ABSTRACT

ELLIOTT III, JAMES JOHN. Resilient Iterative Linear Solvers Running Through Errors. (Under the direction of Frank Mueller.)

Future extreme-scale computer systems may expose incorrect behavior to applications, in order to save energy or increase performance. However, resilience research struggles to come up with useful abstract programming models for reasoning about faults in applications. This work mainly focuses on silent soft errors, that is, errors that do not cause the system to halt and provide no indication that they occurred. The approach presented is not specific to silent soft errors; it is a general model for tolerating abnormal behavior in numerical algorithms. We present findings targeted at silent faults that impact the data used by an algorithm. Silent faults in data, which we refer to as silent data corruption (SDC), may lead to the algorithm operating on incorrect data and presenting invalid outputs. The overarching goal of this work is to ensure that we obtain valid solutions given soft faults.

Existing work in algorithm fault tolerance randomly flips bits in running applications, but this only shows average-case behavior for a low-level, artificial hardware model. Algorithm developers need to understand worst-case behavior with the higher-level data types they actually use, in order to make their algorithms more resilient. Also, we know so little about how soft faults may manifest in future hardware, that it seems premature to draw conclusions about the average case. We argue instead that numerical algorithms can benefit from a numerical unreliability fault model, where faults manifest as unbounded perturbations to floating-point data. Algorithms can use inexpensive "sanity" checks that bound or exclude error in the results of computations. Given a selective reliability programming model that requires reliability only when and where needed, such checks can make algorithms reliable despite unbounded errors. Sanity checks, and in general a healthy skepticism about the correctness of subroutines, are wise even if hardware is perfectly reliable.

© Copyright 2015 by James John Elliott III

All Rights Reserved

Resilient Iterative Linear Solvers Running Through Errors

by
James John Elliott III

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2015

APPROVED BY:

Jon Doyle	Blair Sullivan
Ralph Smith	Mark Hoemmen

Frank Mueller
Chair of Advisory Committee

DEDICATION

To my parents, sister, and baby Zoie.

BIOGRAPHY

James Elliott was born in Shreveport, Louisiana, and grew up on a rural farm. Both of Jamesâ parents are schoolteachers, who left the public school system to open their own preschool, the Summit Nursery School. James attended and then worked at the Summit for most of his childhood. Enthralled by technology, James owes a debt of gratitude to Cheryl Vines who taught him the basics of programming at Saint Mark Cathedral School in a third grade computer science class. The integration of computer science, not gadgetry, into his early education was incredibly impactful. James is a strong proponent of technology education, and a strong opponent of gadgetry shoehorned into the educational system.

James graduated from Louisiana Tech University with a B.S. in Computer Science, a M.S. in Mathematics and Statistics, and a Ph.D. in Computational Analysis and Modeling. James has been involved with the fields of HPC and resilience since 2005, where he studied how to virtualize a cluster using the Xen hypervisor. In 2007, he integrated various benchmarks into the OSCAR cluster management suite as part of a Google Summer of Code project. He worked directly with the Louisiana Optical Network Initiative as a graduate computational science fellow in 2009, where he developed, among other things, an OpenCL implementation of differential quadrature. James transitioned to North Carolina State to pursue a Ph.D. in Computer Science, where he has studied alternatives to checkpoint/restart and soft error resilience. A strong component of Mr. Elliottâs work is the use of analytic modeling, and one day he hopes to demystify the "monster in the closet," that is soft error resilience. Mr. Elliott has also taught at the middle and high school levels as part of the NSF GK-12 Teaching fellowship.

ACKNOWLEDGEMENTS

I would like to thank Frank Mueller for his support and open-mindedness. I also owe a debt of gratitude to Mark Hoemmen for his mentorship and for finding time when I know his is precious.

TABLE OF CONTENTS

LIST OF TABLES . . . . . **ix**

LIST OF FIGURES . . . . . **x**

Chapter 1 Introduction **1**
1.1 Resilience 2
1.2 Resilience and Fault Tolerance 4
1.3 Detecting Errors and Selective Reliability 5
1.4 Resilient Sparse Preconditioned Linear Solvers 8
1.5 Research Hypothesis 10
1.6 Overview 10
1.6.1 Bit Flip Model May Not Reflect Actual Hardware Behavior 11
1.6.2 Skeptical Programming: Bounding Soft Errors 11
1.6.3 Worst-Case Behavior: Adversarial vs. Practitioner 12
1.6.4 Abstract Fault Models Are Superior 13
1.7 Summary of Contributions 13

Chapter 2 Data Representation and Fault Tolerance **15**
2.1 Introduction 15
2.2 Related Work 16
2.3 Project Overview 18
2.4 Fault Model 18
2.4.1 Fault Characterization via Semantic Analysis 19
2.4.2 Fault Characteristics of Perturbed Exponents 21
2.4.3 Operation Centric Fault Model 24
2.5 Fault Model Evaluation 25
2.5.1 Faults in the Mantissa or Sign 27
2.5.2 Modeling Large Vectors 28
2.5.3 Summary 29
2.6 Case Study: Vector Dot Products 29
2.6.1 Computational Challenges 30
2.6.2 Monte Carlo Sampling 30
2.6.3 Per Bit Analysis of Surface Plot 32
2.6.4 Comparison of the Analytic Model and Monte Carlo Sampling 32
2.7 Extension to Matrices and Iterative Solvers 32
2.7.1 Matrix Equilibration 33
2.7.2 GMRES 33
2.7.3 Instrumentation and Evaluation 34
2.7.4 Results 36
2.7.5 Multiple Bit Flips 38
2.8 Conclusion 38

Chapter 3 Model Driven Analysis of Faulty IEEE-754 Scalars 40
3.1 Errors in IEEE-754 Representation 42
3.1.1 Mantissa 44
3.1.2 Exponent 45
3.1.3 Sign 46
3.2 Model Statistics 46
3.2.1 Mantissa 47
3.2.2 Mantissa Errors 49
3.2.3 Exponent 51
3.2.4 Exponent by Range 54
3.2.5 Exponent Errors 55
3.2.6 Summary of Scalar Statistics 58
3.3 Model Error Measures 58
3.3.1 Mantissa 59
3.3.2 Exponent 60
3.3.3 Sign 60
3.4 Scalar Expected Error Measures 61
3.4.1 Expected Relative Error for a Scalar 61
3.4.2 Expected Absolute Error for a Scalar 65
3.4.3 Summary of Scalar Error Statistics 71
3.5 Model Multiplication Error Measures 72
3.5.1 Mantissa 72
3.5.2 Exponent 73
3.5.3 Sign 73
3.5.4 Multiplication Expected Error 74
3.6 Applications to Fault Tolerance 74
3.6.1 Constrained Exponent Bit Flips 75
3.7 Overall Effect of Constrained Exponent Bit Flips 76
3.7.1 Expected Absolute Error Given a Scalar 79
3.7.2 Expected Absolute Error Given Scalar Multiplication 79
3.7.3 Probability of an Absolute Error Larger Than One 84
3.8 Application of Constrained Exponent Bit Flips 85
3.9 Conclusion 86

Chapter 4 Evaluating the Impact of Silent Data Corruption on the GMRES Iterative Solver 87
4.1 Introduction 87
4.1.1 Silent Data Corruption 88
4.1.2 Faults, Failures, and Persistence 88
4.2 Project Overview 89
4.2.1 Assumptions and Justification 90
4.3 Motivation 90
4.3.1 Relation to Prior Work 91
4.3.2 Invariants as Detectors 92
4.4 Sandbox Reliability 93

4.5 GMRES . . . 94
4.5.1 Fault Detection via Projection Coefficients . . . 95
4.5.2 Bounds on the Arnoldi Process . . . 95
4.5.3 Bound Application . . . 96
4.5.4 Error Detection . . . 97
4.6 FT-GMRES . . . 97
4.6.1 FT-GMRES is Based on Flexible GMRES . . . 97
4.6.2 Sandbox Reliability . . . 99
4.6.3 FGMRES' Additional Failure Modes . . . 99
4.6.4 Fault Tolerance via Regularization . . . 100
4.7 Results . . . 101
4.7.1 Sample Problems . . . 101
4.7.2 Time to Solution Experiments . . . 103
4.7.3 Faults in an SPD Problem . . . 104
4.7.4 Faulting in a Nonsymmetric Problem . . . 106
4.7.5 Summary of Findings . . . 107
4.8 Conclusions . . . 108

Chapter 5 A Numerical Soft Fault Model for Iterative Linear Solvers . . . 109
5.1 Introduction . . . 109
5.2 Preconditioned Linear Solvers . . . 110
5.2.1 Soft Faults and Iterative Methods . . . 110
5.2.2 Selective Reliability . . . 111
5.2.3 Implementation . . . 111
5.3 Results . . . 111
5.3.1 Methodology . . . 111
5.3.2 Model Comparisons . . . 112
5.3.3 Computational Effort . . . 113
5.3.4 Expected Overhead Comparison . . . 113
5.4 Conclusion . . . 116

Chapter 6 Selective Reliability and Preconditioned Iterative Linear Solvers . . . 117
6.1 Introduction . . . 117
6.1.1 Fault-Tolerant *Preconditioned* Solvers . . . 118
6.1.2 Related Work . . . 118
6.2 Preconditioned Linear Solvers . . . 119
6.2.1 Selective Reliability . . . 121
6.2.2 Implementation . . . 122
6.3 Preconditioners . . . 122
6.3.1 Algebraic Multigrid . . . 122
6.3.2 Hierarchy and Corruption . . . 123
6.4 Fault Model and Injection Methodology . . . 123
6.4.1 Corrupting Preconditioner Outputs . . . 123
6.4.2 Granularity of Faults . . . 124
6.4.3 SDC and Solvers . . . 124

6.5 Experiment Description . . . 124
6.5.1 Methodology . . . 124
6.5.2 Problem Specification . . . 125
6.5.3 Baseline and Preconditioner Effectiveness . . . 125
6.5.4 Solver Configuration . . . 126
6.6 Experiments . . . 126
6.6.1 Figure Guide . . . 126
6.6.2 Overhead with No Detection . . . 128
6.6.3 Residual and Projection Lengths . . . 128
6.6.4 Tuned Residual Checks . . . 129
6.6.5 Strict Convergence Checks . . . 131
6.6.6 Inexact Krylov . . . 133
6.7 Comparisons . . . 133
6.7.1 ABFT Cholesky Comparison . . . 135
6.7.2 Self-Stabilizing CG . . . 137
6.7.3 Preconditioner Applies and Floating-Point Operations . . . 137
6.7.4 Relation to Bit Flips . . . 138
6.8 Scalability . . . 139
6.9 Conclusion . . . 140

Chapter 7 Conclusion . . . 142

References . . . 144

LIST OF TABLES

Table 1.1 Reliability of HPC Systems 3

Table 2.1 Bit flip absolute error for a scalar λ represented using IEEE-754 double precision, with $\lambda = \lambda_{\text{exp}} \times \lambda_{\text{frac}}$. Where λ_{exp} is the exponent 2^x, and λ_{frac} is the fractional component. 23
Table 2.2 Sample Matrices 36
Table 2.3 Norms of Sample Matrices † 36

Table 3.1 Terminology 42
Table 3.2 Common IEEE-754 Implementations 43
Table 3.3 Statistics Terminology 46
Table 3.4 Expected value for components of faulty scalars. 59
Table 3.5 Error measures for scalars. 60
70table.caption.49
Table 3.7 Expected value of the relative error given a bit flip in a scalar ($\tilde{a}$). 71
Table 3.8 Expected value of the absolute error given a bit flip in a scalar ($\tilde{a}$). 72
Table 3.9 Error measures for faulty multiplication. 74
Table 3.10 Expected value of the absolute error given a bit flip in scalar multiplication ($\tilde{a} \times b$). 75
Table 3.11 Probability that the relative error will be less than one given a bit flip in a scalar. 77
Table 3.12 Probability that the absolute error will be larger than one given a bit flip in scalar multiplication with $(a, b) \in (-1, 1)$. 86

Table 4.1 Sample Matrices 102

Table 5.1 Additional preconditioner applies given no fault detection; percent additional applies in parentheses. 115
Table 5.2 Additional preconditioner applies **with reactive fault tolerance**; percent additional applies in parentheses. 115

Table 6.1 Problem specification 125
Table 6.2 **Maximum** floating-point operation count on a single process for **failure-free** nested CG and nested GMRES solvers, as well as total global floating-point operations. 127
Table 6.3 Maximum operation count for a subdomain given worst-case performance in faulty experiments. 136
Table 6.4 Operation count for solving the Poisson problem using self-stabilizing CG. 276 total iterations: 221 unreliable and 55 reliable. 138

LIST OF FIGURES

Figure 1.1 Taxonomy of faults. 3
Figure 1.2 Taxonomy of errors as seen from the application level. 4
Figure 1.3 A reliable system that does not experience soft faults. 5
Figure 1.4 A system that experiences soft faults, and therefore is unreliable. 6
Figure 1.5 Selective reliability used to create the sandbox programming model and realized as a nested solver implementation. 7

Figure 2.1 Graphical representation of data layout in the IEEE-754 Binary64 specification. 20
Figure 2.2 Relation of exponent, IEEE-754 double precision bias, and what data are actually stored. 22
Figure 2.3 Examples of how a bit flip can impact an exponent represented using the IEEE-754 Binary64 specification. 23
Figure 2.4 Example of perturbed values for large numbers. 26
Figure 2.5 Example of perturbed values for small numbers. 27
Figure 2.6 Probability of observing an absolute error larger than 1. 31
Figure 2.7 Comparison of observed error caused by a flip in the exponent, excluding the most significant bit, for sampled vector sizes having similar relative magnitudes. 33
Figure 2.8 Number of possible absolute errors from dot products in Algorithm 1 in orthogonalization kernel. Class 1: $err < 1.0$ (blue), Class 2: $1.0 \geq err \leq \|\mathbf{A}\|_2$ (light blue), Class 3: $\|\mathbf{A}\|_2 > err$ (yellow), and Class 4: Non-numeric (red). . 37

Figure 3.1 Exponent values, biased exponent values, and storage. 44
Figure 3.2 Expected value and variance of the an N-bit mantissa. 49
Figure 3.3 Expected value and variance of the absolute error from a single bit flip in an N-bit mantissa. 51
Figure 3.4 Variance of the exponent and the dominant term in the growth of the variance as a function of the number of exponent bits. 54
Figure 3.5 Lower and upper bounds for the expected value of a positive exponent error $\left(\mathbb{E}\left[X_{\eta^+}\right]\right)$. 57
Figure 3.6 Expected relative error of the mantissa given a single bit flip. 63
Figure 3.7 Expected absolute error given a bit flip in the mantissa and a scalar less than one. The absolute error is shown for the fewest number of exponent bits, as well as the number of exponent bits for half, single, double, and quad precision. 67
Figure 3.8 Expected absolute error given a bit flip in the exponent and a scalar less than one, greater than one, and over the full range. Half (2^{12}), Single (2^{56}), Double (2^{1014}), and Quad (2^{16170}) precision specifications are highlighted. 70
Figure 3.9 Expected absolute error given a bit flip in the exponent, excluding the most significant bit for values less than one, and values greater than one. 76
Figure 3.10 Probability that the expected relative error is larger than one, given a single bit flip in scalar multiplication for: (a) half, (b) single, and (c) double precision. 78
Figure 3.11 Probability that the expected absolute error is larger than one, given a single bit flip in scalar multiplication for: (a) half, (b) single, and (c) double precision. 80

Figure 3.12 Absolute error experiment for half precision using the worst-case mantissa error for all mantissa errors. 81
Figure 3.13 Probability that the expected absolute error is larger than one, given a single bit flip in scalar multiplication for half precision and scalars in the range $[0, 1)$. 82
Figure 3.14 Probability that the expected absolute error is larger than one, given scalars in the range $[0, 1)$ and a single bit flip in scalar multiplication for (a) single and (b) double precision. 83

Figure 4.1 Taxonomy of faults and scope of this work. 89
Figure 4.2 Sandbox reliability implemented as reliable outer solves and unreliable inner solves. 94
Figure 4.3 Upper Hessenberg and tridiagonal matrices. 103
Figure 4.4 Number of outer iterations required for convergence when solving a Poisson equation given a single SDC event injected in the orthogonalization phase of the inner solve. Vertical bars indicate the start of a new inner solve. 105
Figure 4.5 Number of outer iterations required for convergence when solving the mult_-dcop_03 system of equations given a single SDC event injected in the orthogonalization phase of the inner solve. Vertical bars indicate the start of a new inner solve. 106

Figure 5.1 Overhead comparison with no fault detection, for (a) a random bit flip injection and (b) a numerical fault model. 113
Figure 5.2 Overhead comparison with fault detection on, for (a) a random bit flip injection and (b) a numerical fault model. 114

Figure 6.1 Flowchart for a nested solver implementation. 122
Figure 6.2 Overhead comparison of (a) `FGmres->Gmres->MueLu` and (b) `FGmres->Cg->MueLu`, given no attempt to directly detect or cope with errors. 129
Figure 6.3 Overhead when testing the explicit residual every iteration and using a projection length bound in `FGmres->Gmres->MueLu`. Explicit residual detections are hatched left \ and norm bound detections are hatched right /. 130
Figure 6.4 Overhead comparison given two different frequencies of explicit residual evaluation in `FGmres->Gmres->MueLu`. 131
Figure 6.5 Overhead when using a relaxed inner solver convergence test as well as a projection length bound in `FGmres->Gmres->MueLu`. 133
Figure 6.6 Flowchart for responding to detectors in a nested solver. 134
Figure 6.7 Overhead when using a relaxed inner solver convergence test as well as a projection length bound in `FGmres->Gmres->MueLu`, but no inexact tuning of the inner GMRES convergence rate. 135
Figure 6.8 Global (aggregate) floating point overhead when using a relaxed inner solver convergence test as well as a projection length bound in `FGmres->Gmres->MueLu`. (see Figure 6.5) 139
Figure 6.9 Overhead when using a relaxed inner solver convergence test as well as a projection length bound in `FGmres->Gmres->MueLu`: (a) Bit flips and (b) our pessimistic fault model. 140

Figure 6.10 Overhead when using a relaxed inner solver convergence test as well as a projection length bound in `FGmres->Gmres->MueLu` on a *weakly scaled* version of the Poisson problem. (see Figure 6.5 or 6.9b for comparison.) 141

Chapter 1

Introduction

Research and development has increasingly become model driven, where models are evaluated computationally (in silico), and physical experimentation is minimized (in situ). This explosion of computational science and engineering has driven the need for increasingly powerful systems to perform such computation. Weapons, climate, car frames, baby diapers, and even toothpaste are modeled, and optimizations to the design, manufacturing process, or product stability or longevity are accessed [65]. The systems used to evaluate such models are tasked with demanding performance and storage requirements, and we refer to computing on such devices as high performance computing (HPC) or high-end computing (HEC).

The resources provided by a single computational unit are rarely sufficient. Therefore, multiple systems are joined together to form a single HPC system. This coupling of independent discrete components to form a larger, more capable system is the foundation of supercomputing. The components used to compose these HPC systems are constantly changing. Beginning in the 1980s and continuing until 2004 [40], computer manufacturers obtained computational speedup primarily through increasing the frequency at which the processors execute instructions, i.e., frequency scaling [43], and by optimizing execution and cache reuse. Using these techniques, users were able to increase application performance by upgrading to faster processors. Starting in 2005, frequency scaling no longer continued to be the dominant method for obtaining software speedup. Now, without frequency scaling, the burden of increasing an application's performance lies on the back of the programmer. The programmer can attempt to find a faster algorithm, or the algorithm can be parallelized so that it may run concurrently on multiple processors.

The emergence of multi-core technology started with the release of IBM's Power4 dual core CPU in 2001. Intel made a final attempt at frequency scaling with the Pentium4. Ultimately, Intel drastically changed its processor roadmap and began producing only multi-core chips for the high-end market starting in 2005. The trend of multi-core processors has now evolved to the point of many-core systems. The Intel Xeon Phi architecture now exposes hundreds of lightweight

cores to the application developer. HPC systems have also embraced accelerator technologies such as graphics processing units (GPUs), and advanced solid-state storage controllers that can preprocess data before writing to stable storage, storing only a subset of the data. The impact of these technologies on HPC systems is that a single "node" has become denser in terms of components used, and they have become much denser with respect to transistor count.

While frequency scaling has ceased to enable faster systems, manufacturers have instead increased the amount of parallelism that components support. Single-core CPUs transitioned to multi-core, and now to many-core. These transitions have been partly enabled by the ability to pack more transistors onto a piece of silicon, called a die. These "die shrinks" are made possible by manufacturing components at smaller and smaller scales. A smaller manufacturing process typically results in lower power consumption, but has the negative effect that it becomes increasingly more difficult to distinguish between a one and a zero. Customers also wish to lower peak power requirements, and in general reduce overall power consumption. For this reason device manufactures are pressured to create chips that can operate near the device's threshold voltage, i.e., operate at a voltage where the difference between "on" and "off" is very small. This lowers the power that a chip requires, but also makes the device more susceptible to faults [81, 44, 59]. That is, small energetic disturbances can be sufficient to cause devices to fault.

This leads to the current U.S. Department of Energy (DOE) Exascale roadmap. There are two types of leadership-class computing systems planned to achieve Exascale performance. One system will utilize standard CPUs with GPU accelerators, while the other will follow the many-core approach and use Intel Xeon Phis. Regardless of the architecture approach used, both system types will continue to increase the number of computing units per physical node. This increased component count also increases the probability that some component will fail. Hence, the reliability of the machines is a key focus, (e.g., in [92]), where the concern for reliability is pervasive throughout the "Top 10 Exascale Challenges." The general approach of tolerating faults in HPC systems is called HPC resilience or simply *resilience*, and this term encompasses a superset of fault tolerance.

1.1 Resilience

Resilient computing is a broad response to the inherent unreliability of systems. This area of study is birthed from the unreliability observed in large-scale HPC deployments. For example, Table 1.1 shows the Mean Time Between Failure of specific HPC systems or vendor deployments. The MTBF is a measure of the average time before a component fails, but the key is what constitutes a failure. Prior works in resilience, including my own [28, 91, 90], have focused on a *hard* fault model. A *fault* is the event after which a component does not operate according to its specification. The cause of a fault may be a defect, or the effect of some external event,

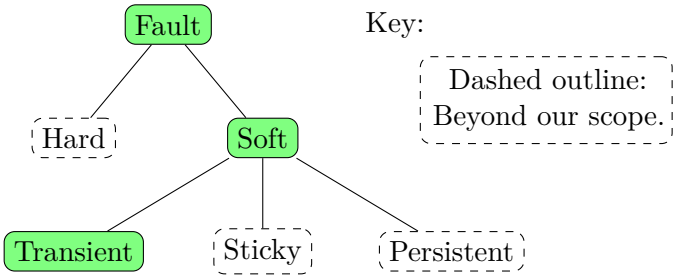


Figure 1.1: Taxonomy of faults.

e.g., a charge particle hitting the gate of a transistor causing it to flip or become stuck. That is, faults are events that occur at the hardware level, and we broadly classify such events as: 1) the device continues to operate or 2) the device fails and ceases to operate. We present a taxonomy of faults in Figure 1.1. Should the device continue to operate, we call this a *soft* fault, and if the device ceases to function, we call this a *hard* fault. Our work focuses on the effects that a soft fault can have.

Table 1.1: Reliability of HPC Systems

System	Number of Cores	MTBF [57]
ASCI Q	8,192	6.5 hrs
ASCI White (2001)	8,192	5 hrs
ASCI White (2003)	8,192	40 hrs
PSC Lemieux	3,016	9.7 hrs
Google	15,000	20 reboots/day
ASC BG/L	212,992	6.9 hrs (LLNL est.)
Exascale Projection	1,000,000+	$<$ 1 hr [92]

Soft faults present several challenges and their effects are the focus of this work. If a fault occurs and the device continues to operate, then it is possible that the data operated on have become corrupt as a result of this fault. That is, a bit could have been flipped at some point, and the results can become fictitious. We consider the case of transient soft faults, i.e., faults that occur, but do not permanently damage the hardware. A fault could have the effect of being sticky or persistent, and the distinction between the two is subjective [87]. The key is that we are focused on the *effects of a soft fault*, which is an error.

An *error* is the manifestation of a soft fault at the application level. For example, a fault can impact a register holding an IEEE-754 floating point value. When this perturbed value is

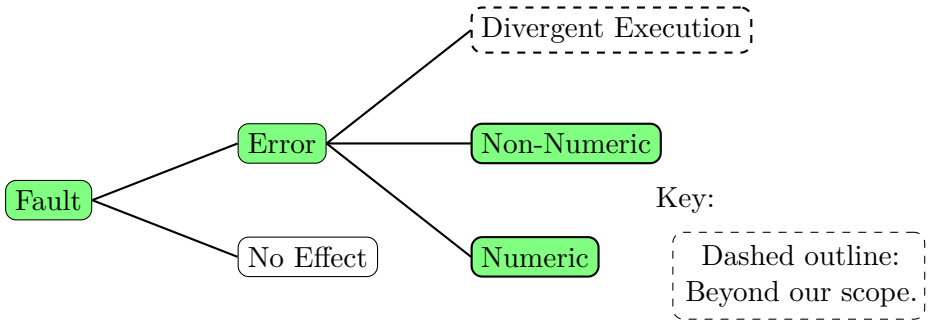


Figure 1.2: Taxonomy of errors as seen from the application level.

used, the application may become erroneous in terms of its calculations. Figure 1.2 presents our view of what errors look like from the application level. This view is extremely generic, but is sufficient for developing resilient algorithms, which is the over-arching goal of this work. Our observation on how errors can affect applications is rooted in the fact that given an error, the application will either continue execution and present a result, or the application will enter some indefinite state where no solution is presented. Our classification of errors covers a broad set of possible faults. For example, if a pointer is corrupted, but dereferencing the pointer is successful, the data read will be treated as whatever numeric data type was intended. This wrong data will then be interpreted leading to a either a valid representation (i.e., a numeric error), or it will lead to an invalid representation creating a non-numeric value. The latter is a good case, as it is easy to detect that an error was made, while the former is particularly damaging, since the algorithm will operate on incorrect values and only introspection of some sort can identify that the values returned are nonsensical. Divergent control paths due to errors are beyond the scope of this work but have been considered elsewhere [75].

1.2 Resilience and Fault Tolerance

Resilience represents a broad response to unreliability in HPC. It encompasses more than fault tolerance alone. The premise underlying resilient computing is that faults are the norm, rather than the exception. The idea that faults, both hard and soft, will become more likely is expected to continue into exascale computing [35, 63, 16]. Thus, novel approaches for scalable resilience, not only at the hardware but also the software level, are required. The premise that faults are the norm implies a more aggressive approach to fault tolerance that was previously conceived. We seek to enable *forward recovery* in the presence of faults, rather than a rollback recovery scheme. That is, we wish to devise algorithms that can tolerate the errors introduced by faults, in a proactive manner without losing forward progress. We think of soft fault resilience as an

adversarial approach to fault-tolerance, whereas prior hard fault-tolerance viewed faults as rare events. We contrast our view by presenting what constitutes a reliable machine if soft faults are not allowed (see Figure 1.3). Without soft faults, the machine only needs to remain functional for a valid solution to be presented. If soft faults are allowed, then the machine loses its *numerical reliability*, which we show in Figure 1.4. Our work focuses on a general research approach to harden algorithms to the errors presented in Figure 1.2 under the assumption that the machine is unreliable as shown in Figure 1.4.

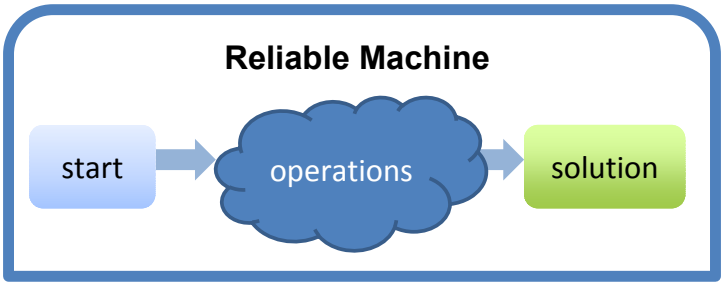


Figure 1.3: A reliable system that does not experience soft faults.

The errors of interest in Figure 1.2, "Non-Numeric" and "Numeric," can be classified as *silent data corruption* (SDC). Silent means that the system gives no indication that a soft fault occurs, and this leads to silent corruption of the data operated on. Our goal is to ensure that algorithms can either tolerate SDC, or the algorithm can indentify that the corruption has occurred, making the error *not* silent. Attempts have been made to quantify the occurrence of SDC. Michalak et al. [70] statistically analyzed failure data obtained from a beam study that subjected hardware to blasts of charged particles. Their findings indicate that this type of error is rare enough that it is still difficult to observe.

1.3 Detecting Errors and Selective Reliability

If machines lack numerical reliability, then there is one robust technique for identifying corrupt values, redundant computing [28, 38, 39, 36]. A general idea underpinning redundant computing is the ability to detect corruption by majority voting. This type of fault tolerance can be realized by replicating computations onto different machines, and then voting on the result obtained. A standard approach is called triple modular redundancy (TMR). The problem with redundancy is that it requires N-redundant copies of data as well as operations on these duplicate data. This implies an N-times cost, which is difficult to imagine in the realm of HPC, where performance is at a premium.

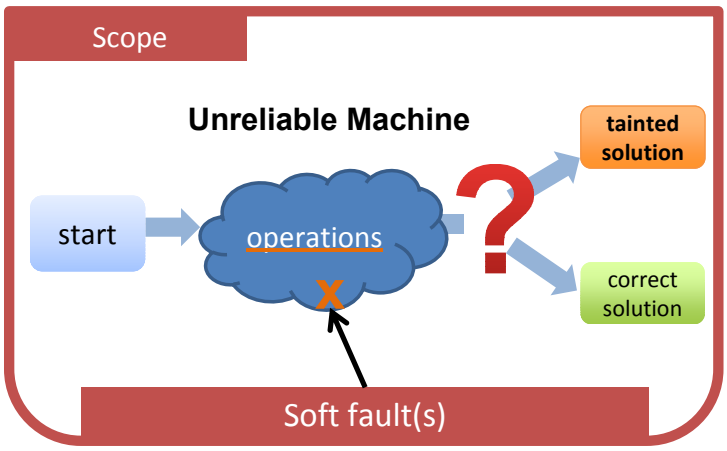
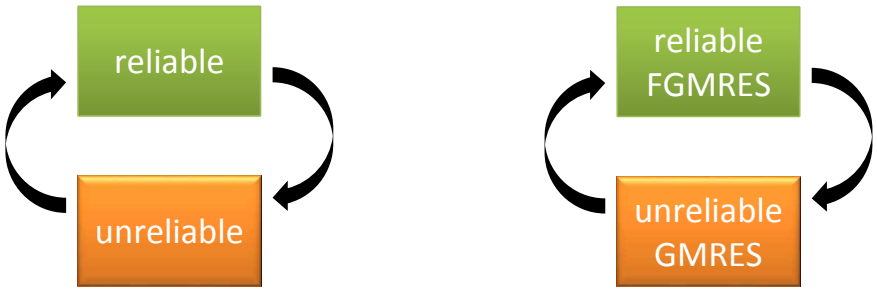


Figure 1.4: A system that experiences soft faults, and therefore is unreliable.

If an application can detect and correct all errors, then it is preserving the illusion of a reliable machine at all levels. We call this an illusion, because no machine is 100% reliable. However, we simply expect that unreliability will be rare enough that it is not a concern. Heroux [47] proposed the idea of *Selective Reliability*, which requires expert knowledge to specify which calculations in an algorithm must be reliable, and which can be run in an unreliable mode. Hoemmen and Heroux [13] realized this concept using a two-level iterative solver, FT-GMRES. FT-GMRES preconditions Flexible GMRES (FGMRES) by GMRES, possibly with its own preconditioner. The outer FGMRES iteration is identified as needing reliability, while the inner GMRES (and its preconditioner) is marked as suitable for unreliability. In this setup, the outer iteration absorbs the error introduced from numerical unreliability, while still making progress towards the correct solution. They use selective reliability to develop a programming model that "sandboxes" unreliable computations, and promises reliability on specific computations.

We illustrate the idea of selective reliability in Figure 1.5a, and then show the realization using nested solvers in Figure 1.5b. The key is that selective reliability does not describe what can be unreliable. Instead, it *only* declares what *must* be reliable. This approach enables phenomenal flexibility. For example, a reliable FGMRES outer solver can wrap complicated (black box) preconditioners, while still promising that if a solution is obtained it will be correct. This approach is in stark contrast to current trends in fault-tolerant algorithms, where algorithm developers are attempting to robustify every numerical method to handle faults.

Selective reliability and nested solvers ensure the correctness of calculations, and enable forward progress by *absorbing* errors. This makes this approach susceptible to high overhead, as some errors may introduce no or low overhead, while others may induce very high-overhead. A



(a) Sandbox programming model. (b) FT-GMRES.

Figure 1.5: Selective reliability used to create the sandbox programming model and realized as a nested solver implementation.

focus of our work is on making algorithmic decisions that, when composed, create a resilience algorithm that avoids high-overhead, while still maintaining correctness.

There are three challenges a linear solver must cope with in an environment that emits soft errors. First, one must assess how and how much incorrect intermediate results might affect a solver. Consider solving the sparse linear system resulting from discretizing a partial differential equation (PDE). If a bit flip corrupts a single entry of the matrix, how much that corruption affects the computed solution depends on its magnitude, properties of the linear system, the solution method, and whether the incorrect value stays corrupted or goes back to its correct value. Second, one must design the solver (possibly with help from the hardware and operating system) to produce the right answer despite occasionally incorrect data or arithmetic, whenever possible. Our philosophy for exascale computing is "forward execution": Instead of reacting to faults by rolling global execution back to a checkpoint, a method known to be limited in scalability [37, 28], we promote non-stop algorithmic / execution approaches where progress continues toward obtaining a result even in the presence of failures. Third, if corruption is bad enough to make forward progress impossible, the solver must detect this scenario and report it back to the user, rather than silently computing the wrong answer. This is especially important for simulations that support high-consequence decisions, where silently wrong answers have unthinkably high cost and high impact. Our goal is to have high confidence in the solution, even if we have lower confidence in the computer hardware that computes it.

Much resilience research has focused on tolerating parallel process failures through standard techniques like checkpoint/restart (C/R) or process replication. The "monster in the closet" [44] is incorrect hardware behavior that does not result in process failure. Faults like incorrect arithmetic or memory corruption may make the application produce incorrect results or increase run time, with no indication from the system that something went wrong. If systems cannot correct these faults before they affect applications' behavior, then the burden of tolerating them

shifts to algorithms: either to detect abnormal behavior and correct it, or to "absorb" its effects while still making progress towards the correct solution.

1.4 Resilient Sparse Preconditioned Linear Solvers

In this work, we study a specific type of unreliability that is believed to become more prominent in future architectures. At extreme scales of computation, applications will have to face the possibility of incorrect computer arithmetic or storage. Incorrect arithmetic (or storage) creates an extremely challenging problem, particularly for algorithms that are used in mission-critical applications or are used in a decision making process. There are many fundamental kernels, and we choose to focus on the solution of large, sparse linear systems. Asanovic et al. [6] classify sparse methods as one of the fundamental "dwarfs" of parallel computing. The solution of linear systems is the foundation of nonlinear solvers, they are at the heart of many iterative eigensolvers, as well as the entire field of implicit time-stepping algorithms. Many scientific computing problems approximate solutions to models evaluated on structured or unstructured grids, which typically lead to large *sparse* linear systems. Much effort is expended by scientists to ensure that systems are sparse, and then this sparsity is exploited for computational and storage efficiency. A term sparse linear solvers is synonymous with is preconditioning. Most linear solvers can be formulated to allow preconditioners to accelerate their performance by *preconditioning* the linear system such that the solver is able to obtain a solution faster and more accurately than if no preconditioner was used.

Sparse linear operations are so crucial that the current- and next-generation assessment of HPC systems are being benchmarked using preconditioned sparse iterative linear solvers rather than dense linear algebra. Dongarra et al. [26] proposed the High Performance Linpack (HPL) benchmark in 1979 for benchmarking floating point computation. HPL solves a dense linear system using a lower/upper (LU) factorization. The LU factorization is a direct method, i.e., it is not iterative, therefore the floating point cost can be precisely modeled. The benchmark measures the runtime, and then determines how many floating point operations (FLOPs) are performed each second (FLOPS). The dense calculations performed in HPL do not represent the poor memory access patterns, load imbalance, and inherent serial components that many modern scientific kernels experience. That is, HPL benchmarks raw floating point performance, but is not characteristic of how systems are actually used. To address this, Heroux and Dongarra [48] have developed a new benchmark, High Performance Conjugate Gradients (HPCG) [51], that is intended to more accurately represent the type of problems and algorithms actually used on peta- and exascale systems.

Unsurprisingly, HPCG is designed around a sparse, preconditioned linear solver, because sparse operations are much more common than large dense factorizations. HPCG is a sparse

preconditioned conjugate gradient solver that uses a local symmetric Gauss-Seidel preconditioner, which is an inherently serial algorithm. The point is that sparse linear solvers are largely considered a defining component of current and next-generation HPC. Further evidence supporting the dominance of sparse solvers is the Mantevo mini-apps [67]. Mini-apps are intended to model general scientific computing patterns. For example, MiniFE performs phases of computation similar to how finite element codes would be structured. The solver used in miniFE is a sparse iterative solver. Sparse techniques are also the state-of-the-art in large graph analysis. Often, graph algorithms seek to determine the dominant eigenvalue or eigenvector, or perform a singular value decomposition on the sparse matrix representation of the graph. Techniques often use the graph Laplacian, which is typically formed as a large sparse matrix.

Because of the depth and importance of sparse linear solutions, we choose to focus our study on the development of resilient preconditioned linear solvers. That is, we study the coupling of current state-of-the-art numerical methods with the unreliability presented by high-performance computing. We constrain our work to ensure that the techniques we propose are competitive with the current linear solvers used.

Our goal is to develop and assess resilient sparse iterative solvers. In general, we seek to advance algorithm-based fault tolerance (ABFT) for sparse iterative solvers. We propose and demonstrate a very different research methodology from prior works. Traditional ABFT has focused on preserving the illusion of a reliable machine at all levels, and has assessed their prior approaches by synthetically injecting bit flips into data. We find this general approach unproductive for assessing fault tolerance techniques, and we believe for iterative solvers, there is a superior research philosophy to follow. First, attempting to enforce reliability at all levels implies modifying all codes such that they are fault tolerant. We feel this is a poor path to follow, because the breadth and depth of numerical codes is enormous. Following this path, one cannot stop after hardening just one or two computational kernels (like sparse matrix-vector multiply) that dominate run time; one must ensure that *all* kernels used in the application are hardened. Given that most scientific codes take years to develop as-is, it seems infeasible to assume we can harden all codes, or even a selected few. Second, assessing resilience techniques should not depend heavily on the definition of a fault or the resulting error. Resilience research is rife with speculation. Scientists must speculate about the reliability of future hardware. Moreover, given a digital machine, all faults can be mapped to some number of bit flips, but this is not helpful at the application level, because those bit flips may or may not impact a sensitive value. For example, the error introduced by a bit flip may lie within the rounding or approximation error of the technique used, rendering the error introduced negligible. Third, for iterative methods, errors can change the runtime of the application, and so the impact of the error is extremely meaningful, not its origin.

1.5 Research Hypothesis

We believe that the research methodology used in many algorithm-based fault tolerance schemes is flawed. We believe that a different method will enable more theoretically sound resilience algorithms, and enable quantifiable resilience. That is, we believe our method enables researchers to clearly specify what faults algorithms and components of algorithms are capable of addressing in a quantifiable way. This leads to a hypothesis that we will test in Chapters 4–6, and motivate experimentally and analytically in Chapters 2–3.

> *I hypothesize that coupling numerical analysis and systems reliability can make iterative numerical methods tolerate many types of soft errors (based on an abstract fault model) more efficiently and with greater theoretical soundness than by using a systems-only approach.*

1.6 Overview

We begin by analyzing what a bit flip in the IEEE-754 specification entails. This is relevant to our work, because prior studies have injected synthetic bit flips. We show experimentally in Chapter 2 that bit flips injected into the representation of floating point data behave in very specific ways. We support our experiments with bounds constructed through numerical analysis, and show that our findings are expected given the bounds derived. In Chapter 3, we rigorously analyze the expected absolute and relative error and prove that the observations from Chapter 2 hold given the standard IEEE-754 formats of half, single, double, and quad precision. We show that the expected relative and absolute errors can be forced to behave predictably for a bit flip in a scalar or in scalar multiplication. Furthermore, we use our analysis to assert that moving to higher-precision formats is not bad. This is particularly important as the quad precision format becomes more utilized. While we thoroughly analyze the errors introduced given a bit flip in floating point data, the purpose of our study is not to advocate the use of bit flips as a research approach. We conclude instead that bit flips are a poor fault model for algorithm design and assessment, because

1. they can introduce a wide range of errors, and
2. the resulting errors are biased.

Our goal is to identify and assess algorithmic techniques for resilience, but not to speculate about the origin of the errors introduced or verify if injection techniques follow a bit-flip error distribution.

1.6.1 Bit Flip Model May Not Reflect Actual Hardware Behavior

Bit flips can manifest all sorts of problems, from corrupting arithmetic results or storage to changing the instruction stream. There are too many ways in which things could go wrong, so it is not clear where to start predicting behavior. For example, what happens if data in a cache are corrupted? It depends on whether the algorithm reads or writes the corrupted data, as well as the cache eviction policy. It is also not clear whether possible *future* hardware, which reports faults back to applications, will behave according to our models. For example, hardware may change how floating-point data is stored, and only ensure that the representation follows the IEEE-754 specification if the users interacts with the data through an API. There is too much uncertainty surrounding exascale already. Using bit flips as a vehicle for algorithm analysis and assessment only adds speculation and we get no benefits from making such unfounded assumptions. Instead, we advocate an adversarial approach. Rather than allow errors to vary from negligible to devastating, we develop a fault model that introduces errors that are always bad. This removes the uncertainty over whether the fault impacted a key value, and allows us to focus on how our algorithmic techniques behave when faced with faults that induce high overhead, or even cause our resilient algorithm to fail. The latter is particularly useful, as we must determine the weaknesses of our approach, if we are to promise that our algorithm is resilient to errors.

1.6.2 Skeptical Programming: Bounding Soft Errors

The "random bit flip" fault model does not reflect what algorithms actually need to know. Algorithm developers do not care whether a network packet dumped garbage into our reduction or a cosmic ray blasted 20 entries in an array. All of these events can be modeled as numeric perturbations in the algorithm. Furthermore, we can *bound* these errors by detecting their effects. Then we can use numerical approaches that can tolerate "large" errors, where "large" means "much larger than rounding error, but not large enough to detect."

Traditional ABFT attempts to preserve the illusion of an always-reliable machine. Instead, we favor an approach more compatible with numerical analysis. First, we bound the error that faults can introduce. Second, we identify methods that can tolerate the largest error possible. This strategy is based entirely on the algorithm theory and the data itself. That is, given specific inputs, we can bound large portions of an algorithm using standard norm bounds and inner product bounds. These bounds enforce that errors committed in intermediate computations do not exceed the theoretical limit imposed by the algorithm in conjunction with the data provided. We demonstrate this approach in Chapter 4.

Because operations are unreliable, the bounds allow us to be *skeptical* of key values. We use a bound on orthogonal projections in Chapter 4, while van Dam et al. [93] use a bound on an

inner product. These bounds work as *filters* rather than error detectors. We make no promise to detect and correct all errors, we merely promise *bounded error*. We refer to this approach as *Skeptical Programming*.

Numerical research provides a continuous stream of results that could be used in our Skeptical Programming strategy. The key factor is that these bounds be always 1) cheap to evaluate, and 2) be determined *before* the algorithm is run. We prefer to determine these bounds *a priori* because the unreliability of the machine may affect any bound computed inside the algorithm. That is, if we allow a bound to depend on unreliable computation, then the bound becomes unreliable as well. We may have to relax (2) in some cases, but we desire (1) to be true. Chapter 6 explores in detail how detectors can drastically influence the computational cost of various resilient algorithms. Specifically, we find that detectors must be of low overhead. Our results from Chapter 4 are showcased, because the overhead to evaluate the detector is small.

1.6.3 Worst-Case Behavior: Adversarial vs. Practitioner

Our analysis of bit flips motivates the development of a better fault injection and algorithm assessment strategy. We show in Chapter 4 that bounding errors tends to be effective, but it is difficult to quantify how effective. Stochastic sampling is a natural tool to use for algorithm assessment. Random sampling, in itself, is not bad, but as a means to justify the correctness of a numerical method, it is inadequate. Numerical methods typically have *proven* behavior and correctness. If operations can be unreliable, then we need to identify the bounds in which a resilient algorithm is reliable. That is, we should understand the smallest and largest errors we may commit. Relying on sampling alone leaves us prone to a practitioner design pattern, where things are fixed only when someone (or a sample) identifies there is a problem. We feel a more adversarial approach is required, and this approach fits naturally with a *bounded error* design methodology. For a given numerical kernel, we wish to know the worst-case error that can be committed, and with this knowledge, we can employ pure and applied mathematicians to aid us in designing methods that can tolerate such error bounds.

When developing an algorithm, we cannot ignore the extreme cases, because if we do so, we have unstated assumptions about the way in which the algorithm can be used. For numerical methods, unstated assumptions make the method nearly worthless, given that we can never anticipate what combinations of data the user will throw at the algorithm. For this reason, we advocate moving from a bit flip model to a more abstract model that evaluates algorithms based on their ability to absorb unexpected numerical variability.

1.6.4 Abstract Fault Models Are Superior

Resilience approaches used in daily practice, such as Checkpoint/Restart (C/R), are attractive because they presume an abstract, minimal fault model. C/R assumes only that checkpoints contain correct state, and are stored reliably to stable (shared, non-volatile) storage. It does not care whether the application failed due to a crashed node, network errors, a power outage, or the job running out of allocation time. This abstract fault model lets C/R recover from many different kinds of faults.

SDC, by definition, does not trigger system actions like a restart. The silent error can manifest in several ways, such as performance variation between parallel processes, convergence to a wrong solution, or even an application crash sufficiently long after a checkpoint is written that contains the tainted state. Given the success of C/R's abstract fault model, why then has soft error research focused so heavily on a low-level, fine-grained "bit flip" fault model? Bits may go wrong if an error is introduced, but this does not aid in the design of algorithms that work at a much higher abstraction layer than bits. Applications care about data types, such as floating-point values and integers, not about the bits which compose them.

We argue that resilient numerical methods should be designed around an abstract fault model of *numerical unreliability*, in much the same way C/R is designed around an abstract model of system unreliability. We present a case for a radically different research methodology that merges numerical analysis with systems fault tolerance. We demonstrate our adversarial fault model in Chapter 5, and then show how we can use this research approach to compose a better resilient preconditioned linear solver in Chapter 6.

We demonstrate that many types of soft errors can be tolerated by employing an abstract fault model that does not presume the origin of the error. This methodology avoids root cause analysis, which may be important for hardware designers, but has no real meaning to numerical algorithm developers. At the numerical method level, soft errors manifest as potentially large numerical error. We present findings supporting our claim that such large errors can be treated using a hybrid approach of system reliability and numerical analysis.

1.7 Summary of Contributions

Our work is focused on the development and assessment of resilient preconditioned linear solvers. To motivate our methodology, we present a thorough analysis of the error that a bit flip in an IEEE-754 scalar introduces. We then use this analysis to motivate a superior fault injection methodology that enables us to assess the overhead of our detection and forward recovery mechanisms in the cases of detectable and undetectable errors.

- Chapter 2 shows an empirical study of a bit flip in dot products, as well as modeling that shows loose bounds on what the absolute and relative errors can be. We expose the probability that errors will behave as "small" or "large," and show how this property relates to the numerical concepts of normalization and matrix equilibration. Furthermore, we show that exploiting this property is practical, and in some cases enabled by the algorithms themselves.
  This work is published at *ScalA'14* [30] and submitted to the *Elsevier Journal of Computational Science* [32].
- Chapter 3 models the analytic expected absolute and relative errors given a bit flip in IEEE-754 scalars. We show general models that are applicable to all standard IEEE-754 formats: half, single, double, and quad precision. We establish proofs that the properties observed in Chapter 2 hold for the standard specifications, and are able to reason about the impact of using different floating point formats.
  This work is submitted to *Arith'16* [33] and *SIAM Journal on Scientific Computing* [33].
- Chapter 4 derives the first error detector for the GMRES algorithm, and assesses the impact of small and large errors on the number of iterations required for convergence using a selective reliability nested solver scheme. We show empirical evidence that relatively small errors introduce lower overhead than very large errors.
  This work is published at *IPDPS'14* [29].
- Chapter 5 develops a fault model and injection methodology that consistently introduces overhead higher than random bit flip injection and is inherently parallel. Furthermore, our fault model and method injects errors in a novel way that lets us control if the error is detectable.
  This work is published at *HPDC'15* [31].
- Chapter 6 combines the detector from Chapter 4 with the fault model and methodology from Chapter 5 to make quantifiable progress in resilient algorithm design. We identify a high-overhead detector using our methodology and enable correct results with overhead substantially lower than the original FT-GMRES algorithm would entail.
  This work is submitted to *PPoPP'16* [34]

Chapter 2

Data Representation and Fault Tolerance

In the field of high-end computing (HEC) the notion of reliability has tended to focus on keeping thousands of physical nodes operating cooperatively for extended periods of time. As chip manufacturing and power requirements continue to advance, soft errors are becoming more apparent [70]. This implies that reliability research must address the case that the machine does not crash, but that outputs during computation may be silently incorrect. There have been many studies into hardening numerical kernels against soft errors, that is the researchers attempt to preserve the illusion of a reliable machine by detecting and correcting all soft errors. We take a more analytical approach. Instead of focusing on detection/correction, we seek to study how the data operated on impacts the errors that we can observe given soft errors in data — called silent data corruption (SDC).

2.1 Introduction

The driving motivation behind our work is the uncertainty surrounding the reliability of an exascale-class machine [35, 63, 16]. We attempt to avoid speculation over what hardware may be used in future (or present) HEC deployments, and instead analyze how a single soft error in an IEEE-754 floating-point number behaves. It has already been shown that existing and decommissioned HEC deployments have suffered from SDC [70, 46]. For the prior reasons, we seek to study the link between the data operated on and soft errors. We intentionally perform our research subject to the IEEE 754 specification, which we believe will be used regardless of the architecture. We also restrict our analysis to single bit flips. This gives us a base line from which to draw higher-level conclusions related to multiple bit flips, and lets us isolate the impact of a bit flip. While the analytic model for IEEE-754 scalars motivates this chapter, we use numerical upper bounds on how a bit flip will influence a scalar. That is, we explore the general characteristics of a bit flip in an IEEE-754 scalar. In Chapter 3, we take a more nuanced

approach and analytically derive the expected value of the absolute and relative error given a bit flip in a scalar or scalar multiplication.

IEEE 754 both defines the binary *representation* of data, and bounds the *rounding error* committed by arithmetic operations. This work focuses on data representation. The effects of rounding error on numerical algorithms, including those studied in this work, have been extensively studied; see e.g., [74]. However, these results generally only apply to *small* errors, such as those resulting from rounding. Bit flips can be huge and thus require different methods of analysis, like those presented in this work.

We present the following contributions:

- We model single bit upsets in IEEE-754 scalars analytically, and extend this modeling to dot products.
- We demonstrate both experimentally (via Monte Carlo sampling) and analytically that dot products performed on normalized numbers have a significantly lower probability of experiencing large error than dot products with values of varying magnitudes.
- We relate our finding that normalized vectors minimize absolute error to matrix equilibration, and correlate this finding to two highly used numerical kernels (Gram-Schmidt orthogonalization and the Arnoldi process).
- We demonstrate the utility of our finding by instrumenting the Generalized Minimum Residual Method (GMRES). We show that for the dot product intensive orthogonalization kernel, we can restrict errors arising from single bit upsets to being less than one, or being very large and easily detected.
- We articulate how studying single bit flips enables us to draw conclusions about multiple bit upsets.

2.2 Related Work

Researchers have approached the problem of SDC in numerical algorithms in various ways. Many take the approach of treating an algorithm as a black box and observing the behavior of these codes when run with soft errors injected. Recently, [55, 56] analyzed the behavior of various Krylov methods and observed the variance in iteration count based on the data structure that experiences the bit flip. Shantharam et al. [82] analyzed how bit flips in a sparse matrix-vector multiply (SpMV) impact the L^2 norm and observe the error as CG is run. Bronevetsky et al. [14, 86] analyzed several iterative methods documenting the impact of randomly injected bit flips into specific data structures in the algorithms and evaluated several detection/correction schemes in terms of overhead and accuracy. Exemplifying the concept of black-box analysis, [66] presents BIFIT for characterizing applications based on their vulnerability to bit flips. Rather

than focusing on how to preserve the illusion of a reliable machine or devising a scheme to inject soft errors, we investigate an avenue mostly ignored, which is how the data in the algorithm can be used to mitigate the impact of a bit flip.

Heroux proposed a radically different approach. Rather than attempt to detect and correct soft errors, he proposes a "selective reliability" programming model to make the algorithm converge *through* soft errors [47]. Hoemmen and Heroux realized this model using nested linear solvers [13]. Sao and Vuduc showed that reliably restarting iterative solvers enables convergence in the presence of soft errors [79]. In the same vein, Elliott et al. showed that bounding the error introduced in the orthogonalization phase of GMRES lets FT-GMRES converge with minimal impact on time to solution [29]. Boley et al. applied backward error analysis to linear systems, in order to distinguish small error due to rounding from inacceptably large error due to transient hardware faults [12]. In general, our work complements this line of research. While Hoemmen and Sao have investigated algorithms that can converge through errors, we show that in certain numerical kernels, the data itself can have a "bounding" effect. For example, coupled with [29], we improve the likelihood that errors fall within the derived bound.

Algorithm-based fault tolerance (ABFT) provides an approach to detect (and optionally correct) faults, which comes at the cost of increased memory consumption and reduced performance [58, 27]. The ABFT work by Huang et al. [58] was proven by Anfinson et al. [4] to work for several matrix operations, and the checksum relationship in the input checksum matrices is preserved at the end of the computation. Consequently, by verifying this checksum relationship in the final computation results, errors can be detected at the end of the computation. Recent work has looked at extending ABFT to additional matrix factorization algorithms [27] and as an alternative to traditional checkpoint/restart techniques for tolerating fail-stop failures [23, 21, 19].

Costs in terms of extra memory and computation required for ABFT may be amortized for dense linear algebra, and such overheads have been analyzed by many (e.g., [2, 10, 62]). Algorithms must be manually redesigned for ABFT support by accounting for numerical properties (e.g., invariants). A more fundamental problem is that traditional checksums and error-correcting codes do not suit floating-point computations well [12]. Such computations naturally commit rounding error, which exact (bitwise) codes forbid. Inexact codes (that use floating-point sums) can be sensitive to rounding error, and commit it themselves. It is possible that more expensive recovery and significantly more redundant storage could help [15]. However, works like [13, 79, 29, 12] suggest that *correcting* faults might not be necessary, if their effects on the algorithm are detectable and bounded. In general, our work favors "opening up the black box" and understanding the effects of soft error on algorithms, rather than trying to detect and correct all such errors before they affect algorithms' behavior.

2.3 Project Overview

To explore the relation between data representation and soft errors, we first construct an analytic model of a soft error in an IEEE-754 floating-point scalar, and then extend this to a dot product. We uncover through analysis that the binary pattern of the exponent can be exploited for fault tolerance. We show this graphically via a case study using Monte Carlo sampling of random vectors, and then extend the idea of data scaling to matrices by using sparse matrix equilibration. To demonstrate the feasibly and utility of our work we analyze the GMRES algorithm and instrument the computationally intensive orthogonalization phase. We count the possible absolute errors that can be introduced via a bit flip in a dot product, and show that scaling data lowers the likelihood of observing large, undetectable errors.

This chapter is organized as follows:

1. In § 2.4, we construct an analytic model of the absolute error for single bit upsets in IEEE-754 floating-point numbers.
2. In § 2.5, we extend our model of faults in IEEE-754 scalars to vectors of arbitrary values, and present examples of how data scaling impacts the binary representation and absolute error we can observe.
3. In § 2.6, we perform a case study using Monte Carlo sampling of 10,404,000,000 random vectors of various magnitudes, and graphically show how the error is minimized when operating on values less than 1.
4. In § 2.7, we link data scaling to sparse matrix equilibration, and instrument and evaluate the impact of a soft error in the computationally intensive orthogonalization phase of GMRES.

2.4 Fault Model

The premise of our work is that a silent, transient bit flip impacts data. Before we can perform any analysis or experimental work, we must define how such a bit flip would impact an algorithm, and how we enforce that the bit flip was transient. To achieve this goal, we build our model around the basic concept that when an algorithm uses data, this translates into some set of operations being performed on the data. Should a bit flip perturb our data, some operation will use a corrupt value, rather than the correct value. The output of this single operation will then contain a tainted value, and this tainted value could cause the solution to be incorrect. Note that a transient bit flip may cause a persistent error in the output depending on how the value is used.

A side benefit of an operation-centric model is that we naturally avoid a pitfall to which arbitrary memory fault injection succumbs, namely that if a bit flip impacts data (or memory) that is never used (read) then this fault *cannot lead to a failure*. Our fault model allows a bit flip to perturb the input to an operation performed on the data, while not persistently tainting the storage of the inputs. This mimics how a transient bit flip would manifest itself, e.g., during ALU activities. As a result, the data that experiences the bit flip need not show signs that it was perturbed. This model allows us to observe the impact of transient flips on the inputs, which results in sticky or persistent error in the result. We then utilize mathematical analysis to model how this persistent error propagates through the algorithm.

2.4.1 Fault Characterization via Semantic Analysis

To derive a fault model we must first understand what a fault is. Since floating-point numbers approximate real numbers and most numerical algorithms use real numbers, we start from the definition of a real-valued scalar $\gamma \in \mathbb{R}$. The range of *possible* values that γ can take is

$$\gamma \in [-\infty, +\infty].$$

We assume that the IEEE-754 specification for double-precision numbers, called *Binary64*, is used to represent these numbers. This means that γ can take a fixed set of numeric values, and these values lie in the range

$$\gamma \in [-1.80 \times 10^{308}, +1.80 \times 10^{308}],$$

or using base two for the exponent

$$\gamma \in [-1.\bar{9} \times 2^{1023}, +1.\bar{9} \times 2^{1023}],$$

where $1.\bar{9}$ indicates the largest possible fractional component, and 1.0 indicates the smallest fractional component. A more informative range is that of $|\gamma|$, excluding 0 and denormalized numbers,

$$|\gamma| \in [2.23 \times 10^{-308}, 1.80 \times 10^{308}], \tag{2.1}$$

and in semi base two

$$|\gamma| \in [1.0 \times 2^{-1022}, 1.\bar{9} \times 2^{1023}]. \tag{2.2}$$

To approximate real numbers, *Binary64* uses 64 bits, of which 11 are devoted to the exponent, 52 for the fractional component (we refer to as the mantissa), and one bit for the sign. Figure 2.1 shows how these bits are laid out. In addition to numeric values, Binary64 includes two non-numeric values, Not-a-Number (NaN) and Infinity (Inf), which may be signed to account for

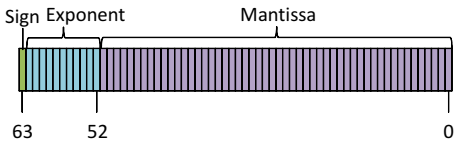


Figure 2.1: Graphical representation of data layout in the IEEE-754 Binary64 specification.

infinity and values that result in undefined operations, e.g., division by zero. The range of values in Equations (2.1) and (2.2) is not continuous and has non-uniform gaps due to the discrete precision, which is a consequence of having a fixed number of bits in the fractional component.

We can further discretize the range of possible values by recognizing that there is a finite number of exponents that are possible given IEEE-754 double precision, e.g.,

$$\gamma \in \{0, \pm\text{Inf}, \pm\text{NaN}, \pm 2^{-1022} \times 1.x, \pm 2^{-1021} \times 1.x, \\ \ldots, \pm 2^{0} \times 1.x, \ldots, \pm 2^{1023} \times 1.x\},$$

where $1.x$ indicates some fractional component.

Analytically, this is expressed as

$$\gamma = (-1)^{sign} \left(1 + \sum_{i=0}^{51} b_i 2^{i-52}\right) \times 2^{e-1023}, \tag{2.3}$$

for IEEE-754 *Binary64*. Note, the specification does not include a sign bit for the exponent. Rather, IEEE floating point numbers utilize a *bias* to allow the exponent to be stored without a sign bit, which we will later exploit for fault-resilience. Another important characteristic that stems from the general approach of expressing numbers in exponential notation is that we can characterize numbers by their order of magnitude. Of particular interest is the following relation:

$$\begin{aligned} &|2^{-1022}| \leq |2^{-1022} \times 1.x| \\ &< |2^{-1021}| \leq |2^{-1021} \times 1.x| < \ldots \\ &< |2^{0}| \leq |2^{0} \times 1.x| < \ldots \\ &< |2^{1023}| \leq |2^{1023} \times 1.x|. \end{aligned} \tag{2.4}$$

This means that we can use the next order of magnitude as an upper bound for errors in the fractional component of a number — which is practically achieved by incrementing the exponent or multiplying by two. We can also analytically model the number of fractional bits that could contribute an error larger than some tolerance, since the error that *could* arise from each mantissa bit is relative to the exponent of the number. This final step is necessary since

the fractional term can take values in the range $[1, 2)$, where the left parenthesis indicates that 2 is not a member of this interval. We can also characterize the error that a perturbed sign bit can contribute, and, like the fractional component, this error is relative to the exponent of the number. Suppose the sign is perturbed in a scalar γ, then we have $\tilde{\gamma} = -\gamma$, the absolute error is $|\gamma - \tilde{\gamma}| = |\gamma - (-\gamma)| = 2\gamma$. This means we can bound the error from a sign bit perturbation by incrementing the exponent of the resulting value.

In summary, we have demonstrated that errors in IEEE-754 floating point numbers can be characterized using the exponent of the numbers. This property allows us to reduce the number of bits we need to consider in a fault model, since we know that a large number of errors are bounded by the relatively small set of possible exponents.

2.4.2 Fault Characteristics of Perturbed Exponents

In the context of IEEE-754 double precision numbers and silent data corruption, we do not model the exponents directly. Instead, we model the biased exponents, as they are the interesting portion of the *data* that allows us to characterize the errors that the majority of the bits present in the data can produce. For instance, in double precision data we can characterize the errors from 53 of the 64 bits using our approach. This type of fault-characterization is impossible if bit flips are injected randomly into the data's memory, as that approach loses the semantic information that is implicitly present in the data.

The Binary64 specification does not store exponents directly, instead it uses a bias of 1023. From § 2.4.1 this means we can characterize *all* faults in double precision data by analyzing perturbations to the possible biased exponents

$$\{0, 1, 2, \ldots, 1023, \ldots, 2046\}.$$

Note that zero is not a biased exponent and has special meaning. In IEEE-754, a zero pattern in the exponent with zeros in the mantissa is used to represent the scalar zero, while a non-zero pattern in the mantissa is used to represent subnormal numbers. We also assume the user does not perform computation on the two non-numeric values NaN and Inf, which are represented using the biased exponent 2047 (all ones). We do include zero in our analysis because it is a valid real number.

Since we are concerned with bit perturbations in the exponent, we express the biased exponents in their binary form, e.g., 11-bit unsigned integers presented in binary. We can further expand Figure 2.2 to show the potential change to the original exponent should a bit flip occur, which will form the basis for our fault model and analytic models.

In the context of bit flips, we can view a bit flip as adding or subtracting from the biased exponent, which in turn translates to multiplying or dividing the number by some power of

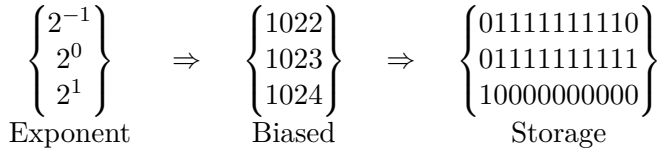


Figure 2.2: Relation of exponent, IEEE-754 double precision bias, and what data are actually stored.

two. We model the impact of a bit flip in the exponent as the original scalar being magnified or minimized by a specific power of two. We illustrate this in Figure 2.3, where reading left-to-right, we have some initial exponent, which is represented using a bias of 1023. The biased exponent translates to a discrete binary pattern. We consider all single bit flips in this binary pattern and compute the actual perturbed exponent. Note, that the perturbation can be modeled independent of the original exponent.

By characterizing the error introduced, we recognize that *all* mantissa bit flips introduce error that has the same exponent as the original number, and a sign bit flip introduces error that is only one order of magnitude larger than the original number. Furthermore, the exponent bits can either introduce large error, or a bit flip introduces error roughly equivalent to the order of magnitude of the original number.

Suppose we can enforce that all numbers used in calculations are less than 1.0, then we know that the majority of the bits will produce error that is also less than one, since 51 of the total 52 mantissa bits will contribute an error less than 1.0. We also see that some of the exponent bits have the potential to contribute an error less than 1.0, which indicates if we can enforce or assume some properties of the data used in the calculations e.g., data less than one, then we can greatly increase the likelihood that a bit flip introduces an error no greater than 1.0. This phenomena is shown in Figure 2.3, where we show empirically that numbers with exponent 2^0 introduce a small error compared to the errors introduced with the exponent 2^1.

In summary, the exponent characterizes the error introduced by the sign or mantissa should a bit flip occur. As discussed in § 2.4.1 and analytically presented in Eq. (2.4), we are able to relate bit upsets to numerical error in terms of the exponent of the original number. Table 2.1 summarizes these analytical bounds for a bit upset in a scalar and highlights the change in order of magnitude. Now that we have characterized a fault in a scalar, we will present a fault model centered around operations on scalars assuming one will be perturbed.

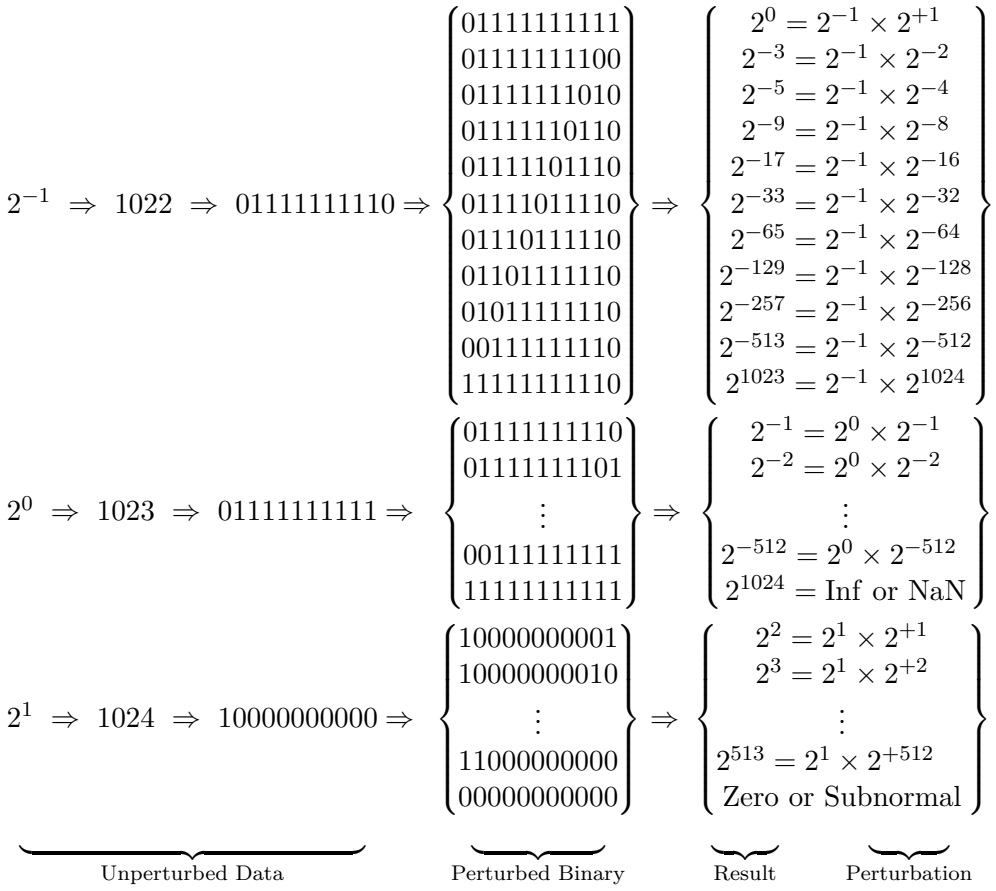


Figure 2.3: Examples of how a bit flip can impact an exponent represented using the IEEE-754 Binary64 specification.

Table 2.1: Bit flip absolute error for a scalar λ represented using IEEE-754 double precision, with $\lambda = \lambda_{\text{exp}} \times \lambda_{\text{frac}}$. Where λ_{exp} is the exponent 2^x, and λ_{frac} is the fractional component.

Bit Location	Absolute Error		Bit Range	Δ Order[†]
	Scalar: $\lvert\lambda - \tilde{\lambda}\rvert$	Multiplication: $\lvert\alpha\beta - \tilde{\alpha}\beta\rvert$		
Mantissa	$\lvert\lambda_{\text{exp}}(1 + 2^{j-52})\rvert$	$\lvert\alpha_{\text{exp}}(1 + 2^{j-52})\beta\rvert$	for $j = 0, \ldots, 51$	0
Exponent$_{1\to0}$	$\lvert\lambda(1 - 2^{-2^j})\rvert$	$\lvert\alpha\beta(1 - 2^{-2^j})\rvert$	for $j = 0, \ldots, 10$ and $bit_{j+52} = 1$	-2^j
Exponent$_{0\to1}$	$\lvert\lambda(1 - 2^{2^j})\rvert$	$\lvert\alpha\beta(1 - 2^{2^j})\rvert$	for $j = 0, \ldots, 10$ and $bit_{j+52} = 0$	$+2^j$
Sign	$\lvert 2\lambda\rvert$	$\lvert 2\alpha\beta\rvert$		1

[†] The change in order of magnitude.

2.4.3 Operation Centric Fault Model

This work distinguishes itself from related work in the field of silent data corruption by developing a fault model that is not based on perturbing arbitrary memory locations. We seek a fault model and experimental methodology that expresses all possible errors, and not the expected error, which is what is obtained through random sampling.

Fault Model for Dot Products

We now describe a realization of our fault model that describes the error that could be injected if an operation in a dot product experiences a single bit upset. We choose the dot product because it is a common operation, and because we will use this model in § 2.7 to model the worst-case errors that could be injected into a phase of the GMRES algorithm.

Given two real-valued n-dimensional vectors $\mathbf{a}, \mathbf{b} \in \mathbb{R}^n$, the dot product is defined as

$$c = \sum_{i=1}^{n} c_i, \quad \text{where} \quad c_i = a_i b_i. \tag{2.5}$$

If we allow a single bit flip to impact the i-th element of the dot product, then we have a perturbed solution $\tilde{c}$, which is the result of a perturbation to either a_i, b_i, or c_i. In the context of our fault model, this captures a bit upset impacting the inputs to the multiplication operator, and it captures a bit upset in the intermediate value, c_i, which is the input to the addition operator.

Using Table 2.1, we have all of the tools necessary to compose an absolute error model for a dot product, i.e., addition is modeled by a fault in a scalar $|\alpha + \beta - (\tilde{\alpha} + \beta)| = |\alpha - \tilde{\alpha}|$. The potential change in order of magnitude is paramount. Consider an exponent flip from $1 \to 0$. These types of exponent bit flips produce an error that is bounded above by the original magnitude of the result, which can be viewed as "zeroing out" the term if a perturbation occurs. Similar to a perturbed scalar, the mantissa can contribute either no change in the order of magnitude, or in the worst case a bit flip causes a carry, which will increment the order of magnitude by one. The order of magnitude for a sign bit flip is exactly the same as that of a perturbed scalar, which introduces an error one order of magnitude larger than the result. These error models can be thought of as the largest additive error that we can inject into a dot product from a bit flip, e.g.,

$$\tilde{c} = \sum_{i=1}^{n} a_i b_i + (\text{error term}). \tag{2.6}$$

In summary, we have composed analytic models for the the absolute error that could be introduced into a dot product. Our models are initially constructed from the IEEE-754 Binary64 model, which we extended to express how a bit upset impacts a singular double precision scalar.

We then composed a model for the multiplication operator, and analytically expressed the absolute error. Using the absolute error, we have a model that explains *how wrong* a dot product can be, assuming a bit flip in one of the input vectors or in an intermediate value. Next, we refine these models to construct strict upper bounds on the error introduced by a bit flip in a dot product.

Error Bounds for a Bit Flip in a Dot Product

The models presented in Table 2.1 make no assumptions about the bits present in the mantissa of the operands. This is problematic if we want to consider *all* possible errors that could be introduced into a dot product. To account for the mantissa, and to create strict upper bounds on the error, we will use the relation presented in Eq. 2.4. From this relation, we know that $\alpha\beta < 2^{\alpha_{\text{exponent}}+1}2^{\beta_{\text{exponent}}+1}$. We can write this as

$$\alpha\beta < 4\alpha_{\text{exp}}\beta_{\text{exp}}, \tag{2.7}$$

where $\alpha_{\text{exp}} = 2^{\alpha_{\text{exponent}}}$. Using Eq (2.7), we are able to account for the mantissa bits, but we can also show that a bit flip in the sign is bounded by Eq. (2.7). The sign bit introduces an absolute error equivalent to incrementing the exponent of the result

$$\alpha\beta < 2\alpha\beta < 4\alpha_{\text{exp}}\beta_{\text{exp}}, \tag{2.8}$$

where $2\alpha\beta$ is the potential error introduced should the sign bit be perturbed, which must be smaller than the bound constructed for the mantissa.

By utilizing Eq. (2.7), we are able to account for all possible mantissas and their potential faults, as well as a perturbation to the sign bit. We will now discuss how to use this model to understand the relationship between the data in an algorithm and the distribution of potential errors that could occur should a bit flip in the data.

2.5 Fault Model Evaluation

In § 2.4 we proposed analytic models for errors should a bit flip occur in IEEE-754 double precision data. We now illustrate how data can impact the size of errors that a bit flip can

create. Consider the following sample vectors

$$\mathbf{u}_{\text{small}} = \begin{bmatrix} 0.5 \\ 0.25 \end{bmatrix}, \ \mathbf{v}_{\text{small}} = \begin{bmatrix} 0.25 \\ 0.5 \end{bmatrix}, \text{ and}$$
$$\mathbf{u}_{\text{large}} = \begin{bmatrix} 2 \\ 4 \end{bmatrix}, \qquad \mathbf{v}_{\text{large}} = \begin{bmatrix} 4 \\ 2 \end{bmatrix}.$$

If we compute the dot product $\lambda = \mathbf{u}_{\text{large}} \cdot \mathbf{v}_{\text{large}}$, we have a finite number of potential errors should a bit flip in the data of $\mathbf{u}_{\text{large}}, \mathbf{v}_{\text{large}}$, or in an intermediate value in the summation. We can experience either $\tilde{2} \times 4 + 4 \times 2$, $2 \times \tilde{4} + 4 \times 2$, or $\tilde{8} + 8$. We have previously shown what $\tilde{2}$ can be (in Figure 2.3), but for clarity we will state what the perturbed values could be (in Figure 2.4). By inspection it is clear that substituting any of the above perturbed scalars into

$$\tilde{2} = \left\{ \begin{matrix} 2^2 \\ 2^3 \\ 2^5 \\ 2^9 \\ 2^{17} \\ 2^{33} \\ 2^{65} \\ 2^{129} \\ 2^{257} \\ 2^{513} \\ \text{Zero} \end{matrix} \right\}, \quad \tilde{4} = \left\{ \begin{matrix} 2^1 \\ 2^4 \\ 2^6 \\ 2^{10} \\ 2^{18} \\ 2^{34} \\ 2^{66} \\ 2^{130} \\ 2^{258} \\ 2^{514} \\ 2^{-1020} \end{matrix} \right\}, \quad \tilde{8} = \left\{ \begin{matrix} 2^4 \\ 2^1 \\ 2^7 \\ 2^{11} \\ 2^{19} \\ 2^{35} \\ 2^{67} \\ 2^{131} \\ 2^{259} \\ 2^{515} \\ 2^{-1018} \end{matrix} \right\}$$

Figure 2.4: Example of perturbed values for large numbers.

the dot product will produce an absolute error greater than one in all cases, and in the event one chooses to substitute the near zero perturbed values, the absolute error of the dot product still has magnitude 8, e.g., $|16 - (0 + 8)|$.

Alternatively, consider the vectors $\mathbf{u}_{\text{small}}$ and $\mathbf{v}_{\text{small}}$. If we compute the dot product, $\lambda = \mathbf{u}_{\text{small}} \cdot \mathbf{v}_{\text{small}} = 0.25$. Then we have possible values to perturb: $\widetilde{0.5}$, $\widetilde{0.25}$, and $\widetilde{0.125}$. We construct these from our model of a perturbed scalar, and present the perturbed variants in Figure 2.5. By inspection, $\widetilde{0.5}$ can contribute an absolute error to the dot product larger than one only once, e.g., $|0.25 - (2^{1022} \times 0.25 + 0.125)|$. Likewise, $\widetilde{0.25}$ and $\widetilde{0.125}$ can perturb the result of the dot product with error greater than one only once, and for all 3 cases the perturbation will change the result by hundreds of orders of magnitude.

$$\widetilde{0.5} = \left\{ \begin{matrix} 2^{0} \\ 2^{-3} \\ 2^{-5} \\ 2^{-9} \\ 2^{-17} \\ 2^{-33} \\ 2^{-65} \\ 2^{-129} \\ 2^{-257} \\ 2^{-513} \\ 2^{1022} \end{matrix} \right\}, \widetilde{0.25} = \left\{ \begin{matrix} 2^{-3} \\ 2^{0} \\ 2^{-6} \\ 2^{-10} \\ 2^{-18} \\ 2^{-34} \\ 2^{-66} \\ 2^{-130} \\ 2^{-258} \\ 2^{-514} \\ 2^{1019} \end{matrix} \right\}, \widetilde{0.125} = \left\{ \begin{matrix} 2^{-2} \\ 2^{-1} \\ 2^{-7} \\ 2^{-11} \\ 2^{-19} \\ 2^{-35} \\ 2^{-67} \\ 2^{-131} \\ 2^{-259} \\ 2^{-515} \\ 2^{1017} \end{matrix} \right\}$$

Figure 2.5: Example of perturbed values for small numbers.

Returning to Figure 2.3 explains what causes bit flips in the exponent to produce either a majority of large or small errors. The binary pattern of the stored biased exponent contains predominantly zeros for numbers greater than one, and predominantly ones for numbers less than one.

One could also obtain primarily ones in the exponent as you approach the extrema of the biased exponents, i.e., numbers larger than 2^{512}. In this case, the biased exponent does contain many ones, however, because the number is sufficiently large, i.e., the absolute error will remain considerably large. This is because if one "zeroes out" a perturbed element in the dot product, the error is proportional to the magnitude of the result.

2.5.1 Faults in the Mantissa or Sign

The error generated by the mantissa or sign bits is relative to the exponent of the number that the flip occurred in. If the exponent is larger than one, then clearly the mantissa or sign bits can generate an error larger than one. Alternatively, if the values all are less than one, then mantissa errors will produce errors less than one because $2^{-1} \times 1.x \leq 2^{0}$. The errors from the sign bit cannot exceed 2 since $2 \times 2^{-1} \times 1.x < 2^{1}$.

It is reasonable to consider that the mantissa generates a carry, as discussed in § 2.4.3. To account for this we construct a strict upper bound by incrementing the exponent of each element of the vectors analyzed, similar to Eq. (2.7). For example,

$$\mathbf{u}_{\text{original}} = \begin{bmatrix} 2.12332 \\ 1.24568 \end{bmatrix} \;\Rightarrow\; \mathbf{u}_{\text{upper bound}} = \begin{bmatrix} 4 \\ 2 \end{bmatrix}. \tag{2.9}$$

We then can evaluate our models on these vectors to determine a strict upper bound on the errors we can experience in a dot product.

2.5.2 Modeling Large Vectors

We have shown how to exhaustively examine each element in a vector, and from this analysis we can determine precisely which absolute errors we could experience. Given large vectors, where the dimension n may have millions or billions of elements, exhaustively searching each element would be time consuming, but it would also be a waste of time. As stated previously, there is a discrete number of exponents supported by the IEEE-754 Binary64 specification. As we have previously shown, the exponent characterizes the faults we can observe, so we only need to consider the 2046 possible biased exponents and the special case of zero. The perturbations that are possible can be determined independent of concrete data values, e.g., we can precompute the perturbations and absolute error because we know the relation stated in Eq. (2.4) and Eq. (2.7).

To analyze arbitrarily large vectors, we construct a lookup table for the absolute error in whatever operation we choose to model (we have chosen products and addition). The table size is 2047×2047, and allows us to consider the error introduced by performing an operation on two exponents, which will map to a unique ij location.

For example, consider the vectors

$$\mathbf{u} = \begin{bmatrix} 1.0 \\ 1.2 \\ 8.0 \\ 0.125 \end{bmatrix}, \text{ and } \mathbf{v} = \begin{bmatrix} 0.125 \\ 0.125001 \\ 0.125002 \\ 1.0 \end{bmatrix}. \tag{2.10}$$

We first extract the biased exponents from the vectors

$$\mathbf{u} \Rightarrow \mathbf{u}_{\text{exponent}} = \begin{bmatrix} 2^0 \times 1.0 \\ 2^0 \times 1.x \\ 2^3 \times 1.0 \\ 2^{-3} \times 1.0 \end{bmatrix} \Rightarrow \mathbf{u}_{\text{biased}} = \begin{bmatrix} 1023 \\ 1023 \\ 1026 \\ 1020 \end{bmatrix} \tag{2.11}$$

Now, we determine an interval of possible values, and account for the mantissa values that may have been truncated

$$u_i \in [1020, 1026] \subseteq [1020, 1027] \text{ for } i = 1, \ldots, 4. \tag{2.12}$$

The range of biased exponents $[1020, 1027]$ will contain all possible values that the original vector contained, and include one value that was larger than any in the vector, the number

corresponding to the biased exponent 1027. Similarly, we can compute the interval for $\mathbf{v}$

$$\mathbf{v} = \begin{bmatrix} 0.125 \\ 0.125001 \\ 0.125002 \\ 0.25 \end{bmatrix} \Rightarrow \begin{bmatrix} 2^{-3} \times 1.0 \\ 2^{-3} \times 1.x \\ 2^{-3} \times 1.x \\ 2^{-2} \times 1.0 \end{bmatrix} \Rightarrow \begin{bmatrix} 1020 \\ 1020 \\ 1020 \\ 1021 \end{bmatrix}, \tag{2.13}$$

which leads to the interval we consider errors

$$v_i \in [1020, 1021] \subseteq [1020, 1022] \text{ for } i = 1, \ldots, 4. \tag{2.14}$$

To allow us to analyze intervals efficiently, we create a lookup table, where each entry computes the relevant perturbations and absolute errors for the operations being modeled. In the case of multiplication, the table has symmetry because multiplication is commutative. In practice, computing the full table $(0, \ldots, 2046)$ is simple and allows one to model errors for arbitrary vectors.

A caveat of the above approach is that we must know the range of values that the vector contains. This can be achieved by directly computing the min and max values for each vector. Alternatively, an approximate range can be determined if the "length" of the vector is known, e.g., the two-norm or *if we know that the data is normalized*, i.e., the two-norm is one. One weakness to the proposed approach is that we do not consider a flip in the accumulating sum, which we have left to future work. We also leave to future work analysis that shows how many of these modeled errors lie within the rounding error bound for pairwise sums.

2.5.3 Summary

We have shown that the range of values used in the dot product has a direct impact on the size of the errors that can be observed. A general rule in floating point algorithms has been to perform operations on numbers as close to the same magnitude as possible, as doing so minimizes the loss of precision. We have now shown that following this rule-of-thumb also gives the benefit of making bit upsets generate a relatively small error when the numbers are no larger than one. Next we present a motivating case study that focuses exclusively on dot products, and then in § 2.7 we show how to exploiting data scaling in an iterative solver.

2.6 Case Study: Vector Dot Products

To begin our investigation, we assess the susceptibility of the dot product of two N-dimensional vectors to a silent bit flip in arithmetic. We make this choice since many linear algebra operations

can be decomposed into dot products, for example, Gram-Schmidt orthogonalization or matrix-vector multiplications.

2.6.1 Computational Challenges

Given a single double-precision number, there are 64 bits that may be flipped. Extending this to an N-dimensional vector, we have $64N$ bits that are candidates for flipping. Accounting for different numbers results in a very large sample space, and, therefore, we utilized a hybrid CPU-GPU cluster and created a parallel code that farms out specific Monte Carlo trials to various nodes. In this context, a trial consists of creating two vectors, which is discussed in the following section, and then determining pass/fail statistics given a bit flip on the input to the dot-product kernel. We utilized the BLAS *ddot*() routine, and aggregated the output for post-processing in MATLAB. Ensuring a sampling error of less than 0.001, which is discussed in § 2.6.4, required nearly 400,000 CPU hours parallelized over the processors of a 1700 core cluster of AMD 6128 Opterons. The large search space coupled with ensuring statistically significant results highlights why an analytic approach is not only more efficient, but may be required for more advanced methods and data structures, e.g., matrices and linear solvers.

2.6.2 Monte Carlo Sampling

We next develop a better understanding of how vector magnitudes impact the expected absolute error should a bit perturb a dot product. To conduct Monte Carlo sampling, we must first determine a mechanism for tallying success, and we must define *success and failure*.

- **Vector Creation**

1) Mantissa generated randomly using C stdlib rand().

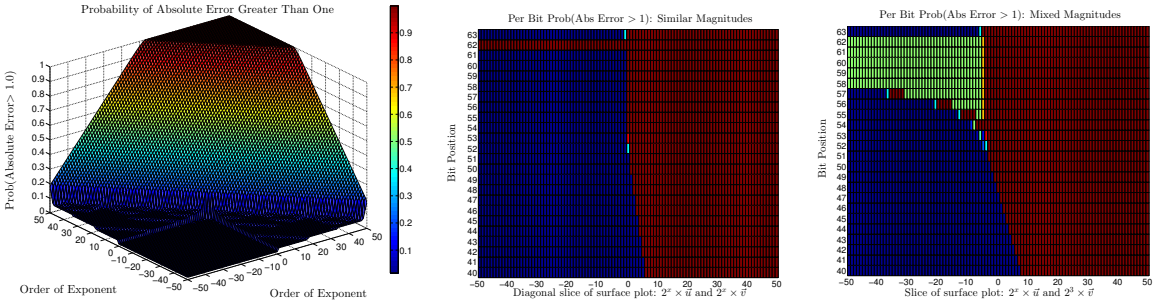
2) For each vector, we fix each element's magnitude to the bit pattern 2^{-50} to 2^{50} (101 bit patterns). This corresponds to the base ten numbers in the range 8.8×10^{-16} to $1.1 \times 10^{+15}$. This range was chosen because 2^{-50} roughly is the machine precision. The numbers in this range are utilizing the highest *precision* that Binary64 offers.

- **Sample definition and Error Calculation**

1) A random sample is defined by generating two random N length vectors and computing the absolute error considering all possible $2 \times 64 \times N$ bit flips.

2) A tally is defined by failure, which we define to be any absolute error that is greater than 1.

3) An empirical estimate of the expected absolute error is computed by dividing the number of failures by the number of bits considered times the vector length times 2 times the number of random samples (M) taken for a given magnitude combination, i.e., $failures/(2 \times 64 \times N \times M)$.



(a) Monte Carlo experiment computing dot products with vectors of various magnitudes. Failure is defined to be an absolute error larger than 1.

(b) Dot product with vectors containing similar magnitudes.

(c) Dot product containing mixed magnitudes.

Figure 2.6: Probability of observing an absolute error larger than 1.

- **Visualization**

1) To visualize the expected absolute error, we construct tallies for each magnitude combination, i.e., 101×101 unique combinations, and each combination is sampled M times.
2) We summarize this information in a surface plot, where the x- and y-axes denote the $\log_2$ of the relative magnitude of the vector $\mathbf{u}$ and $\mathbf{v}$, respectively. The height of the surface plot indicates the probability of seeing an absolute error larger than 1.

Figure 2.6a presents a surface plot as described in the Visualization bullet. To interpret this graph, the x-axis indicates the magnitude that all elements of the vector $\mathbf{u}$ were forced to have while the mantissa was randomly generated. Likewise, the y-axis indicates the magnitude that all elements of the vector $\mathbf{v}$ were forced to have. Each x-y intersection represents 1,000,000 random vector samples, where the dot product was computed and failures tallied. The height of the surface at an (x, y) location indicates the probability of observing an absolute error larger than 1 given a single bit flip. From this surface, one may immediately recognize the unusual structure of these graphs: When both vectors have magnitudes larger than 2^0, the probability of failure is noticeably higher; yet, when both vectors have magnitudes less than or euqal to 2^0, the probability of failure is approaching zero.

The key finding presented in Figure 2.6a, is that when we operate on vectors that are normalized, e.g., values in the range $[0, 1]$, we have a very low probability of seeing a large error should a bit flip occur. The lowest probability, i.e., the flat region in the quadrant $[0, -50] \times [0, -50]$, is precisely $\text{Prob}(\text{Abs Error} > 1) = 0.015625$, which is $1/64$. The single bit that can introduce absolute error larger than one is the most significant exponent bit. Also, should the most significant exponent bit flip, the error is quite large and can be detected [29].

2.6.3 Per Bit Analysis of Surface Plot

To better understand the structure of the surface plot, we take two slices of the surface and look at the per-bit probability of a failure (Figures 2.6b and 2.6c). The slices chosen feature dot products of vectors with *similar* relative magnitudes and dot products of vectors of many magnitudes (the x-axis) with a vector that contains magnitudes up to 2^3. Intuitively, these figures slice from the back-most corner of Figure 2.6a to the front for similar magnitudes (Figure 2.6b), and they slice from the left to right for Figure 2.6c.

We have shown why this shape should be expected in Figure 2.3, and in the example presented in § 2.5. This feature is an artifact of how the exponent is implemented in the IEEE-754 specification, i.e., a biased exponent. The lowest probability presented in the surface plot is $0.015625 = 1/64$, we can graphically show this in Figure 2.6b, where one can see that bit #62 (2^{nd} from the top), is the only bit that can contribute large error. We also show that the sign and mantissa bits can not introduce large error when values are in the range $[0, 1]$.

Conversely, Figure 2.6c shows that when mixing large and small values, we expect to see large errors for faults. The green shading in Figure 2.6c (upper left quadrant) indicates a roughly 50% chance that we see an absolute error larger than one. The reason for this is that values larger than 2 have a binary pattern that introduces large error most of the time (recall Figure 2.4).The increased likelihood of large error from the large numbers, coupled with the low chance from small numbers, creates a scenario where it is equally likely to see both large and small absolute errors. The more we deviate from operating on numbers in the range $[0, 1]$, the closer we get to having a 50/50 chance of seeing a large error (see the mantissa bits slowly becoming green as well).

2.6.4 Comparison of the Analytic Model and Monte Carlo Sampling

In Figure 2.7, we compare the error observed while performing Monte Carlo sampling with the expected error computed from our model. We sampled up to $M = 1$ million random vectors per data point, which implies a Monte Carlo error of $error_{MC} = 1/\sqrt{M} \approx 0.001$. We observe a perfect fit, which is to be expected because we have analytically shown that the exponent bits dictate the size of the absolute error we will observe. Even with random sign and mantissa bits evaluated, we see that the likelihood of experiencing a large error is entirely determined by the exponent bits.

2.7 Extension to Matrices and Iterative Solvers

Having recognized that dot products on numbers less than one can produce errors less than one, we will relate this idea to matrix equilibration. We then provide an example of how to use this

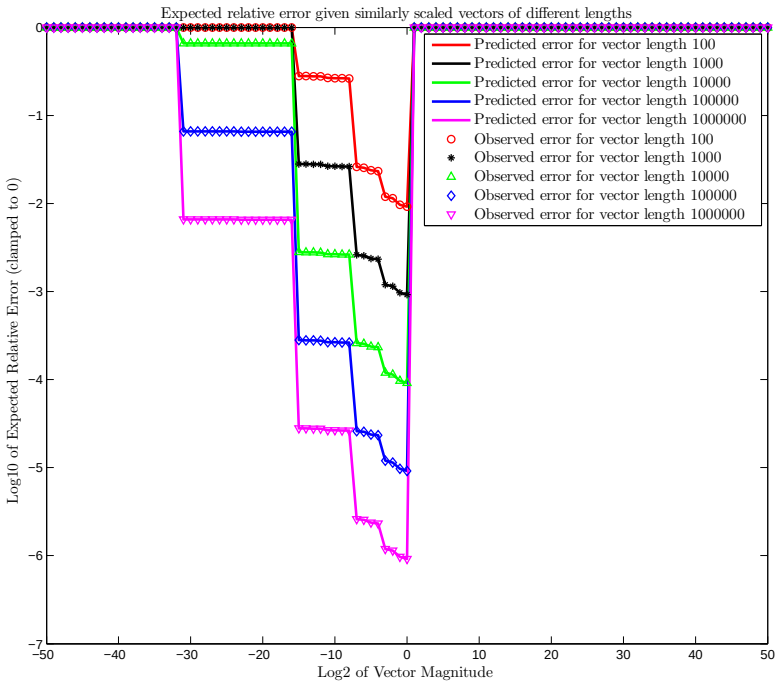


Figure 2.7: Comparison of observed error caused by a flip in the exponent, excluding the most significant bit, for sampled vector sizes having similar relative magnitudes.

concept in an sparse iterative solver (GMRES), while exhaustively counting the possible errors that can be introduced.

2.7.1 Matrix Equilibration

The idea of scaled vectors is analogous to vector normalization, i.e., $\|\mathbf{u}\|_2 = 1$. Applied to matrices in the context of solving linear systems, scaling takes the form of *matrix equilibration*: for a matrix $\mathbf{A}$, scale the rows and columns such that $\|\mathbf{A}\|_\infty = 1$. Scaling can also be performed before a matrix is created, for example the equations leading to the matrix can be scaled prior to assembling a matrix. To scale a sparse matrix after its creation, we use a sparse matrix implementation of LAPACK's equilibration routine DGEEQU [3]. Equilibration does not cause fill, i.e., it will not increase the number of non-zeros. In general, equilibrating a matrix is only beneficial, but equilibration may not be practical in all cases.

2.7.2 GMRES

The Generalized Minimum Residual method (GMRES) of Saad and Schultz [78] is a Krylov subspace method for solving large, sparse, possibly non-symmetric linear systems $\mathbf{Ax} = \mathbf{b}$.

GMRES is based on the Arnoldi process [5], which uses orthogonal projections and basis vectors normalized to length one. Arnoldi and GMRES relate to this work because the orthogonalization phase of Arnoldi is often Modified Gram-Schmidt or Classical Gram-Schmidt, which are dot product heavy kernels.

We present the GMRES algorithm in Algorithm 2.1. The Arnoldi process is expressed on Lines 3–14 in Algorithm 2.1. At its core is the Modified Gram-Schmidt (MGS) process, which constructs a vector orthogonal to all previous basis vectors $\mathbf{q}_i$. The MGS process begins on Line 5 and completes on Line 8. We now describe how we instrument the orthogonalization phase and count the absolute errors that *could* be injected.

Algorithm 2.1 GMRES

Input: Linear system $\mathbf{Ax} = \mathbf{b}$ and initial guess $\mathbf{x}_0$
Output: Approximate solution $\mathbf{x}_m$ for some $m \geq 0$

1: $\mathbf{r}_0 := \mathbf{b} - \mathbf{Ax}_0$ ▷ Initial residual vector
2: $\beta := \|\mathbf{r}_0\|_2$, $\mathbf{q}_1 := \mathbf{r}_0/\beta$
3: **for** $j = 1, 2, \ldots$ until convergence **do**
4: $\quad \mathbf{v}_{j+1} := \mathbf{Aq}_j$ ▷ Apply the matrix A
5: $\quad$ **for** $i = 1, 2, \ldots, j$ **do** ▷ Orthogonalize
6: $\quad\quad h_{i,j} := \mathbf{q}_i \cdot \mathbf{v}_{j+1}$
7: $\quad\quad \mathbf{v}_{j+1} := \mathbf{v}_{j+1} - h_{i,j}\mathbf{q}_i$
8: $\quad$ **end for**
9: $\quad h_{j+1,j} := \|\mathbf{v}_{j+1}\|_2$
10: $\quad$ **if** $h_{j+1,j} \approx 0$ **then**
11: $\quad\quad$ Solution is $\mathbf{x}_{j-1}$ ▷ Happy breakdown
12: $\quad\quad$ **return**
13: $\quad$ **end if**
14: $\quad \mathbf{q}_{j+1} := \mathbf{v}_{j+1}/h_{j+1,j}$ ▷ New basis vector
15: $\quad \mathbf{y}_j := \arg\min_y \|\mathbf{H}(1{:}j+1, 1{:}j)\mathbf{y} - \beta\mathbf{e}_1\|_2$
16: $\quad \mathbf{x}_j := \mathbf{x}_0 + [\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_j]\mathbf{y}_j$ ▷ Compute solution update
17: **end for**

2.7.3 Instrumentation and Evaluation

To demonstrate the benefit of data scaling we have chosen 3 test matrices. We instrument the code and for each dot product in the orthogonalization phase we determine an interval that describes the range of values possible in the vectors. Then using our fault model, we compute the absolute errors that are possible. Since we know the basis vectors ($\mathbf{q}_i$) are normal, the intervals for the values in the vectors are $(-1, 1)$. We compute the min and max for the unknown vector

v, and this determines the interval for the values in **v**. We use the intervals and our fault model to evaluate all absolute errors that can be introduced from a single bit flip in the input vectors. We classify the absolute error into four classes:

1. Absolute error less than 1.0,
2. Absolute error greater than or equal to 1.0, but less than or equal to $\|\mathbf{A}\|_2$,
3. Absolute error greater $\|\mathbf{A}\|_2$.
4. Error that is non-numeric, e.g., Inf or NaN.

We choose to include the 2$^{\text{nd}}$ class of errors due to recent work by by Elliott et al. [29] that demonstrates how to use a norm bound on the Arnoldi process to filter out large errors in orthogonalization.

Classes 1 and 2 are *undetectable*, while Classes 3 and 4 are detectable. Our goal is to ensure that should a bit flip, the error falls into Classes 1, 3, and 4 while minimizing or eliminating the occurrence of Class 2 errors. We refer to Class 2 errors as the *grey area*, as they are undetectable errors that we consider to be large.

Sample Problems

We have chosen three sample matrices to demonstrate our technique. To ensure reproducibility, we did not create any of these matrices from scratch, rather we used readily available matrices. The first matrix arises from a second-order centered finite difference discretization of the Poisson equation. We generated this matrix using MATLAB's built-in Gallery functionality. The second matrix, CoupCons3D, presents a more realistic linear system. It comes from the University of Florida Sparse Matrix Collection [24] and arises from a fully coupled poroelastic problem. The matrix is symmetric in pattern, but not symmetric in values. It is also fairly large, and has explicitly stored zero values. The matrix is poorly scaled, with a mixture of large and small values. The final matrix, mult_dcop_03, is also from the Florida Sparse Matrix Collection. It arises from a circuit simulation problem, and has good scaling inherently. We have summarized the characteristics of each matrix in Table 2.2.

We now scale the Poisson and CoupCons3D matrices and right-hand side vectors such that they are equilibrated. Table 2.3 summarizes the norms for each of our test matrices. We use the infinity norm ($\|A\|_\infty \approx 1$) to measure whether a matrix is well scaled. One can see that both the Poisson and mult_dcop_03 matrices have infinity norms not too much larger than one, while the CoupCons3D matrix is inherently poorly scaled. Our equilibration code ran out of memory when attempting to equilibrate mult_dcop_03, but it is already well scaled.

Table 2.2: Sample Matrices

Properties	Poisson100	CoupCons3D	mult_dcop_03
number of rows	10,000	416,800	25,187
number of columns	10,000	416,800	25,187
nonzeros	49,600	17,277,420	193,216
structural full rank	yes	yes	yes
explicit zero entries	0	5,044,916	0
type	real	real	real
structure	symmetric	nonsymmetric	nonsymmetric
positive definite	yes	no	no

Table 2.3: Norms of Sample Matrices †

Norm	Poisson Equation		CoupCons3D	
	No Scaling	Scaling	No Scaling	Scaling
$\|\mathbf{A}\|_\infty$	8.0	2.0	1.30×10^6	1.0
$\|\mathbf{A}\|_2$	7.999	1.999	1.20×10^6	1.0
$\|\mathbf{A}\|_F$	4.46×10^2	1.12×10^2	2.75×10^6	2.91×10^2

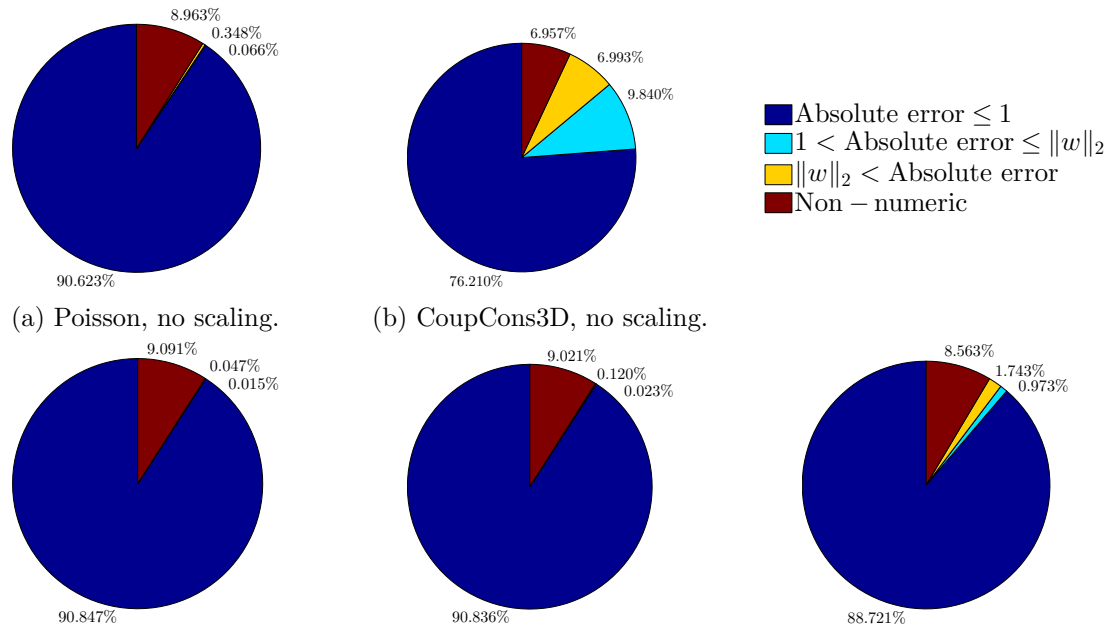
† mult_dcop_03 has $\|\mathbf{A}\|_\infty = 35.5$.

2.7.4 Results

We ran Algorithm 2.1 for 1000 total iterations, using a restart value of 25. By instrumenting the code, we determined the numerical range of values each vector contained, and then computed the possible absolute error that a bit flip could introduce. We classified the absolute error according to § 2.7.3, and counted each class of errors for the duration of the algorithm. Figure 2.8 shows how these errors map to our classes of errors when the matrices are scaled versus not scaled.

A large proportion of the absolute errors possible in orthogonalization fall into Class 1 (undetectable and small). We can explain this distribution given that the vectors $\mathbf{q}_i$ are normalized (a side effect of GMRES being derived from the Arnoldi process). Given normalized vectors, we know that of all the dot products in Gram-Schmidt orthogonalization, at least one of the vectors has data in the interval $(-1, 1)$. We previously established that the interval $(-1, 1)$ aids in minimizing absolute error if a bit perturbs a dot product. Now, we show how equilibrating the input matrices can assist in forcing the non-normalized vector $(\mathbf{v}_{j+1})$ as close as possible to being in the normalized interval.

The results show the benefits of using well-scaled matrices. Figures 2.8a and 2.8e show the Poisson problem (with no equilibration) and the mult_dcop_03 matrices, which both have good



(a) Poisson, no scaling. (b) CoupCons3D, no scaling.

(c) Poisson, equilibrated (d) CoupCons3D, equilibrated. (e) mult_dcop_03, pre-scaled.

Figure 2.8: Number of possible absolute errors from dot products in Algorithm 1 in orthogonalization kernel. Class 1: $err < 1.0$ (blue), Class 2: $1.0 \geq err \leq \|\mathbf{A}\|_2$ (light blue), Class 3: $\|\mathbf{A}\|_2 > err$ (yellow), and Class 4: Non-numeric (red).

scaling (see Table 2.3). These problems experience a higher distribution of absolute errors less than one than the poorly scaled CoupCons3D matrix (see Figure 2.8b). For the matrices that can be equilibrated, we see that scaling the input matrices is never detrimental, and will only improve fault tolerance, e.g., compare CoupCons3D before scaling in Figure 2.8b versus after scaling in Figure 2.8d.

The pie charts are not probabilistic, that is, they do not convey the likelihood of observing such an error. Rather, these charts characterize the possible errors when given specific data. Consider an arbitrary length vector $\mathbf{x}$, we can determine the range of values in the vector, e.g., $x_i \in [a, b]$, but we do not know how many of each value, or in what order they occur. Obtaining fine-grained statistics would involve evaluating every element of the vector, or constructing a probabilistic model that captures the distribution of values in each vector.

Since we consider the impact of a single bit flip, it is sufficient to follow the methodology presented in § 2.5. That is, we may not know the distribution and order of numbers in the vectors, but we can model every possible error by assuming that each value in the interval *could* be used in an operation with *every* value of the other interval. This Cartesian product guarantees that we have counted all possible errors for IEEE-754 double precision numbers in an

interval, including errors that may not occur because the vector does not contain that specific number, or because of the ordering.

Error Distribution

Our results show that scaling tends to produce a distribution of absolute error that is roughly 91% less than or equal to one, while 9% are non-numeric. This is expected when *most* of the numbers are near one. Flipping the most significant exponent bit produces 11111111111, which will generate a non-numeric value. Similarly, the 10 remaining exponent bits will produce error less than one — that is, $1/11 \approx 9\%$ and $10/11 \approx 91\%$. As previously discussed, the mantissa errors are determined entirely by the exponent bits.

2.7.5 Multiple Bit Flips

While this work intentionally focuses on single bit flips, the key finding that normalized data is *better* to operate on when performing dot products, gives some insight into how multiple bit flips in data may behave. For example, we know that a fault in the fractional component of a floating point number will produce an absolute error bound above by the order of magnitude of the original value. That is, we could flip all 52 bits of the mantissa and the error bound from our model would still be valid (e.g., Eq. (2.8) or Eq.(2.4)). In regard to exponent flips, we have shown both analytically and experimentally that when operating on normalized values, only 1 exponent bit per 64 bit value can introduce large error. Should the values all be normalized, flipping $1 \rightarrow 0$ will minimize the value subject to Table 2.1. Experiencing more than a single bit flip would only serve to "shrink" the value even more. We intentionally do not speculate about how and why multiple bit flips can occur, but we have shown that operating on normalized values skews the probability of experiencing large error.

2.8 Conclusion

Our results indicate a clear benefit to good scaling. We have shown that a widely used numerical method (the Arnoldi process coupled with Gram-Schmidt orthogonalization) inherently minimizes absolute error in dot products. Furthermore, standard matrix equilibration algorithms can be used to scale input matrices, which further enhance the inherent robustness of the Arnoldi process. We demonstrated our theoretical finding experimentally by instrumenting the GMRES iterative solver, which is based on the Arnoldi process.

We cannot enforce that data are always normalized. Some linear systems may be inherently poorly scaled, or it may be impractical to equilibrate them. We *can* advocate that scaling, while typically used to improve numerical stability and reduce the loss of precision, can also

benefit fault resilience. We have shown that this result has broad applicability, because many iterative solvers are based on orthogonal projections using normalized vectors, i.e., they create an orthonormal basis. While this work does not propose an end-to-end solution to soft errors, it does indicate that data scaling can help mitigate the impact of such errors should they occur.

Chapter 3

Model Driven Analysis of Faulty IEEE-754 Scalars

A trend in algorithm-based fault tolerance work has been to propose an algorithmic strategy, and then inject synthetic bit flips into data operated on and present either the resulting runtime or whether the final result was correct or incorrect. Examples of this methodology include a large number of related algorithm-based fault tolerance works [14, 58, 94, 23, 22, 17, 82, 83, 79, 13]. The approaches proposed are not the motivation for our work, but the fault model used to motivate and assess such works is. A common theme in algorithm-based fault tolerance is the detection and correction of errors. This is a logical approach, if an algorithm has a mechanism for knowing if its state is corrupted, then it can do something to remedy the problem. We refer to this as fine-grained detection, as the approaches attempt to recover corrupt scalars. For example, Huang and Abraham [58] proposes a scheme for detecting corruption in two dense matrices that are being multiplied together or corruption in the resulting solution matrix. To evaluate the technique, random bits are flipped in some entries of the the matrix operands or solution matrix. This approach is motivated by soft errors in the early years of scientific computing, and the detection/correction technique presented is still actively researched as a means to guard various linear algebra operations. Davies and Chen [22] propose an approach that attempts to detect and correct soft errors (bit flips) in the LU factorization of a matrix. Wu and Chen [94] present an approach for detecting and correction soft errors in Cholesky, QR, and LU factorization. Shantharam et al. [83] propose a detection/correction scheme for a sequential conjugate gradient solver.

In all works listed, the goal is to identify if an operand or result have been silently corrupted by a bit (or bits) being flipped. In the cases of [58, 82, 83, 79, 13] the corruption is introduced into a matrix, which is then multiplied with another matrix [58] or vector [82, 83, 79, 13]. At

the scalar level, this can be modeled as a faulty operand to multiplication and the corrupted result is then accumulated in a summation.

Unfortunately, few researchers have considered the characteristics of the errors that random bit flip injection introduces. This is troubling, because, as we show, the errors introduced can vary drastically. For example, Bronevetsky and de Supinski [14] injected bit flips into random locations of various numerical kernels used in scientific computing and found that for *all* kernels tested they tended to abort roughly 10-12% of the time. We will show that given random bit flip injection, the expected value of the relative error is large approximately 10% of the time. Furthermore, we will show statistics that show how the expected absolute and relative errors behave given random bit flip injection into IEEE-754 scalars. We extend our models to multiplication and show that the expected error (both absolute and relative) can be forced to behave predictably using the standard numerical operations of normalization and matrix equilibration.

The point is that bit flips introduced into the representation of floating point data will behave in two broad categories: The expected error is either small (less than one) or it can extremely large. We show this in three ways: 1) we observe this effect experimentally by injecting bit flips into dot products and tallying the errors that fall into the two categories; 2) we expose this effect by using numerical analysis to bound errors arising from a bit flip in different components of the IEEE-754 representation; 3) we derive this effect by modeling the expected absolute and relative error using the analytic model of IEEE-754 scalars. These analyses are presented in Chapter 2 for (1) and (2), and Chapter 3 for (3).

The ways errors are distributed is important when it comes to assessing ABFT techniques, because detectors may be incapable of detecting relatively small errors. The overhead introduced can also depend on the properties of the fault. For example, in Chapter 4 we show that large errors tend to cause more iterations than small errors. Through statistical analysis, we show that errors will be relatively small most of the time and large infrequently. This broad range of possible errors means that random injection is introducing small errors most of the time.

We also show that the distribution of errors can be skewed by standard numerical techniques such as normalization and matrix equilibration. Based on the high variability of the absolute and relative error models, we argue that researchers should instead introduce errors that are known to be detectable, while clearly indicating which errors are undetectable. It is naïve to assume by default, that an error detector will detect all possible errors. Instead, we advocate a research methodology that shows overheads given detectable errors, and also shows the overhead required to obtain a solution with a desired accuracy when an undetectable error is introduced.

Table 3.1: Terminology

a, b	IEEE-754 scalars.
α	Exponent of scalar. The desired exponent value, e.g., 2^{-7} or 2^5.
β	Raw mantissa of scalar.
$1.\beta$	Complete mantissa $(1 + \beta)$.
$1.\xi$	Error from a bit flip in the complete mantissa $(1 + \xi)$.
$\tilde{a}$	Scalar that experiences a fault.
e	Biased exponent, the unsigned integer value stored in the exponent bits.
$bias$	Bias used in exponent storage.
η	Error introduced from a corrupt biased exponent.
η^+	η values with a positive sign.
η^-	η values with a negative sign.
N	Number of mantissa bits.
M	Number of mantissa values 2^N.
ϵ	2^{-N}.
Z	Number of exponent bits.
K	Number of biased exponents.

3.1 Errors in IEEE-754 Representation

Consider a scalar represented using the IEEE-754 specification,

$$a = (-1)^{sign} \left(1 + \sum_{i=0}^{N-1} b_i 2^{i-N}\right) \times 2^{e-bias}. \tag{3.1}$$

We will analyze scalars represented using Eq. (3.1) by decomposing them into an unsigned form

$$a = \alpha \times 1.\beta, \tag{3.2}$$

where $\alpha = 2^{e-bias}$ and $\beta = \sum_{i=0}^{N-1} b_i 2^{i-N}$. The notation $1.\beta$ is shorthand for $1+\beta$. The specification depends on two parameters: The number of mantissa bits N, and the number of exponent bits Z. Table 3.1 summarizes our notation and terminology. Common implementations of the IEEE-754 specification are listed in Table 3.2 as well as the parameters that can be used to generate our models. While we list various implementations, this work uses *binary64* for examples and experiments.

This analysis is different from floating point rounding analysis, e.g., Higham [54]. The most notable difference is that values can be changed drastically and bit flips do not carry. A question this work does not address is: Which bit flips (and under what conditions) would be masked by

Table 3.2: Common IEEE-754 Implementations

Common Name	Spec. Name	Mantissa Bits (N)	Exponent Bits (Z)
Half Prec.	binary16	10	5
Single Prec.	binary32	23	8
Double Prec.	binary64	52	11
Quad Prec.	binary128	112	15

rounding effects. However, like rounding analysis the relative and absolute errors are used to measure the error introduced.

The mantissa value, $1.\beta$ is bounded above by

$$\begin{aligned} 1.\beta &= 1 + \sum_{i=1}^{N} 2^{-i} \\ &= 1 + \sum_{i=0}^{N-1} ar^i; \text{ for } a = r = 1/2 \\ &= 2 - 2^{-N} \\ &= 2 - \epsilon. \end{aligned}$$

The smallest representable value larger than 1.0 is $1 + 2^{-N}$, and 2^{-N} is often referred to as ϵ (or machine epsilon). The smallest mantissa value is obtained by letting all bits be zero, yielding $1.\beta = 1.0$. The range of the mantissa values is

$$1.0 \leq 1.\beta < 2.0. \tag{3.3}$$

The exponent value is not stored as a signed integer. Rather, it uses an offset, called the bias, allowing the exponent bits to represent signed and unsigned values by subtracting a bias from the stored *biased exponent*. For binary64 values, there are 11 exponent bits allowing unsigned integers in the range of $[0, 2047]$ to be stored. The specification uses two integers from this range to allow special values to be stored. A biased exponent containing all zeros is used to represent subnormal values, which are values with an exponent value of 2^{1-bias} and a mantissa value of $(0 + \beta)$ rather than $(1 + \beta)$. A biased exponent containing all ones indicates either a Not-a-Number (NaN) or an infinity (Inf) if all mantissa bits are zero. The total number of representable exponent values K is

$$K = 2^Z - 2. \tag{3.4}$$

$$\underset{\text{Exponent}}{\begin{Bmatrix} 2^{-1} \\ 2^{0} \\ 2^{1} \end{Bmatrix}} \Rightarrow \underset{\text{Biased}}{\begin{Bmatrix} 1022 \\ 1023 \\ 1024 \end{Bmatrix}} \Rightarrow \underset{\text{Storage}}{\begin{Bmatrix} 01111111110 \\ 01111111111 \\ 10000000000 \end{Bmatrix}}$$

Figure 3.1: Exponent values, biased exponent values, and storage.

Adjusting the biased exponent range to allow for the special values, the unsigned integer bits can take integer values in the range $[1, 2046]$. To allow the exponents to be signed, the range is divided in two, which allows the representation of values in the range $[1, 1023]$. This is expressed analytically as

$$\begin{aligned} bias &= 2^{Z}/2 - 1 \\ &= 2^{Z-1} - 1. \end{aligned} \tag{3.5}$$

The bias value for an 11 bit exponent is 1023. Examples of exponents and biased exponents are shown in Figure 3.1. The representable exponents, (α) in our terminology, are the powers of two in the range

$$\alpha \in \{2^{-1022}, 2^{-1021}, \ldots, 2^{1023}\}. \tag{3.6}$$

The biased exponents e take integer values in the range $[1, 2046]$, or

$$e = \sum_{i=0}^{Z-1} b_i \times 2^i, \tag{3.7}$$

where b_i is the i-th exponent bit. The desired exponent value *alpha* can then be written as $\alpha = 2^{e-bias}$.

3.1.1 Mantissa

Consider a corrupted scalar, $\tilde{a}$. For corruption affecting the mantissa, the corruption can be treated algebraically, regardless of the number of bits affected.

$$\begin{aligned} \tilde{a} &= \alpha \times 1.\tilde{\beta} \\ &= \alpha \times (1.\beta \pm 1.\xi) \\ &= \alpha \times 1.\beta \pm \alpha \times 1.\xi \\ &= a \pm \alpha \times 1.\xi \end{aligned} \tag{3.8}$$

The error term $1.\xi$ belongs to a discrete set of values

$$1.\xi \in \{1.0 + 2^{i-N}\} \text{ for } i = 0, \ldots, N-1. \tag{3.9}$$

The error introduced from a single bit flip is bounded by analyzing the largest and smallest mantissa perturbations. The smallest possible value for ξ is the smallest power of two that can be stored, e.g., $1 + 2^{-N} = 1 + \epsilon \approx 1.0$. The largest value is then $1 + 2^{-1} = 1.5$. A mantissa error from a single bit flip is bounded by

$$1.0 < 1.\xi \leq 1.5. \tag{3.10}$$

3.1.2 Exponent

Suppose a corrupted scalar, $\tilde{a}$, experiences a fault that introduces an error into the exponent bits. An exponent bit flip is really a bit flip in the *biased exponent*. That is

$$\tilde{a} = \tilde{\alpha} \times 1.\beta. \tag{3.11}$$

Expanding $\tilde{\alpha}$,

$$\begin{aligned} \tilde{\alpha} &= 2^{\tilde{e}-bias} \\ &= 2^{e \pm \eta - bias} \\ &= 2^{e-bias} \times 2^{\pm\eta} \\ &= \alpha \times 2^{\pm\eta}. \end{aligned} \tag{3.12}$$

Hence,

$$\tilde{a} = a \times 2^{\pm\eta} \tag{3.13}$$

The error term $2^{\pm\eta}$ is key. Recall, the biased exponent e is an unsigned integer value stored in the exponent bits, as shown in Eq. (3.7). Given a single bit flip in the j-th bit,

$$\begin{aligned} \tilde{e} &= \sum_{i=0}^{Z-1} b_i \times 2^i \pm 2^j \\ &= e \pm \eta. \end{aligned}$$

The multiplicative error term is then $2^{\pm 2^j}$. Given a single bit flip and Z exponent bits, there are $2Z$ possible values for the error term's exponent

$$\pm\,\eta \in \{\pm 2^0, \pm 2^1, \pm 2^2, \ldots, \pm 2^{Z-1}\}. \tag{3.14}$$

Table 3.3: Statistics Terminology

X_*	A random variable (R.V.).
X_β	R.V. taking values of the mantissa ($1.\beta$).
X_ω	R.V. taking values of the reciprocal of the mantissa ($\frac{1}{1.\beta}$).
X_ξ	R.V. taking values of the absolute error introduced into the mantissa ($1.\xi$).
X_α	R.V. taking values of the true exponent (α).
X_η	R.V. taking values of the error introduced by a bit flip in the biased exponent (2^η).
$\mathbb{E}[X]$	The expected value of X.
$\text{Var}(X)$	The variance of X.
$\text{Cov}(X, Y)$	The covariance.

Recognize that error term's exponent takes values that are positive or negative. A plus indicates a bit that is flipped $0 \to 1$, while a minus indicates a bit was flipped $1 \to 0$. Because the error is multiplicative, the sign of η determines if the scalar's exponent is increased (addition in the exponent) or decreased (subtraction in the exponent). For example, given a scalar with exponent $\alpha = 2^0$, the biased exponent is $e = 1023 = 011\ 1111\ 1111$. A bit flip in the lowest order exponent bit toggles a $1 \to 0$. This introduces an error $2^{-\eta}$ with $\eta = 2^0$, or $\alpha \times 2^{-2^0} = \alpha \times 2^{-1}$.

3.1.3 Sign

For an error impacting the sign bit,

$$\tilde{a} = -a. \tag{3.15}$$

3.2 Model Statistics

Using Eq. (3.1) and the form presented in Eq. (3.11), we now analyze statistics for each component and two error measures. The goal is to derive analytic representations of the expected value for discrete operations, and then to determine how the error measures behave when perturbed with a bit flip. Table 3.3 summarizes the variables and notation used for our statistical analysis.

3.2.1 Mantissa

The number of possible values representable using N bits is $M = 2^N$. Recognize that the set of all mantissa values is

$$\begin{aligned} 1.\beta &\in \{1.0, 1+\epsilon, 1+2\epsilon, \ldots, 1+(M-1)\epsilon\} \\ 1.\beta &\in \left\{ 1.0 + \sum_{j=i}^{M} 2^{-N} \right\} \text{ for } i = 1, \ldots, M \\ 1.\beta &\in \{1.0 + 2^{-N}(M-i)\} \text{ for } i = 1, \ldots, M \end{aligned} \tag{3.16}$$

Let X_β be a discrete random variable that takes values from Eq. (3.16) with equally likely probability. The expected value of the mantissa is

$$\begin{aligned} \mathbb{E}[X_\beta] &= \frac{1}{M} \sum_{i=1}^{M} \left[1 + \sum_{j=i}^{M} 2^{-N} \right] \\ &= \frac{1}{M} \sum_{i=1}^{M} \left[1 + 2^{-N}(M-i) \right] \\ &= \frac{1}{M} \sum_{i=1}^{M} \left[1 + 2^{-N}2^N - 2^{-N}i \right] \\ &= \frac{1}{M} \sum_{i=1}^{M} \left[2 - 2^{-N}i \right] \\ &= \frac{1}{M} \left[2M - 2^{-N} \sum_{i=1}^{M} i \right] \\ &= \frac{1}{M} \left[2M - 2^{-N} \frac{M(M+1)}{2} \right] \\ &= 2 - 2^{-N} \frac{M+1}{2} \\ &= 2 - \frac{1 + 2^{-N}}{2} \\ &= 2 - \frac{1}{2} - \frac{2^{-N}}{2} \\ &= 1.5 - \frac{\epsilon}{2} \end{aligned} \tag{3.17}$$

Alternatively, consider the continuous interval $[1, 2]$, the expected value is $\int_1^2 x \mathrm{d}x = \frac{1}{2}x^2|_1^2 = 1.5$.

The variance of the mantissa is defined as

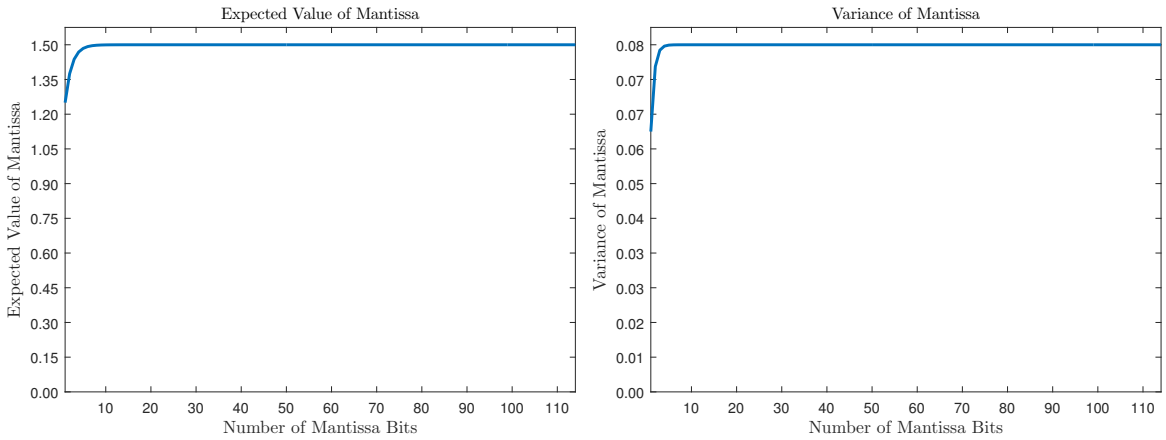
$$\mathrm{Var}(X_\beta) = \mathbb{E}\Big[X_\beta^2\Big] - \mathbb{E}[X_\beta]^2. \tag{3.18}$$

The expected value of X_β^2 is

$$\begin{aligned}
\mathbb{E}\Big[X_\beta^2\Big] &= \frac{1}{M}\sum_{i=1}^{M}\left[1+\sum_{j=i}^{M}2^{-N}\right]^2 \\
&= \frac{1}{M}\sum_{i=1}^{M}\Big[2-2^{-N}i\Big]^2 \\
&= \frac{1}{M}\sum_{i=1}^{M}\Big[4-2^{2-N}i+2^{-2N}i^2\Big] \\
&= \frac{1}{M}\left[4M-2^{2-N}\frac{M(M+1)}{2}\right. \\
&\quad \left.+\,2^{-2N}\frac{M(M+1)(2M+1)}{6}\right] \\
&= 4-2^{1-N}(M+1)+2^{-2N}\frac{(M+1)(2M+1)}{6} \\
&= 4-2-2^{1-N}+\frac{2^{-2N}(2M^2+3M+1)}{6} \\
&= 2-2^{1-N}+\frac{2+3\times 2^{-N}+2^{-2N}}{6} \\
&= 2-2^{1-N}+\frac{2}{6}+\frac{3}{6}2^{-N}+\frac{1}{6}2^{-2N} \\
&= 2-2^{1-N}+\frac{1}{3}+\frac{1}{2}2^{-N}+\frac{1}{6}2^{-2N} \\
&= \frac{7}{3}-2\epsilon+\frac{1}{2}\epsilon+\frac{1}{6}\epsilon^2 \\
&= \frac{7}{3}-\frac{3}{2}\epsilon+\frac{1}{6}\epsilon^2,
\end{aligned} \tag{3.19}$$

and the square of the expectation of $1.\beta$ is

$$\begin{aligned}
\mathbb{E}[X_\beta]^2 &= \left(1.5-\frac{\epsilon}{2}\right)^2 \\
&= \frac{9}{4}-\frac{3}{2}\epsilon+\frac{1}{4}\epsilon^2.
\end{aligned} \tag{3.20}$$



(a) Expected value of the mantissa given N bits. (b) Variance of the mantissa given N bits.

Figure 3.2: Expected value and variance of the an N-bit mantissa.

Substituting Eqs. (3.19) and (3.20) into Eq.(3.18) yields the variance of the mantissa

$$\begin{aligned} \text{Var}(X_\beta) &= \mathbb{E}\left[X_\beta^2\right] - \mathbb{E}[X_\beta]^2 \\ &= \frac{1}{12} - \frac{1}{12}\epsilon^2. \end{aligned} \tag{3.21}$$

Recognize that Eq. (3.21) is an increasing function that rapidly converges to $\frac{1}{12}$. Figure 3.2a plots the expected value of the mantissa as a function of the number of bits used to store the mantissa, while Figure 3.2b shows the variance as function of the number of bits used.

3.2.2 Mantissa Errors

Equation (3.9) shows the set of unscaled mantissa single bit errors. Let X_ξ be a discrete random variable taking values from Eq. (3.9) with equally likely probability. The expected value for an

N-bit mantissa is

$$\begin{aligned}
\mathbb{E}[X_\xi] &= \frac{1}{N}\sum_{i=1}^{N}\left(1+2^{-i}\right) \\
&= \frac{1}{N}\left(N+\sum_{i=1}^{N}2^{-i}\right) \\
&= \frac{1}{N}\left(N+\sum_{i=0}^{N-1}ar^{i}\right);\ \text{for } a=r=1/2 \\
&= \frac{1}{N}\left(N+1.0-2^{-N}\right) \\
&= 1+N^{-1}-\frac{\epsilon}{N}. \qquad (3.22)
\end{aligned}$$

The variance of a mantissa error is

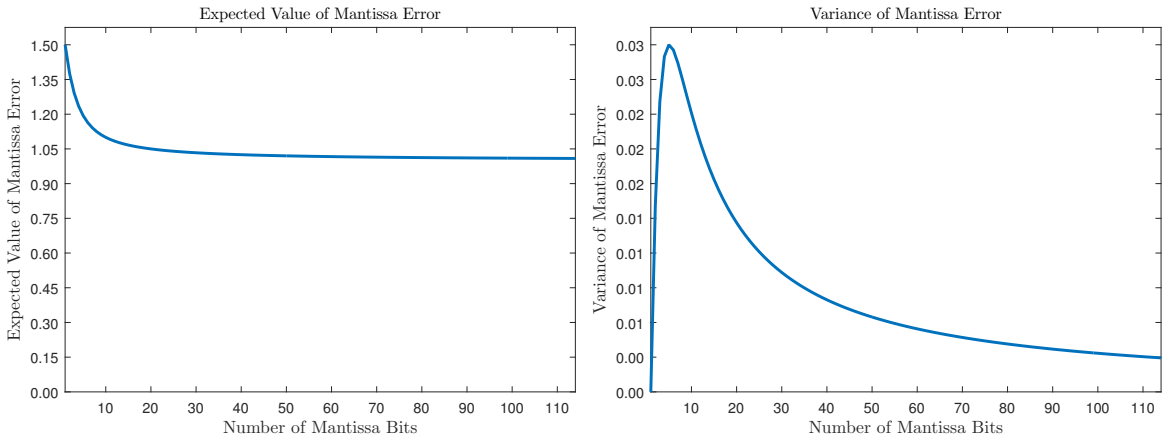
$$\mathrm{Var}(X_\xi) = \mathbb{E}\Big[X_\xi^2\Big] - \mathbb{E}[X_\xi]^2. \qquad (3.23)$$

The expected value of X_ξ^2 is

$$\begin{aligned}
\mathbb{E}\Big[X_\xi^2\Big] &= \frac{1}{N}\sum_{i=1}^{N}\left(1+2^{-i}\right)^2 \\
&= \frac{1}{N}\sum_{i=1}^{N}\left(1+2^{1-i}+2^{-2i}\right) \\
&= \frac{1}{N}\left(N+2[1-\epsilon]+\frac{1}{3}(1-\epsilon^2)\right) \\
&= \frac{1}{N}\left(N+2-2\epsilon+\frac{1}{3}-\frac{\epsilon^2}{3}\right) \\
&= 1+\frac{7}{3N}-\frac{2\epsilon}{N}-\frac{\epsilon^2}{3N}, \qquad (3.24)
\end{aligned}$$

and the squared expected value is

$$\begin{aligned}
\mathbb{E}[X_\xi]^2 &= \left[1+\frac{1}{N}-\frac{\epsilon}{N}\right]^2 \\
&= 1+2\frac{1-\epsilon}{N}+\frac{(1-\epsilon)^2}{N^2} \\
&= 1+\frac{2}{N}+\frac{1}{N^2}-\frac{2\epsilon}{N}-\frac{2\epsilon}{N^2}+\frac{\epsilon^2}{N^2}. \qquad (3.25)
\end{aligned}$$



(a) Expected value of the absolute error.

(b) Variance of the absolute error.

Figure 3.3: Expected value and variance of the absolute error from a single bit flip in an N-bit mantissa.

The variance is obtained by substituting Eqs. (3.24) and (3.25) into Eq. (3.23)

$$\mathrm{Var}(X_\xi) = \frac{1}{3N} + \frac{2\epsilon}{N^2} - \frac{1}{N^2} - \frac{\epsilon^2}{3N} - \frac{\epsilon^2}{N^2}. \tag{3.26}$$

Figure 3.3 plots the expected error and the variance of the error that is introduced if a bit is flipped in the mantissa. Figure 3.3a plots the expected value of X_ξ as a function of the number of bits used to represent the mantissa. Figure 3.3b plots the variance as a function of the number of mantissa bits (N). Note the differences in the y-axes.

Recognize that when $N = 1$, $\mathrm{Var}(X_\xi) = 0$, and as $N \to \infty$, the variance increases before tending towards 0. This increase may be seen by inspecting the derivative with respect to N. The variance reaches a maximum at $N = 5$. As $N \gg 5$, the variance decreases towards zero, and the expected value converges to 1.0. The important fact is that the expected mantissa error is approximately 1.0, and the variance is roughly zero. Because the variance is small, and the mean converges to one, we consider the mantissa errors to be "well behaved".

3.2.3 Exponent

Let X_α be a discrete random variable taking values from Eq. (3.6). The number of representable exponents is K (see Eq. (3.4)). Recall, because the exponent bits store an unsigned integer, K

represents an unsigned count of representable exponents. The expected value of X_α is

$$\begin{aligned}
\mathbb{E}[X_\alpha] &= \frac{1}{K}\sum_{i=1}^{K} 2^{i-bias} \\
&= \frac{1}{K}2^{-bias}\sum_{i=1}^{K} 2^{i} \\
&= \frac{1}{K}2^{-bias}\left[\sum_{i=0}^{K} 2^{i} - 1\right] \\
&= \frac{1}{K}2^{-bias}\left[2^{K+1} - 2\right] \\
&= \frac{1}{K}\left[2^{K+1-bias} - 2^{1-bias}\right]
\end{aligned}$$

Recognize from Eq. (3.5) that K can be written as a function of the $bias$, $K = 2 \times bias$. The expected value may then be expressed as

$$\begin{aligned}
\mathbb{E}[X_\alpha] &= \frac{1}{K}\left[2^{bias+1} - 2^{1-bias}\right] \\
&= \frac{1}{bias}\left[2^{bias} - 2^{-bias}\right].
\end{aligned} \tag{3.27}$$

Recognize the expected value is dominated by 2^{bias}, yielding, an approximation

$$\mathbb{E}[X_\alpha] \approx \frac{2^{bias}}{bias}.$$

Comparing Eq. (3.27) to the continuous expected value on the interval $[2^{-1022}, 2^{1023}]$,

$$\begin{aligned}
\int_{-1022}^{1023} 2^x \mathrm{d}x &= \left.\frac{2^x}{\ln 2}\right|_{-1022}^{1023} \\
&= \frac{2^{bias}}{\ln 2}.
\end{aligned} \tag{3.28}$$

The continuous expected value provides an upper bound on the discrete approximation, that is, $\frac{2^{bias}}{bias} < \frac{2^{bias}}{\ln 2}$.

The variance of the exponent is

$$\mathrm{Var}(X_\alpha) = \mathbb{E}\left[X_\alpha^2\right] - \mathbb{E}[X_\alpha]^2 \tag{3.29}$$

with expected values

$$\begin{aligned}
\mathbb{E}\left[X_\alpha^2\right] &= \frac{1}{K}\sum_{i=1}^{K}\left(2^{i-bias}\right)^2 \\
&= \frac{2^{-2bias}}{K}\sum_{i=1}^{K}2^{2i} \\
&= \frac{2^{-K}}{K}\sum_{i=1}^{K}4^{i} \\
&= \frac{2^{-K}}{K}\left[\frac{4^{K+1}-1}{3}-1\right] \\
&= \frac{2^{-K}}{3K}\left[4^{K+1}-4\right] \\
&= \frac{2^{2-K}}{3K}\left[2^{2K}-1\right] \\
&= \frac{2^{K+2}}{3K}-\frac{2^{2-K}}{3K}.
\end{aligned} \tag{3.30}$$

$$\begin{aligned}
\mathbb{E}[X_\alpha]^2 &= \left[\frac{1}{K}\left(2^{K+1-bias}-2^{1-bias}\right)\right]^2 \\
&= \frac{1}{K^2}\left[2^{K+2}-2^3+2^{2-K}\right].
\end{aligned} \tag{3.31}$$

Substituting Eqs. (3.30) and (3.31) in Eq. (3.29) yields the variance

$$\begin{aligned}
\mathrm{Var}(X_\alpha) &= \mathbb{E}\left[X_\alpha^2\right]-\mathbb{E}[X_\alpha]^2 \\
&= \frac{2^{K+2}}{3K}-\frac{2^{2-K}}{3K}-\frac{1}{K^2}\left[2^{K+2}-2^3+2^{2-K}\right] \\
&= \frac{2^{K+2}}{3K}-\frac{2^{K+2}}{K^2}-\frac{2^{2-K}}{3K}+\frac{2^3}{K^2}-\frac{2^{2-K}}{K^2}.
\end{aligned} \tag{3.32}$$

Recognize the final three terms in Eq. (3.32) are small. The variance of X_α will be dominated by

$$O\left(\mathrm{Var}(X_\alpha)\right) = \frac{2^{K+2}}{3K}-\frac{2^{K+2}}{K^2}. \tag{3.33}$$

To understand the growth of Eq. (3.33) we analyze the limit as K goes to infinity

$$O\left(\mathrm{Var}(X_\alpha)\right) = \lim_{k\to\infty^+}\frac{2^{K+2}K-2^{K+2}3}{3K^2}.$$

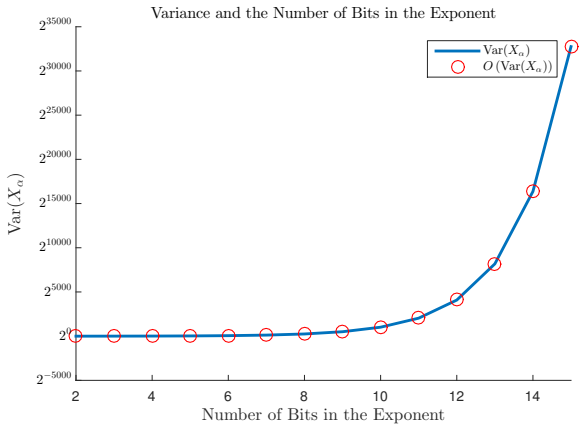


Figure 3.4: Variance of the exponent and the dominant term in the growth of the variance as a function of the number of exponent bits.

With two applications of L'Hôpital's rule, we obtain

$$O\left(\mathrm{Var}(X_\alpha)\right) = \frac{2^{K+2}\ln^2(2)(K-3) + 2^{K+2}\ln(2)\times 2}{6}. \tag{3.34}$$

The limit as $K \to \infty$ is ∞, and the dominant term in the summation is $2^{K+2}\ln(2)/3$, or $2^{2\times bias+2}\ln(2)/3$. Figure 3.4 plots the upper bound on the variance as a function of the number of bits used to represent the exponent, as well as the variance. The variance is large, which means that values are not clustered near the mean. The standard deviation (σ) is the square root of the variance. Recognize that $2^{2\times bias+2} = (2^{bias+1})^2$, hence $\sigma = 2^{bias+1}$. That is, σ is approximately the mean. Compare this to the exponential distribution, where $\mathbb{E}\left[e^{-\lambda x}\right] = 1/\lambda$, and the variance is $(1/\lambda)^2$. That is, our model follows the similar continuous exponential distribution, with the variance approximately the square of the mean.

3.2.4 Exponent by Range

The expected value of the exponent falls into two broad ranges: values with positive exponents, i.e., the biased exponent is larger than the bias, and negative exponents, where the biased exponent is smaller than the bias. Partitioning α in two sets $\alpha^+ \in \{2^0, 2^1, \ldots 2^{bias}\}$ and $\alpha^- \in \{2^{-1}, \ldots, 2^{1-bias}\}$. We assign 2^0 to the positive set, because the resulting value is not a fraction.

Let X_{α^+} be a discrete random variable that takes values from the set of positive exponents with equally likely probability, and let X_{α^-} be a discrete random variable that takes values from the negative set of exponents with equally likely probability. The expected value of the positive

exponents is

$$\begin{aligned}\mathbb{E}[X_{\alpha^+}] &= \frac{1}{bias+1}\sum_{i=0}^{bias+1} 2^i \\ &= \frac{2^{bias+1}-1}{bias+1}.\end{aligned} \tag{3.35}$$

The expected value of the negative exponents is

$$\begin{aligned}\mathbb{E}[X_{\alpha^-}] &= \frac{1}{bias-1}\sum_{i=1}^{bias-1} 2^{-i} \\ &= \frac{1-2^{1-bias}}{bias-1}.\end{aligned} \tag{3.36}$$

Recall that Z is the number of bits used to store biased exponents. To implement the specification, Z must be at least 2, because two values are reserved (zero and 2^Z-1). This requires at least 2 bits, otherwise, no exponents are representable. To store a negative exponent,

$$Z \geq 3, \tag{3.37}$$

because with $Z=2$, only exponents with zero and +1 are representable. Hence, with $Z \geq 3$, $bias \geq 3$ and the term $\frac{2^{1-bias}}{bias-1} < 1.0$. The expected value can then be bounded as

$$\mathbb{E}[X_{\alpha^-}] < \frac{1}{bias-1}. \tag{3.38}$$

The point of deriving Equations (3.35) and (3.36) will become clearer when we apply these findings to draw high-level conclusions about injecting a bit flip into the representation of a floating point value. Specifically, $\mathbb{E}[X_\alpha]$ will continue to be a dominant term in most errors models. Once we have analyzed specific error models, we will show how a small subset of the total possible bit flips can contribute excessively larger error relative to all other bit flips.

3.2.5 Exponent Errors

The expected error introduced from a bit flip in the exponent is the expected value of 2^η. Let X_η be a discrete random variable taking values from Eq. (3.14) with equally likely probability. Partition this set into positive and negative subsets, and let X_{η^+} be a R.V. that takes values from the positive set $\{2^0, \ldots, 2^{Z-1}\}$ and let X_{η^-} be a R.V. that takes values from the negative set $\{-2^0, \ldots, -2^{Z-1}\}$.

$$\begin{aligned}\mathbb{E}[X_\eta] &= \frac{1}{2Z}\sum_{i=0}^{Z-1} 2^{2^i} + \frac{1}{2Z}\sum_{i=0}^{Z-1} 2^{-2^i} \\ &= \frac{1}{2}\left[\mathbb{E}\left[X_{\eta^+}\right] + \mathbb{E}\left[X_{\eta^-}\right]\right]. \end{aligned} \tag{3.39}$$

The form 2^{2^n} has been studied extensively, as $F_n = 2^{2^n} + 1$ is a Fermat number, which are studied in relation to prime numbers. No closed form exists for the sum of the Fermat numbers or of its reciprocals.

Positive Exponent Set

The expected value of the positive set $\mathbb{E}\left[X_{\eta^+}\right]$ can be bounded recognizing that

$$\sum_{i=0}^{Z-1} 2^{2^i} < \sum_{j=0}^{2^{Z-1}} 2^j = 2^{2^{Z-1}+1} - 1. \tag{3.40}$$

Eq. (3.40) exploits the relation $2^{G+1} - 1 = \sum_{k=0}^{G} 2^k$, where G is a positive integer. A lower bound may be constructed by recognizing that

$$2^{2^{Z-1}} < \sum_{i=0}^{Z-1} 2^{2^i}, \text{ for } Z \geq 2. \tag{3.41}$$

The inequality in Eq. (3.41) is strict, because $Z \geq 2$. This gives a bound of

$$2^{2^{Z-1}} < \sum_{i=0}^{Z-1} 2^{2^i} < 2^{2^{Z-1}+1} - 1.$$

The expected value is then bounded by

$$\begin{aligned}\frac{2^{2^{Z-1}}}{Z} < \mathbb{E}\left[X_{\eta^+}\right] < \frac{2^{2^{Z-1}+1} - 1}{Z}, \\ \frac{2^{bias+1}}{Z} < \mathbb{E}\left[X_{\eta^+}\right] < \frac{2^{bias+2} - 1}{Z}. \end{aligned} \tag{3.42}$$

Figure 3.5 plots the expected value and its lower and upper bounds. The upper bound is constructed by summing all positive powers of two, and using this sum as a strict upper bound. The upper bound is an approximation of the continuous expected value $\int_0^{bias+1} 2^x \mathrm{d}x$, which is a good approximation of $\mathbb{E}[X_\alpha]$, as shown in Eq. (3.28). This shows then when

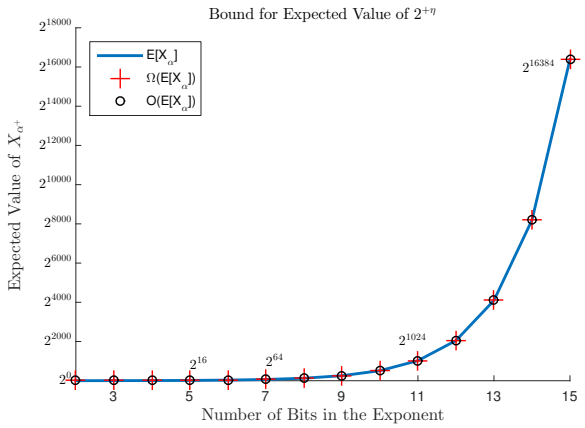


Figure 3.5: Lower and upper bounds for the expected value of a positive exponent error $\left(\mathbb{E}\left[X_{\eta^+}\right]\right)$.

summing exponentially distributed values, the large values will dominate the expected value, $\mathbb{E}\left[X_{\eta^+}\right] \approx 2^{bias}$, which is approximately the mean of the exponents $\mathbb{E}[X_\alpha]$ (see Eq. (3.27)).

Negative Exponent Set

The second set of potential exponent errors fall in the class of negative values for η. The expected value of interest is

$$\mathbb{E}[X_{\eta^-}] = \frac{1}{Z}\sum_{i=0}^{Z-1} 2^{-2^i}.$$

The sum has been shown to be less than one [1]. This gives an upper bound of

$$\mathbb{E}[X_{\eta^-}] < \frac{1}{Z}. \tag{3.43}$$

Exponent Error Expected Value

The expected value of the exponent errors is obtained by substituting back into Eq. (3.39). An upper bound on the expected value is obtained by substituting Eq. (3.40) for $\mathbb{E}\left[X_{\eta^+}\right]$

$$\begin{aligned}\mathbb{E}[X_\eta] &< \frac{1}{2}\left[\frac{2^{2^{Z-1}+1}-1}{Z} + \frac{1}{Z}\right] \\ &< \frac{1}{2}\left[\frac{2^{bias+2}}{Z}\right] \\ &< \frac{2^{bias+1}}{Z}.\end{aligned}$$

A lower bound on the expected value is obtained by substituting Eq. (3.42) for $\mathbb{E}\Big[X_{\eta^+}\Big]$

$$\begin{aligned}\mathbb{E}[X_\eta] &> \frac{1}{2}\left[\frac{2^{2^{Z-1}}}{Z} + \frac{1}{Z}\right] \\ &> \frac{1}{2}\left[\frac{2^{bias+1}}{Z}\right] \\ &> \frac{2^{bias}}{Z}.\end{aligned}$$

Bounding the exponent errors as

$$\frac{2^{bias}}{Z} < \mathbb{E}[X_\eta] < \frac{2^{bias+1}}{Z} \tag{3.44}$$

Clearly, the expected value of an exponent error is dominated by the positive exponent set.

3.2.6 Summary of Scalar Statistics

We summarize the statistics for each component of the model in Table 3.4. These are models for the correct and faulty components of the IEEE-754 representation when perturbed by a single bit flip. In § 3.4, we use these terms to compose statistics for each error measure, and then extend this analysis to specific operations. Key observations are: 1) Mantissa errors are well-behaved, having an expected error of approximately one, with small variance. 2) The expected exponent error, considering both positive and negative exponents, is approximately the expected value of the exponent range. If a bit flips in the exponent $0 \rightarrow 1$ (2^{η^+}), then the expected error is approximately the expected value of the *entire* exponent range. If a bit flips in the exponent $1 \rightarrow 0$ the expected error is less than one. Practically, this means that if bits are flipped at random and each bit is equally likely to be a one or zero, then the expected error introduced will be the expected value of the entire (or positive) exponents.

3.3 Model Error Measures

We now model the absolute and relative errors for a scalar a represented following Eq. (3.2). The absolute error ($error_{\text{abs}}$) presents the actual error that a bit flip introduces

$$error_{\text{abs}} = |a - \tilde{a}|.$$

Table 3.4: Expected value for components of faulty scalars.

Term	R.V.	$\mathbb{E}[\cdot]$	Ref.
$1.\beta$	X_β	$1.5 - \frac{\epsilon}{2}$	Eq. (3.17)
$1.\xi$	X_ξ	$1 + N^{-1} - \frac{\epsilon}{N}$	Eq. (3.22)
α	X_α	$\frac{1}{bias}\left[2^{bias} - 2^{-bias}\right]$	Eq. (3.27)
Pos. α	X_{α^+}	$\frac{2^{bias+1}-1}{bias+1}$	Eq. (3.35)
Neg. α	X_{α^-}	$< \frac{1}{bias-1}$	Eq. (3.38)
2^η	X_η	$\frac{2^{bias}}{Z} < \mathbb{E}[X_\eta] < \frac{2^{bias+1}}{Z}$	Eq. (3.44)
2^{η^+}	X_{η^+}	$\frac{2^{bias+1}}{Z} < \mathbb{E}\left[X_{\eta^+}\right] < \frac{2^{bias+2}-1}{Z}$	Eq. (3.42)
2^{η^-}	X_{η^-}	$< \frac{1}{Z}$	Eq. (3.43)

The relative error ($error_{\text{rel}}$) indicates how large an error is, relative to the correct value

$$error_{\text{rel}} = \frac{|a - \tilde{a}|}{|a|}.$$

3.3.1 Mantissa

Equation (3.8) presents the form of $\tilde{a}$. The absolute error is

$$\begin{aligned} |a - \tilde{a}| &= |a - a \pm \alpha \times 1.\xi| \\ &= |\alpha \times 1.\xi|, \end{aligned} \tag{3.45}$$

and the relative error is

$$\begin{aligned} \frac{|a - \tilde{a}|}{|a|} &= \frac{|\alpha \times 1.\xi|}{|\alpha \times 1.\beta|} \\ &= \frac{1.\xi}{1.\beta}. \end{aligned} \tag{3.46}$$

Note, we may drop the absolute value as the sign is the same between both the perturbed scalar and the correct.

Table 3.5: Error measures for scalars.

Location	Absolute Error	Ref	Relative Error	Ref
Mantissa	$\lvert\alpha \times 1.\xi\rvert$	Eq. (3.45)	$\frac{1.\xi}{1.\beta}$	Eq. (3.46)
Exponent	$\lvert a(1-2^{\eta})\rvert$	Eq. (3.47)	$\lvert 1-2^{\eta}\rvert$	Eq. (3.48)
Sign	$\lvert 2a\rvert$	Eq. (3.49)	2	Eq. (3.50)

3.3.2 Exponent

Equation (3.13) presents the form of $\tilde{a}$. The absolute error is

$$\begin{aligned} |a-\tilde{a} &= |a - a \times 2^{\eta}| \\ &= |a(1-2^{\eta})|, \end{aligned} \tag{3.47}$$

and the relative error is

$$\begin{aligned} \frac{|a-\tilde{a}|}{|a|} &= \frac{|a(1-2^{\eta})|}{|a|} \\ &= |1-2^{\eta}|. \end{aligned} \tag{3.48}$$

3.3.3 Sign

Equation (3.15) presents the form of $\tilde{a}$. The absolute error is

$$\begin{aligned} |a-\tilde{a} &= |a-(-a)| \\ &= |2a|, \end{aligned} \tag{3.49}$$

and the relative error is

$$\begin{aligned} \frac{|a-\tilde{a}|}{|a|} &= \frac{|2a|}{|a|} \\ &= 2. \end{aligned} \tag{3.50}$$

We summarize the scalar error measures in Table 3.5.

3.4 Scalar Expected Error Measures

Having shown statistics for components of the IEEE-754 model, we explore the expected value of the absolute and relative error given a bit flip in an IEEE-754 floating point scalar.

3.4.1 Expected Relative Error for a Scalar

Mantissa

We now compute the expected relative error for a scalar with a mantissa error, $\mathbb{E}\left[\frac{1.\xi}{1.\beta}\right]$. The R.V. X_ξ was defined and analyzed in Eq. (3.22). Let X_ω be a discrete random variable taking values from

$$\frac{1}{1.\beta} \in \left\{ \frac{1}{1.0 + 2^{-N}(M-i)} \right\} \text{ for } i = 1, \ldots, M,$$

with equally likely probability. The expected value is

$$\begin{aligned} \mathbb{E}\left[\frac{1.\xi}{1.\beta}\right] &= \mathbb{E}[X_\xi X_\omega] \\ &= \mathrm{Cov}(X_\xi, X_\omega) + \mathbb{E}[X_\xi]\mathbb{E}[X_\omega], \end{aligned} \tag{3.51}$$

where $\mathrm{Cov}(u, v)$ is the covariance. The covariance is defined to be

$$\begin{aligned} \mathrm{Cov}(u, v) &= \mathbb{E}[(u - \mathbb{E}[u])(v - \mathbb{E}[v])] \\ &= \mathbb{E}[XY] - \mathbb{E}[X]\mathbb{E}[Y]. \end{aligned}$$

We begin with

$$\begin{aligned}
\mathrm{Cov}(X_\xi, X_\omega) &= \mathbb{E}[(X_\xi - \mathbb{E}[X_\xi])(X_\omega - \mathbb{E}[X_\omega])] \\
&= \frac{1}{NM}\sum_{i=1}^{N}\sum_{j=1}^{M}\left[\left(1 + 2^{-i} - \mathbb{E}[X_\xi]\right)(X_\omega - \mathbb{E}[X_\omega])\right] \\
&= \frac{1}{NM}\sum_{i=1}^{N}\sum_{j=1}^{M}\left[\left(1 + 2^{-i} - \left(1 + \frac{1}{N} - \frac{\epsilon}{N}\right)\right)(X_\omega - \mathbb{E}[X_\omega])\right] \\
&= \frac{1}{NM}\sum_{i=1}^{N}\sum_{j=1}^{M}\left[\left(2^{-i} - \frac{1}{N} + \frac{\epsilon}{N}\right)(X_\omega - \mathbb{E}[X_\omega])\right] \\
&= \frac{1}{NM}\sum_{i=1}^{N}\left(2^{-i} + \frac{\epsilon - 1}{N}\right)\sum_{j=1}^{M}(X_\omega - \mathbb{E}[X_\omega]) \\
&= \frac{1}{NM}\left[1 - 2^{-N} + \frac{\epsilon - 1}{N} \times N\right]\sum_{j=1}^{M}(X_\omega - \mathbb{E}[X_\omega]) \\
&= \frac{1}{NM}\left[1 - \epsilon + \epsilon - 1\right]\sum_{j=1}^{M}(X_\omega - \mathbb{E}[X_\omega]) \\
&= 0. \qquad (3.52)
\end{aligned}$$

Therefore, $\mathbb{E}\left[\frac{1.\xi}{1.\beta}\right] = \mathbb{E}[X_\xi]\mathbb{E}[X_\omega]$. The expected value of X_ξ is shown in Eq. (3.22). The expected value of the reciprocal of the mantissa is

$$\mathbb{E}[X_\omega] = \frac{1}{M}\sum_{i=1}^{M}\frac{1}{1 + 2^{-N}(M - i)}.$$

Recognize this as a left Riemann sum approximation of the integral

$$\begin{aligned}
\frac{1}{M}\sum_{i=1}^{M}\frac{1}{1 + 2^{-N}(M - i)} &= \int_0^1 \frac{1}{1 + x}\mathrm{d}x \\
&= \ln(2). \qquad (3.53)
\end{aligned}$$

The error of a left Riemann sum is

$$\begin{aligned}
E_L &= \frac{b - a}{2} \times \max(f'(c)) \times \Delta x \\
&= -\frac{1}{2}\frac{(1 - 0)^2}{M},
\end{aligned}$$

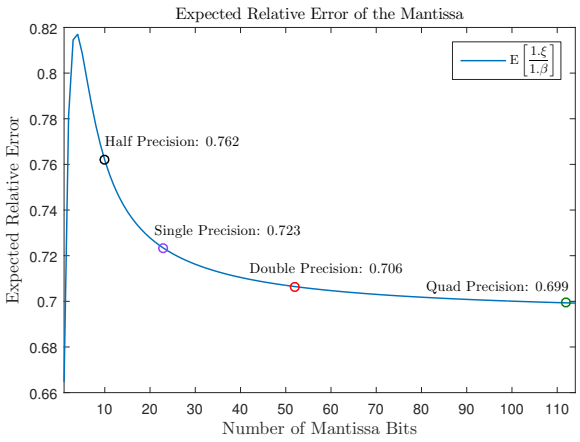


Figure 3.6: Expected relative error of the mantissa given a single bit flip.

where $\Delta x = \frac{b-a}{M}$. Recognize that for $f'(x) = -\frac{1}{(1+x)^2}$, the maximum value over the interval $[0, 1]$ is 1 resulting in the approximation error

$$\begin{aligned} E_L &= -\frac{1}{2}\frac{1}{M} \\ &= -\frac{\epsilon}{2}. \end{aligned}$$

Note $M = 2^N$, hence $1/M = \epsilon$. The expected value of the reciprocal of the mantissa is Eq. (3.53) plus the approximation error

$$\mathbb{E}[X_\omega] = \ln(2) - \frac{\epsilon}{2}. \tag{3.54}$$

Substituting Eqs. (3.54), (3.52), and (3.22) into Eq. (3.51) yields the expected value of the relative error

$$\begin{aligned} \mathbb{E}\left[\frac{1.\xi}{1.\beta}\right] &= \mathbb{E}[X_\xi]\mathbb{E}[X_\omega] \\ &= \left[1 + \frac{1}{N} - \frac{\epsilon}{N}\right] \times \left[\ln(2) - \frac{\epsilon}{2}\right] \\ &= \ln(2)\left[1 + \frac{1}{N}\right] - \frac{2\ln(2) + N + 1}{2N}\epsilon + \frac{1}{2N}\epsilon^2. \end{aligned} \tag{3.55}$$

Given that $N > 0$, then Eq. (3.55) obtains a maximum at $N = 4$. Figure 3.6 plots the expected relative error as a function of the number of bits used in the mantissa.

A key point of Eq. (3.55) is that the expected relative error of a scalar that experiences a bit flip in the mantissa is less than one. In prior work, we loosely bound the mantissa errors by incrementing the exponent. That is, no bit flip in the mantissa could ever generate an error

larger than twice the scalar's exponent. Clearly, our prior upper bound is an extreme case, while the expected relative error is much smaller.

Statistical Independence

Theorem 3.4.1 *The error introduced by a bit flip in the mantissa ($x_\xi = 1.\xi$) is independent of the mantissa value ($x_\beta = 1.\beta$).*

Proof 3.4.1 *The joint probability mass function $p_{X_\xi, X_\beta}(x_\xi, x_\beta) = P(X_\xi = x_\xi \text{ and } X_\beta = x_\beta)$. The marginal probability mass function for the error introduced from a single bit flip in the mantissa is*

$$p_{X_\xi}(x_\xi) = \sum_{x_\beta} p(x_\xi, x_\beta),$$

and the marginal probability mass function for the mantissa is

$$p_{X_\beta}(x_\beta) = \sum_{x_\xi} p(x_\xi, x_\beta).$$

The probability that $X_\xi = x_\xi$ is the probability of experiencing a bit flip in the i-th bit, which is $\frac{1}{N}$. The probability of having a specific mantissa is $X_\beta = x_\beta$, which is $\frac{1}{M}$.

The joint probability of experiencing a specific mantissa error (x_ξ) given a specific mantissa (x_β) is $p(x_\xi, x_\beta) = \frac{1}{NM}$. Two random variables are independent if $p_{X_\xi, X_\beta}(x_\xi, x_\beta) = p_{X_\xi}(x_\xi) p_{X_\beta}(x_\beta)$.

$$p_{X_\xi}(x_\xi) = \sum_{x_\beta} p(x_\xi, x_\beta) = \sum_{i=1}^{M} \frac{1}{NM} = \frac{M}{NM} = \frac{1}{N}$$

$$p_{X_\beta}(x_\beta) = \sum_{x_\xi} p(x_\xi, x_\beta) = \sum_{i=1}^{N} \frac{1}{NM} = \frac{N}{NM} = \frac{1}{M}$$

$$p_{X_\xi, X_\beta}(x_\xi, x_\beta) = \frac{1}{NM} = p_{X_\xi}(x_\xi) p_{X_\beta}(x_\beta)$$

□

The proof for the reciprocal of the mantissa is similar and is omitted. Theorem 3.4.1 is powerful, because it means that the relative error introduced from a bit flip in the mantissa does not depend on the mantissa value. This arises because the error $1.\xi$ is $\pm$, e.g., see Eq. (3.8). Regardless of whether the bit flipped is and 1 to 0 or 0 to 1, the absolute error is the same, i.e., $|\pm 1.\xi| = 1.\xi$. This implies that Figure 3.6 predicts the expected relative error for a mantissa bit flip for all values represented using the IEEE-754 specification.

Exponent

We analyze the expectation of the relative error of the exponent $|1 - 2^{\eta}|$ in two ways. First, recognize that the expected value of the relative error is $\mathbb{E}[2^{\eta} - 1] = \mathbb{E}[X_{\eta}] - 1$. This is not very informative, given that the expected value of η is dominated by the positive exponents. That is,

$$\frac{2^{bias}}{Z} < \mathbb{E}[2^{\eta} - 1] < \frac{2^{bias+1}}{Z} \tag{3.56}$$

Instead, consider the error term broken into positive and negative exponent sets, as shown in Eq. (3.39). The expected relative error given a negative exponent is

$$\mathbb{E}\big[|1 - X_{\eta^-}|\big] = |1 - \mathbb{E}\big[X_{\eta^-}\big]|,$$

which is bounded by

$$|1 - \mathbb{E}\big[X_{\eta^-}\big]| < 1. \tag{3.57}$$

The expected relative error for an error creating a positive exponent is $\mathbb{E}\big[2^{+\eta} - 1\big]$, which lies within the same bound as $\mathbb{E}\Big[X_{\eta^+}\Big]$.

$$\frac{2^{bias+1}}{Z} < \mathbb{E}\Big[2^{\eta^+} - 1\Big] < \frac{2^{bias+2} - 1}{Z}. \tag{3.58}$$

3.4.2 Expected Absolute Error for a Scalar

Mantissa

Let X_{ξ} be a discrete random variable taking values from Eq. (3.9). We break α into two ranges, as we did in § 3.2.4. X_{α} is a discrete random variable taking values from Eq. (3.6), and X_{α^+} and X_{α^-} are discrete random variables taking values from the positive and negative exponent sets. The expected value of the absolute error given a mantissa bit flip is

$$\begin{aligned}\mathbb{E}[\alpha \times 1.\xi] &= \mathbb{E}[X_{\alpha} X_{\xi}] \\ &= \mathrm{Cov}(X_{\xi}, X_{\alpha}) + \mathbb{E}[X_{\xi}]\mathbb{E}[X_{\alpha}].\end{aligned} \tag{3.59}$$

The expected value of X_{ξ} is shown in Eq. (3.22), and the expected value of X_{α} is shown in Eq. (3.27). The covariance will be zero,

$$\begin{aligned}\mathrm{Cov}(X_{\xi}, X_{\alpha}) &= \mathbb{E}[(X_{\xi} - \mathbb{E}[X_{\xi}])(X_{\alpha} - \mathbb{E}[X_{\alpha}])] \\ &= 0,\end{aligned} \tag{3.60}$$

because $\mathbb{E}[(X_\xi - \mathbb{E}[X_\xi])] = 0$, as shown in Eq. (3.52). Substituting Eq. (3.60) into Eq. (3.59),

$$\begin{aligned}\mathbb{E}[\alpha \times 1.\xi] &= \mathbb{E}[X_\xi]\mathbb{E}[X_\alpha] \\ &= \left[1 + N^{-1} - \frac{\epsilon}{N}\right] \times \left[\frac{1}{bias}\left[2^{bias} - 2^{-bias}\right]\right] \\ &\approx \frac{2^{bias}}{bias} + \frac{2^{bias}}{N \times bias} - \frac{2^{-N} \times 2^{bias}}{N \times bias}. \end{aligned} \tag{3.61}$$

Clearly, the expected absolute error for a mantissa bit flip will be dominated by the exponent of the scalar.

We now analyze the expected value over the positive and negative exponent sets. The covariance is zero, therefore, $\mathbb{E}[X_{\alpha^+}X_\xi] = \mathbb{E}[X_{\alpha^+}]\mathbb{E}[X_\xi]$ and $\mathbb{E}[X_{\alpha^-}X_\xi] = \mathbb{E}[X_{\alpha^-}]\mathbb{E}[X_\xi]$. The expected value over the positive set of exponents is

$$\mathbb{E}[X_{\alpha^+}]\mathbb{E}[X_\xi] = \frac{2^{bias+1} - 1}{bias + 1} \times \left[1 + N^{-1} - \frac{\epsilon}{N}\right]. \tag{3.62}$$

The negative set of exponents yields an expected value

$$\mathbb{E}[X_{\alpha^-}]\mathbb{E}[X_\xi] = \frac{1 - 2^{1-bias}}{bias - 1} \times \left[1 + N^{-1} - \frac{\epsilon}{N}\right]. \tag{3.63}$$

Recognize that Eq. (3.63) is a decreasing function in both N and the $bias$. Recall that $bias \geq 3$, because $Z \geq 3$ as shown in Eq. (3.37), and $N > 0$. The expected absolute error given a bit flip in the mantissa and a negative exponent is bounded by

$$\mathbb{E}[X_{\alpha^-}]\mathbb{E}[X_\xi] \leq \frac{9}{16}, \tag{3.64}$$

which we show graphically in Figure 3.7. There are two important points: 1) The expected absolute error is less than one, and 2) the expected absolute error is dominated by the number of exponent bits. In Figure 3.7, eight exponent bits correspond to $bias = 127$, which is used in the single precision specification. Single precision has an expected absolute error of approximately 10^{-2}. Eleven exponent bits result in a bias value of 1023, which is used for IEEE-754 double precision. The expected absolute error for double precision is approximately 10^{-3}. Fifteen exponent bits are used for quad precision, which has a bias of 16383. The expected absolute error for a mantissa bit flip in quad precision is on the order of 10^{-5}.

Figure 3.7 also shows that the absolute error given a bit flip in the mantissa can be forced to be small. Combined with the relative error given a bit flip in the mantissa, as shown in Figure 3.6, the absolute and relative errors are fairly well behaved if the scalar is in the interval $(-1, 1)$, i.e., $\alpha \leq 2^{-1}$.

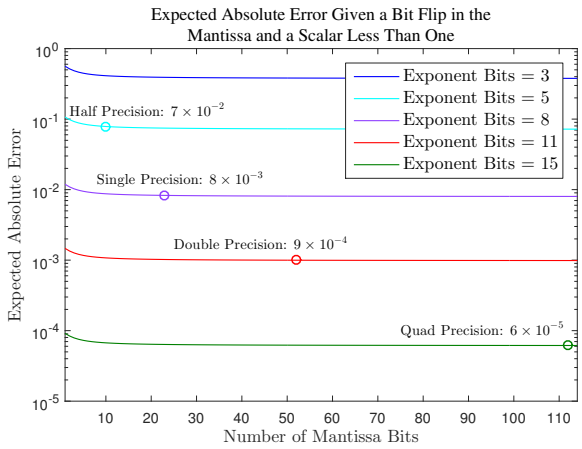


Figure 3.7: Expected absolute error given a bit flip in the mantissa and a scalar less than one. The absolute error is shown for the fewest number of exponent bits, as well as the number of exponent bits for half, single, double, and quad precision.

Exponent

The expected absolute error given a bit flip in the exponent is more tedious. We may algebraically isolate the error term, but the error term depends on the exponent bits. That is, $\alpha = 2^{e-bias}$, and $\tilde{\alpha} = 2^{\tilde{e}-bias} = 2^{e-bias\pm\eta}$. Precisely,

$$\tilde{\alpha} = 2^{-bias} \times 2^{(-1)^{bit_j}\times 2^j} \times \prod_{k=0}^{Z-1} 2^{bit_k \times 2^j}; \text{ for } j\text{-th bit flipped.}$$

The expected absolute error is

$$\frac{1}{ZK} \sum_{i=1}^{K} \sum_{j=0}^{Z-1} \left| 2^{i-bias} \left(1 - 2^{(-1)^{bit_j}\times 2^j} \right) \right|. \tag{3.65}$$

Knowing whether the bit flipped (bit_j) is zero or one is difficult, which makes a closed form difficult to express. Equation (3.65) is computable, and we can identify a critical component of the summation.

First, recognize that for a biased exponent $e \leq bias$, (i.e., $\alpha \leq 2^0$), the most significant exponent bit is always zero.

Lemma 3.4.1 ($e \leq bias \iff exponent_bit_{Z-1} = 0$) *A biased exponent e is less than or equal to the bias, if and only if the most signficant exponent bit is* 0.

Proof 3.4.2 ($e \leq bias \iff exponent_bit_{Z-1} = 0$.) *Suppose* $e \leq bias$ *and* $exponent_bit_{Z-1} = 1$. *The bias is defined to be* $bias = 2^{Z-1} - 1 = \sum_{i=0}^{Z-2} 2^i$. *e is defined to be the unsigned integer*

stored in the exponent bits, hence $e = \sum_{i=0}^{Z-1} exponent_bit_i \times 2^i$. *If* $exponent_bit_{Z-1} = 1$, *then* $e = 2^{Z-1} + \sum_{i=0}^{Z-2} exponent_bit_i \times 2^i > bias$, *which is a contradiction.*

Suppose $e > bias$ *and* $exponent_bit_{Z-1} = 0$. $e = \sum_{i=0}^{Z-2} exponent_bit_i \times 2^i + 0 \leq bias$, *which is a contradiction.* □

The property of biased exponents expressed in Lemma 3.4.1 implies that $bias - 1$ values in the summation will be $|a(1 - 2^{bias+1})|$, where $a \in (-1, 1)$. That is, for all exponents in the negative set, each exponent can always have the most significant bit flipped, creating an absolute error of $|a(1 - 2^{bias+1})|$, where $a \in (-1, 1)$. The scalar's unperturbed exponent will negate this increase in the exponent to some extent. For example, a scalar $x = 2^{-(bias-1)}$ has the unsigned integer 1 stored in binary in its Z exponent bits. With the most significant bit flipped, the unsigned integer $(bias + 1) + 1$ is now stored in x's exponent bits. The $(bias + 1)$ term comes from flipping the most significant bit, and $+1$ is the value already present in x's lower-order exponent bits. The scalar x, now $\tilde{x}$, represents the floating point value $\tilde{x} = 2^2$. As we consider the scalars $x = 2^{-(bias-2)}, 2^{-(bias-3)}, \ldots 2^{-1}$, the faulty scalars will be $\tilde{x} = 2^3, 2^4, \ldots, 2^{bias}$. Given that the correct (unperturbed) scalars are all less than one, the absolute error between the correct and faulty scalars will always be larger than one given a bit flip in the most significant bit. Given that the number representable negative exponents $(bias - 1)$ is much smaller than 2^{bias}, the summation is dominated by the summing of $2^{bias} + 2^{bias-1} + \ldots + 2^2$. This portion of the summation is from the most significant exponent bit flipping. There are $(Z - 1)$ lower-order exponent bits, which can never cause the faulty scalar to be larger than 2. That is, the worst the lower-order exponent flips can do is to create the binary pattern 2^0, which when combined with the largest possible mantissa value would create a scalar $\tilde{x} = 2 - \epsilon$.

For the positive set of exponents, the most significant bit will divide the number by large power of two, but the remaining bit flips will all produce a perturbed scalar that is larger than one. Conversely, the remaining bits of the exponent given a value $a \in (-1, 1)$, will produce a perturbed scalar that is less than two with probability $\frac{2Z-1}{2Z}$, which we show in Theorem 3.4.2. Because $a \in (-1, 1)$ and $\tilde{a} \in (-2, 2)$ and the sign cannot change because we assume the bit flip occurs in the exponent, then $|a - \tilde{a}| < 2$. That is, for the negative set of exponents, the expected

absolute error is bounded by

$$\begin{aligned}
\mathbb{E}[error_{\text{abs}}]_{\alpha^-} &< \frac{2^{bias} - C + 2^{bias-1} - C + \cdots + 2^2 - C + D \times (Z-1)(bias-1)}{Z(bias-1)} \\
&< \frac{\sum_{i=1}^{bias-1} 2^{i+1} - (bias-1) \times C + D \times (Z-1)(bias-1)}{Z(bias-1)} \\
&< \frac{\sum_{i=1}^{bias-1} 2^{i+1} + E \times Z(bias-1)}{Z(bias-1)} \\
&< \frac{2^{bias+1} - 4}{Z(bias-1)} + E, \qquad (3.66)
\end{aligned}$$

where $C = \max\{1.0, 1+\epsilon, 1+2\epsilon, \ldots, 2-\epsilon\} = 2-\epsilon$. That is, C is the largest possible mantissa value. Because we consider the negative set of exponents, the largest (unscaled) mantissa also bounds the scalars $2^{-j} \times 1.\beta \leq C$, because the largest scalar obtainable given the negative exponent set is $|a_{\max}| = |2^{-1} \times (2-\epsilon)| < 1.0$. We introduce a constant, D, which represents the largest possible absolute difference if the most significant exponent bit is not flipped. That is, $D = \max|a - \tilde{a}|$, which will always be less than two if the bit flipped is not the most significant, i.e., $D < 2$. The term $D \times (Z-1)(bias-1)$ accounts for the absolute error of the remaining $Z-1$ exponent bits being flipped in the $bias-1$ scalars, which always produce a faulty scalar strictly less than two, i.e., $|a - \tilde{a}| < 2$. Recognize that $D < 2$ and $C < 2$. To impose a strict (gross) upper bound, we may let $E = 2$, and substitute E for C and D, creating an over estimate of the expected absolute error given a bit flip in the exponent of scalars strictly less than one.

Theorem 3.4.2 *A bit flip in the exponent bits of an IEEE-754 scalar (a) that has a biased exponent $e \leq bias$ will produce a perturbed scalar ($\tilde{a}$) that is less than two, with a probability of $\frac{2Z-1}{2Z}$.*

Proof 3.4.3 *If the k-th bit is flipped $0 \to 1$ and $k \in 0, 1, 2, \ldots, Z-2$, then*

$$\tilde{e} = \left\lfloor \sum_{\substack{i=0 \\ i \neq k}}^{Z-2} exponent_bit_i \times 2^i \right\rfloor + 2^k \leq \sum_{i=0}^{Z-2} 2^i.$$

If the k-th bit is flipped $1 \to 0$ and $k \in 0, 1, 2, \ldots, Z-2$, then

$$\tilde{e} = \left\lfloor \sum_{\substack{i=0 \\ i \neq k}}^{Z-2} exponent_bit_i \times 2^i \right\rfloor \leq \sum_{i=0}^{Z-2} 2^i.$$

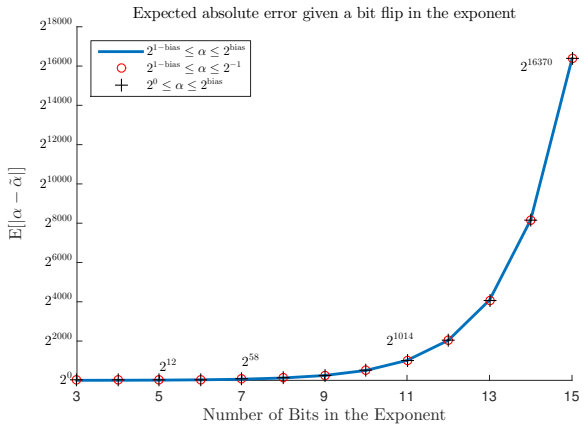


Figure 3.8: Expected absolute error given a bit flip in the exponent and a scalar less than one, greater than one, and over the full range. Half (2^{12}), Single (2^{56}), Double (2^{1014}), and Quad (2^{16170}) precision specifications are highlighted.

Table 3.6: Expected absolute error given a bit flip in exponent for scalar in the range $|a| \geq 1$ compared to the expected value of the positive set of exponents $\mathbb{E}[X_\alpha]$.

Spec.	$\mathbb{E}[X_\alpha]$	Approximate $\mathbb{E}[\lvert a - \tilde{a}\rvert]$; for $\lvert a\rvert \geq 1$.
Half Prec.	2^{12}	2^{12}
Single Prec.	2^{58}	2^{58}
Double Prec.	2^{1014}	2^{1014}
Quad Prec.	2^{16370}	2^{16370}

By Lemma 3.4.1, the most significant bit is zero. Hence, if the bit flipped is $k = Z - 1$, it cannot flip $1 \to 0$.

The probability of creating a faulty scalar with exponent $\tilde{e} \leq$ bias is $Z-1+Z-1+1 = 2Z-1$. The total number of potential bit flips is $2Z$. Hence, if $e \leq$ bias, then the resulting faulty scalar will have an exponent $\tilde{\alpha} \leq 2^0$ with probability $\frac{2Z-1}{2Z}$.

We numerically evaluate the expected absolute error given a bit flip in the exponent in Figure 3.8. Recognize the positive and negative set give approximately the same expected value, because the most significant bit of values $a \in (-1, 1)$ forces the summation to include a large power of two. We also compare the the expected absolute error to that of the expected value of the exponent in Table 3.6. If all exponent bits are allowed to be faulty, then the expected absolute error behaves like the expected value of an exponential function.

Table 3.7: Expected value of the relative error given a bit flip in a scalar ($\tilde{a}$).

Error Loc.	$error_{\text{rel}}$	Constraint	$\mathbb{E}[error_{\text{rel}}]$	Ref.
Mantissa	$\frac{1.\xi}{1.\beta}$	—	$\ln(2)\left[1+\frac{1}{N}\right]$	Eq. (3.55)
Exponent	$\lvert 2^{\pm\eta}-1\rvert$	—	$\frac{2^{bias}}{Z} < \mathbb{E} < \frac{2^{bias+1}}{Z}$	Eq. (3.56)
		positive η	$\frac{2^{bias+1}}{Z} < \mathbb{E} < \frac{2^{bias+2}-1}{Z}$	Eq. (3.58)
		negative η	< 1	Eq. (3.57)
Sign	2	—	2	Eq. (3.50)

Sign

A bit flip in the sign can be treated two ways, we may compute the expected value over the positive and negative sets, or we may treat a as constant, e.g., $\mathbb{E}[2a] = 2a$. Recognize that the negative set of exponents restricts the scalar such that $a \in (-1, 1)$ if α^-. If $a \in (-2, 2)$, then the sign bit will create an absolute error less than one, while if $|a| \geq 2$, then the absolute error will be larger than one.

3.4.3 Summary of Scalar Error Statistics

We summarize the statistics for each error measure derived in § 3.4.1 and § 3.4.2 in Tables 3.7 and 3.8. These are models for the correct and faulty components of the IEEE-754 representation when perturbed by a single bit flip. Consider the wide range of possible errors, e.g., the relative error from an exponent bit flip falls into two broad categories: very large and less than one. Similarly, the absolute error for a mantissa bit flip is dominated by the (non-faulty) exponent of the scalar. We consider an expected measure less than one to be "well behaved". This is a somewhat arbitrary definition, but the motivation stems from numerical analysis. If we can enforce that errors behave predictably, then we can devise schemes to dampen or maintain those errors, or possibly detect a subset of such errors. The choice of one also comes from observations in § 3.2. We have shown that the *expected* error tends to fall into two broad categories: larger than one, and less than one. We now extend these models to the multiplication operation, and the drawn higher-level conclusions about how the expected error models behave.

Table 3.8: Expected value of the absolute error given a bit flip in a scalar ($\tilde{a}$).

Error Loc.	$error_{\text{abs}}$	Constraint	$\mathbb{E}[error_{\text{abs}}]$	Ref.
Mantissa	$\lvert\alpha \times 1.\xi\rvert$	—	$\approx 2^{bias}$	Eq. (3.61)
Pos. α		$1 \leq \lvert a\rvert$	$\approx 2^{bias+1}$	Eq. (3.62)
Neg. α		$\lvert a\rvert < 1$	$\leq \frac{9}{16}$	Eq. (3.63)
Exponent	$\lvert a(1-2^{\pm\eta})\rvert$	—	$\approx 2^{bias+1}$	Eq. (3.35)
Pos. α		$1 \leq \lvert a\rvert$	$\approx 2^{bias+1}$	Eq. (3.35)
Neg. α		$\lvert a\rvert < 1$	$\approx 2^{bias+1}$	Eq. (3.66)
Sign	$\lvert 2a\rvert$		$\lvert 2a\rvert$	—
Pos. α and 2^{-1}		$1/2 \leq \lvert a\rvert$	> 1	—
Neg. α except 2^{-1}		$\lvert a\rvert < 1/2$	< 1	—

3.5 Model Multiplication Error Measures

Extending our analysis to multiplication is straight-forward, because only one operand can be faulty. We treat the non-faulty operand as a constant, and hence it operates as a scaling factor applied to our previous absolute error models and has no impact on the relative error. Consider two scalars, a and b. We now model the absolute and relative error measures for the operation $a \times b$.

3.5.1 Mantissa

For an error impacting the mantissa of one operand, the absolute error for multiplication is

$$\begin{aligned} error_{\text{abs}} &= |ab - \tilde{a}b| \\ &= |ab - b(a \pm \alpha \times 1.\xi)| \\ &= |ab - ab \pm b\alpha \times 1.\xi| \\ &= |b\alpha \times 1.\xi|, \end{aligned} \tag{3.67}$$

where $\tilde{a}$ has the form of Eq. (3.8). The relative error is

$$\begin{aligned} error_{\text{rel}} &= \frac{|ab - \tilde{a}b|}{|ab|} \\ &= \frac{|b\alpha \times 1.\xi|}{|ab|} \\ &= \frac{|\alpha \times 1.\xi|}{|a|} \\ &= \frac{|\alpha \times 1.\xi|}{|\alpha \times 1.\beta|} \\ &= \frac{1.\xi}{1.\beta}. \end{aligned} \tag{3.68}$$

3.5.2 Exponent

For an error impacting the exponent, $\tilde{a}$ takes the form of Eq. (3.13), resulting in an absolute error of

$$\begin{aligned} error_{\text{abs}} &= |ab - \tilde{a}b| \\ &= |ab - ab \times 2^{\pm\eta}| \\ &= |ab(1 - 2^{\pm\eta})|. \end{aligned} \tag{3.69}$$

The relative error is

$$\begin{aligned} error_{\text{rel}} &= \frac{|ab - \tilde{a}b|}{|ab|} \\ &= \frac{|ab(1 - 2^{\pm\eta})|}{|ab|} \\ &= |1 - 2^{\pm\eta}|. \end{aligned} \tag{3.70}$$

3.5.3 Sign

For an error in the sign bit, $\tilde{a} = -a$. The absolute error is

$$\begin{aligned} error_{\text{abs}} &= |ab - \tilde{a}b| \\ &= |ab - b(-a)| \\ &= |ab + ab| \\ &= |2ab|, \end{aligned} \tag{3.71}$$

Table 3.9: Error measures for faulty multiplication.

Location	Absolute Error	Ref	Relative Error	Ref
Mantissa	$\lvert b\alpha \times 1.\xi \rvert$	Eq. (3.67)	$\frac{1.\xi}{1.\beta}$	Eq. (3.68)
Exponent	$\lvert ab(1-2^{\pm\eta}) \rvert$	Eq. (3.69)	$\lvert 1-2^{\pm\eta} \rvert$	Eq. (3.70)
Sign	$\lvert 2ab \rvert$	Eq. (3.71)	2	Eq. (3.72)

and the relative error is

$$\begin{aligned} error_{\text{rel}} &= \frac{|ab - \tilde{a}b|}{|ab|} \\ &= \frac{|2ab|}{|ab|} \\ &= 2. \end{aligned} \tag{3.72}$$

We summarize the error measures for multiplication in Tables 3.9.

3.5.4 Multiplication Expected Error

The expected relative error is the same as our scalar models, e.g., See Table 3.7. The absolute error is simply the scalar model's expected absolute error scaled by the non-faulty operand, which we consider to be b.

Table 3.10 summarizes the expected absolute error. Note that, we have not stated all possible conditions for $|b|$. It is possible that if $b = 1/error_{\text{abs}}$, then you could clearly have an absolute error less than one. We have stated the conditions for $|a|$ and $|b|$ such that we know when the expected value will behave as we have modeled.

3.6 Applications to Fault Tolerance

We have analyzed IEEE-754 scalars that experience a single bit flip, and have shown models for the errors introduced. A key point from the analysis is that the expected error, either absolute or relative, can fall into two very broad categories: greater than one and less than one. A common technique in our analysis has been to partition the exponents into positive and negative exponent sets (α^+, and α^- respectively). We now consider how these characteristics relate to injecting bit flips into scalars (or into scalar multiplication).

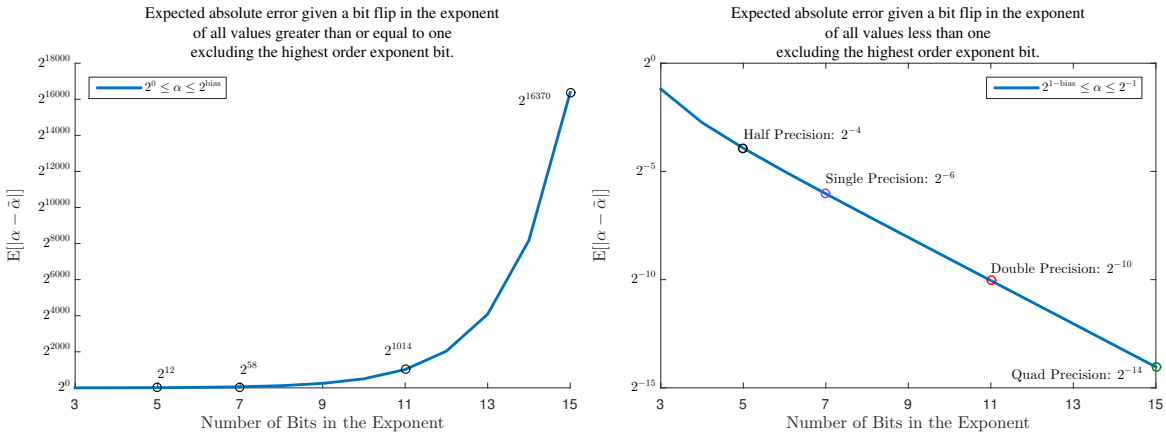
Table 3.10: Expected value of the absolute error given a bit flip in scalar multiplication ($\tilde{a} \times b$).

Error Loc.	$error_{\text{abs}}$	Constraint	$\mathbb{E}[error_{\text{abs}}]$	Ref.
Mantissa	$\lvert b\alpha \times 1.\xi \rvert$	—	$\approx 2^{bias}$	Eq. (3.61)
Pos. α		$1 \leq \lvert b \rvert$ and $1 \leq \lvert a \rvert$	$\approx 2^{bias+1}$	Eq. (3.62)
Neg. α		$\lvert b \rvert < 1$ and $\lvert a \rvert < 1$	$\leq \frac{9}{16}$	Eq. (3.63)
Exponent	$\lvert ab(1-2^{\pm\eta}) \rvert$	—	$\approx 2^{bias+1}$	Eq. (3.35)
Pos. α		$1 \leq \lvert b \rvert$ and $1 \leq \lvert a \rvert$	$\approx 2^{bias+1}$	Eq. (3.35)
Neg. α		$\lvert b \rvert < 1$ and $\lvert a \rvert < 1$	$\approx 2^{bias+1}$	Eq. (3.66)
Sign	$\lvert 2ab \rvert$		$\lvert 2ab \rvert$	—
Pos. α and 2^{-1}		$1 \leq \lvert b \rvert$ and $1/2 \leq \lvert a \rvert$	> 1	—
Neg. α except 2^{-1}		$\lvert b \rvert < 1$ and $\lvert a \rvert < 1/2$	< 1	—

3.6.1 Constrained Exponent Bit Flips

In § 3.4.2, we showed that the most significant bit being flipped dominates the expected absolute error, e.g., see Theorem 3.4.2. We now consider the case that we are able to exclude the most significant bit from the expected value. We compute the expected absolute error excluding the most significant exponent bit for both the negative and positive set of exponents. Figure 3.9a plots the expected absolute error given a bit flip in the positive set of exponents, while excluding the most significant bit. We see the expected absolute error behaves like the expected value of the exponent, e.g., Eqs. (3.35) or (3.27).

Figure 3.9b plots the expected absolute error over the negative set of exponents, while excluding the most significant bit. The point is that if the most significant bit is excluded the expected absolute error is less than one. That is, we can remove the large error term from Eq. (3.66), yielding an expected absolute error that is strictly less than one when given a scalar



(a) Values greater than or equal to one.

(b) Values less than one.

Figure 3.9: Expected absolute error given a bit flip in the exponent, excluding the most significant bit for values less than one, and values greater than one.

less than one, e.g., Eq. (3.73).

$$\mathbb{E}[error_{\text{abs}}]_{\alpha^-} < \overbrace{\frac{2^{bias+1} - 4}{Z(bias - 1)}}^{\text{Excluded}} + E$$
$$\mathbb{E}[error_{\text{abs}}]_{\alpha^-} < 2. \tag{3.73}$$

This property is particularly useful, as it shows that a bit flip in the exponent does not necessarily result in a large error, or even the expectation of a large error. The absolute error given a bit flip can be somewhat well-behaved if α can be constrained to the negative set of exponents.

Equation (3.73) and its original form, Eq. (3.66), present an upper bound on the expected absolute error. This is because E is an upper bound for the absolute error $|a - \tilde{a}| < 2$. If the scalars are constrained such that $a \in (-1, 1)$, the perturbed scalar is only order 2 if the bit flip creates the binary form of the bias. That is, the bit flip must create an exponent that corresponds to 2^0. The mantissa forces the bound to be less than 2, because the mantissa is bounded in the interval $[1, 2)$.

3.7 Overall Effect of Constrained Exponent Bit Flips

We have considered bit flips in specific locations of the representation. What matters is how these various effects interplay. A main observation is that bit flips in values less than one will produce absolute and relative errors less than one most of the time. The expected relative errors

Table 3.11: Probability that the relative error will be less than one given a bit flip in a scalar.

Specification	Mantissa	Exponent	Sign	$\Pr(error_{rel} < 1)$	$\Pr(error_{rel} \geq 1)$
Half Prec.	$\frac{10}{16}$	$\frac{1}{2} \times \frac{5}{16}$	$\frac{0}{16}$	0.78125	0.21875
Single Prec.	$\frac{23}{32}$	$\frac{1}{2} \times \frac{8}{32}$	$\frac{0}{32}$	0.84375	0.15625
Double Prec.	$\frac{52}{64}$	$\frac{1}{2} \times \frac{11}{64}$	$\frac{0}{64}$	0.89844	0.10156
Quad Prec.	$\frac{112}{128}$	$\frac{1}{2} \times \frac{15}{128}$	$\frac{0}{128}$	0.93359	0.06641

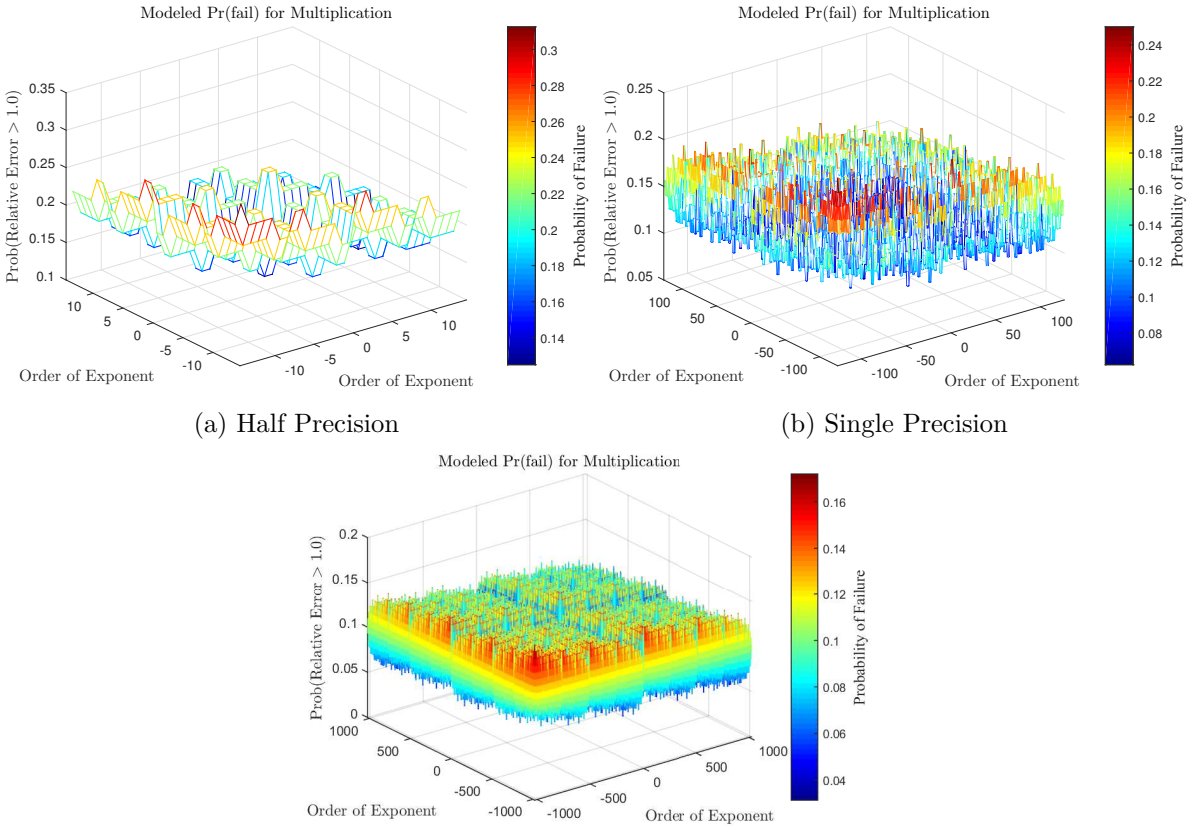
for multiplication, shown in Table 3.7, show that the mantissa is expected to have a relative error of approximately $\ln(2)$. An exponent bit flip can be both less than one and very large.

We consider an IEEE-754 scalar, which has $N_{bits} = Z + N + 1$, where Z is the number of exponent bits, N is the number of mantissa bits, and one sign bit. The probability that a bit will impact a specific region of the representation is then N/N_{bits} for a mantissa bit flip, Z/N_{bits} for an exponent bit flip, and $1/N_{bits}$ for a sign bit flip. Table 3.11 computes the probability that the relative error will be less than one, based on the expected value. The probability increases as the format increases from Half to Quad precision. This is because the mantissa bits dominate the expected value. Assuming bits are equally likely to be one or zero, then only half of the exponent bits will satisfy our condition of relative error less than one. A sign bit flip will always fail.

The values in Table 3.11 also model the expectation of the relative error for multiplication. We explore the behavior of the expected relative error by analyzing two scalars multiplied in half, single, and double precision, each with a fixed exponent, in Figure 3.10. Note that the probability of failure is the probability that the expected error is larger than one. Figure 3.10a evaluates the possible relative errors for half precision. Observe that the center value of the colorbar is approximately 22%. Our model for half precision predicts 21.875% (see Table 3.11). If we compute the expected value across all scalars less than one, i.e., compute the expected value over the entire surface plot in the quadrant $[0, -14]$, we observe a probability of 0.21875, which agrees exactly with our model.

Figures 3.10b and 3.10c repeat our study using single and double precision. We observe that the relative error is larger than one approximately 16% of the time for single precision, and approximately 10% of the time for double precision. Computing the expected value over the scalars that are less than one, we observe the relative error larger than one 15.625% of the time for single precision, and 10.156% of the time for double precision. These values agree exactly with what our model predicts, as shown in Table 3.11.

This result is significant, because it shows that moving to quad precision is not necessarily bad. Naively, it would seem that the more bits used for a representation, the worse the errors should behave. The latter is not true, because the number of mantissa bits has increased significantly more than the number of exponent bits. Quad precision adds only 4 additional exponent bits compared to double precision, but adds 60 additional mantissa bits. The relative error of a bit flip in the mantissa is still quite large, ln(2), but this provides a clear direction for resilient algorithm design.



(a) Half Precision

(b) Single Precision

(c) Double Precision

Figure 3.10: Probability that the expected relative error is larger than one, given a single bit flip in scalar multiplication for: (a) half, (b) single, and (c) double precision.

3.7.1 Expected Absolute Error Given a Scalar

We now compute the probability that the absolute error will be less than one given a scalar less than one. If the scalars are strictly less than one, then the biased exponent will be in the range $[1, 1022]$. We must exclude the bit pattern of 1023, because scalars with exponents 2^0 are strictly less than two. Table 3.10 summarized the expected absolute error for a bit flip in scalar multiplication. We showed in Eq. (3.73) that the expected absolute error for scalars less than one, is dominated by the most significant exponent bit. For scalars less than one, $a \in (-1, 1)$, the mantissa and sign can never introduce an absolute error larger than one. Of all exponent bit flips, the majority will not create an absolute error larger than one. Specifically, there are $bias - 2$ exponents representable for scalars less than one. Because we constrain the scalars to be less than one, but must count the excluded biased exponent 1023, the total number of possible bit flips is

$$Total_{bitflips} = (bias - 1) \times (N + Z + 1).$$

The probability of observing an absolute error less than one is then

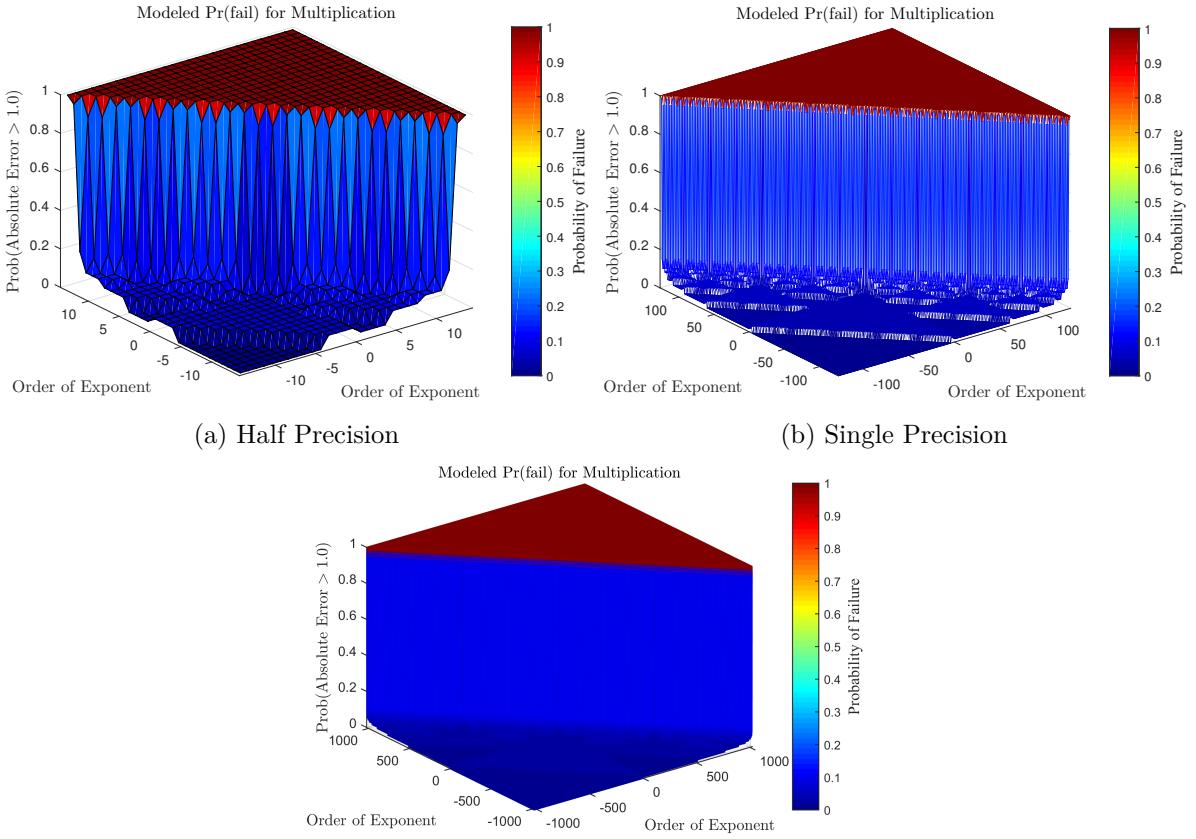
$$\frac{\overbrace{(bias-2)\times N}^{Mantissa} + \overbrace{(bias-2)\times 1}^{Sign} + \overbrace{(bias-2)\times(Z-1)}^{Exponent} - \overbrace{(Z-1)}^{Excluded}}{(bias-1)\times(N+Z+1)}. \tag{3.74}$$

Note that the "Excluded" term removes the $Z - 1$ possible bit flips that would create the excluded exponent 2^0 (i.e., the bias). Because we consider a single bit flip, there are exactly $Z - 1$ bit flips that can create the binary pattern corresponding to 2^0. These excluded bit flips all take the form of flipping a zero to a one, to create the binary pattern $0111 \ldots 1$. That is, we must not count the exponent bit flips that would create the the bias value in the exponent storage. If we allowed these, the absolute error would be bounded by 2.

3.7.2 Expected Absolute Error Given Scalar Multiplication

We compute the probability that the absolute error will be less than one given a bit flip in scalar multiplication. Unlike the relative error, both scalars impact the absolute error. Figure 3.11 visualizes the probability that the absolute error will be larger than one for the half, single and double precision formats. We construct each figure by fixing the exponent of each scalar and using the expected value of the mantissa. In Chapter 2, we constructed similar figures, but used Monte Carlo trials to approximate the expected value of the mantissa. That is, we fixed the exponent and generated random mantissas. Figure 2.6a showed the result of our experiment using dot products rather than scalars. Notice the structure of our plots are very similar. The region showing the probability when both scalars are less than one is flat and near zero. The difference is the gradient that transitions from low probability to high. This is expected, because

outside of the region where both scalars are less than one, the absolute error can be both small and large. Specifically, mantissa bit flips can "fail", e.g., see Eq. (3.67).



(a) Half Precision

(b) Single Precision

(c) Double Precision

Figure 3.11: Probability that the expected absolute error is larger than one, given a single bit flip in scalar multiplication for: (a) half, (b) single, and (c) double precision.

To emphasize the impact of the mantissa, Figure 3.12 repeats our experiment and fixes the mantissa errors such that we observe the largest possible mantissa error for all mantissa bit flips. That is, we always flip the most significant mantissa bit, rather than show the average. Clearly, this is not realistic. We have shown the analytic form of the expected mantissa error in Eq. (3.22), which is much smaller than the largest mantissa error (1.5). The effect of modeling all worst-case mantissa errors is that the mantissa fails more often. This effect is most easily visualized by the reduction in the "drip" effect seen in Figure 3.11a, but the impact is essential in understanding how the low probability region behaves.

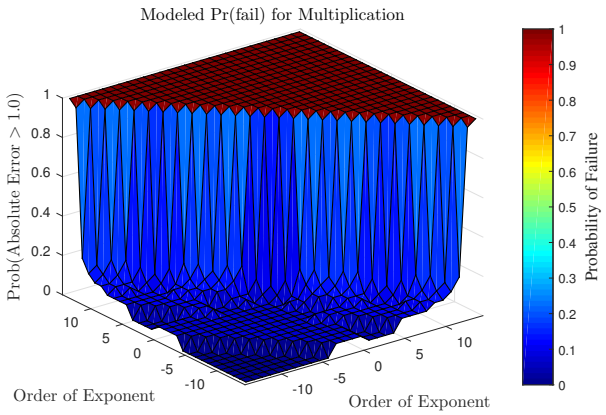


Figure 3.12: Absolute error experiment for half precision using the worst-case mantissa error for all mantissa errors.

We explore the region of scalars strictly less than one in Figure 3.13 for the half precision format. Figures 3.13a and 3.13b perform the experiment using the expected value of the mantissa error, and Figures 3.13c and 3.13d use the worst-case mantissa error. The right-most plots avoid the interpolation that the surface plot introduces and provide a clearer picture of which combination of scalars are producing absolute errors larger than one. The difference between the two experiments is the increased failure of scalars with exponents of 2^{-2} and 2^{-1}. This maps to the two extra orange squares in Figure 3.13d. We show a zoomed-in view for the single and double precision formats in Figures 3.14a and 3.14b, respectively. Recognize the spikes in each. The spikes indicate a higher probability of observing an absolute error larger than one. Due to lack of fidelity with single and double precision, we use half precision in latter examples.

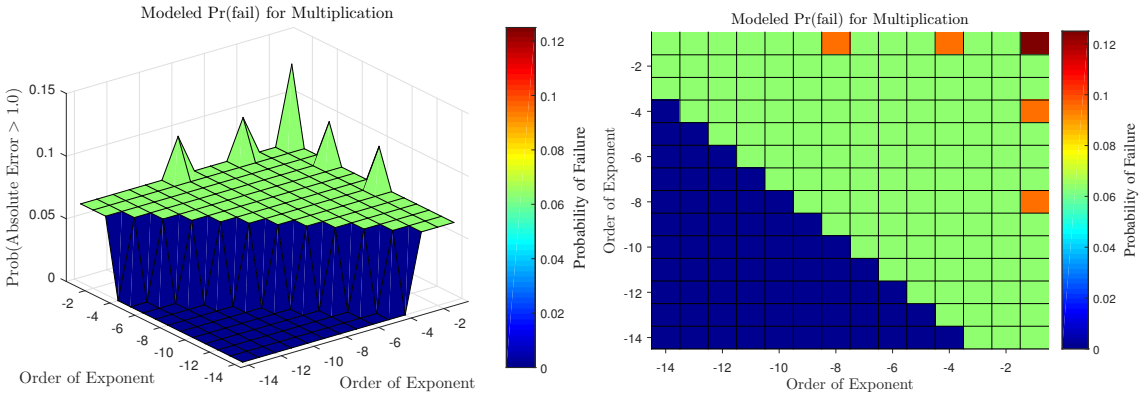
The structure in Figure 3.13 is the result of two side effects introduced from multiplication:

- First, the green spikes map to the excluded bits from Eq. (3.74). Given scalars in the range of $[0, 1)$ there are $Z - 1$ bit flips that create the exponent 2^0, i.e., the bias value. We must exclude the exponent 2^0, because it enables the mantissa bits to fail.
- Second, the non-faulty scalar functions as a "scaling factor". There are cases where the operands are sufficiently small that they annihilate the large error produced by the most significant bit flipping.

We now address the effect of each case.

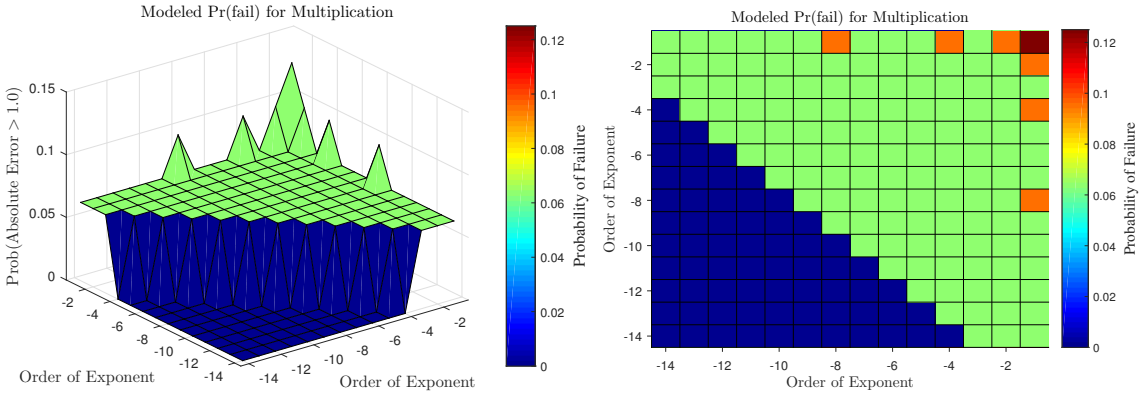
Excluded Bits

To ensure the absolute error remains less than one, we require that the scalars operated on be strictly less than one, e.g., see Eq. (3.74). If we require that both scalars have exponent values



(a) Half precision surface using the expected mantissa error.

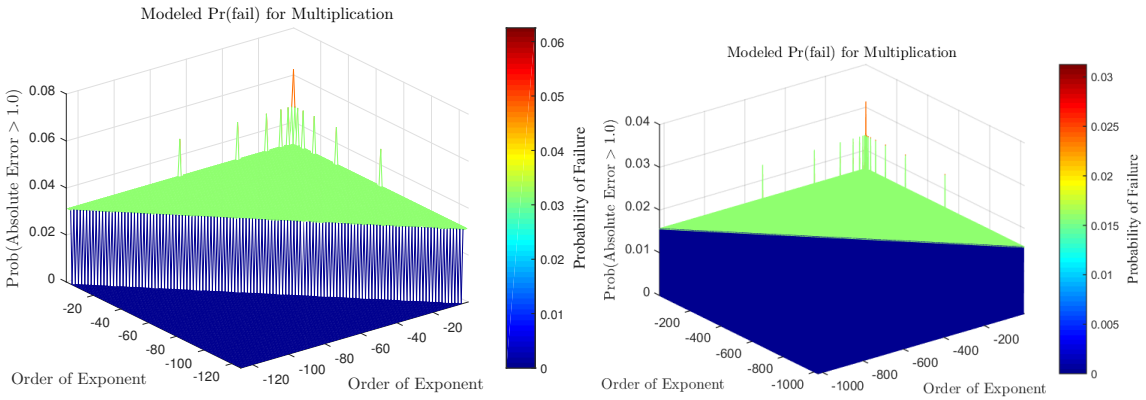
(b) Half precision overhead view using the expected mantissa error.



(c) Half precision surface using the worst-case mantissa error.

(d) Half precision overhead view using the worst-case mantissa error.

Figure 3.13: Probability that the expected absolute error is larger than one, given a single bit flip in scalar multiplication for half precision and scalars in the range $[0, 1)$.



(a) Single precision surface using the worst-case mantissa error.

(b) Double precision surface using the worst-case mantissa error.

Figure 3.14: Probability that the expected absolute error is larger than one, given scalars in the range $[0, 1)$ and a single bit flip in scalar multiplication for (a) single and (b) double precision.

$\alpha \leq 2^{-1}$, then the $Z-1$ bit flips that will create the exponent 2^0 are only impactful if the non-faulty operand has exponent 2^{-1}.

Lemma 3.7.1 *Given two scalars with exponents in the interval* $[2^{-(bias-1)}, 2^{-1})$*, it is impossible for a bit flip in an exponent bit, bit* $\in \{exponent_bit_{Z-2}, exponent_bit_{Z-3}, \ldots, exponent_bit_0\}$*, to create a absolute error larger than one if the non-faulty operand has exponent* $\alpha \leq 2^{-2}$*.*

Proof 3.7.1 *Let two scalars be* $(a, b) \in (-1, 1)$*. Suppose a bit flip perturbs a such that it creates the bias pattern in the exponent, i.e.,* $\tilde{\alpha}_a = 2^0$*. Suppose the exponent for the non-faulty scalar b is less than* 2^{-1}*. The resulting perturbed exponent will be* $\tilde{a} \times b = 2^0 \times 1.\beta_a \times 2^{-2} \times 1.\beta_b = 2^{-2} \times 1.\beta_a \times 1.\beta_b$*.*

Any mantissa, $1.\beta$*, is bounded above by* $1.\beta < 2$*. Hence, the faulty multiplication is bounded by* $|\tilde{a} \times b| < 2^{-2} \times 4$*, and the absolute error cannot be larger than one, because* $|a \times b| < 1$*.* □

The spikes in Figures 3.13c, 3.14a, and 3.14b occur when the non-faulty scalar has exponent 2^{-1}. This effect is symmetric because multiplication is commutative. Recognize the number of spikes in Figure 3.13c is $Z - 1 = 5 - 1 = 4$ along either boundary, because the boundary represents a scalar with exponent 2^{-1}.

Most Significant Bit Flip Mitigated

Next, we explain the two-level structure seen in Figure 3.13c. Given scalars in the range $2^{bias-1}, 2^{-1}$, there are $bias - 1$ such scalars, and therefore $bias - 1$ most significant exponent bits that can be flipped leading to a large absolute error. Given scalar multiplication, it is possible

that if both scalars are sufficiently small, their product will produce an exponent sufficient to cancel the large multiplicative error. Given scalars in the range $[0, 1)$, a bit flip in the most significant exponent bit of an operand in scalar multiplication will produce an absolute error less than one $(bias - 4) \times (bias - 3)$ number of times.

Lemma 3.7.2 *Given two scalars with exponents in the interval* $[2^{-(bias-1)}, 2^{-1})$ *the most significant bit flip in either operand of scalar multiplication will produce an absolute error less than one* $(bias - 4) \times (bias - 3)$ *number of times.*

Proof 3.7.2 *Let two scalars be* $(a, b) \in (-1, 1)$*. Suppose a bit flip perturbs a such that it creates the bias pattern in the exponent, i.e.,* $\tilde{\alpha} = 2^0$*. Suppose the most significant exponent bit is flipped in either operand of scalar multiplication. The multiplicative error introduced by the most significant exponent bit flip is* 2^{bias+1}*. To ensure the absolute error is less than one, the resulting complete exponent (*α*) from the faulty product must be* $\alpha \leq 2^{-2}$*, otherwise the mantissa values can cause failure.*

To have $\alpha \leq 2^{-2}$*, the sum of the exponents must satisfy* $exp_a + exp_b + bias + 1 \leq -2$*. That is, the exponents of the operands must be sufficiently small to cancel the large multiplicative error* 2^{bias+1} *and remain small enough to ensure that the mantissa bits cannot produce a large absolute error.*

The smallest exponent representable is $2^{-(bias-1)}$*. Suppose* $exp_b = -(bias - 1)$*. This yields a bound of* $exp_a - (bias - 1) + bias + 1 \leq -2$*,* $exp_a \leq -4$ *if b has an exponent of* $2^{-(bias-1)}$*. If b becomes larger, than a must become smaller. The number of products satisfying this constraint is the triangle sum of* $\frac{(bias-4)\times(bias-3)}{2}$*. Repeating this analysis considering a as the smallest value, and b as the largest yields* $2 \times \frac{(bias-4)\times(bias-3)}{2}$*, or* $(bias - 4) \times (bias - 3)$ *number of products will cancel the most significant bit flip while ensuring the mantissa cannot force the error to be larger than one.* □

The reasoning used in Proof 3.7.2 is clear when observing Figure 3.13. To compute the number of products that will negate the most significant bit flipping, we only need to sum the blue triangle in Figure 3.13d. For half precision, the bias is 15, and $bias - 4 = 12$. The blue triangular region represents the scalar products that satisfy $exp_a + exp_b + bias + 1 \leq -2$. That is, $-4 + (-14) + 15 + 1 = -2 \leq -2$.

3.7.3 Probability of an Absolute Error Larger Than One

The prior subsections have addressed two effects that multiplication has on the absolute error created should a bit flip in either operand. We now unify the concepts presented to compute the probability that the absolute error will be greater than or equal to one given scalars $(a, b) \in (-1, 1)$ and scalar multiplication.

First, the number of exponents possible given scalars in the interval $(-1, 1)$ is $(bias - 1)$. Multiplication has two operands each having $N + Z + 1$ bits. Therefore, the total number of bits that can be flipped are

$$Total_bits = 2 \times (bias - 1)^2 \times (N + Z + 1). \tag{3.75}$$

The bit flips that can create the bias were addressed in Lemma 3.7.1. These "excluded" bit flips can only be impactful if the one of the scalars has exponent 2^{-1}, and there are only $Z - 1$ such bit flips given all representable exponents in the range $[0, 1)$. The total ways to have a bit flip create 2^0 is

$$Bit_flips_to_bias = 2 \times (Z - 1). \tag{3.76}$$

The most significant exponent bit flip is not impactful for $(bias - 4) \times (bias - 3)$ of the total bit flips.

$$bad_significant_exp_bit_flips = 2 \times (bias - 1)^2 - (bias - 4) \times (bias - 3). \tag{3.77}$$

The probability that the absolute error will be larger than one is

$$\Pr(error_{abs} > 1) = \frac{Bit_flips_to_bias + bad_significant_exp_bit_flips}{Total_bits}. \tag{3.78}$$

We evaluate Eq. (3.78) for the formats half, single, double, and quad precision in Table 3.12. We also compared our modeled probability against what we observed in our experiments using the worst-case mantissa error, and obtained a perfect fit. That is, the difference between our modeled probability and what we observed is zero. Our model represents an upper bound because our proofs and derivations are designed to address the largest possible mantissa values, i.e., $1.\beta = 2 - \epsilon$. This difference is visuallized in Figure 3.13. Figure 3.13a has fewer spikes than Figure 3.13c. This is because the product of the expected mantissa error is $1 + 1/N$, which is less than 1.5 (the worst-case).

3.8 Application of Constrained Exponent Bit Flips

It is not immediately obvious how we could simply remove a bit from consideration in the expected value. Recognize that restricting values to be less than one is the same effect that normalizing a vector or equilibrating a matrix achieves. The concept of operating on values in the interval $(-1, 1)$ is very common in numerical mathematics. For example, the Arnoldi process [5], which forms the basis for many Krylov subspace linear solvers and eigensolvers constructs

Table 3.12: Probability that the absolute error will be larger than one given a bit flip in scalar multiplication with $(a, b) \in (-1, 1)$.

Specification	$\Pr(error_{abs} \geq 1)$
Half Prec.	0.04273
Single Prec.	0.01625
Double Prec.	0.00785
Quad Prec.	0.00391

an orthonormal set of basis vectors. Elliott et al. [30] instrumented the GMRES solver, and showed that the probabilities we have shown, hold inside the Arnoldi process.

This is a strong result. The errors from a bit flip seem extremely unpredictable, yet if data is scaled, then the errors are "well behaved" most of the time. Clearly, our definition of well behaved is arbitrary, but there is strong evidence that bounding errors is an effective fault tolerance mechanism for iterative linear solvers, e.g., see Elliott et al. [29].

Another implication of this work stems from § 3.7.2. Lemma 3.7.2 established that as values become smaller, the expected absolute error goes to zero. That is, given sufficiently small scalars, the product is sufficiently small to cancel the large error introduced. The relative error will always have a 50/50 chance of being large or small, but the absolute error will be less than one the majority of the time. This is important for algorithms that generate normalized values, as these values may becomes very small, but the absolute error will always be less than one.

3.9 Conclusion

We have used analytic modeling and statistical analysis to show how a bit flip in the representation of an IEEE-754 floating point behaves. We have computed the expected absolute and relative error, and shown that these measures fall into two broad categories: less than one, and very large. We have rigorously justified the observations in related work, and shown that the findings in [30] hold for formats besides binary64. We have shown that the number of bits is only marginally important; rather the scaling of the values is more important when it comes to the errors introduced from a bit flip.

Chapter 4

Evaluating the Impact of Silent Data Corruption on the GMRES Iterative Solver

4.1 Introduction

Incorrect arithmetic or corruption of stored data could have dire effects on the execution of a numerical algorithm. Experiments show that a single bit flip in memory can cause certain algorithms to "crash" (terminate abnormally, due to invalid states or actions detected by the application or operating system), "stagnate" (keep running but fail to make progress), or, worst of all, produce the wrong solution, silently.

Rather than focusing exclusively on bit flips, this work studies the impact of Silent Data Corruption (SDC) on the Generalized Minimal Residual Method (GMRES) iterative linear solver. The source of the corruption, while interesting, gives no insight into its impact on the algorithm and the correctness of its result. By generalizing bit flips in floating-point data into potentially unbounded numerical errors, we are able to use mathematical analysis both to reason about algorithms' behavior should an SDC event occur, and to harden them against the event's effects.

Fortunately, some numerical algorithms only need reliability for certain data and phases of computation. If the system can guard just those parts of the algorithm in space and time, then the algorithm can compute the right answer — or at least be able to detect failure and report it "loudly" — despite faults in unreliable phases of execution. This suggests a "layered" approach to the design of reliable numerical algorithms. A reliable outer layer can recover from faults in a less reliable inner layer. If the solver spends most of its time in unreliable mode, it can mitigate the cost of reliable computation in the outer mode. We begin the analysis with GMRES, and then extend it to the Fault-Tolerant GMRES (FT-GMRES) inner-outer iteration.

We present the following contributions:

- We use mathematical analysis of the GMRES algorithm to construct a detector that bounds the error that SDC may introduce.
- We combine the above detection scheme with the *sandbox* reliability model presented in [13].
- We illustrate experimentally that bounded error originating in the faulty inner solve has little impact on time-to-solution.

4.1.1 Silent Data Corruption

In this work, we address a very specific type of fault, i.e., a fault that silently introduces bad data, while not persistently tainting the data that was used in the calculation. For example, let $a = 2$ and $b = 2$, then $c = a + b = 10$, while simplistic, this model presumes no knowledge of the nature of the fault, only that c is incorrect. This model assumes that the machine is unreliable in an unpredictable way, and therefore we are skeptical of the output it presents. This type of unreliability can be mitigated via redundant computation and introspection, but then the cost of running the algorithm increases drastically.

4.1.2 Faults, Failures, and Persistence

Our goal is to ensure that should transient SDC occur, we either obtain the correct solution or make the fault not silent by alerting the user. We consider two perspectives: the user and the system. A fault occurs at the system level, e.g., a bit flips or a node crashes. *A fault becomes a failure if it impacts the user.* Figure 4.1 depicts a visual taxonomy of how we consider faults and the scope of our work. We further classify faults into those that interrupt the user's program (hard faults), and those that do not immediately or ever interrupt the user's code (soft faults). A hard fault results in a failure if the user is running an application (though a checkpoint / restart recovery system can "mask out" hard faults, making them not failures). In contrast, the very nature of soft faults implies that they may emit no indication that something has gone wrong. In the event that soft faults allow the program to continue execution with tainted data, we must understand how algorithms behave in the presence of faulty data. Furthermore, if the algorithm uses tainted data and still obtains the correct solution, then the fault does not constitute a failure. If the soft fault leads to an incorrect solution, then the fault leads to a ***silent* failure**, which is an outcome we wish to make very rare or impossible.

We further classify soft faults by how long the underlying hardware remains faulty. Persistent faults arise from hardware that is permanently faulty, e.g., a stuck bit in memory, or the Intel Pentium FDIV bug [60]. Sticky faults indicate hardware that is faulty for some duration but

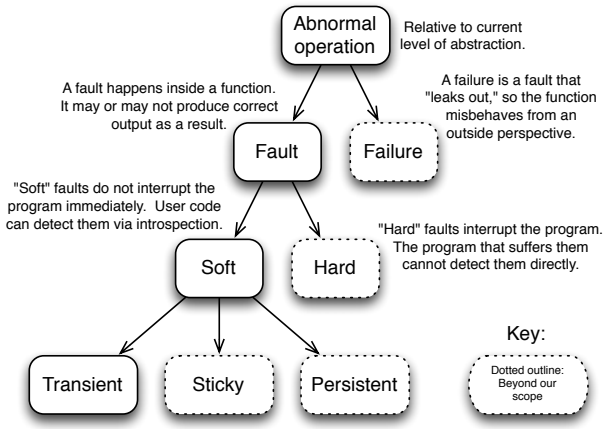


Figure 4.1: Taxonomy of faults and scope of this work.

returns to normal operation. Transient faults occur once, and while the fault is transient the effect of the fault may be persistent.

4.2 Project Overview

To quantify the possible effects of a single silent data corruption event in GMRES, we propose a multifaceted approach. We combine the sandbox reliability model from [47, 13], Flexible GMRES from Saad [76], and mathematical analysis of the GMRES algorithm to create a nested solver strategy that combines an unreliable inner solver with a reliable outer solver, while enforcing that, should SDC occur in the unreliable phase, the error is bounded. We then show through experiments how our scheme "runs through" single SDC events in the unreliable solver. Using this approach, we ultimately seek to present analyses of solvers such that we can quantitatively choose solvers based on their resiliency to single events of SDC.

This chapter is organized as follows:

1. In § 4.4, we describe the sandbox reliability model.
2. In § 4.5, we present standard GMRES, and uncover through mathematical analysis an invariant in the Arnoldi process contained within GMRES.
3. We describe how to check this invariant when using Modified Gram-Schmidt orthogonalization.
4. In § 4.6, we introduce Saad's Flexible GMRES and Heroux et al.'s FT-GMRES, and explain the relation between flexible solvers and the sandbox reliability model.
5. In § 4.7, we compose a nested solver using the Trilinos framework [49], and present experimental results illustrating how our invariant check impacts time-to-solution.

4.2.1 Assumptions and Justification

We restrict SDC to the numerical data used and generated by the algorithm. We explicitly exclude faults in control flow, data structures, loop counters and other metadata used to implement the algorithm. The reason for this exclusion is that these issues represent a different class of problems.

Our assumption that SDC occurs only once is fundamental. The implied source of SDC is typically a bit flip, but we do not restrict our model to silent bit upsets, given that there is limited data available today to base such a model on. We justify our choice of single transient SDC, based on what we do know about bit flips and the reliability of the system:

1. Hardware employs techniques to ensure that so-called "single event upsets" (SEUs) – that is, bit flips – do not occur. Therefore, it is expected that SEUs will be rare events.
2. If we can understand the best- and worst-case scenarios for the error that an SDC can contribute, we will have a baseline to conjecture about multiple bit flips, i.e., multiple occurrences of SDC.
3. There currently is no solid theory, e.g., a statistical distribution, of the rate at which bit flips occur. Therefore, speculation about flip rates may or may not prove useful.
4. Assuming a particular fault rate makes bold assertions about future hardware, especially given the reluctance of hardware manufacturers to divulge this information.

By following this research path, we are able to avoid the pitfalls presented in items 3 and 4 above, and we are able to isolate the impact of SDC without other factors polluting our analysis.

4.3 Motivation

Energy and peak power increasingly constrain modern computer hardware, yet hardware approaches to protect computations and data against errors cost energy. This holds at all scales of computation, but especially for the largest parallel computers being built and planned today. This results from a confluence of factors:

- Increasing parallelism (and therefore more components to fail) [6, 7]
- Decreasing transistor feature sizes, making individual components more vulnerable
- Extremely tight peak power requirements [63], limiting the use of hardware redundancy to increase reliability

As these trends continue, hardware vendors may succumb to the temptation to expose incorrect arithmetic or memory corruption to application codes [61, 63, 71]. Some studies already indicate that this behavior is appearing at the user level [46]. In fact, some researchers actively promote relaxing hardware correctness to save energy [64].

4.3.1 Relation to Prior Work

Much of the prior work on fault-tolerant iterative solvers has taken the approach of assuming some fault model for bit flips, and then injecting bit flips into specific numerical operations [82, 83, 86], or treating the application as a black box and injecting bit flips arbitrarily [14]. A popular operation to analyze is sparse matrix-vector multiply [82, 86], a key kernel in iterative linear solvers. These approaches typically engineer a response that mitigates, detects, or detects and corrects bit flips injected following the assumed fault model. The focus in this type of research has been to detect errors, and then respond – e.g., correct the tainted values, or roll back and resume computation from an assumed valid state – assuming that the fault does not occur frequently enough to cause stagnation.

In addition, all prior work on sparse iterative methods is based on a fault model that assumes multiple bit flips injected at some rate. Most studies are also carried out with little care for whether the bit flipped is a $0 \rightarrow 1$ or a $1 \rightarrow 0$, and most studies flip bits at random locations. We question many of the assumptions made, and in general question the research approach.

SDC is a Rare Event

We begin by questioning models and experiments that assume SDC happens at a sufficiently high rate for multiple events to occur in a single linear solve.

We have a strong reason to believe that SDC is a rare event. Hardware incorporates a fairly large amount of safeguards in-place to protect data and instructions. For example, Intel provides the Machine Check Architecture, which provides reporting of bit errors at the register, cache (L1-L3), QuickPath Interconnect, and DRAM (via ECC) layers. We do not attempt to conjecture about the likelihood of bit flips, rather we turn to the theoretical basis that an algorithm is built on, and study how the algorithm behaves when perturbed within the bounds imposed by mathematical analysis.

Current research by Michalak et al. found SDC occurred rarely [70]. They placed a Roadrunner node in front of a neutron cannon and bombarded it with particles. While the neutron fluxes are far beyond realistic, their observations showed a startlingly low occurrence of SDC, while outright node failure occurred far more frequently. Why then has current SDC research focused on failure rates?

In practice, SDCs should remain rare, even at extreme scales of parallelism. Nevertheless, little if any current research has attempted to explain how a single SDC event impacts an algorithm and ultimately the solution. Research on how to counter multiple bit flips has not provided additional insight on the cause/effect relationship.

Also, an application may attribute much of its run time to linear solves, but typically these are multiple linear solves, e.g., an implicit time stepping algorithm that solves a nonlinear system

at each time step. For example, see Müller and Scheichl where a nonlinear system of size 10^{10} is solved and the linear solver is restricted to 0.003 seconds per solve [73].

Fault Models and Silent Data Corruption

At a higher level, we challenge the research approach of assuming a fault model for SDC. By definition, the origin of silent data corruption is unknown, with one such origin being a silent bit flip. Instead of characterizing SDC, the studies propose solutions to a problem we understand only poorly. It is our goal first to analyze the effects of SDC, and then to propose both specific algorithmic techniques and general heuristics that minimize its impact, should it occur. With this ability, mathematicians, scientists, and engineers can take quantifiable steps to develop algorithms and applications that are inherently resilient to SDC.

In numerical algorithms using IEEE-754 floating-point data, regardless of the cause, SDC will produce either numeric values or the non-numeric infinity (Inf) and not-a-number (NaN) values. Injecting bit flips will produce either type of error, making the act of injecting a bit flip to study transient SDC unnecessary as the outcome could have been achieved by merely setting the memory location equal to some value. We know from the IEEE-754 specification precisely what numeric values are possible, and given the mystery of how, when, and where SDC originates, any of the possible floating-point values are plausible.

We advocate a drastically different approach, namely that SDC impacts the underlying mathematical assumptions that guarantee convergence of an algorithm. Rather than focusing on detecting binary errors, we treat bit flips as numerical errors and evaluate how these errors relate to the theoretical basis that the algorithm is built on. In this sense, we filter values that are theoretically impossible, while accepting variations that are allowable by the theory. While our approach does not "solve" the SDC problem, we exploit modern mathematical techniques, so-called flexible solves, to cope with the bounded error we "run through".

4.3.2 Invariants as Detectors

Numerical algorithms often have invariants that they can check inexpensively to decide whether hardware faults have corrupted an intermediate result enough for it not to be useful. For example, Chen [20] performs additional computation and parallel communication in order to check invariants of the iterative linear solvers GMRES [78], CG, and BiCG. If those invariants are violated, the solver can roll back one or more iterations and resume from the last known correct point. In this work, we develop invariants that require no additional parallel communication and very little extra computation to check. This reduces the amount of state needed to roll back correctly, since we can afford to check these invariants at every iteration. In fact, GMRES (and

variants, like "Flexible GMRES") keeps enough state on its own that, unlike in Chen's work, we do not need to save anything to a persistent store.

Checking invariants naturally fits into the layered approach we mentioned in the introduction. In the case of FT-GMRES, the outer solver (based on Flexible GMRES [76]) can check the results of the unreliable inner solves by computing a residual reliably. The outer solver will never compute the wrong answer, no matter what the inner solves do. We present findings in this work that indicated that a layered approach coupled with our theory-based detector can tolerate a single SDC event with little (if any) impact on convergence.

4.4 Sandbox Reliability

Relaxing reliability of *all* data and computations may result in all manner of undesirable and unpredictable behavior. If instructions, pointers, array indices, and Boolean values used for decisions may change arbitrarily at any time, we cannot assert anything about the results of a computation or the side effects of the program, even if it runs to completion without abnormal termination. One node may even corrupt the state of other nodes, for example by overwriting parts of memory owned by a user-space communication library, or performing incorrect output to a shared filesystem. The least we can do is force the fault-susceptible program to execute in a *sandbox*. This is a general idea from computer security, that allows the execution of untrusted "guest" code in a partition of the computer's state (the "sandbox") that protects the rest of the computer (the "host") from the guest's possibly bad behavior. Sandboxing can even protect the host against malicious code that aims to corrupt the system's state, so it can certainly handle code subject to unintentional faults in data and instructions.

Sandboxes *ensure isolation* of a possibly unreliable phase of execution. They *allow data to flow between reliable and unreliable* phases of execution. Also, they let the host *force guest code to stop* within a predefined finite time, or if the host suspects the guest may have wandered astray. This feature is especially important in distributed-memory computation for preventing deadlock and other failures due to "crashed" or unresponsive nodes. In general, sandboxing converts some kinds of hard faults into soft faults, and limits the scope of soft faults to the guest subprogram.

Sandboxing may be implemented in different ways. For example, the guest may run in a virtual machine on the same hardware as the host, or the host may be implemented as redundant processes or systems. Guests may run on a fast but unreliable subsystem, and the controlling host program may run on a reliable but slower subsystem. We do not specify or depend on a particular implementation of sandboxing in this work.

The fault-tolerant inner-outer iteration, described in § 4.6, uses the sandbox model. There, the guest program performs the task "Solve a given linear system." The host program invokes

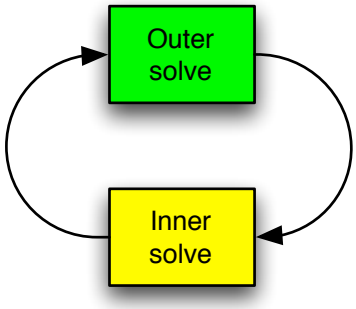


Figure 4.2: Sandbox reliability implemented as reliable outer solves and unreliable inner solves.

the guest repeatedly for different right-hand sides, and the host performs its own calculations reliably. Finer-grained models of reliability may improve the accuracy of the inner solves, which is what our detector in § 4.5.2 accomplishes.

The sandbox model of reliability makes only two promises of the unreliable guest: it returns something (which may not be correct), and it completes in fixed time. These already suffice to construct a working fault-tolerant iterative method, as we will show in § 4.6. However, detecting faults or being able to limit how faults may occur would also be useful. These finer-grained models of reliability can be used to improve accuracy of the iterative method, or to prove more specific promises about its convergence.

4.5 GMRES

The Generalized Minimum Residual method (GMRES) of Saad and Schultz [78] is a Krylov subspace method for solving large, sparse, possibly non-symmetric linear systems of the form $\mathbf{Ax} = \mathbf{b}$. GMRES is based on the Arnoldi process [5], which can also be used to approximate a matrix's eigenvalues and eigenvectors. GMRES has the convenient property that the residual norm of the approximate solution at each iteration is monotonically non-increasing, assuming correct arithmetic and storage. Its use of orthogonal projections and normalized (to length one) basis vectors also has advantages, that we will discuss below.

We begin this section by explaining how to use properties of the Arnoldi process to detect faults in an iteration of GMRES. We then apply the SDC models we developed above to show how to scale the linear system in a way that enhances fault detection and bounds the possible error of the major computational kernels. We will show in future work that these bounds by themselves do not suffice to bound the solution error. Nevertheless, they can, if one makes inexpensive changes to how GMRES computes the solution update coefficients.

4.5.1 Fault Detection via Projection Coefficients

The norms and inner products that occur in each iteration of the Arnoldi process in GMRES have a bounded absolute value. The bound depends on the norm of the preconditioned matrix, which is inexpensive to estimate. We use this bound to detect faults in all the major computational kernels in GMRES.

Algorithm 4.1 GMRES

Input: Linear system $\mathbf{Ax} = \mathbf{b}$ and initial guess $\mathbf{x}_0$
Output: Approximate solution $\mathbf{x}_m$ for some $m \geq 0$

1: $\mathbf{r}_0 := \mathbf{b} - \mathbf{Ax}_0$ ▷ Unpreconditioned initial residual vector
2: $\beta := \|\mathbf{r}_0\|_2$, $\mathbf{q}_1 := \mathbf{r}_0/\beta$
3: **for** $j = 1, 2, \ldots$ until convergence **do**
4: $\quad \mathbf{v}_{j+1} := \mathbf{Aq}_j$ ▷ Apply the matrix A
5: $\quad$ **for** $i = 1, 2, \ldots, j$ **do** ▷ Orthogonalize
6: $\quad\quad h_{i,j} := \mathbf{q}_i \cdot \mathbf{v}_{j+1}$
7: $\quad\quad \mathbf{v}_{j+1} := \mathbf{v}_{j+1} - h_{i,j}\mathbf{q}_i$
8: $\quad$ **end for**
9: $\quad h_{j+1,j} := \|\mathbf{v}_{j+1}\|_2$
10: $\quad$ **if** $h_{j+1,j} \approx 0$ **then**
11: $\quad\quad$ Solution is $\mathbf{x}_{j-1}$ ▷ Happy breakdown
12: $\quad\quad$ **return**
13: $\quad$ **end if**
14: $\quad \mathbf{q}_{j+1} := \mathbf{v}_{j+1}/h_{j+1,j}$ ▷ New basis vector
15: $\quad \mathbf{y}_j := \arg\min_y \|\mathbf{H}(1{:}j+1, 1{:}j)\mathbf{y} - \beta\mathbf{e}_1\|_2$
16: $\quad \mathbf{x}_j := \mathbf{x}_0 + [\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_j]\mathbf{y}_j$ ▷ Compute solution update
17: **end for**

4.5.2 Bounds on the Arnoldi Process

We start our analysis by bounding the dot product which determines the i-th upper Hessenberg entry, $h_{i,j}$ of the j-th Arnoldi iteration. The Arnoldi process is expressed on Lines 3–14 in Algorithm 4.1. At its core is an orthogonalization kernel, which we have chosen to be the Modified Gram-Schmidt (MGS) process. Classical Gram-Schmidt or Householder transformations may also be used. As we will demonstrate, our bound is invariant of the orthogonalization algorithm chosen.

The MGS process begins on Line 5 and completes on Line 8. To bound $h_{i,j}$ on Line 6, we exploit a property of orthogonal projections. It is well known that linear transforms utilizing orthogonal matrices are *isometric*. That is, they preserve the length of the vectors. In $\mathbb{R}^n$,

the dot product of a vector with a unit-length vector is bounded by the length of the first vector. This means that each $h_{i,j}$ entry is bounded by the length of the vector that starts the orthogonalization process (the vector we wish to make orthogonal). To clarify what "starts" the orthogonalization process means, we step back from an algorithmic formulation, and instead write the orthogonalization kernel as a mathematical expression. For clarity, we will use the Classical Gram-Schmidt expression:

$$\mathbf{w} = \left[\mathbf{I} - \mathbf{Q}^{\mathrm{T}}\mathbf{Q}\right]\mathbf{u} \tag{4.1}$$

In Eq (4.1), the vector $\mathbf{u}$ is what "starts" the orthogonalization process, and $\mathbf{w}$ is the resulting vector, which is orthogonal to all vectors in $\mathbf{Q}$, where $\mathbf{Q} = \{\mathbf{q}_1, \ldots, \mathbf{q}_j\}$.

Returning to Algorithm 4.1, what "starts" the orthogonalization process is the vector resulting from Line 4. If we can bound the length of this vector, then we know the maximum absolute value that $h_{i,j}$ can take. Since we want to bound the length of the resulting vector, we take the induced ℓ^2 norm, $\|\mathbf{v}_{j+1}\|_2 = \left\|\mathbf{A}\mathbf{q}_j\right\|_2$,

$$\|\mathbf{v}_{j+1}\|_2 \leq \|\mathbf{A}\|_2 \left\|\mathbf{q}_j\right\|_2 . \tag{4.2}$$

We can further reduce the bound, recognizing that the basis vector $\mathbf{q}_j$ is a unit vector, i.e., $\|\mathbf{q}\|_2 = 1$. We may deal with $\|\mathbf{A}\|_2$ in several ways:

1. $\|\mathbf{A}\|_2$ is defined to be the largest singular value, e.g., $\sigma_{\max}(\mathbf{A})$, or
2. the 2-norm is bounded above by the Frobenius norm, which is likely cheaper to compute than the largest singular value.

This leads us to an upper bound on *all* entries in the upper Hessenberg matrix

$$|h_{i,j}| \leq \|\mathbf{A}\|_2 \leq \|\mathbf{A}\|_F . \tag{4.3}$$

The bound presented in Eq. (4.3) is crucial, as it demonstrates that the upper Hessenberg entries are bounded entirely by the input matrix. In § 4.6, we discuss Flexible GMRES with GMRES (Algorithm 4.1) as a preconditioner. In this scenario, the bound presented is invariant for all applications of the preconditioner, or, in other words, the bound depends *only* on the input matrix.

4.5.3 Bound Application

We have shown what the theoretical upper limit is for the values in the upper Hessenberg. This essentially tells us what is *theoretically possible* inside the Arnoldi process. Using this approach

to construct an SDC detector is significant. By building a detection scheme in this way, we know precisely what errors we can detect, and, more importantly, we know what is not detectable.

The important factor to keep in mind is that exactly how an error is committed is irrelevant, the norm bounds allow us to filter out values that are invalid by theory — we either detect a large error or commit a small error, and in § 4.6 we will demonstrate how restricting the magnitude of the error committed allows Flexible GMRES to tolerate the error.

4.5.4 Error Detection

In the context of error detection, we can only detect an error that exceeds the bound on the upper Hessenberg entry h_{ij}. To do this, we insert a conditional between Lines 6 and 7 and Lines 9 and 10 and test whether $|h_{ij}| \leq \|\mathbf{A}\|_F$. Should this condition be invalid, then we assume that we have committed an error at some point.

4.6 FT-GMRES

This section describes the Fault-Tolerant GMRES (FT-GMRES) algorithm, a Krylov subspace method for an iterative solution of large sparse linear systems of the form $\mathbf{Ax} = \mathbf{b}$. FT-GMRES computes the correct solution $\mathbf{x}$ even if the system experiences uncorrected faults in both data and arithmetic [13]. It promises "eventual convergence", i.e., it will always either converge to the right answer, or (in rare cases) stop and report immediately to the caller if it cannot make progress. FT-GMRES accomplishes this by dividing its computations into *reliable* and *unreliable* phases, using the sandbox model of reliability described in § 4.4. Rather than rolling back any faults that occur in unreliable phases, as a checkpoint / restart approach would do, FT-GMRES "rolls forward" through any faults in unreliable phases, and uses the reliable phases to drive convergence. FT-GMRES can also exploit fault detection in order to correct corrupted data during unreliable phases.

4.6.1 FT-GMRES is Based on Flexible GMRES

FT-GMRES is based on Flexible GMRES (FGMRES) [76]. FGMRES, presented in Algorithm 4.2, extends the Generalized Minimal Residual (GMRES) method of Saad and Schultz [78] by "flexibly" allowing the preconditioner to change in every iteration. An important motivation of flexible methods are "inner-outer iterations," which use an iterative method itself as the preconditioner (e.g., use GMRES as a preconditioner). In this case, "solve $\mathbf{q}_j := \mathbf{M}_j\mathbf{z}_j$" Line 4 means "solve the linear system $\mathbf{Az}_j = \mathbf{q}_j$ approximately using a given iterative method." For example, suppose GMRES is implemented as a function $\mathbf{x} = \mathsf{gmres}(\mathbf{A}, \mathbf{b})$, meaning *solve* $\mathbf{Ax} = \mathbf{b}$ *for* $\mathbf{x}$. Then Line 4 is equivalent to $\mathbf{z}_j = \mathsf{gmres}(\mathbf{A}, \mathbf{q}_j)$.

Algorithm 4.2 Flexible GMRES (FGMRES)

Input: Linear system $Ax = b$ and initial guess x_0
Output: Approximate solution x_m for some $m \geq 0$

$\mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0$ ▷ Unpreconditioned initial residual
$\beta := \|\mathbf{r}_0\|_2$, $\mathbf{q}_1 := \mathbf{r}_0/\beta$
for $j = 1, 2, \ldots$ until convergence **do**
    Solve $\mathbf{q}_j = \mathbf{M}_j\mathbf{z}_j$ ▷ Apply current preconditioner
    $\mathbf{v}_{j+1} := \mathbf{A}\mathbf{z}_j$ ▷ Apply the matrix $\mathbf{A}$
    for $i = 1, 2, \ldots, k$ **do** ▷ Orthogonalize
        $h_{i,j} := \mathbf{q}_i \cdot \mathbf{v}_{j+1}$
        $\mathbf{v}_{j+1} := \mathbf{v}_{j+1} - h_{i,j}\mathbf{q}_i$
    end for
    $h_{j+1,j} := \|\mathbf{v}_{j+1}\|_2$
    Update rank-revealing decomposition of $\mathbf{H}(1{:}j, 1{:}j)$
    if $\mathbf{H}(j+1, j)$ is less than some tolerance **then**
        if $\mathbf{H}(1{:}j, 1{:}j)$ not full rank **then**
            Did not converge; report error
        else
            Solution is $\mathbf{x}_{j-1}$ ▷ Happy breakdown
        end if
    else
        $\mathbf{q}_{j+1} := \mathbf{v}_{j+1}/h_{j+1,j}$
    end if
    $\mathbf{y}_j := \arg\min_y \|\mathbf{H}(1{:}j+1, 1{:}j)\mathbf{y} - \beta\mathbf{e}_1\|_2$
    $\mathbf{x}_j := \mathbf{x}_0 + [\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_j]\mathbf{y}_j$ ▷ Compute solution update
end for

This *inner solve* step preconditions the *outer solve* (in this case FGMRES). Changing right-hand sides and possibly changing stopping criteria for each inner solve means that if one could express the "inner solve operator" as a matrix, it would be different on each invocation. This is why inner-outer iterations require a flexible outer solver.

Flexible methods let the preconditioner change significantly from one iteration to another; they do not depend on the difference between successive preconditioners being small. This is the key observation behind FT-GMRES: flexible iterations allow successive inner solves to differ arbitrarily, even unboundedly. This suggests modeling faulty inner solves as "different preconditioners." Taking this suggestion leads to FT-GMRES.

There are flexible versions of other iterative methods besides GMRES, such as CG [45] and QMR [89], which could also be used as the outer solver. We chose FGMRES because it is easy to implement, robust, and can handle nonsymmetric linear systems. Experimenting with other flexible outer iterations is future work.

4.6.2 Sandbox Reliability

FT-GMRES further specifies different reliability for inner and outer solves. Only inner solves (Line 4) are allowed to run unreliably. FT-GMRES expects that inner solves do most of the work, so inner solves run in the less expensive unreliable mode. Inner solvers need only return with a solution in finite time (see §4.4). That solution may be completely wrong if errors occurred.

This inner-outer solver approach reduces disruption of existing solvers. The outer FGMRES iteration wraps any existing solver with any preconditioner that it might be using as the inner solver. Any solver works, but since we have developed a fault detector in § 4.5, we chose GMRES as the inner solver.

4.6.3 FGMRES' Additional Failure Modes

FGMRES (and therefore FT-GMRES) have an additional failure mode beyond those of standard GMRES. On Line 11 of standard GMRES (Algorithm 4.1), $h_{j+1,j} = 0$ indicates that the current iteration produced an invariant subspace. This means either that we converged to the exact solution, or that the solve cannot make further progress given the initial guess. For FGMRES, if the quantity $h_{j+1,j} = 0$, this does not necessarily indicate either case. This is because $\mathbf{H}(1{:}j, 1{:}j)$ is always nonsingular in GMRES if j is the smallest iteration index for which $h_{j+1,j} = 0$, whereas in FGMRES, $\mathbf{H}(1{:}j, 1{:}j)$ may nevertheless be singular in that case. (This is Saad's Proposition 2.2 [76].) This can happen even in exact arithmetic. It may occur due to unlucky choices of the preconditioners, e.g., $\mathbf{M}_j^{-1} = \mathbf{A}$ and $\mathbf{M}_{j+1}^{-1} = \mathbf{A}^{-1}$. In practice, this case is rare, even when inner solves are subject to faults. Furthermore, it can be detected inexpensively, since there are algorithms for updating a rank-revealing decomposition of an $m \times m$ matrix in $O(m^2)$ time (see

e.g., Stewart [88]). This incurs no more time than it takes to update the QR factorization of the upper Hessenberg matrix at iteration m. The ability to detect this rank deficiency ensures "trichotomy" of FGMRES: it either

1. converges to the desired tolerance,
2. correctly detects an invariant subspace, with a clear indication ($h_{j+1,j} = 0$ and $\mathbf{H}(1{:}j, 1{:}j)$ is nonsingular), or
3. gives a clear indication of failure (detected rank deficiency of $\mathbf{H}(1{:}j, 1{:}j)$).

We base FT-GMRES' "eventual convergence" on this trichotomy property. In the following section, we will discuss how the techniques used to detect the third failure case can also be used to keep the *inner* solves' solutions bounded, as long as faults in the inner solves are bounded.

4.6.4 Fault Tolerance via Regularization

Both GMRES and Flexible GMRES compute the solution update coefficients ($\mathbf{y}_j$ in all algorithms) by solving a small least-squares problem. This problem originates from projecting the matrix $\mathbf{A}$ onto the Krylov basis, so we call it the *projected least-squares problem.* At iteration k (counting from $k = 1$), it has the form

$$\text{Find } \mathbf{y} \text{ satisfying } \min_y \|\mathbf{H}_k \mathbf{y} - \beta \mathbf{e}_1\|_2, \tag{4.4}$$

where $\mathbf{H}_k$ is a $k+1$ by k upper Hessenberg matrix, $\mathbf{y}$ the k coefficients of the solution update, β the norm of the initial residual vector, and $\mathbf{e}_1$ the length $k+1$ vector whose first entry is one and whose remaining entries are zero.

Saad and Schultz [78] solve this problem by a structured QR factorization. This method lets implementations keep the intermediate reductions of Steps 1 and 2 at each iteration. This makes the cost to compute the solution update $O(k^2)$ coefficients rather than $O(k^3)$. However, it can produce unboundedly inaccurate coefficients if the upper triangular matrix $\mathbf{R}_k$ is singular or ill-conditioned. In GMRES without faults, this does not normally occur if the matrix $\mathbf{A}$ is not numerically rank deficient. A numerically rank-deficient upper Hessenberg matrix normally indicates convergence[1] at the iteration where it becomes rank deficient.

Linear least-squares problems like (4.4) always have a solution. However, a singular upper Hessenberg matrix may make the solution set infinite, with unbounded norm. Unbounded norm in GMRES' update coefficients means unbounded error in its solution to $\mathbf{Ax} = \mathbf{b}$. A *nearly* singular upper Hessenberg matrix may similarly result in large inaccurate coefficients, by increasing sensitivity to rounding error in the triangular solve. In § 4.6.3, we recommended detecting this case by using a rank-revealing decomposition that supports incremental updates

[1] It may also indicate that the algorithm cannot make further progress for the user's choice of initial guess, given $\mathbf{A}$ and $\mathbf{b}$.

in order to preserve the $O(k^2)$ cost while detecting rank deficiency. This only detects whether the matrix is close to singular; it does not prescribe a policy for handling (near) singularity.

We define this policy by introducing an additional constraint, that the solution to (4.4) have minimum norm. We can do this by using a rank-revealing decomposition that truncates zero singular values. We can also introduce a tolerance in order to allow small but nonzero singular values. This approach bounds the update coefficients as a function of the largest singular value of the upper Hessenberg matrix, divided by the least singular value not truncated. This is more *robust* than Saad and Schultz's method, where "robust" means "insensitivity to errors in the input."

We can apply the robust technique to the upper triangular system $\mathbf{R}_k\mathbf{y} = \mathbf{z}_k$, after computing and applying the Givens rotations. This is equivalent (in terms of accuracy with respect to rounding error) to a rank-revealing factorization of $\mathbf{H}_k$, and lets us easily switch "robustness" on or off for experiments. We implemented the following approaches to solve $\mathbf{R}_k\mathbf{y} = \mathbf{z}_k$:

1. Standard triangular solve (Saad and Schultz's approach)
2. Attempt a standard triangular solve, and only use a rank-revealing method if its solution has `Inf` or `NaN` values
3. Always use a rank-revealing method

For our experiments, we used a singular-value decomposition as the rank-revealing factorization, as an easier to implement and more accurate substitute for the factorization suggested in our previous work. We recommend either Approach 1 or 3. Approach 2 conceals the natural error detection that comes with IEEE-754 floating-point data, without detecting inaccuracy or bounding the error.

4.7 Results

To evaluate FT-GMRES and our inner solver bound we explore the impact on time-to-solution (iteration count) given a fault in *all* inner solves. To perform these experiments we developed a two-level solver ("nested solver") that uses FT-GMRES as the outer solver, and GMRES as the inner solver (preconditioner). We used the Trilinos framework [49] with FT-GMRES and GMRES implemented as Tpetra operators.

4.7.1 Sample Problems

We have chosen two sample matrices to demonstrate our technique. To ensure reproducibility, we did not create either of these matrices from scratch, rather we used readily available matrices. The first matrix is fairly common and arises from the finite difference discretization of the Poisson equation. This matrix is symmetric and positive definite, meaning that it could be solved

using the Conjugate Gradient method. We generated this matrix using MATLAB's built-in Gallery functionality. The second matrix chosen presents a more realistic linear system. The mult_dcop_03 matrix comes from the University of Florida Sparse Matrix Collection [24]. It arises from a circuit simulation problem. The matrix is nonsymmetric and not positive definite, meaning Conjugate Gradient could not be used to solve the system. The matrix is fairly small, but is very ill-conditioned, which means that small perturbations may have a large impact. We have summarized the characteristics of each matrix in Table 4.1. Note that we have included the

Table 4.1: Sample Matrices

Properties	Poisson Equation	mult_dcop_03
number of rows	10,000	25,187
number of columns	10,000	25,187
nonzeros	49,600	193,216
structural full rank?	yes	yes
nonzero pattern symmetry	symmetric	nonsymmetric
type	real	real
positive definite?	yes	no
Condition Number	6.0107×10^3	7.27261×10^{13}
Potential Fault Detectors		
$\|\mathbf{A}\|_2$	8	17.1762
$\|\mathbf{A}\|_F$	446	42.4179

potential fault detectors in Table 4.1. These represent the upper bound on what is acceptable for an upper Hessenberg entry.

Significance of Test Problems

The test problems above represent two classes of matrices: symmetric positive definite (SPD) and nonsymmetric. Different linear solvers require matrices to have specific attributes. Conjugate Gradient expects an SPD matrix, while GMRES can accept both symmetric and nonsymmetric matrices. Relevant to this work, the **H** matrix discussed throughout this chapter has a unique structure if the input matrix is symmetric. By structure, we refer to the nonzero pattern of **H**. For nonsymmetric systems, **H** is upper Hessenberg, while for SPD systems, **H** is tridiagonal (a special case of upper Hessenberg), e.g., see Figure 4.3. The fact that solving the Poisson matrix with GMRES *should* create a tridiagonal matrix is key. This means that specific dot products in the orthogonalization phase should create entries "near zero". If we perturb those entries (as we are about to do) we can see large penalties in time to solution.

$$\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \end{bmatrix} \quad \text{vs.} \quad \begin{bmatrix} \times & \times & 0 & 0 \\ \times & \times & \times & 0 \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \end{bmatrix}$$

Figure 4.3: Upper Hessenberg and tridiagonal matrices.

4.7.2 Time to Solution Experiments

In this experiment, we solve a linear system and determine how many iterations are required to obtain a solution. This is a *failure-free* run that tells us how many outer iterations (and inner iterations) are required to obtain a solution. We then solve the same linear system again (same matrix, right-hand side, and initial guess), and, on the first iteration of the first inner solve, we perturb the upper Hessenberg entry ($h_{i,j}$) on the first iteration of the orthogonalization loop (Line 6 in Algorithm 4.1). We then repeat this process, applying the fault on all possible inner solve iterations on the first step of the orthogonalization process. Note that each experiment injects a single occurrence of SDC.

For example, in Figure 4.4 the x-axis denotes a specific experiment. The x-axis range is determined by how many outer iterations are required to obtain a solution in a failure-free environment (the aggregate number of inner iterations is $25 \times$ num_outer). The y-axis denotes the number of outer iterations required to obtain a solution given a specific error injected. The three top subplots represent error introduced at the start of orthogonalization, while the lower three subplots represent error introduced at the end of orthogonalization.

The choice to inject the fault on the first iteration of the orthogonalization loop is justified as follows: By faulting early in the orthogonalization phase, you "corrupt" the basis vector from the start, i.e., because we choose Modified Gram-Schmidt the fault will "taint" all subsequent iterations of the orthogonalization loop (worst-case scenario).

Fault Values

To inject a fault, we only need to modify or replace the current $h_{i,j}$ in Algorithm 4.1 Line 6 with an incorrect value. Directly injecting `NaN` or `Inf` reveals nothing, since we can clearly detect such faults. We inject an SDC that that represents 3 classes of faults, and these fault values are relative to the *correct value*:

1. very large, $\tilde{h}_{i,j} = h_{i,j} \times 10^{+150}$,
2. slightly smaller, $\tilde{h}_{i,j} = h_{i,j} \times 10^{-0.5}$, and
3. very small (nearly zero), $\tilde{h}_{i,j} = h_{i,j} \times 10^{-300}$.

In this experiment, we only record how many iterations it takes to obtain a solution. It should be noted that for this experiment parallelism is not a factor, and we are interested in observing how the solvers behave when perturbed. In particular, this experiment investigates the solver's behavior when undetectable faults are injected, and it demonstrates a benefit from filtering obviously faulty (i.e., large) values. In the following figures, class 2 and 3 faults represent undetectable faults, while class 1 represents a case that we could detect and to which we could respond, e.g., by halting the application or restarting the inner solve.

4.7.3 Faults in an SPD Problem

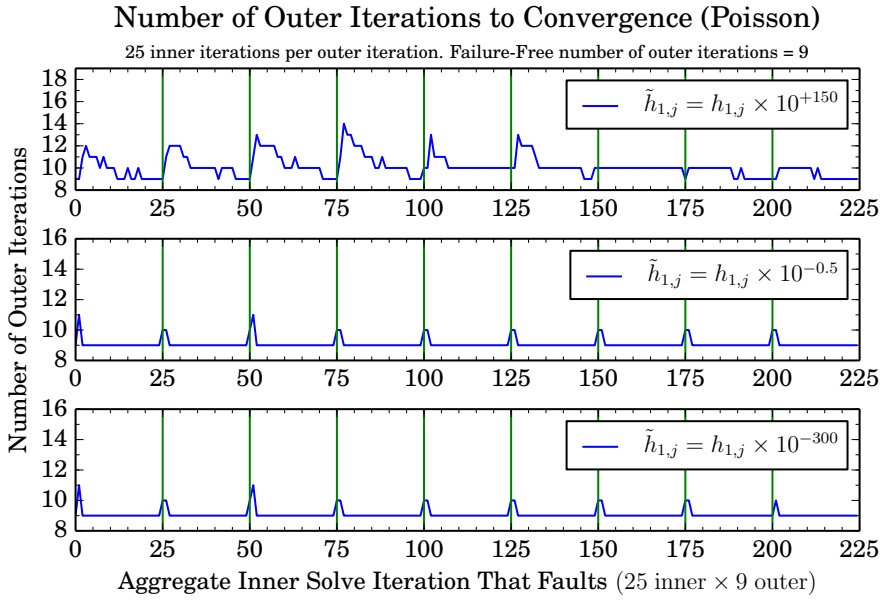
Figure 4.4 illustrates the case of using GMRES to solve an SPD system of equations. **In a failure-free solve FT-GMRES required 9 outer iterations, with each inner solve performing 25 inner iterations.** In this case, **H** should be tridiagonal, meaning that in Figure 4.4a for the first inner solve, the first entry created by the Modified Gram-Schmidt (MGS) loop, $h_{1,*}$, should be zero from inner iteration 3 onward. In contrast, Figure 4.4b faults on the last iteration of the MGS loop, and the last entry in this column of **H** can theoretically be nonzero.

Faulting on the First Modified Gram-Schmidt Iteration

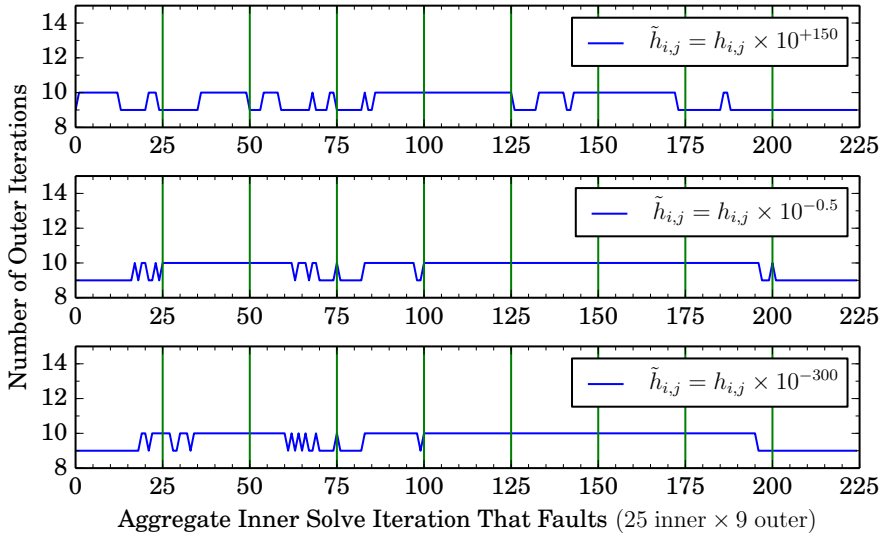
In Figure 4.4a, we see a large penalty in time to solution for large faults. This is due to making entries in **H** that should be zero, clearly nonzero. In contrast, if we only slightly perturb these "near zero" entries (class 2 and 3 errors), we see very little impact on time to solution. The largest increase in outer iterations is two, while the majority of experiments resulted in no increase in time to solution. It should be noted that if our fault detector on $h_{i,j}$ was used, the top plot (large fault) would not be possible.

Faulting on the Last Modified Gram-Schmidt Iteration

Faulting on the last Modified Gram-Schmidt iteration is much different from faulting on the first for an SPD problem, because the last $h_{i,j}$ entry created in the orthogonalization phase could theoretically be nonzero. From figure 4.4b we see that the worst case is that we incur one additional outer iteration. Considering both faults at the start and end of the MGS process, we see that with our detector we see a maximum increase in outer iterations to be 2, in contrast if our detector were not used, we see increase in outer iterations of 5.

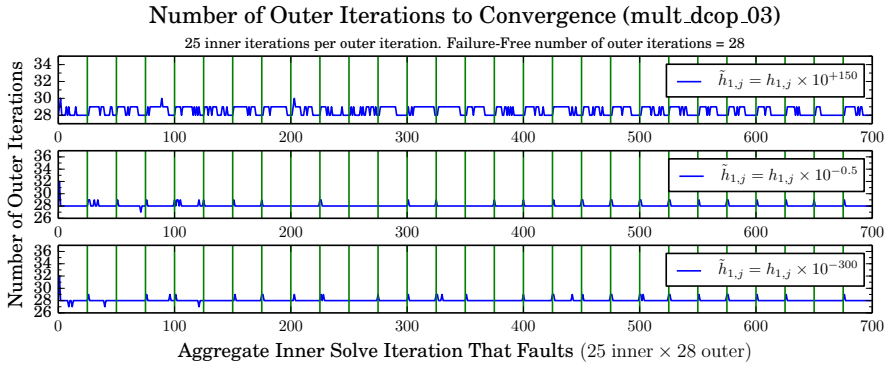


(a) SDC on the first iteration of the Modified Gram-Schmidt loop.



(b) SDC on the last iteration of the Modified Gram-Schmidt loop.

Figure 4.4: Number of outer iterations required for convergence when solving a Poisson equation given a single SDC event injected in the orthogonalization phase of the inner solve. Vertical bars indicate the start of a new inner solve.



(a) SDC on the first iteration of the Modified Gram-Schmidt loop.

(b) SDC on the last iteration of the Modified Gram-Schmidt loop.

Figure 4.5: Number of outer iterations required for convergence when solving the mult_dcop_03 system of equations given a single SDC event injected in the orthogonalization phase of the inner solve. Vertical bars indicate the start of a new inner solve.

4.7.4 Faulting in a Nonsymmetric Problem

We now consider a problem that is not symmetric, meaning that all $h_{i,j}$ we perturb may be zero, but could also be nonzero — but each entry in $\mathbf{H}$ is still subject to the bound from Eq. (4.3). **In a failure-free solve FT-GMRES required 28 outer iterations, with each inner solve performing 25 inner iterations.** As in our prior analysis, we consider faults in both the first and last iteration of the Modified Gram-Schmidt process.

Faulting on the first Modified Gram-Schmidt iteration

As expected, in Figure 4.5 we see a very different characteristic for faults on the first MGS iteration. For large faults we see a maximum increase in time to solution to be 2 outer iterations. For small faults, we see that the first iteration of the MGS of the first inner solve is extremely vulnerable to small faults. For class 2 and 3 faults, we see a maximum increase in outer iterations of 4. If we ignore the first 3 iterations of the inner solve we see at most 1 additional outer

iteration. The worst-case increase in time to solution actually occurs on the 2^{nd} inner solve iteration, and we leave to future work further analysis of the Arnoldi process to explain this phenomenon. This indicates that additional robustness should be added at the very start of the first inner solve, and we discuss this briefly in our summary.

Faulting on the last Modified Gram-Schmidt iteration

Faulting on the last iteration of the orthogonalization loop again presents a worst case compared to faulting early. That is, by faulting on the last orthogonalization iteration, we see an increase in outer iterations in more cases. We do not the see the sharp increase in iteration count for faults early in the first inner solve iterations. Note, that the first MGS iteration is also the last on the first inner solve iteration, and as stated previously, the first iteration did not exhibit a large increase in iterations.

4.7.5 Summary of Findings

A common feature seen between both SPD and nonsymmetric solves is that faulting early in the first inner solves' orthogonalization is universally bad, resulting in a 33% increase in time to solution for the Poisson problem and 14% increase in time to solution for the mult_dcop_03 problem. These percent increases are not general findings, but we believe this characteristic will hold true in most, if not all, cases. This may indicate that additional effort should be expended early in the first inner solve.

Performance Characteristics of GMRES

The amount of work per-iteration of GMRES increases linearly. This is seen in Algorithm 4.1 in the orthogonalization phase, where the inner loop iterates from 1 to j. Adding redundant computation early in the inner solve would have minimal performance impact because the orthogonalization kernel has substantially less work to perform than in latter iterations. If we included additional robustness only on the first invocation of the inner solver, we can mitigate the one edge-case where we see high variability in time to solution. We leave this to future work.

Filtering values is cheap and effective

In all experiments, we find that exploiting the bound on the upper Hessenberg entries is beneficial, and, in doing so, we typically observe one additional outer iteration as the penalty should a single SDC event occur. We believe that this research approach will yield additional invariants that are cheap to evaluate, and that by combining light-weight mechanisms we drastically reduce the damage that SDC can introduce.

4.8 Conclusions

In summary, we developed a cheap fault detector for the computational intensive orthogonalization stage of GMRES. We then present the FT-GMRES algorithm, and discuss robustness improvements in the local least squares solve. We then explained how are detector and robustness modifications can be used to limit the amount of error that the inner solve may return.

We presented results from two experiments on common classes of matrices that illustrate that our filtering technique is beneficial, and identified the early stages of the first inner solve as being the most vulnerable. Furthermore, we observe that the inner/outer iteration scheme based on FGMRES is extremely robust to single events of SDC in the orthogonalization phase. We find that this nested approach, even when not coupled with invariant checks can cope with even large perturbations introduced by SDC.

Chapter 5

A Numerical Soft Fault Model for Iterative Linear Solvers

5.1 Introduction

Recent studies indicate that large parallel computers will continue to become less reliable as energy constraints tighten, component counts increase, and manufacturing sizes decrease [63, 44]. This unreliability may manifest in two different ways: either as "hard" faults, which cause the loss of one or more parallel processes, or as "soft" faults, which cause incorrect arithmetic or storage, but do not kill the running application. Large-scale systems today experience frequent process loss, from which applications recover using variations of checkpoint / restart (C/R). with current research looking at optimizing the process through, e.g., multi-level checkpointing [72] or by using domain knowledge of algorithms to create checkpoint schemes that have lower overhead [19].

This chapter focuses on soft faults. Specifically, we consider those that corrupt data or computations, without the hardware or system detecting them and notifying the application that a fault occurred. We call this type of soft fault Silent Data Corruption (SDC). SDC is much less frequent than process failures, but much more threatening, since the application may silently return an incorrect answer. In physical simulations, the wrong answer could have costly and even life-threatening consequences. Users' trust in the results of numerical simulations can lead to disaster if those results are wrong, as for example in the 1991 collapse of the Sleipner A oil platform [80]. Unlike with hard faults, applications currently have few recovery strategies. Hardware *detection* without correction may cost nearly as much as full hardware correction. Hardware vendors can harden chips against soft faults, but doing so will increase chip complexity and likely either increase energy usage or decrease performance. An open field of research and the focus of this work is designing algorithms that can tolerate SDC.

5.2 Preconditioned Linear Solvers

Our fault model assesses iterative, possibly preconditioned, linear solvers under faults that are not detectable in standard implementations, and that can remain undetectable using low overhead detectors. The model introduces a *pessimistic* fault. This accounts for our lack of knowledge of exactly which physical events can lead to the worst case for a particular problem and solver combination. We argue that if a numerical approach can tolerate these types of perturbations, then it should be able to tolerate transient arithmetic errors. This minimizes fruitless speculation about how faults manifest in real hardware, and instead asks whether an algorithm can handle challenging numerical faults. The latter presents a fault model that we show produces a much worse case than random bit flips. If it is true that future hardware will allow some transient soft errors, we should assess fault tolerance in algorithms based on a worst-case scenario, rather than the extremely biased case of random bit flips.

5.2.1 Soft Faults and Iterative Methods

Given a direct solver, if a soft fault corrupts arithmetic, the method will reach a (possibly unacceptable) solution in a bounded, known number of steps. Iterative methods behave differently. They may 1) "converge through" the error, taking no more iterations than in the error-free case; 2) converge but take more iterations; 3) *stagnate* — reach the maximum iteration count without improving the initial approximation; or 4) become divergent — oscillate wildly or have rapid error growth such that the solver "explodes" toward infinity. In the latter two cases, the solver fails to produce an acceptable solution. Stagnation relates to the *maximal attainable accuracy*, which bounds below the accuracy an iterative solver can reach in finite-precision arithmetic. If a soft error introduces error sufficient to damage the maximal attainable accuracy, then the solver may stagnate.

Pessimistic faults have mathematical interpretations. For example, they may introduce a fictitious, abnormally large eigenvalue to the matrix A. Iterative solvers approximate the solution as a linear combination of basis vectors that are weighted by the largest eigenvalues in the system. The fault will thus make the solver converge to a bogus solution dominated by the fictitious eigenvalue. Also, iterative solvers are often used for solving discretized versions of elliptic partial differential equations (PDEs). Their solutions must satisfy the *maximum principle*: their maximum must be found on the boundary. One may view a soft fault as a (transient) violation of this principle. Alternatively, a "bad" soft error may make the problem appear to have a nonmathematical discontinuity.

Our fault model evolves from these mathematical interpretations. We model faults as a specific MPI process returning a bad vector from its preconditioner application. We generate faults in two ways: a fault may 1) scale its contribution to the global vector, or 2) permute

its local portion of the global vector. Permutations preserve the vector's norm, while making its contents incorrect. This models discontinuity. Scaling increases or decreases the norm of the vector predictably. This, or directly corrupting inner product or norm results, perturbs the eigenvalue approximations. Corrupting the basis vectors also makes the algorithm search for a solution in the wrong direction. Our numerical fault model suffices to cause stagnation or divergence in non-restarted solvers.

5.2.2 Selective Reliability

Our fault-tolerance strategy rests on relating numerical methods that naturally correct errors to system-level fault tolerance. In particular, we assume a *selective reliability* or "sandboxing" programming model [47] that lets algorithm developers isolate faults to certain parts of the algorithm in a coarse-grained way. In our scheme, we enforce that the outer solver be reliable, while letting the inner solver run in an unreliable mode. We aim to spend most of our computation time in cheap "unreliable" computations, while minimizing the time we spend in the presumably expensive outer solve.

Analytically, any faults that occur in the inner solver manifest as a "different preconditioner" to the outer solver. We choose Flexible GMRES [77] as the outer solver, since it can tolerate a preconditioner that changes between iterations. As an inner solver, we use the Generalized Minimal Residual Method (GMRES) from Saad and Schultz [78]. We show results for this inner/outer solver system, called FT-GMRES, that uses a multigrid preconditioner (MueLu) and solves a Poisson problem. Multigrid is the preferred preconditioner for Poisson problems.

5.2.3 Implementation

We implemented our solvers using the Tpetra [9] sparse linear algebra package in the Trilinos framework [52] and validated them against both MATLAB and the solvers in Trilinos' Belos package [11]. Implementing our solvers using Trilinos lets us benefit from the scalability and performance of its sparse matrices and dense vectors.

5.3 Results

5.3.1 Methodology

We previously described how we corrupt the preconditioner's output. To evaluate the impact of our preconditioned solvers in the presence of SDC, we perform the following steps:

1. Solve the problem injecting no SDC, and compute the number of times, K, the preconditioner was applied.

2. For all j in $[1, K]$, reattempt the solve, introducing SDC at the j-th preconditioner application. This results in K total solves.
3. For all K solves with SDC, compute the relative percent of *additional* preconditioner applies over the SDC-free solve, e.g., $\frac{Applies_{observed} - Applies_{FailureFree}}{Applies_{FailureFree}} \times 100$. If $observed - FailureFree < 0$, i.e., SDC accelerated convergence, we record zero overhead.
4. Repeat Steps 2 and 3, letting various numbers of MPI processes participate in the SDC injection.
5. Repeat Steps 2-4, varying the scaling factor applied to the SDC.
6. For each combination of scaling factor and number of faulty processes, plot the *average* number of additional preconditioner applies as a percentage. 0% means no additional applies; 100% means twice as many.

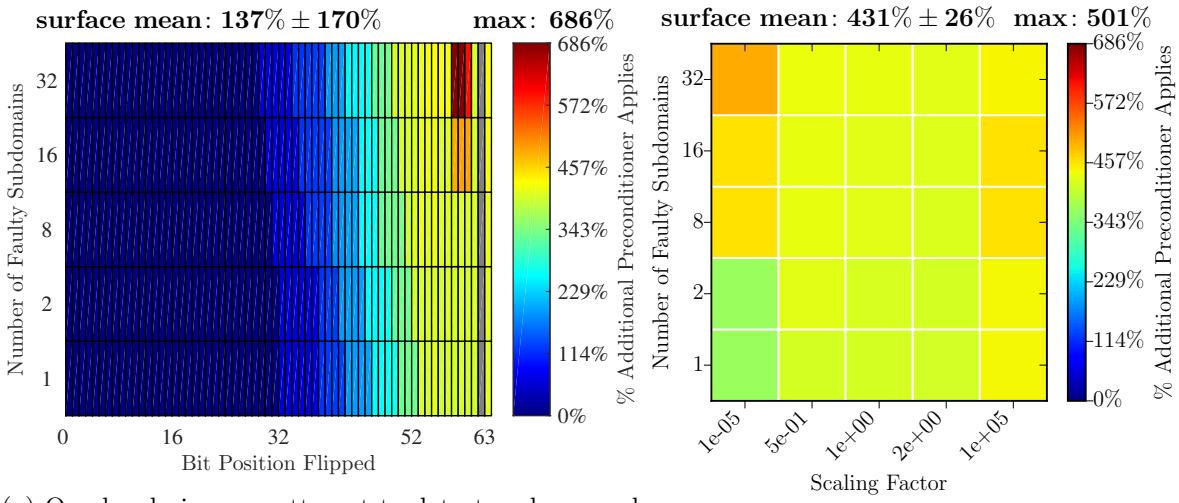
5.3.2 Model Comparisons

Fig. 5.1 shows a side-by-side comparison of the overhead introduced from random bit flips and our numerical fault model. Here, we use no detection mechanism and force our solver to roll through all errors.

Each plot represents a different fault model, so the results cannot be compared geometrically. The intent of the figure is to illustrate the overhead we observe given faults from each model. Recognize the overheads have roughly the same range, yet the variance in Fig. 5.1a is considerably higher than our model (Fig. 5.1b). We address this in greater detail in § 5.3.4.

Note, Fig. 5.1a shows the highest overhead when all 32 subdomains inject the 58th or 59th exponent bit flips. This is not a weakness in our model. Those specific faults introduce very large magnifications into the vector, but not large enough to create an infinite or not-a-number value. Elliott et al. [29] explored exactly this scenario of faults and proposed a low overhead detector that efficiently filters such errors with $O(1)$ cost. For this exact reason our results analyze scaling factors that would slip through such a filter. Only the largest scaling factor, 1×10^5, would be detected by a projection bound.

Next, we enable both explicit residual ($\|Ax - b\|$) and projection bound tests per each inner iteration in Fig. 5.2. The resulting colorbar bounds are similar for both the numerical model and the bit flip model. That is, both models require a maximum overhead in the range of $100\% - 120\%$. This also exposes the trouble with bit flip injection: bit position does not affect the introduced overhead consistently. For example, exponent bits sometimes introduce high overhead, while mantissa bits can introduce overhead proportional to exponent bits. Notice that the right-most column of the numerical fault model is not the highest overhead — this is due to a very low overhead detector, whereas checking the explicit residual requires a preconditioner apply.



(a) Overhead given no attempt to detect and respond to faults with a random bit flip model.

(b) Overhead given no attempt to detect and respond to faults with a numerical fault model.

Figure 5.1: Overhead comparison with no fault detection, for (a) a random bit flip injection and (b) a numerical fault model.

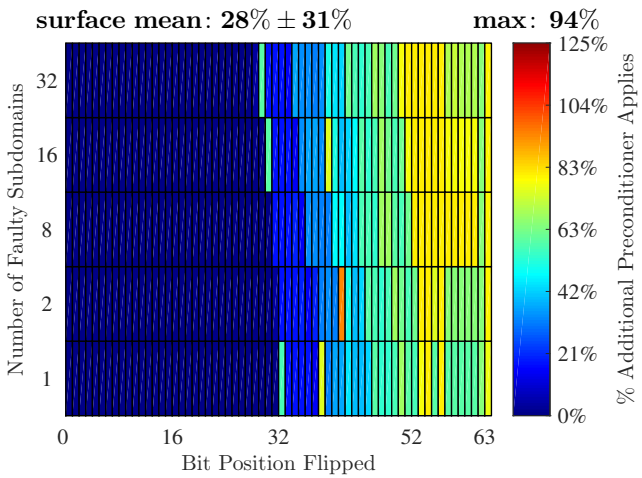
5.3.3 Computational Effort

Our fault model captures hard to detect, yet numerically challenging faults. The largest overhead is easily characterized by our model and requires substantially less experimentation. For example, each shaded region constitutes one trial (of which we compute a mean). Clearly evaluating 25 unique experiments is much cheaper than evaluating 64×5 experiments. Moreover, if the experiment is not designed to account for the bias introduced by random bit flips, the mean will approximate optimistic overheads.

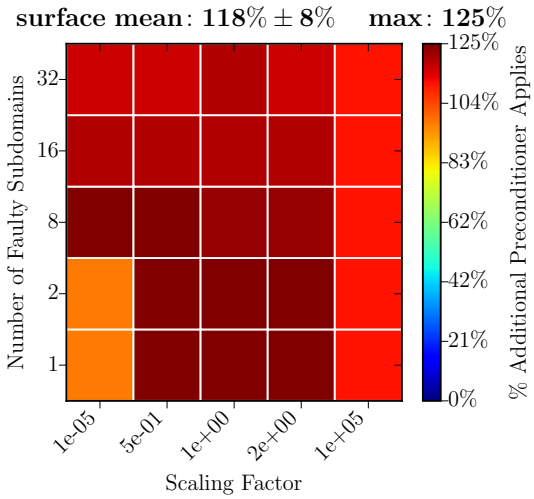
5.3.4 Expected Overhead Comparison

We compute the expected overhead across all experiments, i.e., compute the expected value for each row of these graphs. It becomes clear by inspection that our approach captures a worst-case scenario well beyond that of random bit flipping. That is, we are ensuring our algorithms can tolerate "bad" undetectable errors — error cases that random bit flips fail to expose — since we lack knowledge of exactly which values are the most sensitive. If an algorithm can handle our fault model, it can certainly handle the errors introduced by random bit flips.

We now compute the expected overhead given all samples for a given number of faulty subdomains. This computes the expected value for a "row" of the prior figures. For our numerical model, this entails grouping all scaling factors together, while for the bit flip model this considers an equally likely chance of flipping any of the 64 bits in the IEEE-754 representation.



(a) Overhead when utilizing detection with a random bit flip model.



(b) Overhead when utilizing detection with a numerical fault model.

Figure 5.2: Overhead comparison with fault detection on, for (a) a random bit flip injection and (b) a numerical fault model.

Table 5.1 summarizes the expectation across all experiments for a given number of faulty subdomains when no reactive fault tolerance is used. This corresponds to Figures 5.1a and 5.1b. Clearly, our numerical faults are much worse than bit flips. Our model tends to create roughly **25** additional preconditioner applies, because our inner solver iterates 25 iterations (max) per inner solve. Our faults are sufficient to require an entire inner solve. We then obtain our solution in the next inner iteration, requiring roughly the failure free number of iterations (6). This gives a total iteration (and preconditioner apply) count of approximately 25+6. Our model has significantly smaller variance than what random bit flips would have introduced. This indicates that our model *consistently* introduces poor behavior, which is our intent.

Table 5.2 analyzes the overhead if we check explicit residuals and projection lengths [29] inside the inner solver (Fig. 5.2a). Again, we show that random bit flips present very optimistic overheads. The reason for this was rigorously addressed by Elliott et al. [30]. Even with fault detection enabled, our fault model is still sufficient to show high overhead. This is desired. The faults we introduce are not necessarily detectable immediately. This forces our solvers to iterate 2-3 iterations before finally reaching a divergent state that is detectable. These are precisely the events we wish to study — faults that are undetectable, yet cause the solvers to eventually reach an invalid state. This motivates the study of low-overhead detection mechanisms.

Table 5.1: Additional preconditioner applies given no fault detection; percent additional applies in parentheses.

	Additional Preconditioner Applies			
Faulty	Bit Flips		Numerical	
Subdomains	mean	StdDev	mean	StdDev
1	7.28 (121%)	10.94	24.73 (412%)	5.05
2	7.56 (126%)	11.08	24.93 (416%)	5.07
8	8.11 (135%)	11.35	26.40 (440%)	1.90
16	8.56 (143%)	12.08	26.43 (441%)	1.89
32	9.51 (158%)	13.38	26.93 (449%)	2.85

Table 5.2: Additional preconditioner applies **with reactive fault tolerance**; percent additional applies in parentheses.

	Additional Preconditioner Applies			
Faulty	Bit Flips		Numerical	
Subdomains	mean	StdDev	mean	StdDev
1	1.49 (25%)	2.54	7.00 (117%)	2.12
2	1.56 (26%)	2.48	7.00 (117%)	2.12
8	1.69 (28%)	2.35	7.27 (121%)	1.72
16	1.81 (30%)	2.74	7.07 (118%)	1.74
32	1.83 (30%)	2.54	6.97 (116%)	1.83

5.4 Conclusion

We have presented results based on a fault model that allows us to characterize the numerical errors introduced by faults, and have shown that this model encompasses the range of overhead that the random bit flip model can introduce. Our fault model does not aim to predict the actual behavior of SDC. Rather, it shows a case sufficiently "bad" for us to assess how our fault-tolerance strategies behave when presented with very damaging SDC.

Our approach is a very different way of assessing preconditioned iterative linear solvers given an uncertain fault model. Rather than focus on what specifically constitutes a fault, we force our solvers to work through numerically challenging events. We specifically tune our fault model to inject errors that are not necessarily detectable. Our errors live inside the solvers' valid norm bounds, and empirically we observe our errors may cause divergence in latter iterations rather than immediately.

We compare our model to that of random bit flips, showing that random bit flip injection is *not* likely to show worst-case overhead. We support this through a methodical injection of bit flips, and by computing statistics over all experiments, as well as per bit position. Furthermore, we show our approach produces very predictable variance, irrespective of the number of processes that are faulty.

Chapter 6

Selective Reliability and Preconditioned Iterative Linear Solvers

6.1 Introduction

As semiconductors in modern microprocessors get smaller, and computers get increasingly parallel and energy-constrained, we expect that hardware will misbehave more and more [35, 63, 16, 44]. "Misbehavior" includes both *hard* faults that stop the running program, and *soft* faults, in which computers *silently* corrupt values or arithmetic results, and the program runs to completion and gets the wrong answer without any indication. Both kinds of faults matter a lot at extreme scales of parallelism, but the same issues (increasing parallelism, decreasing semiconductor feature sizes, and ever tighter energy and power constraints) may show up in hardware at all scales.

Soft faults particularly worry both users and algorithm developers. First, they may be hard to detect, especially if the results "look sensible." Second, even if an algorithm can recover, they may increase its run time dramatically (see e.g., [29]). As an example of the latter, consider a transient fault in the solution vector of a Jacobi iteration, run on an M-matrix. Third, they may have many possible sources in hardware, which means that it is hard to predict how they might corrupt data or computations. The continued rapid evolution of computer hardware adds even more uncertainty.

We present the following contributions:

- We develop an algorithm-based fault tolerance approach for iterative linear solvers that enables the use of opaque preconditioners without algorithmic or code modifications.
- We demonstrate our approach for two different iterative solvers: the method of Conjugate Gradients (CG) and the Generalized Minimal Residual Method (GMRES). We evaluate each solver with a state-of-the-art algebraic multigrid (AMG) preconditioner.

- We explore the performance of two detection techniques in the context of single faults and compute the overhead of our approach with these detectors activated.
- We compare our numerical fault model to flipping specific bits in the output of the preconditioner and discuss the strengths and weaknesses of each fault model.
- We assess the impact of residual evaluation and propose a strategy that lowers solver overheads under faults by 100%.
- We compare directly against related work in the field and show our approach requires orders of magnitude less computation.

6.1.1 Fault-Tolerant *Preconditioned* Solvers

Many engineering and scientific applications solve large sparse linear systems using iterative algorithms. Almost all of these solves use *preconditioners*. They may take time to set up and will add to the time per iteration, but aim to save overall run time by reducing the total number of iterations. An important contribution of our work is that we treat preconditioners as *opaque*. That is, we do not need to modify the preconditioning algorithm or implementation. This matters because preconditioners are often orders of magnitude more complex than the iterative solvers that utilize them, in terms of both algorithms and lines of code. For example, one can express CG in a dozen lines of code, but the algebraic multigrid implementation MueLu [42] we use in this work includes over half a million lines of code from several Trilinos packages, as of the 12.0 Trilinos release, not counting third-party sparse direct factorization libraries for coarse-grid solves.

6.1.2 Related Work

Prior work in algorithm-based fault tolerance for linear algebra problems focused on introducing checksums or other error-correcting codes into the algorithm itself. One designs the algorithm to maintain a checksum relationship either while it is running, or after it completes [23, 22, 18, 83]. This requires in-depth knowledge of and modifications to the algorithms and data structures. Even if this could be done for preconditioner algorithms as complicated as algebraic multigrid, adding checksums to an implementation would be impractical and error-prone, given the sheer amount of code to modify. Such a code would need to be rewritten from the ground up to use checksums. Sao and Vuduc[79] proposed Self-Stabilizing Conjugate Gradient (SS-CG), which is a selective reliability approach based on performing reliable correction steps periodically in the Conjugate Gradient algorithm. Unfortunately, SS-CG does not allow preconditioning.

Rather than attempting to rewrite every method to use an encoding scheme, we advocate a selective reliability approach that focuses the fault tolerance effort on iterative solvers rather than preconditioners. The primary focus of this work is the use of preconditioners in linear

solvers, where the preconditioner may silently introduce faults. That is, we intentionally do not explore hardening the preconditioner, which has been explored by [17]. We intentionally choose a discretization of the Poisson equation and an algebraic multigrid (AMG) preconditioner to assess our approach. For many Poisson-like problems, AMG has provably optimal efficiency; it requires only a small constant number of solver iterations. As a result, even small inefficiencies due to faults will manifest as high *relative overhead*. That is, rather than compare our solver against an arbitrary problem, we choose an extremely effective base-line case. This exacerbates the overhead we incur for dealing with faults, and we show that even with such a challenging base-line we are able to obtain solutions at 2x cost (or less) even though we utilize triple modular redundancy.

We also assess a concept presented by Elliott et al. [29] showing that enforcing bounded error is effective for guarding iterative linear solvers against data corruption. We consider the "Skeptical Programming" approach and draw conclusions about the amalgamation of these approaches. The faults that motivate these fault tolerance approaches require scale to be observed. We conjecture that any approach has to consider the implications of both strong and weak scaling. Specifically, we evaluate a preconditioning strategy that runs at scale and keeps faults local, while also evaluating a preconditioning approach that may spread corruption to other processes as part of preconditioning. We also propose an enhancement to convergence testing that allows our solvers to cut overhead by 100%.

This chapter is organized as follows:

1. In §6.2, we introduce the preconditioned linear solvers we evaluate.
2. In §6.3, we describe the type of preconditioner we chose and discuss how it can propagate errors.
3. In §6.4, we describe our fault injection methodology, and explain how we characterize faults.
4. In §6.6, we present findings that show the average number of extra preconditioner calls, given various numbers of faults.
5. In §6.7, we compare our overheads to two competing approaches to soft error resilience.
6. In §6.8, we observe our approach when scaled.

6.2 Preconditioned Linear Solvers

We consider two solvers for two classes of problems. The Generalized Minimal Residual Method (GMRES) [78] can solve nonsymmetric problems. The Method of Conjugate Gradients (CG) [53] can only solve symmetric positive definite (SPD) linear systems, but is faster for doing so. CG is used in the NAS Parallel Benchmarks [8] and in Mantevo miniapps like HPCCG [50]. Both

iterative solvers are popular in numerical simulations, e.g., Sierra's low Mach fluid application [68].

Linear solvers often utilize preconditioners as a means to accelerate convergence. Specifically, preconditioning is a transformation that attempts to improve some aspect of the linear system. We consider preconditioning in two different ways. First, we consider an algorithm that applies the preconditioner initially and every iteration thereafter, but does not require a preconditioner application to compute a solution, Preconditioned Conjugate Gradients (CG), which is shown in Algorithm 6.2; Second, we consider an algorithm that does not apply the preconditioner initially, but requires a preconditioner application every iteration and a preconditioner application to compute a solution, Right-preconditioned GMRES, which is shown in Algorithm 6.1. GMRES applies the preconditioner in lines 5 and 14. Note that the solution update (Line 14) need not be calculated every iteration, and is often not computed until the solver exits. Algorithm 6.2

Algorithm 6.1 (Right-preconditioned) GMRES

Input: Linear system $Ax = b$ and initial guess x_0
Output: Approximate solution x_m for some $m \geq 0$

1: $r_0 := b - Ax_0$ ▷ Unpreconditioned initial residual vector
2: $\beta := \|r_0\|_2$
3: $q_1 := r_0/\beta$
4: **for** $j = 1, 2, \ldots$ until convergence **do**
5: Solve $Mz_j = q_j$ for z_j ▷ Apply preconditioner
6: $v_{j+1} := Az_j$ ▷ Apply the matrix A
7: **for** $i = 1, 2, \ldots, j$ **do** ▷ Orthogonalize
8: $h_{i,j} := q_i^* v_{j+1}$
9: $v_{j+1} := v_{j+1} - q_i h_{i,j}$
10: **end for**
11: $H(j+1, j) := \|v_{j+1}\|_2$
12: $q_{j+1} := v_{j+1}/H(j+1, j)$ ▷ New basis vector
13: $y_j := \arg\min_y \|H(1 : j+1, 1 : j)y - \beta e_1\|_2$
14: $x_j := x_0 + M^{-1}[q_1, q_2, \ldots, q_j]y_j$ ▷ Solution update
15: **end for**

presents preconditioned CG. The preconditioner applications occur on Lines 2 and 8. Note that preconditioned CG does not require a preconditioner application to compute its solution update.

Our results center around observing the number of extra preconditioner applications relative to solving the problem without SDC. That is, we observe the impact of $\tilde{z} = M^{-1}w$, where $\tilde{z}$ indicates the corrupted output of a preconditioner call. In § 6.3 we give more details on how we

Algorithm 6.2 Preconditioned CG

Input: Linear system $Ax = b$ and initial guess x_0
Output: Approximate solution x_m for some $m \geq 0$

1: $r_0 := b - Ax_0$ ▷ Unpreconditioned initial residual vector
2: Solve $Mz_0 = r_0$ for z_0 ▷ Apply preconditioner
3: $p_0 := z_0$
4: **for** $j = 1, 2, \ldots$ until convergence **do**
5: $\alpha_j := (r_j, z_j)/(Ap_j, p_j)$
6: $x_{j+1} := x_j + \alpha_j p_j$
7: $r_{j+1} := r_j - \alpha_j Ap_j$
8: Solve $Mz_{j+1} = r_{j+1}$ for z_{j+1} ▷ Apply preconditioner
9: $\beta_j := (r_{j+1}, z_{j+1})/(r_j, z_j)$
10: $p_{j+1} := z_{j+1} + \beta_j p_j$
11: **end for**

decompose problems across multiple processes, and in § 6.4 we explain what $\tilde{z}$ looks like given faults on some (or all) parallel processes.

6.2.1 Selective Reliability

Our fault-tolerance strategy rests on relating numerical methods that naturally correct errors to system-level fault tolerance. In particular, we assume a *selective reliability* or "sandboxing" programming model [47] that lets algorithm developers isolate faults to certain parts of the algorithm in a coarse-grained way. In our scheme, we enforce that the outer solver be reliable, while letting the inner solver and its preconditioner run in an unreliable mode. An underlying goal is to spend most of our computation time in cheap "unreliable" computations, while minimizing the time we spend in expensive triple modular redundancy outer solve computations.

Analytically, any faults that occur in the inner solver manifest as a "different preconditioner" to the outer solver. The outer solver is chosen to be Flexible GMRES [77], which can tolerate a preconditioner that changes between iterations. The choice of Flexible GMRES as the outer "wrapper" allows our solvers to absorb any corruption introduced by the inner solver.

In our scheme, the preconditioner apply in FGmres is implemented as calling the inner solver (Algorithm 6.1 or 6.2), which will call its own preconditioner (MueLu). We provide more detail on how these approaches are implemented in Figure 6.1 and (later) in Figure 6.6. In this chapter, we express our solvers as `OuterSolver->InnerSolver-> Preconditioner`. For example `FGmres->Gmres` captures the outer solver `FGmres` and the inner solver `Gmres`. Should the inner solver use a preconditioner, e.g., `MueLu`, we then write `FGmres-> Gmres->MueLu`.

6.2.2 Implementation

We implemented our solvers using the Tpetra [9] sparse linear algebra package in the Trilinos framework [52] and validated them against both MATLAB and the solvers in Trilinos' Belos package [11]. Implementing our solvers using Trilinos lets us benefit from the scalability and performance of its sparse matrices and dense vectors. In addition, basing our research on Trilinos also gives us access to a wealth of numerical algorithms, on which we elaborate in § 6.3. We present a flowchart of a nested solver in Figure 6.1.

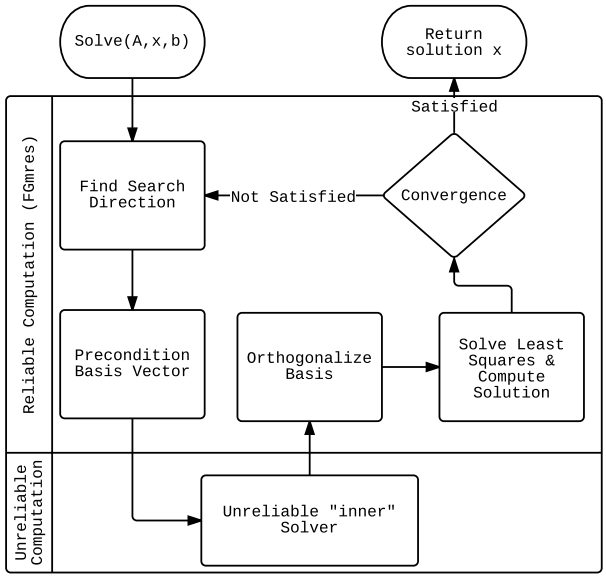


Figure 6.1: Flowchart for a nested solver implementation.

6.3 Preconditioners

This work explores how solvers behave in the presence of faulty preconditioners. Given that our solvers are parallel, we must consider parallel preconditioners. We choose a depth, rather than breadth study, and focus on a single widely used preconditioning strategy that is particularly effective for strongly elliptic problems, such as the Poisson equation. Poisson-like equations arise in many areas, such as the pressure term in the Navier-Stokes equations (fluid flow).

6.3.1 Algebraic Multigrid

Algebraic Multigrid (AMG) is a robust multilevel preconditioner. While geometric Multigrid requires knowledge of a grid, AMG operates directly on the matrix. In a setup phase, restriction

operators are defined that "coarsen" the matrix, creating consecutively smaller matrices. Likewise, prolongation operators are determined that interpolate the coarse level information back to finer levels of the multigrid hierarchy. Coarsening from the finest level to coarsest and back is referred to as a V-cycle. Prior to prolongation, AMG applies a smoother to the current level. Smoothers are often cheap solvers, e.g., a single sweep of Jacobi or Gauss-Seidel. We choose a single Gauss-Seidel sweep at all but the coarsest level, and we use SuperLU [25] as the solver for our coarsest matrix.

6.3.2 Hierarchy and Corruption

Due to AMG's hierarchical structure, a fault in a multigrid method may propagate from the process where the fault occurs to other processes. Should an SDC occur at the coarsest level, it is possible that half (or all) of the nodes absorb some amount of corruption into their final solutions. Alternatively, if SDC occurs in one process' data at the finest level, the error will remain local if no further V-cycles are performed. (In our setup, with MueLu as a preconditioner, we enforce only one V-cycle.)

6.4 Fault Model and Injection Methodology

This work is motivated by the premise that soft errors will become more likely in future systems, and that SDC has been observed in current systems. Much uncertainty remains about how often soft errors will occur and how they will manifest in applications. Most prior work modeled soft errors as one or more flipped bits, e.g., [86]. Researchers injected bit flips and observed their effects on running applications [82, 22, 79]. These works showed that current algorithms *can* misbehave badly if their data are corrupted. In the presence of SDC, iterative methods can return the wrong answer, fail to converge and iterate forever, or require more iterations to obtain a solution.

For this reason, we intentionally use a fault model that is abstract and brings out the worst behavior in iterative solvers. We use a numerical fault model [31], that has been shown to consistently trigger more severe faults. This results in poor performance with substantially less variance than a random bit flip model.

6.4.1 Corrupting Preconditioner Outputs

We denote the vector output of a preconditioner as z (see, e.g., Line 8 of Algorithm 6.2). In numerical analysis, vectors are described using norms. GMRES minimizes the residual error with respect to the L-2 norm, and CG minimizes the A-norm of the solution error, which is also an L-2 norm of a different vector. For this reason, we choose to characterize errors with respect

to the L-2 norm. This leads us to two classifications of errors: Those that change the L-2 norm of the output, and those that preserve its L-2 norm.

Following the approach of [31], a fault constitutes a *faulty subdomain* permuting its portion of the global vector. *Permutations* preserve the L-2 norm. We also consider the case that this bad error changes the L-2 norm. To *change* the L-2 norm, a faulty subdomain may also *scale* its portion of the vector by some constant (faulty domains *always* permute). A single permuted subdomain's vector is sufficient to cause the solver to stagnate, never producing a solution that meets our convergence criteria.

6.4.2 Granularity of Faults

We let subdomains (one subdomain per MPI process) return completely corrupt solutions. That is, we consider faults at the MPI process level rather than single values. The intent of such a pessimistic model is to capture poor behavior, e.g., a black box library may perform operations that affect more than one entry of its output vector, and possibly even more than one process. For example, an incorrect pivot in a sparse factorization for AMG's coarse-grid solve may cause incorrect values on *all* processes.

6.4.3 SDC and Solvers

In our preliminary work, GMRES was chosen as the inner solver because it is commonly accepted as "more robust" than CG. Given that CG can only solve SPD linear systems, if a fault occurs in CG, the error can cause problems by appearing to be nonsymmetric [69], and CG can behave very poorly. It is for these reasons that CG is not considered as an outer solver. SDC may also change the sign of key values, e.g., a negative projection length.

6.5 Experiment Description

6.5.1 Methodology

We described in § 6.4 how we corrupt the preconditioner's output. To evaluate the impact of our preconditioned solvers in the presence of catastrophic SDC, we perform the following steps:

1. Solve the problem without introducing SDC, and compute the number of times, K, the preconditioner was applied.
2. For all j in $[1, K]$, reattempt the solve, introducing SDC at the j-th preconditioner application. This results in K total solves.

3. For all K solves with SDC, compute the relative percent of *additional* preconditioner applies over the SDC-free solve, e.g., $\frac{Applies_{observed} - Applies_{FailureFree}}{Applies_{FailureFree}} \times 100$. If $observed - FailureFree < 0$, i.e., SDC accelerated convergence, we record zero overhead.
4. Repeat Steps 2 and 3, letting various numbers of MPI processes participate in the SDC injection.
5. Repeat Steps 2-4, varying the scaling factor applied to the SDC.
6. For each combination of scaling factor and number of faulty processes, plot the *average* number of additional preconditioner applies as a percentage. 0% means no additional applies; 100% means twice as many.

6.5.2 Problem Specification

Our Poisson problem is generated from a 3D finite element discretization. Since our matrix is sparse, a common measure of sparsity is the average number of nonzeros (nnz) per row (denoted $\overline{nnz}_{\mathrm{r}}$). We also report the total number of nonzeros in Table 6.1. Using MueLu's default partitioning (and repartitioning), we obtain a 4 level hierarchy, which is configured to perform one V-cycle per application.

Table 6.1: Problem specification

Global Size	Global nnz	Local Size	$\overline{nnz}_r$	Num. Proc.
3,263,696	86,936,980	101,990	27	32

6.5.3 Baseline and Preconditioner Effectiveness

In the following sections, we present relative overheads. The choice of what to compare against is paramount. If we choose to compare against an inefficient solver, then it is trivial for our technique to excel. Instead, we compare against a very good solver. We chose for our baseline `FGmres->Gmres->MueLu`, with the inner solver configured to perform only the work necessary to obtain a solution, that is without fault tolerance overhead. We choose to compare against a nested solver with a reliable outer shell, because our assumption is that in the future, applications will need to use our approach all the time. A reasonable question is why not `FGmres->Cg->MueLu`. To address this, we breakdown our failure-free runs in Table 6.2. We show the maximum work performed by a single process as well as the total aggregate work performed by all processes. Recognize that CG costs more than GMRES in terms of computation. We are not considering storage overhead in this work (CG would have a slight advantage).

The key is that efficient preconditioners diminish the differences between CG and GMRES. In this case, GMRES converged in 7 iterations, while CG required 9. GMRES's work increases linearly, and so as the iteration count increases, GMRES quickly outpaces the work performed in a CG iteration. To obtain the total number of floating-point operations, multiply the local total in Table 6.2 by 32. In our graphs, we use the slightly smaller correct count (obtained through our instrumented code). This accounts for the slight load imbalance that occurs when MueLu repartitions the work at each level of the hierarchy. One may also use this table to determine the work needed to solve this problem without a nested solver, i.e., the "Sub Total." As an "always-on" overhead, our cost is low. We observe that the inner preconditioner (MueLu) is the dominant cost, not the solver.

6.5.4 Solver Configuration

Our solvers have two key parameters: The maximum number of inner iterations (25), and the stopping criteria for the relative residual norm 1×10^{-8}. Our outer solver uses triple modular redundancy, and so the cost of an outer iteration is equal to 3×FGmres(m), where m is the current outer iteration number. Note, the inner solver never restarts, nesting solvers is often considered a *smarter* way to imlpement restarted linear solvers [84].

6.6 Experiments

We now systematically evaluate our solvers, incrementally adding features. There are two main challenges:

1. Fault detection using the explicit residual can nearly double the cost of a GMRES iteration.
2. Many faults make the inner solver **stagnate**, so that it iterates without making progress.

The second is particularly damaging because the *implicit* residual, the residual obtained in GMRES as the residual of the small upper Hessenberg least-squares problem, may decrease to the desired level, but the *explicit* (true) residual $\|Ax - b\|$ may not. If we use the strict convergence criteria $\|Ax_k - b\|/\|b\|$, we may incur high overhead trying to converge, while the maximal attainable accuracy has been damaged enough to cause the explicit residual to cease making progress. This causes our inner solve to iterate to its maximum number of iterations, while making little progress. We will explore this in detail below.

6.6.1 Figure Guide

We present overheads in a unique way. Each square of a figure represents the average overhead we observed for that specific fault scenario, when injecting a fault at every possible iteration and computing the mean overhead. The y-axis indicates the number of subdomains that participate

Table 6.2: **Maximum** floating-point operation count on a single process for **failure-free** nested CG and nested GMRES solvers, as well as total global floating-point operations.

	Operation	#	Model	Single Op. Cost	Total	Percent Work
FGmres->Gmres->MueLu	**Iterations**	**7**				
	SpMV†	8	2 nnz	5.51e+6	4.41e+7	17.3%
	PCApply‡	8	-	1.97e+7	1.57e+8	61.9%
	Dot	28	2n	2.04e+5	5.71e+6	2.2%
	Update	36	2n	2.04e+5	7.34e+6	2.9%
	Norm	8	2n	2.04e+5	1.63e+6	0.6%
	Scale	8	n	1.02e+5	8.16e+5	0.3%
	Sub Total				2.17e+8	85.3%
	Outer Iter.	1	-	-	3.73e+7	14.7%
	Local Total				2.54e+8	100%
	Global Total				**8.14e+9**	
FGmres->Cg->MueLu	**Iterations**	**9**				
	SpMV†	10	2 nnz	5.51e+6	5.51e+7	18.5%
	PCApply‡	10	-	1.97e+7	1.97e+8	66.0%
	Dot	19	2n	2.04e+5	3.88e+6	1.3%
	Update	26	2n	2.04e+5	5.30e+6	1.8%
	Norm	0	2n	2.04e+5	0.0	0.0%
	Scale	0	n	1.02e+5	0.0	0.0%
	Sub Total				2.61e+8	87.5%
	Outer Iter.	1	-	-	3.73e+7	12.5%
	Total				2.98e+8	100%
	Global Total				**9.55e+9**	

† nnz = Local $nnz(A)$; ‡ see Gahvari et al. [41]

in the fault. The x-axis indicates a scaling factor that is applied to the permuted output of the preconditioner. The choice of scaling factors is not arbitrary. The fault model we use is designed to create errors that preserve the L-2 norm, as well as those that increase or decrease it. In prior work, we developed a cheap detector for "large" perturbations to the norm. All but five (the right-most column) of faults are likely to be undetectable by the norm bound [29].

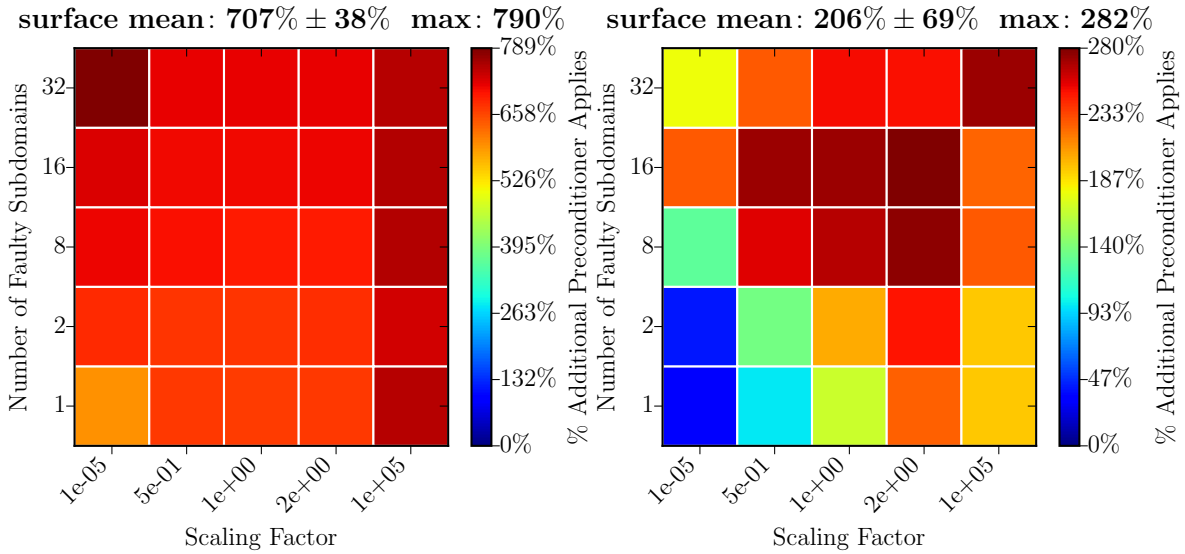
6.6.2 Overhead with No Detection

First, consider how our scheme behaves if we do nothing and allow the inner solver to return detectably incorrect results to the outer solver, i.e., allowing large errors to corrupt the inner solver's solution. This is dangerous, as the Flexible GMRES (the outer solver) can easily break if the inner solver fluctuates too much. We omit a graph showing this behavior, but the overhead is easy to compute: 1) the outer solver will never converge, and 2) it will iterate until it reaches whatever limit is imposed on it. This can lead to very significant overhead. Figure 6.2a shows how FT-GMRES behaves given our pessimistic faults. These faults are sufficient to stagnate a standard solver, i.e., we observe that the inner solver becomes divergent or stagnant. This "wastes" 25 inner iterations and returns detectably incorrect results to the outer solver. The outer solver absorbs this result as a basis for the solution. These highly inaccurate inner solves corrupt the state of the outer solver, leading to 3-4 outer iterations of failure-free execution before we are able to obtain a solution. If we increase the scaling factor, e.g., 1×10^{100}, we can break the outer solver, forcing it into divergent behavior.

We intentionally allow the color bars to differ. This is because CG performs significantly better than GMRES. Note that we are not changing the sign with our scaling factor. This breaks CG in a single iteration, but this is easy to detect. We think this drastically different behavior is due to GMRES preserving state (having "memory") while CG is mostly stateless. A fault in GMRES is embedded into the subspace that is built, whereas CG is effectively a 3-term recurrence. This gives CG as an inner solver a chance to recover without wasting an entire inner solve. GMRES appears to have difficulty recovering from corruption to its basis.

6.6.3 Residual and Projection Lengths

Having shown how "bad" things can be, we evaluate two detectors. These detectors only work inside GMRES. The first, the explicit residual, exploits GMRES' promise of a monotonic non-increasing residual. The second is the projection length bound, which tests the values on Line 8 of Algorithm 6.1. Using Figure 6.2a, we repeat the experiment without detectors enabled in Figure 6.3. We have fixed the color bar range to that of CG's overhead. This allows an easy comparison between GMRES and CG as inner solvers. We hatch the cells based on whether a detector indicated an incorrect value.



(a) `FGmres->Gmres->MueLu` (b) `FGmres->Cg->MueLu`

Figure 6.2: Overhead comparison of (a) `FGmres->Gmres->MueLu` and (b) `FGmres->Cg->MueLu`, given no attempt to directly detect or cope with errors.

It is expected that the projection length detector will not be able to detect any corruption that is small, which corresponds to the left half of the plot. This does not imply that the small errors cannot lead to large detectable errors. E.g., consider the top most row, where many subdomains return corrupted values.

The explicit residual (hatched '\' in Figure 6.3) is an excellent detector, and can detect the majority of errors. Recognize the explicit residual detected no errors in the right-most column. This is because the norm bound detected all errors before the residual test was performed. The experiments (squares) that benefited from norm bound detections show lower overhead than those with explicit residual detections. This is because the norm bound enables the solver to exit without wasting effort to compute a solution update that will likely be wrong. The overhead is still quite high, with an average overhead of approximately 200%, which means the solve costs 3x as much as the original failure free one in Table 6.2.

6.6.4 Tuned Residual Checks

Testing the explicit residual every iteration lowered overhead compared to doing nothing, but the test is too expensive. Specifically, the solution update nearly doubles the cost of a GMRES iteration. This is because the solution update requires $k+1$ vector updates to compute the linear combination of basis vectors weighted by the least squares solution. The solution update also requires a preconditioner application. Given that orthogonalization requires k updates and

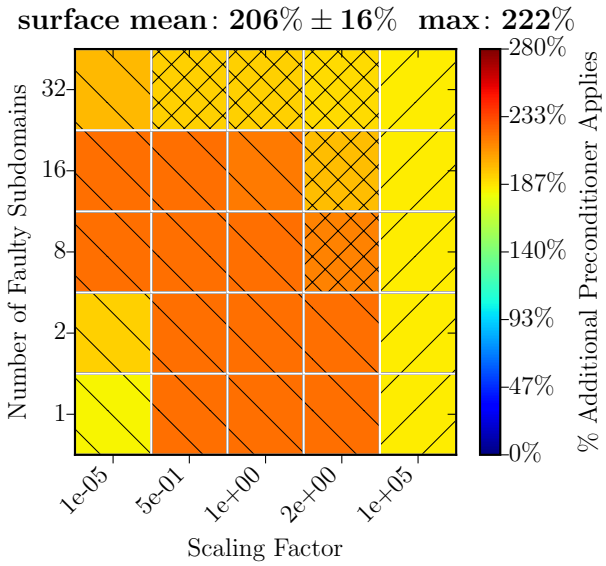
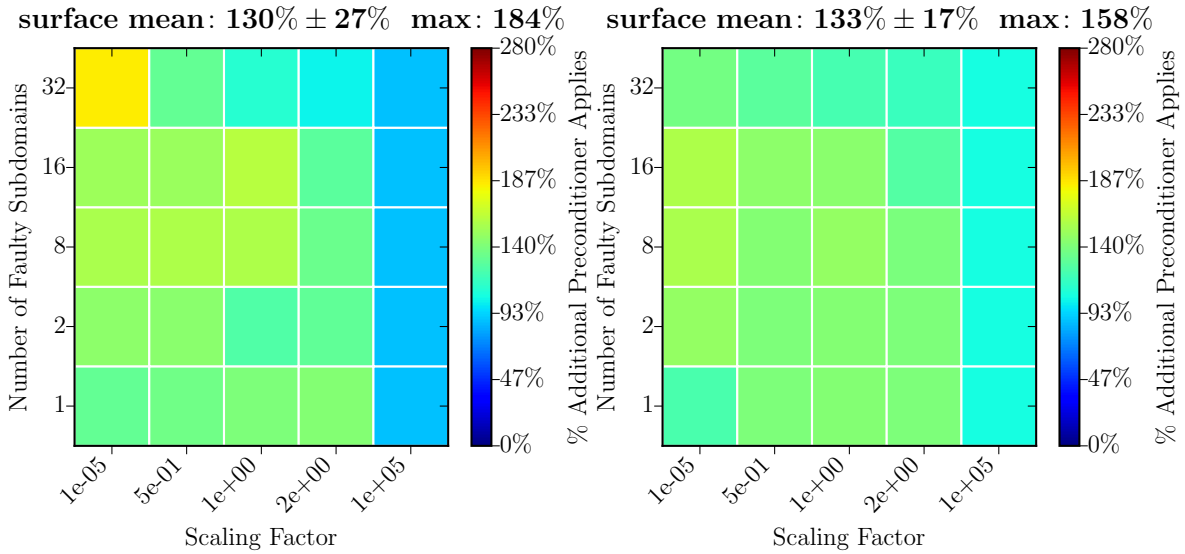


Figure 6.3: Overhead when testing the explicit residual every iteration and using a projection length bound in `FGmres->Gmres->MueLu`. Explicit residual detections are hatched left \ and norm bound detections are hatched right /.

k dot products, the solution update is prohibitively expensive, but we must have a solution to enforce monotonicity (we rarely see monotonicity violations from the implicit residual).

Now, we test the explicit residual at different frequencies: 1) we test every 3 iterations, and 2) we test only once. **In all cases, we always test the explicit residual if we believe we have converged, due to the implicit residual meeting our convergence tolerance.** Figure 6.4 shows two overhead plots, Figure 6.4a evaluates the explicit residual only once, unless the implicit residual converges. In the case the implicit residual converges, a solution must be computed so the solver can test the explicit residual to determine if the solver has really converged. Recognize the right-most column now shows overhead equivalent to 2x (100%) of the original failure-free run. This is possible because the detector is extremely cheap, and we have removed some of the expensive solution updates. Recall that our outer solver is operating in triple modular redundancy, yet we are now seeing costs proportional to 2x. Unfortunately, we still incur near 200% overhead in a substantial number of experiments. Figure 6.4b attempts to find a middle ground between testing every iteration and testing only once. The latter shows a slight improvement overall, but increases the overhead in the right-most column, where the projection bound is very effective.



(a) Explicit residual evaluated once.

(b) Explicit residual evaluated every 3 iterations.

Figure 6.4: Overhead comparison given two different frequencies of explicit residual evaluation in `FGmres->Gmres->MueLu`.

6.6.5 Strict Convergence Checks

Previously, we found that testing only once or thrice offered lower overhead than testing every iteration, but we still observe high overheads. We identified the cause of this overhead to be our convergence test. We know the implicit residual is highly inaccurate in a faulty environment, which is why we always rely on the explicit residual to determine if the inner solver should return. The implicit residual easily becomes inaccurate because it represents the residual of the problem being solved, not necessarily the problem we want solved. This is because faults corrupt the basis, and then taint all subsequent basis vectors (and the upper Hessenberg matrix). What tends to happen is, the implicit residual converges, but the explicit residual stagnates. This is because our faults damage the maximal attainable accuracy.

Our "strict" convergence test is shown in Algorithm 6.3. We modified our inner solver convergence test so that we avoid repeated solution updates if the explicit residual is stagnant, while the implicit residual is convergent. Ideally, if we had a condition number estimate, we could test the gap between the explicit and implicit. This is not practical since obtaining a condition estimate requires work proportional to a linear solve. Instead, we exploit the knowledge that we have nested solvers. That is, we would rather make some progress if the implicit residual is converged, and we would prefer not to incur the high cost of computing a solution update often.

We modify our inner solver's convergence test to Algorithm 6.4. This effectively lets the inner solver "believe the lie" the implicit residual tells, but only if the implicit residual is not

Algorithm 6.3 Strict Inner Solver Convergence Test

Input: implicit residual
Output: boolean: converged

```
1: if implicit residual is less than some tolerance then
2:     Compute solution
3:     if explicit residual is less than some tolerance then
4:         return converged=True
5:     else if explicit residual > prior explicit residual then
6:         return last valid solution or RHS, exit inner solve
7:     else
8:         return converged=False
9:     end if
10: end if
```

lying too much. We still get the benefits of enforcing monotonicity, which has a side effect of ensuring the basis vectors returned to the outer solver are reasonable. In the worst case, we return the right-hand side, which allows the inner solver to be "rejected" and the outer solver makes progress as if no preconditioner was present (i.e., the inner solver becomes the identity matrix!)

Algorithm 6.4 Relaxed Inner Solver Convergence Test

Input: implicit residual
Output: boolean: converged

```
1: if implicit residual is less than some tolerance then
2:     Compute solution
3:     if explicit residual > prior explicit residual then
4:         return last valid solution or RHS, exit inner solve
5:     end if
6:     return converged=True
7: end if
```

We now repeat the same experiment using our detectors plus relaxed inner solver convergence. Figure 6.5 shows a marked improvement over Figure 6.2a. The techniques we use are not problem specific either, and focus instead on avoiding expensive unreliable operations. To help summarize how these fault tolerance techniques come together, we illustrate how our nested approach works in Figure 6.6.

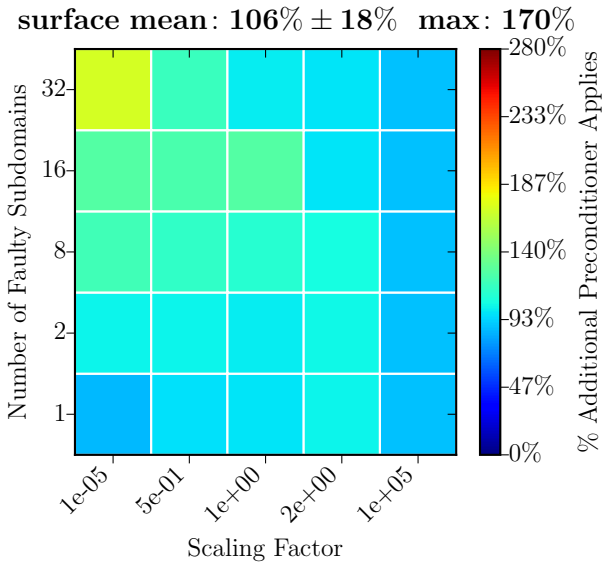


Figure 6.5: Overhead when using a relaxed inner solver convergence test as well as a projection length bound in `FGmres->Gmres->MueLu`.

6.6.6 Inexact Krylov

Inexact Krylov theory [85] rigorously addresses how accurate a Krylov method needs to be. Practically, this means the convergence criteria can be allowed to vary based on monitoring the residual norm. Nested solvers in particular can benefit from inexact tuning, since the inner solver's convergence criteria can be relaxed as the outer solver makes progress. It has been difficult to see a benefit from inexact Krylov in prior studies, because our inner solver's convergence test was too strict. To illustrate the savings that inexact Krylov gives us, we have repeated the experiment with inexact tuning turned off in Figure 6.7. We see a noticeable increase in overhead, particularly for experiments that made faults small (left-most column). This means that those particular experiments did not require as many iterations on the next inner solve as our fixed convergence criteria required. In other words, the outer solver did not need the inner solver to work as hard the next time it was invoked. This is an encouraging finding, as detecting or coping with small errors has been difficult.

6.7 Comparisons

We now analyze our final overheads and compare these against our nested Conjugate Gradients solver. In a sense, CG already has a "relaxed" convergence test, as its implicit residual is computed as part of the algorithm and is always used for convergence testing. Table 6.3 compares the highest overhead observed from Figures 6.5 and 6.2b. Even though GMRES is

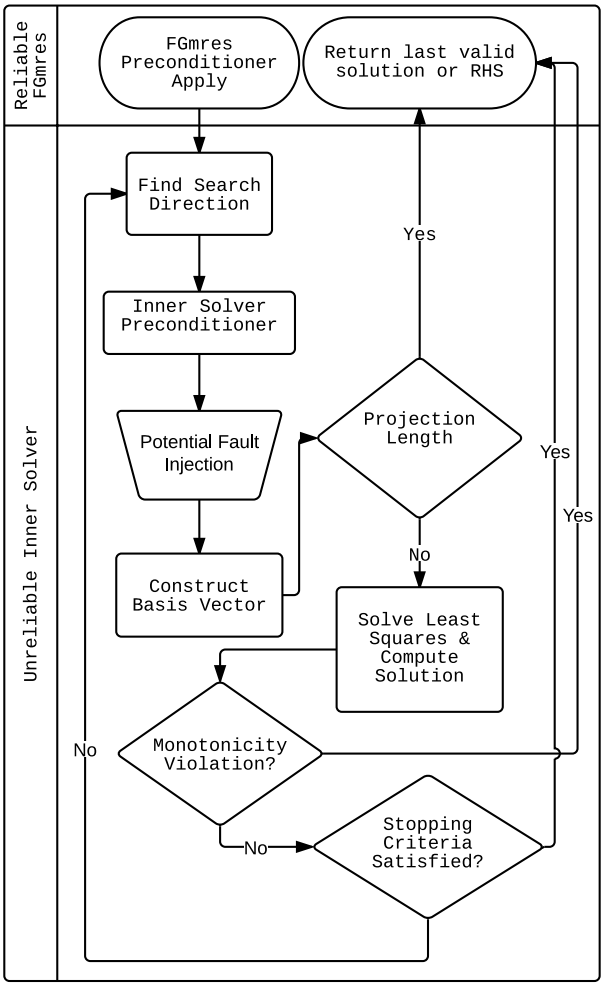


Figure 6.6: Flowchart for responding to detectors in a nested solver.

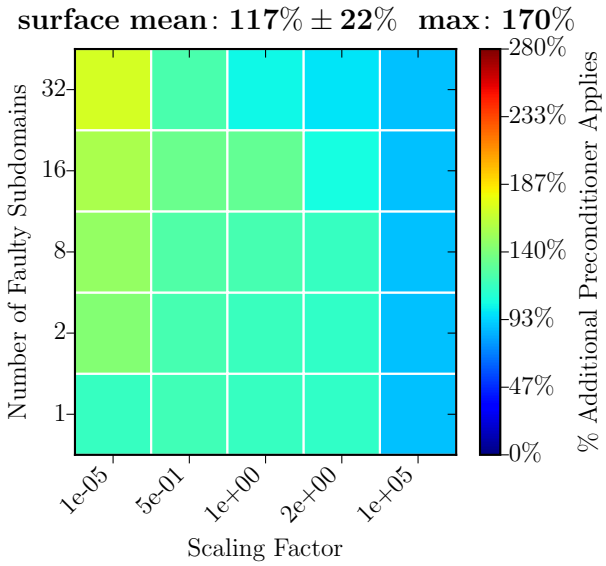


Figure 6.7: Overhead when using a relaxed inner solver convergence test as well as a projection length bound in `FGmres->Gmres->MueLu`, but no inexact tuning of the inner GMRES convergence rate.

typically considered an "expensive" solver, when coupled with a good preconditioner it beats CG as an inner solver. While GMRES does require more work per iteration, it also gives several strong benefits: Monotonicity of the residual; Not sensitive to sign errors.

6.7.1 ABFT Cholesky Comparison

Since we have presented results given an extremely pessimistic fault model, our overheads appear extremely high. This is because we have chosen to compare our work against a very high bar, that being a state-of-the-art preconditioner. Our final solver configuration results in relatively low overhead. We get the right answer in roughly twice the work of today's solvers, and that is with an extremely difficult fault model. Given no faults, we incur a very low "always-on" cost. Our low always-on cost is key, as we should not expect devastating errors to occur often, we simply wish to be prepared should they manifest.

We select the cheapest checksum type solver for solving an SPD problem, a Cholesky factorization. This is cheaper than a QR or LU factorization, and Wu and Chen have shown a checksum ABFT variant [94]. We now compare our worst-case operation count (Table 6.3) to such an approach. Our Poisson matrix's characteristics are specified in Table 6.1. To determine the cost of a dense ABFT Cholesky factorization ($A = LL^T$) we use [94, Table 2]. The cost for

Table 6.3: Maximum operation count for a subdomain given worst-case performance in faulty experiments.

	Operation	Inner Iter.		Single Op. Cost	Total	Percent of Work
		1	2			
FGmres->Gmres->MueLu	**Iterations**	**9**	**11**			
	SpMV	9	11	5.51e+6	1.21e+8	17.2%
	PCApply	10	12	1.97e+7	4.33e+8	61.5%
	Dot	45	66	2.04e+5	2.26e+7	3.2%
	Update	55	78	2.04e+5	2.71e+7	3.9%
	Norm	10	12	2.04e+5	4.49e+6	0.6%
	Scale	10	12	1.02e+5	2.24e+6	0.3%
	Sub Total				(6.11e+8)	(86.7%)
	Outer Iterations				9.39e+7	13.3%
	Local Total				7.05e+8	100%
	Global Total				**2.25e+10**	
FGmres->Cg->MueLu	**Iterations**	**25**	**5**			
	SpMV	25	5	5.51e+6	1.65e+8	19%
	PCApply	25	5	1.97e+7	5.90e+8	67%
	Dot	50	10	2.04e+5	1.22e+7	1%
	Update	75	15	2.04e+5	1.84e+7	2%
	Sub Total				(7.86e+8)	(89%)
	Outer Iterations				9.39e+7	11%
	Local Total				8.80e+8	100%
	Global Total				**2.82e+10**	

the factorization is given by

$$Cost_{\text{factorization}} = N^3/3 = O(10^{19}). \tag{6.1}$$

Once factored (yielding L), the solution for a given right-hand side is obtained by performing two triangular solves, once with L^T, and again with L. These triangular solves require approximately

$$Cost_{\text{triangle solves}} = 2 \times O((N+1)N/2) = 2 \times O(10^{13}). \tag{6.2}$$

This comparison does not include the overhead introduced by the checksum operations: the cost to maintain the checksums, error location, error correction. We feel it should be clear that $O(10^{10}) \ll O(10^{19})$, e.g., see Table 6.3. Practically, this shows why sparse approaches are used when possible. This also motivates the study of resilient sparse direct approaches.

We did not account for the setup cost of MueLu or the cost of computing our projection length detector. The latter requires computing a matrix norm, which is $O(nnz)$. MueLu's setup is not free, but, e.g., setup is a small amount of work relative to a solve [68]. Even if we doubled our operation counts we are still well below those of [94].

6.7.2 Self-Stabilizing CG

Another option to compare against is Sao and Vuduc's Self-Stabilizing CG [79]. Unfortunately, SS-CG does not allow preconditioning. Table 6.4 reports the operation counts required to solve our Poisson matrix using SS-CG **with no faults injected**. Clearly, injecting corruption would make these counts worse. Our solver required substantially less work, thanks to our inner preconditioner. Our cost, with faults, is significantly lower than that of SS-CG with no faults, $O(10^{10})$ versus $O(10^{11})$. This highlights why we seek to enable *preconditioned* sparse iterative methods. This does not degrade the usefulness of SS-CG, but when compared against a solver coupled with a good preconditioner, it becomes very difficult to stay competitive. We also considered modifying SS-CG to allow preconditioning. The modifications effectively changed the algorithm to that of Nonlinear CG, the details of this are beyond the scope of this work.

6.7.3 Preconditioner Applies and Floating-Point Operations

We report cost by the additional number of preconditioner applications as a percent relative to a fault-free run, with no fault tolerance overhead. A reasonable question is how applies relate to floating-point operations (FLOPs). We present FLOP *counts*, not *rates*. That is, we have not measured the relative performance of a machine, which would yield floating-point operations *per second*. Instead, we have measured the work performed invariant of the machine on which we run. Table 6.3 shows that our inner solver's preconditioner, MueLu, accounts for over 60%

Table 6.4: Operation count for solving the Poisson problem using self-stabilizing CG. 276 total iterations: 221 unreliable and 55 reliable.

	Operation	#	Single Op. Cost	Total	Percent Work
Self-Stabilizing-CG	**Iterations**	**221**			
	SpMV	221	5.51e+6	1.22e+9	34%
	Dot	443	2.04e+5	9.04e+7	3%
	Update	663	2.04e+5	1.35e+8	4%
	Sub Total			1.44e+9	41%
	Reliable	**55**			
	SpMV	110	5.51e+6	1.82e+9	51%
	Dot	220	2.04e+5	1.35e+8	4%
	Update	220	2.04e+5	1.35e+8	4%
	Local Total			3.53e+9	100%
	Global Total			**1.13e+11**	

of the work. It follows that the dominant cost will characterize the work performed, which we show in Figure 6.8. Compared to Figure 6.5, the relative overheads are identical. This measure would not be accurate if our solvers required a varying number of outer iterations based on a fault. Our solvers with faults required 2 outer iterations always, hence the the outer solvers cost is a constant (see Table 6.3's Outer Iteration cost).

6.7.4 Relation to Bit Flips

We designed our fault model to make iterative solvers perform poorly. Instead of speculating on what a fault is, we focus on the *effect* a fault can have, which is silent corruption of algorithms' state. We now briefly compare our findings to that of bit flip fault injection in the IEEE representation of a number. We introduce faults at the same location as our model, the output of the preconditioner. Since we considered corruption that "wrecked" some number of subdomains, we allow bit flips to corrupt the entire output vector of the preconditioner, i.e., 101,990 bit flips per faulty subdomain. We systematically perform this corruption a single bit position at a time. This allows us to see how bit position impacts overhead. We have replotted Figure 6.5 allowing the overhead (color bar) to not exceed the maximum we observe ($\approx 160\%$) in Figure 6.9b. We also show the overhead for systematic bit flip injection in Figure 6.9a. We forced the bit flip color bar to have the same maximum as our numerical fault experiment. This is because the bit flip experiments generated a maximum overhead of 148%, and this only occurred once. In general, if bits had been flipped at random, the large swatch of dark blue (zero overhead) would

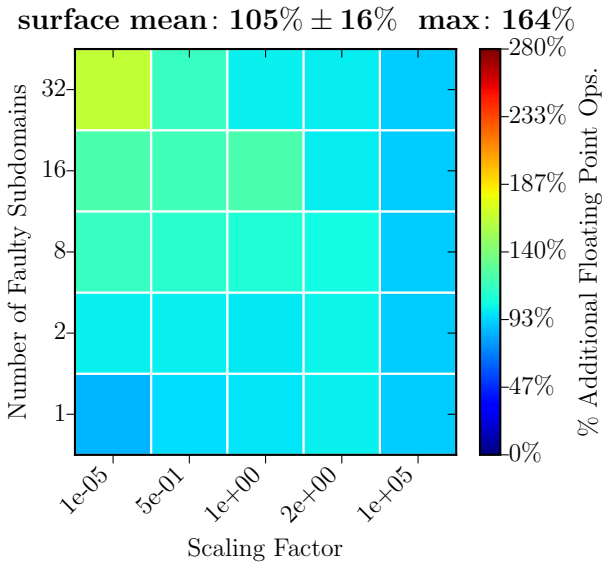


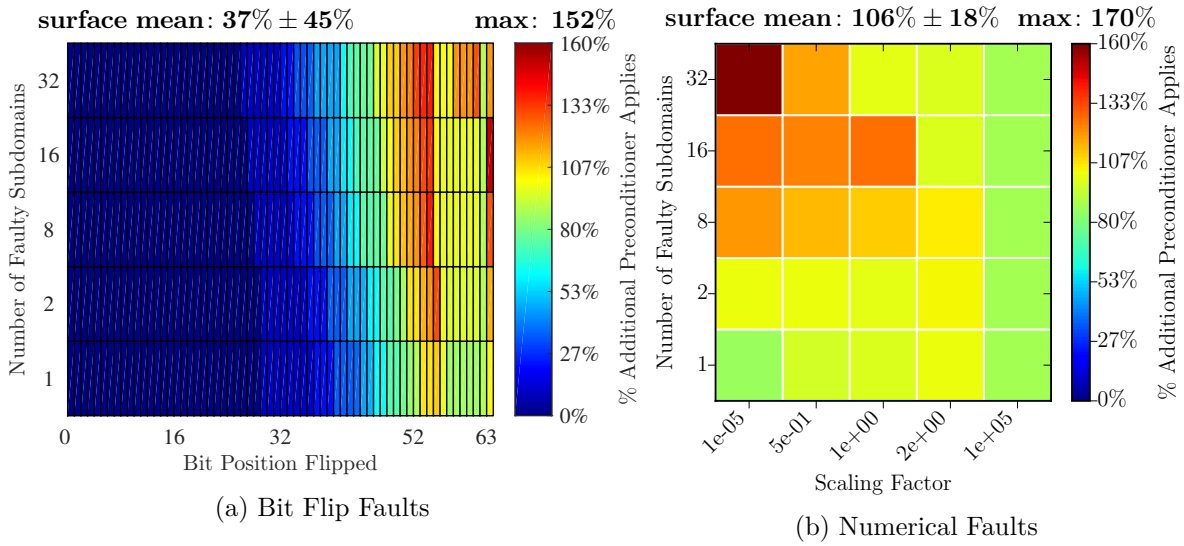
Figure 6.8: Global (aggregate) floating point overhead when using a relaxed inner solver convergence test as well as a projection length bound in `FGmres->Gmres->MueLu`. (see Figure 6.5)

dominate the expected value. Also, flipping only exponent bits would not suffice to cause our solvers to exhibit poor behavior [30]. This has two interpretations: 1) we may congratulate ourselves on running through millions of bit flips, or 2) (the correct interpretation) we can draw no conclusion from these results because the variance is significant.

This exacerbates the point of our numerical model: we are not trying to discover the "expected" manifestation of faults. It is possible that a single bit flip in a sensitive operation could cause high overhead. We do not know exactly where (or when) such an event can occur. But the result (poor behavior of the solver) will be the same outcome. We specifically focus on silent data corruption, i.e., the faults that do not crash the system, and therefore our numerical model is doing a much better job of showing us how our solvers behave when things go catastrophically wrong.

6.8 Scalability

In § 6.6 we used a fixed number of processes and increased the number of faulty subdomains. In a sense, we "strong scaled" our faults. The more subdomains that introduced corruption, the larger the global amount of corruption becomes. We now *weakly scale* our problem to 320 subdomains, and allow the same percentage of faulty subdomains. We introduced no new calculations to our solvers, and our detectors take place around collective operations, e.g., dot products use an Allreduce, which is how projection lengths are tested. Since we show overhead relative to



(a) Bit Flip Faults

(b) Numerical Faults

Figure 6.9: Overhead when using a relaxed inner solver convergence test as well as a projection length bound in `FGmres->Gmres->MueLu`: (a) Bit flips and (b) our pessimistic fault model.

a failure free run, the relative overhead naturally accounts for the increased difficulty of the problem solved. The global size of our matrix is now $32,258,556$ with $865,520,320$ nonzeros. In Figure 6.10, we observe approximately the same overhead. Weak scaling does slightly reduces the impact of faults. We suspect this may be caused by two factors: 1) the increased condition number of the linear system, and 2) the increased impact of numerical error (e.g., rounding error). Both factors are closely related, but are beyond the scope of this work. We leave further investigation of weak scaling to future work. Our hypothesis is true: Our approach "scales" in the sense that we see the same increase in overhead at large scales, given equal percentages of corruption.

6.9 Conclusion

This work demonstrates that iterative linear solvers can get the right answer despite incorrect arithmetic or storage in their preconditioners. They can do so without algorithmic or implementation changes to preconditioners by combining selective reliability and inner-outer iterations. Fault detection in inner solves need not catch all incorrect preconditioner results in order to reduce overhead much below just running the solver twice. This justifies even an expensive implementation of reliability in outer solves, since most of the work goes into inner solves with their more effective preconditioner. We have also shown that analytical approaches that detect and filter out large errors scale well and significantly reduce faults' overhead. This is particularly

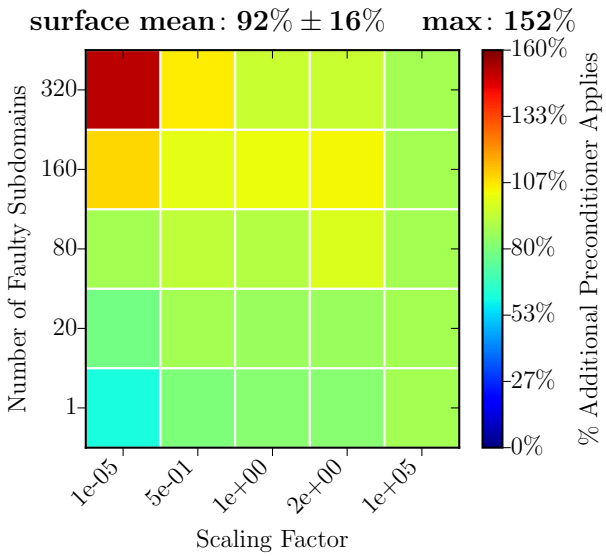


Figure 6.10: Overhead when using a relaxed inner solver convergence test as well as a projection length bound in `FGmres->Gmres->MueLu` on a *weakly scaled* version of the Poisson problem. (see Figure 6.5 or 6.9b for comparison.)

true for effective preconditioners like algebraic multigrid, that require only a few solver iterations. We have also shown that the overhead of detection is critical. While some detectors are extremely effective, the cost to evaluate the detector is proportional to the work it seeks to verify.

We have presented results based on a fault model that allows us to characterize the numerical errors introduced by faults, and have shown that this model encompasses the range of overhead that the bit flip model can introduce. Our fault model does not aim to predict actual behavior of future SDC. Rather, it shows how our fault tolerance strategies behave when presented with very damaging SDC. We also compared our approach against two recent algorithm based fault tolerance works. In both cases, our approach requires substantially less work. We have shown our approach preserves the efficiency that is desired for large sparse systems of equations.

Whether SDC turns out to be a real "monster in the closet" or not, our findings are relevant for other fields of research. We observe a consistent trend in our data: Faults that increase the 2-norm are worse than faults that maintain or decrease the 2-norm. We also see that when the number of faulty ranks is low, returning data that is wrong but "small" (in the L-2 norm sense) is clearly better than returning results that are incorrect by large orders of magnitude. Future work will pursue with our collaborators this common strategy for recovery from both soft and hard faults. Finally, we will investigate the development and use of programming models that provide selective reliability.

Chapter 7

Conclusion

Expectations of the reliability of future computing hardware have suggested that soft errors may become more likely. We have presented a different approach for both hardening and assessing algorithmic components given an uncertain fault model. We have shown experimentally and analytically that bit flips in floating point data behave exactly as the model prescribes. Furthermore, we have shown how the existing techniques of normalization and equilibration can be used to skew the distribution of errors such that a bit flip in the representation will introduce a small error most of the time. We have demonstrated the applicability of finding bit flips to the GMRES solver, and shown that the characteristics of data are very important when considering soft error resilience.

We have argued that bounding errors is an effective technique for iterative linear solvers, and shown experimentally that bounded errors introduce lower overhead than unbounded. Additionally, we have derived a new error detector for the Arnoldi process, and shown how this bound can be applied to the GMRES iterative linear solver. Our error detector is based on a theoretical model of data corruption, and is extremely low overhead to implement. We have shown that an abstract fault model that introduces pessimistic errors is more useful for designing and assessing resilient algorithms. We compare the overhead introduced by our numerical fault model to those of synthetic bit flips, and show that our approach is consistently bad, even if the numerical errors are very small. We then use our fault model to assess various algorithmic knobs available to linear solvers, and construct a solver configuration that handles catastrophic errors with overheads of 1-2x the cost of solving the system with no faults injected. Furthermore, we show that our approach is capable of using arbitrary numerical routines as preconditioners, and that our resilient solver works through errors more than an order of magnitude more efficiently that related works. These findings validate the hypothesis of this dissertation.

Acknowledgment

This work was supported in part by grants from NSF (awards 1058779 and 0958311). This material is based upon work supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under Program Manager Dr. Karen Pao. Sandia National Laboratories is a multiprogram laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. This research was supported by the Consortium for Advanced Simulation of Light Water Reactors (http://www.casl.gov), an Energy Innovation Hub (http://www.energy.gov/hubs) for Modeling and Simulation of Nuclear Reactors under U.S. Department of Energy Contract No. DE-AC05-00OR22725.

REFERENCES

[1] B. Adamczewski. The many faces of the Kempner number. *J. Integer Seq.*, 16(2):34, 2013.

[2] A. Al-Yamani, N. Oh, and E. J. McCluskey. Performance evaluation of checksum-based ABFT. In *Symposium on Defect and Fault Tolerance in VLSI Systems (DFT 2001)*, pages 461–466, Oct. 2001.

[3] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, PA, USA, third edition, 1999. ISBN 0-89871-447-8.

[4] C. J. Anfinson and F. T. Luk. Linear algebraic model of algorithm-based fault tolerance. *IEEE Transactions on Computers*, 37(12):1599–1604, 1988.

[5] W. E. Arnoldi. The principle of minimized iterations in the solution of the matrix eigenvalue problem. *Quarterly of Applied Mathematics*, 9:17–29, 1951.

[6] K. Asanovic, R. Bodik, B. Christopher, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec. 2006. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html`.

[7] K. Asanovic, R. Bodik, J. W. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. A. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. A. Yelick. A View of the Parallel Computing Landscape. *Comm. ACM*, 52(10):56–67, 2009.

[8] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991. URL `citeseer.ist.psu.edu/article/bailey94nas.html`.

[9] C. G. Baker and M. A. Heroux. Tpetra, and the use of generic programming in scientific computing. *Scientific Programming*, 20(2):115–128, 2012.

[10] P. Banerjee, J. T. Rahmeh, C. Stunkel, V. S. Nair, K. Roy, V. Balasubramanian, and J. A. Abraham. Algorithm-based fault tolerance on a hypercube multiprocessor. *Computers, IEEE Transactions on*, 39(9):1132–1145, 1990.

[11] E. Bavier, M. Hoemmen, S. Rajamanickam, and H. Thornquist. Amesos2 and Belos: Direct and iterative solvers for large sparse linear systems. *Scientific Programming*, 20(3):241–255, 2012.

[12] D. Boley, G. H. Golub, S. Makar, N. Saxena, and E. J. Mccluskey. Floating point fault tolerance with backward error assertions. *IEEE Transactions on Computers*, 44:302–311, 1995.

[13] P. G. Bridges, K. B. Ferreira, M. A. Heroux, and M. Hoemmen. Fault-tolerant linear solvers via selective reliability. *ArXiv e-prints*, June 2012. Provided by the SAO/NASA Astrophysics Data System.

[14] G. Bronevetsky and B. de Supinski. Soft error vulnerability of iterative linear algebra methods. In *Proceedings of the 22nd Annual International Conference on Supercomputing*, ICS '08, pages 155–164, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-158-3. doi: 10.1145/1375527.1375552.

[15] E. J. Candès and P. Randall. Highly robust error correction by convex programming. *IEEE Trans. Inform. Theory*, 54:2829–2840, 2006.

[16] F. Cappello, G. A. A. Geist, W. D. B. Gropp, L. V. S. Kale, W. T. C. B. Kramer, and M. Snir. Toward exascale resilience. Technical Report TR-JLPC-09-01, University of Illinois at Urbana-Champaign (UIUC) - Institut National de Recherche en Informatique et en Automatique (INRIA) Joint Laboratory on PetaScale Computing, June 2009. URL `http://institutes.lanl.gov/resilience/docs/TowardExascaleResilience.pdf`.

[17] M. Casas, B. R. de Supinski, G. Bronevetsky, and M. Schulz. Fault resilience of the algebraic multi-grid solver. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 91–100, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1316-2. doi: 10.1145/2304576.2304590. URL `http://doi.acm.org/10.1145/2304576.2304590`.

[18] Y. Chen and Y. Deng. A detailed analysis of communication load balance on BlueGene supercomputer. *Comp. Phys. Comm.*, 180(8):1251–1258, 2009.

[19] Z. Chen. Algorithm-based recovery for iterative methods without checkpointing. In *Symposium on High-Performance Parallel and Distributed Computing*, pages 73–84, June 2011.

[20] Z. Chen. Online-ABFT: An online algorithm based fault tolerance scheme for soft error detection in iterative methods. In *PPOPP*, PPoPP '13, pages 167–176, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1922-5. doi: 10.1145/2442516.2442533.

[21] Z. Chen and J. Dongarra. Algorithm-based fault tolerance for fail-stop failures. *IEEE Trans. Parallel Distrib. Syst.*, 19:1628–1641, Dec. 2008.

[22] T. Davies and Z. Chen. Correcting soft errors online in LU factorization. In *Proceedings of the 22^{nd} International Symposium on High-Performance Parallel and Distributed Computing*, pages 167–178, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1910-2. doi: 10.1145/2462902.2462920.

[23] T. Davies, C. Karlsson, H. Liu, C. Ding, and Z. Chen. High performance LINPACK benchmark: A fault tolerant implementation without checkpointing. In *Proceedings of the 25^{th} Annual International Conference on Supercomputing*, pages 162–171, May 2011.

[24] T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*, 38(1):1:1–1:25, 2011.

[25] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Analysis and Applications*, 20(3): 720–755, 1999.

[26] J. J. Dongarra, P. Luszczek, and A. Petitet. The linpack benchmark: Past, present, and future. *Concurrency and Computation: Practice and Experience*, 9(15):803–820, Aug. 2003. ISSN 1532-0634.

[27] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra. Algorithm-based fault tolerance for dense matrix factorizations. *SIGPLAN Not.*, 47(8):225–234, Feb. 2012.

[28] J. Elliott, K. Kharbas, D. Fiala, F. Mueller, C. Engelmann, and K. Ferreira. Combining partial redundancy and checkpointing for HPC. In *International Conference on Distributed Computing Systems*, 2012. doi: DOI10.1109/ICPP.2012.21.

[29] J. Elliott, M. Hoemmen, and F. Mueller. Evaluating the impact of SDC on the GMRES iterative solver. In *28th IEEE International Parallel & Distributed Processing Symposium*, Phoenix, USA, May 2014.

[30] J. Elliott, M. Hoemmen, and F. Mueller. Exploiting data representation for fault tolerance. In *Proceedings of the 5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, pages 9–16, 2014. ISBN 978-1-4799-7562-4. doi: 10.1109/ScalA.2014.5. URL `http://dx.doi.org/10.1109/ScalA.2014.5`.

[31] J. Elliott, M. Hoemmen, and F. Mueller. A numerical soft fault model for iterative linear solvers. In *Proceedings of the 24nd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, Portland, OR, 2015. ACM. doi: 10.1145/2749246.2749254.

[32] J. Elliott, M. Hoemmen, and F. Mueller. Exploiting data representation for fault tolerance. *Elsevier Journal of Compuitational Science*, ( ): , 2015.

[33] J. Elliott, M. Hoemmen, and F. Mueller. On the expected error of a bit flip in an IEEE-754 scalar. *SIAM J. Sci. Comput.*, ( ): , 2016.

[34] J. Elliott, M. Hoemmen, and F. Mueller. Selective reliability and preconditioned linear solvers. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, Barcelona, Spain, 2016. ACM.

[35] E. N. M. Elnozahy, R. Bianchini, T. El-Ghazawi, A. Fox, F. Godfrey, A. Hoisie, K. McKinley, R. Melhem, J. S. Plank, P. Ranganathan, and J. Simons. System resilience at extreme scale. Technical report, Defense Advanced Research Project Agency (DARPA), 2008. URL `http://institutes.lanl.gov/resilience/docs/IBMMootazWhitePaperSystemResilience.pdf`.

[36] K. Ferreira, J. Stearley, J. H. L. III, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. Bridges, and D. Arnold. Evaluating the viability of process replication reliability for exascale systems. In *Supercomputing*, Nov. 2011.

[37] K. Ferreira, K. Pedretti, P. G. Bridges, R. Brightwell, D. Fiala, and F. Mueller. Evaluating operating system vulnerability to memory errors. In *Workshop on Runtime and Operating Systems for Supercomputers*, June 2012. doi: DOI10.1145/2318916.2318930.

[38] D. Fiala, K. Ferreira, F. Mueller, and C. Engelmann. A tunable, software-based DRAM error detection and correction library for HPC. In *Workshop on Resiliency in High Performance Computing (Resilience) in Clusters, Clouds, and Grids*, pages 110–121, Sept. 2011. doi: DOI10.1007/978-3-642-29740-3_29.

[39] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *Supercomputing*, pages 78:1–78:12, Nov. 2012.

[40] L. Flynn. Intel halts development of 2 new microprocessors. *The New York Times*, (8), May 2004.

[41] H. Gahvari, A. H. Baker, M. Schulz, U. M. Yang, K. E. Jordan, and W. Gropp. Modeling the performance of an algebraic multigrid cycle on HPC platforms. In *Proceedings of the International Conference on Supercomputing*, pages 172–181, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0102-2. doi: 10.1145/1995896.1995924. URL `http://doi.acm.org/10.1145/1995896.1995924`.

[42] J. Gaidamour, J. Hu, C. Siefert, and R. Tuminaro. Design considerations for a flexible multigrid preconditioning library. *Scientific Programming*, 20(3):223–239, 2012. doi: 10.3233/SPR-2012-0344.

[43] D. Geers. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, May 2005. ISSN 0018-9162. doi: 10.1109/MC.2005.160.

[44] A. Geist. What is the monster in the closet? Invited Talk at Workshop on Architectures I: Exascale and Beyond: Gaps in Research, Gaps in our Thinking, Aug. 2011.

[45] G. H. Golub and Q. Ye. Inexact preconditioned conjugate gradient method with inner-outer iteration. *SIAM J. Sci. Comput.*, 21:1305–1320, 1999.

[46] I. S. Haque and V. S. Pande. Hard data on soft errors: A large-scale assessment of real-world error rates in GPGPU. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 691–696, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4039-9. doi: http://dx.doi.org/10.1109/CCGRID.2010.84. URL `http://dx.doi.org/10.1109/CCGRID.2010.84`.

[47] M. A. Heroux. Scalable Computing Challenges: An Overview. Minisymposium talk at SIAM Annual Meeting: Supercomputing Challenges: Petascale and Beyond, July 2009.

[48] M. A. Heroux and J. Dongarra. Toward a New Metric for Ranking High Performance Computing Systems. Technical Report SAND2013-4744, Sandia National Laboratories, 2013.

[49] M. A. Heroux, R. Bartlett, V. H. R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring, and A. Williams. An Overview of Trilinos. Technical Report SAND2003-2927, Sandia National Laboratories, 2003.

[50] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, , and R. W. Numrich. Improving performance via mini-applications. Technical Report SAND2009-5574, Sandia National Laboratories, September 2009.

[51] M. A. Heroux, J. Dongarra, and P. Luszczek. HPCG technical specification. Technical Report SAND2013-8752, Sandia National Laboratories, 2013.

[52] M. A. Heroux et al. An overview of the Trilinos project. *Transactions on Mathematical Software*, 31(3):397–423, Sept. 2005.

[53] M. R. Hestenes and E. Stiefel. Methods of Conjugate Gradients for Solving Linear Systems. *Journal of Research of the National Bureau of Standards*, 49(6):409–436, Dec. 1952.

[54] N. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, second edition, 2002. doi: 10.1137/1.9780898718027. URL `http://epubs.siam.org/doi/abs/10.1137/1.9780898718027`.

[55] V. Howle and P. Hough. The effects of soft errors on krylov methods. Invited Talk. SIAM Parallel Processing., Feb. 2012.

[56] V. Howle, P. Hough, M. A. Heroux, and E. Durant. Soft errors in linear solvers as integrated components of a simulation. Invited talk at the Copper Mountain Conference on Iterative Methods, Apr. 2010.

[57] C.-H. Hsu and W.-C. Feng. A power-aware run-time system for high-performance computing. In *Supercomputing*, 2005.

[58] K.-H. Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Trans. Comput.*, C-33(6):518–528, June 1984.

[59] A. A. Hwang, I. A. Stefanovici, and B. Schroeder. Cosmic rays don't strike twice: understanding the nature of DRAM errors and the implications for system design. In *Architectural Support for Programming Languages and Operating Systems*, pages 111–122, 2012.

[60] Intel. FDIV replacement program: Description of the flaw, July 2004. Available online: `http://support.intel.com/support/processors/pentium/sb/CS-013007.htm` [last accessed 04 Sep 2013].

[61] T. Karnik, P. Hazucha, and J. Patel. Characterization of soft errors caused by single event upsets in CMOS processes. *IEEE Trans. Dependable Secur. Comput.*, 1:128–143, April 2004. ISSN 1545-5971. doi: http://dx.doi.org/10.1109/TDSC.2004.14. URL `http://dx.doi.org/10.1109/TDSC.2004.14`.

[62] Y. Kim, J. S. Plank, and J. J. Dongarra. Fault tolerant matrix operations using checksum and reverse computation. In *Symposium on the Frontiers of Massively Parallel Computing*, pages 70–77, Oct. 1996.

[63] P. Kogge et al. ExaScale computing study: Technology challenges in achieving exascale systems. Technical report, Defense Advanced Research Project Agency, Information Processing Techniques Office, 2008. URL `http://users.ece.gatech.edu/mrichard/ExascaleComputingStudyReports/exascale_final_report_100208.pdf`.

[64] D. Lammers. The era of error-tolerant computing. *IEEE Spectr.*, 47(11):15, Nov. 2010.

[65] T. Lange. Complex engineered systems at proctor & gamble. Smoky Mountain2 Computational Science and Engineering Conference, Sept. 2012. URL `http://computing.ornl.gov/workshops/FallCreek12/presentations/Lange-ComplexEngineeredSystems-SMC12.pdf`.

[66] D. Li, J. Vetter, and W. Yu. Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool. In *Supercomputing*, Nov. 2012.

[67] P. Lin, C. Vaughn, R. Barrett, M. A. Heroux, and A. Williams. Mini-applications: Vehicles for co-design. Technical report, Sandia National Laboratories, Nov. 2011. Best Conference Poster Award.

[68] P. Lin, M. Bettencourt, S. Domino, T. Fisher, and M. Hoemmen. Towards extreme-scale simulations with next-generation Trilinos: a low mach fluid application case study. In *Workshop on Large-Scale Parallel Processing*, May 2014.

[69] A. Meek, V. Howle, and M. Hoemmen. Fault Tolerant QMR. Minisymposium talk at SIAM Computational Science and Engineering, Feb. 2013.

[70] S. Michalak, A. Dubois, C. Storlie, H. Quinn, W. Rust, D. DuBois, D. Modl, A. Manuzzato, and S. Blanchard. Assessment of the impact of cosmic-ray-induced neutrons on hardware in the Roadrunner supercomputer. *Device and Materials Reliability, IEEE Transactions on*, 12(2):445–454, 2012. ISSN 1530-4388. doi: 10.1109/TDMR.2012.2192736.

[71] N. Miskov-Zivanov and D. Marculescu. Soft error rate analysis for sequential circuits. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '07, pages 1436–1441, San Jose, CA, USA, 2007. EDA Consortium. ISBN 978-3-9810801-2-4. URL `http://portal.acm.org/citation.cfm?id=1266366.1266680`.

[72] A. Moody, G. Bronevetsky, K. Mohror, and B. de Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Supercomputing*, Nov. 2010.

[73] E. Mueller and R. Scheichl. Massively parallel solvers for elliptic PDEs in numerical weather and climate prediction. *ArXiv e-prints*, June 2013. URL `http://arxiv.org/abs/1307.2036`.

[74] C. C. Paige, M. Rozložník, and Z. Strakoš. Modified Gram-Schmidt (MGS), least squares, and backward stability of MGS-GMRES. *SIAM J. Matrix Anal. Appl.*, 28(1):264–284, 2006.

[75] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software implemented fault tolerance. In *International Symposium on Code Generation and Optimization*, pages 243–254, 2005. doi: 10.1109/CGO.2005.34.

[76] Y. Saad. A flexible inner-outer preconditioned GMRES algorithm. *SIAM J. Sci. Comput.*, 14(2):461–469, Mar. 1993. ISSN 1064-8275. doi: 10.1137/0914028. URL `http://dx.doi.org/10.1137/0914028`.

[77] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003. ISBN 0898715342.

[78] Y. Saad and M. H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7(3):856–869, July 1986. ISSN 0196-5204. doi: 10.1137/0907058.

[79] P. Sao and R. Vuduc. Self-stabilizing iterative solvers. In *Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, pages 4:1–4:8, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2508-0. doi: 10.1145/2530268.2530272.

[80] J. Schlaich and K.-H. Raineck. Die Ursache für den Totalverlust der Betonplattform Sleipner A. *Beton- und Stahlbetonbau*, 88:1–4, 1993.

[81] B. Schroeder, E. Pinheiro, and W.-D. Weber. DRAM errors in the wild: a large-scale field study. In *SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 193–204, 2009.

[82] M. Shantharam, S. Srinivasmurthy, and P. Raghavan. Characterizing the impact of soft errors on iterative methods in scientific computing. In *Proceedings of the 25th International Conference on Supercomputing*, ICS '11, pages 152–161, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0102-2. doi: 10.1145/1995896.1995922.

[83] M. Shantharam, S. Srinivasmurthy, and P. Raghavan. Fault tolerant preconditioned conjugate gradient for sparse linear system solution. In *Proceedings of the 26th ACM*

International Conference on Supercomputing, ICS '12, pages 69–78, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1316-2. doi: 10.1145/2304576.2304588.

[84] V. Simoncini and D. B. Szyld. Flexible inner-outer krylov subspace methods. *SIAM Journal on Numerical Analysis*, 40(6):2219–2239, 2002. doi: 10.1137/S0036142902401074. URL `http://dx.doi.org/10.1137/S0036142902401074`.

[85] V. Simoncini and D. B. Szyld. Theory of inexact Krylov subspace methods and applications to scientific computing. *SIAM J. Sci. Comput.*, 25(2):454–477, Feb. 2003. ISSN 1064-8275. doi: 10.1137/S1064827502406415.

[86] J. Sloan, R. Kumar, and G. Bronevetsky. Algorithmic approaches to low overhead fault detection for sparse linear algebra. In *Proceedings of the 2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12, Washington, DC, USA, 2012. ISBN 978-1-4673-1624-8. URL `http://dl.acm.org/citation.cfm?id=2354410.2355166`.

[87] V. Sridharan, N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi. Memory errors in modern systems: The good, the bad, and the ugly. *SIGARCH Comput. Archit. News*, 43(1):297–310, Mar. 2015. ISSN 0163-5964. doi: 10.1145/2786763.2694348. URL `http://doi.acm.org/10.1145/2786763.2694348`.

[88] G. W. Stewart. Updating a rank-revealing ULV decomposition. *SIAM J. Matrix Anal. Appl.*, 14(2):494–499, April 1993.

[89] D. B. Szyld and J. A. Vogel. FQMR: A flexible quasi-minimal residual method with inexact preconditioning. *SIAM J. Sci. Comput.*, 23(2):363–380, 2001.

[90] N. Taerat, N. Naksinehaboon, C. Chandler, J. Elliott, C. Leangsuksun, G. Ostrouchov, and S. L. Scott. Using log information to perform statistical analysis on failures encountered by large-scale hpc deployments. In *High Availability and Performance Computing Workshop*, HAPCW '08, pages 6–15. IEEE Computer Society Press, Apr. 2008.

[91] N. Taerat, N. Naksinehaboon, C. Chandler, J. Elliott, C. Leangsuksun, G. Ostrouchov, S. L. Scott, and C. Engelmann. Blue gene/l log analysis and time to interrupt estimation. In *Proceedings of the International Conference on Availability, Reliability and Security*, AReS '09, pages 173–180. IEEE Computer Society Press, Mar. 2009.

[92] U.S. Department of Energy. Top ten exascale research challenges. Technical report, U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, Feb. 2014. URL `http://science.energy.gov/~/media/ascr/ascac/pdf/meetings/20140210/Top10reportFEB14.pdf`.

[93] H. J. J. van Dam, A. Vishnu, and W. A. de Jong. A case for soft error detection and correction in computational chemistry. *Journal of Chemical Theory and Computation*, 9(9):3995–4005, 2013. doi: 10.1021/ct400489c. URL `http://pubs.acs.org/doi/abs/10.1021/ct400489c`.

[94] P. Wu and Z. Chen. FT-ScaLAPACK: Correcting soft errors on-line for ScaLAPACK Cholesky, QR, and LU factorization routines. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, pages 49–60, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2749-7. doi: 10.1145/2600212.2600232. URL `http://doi.acm.org/10.1145/2600212.2600232`.