2016

Enhancing Program Soft Error Resilience through Algorithmic Approaches

Sui Chen
Louisiana State University and Agricultural and Mechanical College, csui1@lsu.edu

ENHANCING PROGRAM SOFT ERROR RESILIENCE THROUGH ALGORITHMIC APPROACHES

A Thesis

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering

in

The School of Electrical Engineering and Computer Science

Division of Electrical and Computer Engineering

by
Sui Chen
B.E., Shanghai Jiao Tong University, Shanghai, China 2011
December 2016

Acknowledgments

I would like to thank Professor Lu Peng for guiding me and sharing his knowledge and experience on research and standing up to tough life. I would also like to thank our collaborator Greg Bronevetsky for introducing me to this research field and encouraging me during the poster presentation, and professor Bin Li for his help on the statistical methods used in this project.

We are grateful to Vishal Sharma and Arvind Haran, the authors of the original KULFI and for granting us permission to modify it for our experiment purposes. We are also appreciative of the opportunity to be involved in and contribute to KULFI.

Thanks go to Prof Gerald Baumgartner and Prof Xiangwei Zhou for their involvement in the committee for this thesis.

Thanks also go to fellow coworkers and friends who give me psychological support and enrich my life here, and to the students who continue on to further explorations in this project.

Table of Contents

List of Tables

List of Figures

Abstract

The rising count and shrinking feature size of transistors within modern computers is making them increasingly vulnerable to various types of soft faults. This problem is especially acute in high-performance computing (HPC) systems used for scientific computing, because these systems include many thousands of compute cores and nodes, all of which may be utilized in a single large-scale run.

The increasing vulnerability of HPC applications to errors induced by soft faults is motivating extensive work on techniques to make these applications more resilient to such faults, ranging from generic techniques such as replication or checkpoint/restart to algorithm-specific error detection and tolerance techniques.

Effective use of such techniques requires a detailed understanding of how a given application is affected by soft faults to ensure that (i) efforts to improve application resilience are spent in the code regions most vulnerable to faults, (ii) the appropriate resilience techniques is applied to each code region, and (iii) the understanding be obtained in an efficient manner.

This thesis presents two tools: FaultTelescope helps application developers view the routine and application vulnerability to soft errors while ErrorSight helps perform modular fault characteristics analysis for more complex applications. This thesis also illustrates how these tools can be used in the context of representative applications and kernels. In addition to providing actionable insights into application behavior, the tools automatically selects the number of fault injection experiments required to efficiently generation error profiles of an application, ensuring that the information is statistically well-grounded without performing unnecessary experiments.

Chapter 1
Introduction

Soft faults can cause errors in HPC and big data analytics programs, which can then cause incorrect results, interruption of service or system downtime. The occurrence of such faults increase as device features shrink while system scales and density rise. Because of this, it is necessary to understand the faults and develop countermeasures to prevent undesirable consequences.

This thesis discusses characterizing the errors in programs caused by soft faults and enhancing program resilience to such faults using algorithmic approaches. The algorithmic approaches include both exploiting algorithmic invariants used in the applications and algorithms used for analyzing experimental data.

The goal of the development of such tools is three-fold: (1) Help developers improve program resilience with focused efforts on vulnerable code regions, (2) Determine appropriate resilience techniques for each code region and (3) Complete the first two goals in an efficient manner.

This thesis starts by presenting in Chapter 2 the background relevant to soft faults, numerical programs and fault injection techniques. Chapter 3 discusses the first tool, FaultTelescope, which completes the first two tasks, namely, to help a developer and demonstrate how to evaluate fault resilience with it. The tool is applied to various numerical routines and scientific computing programs that build on these routines. Chapter 4 discusses the second tool, ErrorSight, which completes the third task of obtaining fault characteristics efficiently using modular analysis. The tool is applied to several big data and scientific computation programs which contain iterative algorithms. In Chapter 5 we conclude this thesis.

Chapter 2

Background

2.1 Soft Faults

As HPC systems approach Exascale and big data analytics becoming prevalent, the circuit features in the hardware of these computational systems will shrink while their overall size will grow, both at a fixed power limit. These trends imply that soft faults in electronic circuits will become an increasingly significant problem for programs that run on these systems. Soft faults are transient corruptions of the states of electronic circuits caused by physical phenomena such as strikes by neutrons, alpha particles [4, 31] or thermal electrical noise [26]. They can affect processor latches and registers, which could cause the program to crash or worse, silently return incorrect results [11]. Today the error rates in DRAMs have been reported to reach 70,000 FITs (failures per billion device hours) per Mbits [39]. As the feature sizes of electronic circuits shrink, technology scaling will exacerbate soft errors [30] due to the fact that each circuit element will hold less charge and can thus be disrupted more easily. In particular, processors in 2020 are expected to have feature sizes (DRAM $\frac{1}{2}$ Pitch) of approximately 5 to 7 nm [22], which is approximately 10 to 14 silicon atoms (5 Å per atom) across. These phenomena make it imperative to develop mechanisms to make HPC and big data systems resilient to soft faults.

The importance of analyzing and quantifying the impact of errors on program behavior is demonstrated in various studies. As Du et al have shown [14, 15], resilience is becoming a quality measurement of linear solver packages. A detailed study of output accuracy is found in several fault injection frameworks. For example, Debardeleben et al [12] document how the numeric error caused by an injected fault evolves over time. Probabilistic modeling has been used by Chung et al [10] to help compute the expected recovery time, which cannot be measured easily for very large scale programs. Sloan et al [41] have discussed the use of

algorithmic checks over sparse linear algebra kernels and focused mainly on reducing false positive and false negative in error detection.

2.2 Fault Injection

The occurrence of soft faults resilience studies are relatively rare, and only on very large scale systems do soft errors occur frequently enough to be observed in a short time span. As a result, many resilience studies rely on fault injection, a technique for introducing faults to running programs. The program states of the running programs are modified to reflect software-level manifestation of low-level faults. In addition to software-level fault injectors, there exist tools that simulate various types of faults in hardware components, ranging from transistor-level faults to fail-stop crashes of entire compute nodes [27, 1, 20].

The tools in this thesis are based on the KULFI fault injector [2], which models faults as single bit flips in the outputs of a randomly selected instruction of a program compiled into the LLVM instruction set. LLVM is a compiler infrastructure that uses a Static Single Assignment (SSA)-based compilation strategy that is capable of supporting arbitrary programming languages [25].

2.3 Fault Resilience

The resilience problem must be addressed at all levels. On the physical level, efforts in materials science and circuit design techniques are made to improve resilience, but the cost of building processors sufficiently reliable for a large HPC or big data system is still prohibitive. On the digital logic level, mechanisms such as error correcting codes (ECC) have been very effective at making memories and caches resilient to soft faults [28]. However, as total system memories are expected to grow by 100x to 350x to reach Exascale [13], their increased fault vulnerability will require more elaborate and expensive ECC to be deployed. Further, ECC is more expensive for protecting core-internal states such as latches and is significantly less effective for checking the correctness of computations. On the processor architecture level, designs that incorporate instruction replication [36] offer fine-grained error detection and rollback but require more power as well as novel hardware features

that are unlikely to be included in the commodity processors used in HPC systems for cost reduction reasons.

The limitations of hardware-level resilience solutions have motivated significant work on the design of software-level mechanisms that can enable programs to execute productively on unreliable hardware. The most general approach is replication of computations across core or nodes [17, 24], which is very easy to use but can incur a high overhead due to repeated computation, result comparison, and management of non-determinism across replicas. There has also been extensive work on hand-coded, more efficient algorithm-specific techniques [40, 21] that verify the algorithmic invariants hold. Because these mechanisms usually only address error detection, to achieve full resilience they must be supported by other techniques, such as checkpoint-restart [32] and pointer replication [9]. For these general techniques, the user has to decide smartly when and where to deploy them effectively at a reasonable cost. For algorithm-specific techniques, the user needs to gain a thorough understanding of the algorithm in question to develop fault resilience techniques. Both would require significant amount of efforts.

Chapter 3
Evaluating Comprehensive Fault Resilience Mechanisms In Numerical Programs

3.1 Overview

As mentioned in the Introduction, this chapter presents FaultTelescope, a comprehensive approach to supporting both needs in the form of (1) statistically well-grounded fault injection studies and (2) exploration of how the configuration of a resilience mechanism affects the performance and resilience of individual kernels as well as the entire program. This tool generates actionable insights by presenting program vulnerabilities and impact of fault resilience mechanisms in an intuitive way.

FaultTelescope supports resilience studies by integrating with the KULFI fault injector [2], which models faults as single bit flips in the outputs of a randomly selected instruction of a program compiled into the LLVM instruction set. LLVM is a compiler infrastructure that uses a Static Single Assignment (SSA)-based compilation strategy that is capable of supporting arbitrary programming languages [25]. FaultTelescope presents the results of resilience studies to developers by providing visualizations of how program states and output are affected by injected errors. The errors are expressed via developer-specified error metrics. Furthermore, FaultTelescope computes confidence intervals of the presented data to enable developers to make well-grounded conclusions, while balancing the benefits from improved confidence intervals of the analysis and the cost of running more fault injection experiments.

A key issue developers face is that different types of faults manifest themselves differently to software. For each possible fault type developers need to select the most appropriate resilience mechanism for detecting and tolerating the fault, as well as the best configuration of the mechanism. The choice of mechanism and its configuration has a no-

ticeable effect on the performance and resilience of the program. Furthermore, a wrong choice may render the program more vulnerable to errors than it originally is[7]. FaultTelescope helps program developers choose the best way to manage all the fault types their programs may be vulnerable to by helping them experimentally measure the effectiveness of various resilience mechanisms and the implication of their configurations. To reduce the cost of searching a large parameter space, FaultTelescope directs developers to first focus on key kernels and then on the entire program.

On a high level, FaultTelescope provides a comprehensive suite of capabilities that help program developers bridge the gap between low-level faults and software-level resilience solutions. FaultTelescope consists of:

- Efficient architectural level fault injection with KULFI
- Statistically sound computations of confidence intervals of fault characteristics
- Hierarchical analysis that operates on kernels through entire programs

The FaultTelescope approach is evaluated in the context of three programs that represent different application domains: the LASSO [5] solver for the linear solvers domain, the DRC [38] HiFi audio filter for the signal processing domain, and the Hattrick [35] gravity simulator for the differential equation solvers domain. This chapter demonstrates the utility of this comprehensive resilience toolchain for helping developers explore the vulnerability properties of their programs.

The rest of the chapter is organized as follows. Section 3.2 gives an overview of the experimentation methodology and error model used in FaultTelescope. Section 3.3 presents the structure of the target programs. Section 3.4 describes the fault resilience mechanisms used. Section 3.5 presents how FaultTelescope finds fault characteristics and performance/resilience tradeoffs. Section 3.6 presents the algorithm used for selecting the number of fault injection experiments needed for statistically-grounded analysis. We conclude this chapter in section 3.7.

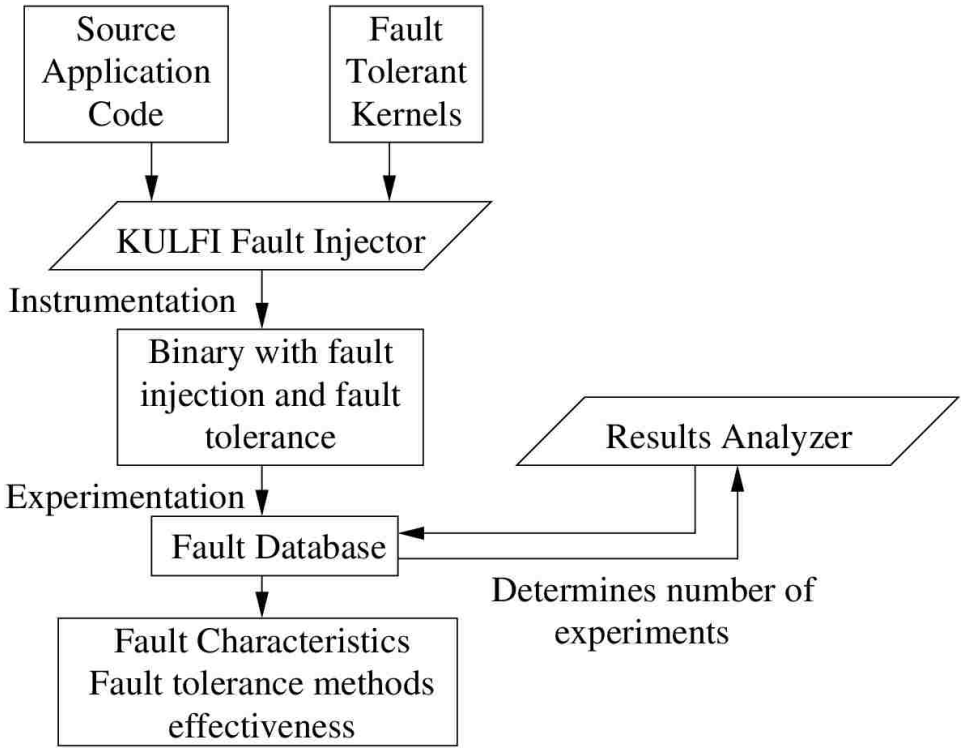

Figure 3.1: Overall workflow of ErrorSight

3.2 Design of FaultTelescope

The workflow of FaultTelescope is described in Figure 3.1. It performs a fault injection campaign on a target program by executing the entire program and/or individual routines in the program multiple times. During each run, a single bit flip is injected in a randomly-selected Dynamic Fault Site (a dynamic LLVM instruction, which is an instance of a static LLVM instruction in the program's binary image). Information including source code location that corresponds to the fault site and the final outcome of the program will also be recorded. The final outcome of one run falls into one of these categories:

- Correct Result: The program runs to completion and outputs the correct result, as if no error occurred at all.

- Abnormal Termination: Program performs abnormal action such as dereferencing invalid pointers, encountering numerical explosion, or entering an infinite loop, which then triggers user-defined or system-defined exception handlers, resulting in the pro-

gram being terminated.

- Incorrect Result: Program runs to completion, but produces results that exceed the user-defined error bound and is considered incorrect. In this chapter we quantify the magnitude of errors using the RMSD (root-mean-squared deviation) between the incorrect result and the correct result.

The information above is stored in the fault database for analysis and visualization. The result analyzer uses the information to determine the number of experiments needed for obtaining a statistically-grounded conclusion about the fault characteristics of programs. The fault characteristics of a program is quantified by the probability of each of the outcomes and the distribution of RMSD in incorrect results.

3.3 Target Applications

We demonstrate the use of FaultTelescope on three programs, which represent three application domains. The fault resilience mechanisms utilized by each program are summarized in Table 3.1. The details of the mechanisms will be discussed in the Section 3.4.

3.3.1 LASSO

The LASSO [5] program is an implementation of the Alternating Direction Method of Multipliers algorithm for solving under-constrained linear problems $Ax = b$ for x (A has fewer rows than columns) while minimizing the cost function $\frac{1}{2}\,||Ax - b|\,|_2^2 + \lambda \cdot ||x|\,|_1$. It represents the linear solver application domain. It uses 64-bit precision and spends most of its time in the following linear algebra operations from the GNU Scientific Library (GSL) [18]: matrix-matrix multiplication (MMM), matrix-vector multiplication (MVM), rank-k update (RK) and Cholesky decomposition (CD).

Our experiments focus on matrices A of size $\{40,\ 80,\ 200,\ 400,\ 600,\ 800\} \times 500$ as input. The values in A and b are generated by sampling a normal distribution with a mean of 0 and a σ of 0.08 and 0.005 respectively.

3.3.2 DRC

Dynamic Range Compression (DRC) [38] is a sequential program that generates filters for high-fidelity audio systems, compensating for the reflection of sounds in a room using impulse response measurements of the audio equipment and the positions of the listeners. It represents the signal processing application domain. DRC inputs are stored in Pulse Code Modulation (PCM) format, which is an array of 32-bit floating point numbers representing the samples at each sample time. Computation is done in 32-bit precision. Most of the execution time is spent in the GSL implementation of Fast Fourier Transform (FFT) and a DRC-internal implementation of Finite Impulse Response (FIR) filter generation. The input used in this chapter is a PCM audio file of size 768 kilobytes, which is internally resampled at 30, 40, 50, 60 or 70 KHz during computation.

3.3.3 Hattrick

Hattrick [35] is a sequential program that simulates the motion of celestial bodies under the effects of gravity to help discover extra-solar planets by inferring their existence from Transit Timing Variations. It represents the n-body simulation application domain. Hattrick uses 64-bit precision and spends most of its execution time in the GSL Runge-Kutta (RK) ordinary differential equation solver. The solver computes the position of the planets and adjusts step size automatically to reach the accuracy target defined in the user's input. A given input is described using three parameters: P is the number of planets, T is the amount of time to simulate, and A is the user-defined accuracy target. In our experiments we considered the following four inputs: $P2T2090A15$, $P2T3090A15$, $P2T4090A15$ and $P3T2090A11$, where $A15$ and $A11$ denote accuracy targets of $1e-15$ and $1e-11$, respectively.

3.4 Resilience Mechanisms

This section presents the fault characteristics on routines used by the three target programs, and how the fault tolerance mechanisms listed in Table 3.1 protect the programs from soft errors. The fault characteristics are quantified by the probability of outcomes

Table 3.1: The resilience mechanisms applied to each major routine of each program.

Routine		Algorithmic Detector	Checkpointing	Pointer Replication
ADDR	MM SYRK MVM CD	Linear encoding Thresholds: 1e-5 to 1e-8	Inputs	None
DRC	FFT FIR	Parseval's theorem. Sum conservation. Thresholds: 1e-6 to 1e-8.	Inputs	None
Hattrick	RK	Variable step-size	Periodic Timesteps in period: 1, 1e4	1. None 2. All pointers, checked at one code location 3. All pointers, checked on each use

which are correct result, abnormal termination and incorrect result.

In Sections 3.4 and 3.5, we consider an output to be correct only if it is *identical* to the output from the run without fault injection (the golden output). For the outputs of program runs that are not correct, we quantify the error using the root-mean-square deviation (RMSD), the difference between two values, which could be a scalar or a vector. It is computed using the formula $RMSD(x, x_{gold}) = \sqrt{\frac{\sum_{t=1}^{n}(x_t - x_{gold,t})^2}{n}}$, where n is the number of elements of the output vectors.

Program developers may take round-off errors and limited machine precision into account and set program-specific correctness thresholds. An output is considered correct if the error is under the program-specific threshold. Unlike real-life programs, the routines in this section are deterministic and much simpler and we choose to only consider an output to be correct only if it is identical to the golden output to better illustrate how the fault resilience mechanisms affect their fault characteristics.

3.4.1 Error Recovery

A light-weight in-memory checkpointing recovery method is deployed to all routines in order to enable recovery from abnormal terminations such as segmentation faults. This is

done by installing a signal handler with the `sigsetjmp` system call and backing up inputs at the entry points of the routines.

3.4.2 Algorithmic Error Detection

- **Cholesky Decomposition (CD)**

The Cholesky Decomposition is a decomposition of the form $A = LL^T$, where L is lower-triangular with a positive diagonal. This operation must maintain the identity $Ax = L(L^T x)$ [21], which is checked by the fault resilient CD algorithm in $O(n^2)$ operations. It is significantly faster than the deterministic CD algorithm which takes $O(n^3)$ operations. GSL implements an iterative algorithm that runs faster than $O(n^3)$ but our experiments show that our checker is still significantly faster.

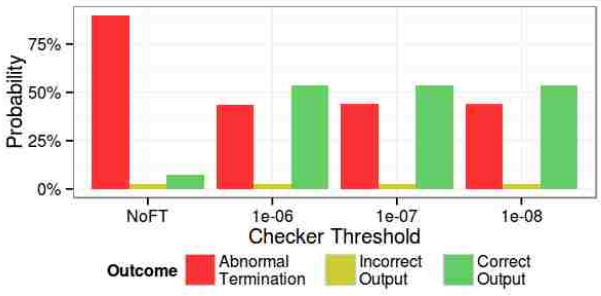

Figure 3.2: Fault characteristics of Cholesky Decomposition given input size 500x500.

GSL's Cholesky Decomposition (CD) routine contains a built-in error checker that terminates the program when the input matrix is not positive-definitive. As a result, injected errors frequently cause the input to be non-positive-definitive, resulting in most runs of the original CD being terminated. On the other hand, the runs that complete usually contain very small errors.

The use of these resilience mechanisms has a significant effect on the probability of abnormal terminations, as is shown in Figure 3.2.

In CD, the positive-definitiveness of matrix A is checked when A is updated at each iteration. The outcome probabilities of the non-fault-tolerant (NoFT) CD suggest that most errors would cause abnormal termination and the chance of producing an incorrect output without triggering the error check is very low. In other words, a run would either terminate abnormally or finish with no error. The probabilities of outcomes of the fault-

tolerant CD suggest that with the added rollback capability, many runs are able to finish with a correct output. The choice of result checker threshold (1e-06, 1e-07 and 1e-08) does not affect the proportion of correct outputs in the outcome. We use $1e-06$ in the programs in Section 3.5.

- **Fast Fourier Transform (FFT)**

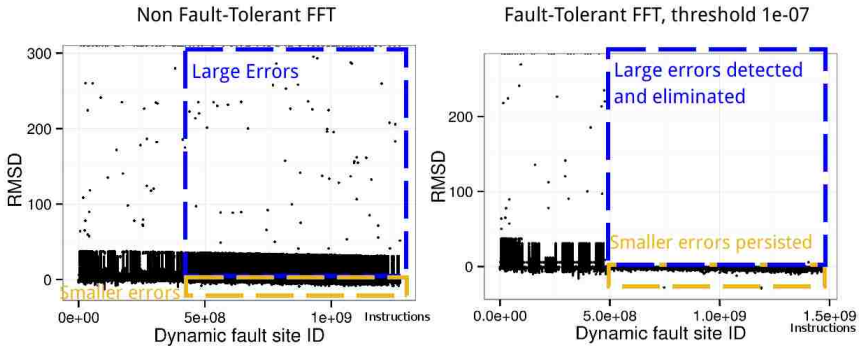

Figure 3.3: Detailed characteristics of fault resilient FFT with checker threshold 1e-07

FFT computes the transform $X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi kn/N}$ for a radix k. The result is checked using Parseval's theorem: $\sum_{n=0}^{N-1} |x[n]|^2 = \frac{1}{N} \sum_{k=0}^{N-1} |X[k]|^2$, where x is the original function and X is its transform. Intuitively it means that the energy of the original function is preserved by the transform.

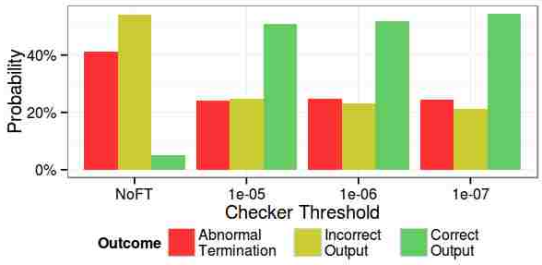

Figure 3.4: Fault characteristics of FFT with input size 4M.

This check takes $O(n)$ operations, which is smaller than $O(nlog(n))$ or $O(n^2)$ for the FFT algorithm, depending on the FFT radix. (For example, for a radix $n = 2 \cdot 3 \cdot 19999$ transform, the $O(n^2)$ scaling would dominate.)

Figure 3.4 summarizes the fault characteristics of different versions of FFT: the possibility of incorrect outputs is significantly reduced by the error checkers. In fact, most of the errors are very large and they can be detected with a lenient threshold such as 1e-05. Figure 3.3 is a temporal error graph, which shows the magnitude of errors (Y axis) caused by faults injected at different dynamic fault sites (dynamic LLVM instructions) (X axis).

The errors are measured with the root-mean-square deviation (RMSD).

Comparing the temporal RMSD graphs one could see that the checker removes larger errors, but smaller errors persisted. A stricter error checker slightly improves the probability of correct outputs.

We choose 1e-07 as the checker threshold and take a closer look at the impact of the smaller errors on a whole program in Section 3.5.

- **Finite Impulse Response Filter Generation (FIR)**

This algorithm generates a sample of the function $sinc(x) = \frac{sin(x)}{x}$ and modulates it with a Blackman window. The result is checked using the invariant $\int_{-\infty}^{\infty} sinc(x)dx = 1$, throughout our experiments. Computing the sum requires $O(n)$ additions and is faster compared to the $O(n)$ trigonometric function evaluations of the original FIR generation algorithm.

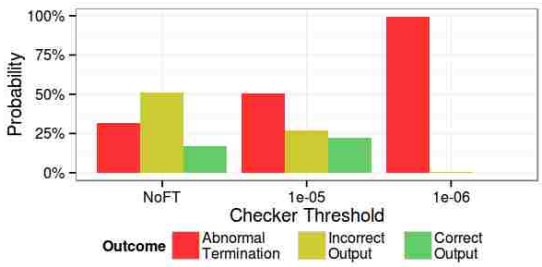

Figure 3.5: Fault characteristics of FIR of input parameter 512K.

The checker threshold 1e-06 is too tight and causes many false alarms, resulting in many runs terminated, as can be seen in Figure 3.5. We choose 1e-05 as the error checker threshold for FIR for it increases the probability of correct outputs.

- **Matrix-Matrix Multiplication (MM)**

The matrix-matrix multiplication (MM) computes $C = AB$. The result is checked using a matrix vector multiplication (MV) on the identity $(AB)x = A(Bx)$, where x is an error-checking vector (we use a vector of all 1s). The checker takes $O(n^2)$ operations and is asymptotically faster than MMM which takes $O(n^3)$ operations.

Figure 3.6 shows the fault characteristics of different versions of the MMM routine. We see from the figure that error checker thresholds 1e-07 and 1e-08 correct more wrong results than 1e-06 does. In the experiments we use 1e-06, 1e-07 and 1e-08 as the error checker thresholds for MM and see how it affects the whole program.

- **Symmetric Rank-K Update**

Symmetric Rank-K Update (SYRK) computes $\alpha AA^T + \beta B$, where A and B are matrices.

The result is checked via the identity $(AA^T)x = A(A^Tx)$, where x is an error-checking vector. We use a vector of all 1s. The checker takes $O(n^2)$ operations. Compared to SYRK which takes $O(n^3)$ operations, the check is much faster. The error checker and recovery in the fault-tolerant SYRK fix many runs with incorrect results as is shown in Figure 3.7. However, some of the incorrect runs are not corrected. This is mainly due to the checker works in a recursive fashion and involves many addition operations and round-off errors would accumulate during the process. As a result, the checker always decides these runs are incorrect and keeps repeating until the attempt limit is exceeded.

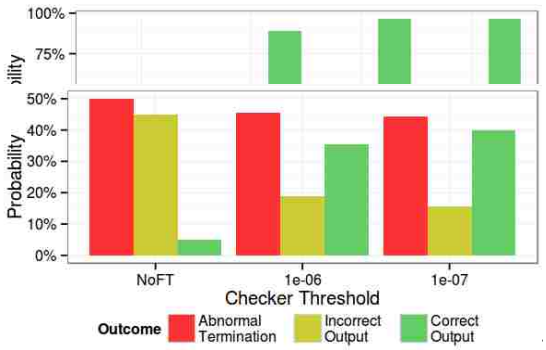

Figure 3.6: Fault characteristics of 500x500 Matrix-Matrix Multiplication.

Figure 3.7: Fault characteristics of 500x500 Symmetric Rank-K Update.

We use 1e-06, 1e-07 and 1e-08 as the error checker thresholds for RK in the experiments in Section 3.5.

- **Matrix-Vector Multiplication**

The Matrix-vector multiplication(MVM) computes Ax, where A is a matrix and x is a vector.

It is checked via the identity $(x^TA)x = x^T(Ax)$. The complexity of computing x^TA takes $O(n^2)$ addition operations. In contrast, the original MVM takes $O(n^2)$ multiplication operations. Since MVM is applied in Lasso many times to the same matrix with different vectors, the vector x^TA can be reused,

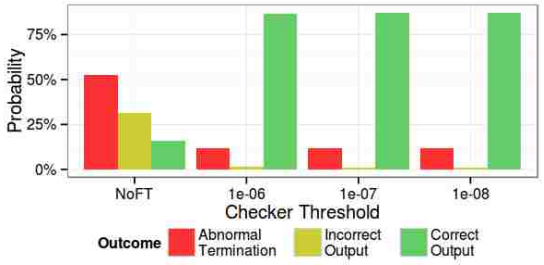

Figure 3.8: Fault characteristics of 500x500 Matrix-Vector Multiplication.

amortizing the cost.

We use 1e-06, 1e-07 and 1e-08 as the error detector thresholds for MV in the experiments. Figure 3.8 summarizes the fault characteristics of Matrix-Vector Multiplication.

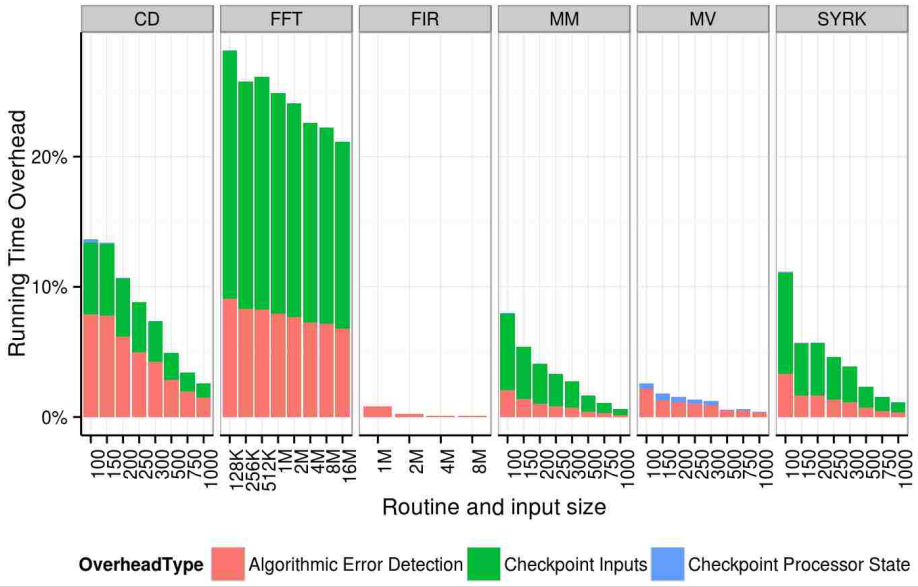

Figure 3.9: Overhead of fault resilience mechanisms for linear algebra kernels, FFT and FIR

The performance overhead of the algorithmic checks for linear algebra and FFT routines are listed in Figure 3.9. The Runge-Kutta integrator will be discussed separately since its resilience is not achieved through algorithmic invariants.

- **Runge-Kutta Integrator**

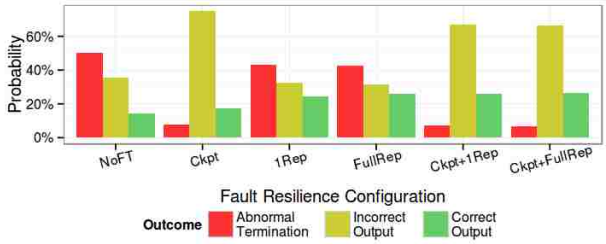

Figure 3.10: Fault Characteristics of the RK4 Integrator

Table 3.2: Overhead of different versions of the RK4 Integrator

Overhead	
Ckpt	<1% (Negligible)
1Rep / 1Rep+Ckpt	21.4%
FullRep / FullRep+Ckpt	54.3%

The 4th order Runge-Kutta method (RK4) is a method for numerically solving an ordinary differential equations of the form $\frac{dy}{dx} = f(y, x)$. It advances the variable x by steps of size h and computes the value y at the next point $x + h$ using the derivative $\frac{dy}{dx}$ at x. GSL's RK4 integrator implementation uses adaptive step-size control where it attempts a smaller $\frac{h}{2}$ and compares the result with that from step size h. If the difference between the two results exceeds a threshold τ, it switches to a smaller step size to maintain accuracy. If it is smaller than $\frac{\tau}{2}$, the algorithm switches to a larger step size to make faster progress. An error resulting from a soft fault can cause the two results to diverge. If the divergence is greater than τ, the step size is decreased, the result is computed again, and the error is masked. If the divergence is smaller, the error will persist until the program finishes.

To protect against abnormal termination, a checkpoint is made at every fixed number of iterations. The number has to chosen wisely: an interval that is too short (for example 1) would incur much overhead in checkpointing, while an interval that is too long (for example, 10^6) means it would take much longer to recover the program states from the last checkpoint to the current time step. From our experiments, the choice of 10000 makes an optimal balance. This routine is tested with the second-order nonlinear Van der Pol oscillator equation in the GSL documentation [18]. Its resilience properties are shown in Figure 3.10 and Table 3.2 respectively.

3.5 Result Analysis

In this section, we present how the fault resilience mechanisms can protect the programs from single bit-flip errors and the performance overhead of the mechanisms. We show that the choice of fault checker threshold and replication strategy can affect the performance overhead and/or accuracy under certain circumstances.

We present the confidence interval of probabilities of all three outcomes of every program configuration with rectangles on a 2-D plane. The binomial confidence intervals of the possibilities of abnormal termination and perfect output are mapped to the X and Y axes respectively. The probability of incorrect results is one minus the sum of the other

two and can be mapped to the distance towards the line segment passing (1,0) and (0,1). Intuitively, an area on the top-left means 100% correct outputs with no abnormal terminations and is desirable. These visualizations give a clear overview of the fault resilience characteristics of the programs under various configurations.

3.5.1 Lasso

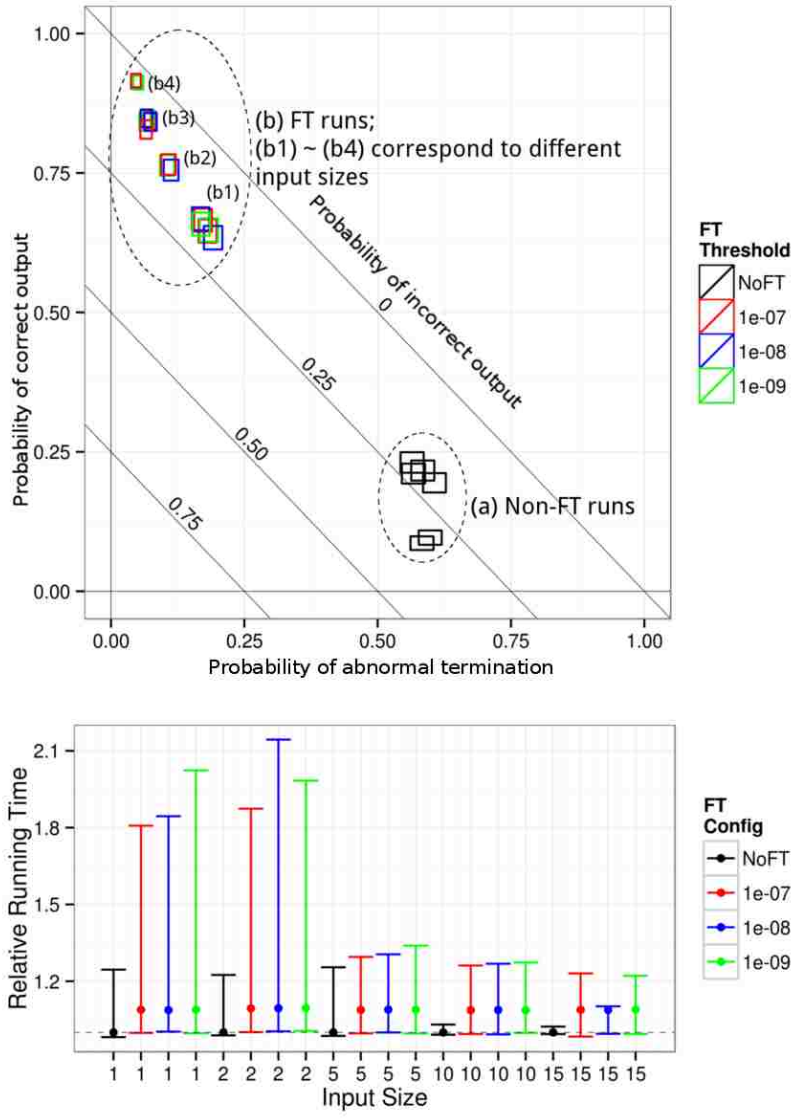

Figure 3.11: Fault Characteristics and Resilience Overhead of Lasso.

Figure 3.11 presents the characteristics and running time of the original and fault-resilient Lasso.

From the fault characteristics figure we can see the clusters that clearly reflect the

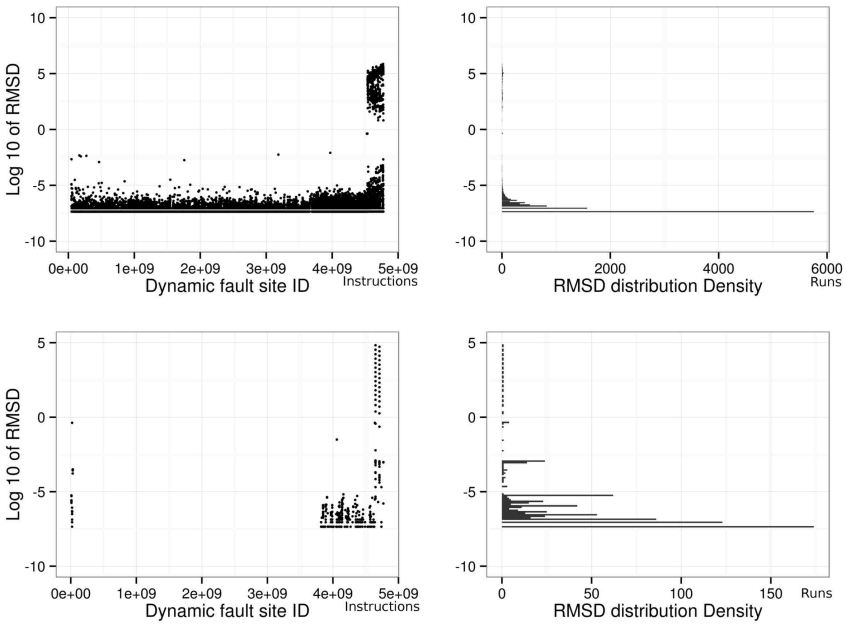

Figure 3.12: Detailed fault characteristics of LASSO, without (top) and with (bottom) fault resilience (Error checker threshold set to 1e-07).

effectiveness of the fault resilience techniques:

- The rectangles around the bottom-right cluster represent runs of non-fault-tolerant (Non-FT) Lasso ((a) in Figure 3.11). For these runs, the probability of abnormal termination is high and the probability of of producing correct results is low.

- The rectangles around the top-left cluster represent fault-tolerant Lasso ((b) in Figure 3.11). For those runs, the probability of abnormal terminations is low and the probability of correct outputs is high. Further, cluster (b) is divided into sub-clusters corresponding to input sizes ($b1 = \{40, 80\} \cdot 500$, $b2 = 200 \cdot 500$, $b3 = \{400, 600\} \cdot 500$, $b4 = 800 \cdot 500$). The temporal error graph of the input size {20, 500} are shown in Figure 3.12 as an example of how errors in the outputs are removed.

As input scales up, the overhead of the fault-resilient Lasso gradually decreases, just as the individual routines do. In the meantime, the probability of perfect runs increases while the probability of abnormal termination and running time overhead decreases, as is shown

in cluster (b). This is because when the input size gets larger, a greater fraction of time is spent in `cblas_dsyrk` (the Rank-K update). Therefore the overall program resilience characteristics would be shaped by the characteristics of this routine.

On the other hand, the error checker threshold used does not noticeably affect correctness or performance. The thresholds chosen (1e-07, 1e-08 and 1e-09) for the algorithmic checkers are all adequate for fixing incorrect runs. The rollback mechanisms are very useful for recovering from abnormal terminations.

3.5.2 DRC

Figure 3.13 shows the error characteristics and running time overhead of the original and the fault-tolerant DRC.

From the fault characteristics figure we can see the two clusters that clearly reflect the effectiveness of the fault resilience techniques:

- Runs of the non-fault-tolerant DRC are clustered around the center-left region ((a) in Figure 3.13), indicating smaller probabilities of abnormal termination and greater probabilities of producing perfect results.

- Runs of the fault-tolerant DRC are clustered around the top-left region ((b) in Figure 3.13). The choice of fault checker threshold does not separate the runs.

Overall the characteristics (in terms of the chance of abnormal termination, correct and incorrect answer) of DRC and Lasso are similar. However, the choice of fault checker threshold has a much more significant impact on performance on DRC than it has on LASSO. The performance overhead of a fault-tolerant DRC with error checker threshold 1e-08 is significantly greater than 1e-05. This is because the checker threshold 1e-08 is so tight that it considers results from many non-faulty runs to be incorrect, giving many false alarms. In fact, 1e-08 is below the precision of single-precision floating point representation which is roughly $2^{-23} \approx 1e-07$. From the detailed temporal error graph in Figure 3.14, many incorrect runs with RMSDs greater than 1e-06 are corrected.

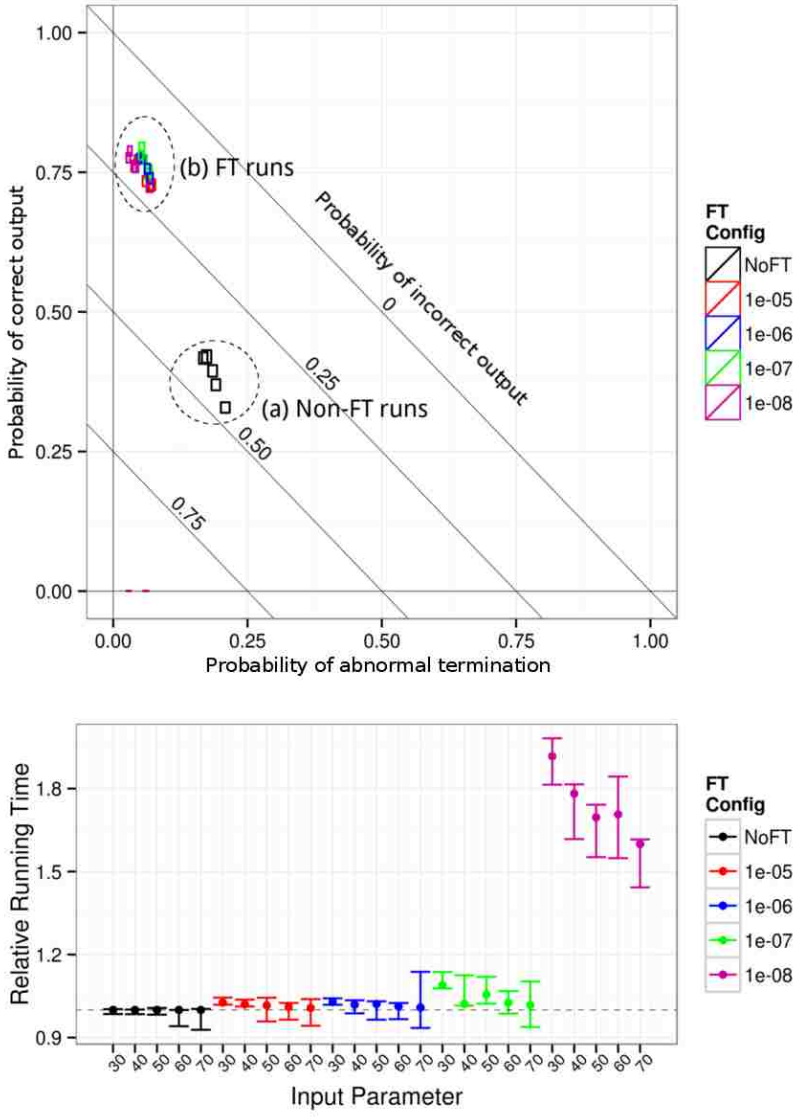

Figure 3.13: Fault Characteristics and Resilience Overhead of DRC

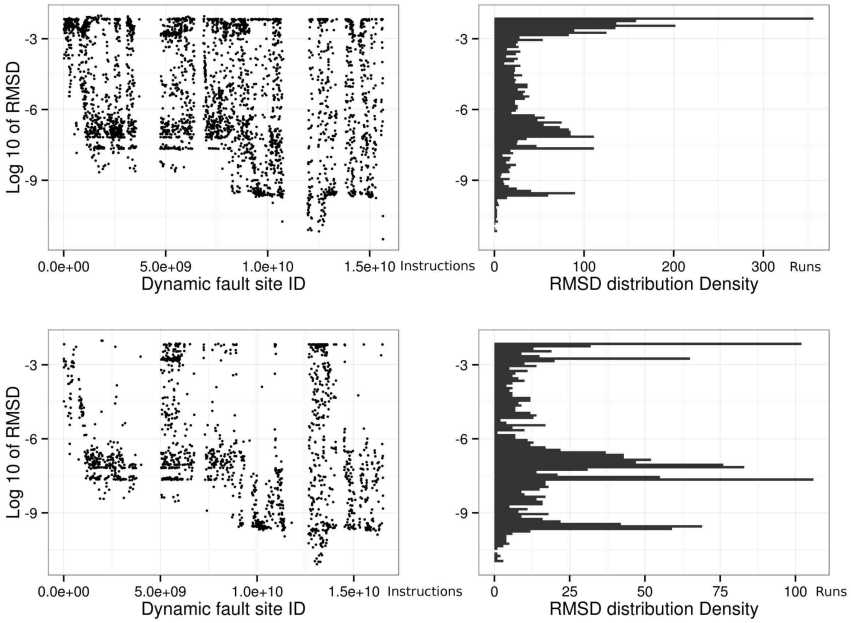

Figure 3.14: Detailed fault characteristics of the original (top) and fault-tolerant (bottom) DRC (Error checker threshold set to 1e-06).

3.5.3 Hattrick

The Hattrick program is very different from DRC and Lasso, and so are its characteristics.

Figure 3.15 shows the overall error characteristics of Hattrick. We can observe from the figure that:

- The bottom-right cluster (a) are runs of the non-fault-tolerant Hattrick. They have the highest probability of abnormal terminations and incorrect results.
- The top-right cluster (b) contains runs with *only pointer replication*. Its probability of abnormal termination is slightly reduced while the chance of perfect results is increased. From the figures it can be seen the degree of replication has only a slight influence on the outcomes (In contrast, the presence of replication has a great influence.)
- The bottom-left (c) cluster contains runs with *only checkpointing*. More runs complete but the proportion of perfect runs remained almost the same.

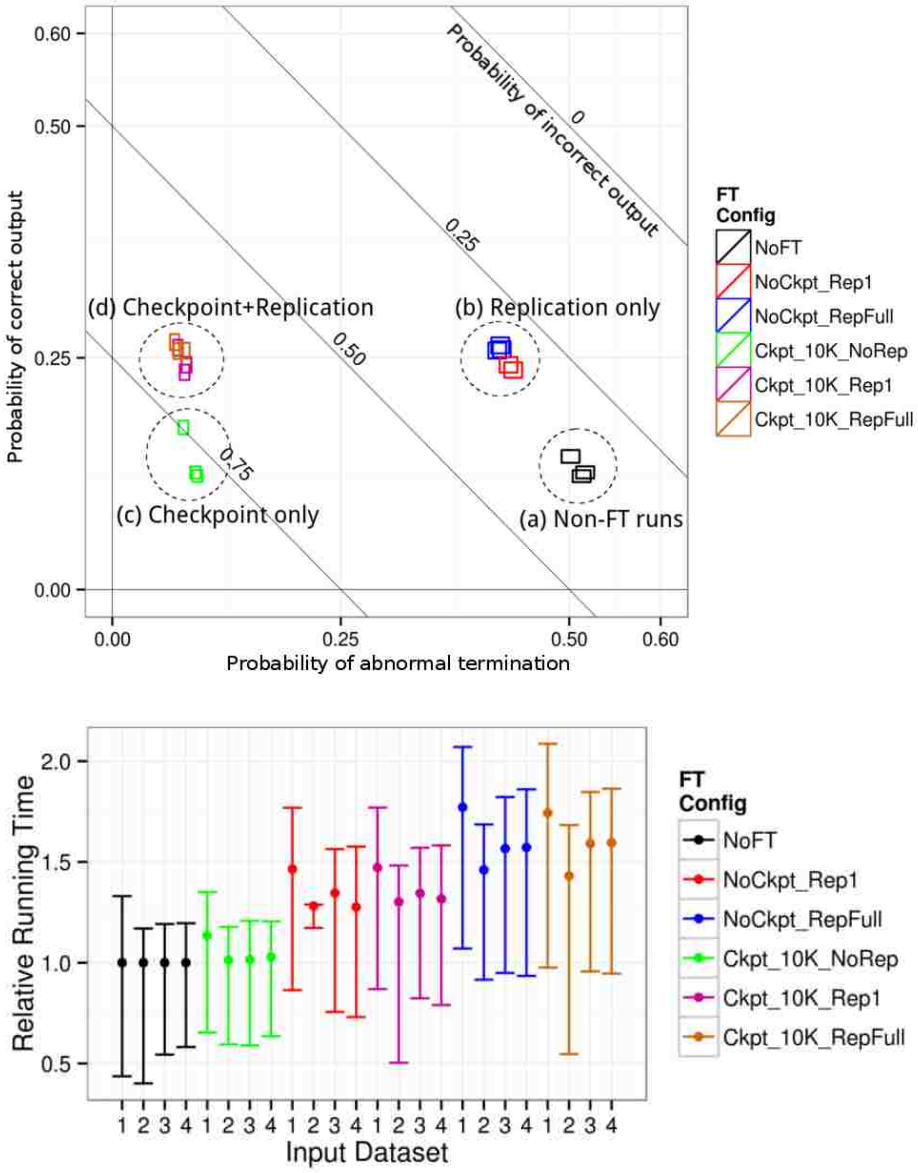

Figure 3.15: Fault Characteristics and Resilience Overhead of Hattrick.
(Inputs 1 to 4 correspond to P2T2090A15, P2T3090A15, P2T4090A15 and P3T2090A11 respectively)

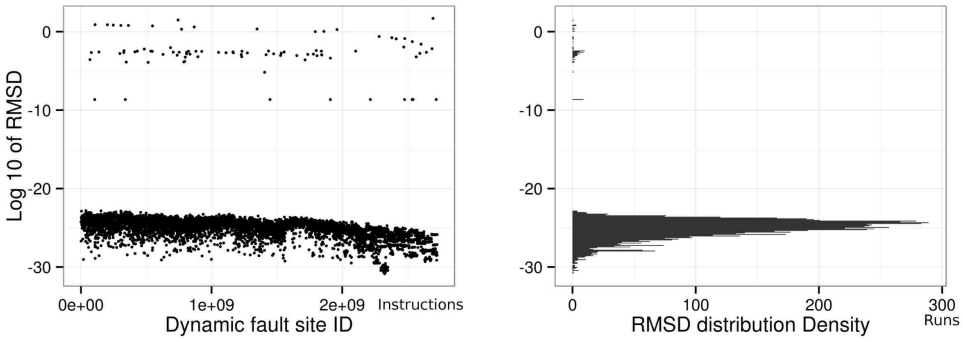

Figure 3.16: Detailed fault characteristics of Hattrick.
(The dynamic RMSD plots are almost visually identical for both fault-tolerant and non-fault-tolerant versions and only is shown here for brevity.)

- The top-left cluster (d) contains runs with *both checkpointing and replication*. They are as tolerant to abnormal termination as the cluster (c) and produce as many perfect results as cluster (b).

From the four clusters we can see that checkpointing and replication improve resilience in two different directions: checkpointing fixes abnormal terminations and "moves" a cluster towards the left. It does not increase the probability of correct results.

Figure 3.16 shows that a single bit-flip error is likely to cause Hattrick to produce a very small error in its outputs which is well below the user-specified accuracy bound of 1e-10 and 1e-15 (most runs have an RMSD of smaller than 1e-20). However, in rare cases, it can cause greater errors (the ones with RMSDs ranging between 1e-10 and 1). Either way, the errors persist through the program lifetime.

In comparison, replication effectively increases the probability of correct results, but the amount of increase is not affected by the degree of replication.

Performance-wise, checkpointing at the chosen interval of 10000 timesteps is almost free of overhead. Our study suggests that the checkpointing overhead becomes noticeable when the checkpointing interval is small enough (less than 100 time steps). A smaller interval does not significantly improve the probability of correct results. On the other hand, replication incurs greater performance overhead than checkpointing. In case of a high degree of replication, the overhead is even higher.

3.6 Determining Number of Experiments

The propagation of errors in programs is an inherently complex process, and it is difficult to model the error characteristics in the output analytically. However we can empirically characterize a program with experiments. To do so we need to answer the question of what is the number of fault injection experiments needed to accurately characterize a program. We build a statistical model in FaultTelescope to determine whether or not a number is large enough. This section shows the model-building process.

We start by observing the correlation between the Dynamic Fault Site ID and the error magnitude in incorrect results. To illustrate the correlation, the temporal error graphs in Figures 3.3, 3.12, 3.14 and 3.16 show that faults with Dynamic Fault Site IDs close to each other are likely to incur errors of similar magnitudes. A statistical model is built based on this observation. The model has three observation variables: (1) the Dynamic Fault Site ID, (2) the Static Fault Site ID (one Static Fault Site corresponds to one LLVM instruction in the program image) and (3) index of the flipped bit as observations. The model has two response variables: (1) program outcome and (2) the error magnitude in an incorrect run.

Given a combination of the observation variables, the model first categorizes a program run into one of three classes with the 1st-level categorization model: "Abnormal Termination", "Incorrect Result" and "Correct Result". Then, for the "Incorrect Result" runs it predicts the RMSD of the result error using the 2nd-level regression model. Its structure is illustrated in Figure 3.17.

The accuracy of the model is evaluated using two metrics:

- 1st-level categorization model: misclassification rate. Since we have 3 categories, the chance of a correct random guess is 33.3%, which means 66.67% misclassification rate. With the knowledge of the training set, the tree model should produce a misclassification rate smaller than 66.67%. A lower misclassification means a more accurate model.

- 2nd-level regression model: R-Squared, or $1 - \sum_{i=1}^{N} (\hat{y}_i - y_i)^2 \ / \ \sum_{i=1}^{N} (y_i - \bar{y})^2$, de-

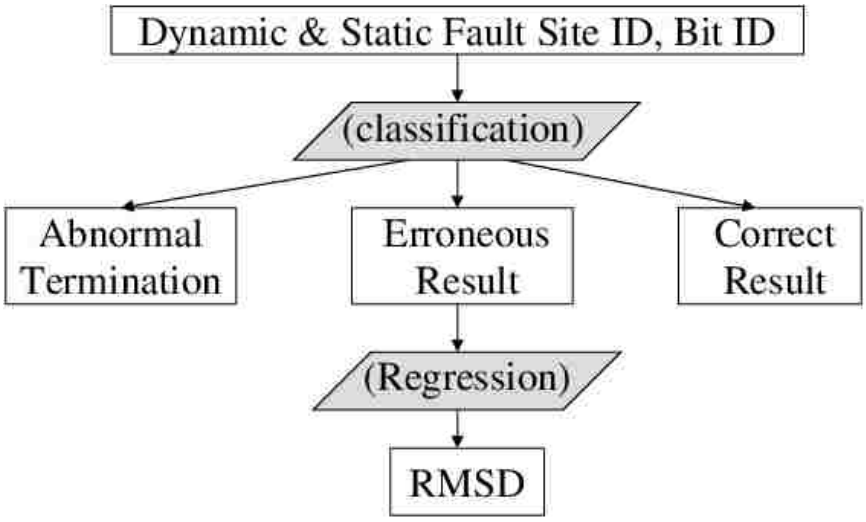

Figure 3.17: Structure of the FaultTelescope evaluation models.
Shaded procedures are where the tree model is applied.

scribes how much of the variance in the data the model is able to capture. (The R-square is not applicable to the 1st-level classification.) A greater R-Squared means a more accurate model.

FaultTelescope selects the number of experiments incrementally, by performing more and more experiments and observing the effect of the additional training data on the accuracy of the model. For a given sample, FaultTelescope performs a two-fold cross-validation for the model (train on half the data then predict for the other, and vice versa) to obtain the misclassification rate and R-squared. When FaultTelescope finds the sample size where the accuracy of the model stops improving as it increases, it stops the fault injection campaign since this number of samples is sufficient to build an accurate model of the relationship between the observations (Dynamic, Static Fault Site ID and Bit ID) and responses (Outcome and RMSD) considered by FaultTelescope. After the sample size is reached, additional improvements in accuracy can only come from adding more features into the models, not by running more experiments.

Figure 3.18 illustrates the procedure using experiments on the Matrix Vector Multiplication routine, executed on 500x500 matrices. As the number of fault injection experiments increases, we see that the misclassification rate drops while the R-square converges steadily

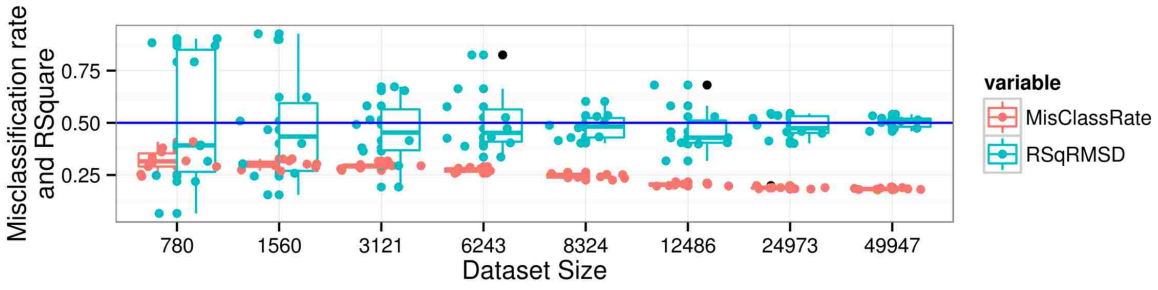

Figure 3.18: Trend of R-square and misclassification rate as dataset size grows. (A random guess = misclassification rate of 66.7%)

until they stabilize at a sample size of 49947 experiments. As the data shows, this sample size is sufficient for the purposes of FaultTelescope's visualization and is much smaller than the $\sim 1e9$ experiments required to fully explore the experimental space. This is the sample size chosen for this routine. FaultTelescope employs the same procedure for all other routines and programs.

3.7 Conclusion

We present FaultTelescope, a tool that supports developers in making programs resilient to errors induced by soft faults. FaultTelescope collects information about a program by carrying out fault injection campaigns. With the information, it then visualizes the relationship between the time a fault occurs and its effect on program results. With statistical analysis on the results, FaultTelescope helps developer draw conclusions on the program's fault characteristics and the effectiveness of the fault resilience techniques with a high confidence.

We demonstrated the use of FaultTelescope for the Lasso, DRC and Hattrick programs. The results suggest that an HPC numerical program developer should take the following into consideration when writing fault-resilient programs:

- Algorithm-specific error checkers are effective at detecting incorrect program results, as illustrated in our experiments with MVM, SYRK and FFT. During the process the developer should realize that the precision limit of the checker may make it difficult to correct all results. Example of this is the checker for FFT.

- The probability of certain routines producing correct results can be significantly improved if they are protected from abnormal terminations. Examples are the Cholesky Decomposition and the Runge-Kutta integrator. Their outputs are not sensitive to faults compared to other linear algebra routines but are more vulnerable to abnormal termination.

- The RK4 Integrator routine demonstrates characteristics significantly different from those of linear algebra, FFT and FIR routines. It also requires different resilience techniques, namely replication and checkpointing. It's advisable to apply checkpointing first due to its effectiveness and low overhead. Replication trades performance for enhanced accuracy.

Actionable conclusions and tradeoffs in many other programs can be discovered with the FaultTelescope workflow in a similar fashion. We believe FaultTelescope can benefit the production of fault resilient numerical programs.

Chapter 4
Soft Error Resilience in Big Data Kernels through Modular Analysis

4.1 Overview

As mentioned in the Background, this chapter discusses the tool ErrorSight which assists software developers in writing fault-resilient big data applications. This tool furthers FaultTelescope discussed in Chapter 3 through efficient generation of error profiles leveraging the predictive power of the Boosted Regression Tree model. We use four big data kernels to illustrate the modular analysis mechanism of ErrorSight and show its usefulness in the development of numerical fault-resilience in big data.

Similar to performance analysis tools that quantify the resource utilization of various application regions, ErrorSight helps developers understand the impact of soft faults on their application state and how their impacts flow through application logic. Developers can use this information to

- Focus their efforts on code regions and data structures the errors in which have the most significant impact on application results, and
- Understand how the errors propagate as the program runs,

and produce fault-resilient software more efficiently.

ErrorSight begins by running fault injection plans, where the program is executed a large number of times (Table 4.4) with one error modelling physical soft faults injected into its program state each time. The errors are injected into registers and can propagate through expressions and memory operations as the program continues. By observing the flow of these errors through application state and their impact on application output, ErrorSight creates a profile of the errors that have the most significant impact on application output and how they propagate to the output. As the data begin to accumulate, a statistical

model is trained with the data in hand, which is then cross-validated. This model captures the error propagation patterns and can predict the magnitude of errors down through the propagation chain, thereby saving the cost of fault injection experiments needed for an accurate characterization of the impact of soft errors on this program. The results are then presented to the user in an intuitive way, informing the user of the necessary changes in the software needed to improve fault resilience.

While fault injection is used ubiquitously to quantify application resilience properties, ErrorSight incorporates novel capabilities specifically designed to improve developers' ability to make their software resilient in addition to evaluate resilience after the fact. First, ErrorSight quantifies the impact of errors on application state in terms of high-level concepts using developer-specified distance metrics to measure the difference between a given data structure in a fault injected run and the same data structure at the same execution point in a reference application run. For example, errors in numerical vectors may be quantified using the root mean square deviation metric, while errors in strings may be measured using the edit distance metric. This enables developers to reason about the impact of errors at the same level of abstraction they use as part of their regular development efforts, which improves their productivity. Second, ErrorSight tracks the propagation of errors through each fault injected execution to make it possible for developers to query where the errors that most critically affect application outputs originate from, and how they flow through application logic. This enables developers to design resilience techniques that detect the most critical error types (i) soon after they occur, (ii) at application locations that are highly sensitive to errors (e.g. control logic), or (iii) at application locations where errors may be easily identified (e.g. where critical errors induce usually large values in some application variable). Finally, ErrorSight statistically models the propagation of errors from the inputs to the outputs of individual code regions to (i) enable application developers to understand application resilience properties in a modular fashion (e.g. important for library writers and developers of large applications) and (ii) reduce the number of fault injection experiments

needed to comprehensively analyze an applications resilience properties. To ensure that developers can make well-grounded conclusions based on these models ErrorSight reports confidence intervals for all model predictions.

Overall, the main contribution in this chapter are:

- We developed the tool ErrorSight, which provides useful guidance for the user in writing fault-resilient software,
- We developed an algorithm that can substantially reduce the cost of fault injection experiments,
- We observed three kinds of error propagation patterns, namely "maintaining", "shrinking" and "magnifying", and
- We demonstrated the usage of ErrorSight and showed how to apply fault resilience to one big data kernel.

The chapter is organized in a way that follows the workflow of ErrorSight. Section 4.3 describes the design, from the error model to error propagation and the error characterization algorithm. Section 4.4 discusses the Big Data kernels and the driver programs used in this chapter and gives an analytical analysis of the the error propagation patterns that will be corroborated with results in Section 4.5. We complete the chapter by showing how to add fault resilience using ErrorSight in Section 4.5.5.

4.2 Related Works

ErrorSight complements the broad range of existing work done by software resilience community. It can take advantage of existing fault injection tools such as NFTAPE [42] and KULFI [2], as well as recent approaches such as Relyzer [23] that leverage redundancy in the way different errors propagate to reduce the number of fault injections needed to comprehensively understand the impact of errors on applications.

ErrorSight supports developer efforts to design and deploy resilience mechanisms. This includes the use of generic mechanisms such as redundancy [8] and OS segmentation

violation detection, as well as application-level techniques [16, 15], both of which require tools to quantify the flow of errors though key application sub-routines (e.g. GMRES solver for Elliot et al [16] and LU factorization for Du et al [15]).

Finally, ErrorSight can be incorporated into emerging resilience-aware programming models such as Containment Domains [10], which enables application developers to organize their resilience mechanisms hierarchically. In this context ErrorSight can serve the same role as debuggers and performance analyzers do in traditional programming models.

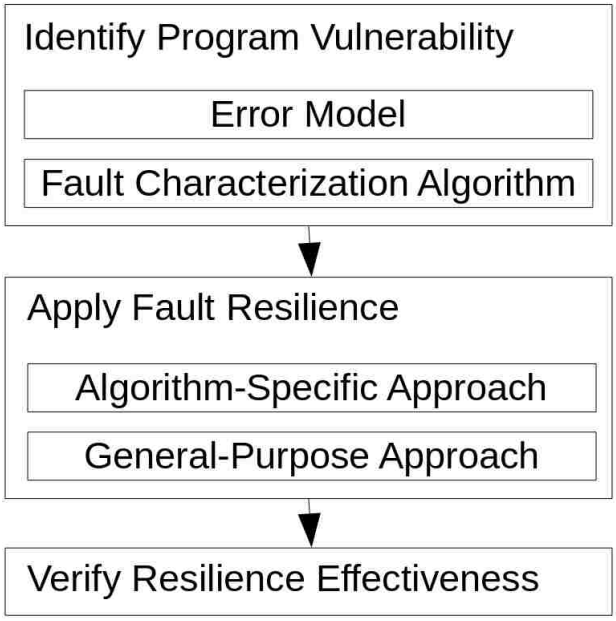

Figure 4.1: Workflow of ErrorSight.

4.3 Design of ErrorSight

ErrorSight performs fault injection campaigns, tracks the execution of a program and log its program states, builds a non-parametric tree-based predictive model named Boosted Regression Tree to predict the error propagation in the program, and then obtains the error characteristics of an application at the source code level. This section introduces each of these steps in order, as is illustrated in Figure 4.1.

4.3.1 Error Model

We use the KULFI [2] error injection framework throughout the experiments, which is based on LLVM [25]. It uses a static single assignment (SSA) compilation strategy which is capable of supporting arbitrary programming languages. The source code is compiled into LLVM byte code representing the LLVM instruction set. As a result, there is a one-to-many mapping between the entities in the source code (statements, expressions) and the instructions, as is illustrated in Figure 4.2.

Source Code	ID	Instruction (Static Fault Site)
`120: for (i = 0; i < nlocalverts; ++i) pred[i] = -1`	4	`%51 = load i64* %i, align 8`
	5	`%52 = load i64** %3, align 8`
	6	`%53 = getelementptr inbounds i64 %52, i64 %51`
	7	`store i64 -1, i64* %53, align 8`

Figure 4.2: Example of static fault site to source code mapping.

The SSA semantics determine that instructions producing outputs write to at most one register. We consider bit flips in these output registers and define one instruction in the program image to be a *static fault site*. A dynamic instance of a static fault site is defined as a *dynamic fault site*. There is a one-to-many mapping between static and dynamic fault sites.

For each run in the fault injection campaign, a bit in a dynamic fault site is chosen for fault injection. We only inject one bit flip per run because multi-bit flip events are relatively rare.

```
Run Dynamic Bit Static    Is  Num     Error
ID     FSID  ID   FSID  Init Iter    Metric
3  10054576   1     78     1    2 -4.667642
4  10054576   1     78     0    3 -6.372848
5  10054576   1     78     0    4 -7.477853
6  10054576   1     78     0    5 -8.271486
7  10054576   1     78     0    6 -9.056564
```

Figure 4.3: Example trace of KMeans. A bitflip is injected at the 10054576'th dynamic fault site at iteration 2 and propagates through iteration 6.

To quantify and track the propagation of errors, the intermediate results and data structures are compared with those of a fault-free run. For the applications in this chapter, error metrics defined in Table 4.3 are measured for the entities of interest at run time. Figure 4.3 is an example trace form the KMeans program.

4.3.2 Modular Analysis of a Program

The modular analysis of a program is the theoretical basis the regression model is built upon. A program consists of "entities" including values and expressions, operated on by

```
1   for (int iter=0; iter<10; iter++) {
2      for (int i=0; i<10; i++) {
3         b[i] = a[i] + a[i+1] + f(b[i])
4      }
5      for (int i=0; i<10; i++) {
6         a[i] = b[i];
7      }
8   }
```

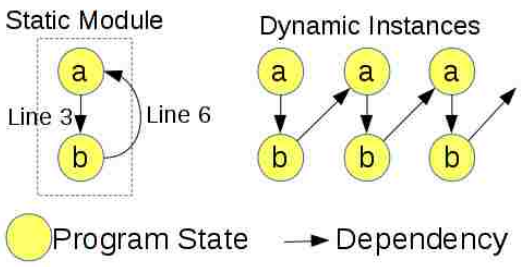

Table 4.1: Example of a module.

"modules" which read and store data and perform computation. A modular structure of an application is a graph consisting of nodes representing "entities" and arcs representing "routines." The graph may be considered a coarse version of the data dependency graphs generated by a compiler. To put this into perspective, consider the program in Figure 4.1:

We consider the loop body to be a module consisting of two entities, arrays `a` and `b`. In line 3 `b` is updated using values of `a`. This line corresponds to the dependency arc flowing from `a` to `b` in Figure 4.1. Similarly line 6 corresponds to the arc flowing from `b` to `a`. The dependency graph may be unrolled with entities `a` and `b` duplicated for each iteration. In the unrolled form, self-loops in the graph are to be replaced by edges between incarnations of the nodes in different iterations.

The effect the arcs have on the errors are captured by the regression model, described in Section 4.3.5.

4.3.3 Error Propagation

On the microscopic level, error propagation refers to the incident of an output affected by bit flip is used as the input of other instructions.

On a higher level, we consider the following two cases in the model used in ErrorSight:

- 1. Propagation between entities: An error originates from a bit flip, and then propagates from one entity to another entity following the arcs between them.
- 2. Propagation between time steps: Propagation between entities repeats as the main loop in the program advances. Example is the propagation from `b` to the `a` in the

next iteration on line 6 in Figure 4.1.

The two types of conflicts are complementary which can be used to model most data flow found in iterative applications. By including relevant inputs/outputs of modules and the time step into the set of dependent variables, we can build predictive models to predict either type of error propagation. We consider two types of models, named Model 1 and Model 2, as described in Figure 4.4:

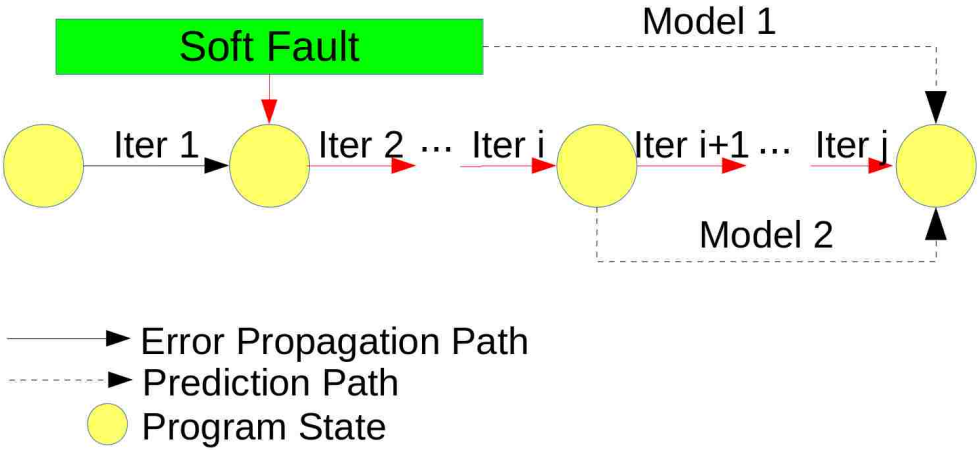

Figure 4.4: Prediction models on the error propagation path.

The details of the models are described in Table 4.2. Both models are realized using the Boosted Regression Tree model described in Section 4.3.5.

Model	**Input**	**Output**
#1	Static/Dynamic FSID, Bit ID	Error Metric
#2	Error Metric at iteration i	Error Metric at iteration j

Model	**Algorithms**
#1	Regression Tree
#2	Linear Regression, Segmented Linear Regression or Regression Tree

Table 4.2: Details of Model 1 and Model 2.

The models correspond to two cases in which we need to characterize the error of an application at iteration j, which may be affected by a bit flip that occurred in iterations 0 through i, with different input to the model in each case:

- Model 1 relates the information of a bit flip to the error in the program state. It

requires the program be run to iteration j. This corresponds to Algorithm 1.

- Model 2 relates the error in program states in iteration i and iteration j. It needs to observe the program states at iterations i and j. This step is involved in Algorithm 2.

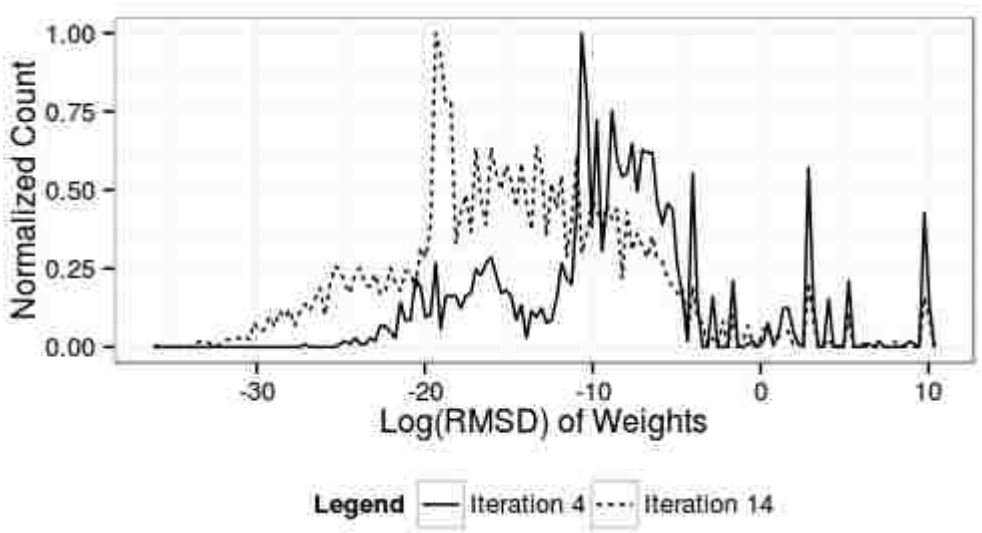

Figure 4.5: Histogram of root mean square distances in the page weights in PageRank in iterations 4 and 14.

For both models, the characterization output is an aggregate of error metrics, represented using histograms which conveniently represents the fault characteristics of the whole application. Figure 4.5 shows how the distribution of the error metric RMSD (Root Mean Square Distance) in the page weights of Pagerank (described in Section 4.4) changes between iterations 4 and 14. We use the Earth Mover Distance [37] (EMD) to quantify the difference between two histograms. As the name suggests, a greater distance means a greater difference in the probability masses. In Section 4.5.3 it will be used to quantify the goodness of prediction.

The two models are used to construct fault characterization algorithms.

4.3.4 Fault Characterization Algorithms

ErrorSight uses an efficient inter-iteration algorithm for fault characterization. It is based on the baseline fault characterization algorithm.

- **Baseline Fault Characterization Algorithm**

Algorithm 1: Baseline Fault Characterization Algorithm

Input: Program p
Result: Empirical Error distribution at the end of iteration N

1 $rsq_{prev} \leftarrow 0$;
2 $rsq \leftarrow 0$;
3 $NF \leftarrow Number\ of\ dynamic\ fault\ sites$;
4 $num\_inj \leftarrow 1000$;
5 $errors \leftarrow \emptyset$;
6 **while** $rsq - rsq_{prev} > \epsilon$ **do**
7 **for** $fsidin(0toNFstep\frac{NF}{num\_inj})$ **do**
8 $p_{fsid} \leftarrow p\ with\ bit\ flip\ at\ dynamic\ fault\ site\ \#fsid$;
9 run p_{fsid} until completion ;
10 $errors \leftarrow errors \cup error(p_{fsid})$;
11 **end**
12 $RT \leftarrow Model1(errors)$;
13 $rsq_{prev} \leftarrow rsq$;
14 $rsq \leftarrow CrossValidate(RT, errors)$;
15 $num\_inj \leftarrow 2 \cdot num\_inj$;
16 **end**
17 **return** $(errors)$

The baseline fault injection algorithm is listed in Algorithm 1. This algorithm incrementally increases the number of fault injection experiments until the R-Squared value measured from the validation step (Line 14) suggests the sample size is large enough for an accurate model. Every fault injection run has to be run to completion in order to obtain the errors (Line 9). The number of dynamic fault sites is not directly related to the number of experiments needed.

Cross-validation is achieved by splitting the collected errors into a training set used to train the model and a test set use to evaluate the R-squared value. The R-squared value quantifies how much the model can explain the uncertainty of the real underlying distribution of errors.

Most of the cost in Algorithm 1 is incurred by Line 9 (running program to completion after fault injection), which Algorithm 2 seeks to improve.

Algorithm 2: Inter-Iteration Efficient Characterization Algorithm

Input: Program p; Fault site counts at iteration i, NF_i $(i \in (0, 1, 2, ..., N))$
Result: Error distribution at the end of iteration N

1 $NF \leftarrow Number\ of\ dynamic\ fault\ sites$;
2 $errors_1, errors_2, ..., errors_N \leftarrow \emptyset$;
3 $errors \leftarrow \emptyset$;
4 **for** $i = 0, 1, ..., N$ **do**
5 $\quad num\_inj \leftarrow 100$; $rsq_{prev} \leftarrow 0$; $rsq \leftarrow 0$; $runs_i \leftarrow \emptyset$;
6 $\quad$ **while** $rsq - rsq_{prev} > \epsilon$ **do**
7 $\quad\quad$ **for** $fsid = (NF_{iter}\ to\ NF_{iter+1}\ step\ \frac{NF_{iter+1} - NF_{iter}}{num\_inj})$ **do**
8 $\quad\quad\quad p_{fsid} \leftarrow Program\ with\ bit\ flip\ inserted\ at\ fsid$;
9 $\quad\quad\quad runs_i \leftarrow runs_i \cup p_{fsid}$ run p_{fsid} until iteration i;
10 $\quad\quad\quad errors_i \leftarrow errors_i \cup error(p_{fsid})$;
11 $\quad\quad$ **end**
12 $\quad\quad m1 \leftarrow Model1(errors_i)$;
13 $\quad\quad rsq_{prev} \leftarrow rsq$;
14 $\quad\quad rsq \leftarrow CrossValidate(m1, errors_i)$;
15 $\quad\quad num\_inj \leftarrow 2 \cdot num\_inj$;
16 $\quad$ **end**
17 $\quad rsq2_{prev} \leftarrow 0$;
18 $\quad rsq2 \leftarrow 0$;
19 $\quad n2 \leftarrow 1$;
20 $\quad$ **while** $rsq2 - rsq_{prev} > \epsilon$ **do**
21 $\quad\quad newruns \leftarrow sample(runs_i, 2 \cdot n2)$;
22 $\quad\quad subset \leftarrow sample(newruns, n2)$;
23 $\quad\quad test \leftarrow newruns \setminus subset$; $m2 \leftarrow Model2(error(subset))$;
24 $\quad\quad$ run $test$ to completion;
25 $\quad\quad rsq2_{prev} \leftarrow rsq2$;
26 $\quad\quad rsq2 \leftarrow Validate(m2, test)$;
27 $\quad\quad n2 \leftarrow 2 \cdot n2$;
28 $\quad$ **end**
29 $\quad errors \leftarrow errors \cup errors_i$;
30 **end**
31 **return** $(errors)$

- **The Efficient Algorithm**

In Algorithm 2, errors will be collected for every iteration (Line 10), just like in Algorithm 1. Instead of running until completion, we only selectively run the program until the end of the iteration where the error gets injected.

We confirm the number of experiments with the same validation procedure and incre-

mental increase of sample size (Lines 13 through 15) as in Algorithm 1. A subset of the instances will be run to completion to build Model 2 (Line 21). Another subset will be run to completion to serve as the validation set (Line 22). The same validation procedure is also applied to Model 2 (Lines 25 through 27) and the subset of instances run to completion is incrementally increased. When the trained Model 2 has become accurate enough we project all the errors to the end of the program. By doing this we save the cost of having to run the rest of the instances until completion. The same procedure is repeated for all the iterations to obtain the fault characteristics of the program.

4.3.5 Boosted Regression Trees

We propose to use the Boosted Regression Tree method for predicting the distribution of errors at the output of modules and the propagation of modules. Being an aggregate technique that aims to providing good prediction quality by combining the predictive power of numerous weaker predictors, the Boosted Regression Tree is based on the classic Classification And Regression Tree (CART) [6] and Boosting, which builds and combines a collection of trees by penalizing erroneous predictions and preserving correct predictions.

CART is a recursive binary partitioning algorithm and is an alternative to traditional parametric models for regression problems. The term "binary" indicates it has the power to split the input space into two regions and models the response by a constant for each region. The region may be further subdivided to give a better fit of the input space. To illustrate with Figure 4.6, the data set with the Dynamic Fault Site ID as the independent variable and the error metric in question (RMSD is used as an example) as response may be divided into 7 non-overlapping regions by a decision tree.

The detailed BRT algorithm used in our chapter is described in Algorithm 3. In the algorithm, $I(\cdot)$ is an indicator function which returns 1 if the condition is satisfied or otherwise 0. The v, named as "shrinkage parameter", controls the learning rate of the BRT. In this study we use the value of 0.1 which results in faster learning speed and better prediction accuracy.

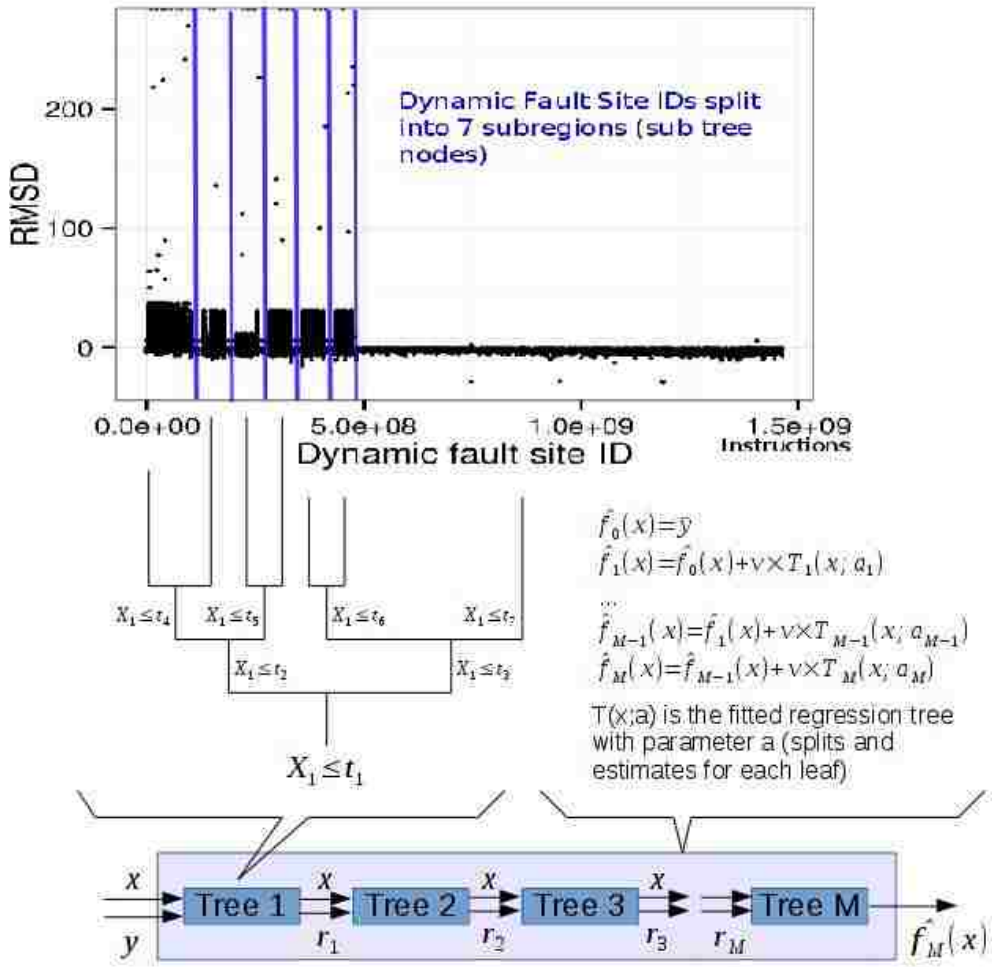

Figure 4.6: Example of one iteration of the BRT training process.

From the user's point of view, the BRT is capable of capturing complex, multi-variate functions without the knowledge of the underlying distributions. Such knowledge is not required by BRT. Also, BRT is unaffected by outliers.

The BRT is able to determine the relative importance of variables. The importance is measured based on the number of times a variable is selected for splitting, weighed by the squared improvement to the model as a result of each split, and then average over all trees. A higher number indicates greater importance.

In this chapter, we use the BRT to predict the propagation of errors after a certain number of time steps. The process is described in Section 4.3.3.

Algorithm 3: BRT algorithm used in this chapter

Initialize $\hat{f}_0(x_i) = \bar{y}$, where $\bar{y}$ is the average for $\{y_i\}$;
for $m = 1, 2, ..., M$ **do**
    Compute the current residuals $r_{im} = y_i - \hat{f}_{m-1}(x_i)$, $i = 1, ..., n$;
    Partition the input space into H disjoint regions $\{R_{hm}\}_{h=1}^{H}$ based on $\{r_{im}, x_i\}_{i=1}^{n}$;
    For each region, compute the constant fit $\gamma_{hm} = argmin_\gamma \sum (r_{im} - \gamma)^2$;
    Update the fitted model $\hat{f}_m(x) = \hat{f}_{m-1}(x) + v \times \gamma_{hm} I(x \in R_h m)$;
end

4.4 The Big Data Kernels

In this section we describe the big data kernels and do a simple analysis of the error propagation characteristics based on our understanding of the underlying algorithms. The patterns are captured by the statistical model described in Section 4.3.5.

For the benchmarks, The PageRank and Breadth-First-Search (BFS) algorithms are Big Data algorithms by themselves; the classic K-Means algorithm may serve as an unsupervised clustering algorithm on its own and can also serve as a preprocessing step in more complex learning tasks; the Stable Fluid Solver is based on linear solvers that are also used in a variety of programs.

4.4.1 PageRank

- **Error Propagation through the Pagerank Loop**

The PageRank algorithm computes the importance for each webpage in a network, which is expressed as a graph. The PageRank loop can be expressed as a linear system $I = GI$. The G matrix is the "Google Matrix" which is derived from the graph topology. The I vector is the importance ranking vector which the algorithm tries to find out.

The algorithm used is a modified Power Method [3], which computes $I \leftarrow GI$ in every iteration.

The algorithm has the following desirable properties:

- As the algorithm makes progress the I will always converge.
- I converges to a value independent of the initial value of I.

- Information of the graph will not get lost (i.e. I will not be a zero vector.)

The convergence property can be explained with eigenvalues. Assume a vector I_0 can be expressed as the sum of the eigenvectors of G, that is,

$$I_0 = c_1 v_1 + c_2 v_2 + ... + c_N v_N$$

Applying the definition of eigenvectors ($Gv_n = \lambda_n v_n$), we have:

$$I_k = G^k I_0 = c_1 \lambda_1^k v_1 + c_2 \lambda_2^k v_2 + ... + c_N \lambda_N^k v_N$$

Note that the eigenvalues λ_n are sorted by their magnitudes in descending order. One characteristic of the Google Matrix is $|\lambda_1| = 1$ and $|\lambda_2| = 0.85$ and the magnitude of all other eigenvalues are smaller than 0.85. This means I_k converges to $c_1 v_1$. After normalization, it becomes v_1.

When an error is injected it would only affect the convergence speed of the algorithm rather than the destination of convergence, unless I or the graph data is corrupted. The modular structure of PageRank is shown in Figure 4.7.

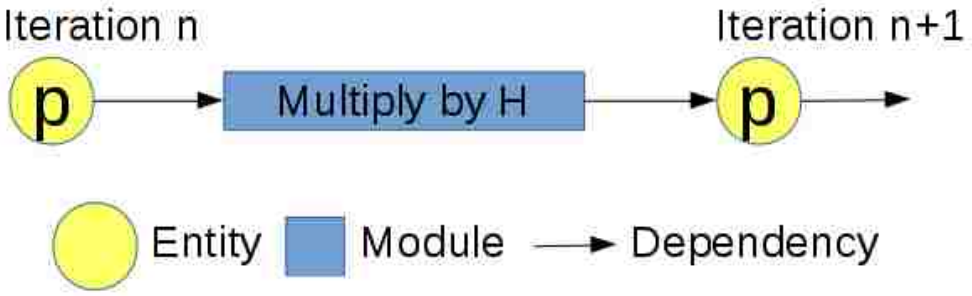

Figure 4.7: Modular structure of Pagerank.

4.4.2 K-Means

The K-Means is an unsupervised and iterative clustering algorithm. In this chapter we used the K-Means implementation from [29]. The algorithm finds the K clusters by minimizing the sum of Indra-cluster distance $S = \sum_{i=1}^{k} \sum_{x \in S_i} ||x - \mu_i||^2$. The algorithm

consists of a loop of `recomputation` and `reassignment` routines which update the cluster centers (μ_i) and cluster memberships (S_i), as is shown in Figure 4.8.

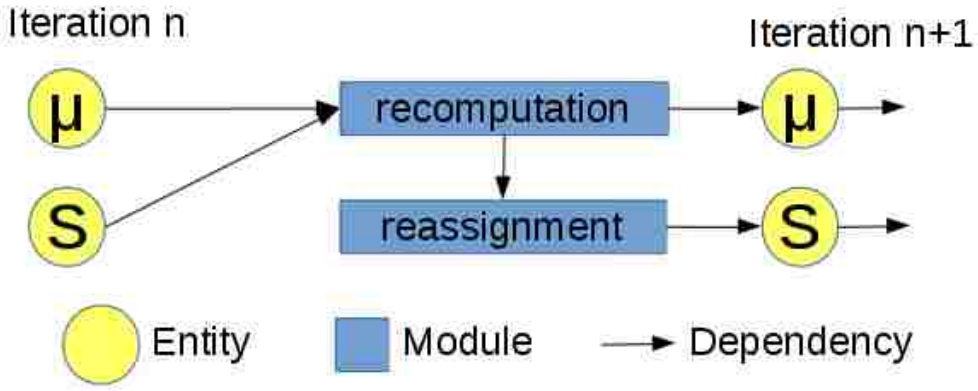

Figure 4.8: Modular structure of K-Means.

We quantify the correctness of two clustering results with the quantity Error Factor. Given two clustering results S_1 and S_2, the Error Factor is defined as $EF = 1 - \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} [[c_1(i) = c_1(j)] = [c_2(i) = c_2(j)]] \; / \; k(k+1)$, where $c_1(x)$ and $c_2(x)$ denote the cluster x belongs to under clustering S_1 and S_2. The $[\cdot]$ is a boolean function which evaluates to 1 when the condition is satisfied and 0 otherwise. The nominator traverses through all pairs. If the two pairs belong to the same cluster in both S_1 and S_2, it is incremented by 1. The denominator is the total number of pairs. If S_1 and S_2 are identical clusters, EF will be zero. Note that the clusters need only contain the same data points but not the same cluster ID. For example, cluster IDs [1,1,2,2] and [2,2,1,1] assigned to four data points are identical because the first two points belong to the same cluster and so do the last two points.

With Error Factor, we can quantitatively compare the results from two runs. We also have the foundations to analyze the correlation between the error in the cluster centers and the Error Factor.

- **Error Propagation through the Reassignment Step**

  In the `reassignment` step, data points are assigned to clusters whose centers are the closest as measured by Euclidean distance. Assume one cluster center is perturbed by

a small amount $\vec{e}$ towards the direction perpendicular to one of the boundaries of its Voronoi cell, that boundary would move by an amount of $\frac{1}{2}\vec{e}$, because the boundary is the perpendicular bisector of the line segment connecting to the centers of the adjacent cells.

Assuming the input dimensionality is N, the moving of the boundary sweeps through an volume in the $(N-1)$ dimensional space of $L|\vec{e}|$, where L is the area/length of the boundary.

Assume the data points are evenly distributed in the regions with a probability p, the swept volume contains $Lp|\vec{e}|$ data points. The cluster membership of these points will be altered. This would cause the nominator in the Error Factor to decrease by $Lp|\vec{e}|(n-1)$, which is a linear function of $|\vec{e}|$. Thus we expect EF to be the linear function of the square root of the L2 norm when the error is small.

4.4.3 Stable Fluid Simulation

We implemented a 2D fluid simulation program based on the three algorithms (Jacobi, Gauss-Seidel and Conjugate Gradient) described in [19]. The solvers update the elements of a grid repeatedly by solving the Navier-Stokes (NS) equations $\frac{\partial \vec{u}}{\partial t} = -(\vec{u}\cdot\nabla)u + v\nabla^2 u + f$ and $\frac{\partial d}{\partial t} = -(u\cdot\nabla)d + k\nabla^2\rho + S$, where $\vec{u}$ represents the velocity field, and the d represents the density field. Since we simulate fluid in 2 dimensions, u may be written as (u, v) where u and v represent the velocity along the X and Y axis respectively. In this chapter, we consider them two entities because each of them goes through the routines listed below.

The Fluid Simulation program operates on the discretized form of the NS equation. It consists the following routines as illustrated in Figure 4.9:

- `Diffuse`, which solves the first term in the NS equation. It solves a sparse linear system with elements scattering on a band spanning the main diagonal line. All elements except the ones on the band are zeroes. This routine is applied on both the density (d) and velocity (u and v) fields.

- `Advect`, which moves the density through a static velocity field and solves the second

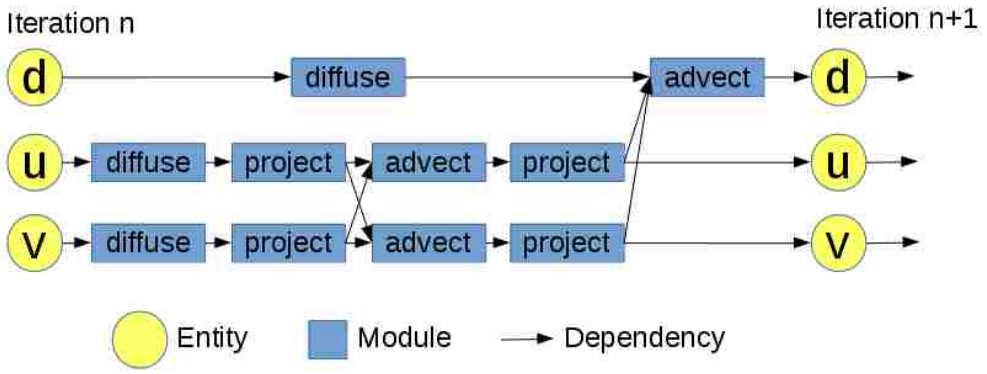

Figure 4.9: Modular structure of Stable Fluid Simulation.

term in the NS equation.

- `Project`, which subtracts the gradient field from the current velocities. It solves another sparse linear system which is similar to the one in `Diffuse`.

For the `Diffuse` and `Project` routines, one of the Jacobi, Gauss-Seidel and Conjugate Gradient solvers may be used.

The solutions produced by the Jacobi and Gauss-Seidel solvers are nearly identical. The solution produced by the Conjugate Gradient is slightly different, with a L2-norm of around 1e-07. This will affect the characteristics of the initial errors.

We start by discussing the `Advect` routine because it is a good example of how errors can propagate between entities.

- **Error Propagation through the Advect Routine**

The `Advect` routine propagates the errors from the u and v arrays into the d array and exhibits an easily understandable error propagation pattern. This is because of the this routine computes for each cell the density mass which ends up at each of them at the end of a time step.

For example, the center of the cell $(10, 10)$ is $(9.5, 9.5)$. Assume the velocity field at this cell is $(1, 1)$ and we use a time step of 1. The `Advect` routine traces the center backwards to $(9.5, 9.5) - (1, 1) \cdot 1 = (8.5, 8.5)$, adds up the density at the cells surrounding this point $((8, 8), (8, 9), (9, 8), (9, 9))$ weighted by their distance to $(8.5, 8.5)$. If the velocity at this field

contains an error $\vec{e}$, the back-tracked point would have become $(9.5, 9.5) - ((1, 1) + \vec{e}) \cdot 1$. It can be seen that if the magnitude of the error $|\vec{e}|$ is small it will only affect the weights of the cells surrounding the source cells. Since the weights are a linear function of $|\vec{e}|$, we can expect the error in $\vec{d}$ to be a linear function of $|\vec{e}|$ as well.

If $|\vec{e}|$ is larger it will alter the source cells or even make them go out-of-boundary. In this case the error would not be linear to $|\vec{e}|$. Depending on the way boundary conditions are enforced, the erroneous subscripts may be clamped at the boundary of the field.

- **Error Propagation through the Linear Solvers**

We can view the linear solvers in the `Diffuse` and `Project` routines as solving the equation $Ax_i = x_{i-1}$, where x could be substituted with d, u or v and A is the sparse linear system. When an error e is added to the input x_{i-1} we are essentially solving $A(x_i + e_i) = x_{i-1}$. This means the system has become the sum of two systems, whose starting value at time step $i - 1$ are x_{i-1} and e.

The characteristics of the linear solver is not relevant to how e_i would change unless it is smaller than the precision bound of the solver. The characteristics of the solver does affect the initial error e, if the bit flip occurs during its execution.

4.4.4 Breadth First Search (BFS)

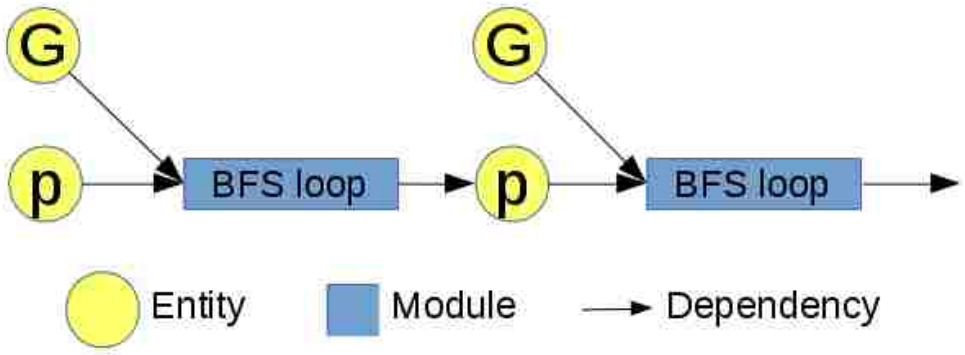

Figure 4.10: Modular structure of BFS.

The BFS program is a reference implementation of the Graph500 benchmark [33]. It is divided into two phases. In the first phase the program generates a graph, and in

the second phase a series of bread-first-search from randomly-chosen starting nodes (the roots) are performed on the graph. The second phase is completed by building the BFS tree, which is represented with a precedence list. In each iteration of the tree-building process, the "frontier" of the current precedence list (p) is being pushed forward using the topological structure of the graph (G). The modular structure is shown in Figure 4.10. When the BFS tree is completed, each node will be assigned a level, which is the distance from the root of the tree.

4.4.5 Error Metric

We list the error metrics used in the entities of the four Big Data kernels in Table 4.3. The metrics are computed from the most relevant variables in each of the programs.

Program	**Error Metric(s)**
Fluid	L2 Norm of the error in the density field (d)
K-Means	L2 Norm of the cluster centers vector; Error Factor of membership
PageRank	L2 Norm of the page weights
BFS	Proportion of nodes being assigned a wrong level

Table 4.3: Error Metrics Used for the Programs
(The Root Mean Square Deviation, RMSD is by definition the L2-Norm.)

4.5 Experimental Results

4.5.1 Input Configuration and Input Generation

The inputs to the Big Data kernels and the number of iterations of the main loop of respective programs utilizing the kernels are listed in Table 4.4.

Program	**Input**	**Iterations**	**Num. Experiments**
Fluid	A 50x50 grid initialized with a simple pattern	10	24374
KMeans	Dimension-reduced data containing 1797 hand-written characters	15	75075
PageRank	Amazon web dump containing 65536 nodes [43]	14	15057
BFS	Randomly-generated graph with 8192 nodes and 10650 edges	7	47984

Table 4.4: Program Inputs and Number of Iterations of the Main Loop.

4.5.2 Propagation of Errors

This section discusses how the error metrics change as the errors propagate during run time.

- **PageRank**

Figure 4.11 shows the traces of a subset of injected errors in the PageRank program. Most errors monotonously decrease in a stable way as the iteration count increases. In comparison, although using a similar linear-algebraic algorithm, the Fluid Simulation program tends to see error metrics that preserve their magnitudes without either magnifying or dampening.

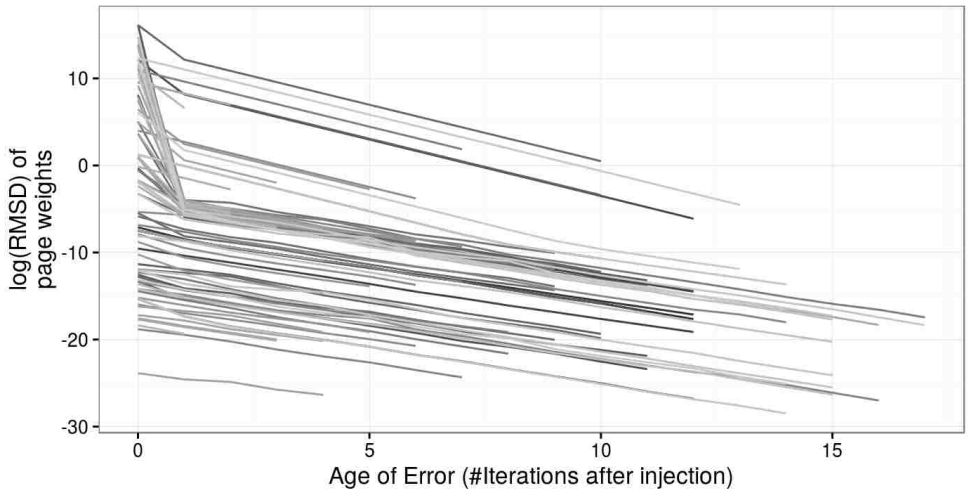

Figure 4.11: Traces of a subset of injected errors in PageRank. X axis denotes the "age of a bit flip error"
(number of iterations after error injection). Y axis denotes the error metric. Different colors represent different runs.

- **K-Means**

Figure 4.12 shows the traces of a subset of injected errors in the KMeans program. The age of the error is mapped onto the X axis. The Error Factor is mapped to the Y axis. It can be seen that the propagation pattern is not uniform; some of the corrupted runs would re-converge to the correct run in a short number of iterations but some could not.

Figure 4.12: Trace of a subset of errors injected in KMeans.
Colors represent different runs.

- **Fluid Simulation**

Figure 4.13 shows the trace of a subset of errors induced by bit flips in the Fluid Simulation program. The age of the error, which is the number of time steps passed since error injection, is mapped onto the X axis. The errors are injected at random positions, which could be in any iteration. The RMSD in the d field is mapped to the Y axis.

From the figure it can be seen that the error magnitudes tend to change gradually as time step advances. The magnitude also tends to stabilize. The trend at which the magnitudes change is dependent on the initial magnitudes. To illustrate, the initial magnitude of CG are mainly distributed between $[10^{-7.5}, 10^{0}]$, which is different from that of GS, $[10^{-20}, 10^{0}]$. The final magnitudes are also different.

- **Breadth First Search (BFS)**

Figure 4.14 shows the trace of 100 errors induced by bit flips in the BFS program. The age of the error (number of iterations after injection) is mapped to the X axis. The proportion of nodes that would receive a wrong level based on the intermediate BFS search tree at individual time steps are mapped to the Y axis.

As we can see from the figure, most bit-flip induced errors in BFS monotonously increase. In some cases, the result would become completely incorrect due to critical data

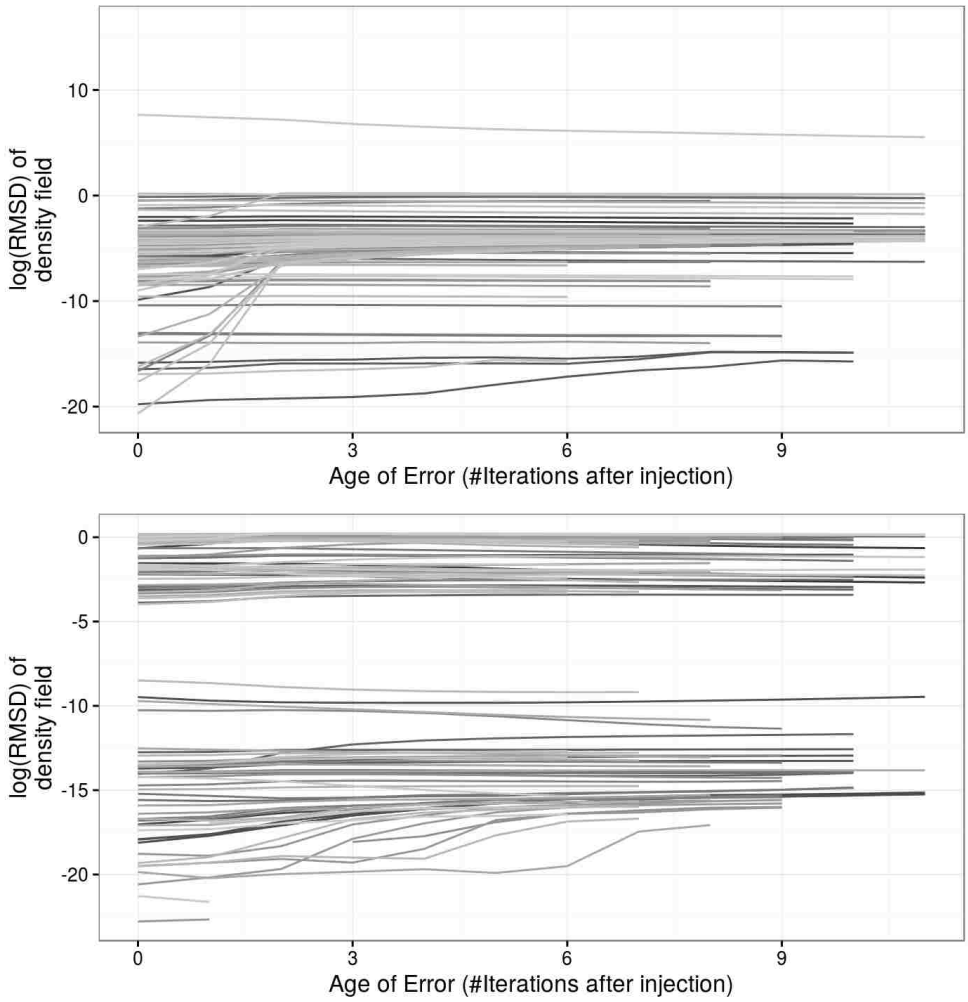

Figure 4.13: Traces of a subset of errors injected in Fluid Simulation with the Conjugate Gradient (CG) solver (top) and the Gauss-Seidel (GS) solver (bottom.)
Colors represent different runs.

structure being corrupt.

4.5.3 Model Training and Accuracy

This section discusses the accuracy of Model 1 and Model 2 described in Section 4.3.3. For Model 1 we quantify how much it is able to model the relationship between the dynamic fault site information to the distribution of errors, namely how a bit-flip propagates to program states. For Model 2 we quantify how much it is able to model the propagation of errors between time steps.

Accuracy for both models is quantified by comparing against ground truth. We com-

Figure 4.14: Traces of a subset of errors injected in Breadth First Search. Colors represent different runs.

pute the earth mover distance (EMD) between the predictions and the actual RMSD at the end iteration, denoted EMD_1. We also compute the earth mover distance (EMD) between the RMSD distribution at the beginning iteration and the end iterations, denoted EMD_2. We compute the ratio $\frac{EMD_1}{EMD_2}$. Thus, a *smaller* the ratio means a more accurate prediction.

The beginning and ending iteration numbers are (4, 14) for PageRank, (1, 15) for KMeans, and (4, 10) for Fluid Simulation.

For each application, we vary the proportion of the examples used for training and see how the prediction quality varies. We pick the traces by their unique combination of fault injection parameters (DynamicFSID, BitID) into the training and test set. The Static Fault Site ID (StaticFSID) is implied by DynamicFSID so it doesn't need be included.

We measure the prediction error on the entire data set. That means the training set and the prediction output from the test set together make up the error distribution at the ending iteration.

- **Pagerank**

Due to the simplicity in the error propagation patterns, a segmented linear regression model is enough for capturing the error propagation pattern of PageRank, as shown in Figure 4.15, Model 1 needs 75% of the input data for training to reach the maximum pre-

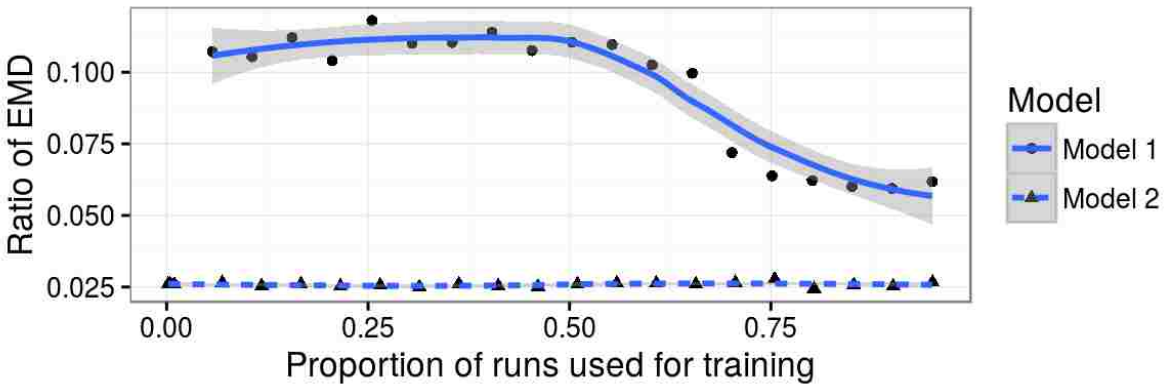

Figure 4.15: Model accuracy for PageRank

dictive power while Model 2 needs only a few data points to reach the maximum predictive power.

- **KMeans**

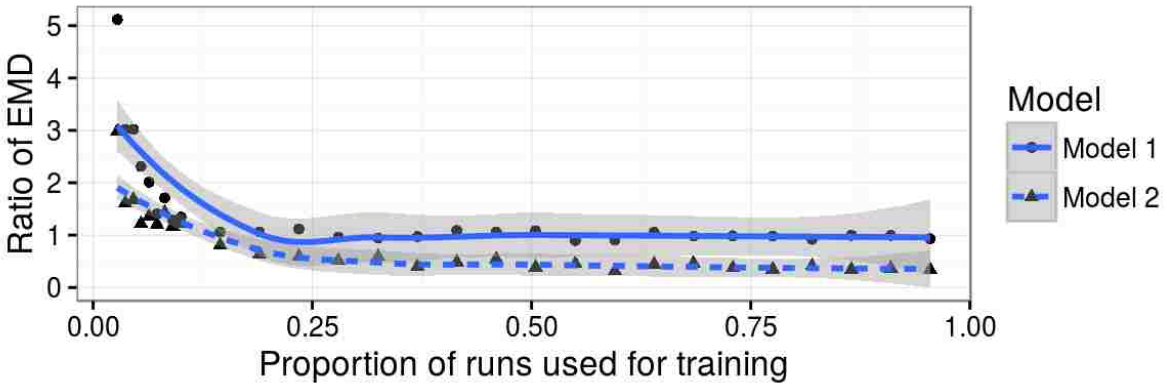

Figure 4.16: Model accuracy for KMeans

We had to use a regression tree to capture the error propagation pattern of KMeans, because there is one segment in the range of the input RMSD that does not have a one-to-one mapping. The correctness improves as training set size increases, as shown in Figure 4.16, Both Model 1 and Model 2 need 25% of the input data for training to reach maximum predictive power.

- **Fluid Simulation**

A segmented linear regression model is used for Fluid as is shown in Figure 4.17 because the RMSD propagation pattern is simple. Most of the changes in the RMSD are in

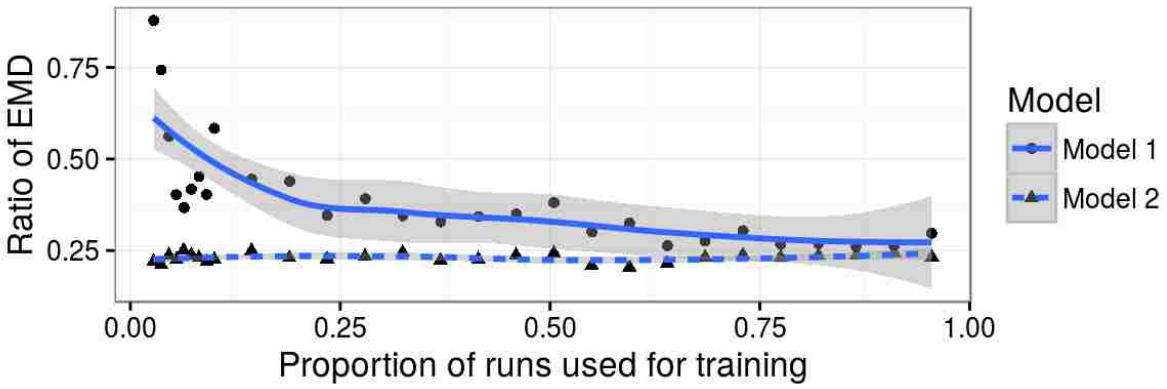

Figure 4.17: Model accuracy for Fluid Simulation

predictable directions. From the results, Model 1 needs 25% of the input data for training to reach the maximum predictive power. Model 2 needs only a few data points to reach the maximum predictive power. The performance of Model 2 is very stable. Even with very few training examples, Model 2 is able to capture the RMSD changes.

- **Breadth First Search (BFS)**

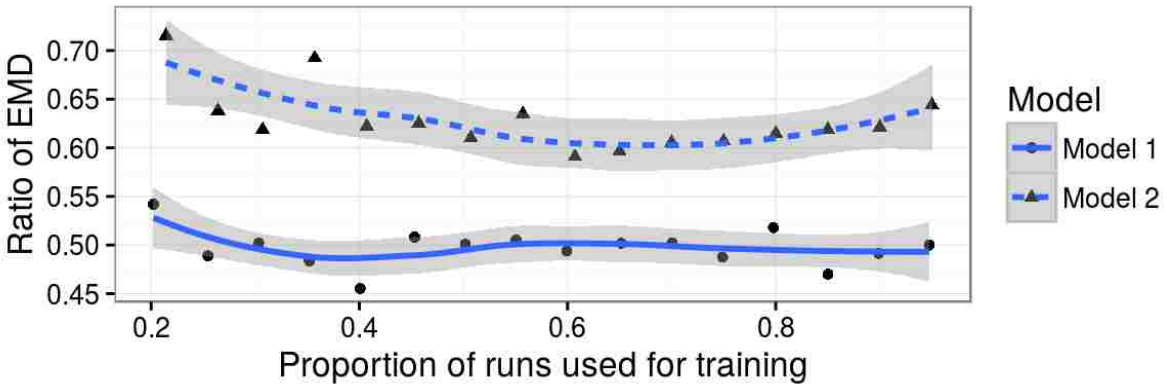

Figure 4.18: Model accuracy for BFS.

The regression tree is used to capture the error propagation pattern of BFS because of the non-linear pattern, as is shown in Figure 4.18, Model 1 needs about 60% of the input data for training to reach the maximum predictive power. Model 2 needs about 50% of the input for training to reach the maximum predictive power. It is worth noting that Model 2 suffers from over-fitting when the proportion of data used for training is high.

4.5.4 Factors affecting Model Accuracy

- **Model 1**

The independent variables of Model 1 are listed in Table 4.2. However, not all of the variables are equally relevant to the final RMSD:

Program	DynamicFSID	StaticFSID	BitID	NumIter
Fluid	0.523348	0.316858	0.118150	0.041644
KMeans	0.016321	0.691838	0.290197	0.001644
Pagerank	0.445098	0.327972	0.129911	0.097019
BFS	0.940505	0	5.949493	0

Table 4.5: Variable Relevance in Model 1.

As Table 4.5 shows, the most relevant variable for Fluid and Pagerank is DynamicFSID, and for KMeans, the most relevant variables are StaticFSID and BitID.

DynamicFSID being irrelevant in KMeans suggests the shape of the error trajectories is not affected by which iterations are being injected errors. In other words, it is uncertain whether the error would be dampened or amplified across iterations. In contrast, the patterns in Fluid and Pagerank are more stable, as can be seen from Figures 4.13, 4.12 and 4.11.

For BFS, DynamicFSID is most relevant, followed by BitID. StaticFSID and NumIter are completely irrelevant. The reason is because error injected into all but only a few of the static faults are masked and will not result in any observable error in program states.

- **Model 2**

Model 2 takes the error distribution at the beginning and end iterations. As a result, the relationship between the errors, visualized in Figure 4.19, determines the model's prediction quality.

Visually, there is linear correlation between the errors: greater errors at the beginning iteration means greater errors at the ending iteration. The only exception is when the error at the beginning iteration is small enough, the output error would be constant in Fluid. Same for Pagerank if the error at the beginning iteration is large enough. For these cases segmented linear regression would be enough for capturing the shapes. To fix the effects

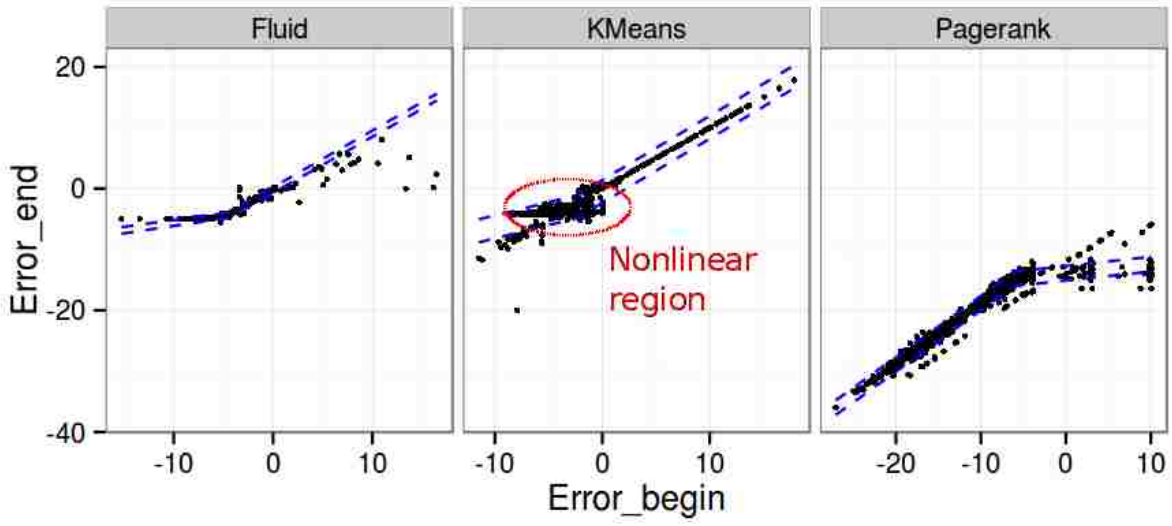

Figure 4.19: Errors in program variables at the beginning and ending iterations (X and Y axes). Dashed lines are prediction intervals of segmented linear models.

caused by outliers, we have removed the top and bottom 5% of the input data.

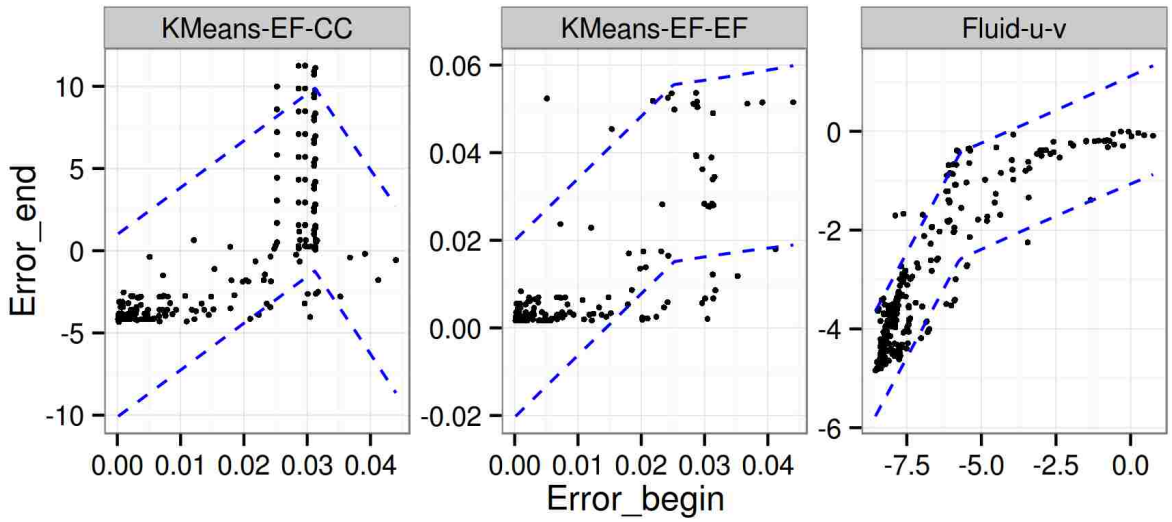

Figure 4.20: *Undesirable* choices of variables for Model 2

However, there exists a non-linear region in K-Means which affects the predictive power of the segmented linear regression. The region is highlighted in Figure 4.19. One X coordinate in this region may correspond to two Y axis, which forces the predictive interval to become larger and results in greater error in the predicted errors. To fix this we decided to

use the regression tree, which is more complex than line segments and can better capture the shape by further subdividing the input data set.

Since there exists multiple program variables, we should find the ones that most accurately capture the error propagation patterns with the best accuracy. Actually, certain variable combinations may make prediction more difficult. Figure 4.20 shows the choices that are not desirable for building Model 2.

- **Cost Saved by the Inter-Iteration Efficient Fault Characterization Algorithm**

Consider performing a NF fault injection experiments into a program that runs for N iterations. Algorithm 1 runs all instances to completion, and the cost measured in number of program iterations is $NF \cdot N$. Algorithm 2 picks a fraction from each iteration and run to completion, and the cost measured in number of program iterations is $\sum_{i=1}^{i=N} i + (N - i) \cdot \mu$, where μ is the proportion between instances in an iteration run to completion and the total number of instances with faults injected at that iteration.

With results from Section 4.5.3 we set μ to 0.01 for Fluid and Pagerank and 0.25 for KMeans. By plugging in the numbers we could obtain the costs in Table 4.6:

Program	**Iter**	**Fraction**	**Alg. 1 Cost**	**Alg. 2 Cost**	**Saving**
Fluid	10	1%	100	55.45	45.6%
KMeans	15	25%	225	146.25	35.0%
Pagerank	14	1%	196	105.91	45.9%
BFS	7	50%	49	42.00	14.3%

Table 4.6: Cost to characterize the effect of soft faults on a program

4.5.5 Applying Fault Resilience Techniques

With the results obtained in Section 4.5, ErrorSight produces the error profile of a program and shows the expected error metric caused by a bit flip on the instructions that correspond to each source line. With this information, the developer can use to decide how to apply fault resilience techniques. In this table, column Mean Error (ME) shows the expected error that would appear in the final output if a bit flip is injected into a dynamic instruction that corresponds to this line of source code. The column Probability (P) shows

the probability that a dynamic fault site belongs to this line. The column Product (Prod) is a product of ME and P. Intuitively, the sum of all the entries in the Prod column is the weighted sum of the ME column, which is the expected error in the final output of the program should a bit flip occurs randomly during its run time. Columns without and with "FT" represent the metrics from the original and the fault-tolerant versions of the program.

Line No.	**Source Code**	**Mean Error (ME)**	**Probability (P)**	**Product (Prod)**	**Mean Error FT**	**Probability FT**	**Product FT**
120	`for(i=0; i<nlocalverts; ++i) pred[i]=-1`	2.41e-1	1.23e-1	2.98e-2	1.22e-4	5.26e-2	6.42e-6
192	`for(i=0; i<oldg_count; i++) {`	1.06e-1	7.10e-2	7.51e-3	1.06e-1	1.0e-3	1.06e-4
200	`for(j=g->rowstarts[ VERTEX_LOCAL(oldg[i])]; j<j_end; j++) {`	6.59e-3	3.32e-1	2.19e-3	3.63e-3	4.80e-1	1.74e-3
208	`if(!TEST_VISITED(tgt)) {`	4.08e-3	3.67e-1	1.5e-3	3.32e-3	3.09e-1	1.03e-3
209	`SET_VISITED(tgt);`	1.43e-3	6.53e-2	9.35e-5	1.06e-3	4.60e-2	4.90e-5
	(Other)	N/A	N/A	2.45e-4	N/A	N/A	5.18e-4
	(Sum)			4.13e-2			3.44e-3

Table 4.7: Static Fault Site to Source Code Mapping of Breadth First Search.

We choose to make the source code lines that are most vulnerable to the Breadth First Search (BFS) shown in Table 4.7. In this Table the greatest value in the Prod column belongs to Line 120 of the source code of BFS. This means that this line is the most significant contributing factor to the overall resilience of the program.

We manually triplicated the pointer dereferencing and value assignment operations in the loop, and performed a Byzantine error check [34] before incrementing the loop index and writing to the `pred` array, namely, if one replica of a pointer is corrupt, the other two are used to correct it, and if two or more replicas are corrupt, the loop is restarted from the beginning. This effectively reduced the occurrence of out-of-loop-boundary errors and the assignment of wrong values.

Figure 4.21 shows the fault-resilience source code and the resultant change in the mean error of the entire program after fault resilience is applied to Line 120. The mean error

```
        Original Line 120        |          Fault-Resilient Line 120
                                 |
for (i = 0; i < nlocalverts; ++i) | int64_t* p1 = pred, *p2 = p1, *p3 = p2;
   pred[i] = -1;                 | int64_t* end1 = &(pred[nlocalverts]),
                                 |         *end2=end1, *end3=end2;
                                 | while (p1 < end1 || p2 < end2 || p3 < end3) {
                                 |     if ( !(p1 == p2 && p1 == p3) ) {
                                 |         if (p1 == p2) { p3 = p2; }
                                 |         else if(p2 == p3) { p1 = p2; }
                                 |         else if(p1 == p3) { p2 = p3; }
                                 |         else { goto retry; }
                                 |     } else { *p1 = -1; }
                                 |     p1++; p2++; p3++;
                                 |     if ( !(end1 == end2 && end2 == end3) ) {
                                 |         if (end1 == end2) { end3 = end1; }
                                 |         else if (end2 == end3) { end1 = end2; }
                                 |         else if (end3 == end1) { end2 = end3; }
                                 |         else { goto retry; }
                                 |     }
                                 | }
```

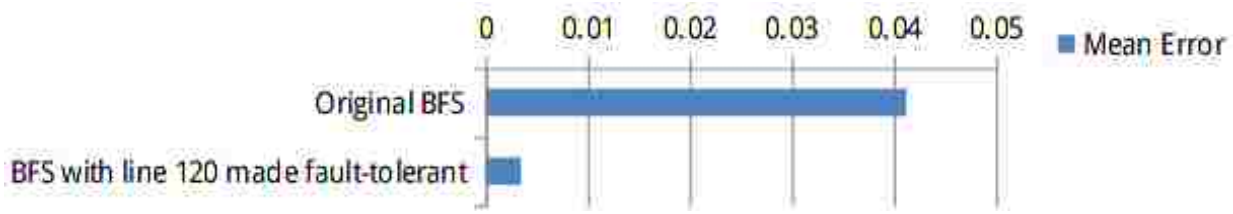

Figure 4.21: Triplication fault resilience mechanism used on Line 120 and the resultant change in the mean error of the entire BFS program.

of the application is reduced by a magnitude, from 4.13e-2 to 3.44e-3. Table 4.7 indicates the Mean Error resulting from the fault-resilient version of Line 120 has been reduced from 2.98e-2 to 6.42e-6. After this, Line 120 is no longer the main contributor of errors in this BFS. In addition, the modification does not introduce significant overhead because Line 120 was not a hotspot in the original program.

4.6 Conclusion

In this chapter we have proposed ErrorSight, a tool aimed at helping the developers to write fault-resilient programs. We demonstrated with four Big Data kernels that it can efficiently capture the error propagation patterns that a human developer can analytically obtain, and use the patterns to construct a predictive model to save the error characteriza-

tion cost, and showing the application developer which part of the source code is the most significant vulnerable part of a numerical program. With this information, the developer can then apply fault resilience mechanisms to the program and significantly improve its resilience under a faulty environment.

Chapter 5
Summary and Future Work

With exascale high performance approaching and big data analytics becoming a reality, along with the increasing density of computational units, we will be expecting more and more soft faults during long-running operations, which could threaten the accuracy of the outputs and impact the quality of service. Understanding the faults and developing countermeasures for them helps us to minimize the vulnerability of programs.

We proposed two tools to help achieve this goal. FaultTelescope builds a comprehensive fault profile of a program in question and helps the programmer to focus efforts on the most vulnerable code regions. ErrorSight further reduces the cost in building such fault profiles by leveraging modular analysis. Both tools used statistical algorithms to ensure the results are well-grounded.

The tools are used on a series of numerical routines and a few scientific computing and big data programs. The tools are proven to be able to effectively detect the vulnerabilities to soft errors. By focusing on the vulnerable parts and applying the algorithmic invariants, the overall resilience of the programs in question can be greatly improved.

The evolution of big data programs in terms of workloads and tools calls for updated approaches in fault resilience research. This project will be continued with the exploration of more big data programs and software stacks (for example, map-reduce based algorithms running on the Spark software stack.) The future works will aim at developing updated tools and fault resilient software.

References

[1] *Fault Injection into VHDL Models: Experimental Validation of a Fault-Tolerant Microcomputer System*, 1999.

[2] Kulfi fault injector. https://github.com/quadpixels/kulfi, 2013.

[3] D. Austin. How google finds your needle in the web's haystack. http://www.ams.org/samplings/feature-column/fcarc-pagerank, 2010.

[4] R. C. Baumann. Radiation-Induced Soft Errors in Advanced Semiconductor Technologies. *IEEE Transactions on Device and Materials Reliability*, 5(3):305–316, September 2005.

[5] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein. Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers. *Foundations and Trends in Machine Learning*, 3(1):1–122, June 2011.

[6] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen. *Classification and Regression Trees*. CRC Press, 1984.

[7] G. Bronevetsky and B. de Supinski. Soft Error Vulnerability of Iterative Linear Algebra Methods. In *International Conference on Supercomputing*, 2008.

[8] F. Cappello, A. Geist, B. Gropp, S. Kale, B. Kramer, and M. Snir. Toward Exascale Resilience. In *International Journal of High Performance Computing Applications*, 2009.

[9] M. Casas, B. R. de Supinski, G. Bronevetsky, and M. Schulz. Fault Resilience of the Algebraic Multi-Grid Solver. In *International Conference on Supercomputing*, pages 91–100, 2012.

[10] J. Chung, I. Lee, M. Sullivan, J. H. Ryoo, D. W. Kim, D. H. Yoon, L. Kaplan, and M. Erez. Containment domains: A scalable, efficient, and flexible resilience scheme for exascale systems. *2012 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, Nov. 2012.

[11] C. da Lu and D. A. Reed. Assessing Fault Sensitivity in MPI Applications. In *Supercomputing*, November 2004.

[12] N. DeBardeleben, S. Blanchard, Q. Guan, Z. Zhang, and S. Fu. Experimental framework for injecting logic errors in a virtual machine to profile applications for soft error resilience. In *Euro-Par 2011: Parallel Processing Workshops*, volume 7156 of *Lecture Notes in Computer Science*, pages 282–291. Springer Berlin Heidelberg, 2012.

[13] N. DeBardeleben, S. Blanchard, V. Sridharan, S. Gurumurthi, J. Stearley, and K. Ferreira. Extra bits on sram and dram errors - more data from the field. *IEEE Workshop on Silicon Errors in Logic - System Effects (SELSE)*, 2014.

[14] P. Du, P. Luszczek, and J. Dongarra. High Performance Dense Linear System Solver with Soft Error Resilience. *2011 IEEE International Conference on Cluster Computing*, pages 272–280, Sept. 2011.

[15] P. Du, P. Luszczek, and J. Dongarra. High Performance Dense Linear System Solver with Resilience to Multiple Soft Errors. *Procedia Computer Science*, 9:216–225, Jan. 2012.

[16] J. Elliott, M. Hoemmen, and F. Mueller. Evaluating the Impact of SDC on the GMRES Iterative Solver. In *Proceedings of the 28th International Parallel and Distributed Processing Symposium (IPDPS 2014)*, 2014.

[17] K. Ferreira, J. Stearley, I. James H. Laros, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold. Evaluating the viability of process replication reliability for exascale systems. In *Supercomputing*, 2011.

[18] F. S. Foundation. Gnu scientific library – reference manual, 2011.

[19] A. G. Goncalo Amador. Linear Solvers for Stable Fluids: GPU vs CPU. In *17th Encontro Portugues de Computacao Grafica (EPCG'09)*, 2009.

[20] M. C. Hsueh, T. K. Tsai, and R. K. Iyer. Fault Injection Techniques and Tools. *IEEE Computer*, 30(4):75–82, November 1997.

[21] K.-H. Huang and J. A. Abraham. Algorithm-Based Fault Tolerance for Matrix Operations. In *International Conference on Dependable Systems and Networks (DSN)*, pages 161–170, 2010.

[22] ITRS. International Technology Roadmap for Semiconductors. Technical report, 2013.

[23] S. Kumar, S. Hari, S. V. Adve, H. Naeimi, and P. Ramachandran. Relyzer: Exploiting Application-Level Fault Equivalence to Analyze Application Resiliency to Transient Faults . In *Proceedings of the 17th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012)*, 2012.

[24] C. LaFrieda, E. Ipek, J. F. Martinez, and R. Manohar. Utilizing dynamically coupled cores to form a resilient chip multiprocessor. In *International Conference on Dependable Systems and Networks (DSN)*, pages 317–326, 2007.

[25] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis and transformation. pages 75–88, San Jose, CA, USA, Mar 2004.

[26] H. Li, J. Mundy, W. Patterson, D. Kazazis, A. Zaslavsky, and R. I. Bahar. Thermally-induced soft errors in nanoscale CMOS circuits. In *IEEE International Symposium on Nanoscale Architectures (NANOARCH)*, pages 62–69, 2007.

[27] M.-L. Li, P. Ramachandran, U. R. Karpuzcu, S. Kumar, S. Hari, and S. V. Adve. Accurate microarchitecture-level fault modeling for studying hardware faults. In *International Symposiumn on High-Performance Computer Architecture*, 2009.

[28] X. Li, M. C. Huang, K. Shen, and L. Chu. A realistic evaluation of memory hardware errors and software system susceptibility. In *USENIX Annual Technical Conference*, 2010.

[29] W.-K. Liao. Parallel K-Means Data Clustering, 2013.

[30] L. W. Massengill, B. L. Bhuva, W. T. Holman, M. L. Alles, and T. D. Loveless. Technology scaling and soft error reliability. In *IEEE Reliability Physics Symposium (IRPS)*, pages 3C.1.1–3C.1.7, 2012.

[31] S. Michalak, K. W. Harris, N. W. Hengartner, B. E. Takala, and S. A. Wender. Predicting the Number of Fatal Soft Errors in Los Alamos National Laboratory's ASC Q Supercomputer. *IEEE Transactions on Device and Materials Reliability*, 5(3), September 2005.

[32] A. Moody, G. Bronevetsky, K. Mohror, and B. De Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–11, Nov 2010.

[33] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang. Introducing the Graph 500. *Cray Users Group (CUG)*, 2010.

[34] T. Nanya and H. Goosen. The Byzantine Hardware Fault Model. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 8, pages 1226–1231, Nov 1989.

[35] W. T. Olson. Hattrick n-body simulator. http://code.google.com/p/hattrick-nbody, 2014.

[36] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *International Symposium on Computer Architecture (ISCA)*, pages 25–36, 2000.

[37] Y. Rubner, C. Tomasi, and L. Guibas. A Metric for Distributions with Applications to Image Databases. In *Proceedings of the Sixth International Conference on Computer Vision (ICCV 1998)*, pages 59–66, Jan 1998.

[38] D. Sbragion. Drc: Digital room correction. http://drc-fir.sourceforge.net/, 2014.

[39] B. Schroeder, E. Pinheiro, and W.-D. Weber. DRAM Errors in the Wild: A Large-Scale Field Study. In *Proceedings of SIGMETRICS 2009*, 2009.

[40] J. Sloan, D. Kesler, R. Kumar, and A. Rahimi. A Numerical Optimization-based Methodology for Application Robustification: Transforming Applications for Error Tolerance. In *International Conference on Dependable Systems and Networks (DSN)*, pages 161–170, 2010.

[41] J. Sloan, R. Kumar, and G. Bronevetsky. Algorithmic approaches to low overhead fault detection for sparse linear algebra. *Dependable Systems and Networks*, (Section III), 2012.

[42] D. T. Stott, B. Floering, D. Burke, Z. Kalbarczyk, and R. K. Iyer. NFTAPE: A Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors. In *Proceedings of the 2000 IEEE International Computer Performance and Dependability Symposium (IPDS 2000)*, 2000.

[43] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu. BigDataBench: A Big Data Benchmark Suite from Internet Services. In *Proceedings of the 20th International Symposium on High-Performance Computer Architecture (HPCA 2014)*, 2014.

Vita

Sui Chen was born on June 1st, 1990, in Changsha, Hunan, China. He received his Bachelor of Engineering in Information Security from Shanghai Jiao Tong University, Shanghai, China in June 2011. He began his graduate studies at Louisiana State University in August of 2011. His technical interests are fault resilience, GPU and transactional memory.