_1(0) = theta(0);
    theta(0) = rho(0) / (gamma_1(0) * beta(0));
47  gamma(0) = 1.0 / sqrt(1.0 + theta(0) * theta(0));

49  if (gamma(0) == 0.0)
      return 4;
51
    eta(0) = -eta(0) * rho_1(0) * gamma(0) * gamma(0) /
53           (beta(0) * gamma_1(0) * gamma_1(0));

55  if (i > 1) {
      d = eta(0) * p + (theta_1(0) * theta_1(0) * gamma(0) * gamma
        (0)) * d;
57    s = eta(0) * p_tld + (theta_1(0) * theta_1(0) * gamma(0) *
        gamma(0)) * s;
    } else {
59    d = eta(0) * p;
      s = eta(0) * p_tld;
61  }
    x += d;
63  r -= s;
    if ((resid = norm(r) / normb) <= tol) {
65    tol = resid;
      return 0;
67  }

69  tol = resid;
    return 1;
71 }
```

Listing A.6 The QMR method for solving the symmetric positive-definite system $A \cdot \vec{x} = \vec{b}$

Appendix B

Exploring Deep Learning Models for Silent Data Corruption in Scientific Kernels: A Case Study

This study was conducted as a joint work with another colleague from Pacific Northwest National Laboratory. This work is provided as an example usage of the data collected in this thesis being used in other studies.

As exascale computing takes off, the probability of soft/silent errors inside scientific kernels also increases. To help with this problem, several efforts in detecting and correcting such errors in scientific workloads have taken place. However, because of the size and complexity of the workloads, errors might have different effects on the computation which can be positive (earlier termination) or negative (incorrect results). Collecting and processing all this data is a herculean task since the error data is vast and multi dimensional. Due to its ability to handle such data, machine learning is a promising technique for this type of application. This paper presents a study of using soft error data to build a deep neural net model and predicting the behavior of the application (as anomalous or un-affected) for an iterative solver kernel: Conjugate Gradient.

B.1 Introduction

Supercomputers are vast collections of distinct components that efficiently interact with each other in order to gain scientific insight. However, with such vast number of components, the probability that a undetected error might creep into the computation increases. Such errors are known as soft errors (named because they are transient and do not represent a

permanent failure on the computational system). Due to this, there have been a Renaissance on research on how to detect and to correct these spurious errors. Several routes have been attempted on solving these problems. Applications might select more hardy computational kernels to mitigate the errors impact on their simulations; Runtime systems might incorporate sophisticated detection mechanisms to detect and to correct these faults; and hardware might protect key components (like caches and memory) with error correcting codes (ECC). As of late, some of the major techniques used to predict these types of faults come from the machine learning (ML) field, including support vector machines, random forests and deep neural networks [72].

However creating network topologies for machine learning is a fuzzy process. Currently, it is considered an art in which in-depth knowledge of the problem and data characteristics are necessary or a design space exploration is required to find the correct combination of features and network characteristics. Although new techniques such as reinforcement learning [43], deep belief [69] networks and other semi supervised and unsupervised ML techniques can help in this process, they can be a bit cumbersome to understand and use effectively with certain data types. Nevertheless, current ML frameworks offer rapid prototyping features that allow this exploration to take place seamlessly.

This paper presents a study on how to build a neural network for a soft error dataset. We acquire this dataset and extracted relevant features such data specific characteristics and runtime behaviors that enhances the learning ability of the neural net. We tested different batch sizes, topologies, network sizes and gradient updates optimizers and found out that with minimal tweaking we can achieve up to 90% accuracy in our testing set using a Multi Layer Perceptron (MLP) deep neural network.

B.2 Related Work

Because of the importance of machine learning and fault detection, there have been several efforts that aim to married them. Below, there are a few examples of how ML techniques are being used in the field of fault detection and in High Performance Computing in general.

The MAchine-learning-based silent data CORruption Detection framework (MACORD) [72] is a dynamic framework for detecting silent data corruption in High Performance Computing kernels. It uses a “zoo” of machine learning techniques and select adaptively the best fit at runtime. Based on their experiments, they show impressive recall and precision numbers for real-world scientific applications. This framework lacks a deep neural network component and our research could fit into it as part of the zoo of applications.

The IPAS framework [48] is an instruction duplication framework implemented in the LLVM compiler infrastructure. It starts with the premise that some of the soft / silent errors are naturally masked by the algorithms and architectures. Thus, only the subset with visible effects in the outcome of the scientific kernel need to be considered. This framework uses machine learning and compiler analysis to learn / predict about possible non-benign errors and how to identify vulnerable regions. They used a support vector machine formulation. Because of the way that the framework is structured, a deep neural net can be used as an engine to the framework to increase its accuracy.

The CrystalBall framework [88] is another compiler based framework that uses Long Short Term Memory networks (LSTM) to statically identify hot paths (the most likely visited paths) on the logic structure of the program (expressed as chains of basic blocks). Although the metric in this case is performance, the same concepts can be applied to silent data corruption detection and vulnerability region selection.

B.3 Methodology

To create our experimental setup, we acquire soft error data from a scientific kernel: a Krylov space iterative solver. This data is separated into training and testing sets with training being a small fraction (around 5%) of the total data set randomly selected. we decided for this setup since larger fractions will results in longer training times with very little gain on training/testing accuracy. We created a special case in which 80% of data was used for training and 20% for testing and we achieved similar accuracy but the execution time blow up to 10x times. The collection of the data and the network design will be discussed in this section. However, we should revisit some general basic concepts beforehand.

B.3.1 ML Basic Concepts

During the progression of this paper, we use certain terms that should be familiar to the machine learning crowd. We collected these terms in this section for easy reference and clarification. The neural network is composed of one input and one output layers and one or more *hidden layers*. The output layer usually ends up spitting up a “guess” to a series of bins or categories. If the guess is correct, it counts to a positive accuracy metric. If it is not, it decreases it. There exists many types of hidden layers with different purposes like *convolutions* for extracting information from images with locality, *pooling* layers that reduces the information from layer to layer or *fully connected* layers that have all connections are active. The main operation inside the neural network involve weights and bias and a linear

algebra operation (usually matrix vector multiply). Each layer can have different number of *neurons* (the basic unit of the network in which the weights and bias vectors live) and we refer to this as the height of the layer. The topology of the network is composed of the type of layers, the height of the layers and the number of them across the length of the network. A network learns when it is exposed to data, followed by a comparison with ground truth and then any corrections or updates are propagated through the network (in a process called *backpropagation*). The update rules are controlled by *optimizers* that implement a version of the gradient descent algorithm to find minima on an optimization problem. There are many enhancements to the usually gradient descent algorithm and the optimizers take advantages of this, like momentum which is the mathematical equivalent of accelerating the gradients based on gradient history, or using the data geometry and the gradient's statistical properties across its history. The phase during which a network is learning is called *training* and it is usually accomplished using a relevant dataset that is representative of data yet to come. If a network has a pathological behavior, such as having a very high miss predicting rate when new data is presented, we said that the network *over fitted* and it requires more data to train. During the *testing* phase, never-before-seen data is presented to the network and metrics such as accuracy are calculated to see the effectiveness of the network.

Finally, a deep neural network can be parallelized (distributed) in two fashions. Under *data parallelism*, the network model is replicated across the computational resources and the batches and total problem are divided across the available resources. This allow fast distribution and easy parallelization but it limits the size of the networks that can be parallelized. On the other hand, under *model parallelism*, parts of the model are distributed across the computational resource. Due to how the model is partitioned, the time and/or accuracy might require adjustment to achieve the given constrains.

B.3.2 Injection Data

Using a sequential implementation of the Conjugate Gradient algorithm [64] and its pseudocode is shown in Algorithm 1, we do a source level error injection in selected vectors using a predefined probability density function (i.e. Beta distribution for these experiments). We selected four data sets from the University of Florida Sparse Matrix Collection [21]. Using these matrices, we collected the entire trace execution for the CG kernel with the following metadata: current residual error, status of key data structures, and iteration number. An entry is marked as anomalous if the execution would result in a different convergence behavior than a previous calculated golden value. A full description of the creation of the error data can be found in [46] and a pictorial description is shown in Figure B.1.

Algorithm 1 Preconditioned Conjugate Gradient

procedure CG(A, x, b, M, $maxIter$, tol)
  $normb = \|b\|$
  $r = b - A * x$
  if normb == 0.0 **then**
    $normb = 1$
  if $(resid = \|r\|)/(normb) <= tol$ **then**
    $tol = resid$
    $maxIter = 0$
    return 0
  for i = 1 .. maxIter **do**
    $z = solve(M, z)$
    $\rho = r \bullet z$
    if i == 1 **then**
      $p = z$
    else
      $\beta = \rho/\rho_1$
      $p = z + \beta * p$
    $q = A * p$
    $\alpha = \rho/(p \bullet q)$
    $x += \alpha * p$
    $r -= \alpha * q$
    if $(resid = \|r\|/normb) <= tol$ **then**
      $tol = resid$
      $maxIter = i$
      return 0
    $\rho_1 = \rho$
  $tol = resid$
  return 1

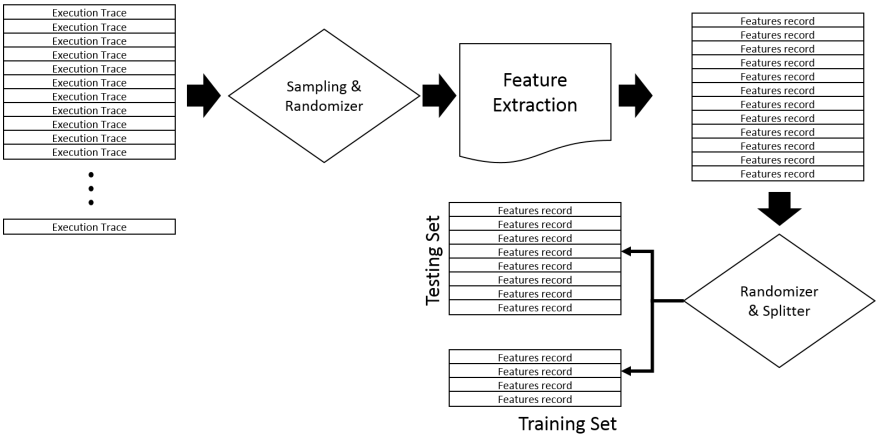


Fig. B.1 Data collection and sampling methodology

After the entire set is collected, the traces are randomized and sampled and selected a 11 million points set for our network tests which we divide into training (5%) and testing sets (the rest). The sets are created from the error data by analyzing the trace's behavior and extracting the set of features to be used by our neural net. We divide these features into roughly three categories that will be explored next.

B.3.3 Feature Selection and Network Topology

We have identified 15 distinct features to be used in our deep neural network methodology. Roughly they can be divided into three categories: application based, dataset based and runtime based features. The application-based set has been used in the literature to compare other machine learning detector methods [46]. Under this category, we use the norm of the residual vector [52], the checksum values that encode matrix-vector multiplication and vector linear operations [76], and the orthogonality relationships between certain computational vectors [14]. In the case of the orthogonality relationship, this metric is inherent only to the type of iterative solver that is used and might not be true in other types of solver such as Biconjugate Gradient, Biconjugate gradient stabilized, Quasi Minimal Residual methods, etc.

In the case of runtime based features, we see an experiment's trace as a sequence of iterations and consider the rate of change or other second order characteristics of important application's variables across a pre-selected iteration window. For a visual representation, please refer to Figure B.2 which presents the residual values of a solver execution over the loop iterations. For our experiments, we used the residual minimum (Min) and maximum (Max) over the window, the rate of change over such interval ($\delta r/\delta i$) and the difference between current residual and the target residual (Diff) (i.e. convergence condition). These values describe the application state over a time window during its execution. In this fashion, we give the network a poor man's version of recent events memory in the interval. The selected interval for our current experiments is four running iterations of the current run.

Finally, in the case of data set characteristics, we use the dataset properties such as workload id, number of non zeroes entries in the matrix, minimum eigen value and the matrix norm. These properties and the matrices comes from the University of Florida Sparse Matrix Collection [21] and are listed in Table B.1.

The collected data represents a high dimensional set since each entry has around 15 associated data points that are used by the neural net to guide its learning. Previous experiments [46] with a reduced set of features showcases an accuracy in the higher 70% for a similar type of network. In this paper, we showcase that the addition of the new features increases

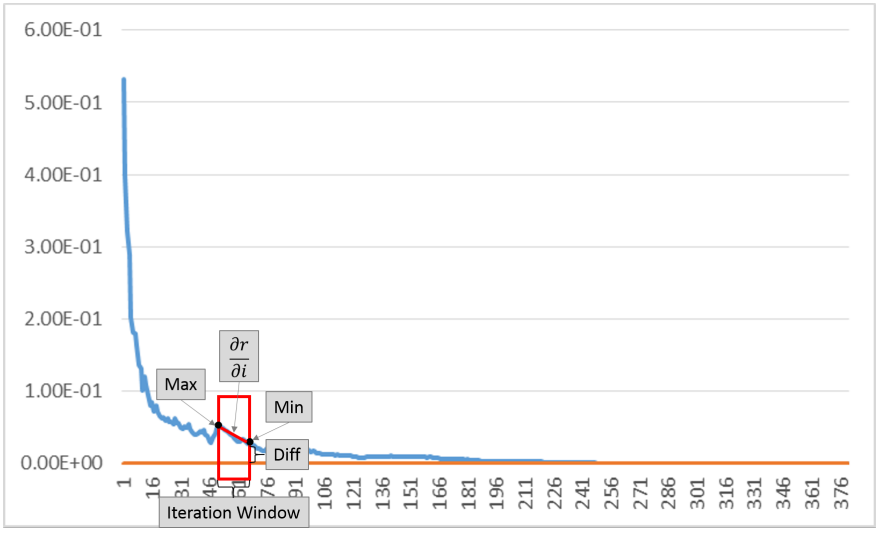


Fig. B.2 Example of Runtime features selection over a solver execution. X axis represents the iteration number and Y is the residual number across the execution.

Table B.1 Data Set characteristics and description. The NNZ ratio refers to the number of non zeroes over the size of the matrix. The Norm column is the normal of the sparse matrix. The min(SVD) column is the minimum Single value decomposition. Finally, the Cond column is the condition of the sparse matrix

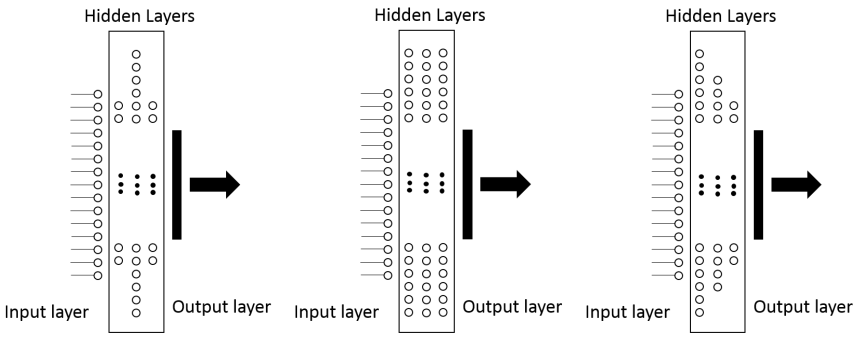
ID	Name	Description	NNZ Ratio	Norm	Min (SVD)	Cond
0	BCSSTK15	Stiffness Matrix. Module of an offshore platform	0.008	6.5e9	1	6.5e9
1	Kuu	The MathWorks Inc.: Symmetric positive definitive matrix	0.007	54.1	0.003	15758
2	Press_Poisson	ACUSIM, Inc: computational fluid dynamics problem	0.003	26.02	1.3e-5	2.04e6
3	STS4096	Structural Engineering matrix (finite-element), Fabio Cannizzo	0.004	3.1e8	1.42	2.17e8

this number and the selection of the optimizer also has an impact on the speed and accuracy of the net.

We chose a Multi Layer Perceptron (MLP) with an input and an output layers and 3 hidden layers with different configurations in the number of neurons. The output layer is a binary classifier and the activation function is ReLu based. Among the configurations, we have one network with fully connected layers with the same number of neurons per layer. This network is called the "rectangular" topology. Another network has half the neurons in the first and third hidden layers and we called it the "diamond" topology. In the final selected network, each layer has its number of neurons halved as it goes. We called this topology the "triangle" topology. Figure B.3 illustrates these networks.

The rationale behind these designs is that for these networks, information might be systematically reduced over the length of the network so it might require less neurons (hence the triangle). On the other hand, more details or features might emerge during the length of the network and collapse again (hence the diamond). The rectangle topology is the baseline network with full layers. Each topology has an impact on the actual size of the network and its variables (hence its memory footprint). For example, in the case that the network is using 256 neurons as its largest layer, the rectangle topology has *135,891* trainable variables, the diamond one has *68,179* trainable variables and the triangle has *45,267* trainable variables (based on models implemented using the Keras API). Finally, we did not consider convolution layers since the data that we are using is text and lack the temporal characteristics that would be advantageous for such layers.

As extra steps in our scripts, we call a normalization function to make sure that all the input data is scaled to the unit norm, we added extra functions to adaptively reduce learning rate and invoke early termination to the models to reduce the training time.



(a) Diamond (b) Rectangle (c) Triangle

Fig. B.3 Different network topologies

B.4 Experimentation and Analysis

Using the parsed data, we conducted several sets of experiments using high performance clusters. Machine learning frameworks are notoriously data intensive and good candidates for parallel execution. However, the complexity of the data might not be adequate to scale up in large clusters. Our data sets have relative high dimensions. However, the current sets are not large enough to justify hundreds of computer nodes. For this reason, we selected two small High Performance clusters to showcase their promise for scalability. We start by exploring several combinations of batch sizes, number of neurons and optimizers. Then we select the best testing accuracy candidates and parallelize these cases using a data parallel model. We describe our software and hardware tested in this section and then we showcase our results for testing and training accuracies, together with the timing per epoch for the selected networks. Finally, we present the scalability of the selected networks using the user transparent Tensorflow included in the MaTEx [82] framework.

B.4.1 Hardware and software infrastructures

For our hardware platform, we use two HPC clusters. The first one, named PUMA, consists of 8 Pascal GPUs (P100) with Intel Xeon E5 2680 cpus as their hosts (one GPU per CPU host). The Intel CPUs have two sockets with 10 physical cores per sockets. This cluster's memory hierarchy consists of 32 KiB for L1 cache, 256 KiB for L2 cache and 25 MiB of L3 cache with 720 GiB of main memory per node. The second cluster, named MARIANAS, consists of 25 nodes each with 2 P100 GPUs. However, we use 4 nodes with 2 GPUs each for our experiments for comparison. The GPUs in these clusters are identical to the ones in the PUMA testbed, but the hosts differ quite a bit. There are two sockets with only 8 cores each with a single hardware thread for the Intel CPU host. In the case of the caches, the first two levels are identical but the third one is only 20 MiB. Finally, the main memory in each of the Marianas nodes is only 64 GiB. Both clusters uses Infiniband as their network fabric.

As our software testbed, we used the MaTEx Tensorflow version 1.5 with the Keras package version 1.2.2 [82]. This current version uses CUDA 9.0 and CUDNN version 7 that are optimized to take advantage of the Pascal GPUs. This version of Tensorflow has a user transparent MPI layer that is optimized for HPC clusters. This framework uses the data parallel model of neural network execution in which copies of the model are replicated across the computing resources and the data batches are divided and distributed across these resources. The underlying framework takes care of the distribution and communication of data and results across the cluster and ensures that the final model is the same as if it was

run on a single node. A graphical representation of how a net would run under the MaTEx framework is illustrated in figure B.4.

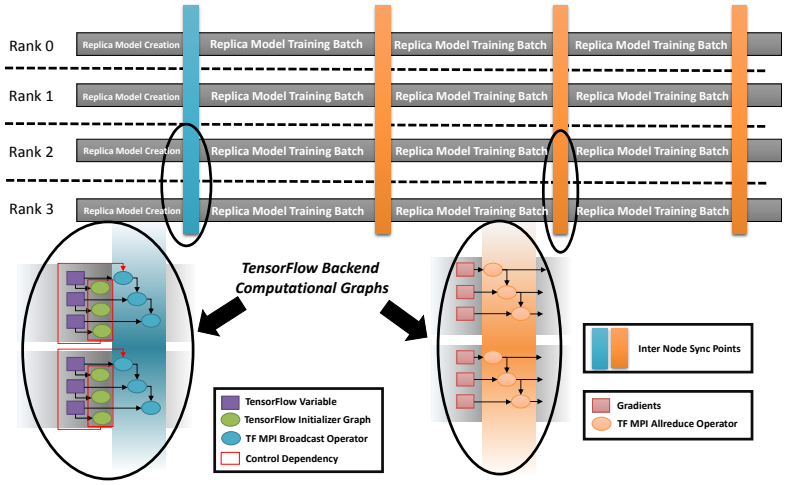


Fig. B.4 User Transparent Distributed Tensorflow Design

B.4.2 Accuracy numbers for Training and testing

For this set of experiments, we explored different sets of batches (16, 32, 128 and 256) and network sizes (128, 256 and 512) to find the best accuracy numbers for both testing and training. Moreover, the experiments show that in some of the exploration space the optimizer have a great impact on the final accuracy. However, before we jump into the results, we need to have a word about optimizers.

Selected Optimizers

We are going to be testing all the default optimizers on the Tensorflow framework with their default values. In this subsection, we will briefly touch on each of the optimizers and how they build upon each other.

The Stochastic Gradient Descent (**SGD**) optimizer is used to minimize an objective function on an optimization problem using an approximation of the Gradient Descent algorithm. It is an iterative method which tries to find local maxima or minima at each iteration. In practice, the optimizer is sensitive to its initial condition (such as learning rate) and requires careful tuning. The ADADELTA (**DELT**) optimizer [87] builds on top of the vanilla stochastic gradient descent method and it dynamically adapts using only first order statistical information. The ADAGRAD (**GRAD**) optimizer [28] incorporates the knowledge of data geometry gained from previous iterations and adapts it to gradient based learning process through

proximal functions. In the case of the Adaptive Moment Estimator (**ADAM**) optimizer [47], it calculates the learning rates for distinct parameters using estimates of first (mean) and second (variance) moments of the gradients. The ADAMAX (**MAX**) optimizer [47] is an extension to ADAM in which the update rule uses the infinite norm instead of the L2 norm. Advantages includes a simpler bound for the magnitude of parameter updates and there is no need to correct for the initialization bias. The NADAM (**NADA**) optimizer [27] is just ADAM with a Nesterov Momentum which accelerates convergence behavior in case of local minima. Under the RMSProp (**RMS**) optimizer [39], the weight's learning rate are divided by a running average of the recent historical gradients.

Accuracy for Testing and Training

As described in Section B.3.2, we use a small training set with a large testing set to test the optimizers provided by Tensorflow. On our tables, the **BS** title corresponds to Batch Size, **NS** is the size of the largest layer, **TS** is the testing accuracy and **TR** is the training accuracy.

As shown in Table B.2, the best optimizer for the diamond topology is the ADAM one with similar accuracies in both training and testing (with 83.9% and 89.9% respectively) and with a batch size of 16 with a network size of 256. However, in the case of other optimizers, they ranges in the 70s and 80s with the exception of the Stochastic Gradient Descent one that stay around 50% due to ill-chosen learning rate.

Table B.2 Accuracy for both testing and training phases of the diamond network for different Tensorflow's optimizers

		DELT		GRAD		ADAM		MAX		NADA		RMS		SGD	
BS	NS	TS	TR	TS	TR	TS	TR	TS	TR	TS	TR	TS	TR	TS	TR
16	256	64.3	83.8	58.3	83.9	89.9	83.9	60.3	83.9	88.8	83.8	50.2	56.9	50.0	58.3
16	512	58.4	83.7	75.3	83.9	88.8	83.9	89.5	83.9	80.6	83.9	50.2	56.9	50.0	56.8
32	256	63.8	83.8	56.7	83.6	79.7	83.9	89.9	83.9	68.8	83.8	50.2	73.3	50.0	56.7
32	512	58.4	83.8	68.1	83.9	84.3	83.8	88.3	83.9	71.6	83.9	50.2	56.9	50.0	56.7
64	256	63.0	83.8	57.9	83.8	57.7	83.7	89.6	83.9	63.5	83.8	63.2	83.8	50.0	56.8
64	512	70.5	83.8	64.9	83.9	70.7	83.7	75.3	83.9	71.3	83.7	50.2	56.9	50.0	56.8
128	256	58.4	83.7	56.0	83.5	56.4	83.4	64.1	83.9	58.4	83.8	50.2	56.9	50.0	56.8
128	512	73.0	83.7	68.3	83.9	66.4	83.9	83.6	83.9	61.0	83.8	55.1	83.7	50.0	56.8
256	256	56.7	83.6	55.2	83.4	55.7	83.5	61.0	83.7	54.9	83.3	57.1	82.9	50.0	56.8
256	512	79.6	83.6	67.6	83.7	69.3	83.8	59.4	83.8	59.7	83.5	63.8	82.7	50.0	56.8

For the diamond network, as the batch size increases, the testing accuracy decreases across all optimizers, with the exception of ADADELTA. As shown in Table B.3, the network size has small impact on the runtime, but the batch size does have a significant impact

since doubling the batch size decreases the runtime almost by half (on a single node). This showcases the impact that locality has on the execution time, the cost of memory transfers and how the framework overhead affects the runtime. However, at least for this topology, smaller batch size are beneficial for training with our data set so we should take the hit on runtime.

Table B.3 Timing per phase in seconds for the best batch / network size for each optimizer for the diamond topology

BS	NS	DELT	GRAD	ADAM	MAX	NADA	RMS	SGD
16	256	174.1	133.6	166.1	151.6	178.5	105.3	124.6
16	512	178.6	145.2	174	150.2	186.7	83.5	121
32	256	87.6	66.9	84.6	75	89.7	70.7	61.6
32	512	90.4	69.7	85.1	76.9	92.2	65.8	62.5
64	256	43.6	35.5	45	37.3	46.9	35.5	30.4
64	512	44.5	35.5	42.1	39.1	45.4	34.8	31
128	256	22.0	16.6	24.9	19.3	22.9	18.2	15.7
128	512	22.8	17.5	25.4	20.1	23.8	18.5	15.7
256	256	11.2	8.5	10.7	9.4	11.2	9.3	7.6
256	512	11.2	9.3	12.6	10.2	12.1	9.4	8.2

In the case of the rectangle topology, Table B.4 showcase ADAM as the clear winner in terms of testing accuracy (with 89.83%) with ADAMAX and ADAGRAD being second and third. In the case of the RMSProp, over fitting might seem to be an issue since the training provide promising results but the testing accuracy is abysmal (around 57%). As seen in the diamond topology, the SGD optimizer remains last.

As in the case of diamond network, the rectangle network showcases (c.f. Table B.5) the same timing per epoch behavior in which doubling the size of batch results in a reduction in timing per set (due to better use of resources and low framework overhead).

Finally, the triangle topology exhibits the best testing accuracy with 90% but it showcases losses in optimizers like ADADELTA (67.7%) and RMSProp (73.5%). As expected, SGD showcases the same behavior as in the other topologies. In accordance with previous tables, Table B.7 shows the same behavior as the other two in which doubling the batch size reduces the runtime.

Across the board we see that the ADAM and its enhancements provide a better accuracy than the SGD and RMSProp based ones. However, these numbers might be a bit misleading since the SGD optimizer is very sensitive to initial conditions and it will require a tunning process to find the optimal learning rate (if one exists). Even then, the selected optimizer do a good job in predicting the behavior of the application in the presence of faults with our data set.

Table B.4 Accuracy for both testing and training phases of the rectangle network for different Tensorflow's optimizers

		DELT		GRAD		ADAM		MAX		NADA		RMS		SGD	
BS	NS	TS	TR	TS	TR	TS	TR	TS	TR	TS	TR	TS	TR	TS	TR
16	256	58.0	83.8	84.2	83.9	80.5	83.9	75.6	83.9	58.7	83.7	50.2	56.9	50.0	57.1
16	512	80.8	83.7	83.7	83.8	79.4	83.8	56.5	83.9	75.5	83.8	50.2	56.9	50.0	56.8
32	256	59.4	83.8	66.8	83.9	84.4	83.8	87.0	83.9	77.2	83.8	50.0	56.9	50.0	56.7
32	512	59.7	83.8	79.2	83.9	68.3	83.8	89.7	83.9	59.7	83.8	50.2	56.9	50.2	56.7
64	256	71.0	83.8	58.5	83.9	89.8	83.9	87.3	83.9	76.3	83.8	50.2	56.9	50.0	56.8
64	512	78.8	83.8	78.4	83.9	86.6	83.8	89.7	83.9	57.8	83.7	50.2	56.9	50.0	56.8
128	256	62.1	83.8	58.2	83.8	57.7	83.6	88.8	83.9	58.4	83.6	50.0	56.9	50.0	56.8
128	512	74.8	83.7	76.8	83.9	83.5	83.8	88.4	83.8	55.0	83.6	50.2	56.9	50.0	56.8
256	256	77.9	83.7	56.3	83.4	63.6	83.7	66.8	83.8	57.0	83.6	57.1	82.8	50.0	56.8
256	512	63.03	83.5	76.1	83.9	63.7	83.3	59.5	83.7	59.1	83.5	56.6	82.9	50.0	56.8

Table B.5 Timing per phase in seconds for the best batch / network size for each optimizer for the rectangle topology

BS	NS	DELT	GRAD	ADAM	MAX	NADA	RMS	SGD
16	256	173.3	133.6	162	150.9	181.8	106	121.3
16	512	187.4	146.3	175.2	165.1	193.9	131.5	136.3
32	256	86.8	70.6	84.5	75.3	89.5	69	59.8
32	512	90.7	75.1	88.1	80.5	96.7	44	67.9
64	256	43.7	34.6	42.2	39.1	46.3	32.6	31.9
64	512	46.2	37.5	45	41.7	48.2	33	32.9
128	256	22	17.5	20.6	20.2	22.9	17.17	15.7
128	512	22.9	18.4	22.4	20.2	24.7	18.17	16.6
256	256	11.2	8.5	10.7	10.3	12.1	9.4	7.6
256	512	12.1	9.4	11.5	10.3	12.1	10.1	8.5

Table B.6 Accuracy for both testing and training phases of the Triangle network for different Tensorflow'optimizers

		DELT		GRAD		ADAM		MAX		NADA		RMS		SGD	
BS	NS	TS	TR	TS	TR	TS	TR	TS	TR	TS	TR	TS	TR	TS	TR
16	256	58.5	83.8	58.7	83.9	72.3	83.9	61.7	83.9	79.7	83.8	59.5	83.9	50.0	56.8
16	512	62.2	83.9	83.4	83.9	86.0	83.8	88.6	83.9	88.8	83.9	50.2	56.9	50.0	58.6
32	256	67.7	83.8	56.9	83.5	82.5	83.8	86.7	83.9	89.7	83.8	50.0	56.9	50.0	56.5
32	512	58.0	83.8	65.1	83.9	88.8	83.9	90.0	83.9	58.7	83.7	50.2	56.9	50.0	56.5
64	256	58.2	83.8	55.8	83.6	66.7	83.8	89.8	83.9	69.2	83.8	50.0	56.8	N/A	N/A
64	512	62.1	83.8	64.0	83.9	60.1	83.7	87.3	83.9	58.3	83.7	50.2	56.9	50.0	56.8
128	256	56.4	83.7	55.2	83.5	60.9	83.6	73.7	83.8	59.5	83.3	73.5	83.7	50.0	56.8
128	512	56.5	83.7	63.6	83.8	57.0	83.5	60.1	83.9	57.3	83.8	54.6	83.5	50.0	56.8
256	256	65.4	83.7	55.1	83.4	62.8	83.7	68.8	83.8	83.4	82.8	55.5	83.2	50.0	56.8
256	512	57.4	83.7	57.0	83.5	66.2	83.5	58.0	83.7	58.3	83.4	54.9	83.3	50.0	56.8

Table B.7 Timing per phase in seconds for the best batch / network size for each optimizer for the triangle topology

BS	NS	DELT	GRAD	ADAM	MAX	NADA	RMS	SGD
16	256	169.7	137.4	161.5	148.2	176.8	141.1	120.1
16	512	175.8	135.4	165.3	148.9	186.8	107.3	122.1
32	256	85.9	66	81.9	74.7	89.5	63.2	60
32	512	88.6	68.1	83.6	76.1	93.1	60	61.6
64	256	43.1	32.8	42.1	37.3	35	30.1	N/A
64	512	44.5	34	42.8	38.2	46.3	35.5	31.6
128	256	21.7	17.4	20.6	19.2	22.9	17.5	15.6
128	512	22	17.5	21.4	19.3	23.8	18.4	15.7
256	256	11.1	8.4	10.6	9.5	11.2	9.3	7.6
256	512	11.2	8.5	21.1	9.4	12.1	9.4	7.6

B.4.3 Runtime and Scaling

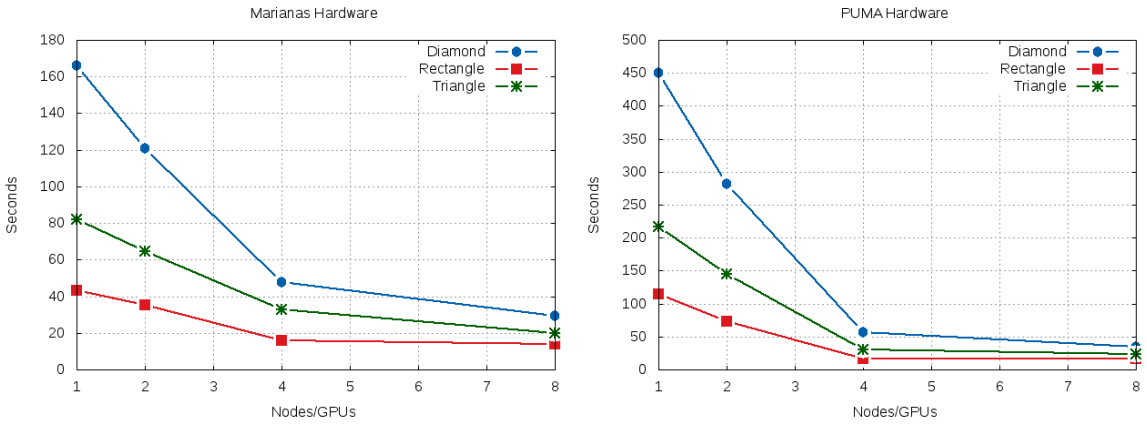
Since running smaller batches seem to be beneficial for our problem, we need find a way to accelerate our computation since the smaller batches results in the slowest time per epochs. Thus, using the batch sizes, network sizes and optimizer obtained from the last step, we take advantage of the distributed capabilities of MaTEx to accelerate our computation. To showcase this, we ran a strong scalability study over the HPC clusters. Figure B.5 shows the time per phase/epoch for each execution running on 1, 2, 4 and 8 GPUs in both clusters. In the Marianas cluster, we see that the Diamond topology has a good relative speedup on 2 nodes with 1.4x, on 4 nodes with 3.5x and on 8 nodes with 5.7x. In the case of the rectangle topology, the speedup is not as good but still shows some gains with 1.2x, 2.8x and 3.2x for 2, 4, and 8 GPUs, respectively. Under this network, it seems that communication costs are overtaking the gains of the parallelization. Finally, for the triangle topology, we still do not have as good as speedup as in the diamond case but still shows improvement and room to grow, i.e. 1.3x, 2.5x and 4.1x, respectively.

In the PUMA cluster case, an interesting behavior is evident. The single node performance is abysmal, but when we introduce new nodes the speedup grow super-linearly thanks to the increased resources in both host memory and host compute power. As before, diamond produces the best scalability behavior with 1.6x on 2 GPUs, 8x on 4 GPUs and 13x in 8 GPUs. The rectangle topology do not show any scalability after 4 GPUs, topping up at 7x on 4. Finally, triangle showcases a small increase in speedup between 4 and 8 (7.3x to 9.3x). Thus, in PUMA with these batch sizes, there seems not be a reason to increase the nodes. However, by introducing the new resources in the 2, 4 and 8 cases, we match the performance of the Marianas cluster due to the extra resources.

Using these data, we can infer that the diamond based topology is the best one to conduct scalability studies with our dataset while still providing a high degree of accuracy for both testing and training regimes.

B.5 Conclusions and Future Work

As showcased in this experiments, the network topology does have an impact on the model result but the actual main aspect is the selection and tuning of the batch size and the selected optimizer. As is well known, ADAM and its ilk are very well suited for start-fast prototyping which it seems that it is not a characteristic of the SGD optimizer. As future work, we can collect a much bigger dataset. We have access to larger dataset containing more iterative solvers, error distribution and data structures that rounds up to tera bytes of data. Thus, the need for parallel and distributed execution. The MLP topology might not be an ideal one for



(a) MARIANAS (b) PUMA

Fig. B.5 Scaling results for the different network topologies on two HPC clusters

distributed computation due to its large communication versus computation ratio; although it has some scalability opportunities, it is easily overwhelmed by communication costs as more nodes are added. Further investigation in new types of layers and how the data can be transformed to better suit their needs is also planned to take place in the near future. This paper presented a small study on using a MLP network to predict a HPC kernel behavior in the presence of faults. We ran several scenarios between batch sizes, network topologies and layer sizes to hone in a suitable configuration. Moreover, due to its extensive data and computing needs, we briefly explore of this specific network would make a good candidate for parallelization. We ended up with a configuration that gives us around 90% accuracy in our test set. However, this is just the first step of a larger exploration study with larger data sets and more computational needs.