LAWRENCE LIVERMORE NATIONAL LABORATORY

LLNL-SR-666073

A Framework For Evaluating Comprehensive Fault Resilience Mechanisms In Numerical Programs

S. Chen, L. Peng, G. Bronevetsky

January 13, 2015

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

A Framework For Evaluating Comprehensive Fault Resilience Mechanisms In Numerical Programs

Authors will be added later

Abstract

As HPC systems approach Exascale, their circuit feature will shrink, while their overall size will grow, all at a fixed power limit. These trends imply that soft faults in electronic circuits will become an increasingly significant problem for applications that run on these systems, causing them to occasionally crash or worse, silently return incorrect results. This is motivating extensive work on application resilience to such faults, ranging from generic techniques such as replication or checkpoint/restart to algorithm-specific error detection and resilience techniques. Effective use of such techniques requires a detailed understanding of (1) which vulnerable parts of the application are most worth protecting (2) the performance and resilience impact of fault resilience mechanisms on the application. This paper presents FaultTelescope, a tool that combines these two and generates actionable insights by presenting in an intuitive way application vulnerabilities and impact of fault resilience mechanisms on applications.

Keywords: Soft faults, High-performance computing, Numerical errors, Fault resilience

1. Introduction

The increasing size and complexity of HPC systems is making them increasingly vulnerable to soft faults, which are transient corruptions in the states of electronic circuits caused by physical phenomena such as strikes by neutrons or alpha particles [1, 2] or thermal electrical noise [3]. They can affect the state of processor latches and registers, and may cause the application to crash or silently return incorrect results [4]. As the feature sizes of electronic circuits shrink, technology scaling will make soft errors a larger problem [5] 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 14 nm [6], which is approximately 28 silicon atoms (5 Å per atom) across. These phenomena make it imperative to develop techniques to make HPC systems resilient to soft faults.

The resilience problem must be addressed at all levels. While materials science and circuit design techniques can be used to improve resilience, they come at a cost in reduced power efficiency and performance that can become prohibitive if processors need to be sufficiently reliable to build a large HPC system. Techniques such as error correcting codes (ECC) have been very effective at making memories and caches resilient to soft faults [7]. However, as total system memories of systems are expected to grow by 100x to 350x to reach Exascale [8] their increased fault vulnerability will require more elaborate and expensive ECC to be deployed. Further, ECC is more expensive for protecting core-internal state such as latches and is significantly less effective for checking the correctness of computations. Processor designs that incorporate instruction replication [9] 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 techniques that can enable applications to execute productively on unreliable hardware. The most general approach is replication of computations across core or nodes [10, 11], 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 much cheaper but highly manual algorithm-specific techniques [12, 13] that verify that algorithmic invariants hold. Because these techniques usually only address error detection, to achieve full resilience they must be supported by other techniques, such as checkpoint-restart [14] and pointer replication [15].

To design and deploy software-level resilience schemes application developers need tools to quantify the effect of faults on their applications and support for choosing the most appropriate resilience technique for each type of fault. This paper presents FaultTelescope, a comprehensive approach to supporting both needs in the form of (i) statistically well-grounded fault injection studies and (ii) exploration of how the configuration options of resilience mechanism affect the performance and resilience of individual application kernels and the overall application.

The importance of analyzing and quantifying the impact of errors on application behavior is demonstrated in various studies. As Du et al have shown [16, 17], resilience is becoming a measurement of quality of linear solver packages. Detailed study of output accuracy is found in several fault injection frameworks. For example, Debardeleben et al [18] document how the numeric error caused by an injected fault evolves over time. Probabilistic modeling has been used by Chung et al [19] to help compute the expected recovery time, which cannot be measured easily for very large scale applications. Sloan et al [20] has discussed the use of algorithmic checks over sparse linear algebra kernels and focused mainly on reducing false positive and false negative in error detection.

Resilience studies like these are enabled by fault injection, a technique where the application is executed multiple times. During each execution the application's state is injected with some software-level manifestation of a low-level fault. There exist many fault injection tools the simulate various types of faults in hardware components, ranging from transister-level faults to fail-stop crashes of entire compute nodes [21, 22, 23].

FaultTelescope supports resilience studies by integrating with the KULFI fault injector [24], which models faults as single bit flips in the outputs of a randomly selected instructions of applications compiled into the LLVM instruction architecture. FaultTelescope presents the results of such studies to developers by providing visualizations of how application state and output is affected by injected errors, expressed via developer-specified error metrics. Further, FaultTelescope computes confidence intervals of the presented data to enable developers make well-supported conclusions, while balancing the benefits of improving the confidence of the analysis against the cost of running more fault injection experiments.

Finally, a key issue developers face is that different types of faults manifest themselves very differently to software. For each possible fault type developers need to select the most appropriate resilience technique for detecting and tolerating the fault, as well as the best configuration of the technique. The choice of technique and its configuration can have a significant effect on the performance and resilience of the application and the wrong choice can make the application more vulnerable to errors than it was originally [25]. FaultTelescope helps application developers choose the best way to manage all the fault types their application may be exposed to by enabling them experimentally measure the effectiveness of various resilience mechanisms and their configurations. To reduce the cost of searching a large parameter space FaultTelescope enables developers to first focus on key application kernels and then on the overall application.

At a high level, FaultTelescope provides a comprehensive suite of capabilities that help application developers bridge the gap between low-level faults and software-level resilience solutions the developers can be confident in. FaultTelescope includes

- Efficient architectural-level fault injection based on KULFI
- Statistically sound computations of confidence intervals of the experimental data
- Hierarhical analysis that operates from kernels to entire applications

The FaultTelescope approach is evaluated in the context of three applications that represent different application domains: the LASSO [26] solver for the Linear Solvers domain, the DRC [27] HiFi audio filter for the Signal Processing domain, and the Hattrick [28] gravity simulator for the Differential Equation Solvers domain. This paper demonstrates the utility of comprehensive resilience toolchain for helping developers explore the vulnerability properties of their application.

The rest of the paper is organized as follows. Section 2 gives an overview of the experimentation methodology and error model used in FaultTelescope. Section 3 presents the structure of the target applications and the fault resilience techniques used. Section 5 presents how FaultTelescope is used to identify vulnerability and performance/resilience tradeoff. Section 6 presents the algorithm used for selecting the number of fault injection experiments needed in our experiment.

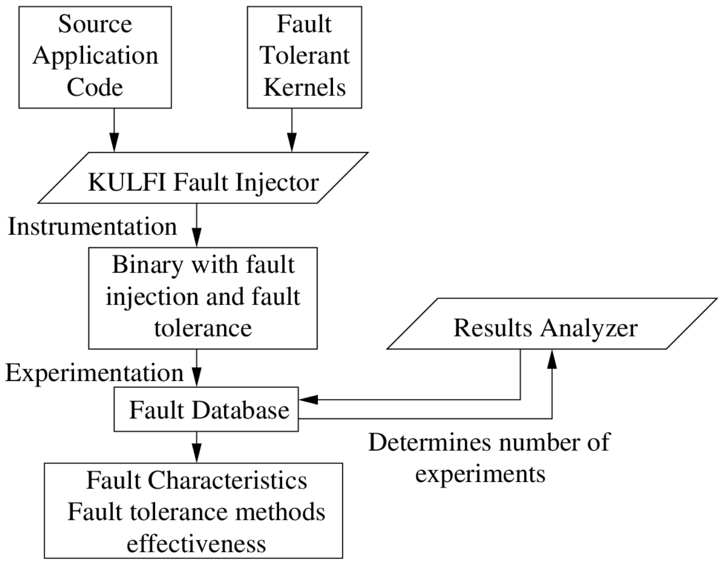

Source Application Code
Fault Tolerant Kernels
KULFI Fault Injector
Instrumentation
Binary with fault injection and fault tolerance
Results Analyzer
Experimentation
Fault Database
Determines number of experiments
Fault Characteristics Fault tolerance methods effectiveness

Figure 1: Overall workflow of FaultTelescope

2. Design Overview

The workflow of FaultTelescope is described in Figure 1 . It performs a fault injection campaign on a target application by executing the entire application and/or individual routines in the application multiple times. A single bit flip is injected each time in a randomly-selected dynamic instruction (a Fault Site). Information including source code location that corresponds to the fault site and the final outcome of the application are also recorded. The final outcome must fall 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 shows abnormal phenomena such as dereferencing invalid pointers, triggering numerical explosion, or entering an infinite loop, which then triggers user-defined or system-defined exception handlers, resulting in the program being terminated.
- Incorrect Result: Program runs to completion, but produces results that exceed the user-defined error bound. In our paper we quantify the magnitude of errors using the L2-norm of its difference against the golden run.

The information above is stored in the fault database for analysis and visualization. The result analyzer uses the database to determine the number of necessary experiments for obtaining a statistically-grounded conclusion about the fault characteristics and resilience of applications.

3. Target Applications

We demonstrate the use of FaultTelescope on three applications, which represent three application domains.

3.1. LASSO

The LASSO [26] program is an implementation of the Alternating Direction Method of Multipliers (ADDR) for solving under-constrained linear problems $Ax = b$ for x (A has fewer rows than columns) while minimizing some function $f(x)$. 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 [29]: matrix-matrix multiplication (MMM), matrix-vector multiplication (MVM), rank-k update (RK) and Cholesky decomposition (CD).

We use ADDR to solve Lasso problems, where the function $\frac{1}{2}\,||Ax - b|\,|_2^2 + \lambda \cdot ||x|\,|_1$ is minimized. Our experiments focus on linear systems 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.

3.2. DRC

DRC [27] is a sequential program that generates filters for HiFi audio systems to compensate for the reflection of sounds in a room by using impulse response measurements of the audio equipment and considering the positions of the listeners. It represents the signal processing application domain. DRC's inputs are stored in Pulse Code Modulation (PCM) format, which is an array of 32-bit floating point numbers representing the samples at each time step. The computations are performed using 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 paper is an audio file of size 768 kilobytes, which is sampled at the rate of $\{30, 40, 50, 60, 70\}KHz$.

3.3. Hattrick

Hattrick [28] is a sequential application that simulates the motion of 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 solver for Ordinary Differential Equations to solve the system's equations of motion using the Runge-Kutta method (RK). 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 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.

Routine		Algorithmic Detector	Checkpointing	Pointer Replication
ADDR	MM SYRK MVM CD	Linear encoding Thresholds: 1e-5 to 1e-8	Inputs	No
FRC	FFT FIR	Parseval's theorem. Sum conservation. Thresholds: 1e-6 to 1e-8.	Inputs	No
Hattrick	RK	Variable step-size	Periodic Timesteps Period: 1, 1e4	1. None 2. All pointers, checked at one code location 3. All pointers, checked on each use

Table 1: The resilience techniques applied to each major routine of each application.

The algorithmic error checks utilized by each of the applications are summarized in Table 1, whose resilience techniques will be discussed in the following Section 4.

4. Resilience Techniques

This section presents the fault characteristics on routines used by the three target applications, and how these techniques can protect the applications from soft errors. Note that we are using a more strict criterion when considering whether an application output is correct. In this section we are considering only outputs *identical* to the golden output produced without fault injection as "correct", while in real life an output may be acceptable if it's within a certain distance from the golden output.

4.1. Error Recovery

A light-weight in-memory checkpointing recovery method is deployed to all routines in order to enable recovery from segmentation fault exceptions. This is done by installing a signal handler with the `sigsetjmp` system call and backing up inputs at the entry points of the routines.

4.2. Algorithmic Error Detection

Here we describe the fault resilience mechanisms applied to the routines.

4.2.1. Cholesky Decomposition

GSL's CD routine has a built-in assertion that terminates the program when the matrix is not positive-definitive. Since most injected errors cause the assertion to be violated, most runs of the original CD are terminated, while those that complete generally finish with very small errors.

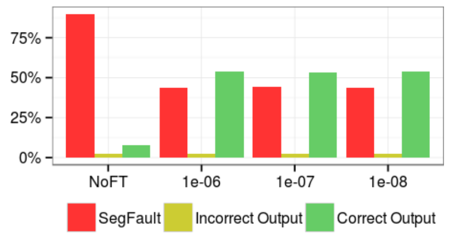


Figure 2: Fault characteristics of Cholesky Decomposition given input size 500x500.

Resilience is applied to CD by observing that CD decomposes matrix A into $L \cdot L^T$ where L is lower-triangular with a positive diagonal. This operation must maintain the identity $Ax = L \cdot (L^T x)$ [13], which can be computed in $O(n^2)$ time. This is significantly cheaper than the $O(n^3)$ complexity of the deterministic CD algorithm. GSL implements an iterative algorithm that runs faster than $O(n^3)$ time but our experiments show that our checker is still significantly faster.

The use of these resilience techniques have a significant effect on the vulnerability of CD to injected errors. Many runs that would otherwise trigger the assertions finish with very small errors, which means the damage to the inputs by a single bit flip error is insignificant, and can be easily recovered through input backup. Cholesky Decomposition benefited significantly from the generic segmentation fault error handler.

Figure 2 summarizes the fault characteristics of different versions of CD. As is suggested by the data, this routine benefits more from being able to recover than actually correcting the outputs while using a tighter error checker threshold does not correct more runs.

4.2.2. FFT

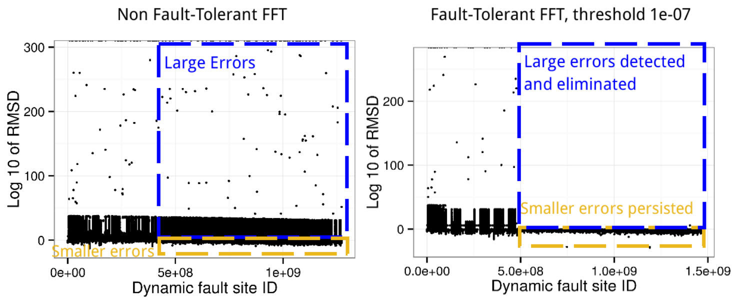


Figure 3: Detailed characteristics of fault resilient FFT with checker threshold 1e-07

FFT decomposes a function into a sum of sine waves of different frequencies: $f(x) = \sum_{n=0}^{N-1} x_n e^{-i2\pi kn/N}$ for some constant k. It 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.

This check runs in $O(n)$ time, as compared to $O(nlog(n))$ or $O(n^2)$ for the FFT algorithm itself (depending on the type of FFT problem).

Figure 4 summarizes the fault characteristics of different versions of FFT: the possibility of producing incorrect output is significantly reduced by the error checkers. In fact, since most of the errors very large such that they can be easily detected with even a lenient detector threshold such as 1e-05. Figure 3 is a temporal RMSD graph, which plots the runs with incorrect results onto an scatterplot graph. For each of the points, the X coordinate corresponds to the Dynamic Fault Site ID, or how late in the lifespan of the routine is the error injected. The Y coordinate corresponds to the magnitude of the errors.

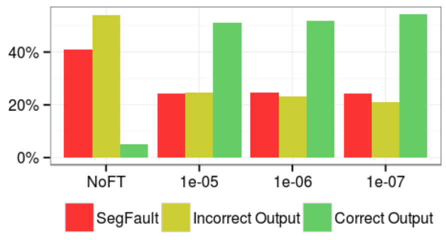


Figure 4: Fault characteristics of FFT given input size 4M.

Comparing the two figures one could see that after the error checker is applied the larger errors are eliminated, but smaller errors persisted. For these smaller errors, a more strict error checker threshold is only slightly helpful in correcting them, which translates into a marginal benefit on the correct output rate.

Since we are already seeing a great improvement in correct runs after applying error check on FFT, we choose 1e-07 as the checker threshold and take a closer look at the impact of the smaller errors on a whole application in Section 5.

4.2.3. Finite Impulse Response Filter Generation (FIR)

The DRC FIR filter algorithm generates a series of samples over the function $sinc(x) = \frac{sin(x)}{x}$ and modulates it with a Blackman window. It is checked using the invariant $\int_{-\infty}^{\infty} sinc(x)dx = 1$, which is preserved by the Blackman window throughout our experiments. Computing the sum requires $O(n)$ additions, which is faster compared to the $O(n)$ trigonometric function evaluations of the non-iterative FIR generation algorithm.

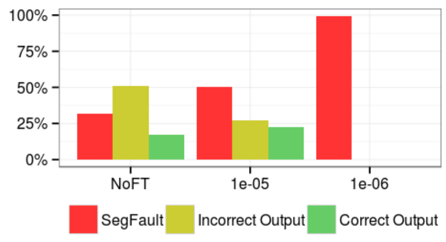


Figure 5: Fault characteristics of FIR of input parameter 512K.

Table 5 shows a summary of the fault characteristics of different versions of FIR. It is clear that the checker threshold 1e-06 is so tight that when it's applied to FIR it caused many false alerts, resulting in many failed runs. In this case, the error checker is not reliable. As a result, we choose 1e-05 as the error checker threshold for FIR.

4.2.4. Matrix-Matrix Multiplication

$A \cdot B$ is checked using a matrix vector multiplication, via the identity: $(A \cdot B) \cdot x = A \cdot (B \cdot x)$, where x is an error-checking vector (we use the vector of all 1s). The checker is asymptotically faster than MMM, with MVM taking $O(n^2)$ time and MMM $O(n^3)$ time.

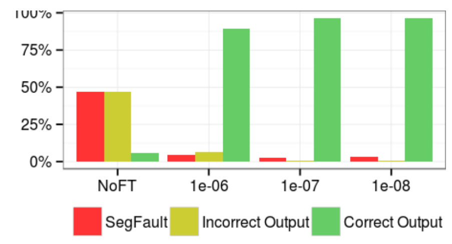


Figure 6: Fault characteristics of Matrix-Matrix Multiplication with input parameter 500x500.

Table 6 shows the fault characteristics of different versions of the MMM routine. It could be seen from the figure that error checker threshold 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.

4.2.5. Rank-K Update

This algorithm computes $\alpha A \cdot A^T + \beta B$, where A and B are matrices.

Its results are checked via the identity $(A \cdot B) \cdot x = A \cdot (B \cdot x)$, where x is an error-checking vector (we use the vector of all 1s). Since checking is done using matrix-vector multiplication, which takes $O(n^2)$ time, as compared to $O(n^3)$ time for RK, this check is very efficient. Like CD, the error checker and recovery for Rank-K update fix many runs with incorrect results and produce correct results. However, some of the incorrect runs are not corrected. This is mainly due to round-off errors in the checker since the checker works in a recursive fashion and involves many addition operations.

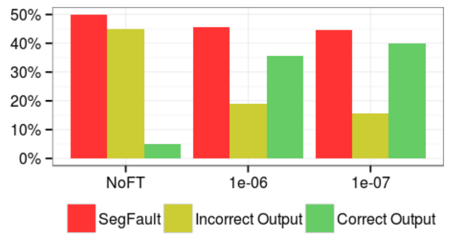


Figure 7: Fault characteristics of Rank-K Update with input parameter 500x500.

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

4.2.6. Matrix-Vector Multiplication

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

It is checked using a similar identity $(x^T A)x = x^T(Ax)$. The complexity of computing $x^T A$ is $O(n^2)$, the same as the original MVM but using additions rather than multiplications. Since MVM is applied in Lasso many times to the same matrix with different vector, the vector $x^T A$ can be reused, amortizing the cost of computing it.

We use 1e-06, 1e-07 and 1e-08 as the error detector thresholds for MV in the experiments.

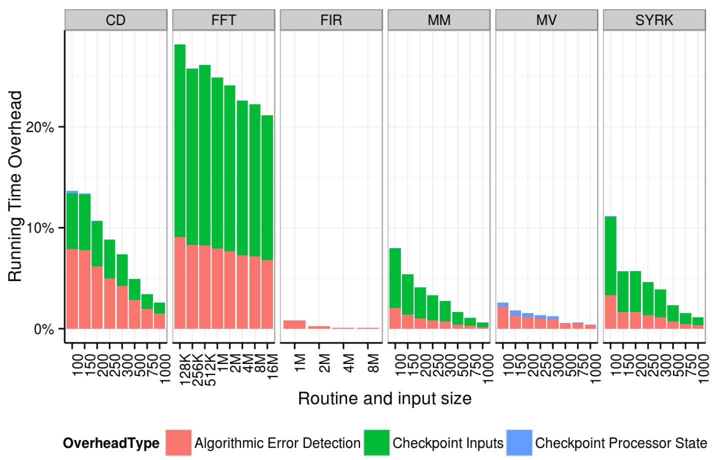


Figure 9: Overhead of fault resilience mechanisms for linear algebra kernels, FFT and FIR

Table 2 summarizes the fault characteristics of Matrix-Vector Multiplication.

The overhead of these algorithmic checks for linear algebra, FFT and FIR are shown in Figure 9. As the figure suggests, the overheads are expected to become lower as input size increases.

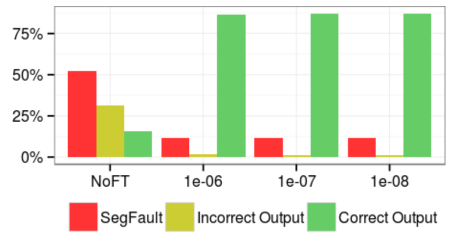


Figure 8: Fault characteristics of Matrix-Vector Multiplication with input parameter 150x150.

4.2.7. Runge-Kutta Integrator

RK4, which stands for 4th order Runge-Kutta method, is a method for solving 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 Runge-Kutta 4 integrator implementation uses adaptive step-size control where it simultaneously uses two step sizes h and $\frac{h}{2}$. If the difference between the two computations exceeds a threshold τ, it switches to the smaller step size to maintain accuracy. If it is smaller than $\frac{\tau}{2}$, the algorithm switches to a larger step size. The adaptive step size control algorithm naturally tolerates soft errors that occurs during either computation. A checkpoint is made every 10000 iterations (denoted as Ckpt in configurations). We chose 10000 because this makes the overhead of checkpointing almost negligible (to compare, an interval of 1, 10 and 100 are not, but they perform almost as well as 10000.) This routine is tested with the second-order nonlinear Van der Pol oscillator equation in the GSL documentation [29].

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

Table 2: Overhead of different versions of the RK4 Integrator

5. Result Analysis

In this section, we present how the fault resilience mechanisms can protect the applications from single bit-flip errors and the performance cost it takes to achieve the protection. We show that the choice of fault checker threshold and replication strategy can have an impact on the performance overhead and/or accuracy under certain circumstances.

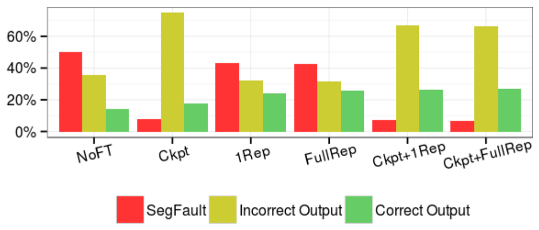


Figure 10: Fault Characteristics of the RK4 Integrator

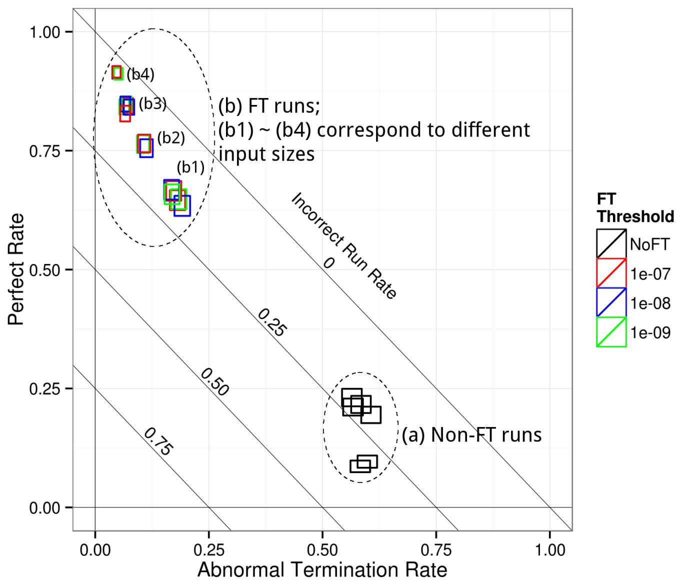
We show the confidence interval of probabilities of all three outcomes of every application configuration by presenting them with rectangles on a 2-D plane. The Binomial Confidence Intervals of the rates of abnormal termination and perfect output are mapped to the X and Y axes respectively. The rates 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, it is more desirable to have a rectangle on the top-left, which means 100% correct outputs and no abnormal terminations. Intuitively, these visualizations give a clear overview of the fault resilience characteristics of application runs.

5.1. LASSO

Figure 11 presents the characteristics and running time overhead of LASSO before and after applying fault resilience.

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

- The rectangles around the bottom-right cluster represent non-fault-tolerant (Non-FT) runs ((a) in Figure 11). For these runs, the probability of abnormal termination is high and the probability of of producing perfect (identical) results is low.
- The rectangles around the top-left cluster represent fault-tolerant ones ((b) in Figure 11). For those runs, the probability of abnormal termination is low and the probability of producing a correct output is high. Further, cluster (b) is divided into sub-clusters corresponding to input sizes ($b1 =$



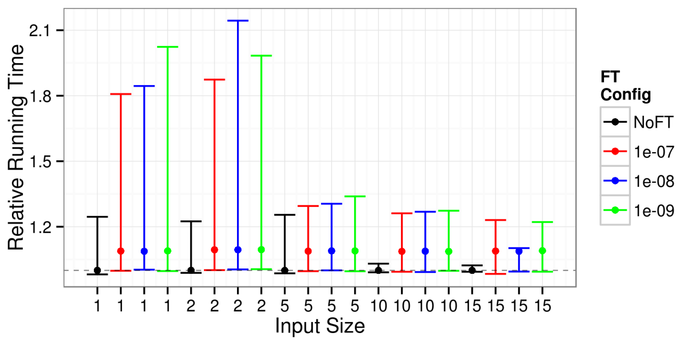


Figure 11: Fault Characteristics and Resilience Overhead of LASSO.

$\{40, 80\} \cdot 500$, $b2 = 200 \cdot 500$, $b3 = \{400, 600\} \cdot 500$, $b4 = 800 \cdot 500$). Their RMSD histogram (as illustrated at the bottom of Figure 11 would be only slightly different) and are omitted.

The trend the fault-tolerant cluster evolves with input size echoes with the diminishing trend of the running time overhead. As input size goes up, the perfect run rate would increase while the SegFault rate and running time overhead would decrease. 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 application resilience characteristics would be shaped by that of this routine.

On the other hand, the error checker threshold used does not have a significant on correctness or performance. The thresholds chosen here for the algorithmic checkers are all adequate for eliminating incorrect runs. The rollback

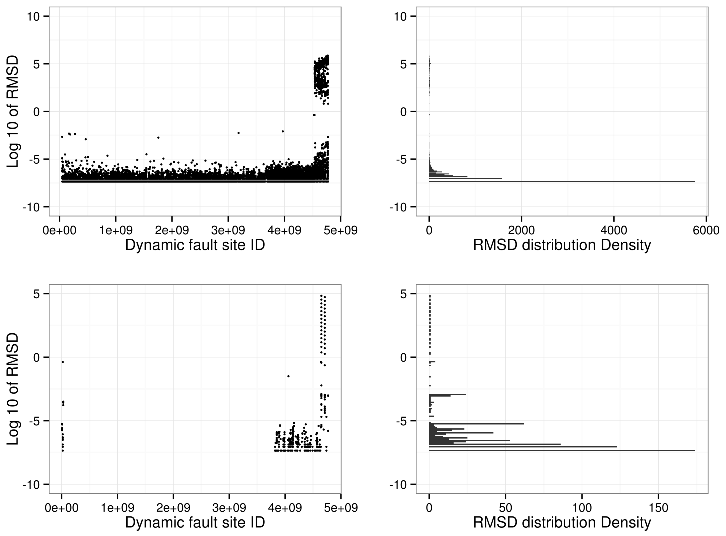


Figure 12: Detailed fault characteristics of LASSO, without (top) and with (bottom) fault resilience (Error checker threshold set to 1e-07).

mechanisms are very useful for recovering from exceptions as well.

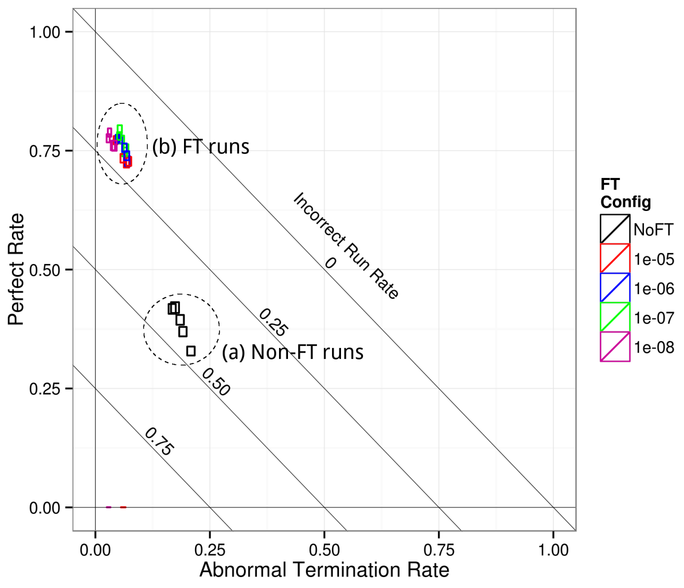
5.2. DRC

Figure 13 shows the error characteristics and running time overhead of DRC before and after applying fault resilience.

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

- The non-fault-tolerant runs of DRC are clustered around the center-left region ((a) in Figure 13), indicating smaller chances of abnormal termination and greater chances of producing perfect results.
- Fault-tolerant runs of DRC are clustered around the top-left region ((b) in Figure 13). Regardless of their thresholds chosen and the computation precision, the error checkers are roughly equally effective.

Overall the characteristics (in terms of the chance of abnormal termination, correct and incorrect answer) of DRC and Lasso are very similar. However, the choice of fault checker threshold does have a much more significant impact on performance on DRC than it does on LASSO. The overhead induced by error checkers increase as the threshold decreases from 1e-05, the most lenient checker threshold, to 1e-08, the tightest checker threshold. In fact, error checker 1e-08 is so tight that it even considers results from a non-faulty run to be incorrect. This is because the 1e-08 is already below the machine error on adding up a large amount of single-precision data. One can see the phenomenon from the detailed temporal RMSD plot in Figure 14. The incorrect runs with RMSDs greater than 1e-6 are largely eliminated.



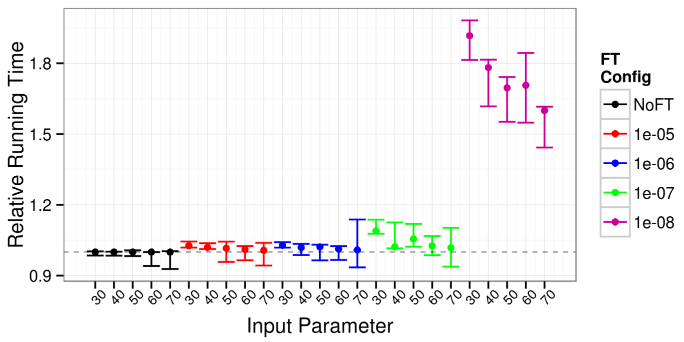


Figure 13: Fault Characteristics and Resilience Overhead of DRC

5.3. Hattrick

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

Figure 15 shows the overall error characteristics of Hattrick. We can observe from the figure that:

- The bottom-right cluster contains non-fault-tolerant runs. They have the highest chance of producing abnormal terminations and incorrect results.
- The top-right cluster contains runs with *only pointer replication*. It slightly reduces the abnormal termination rate while increasing the chance of producing perfect results. From the figures it can be seen the degree of replication has only a slight influence on the outcomes (In contrast, whether or not replication is present has a great influence.)

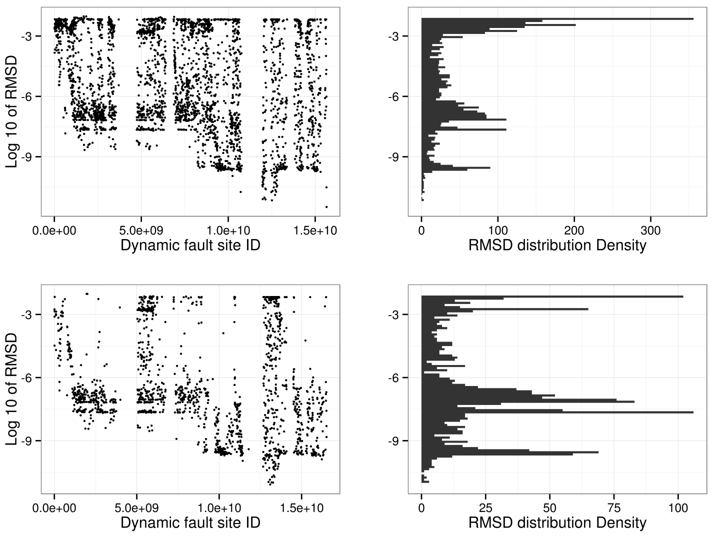


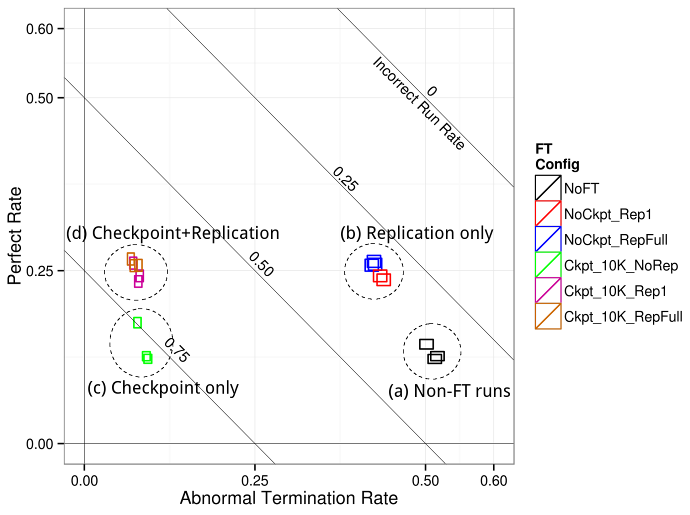
Figure 14: Detailed fault characteristics of DRC, without (top) and with (bottom) fault resilience (Error checker threshold set to 1e-06).

- The bottom-left cluster contains runs with *only checkpointing*. More runs complete but the proportion of perfect runs remained relatively silent.
- The top-left cluster contains runs with *both checkpointing and replication*. They are as tolerant to SegFaults as the cluster(c) and produce as much perfect results as cluster (b).

From the four clusters we can see that checkpointing and replication improve resilience in two orthogonal directions: checkpointing fixes abnormal terminations and "moves" a cluster towards the left. It does not, however, have the ability to raise or maintain the chance of producing correct results. In fact, Figure 16 explains the reason behind this: a single bit-flip error would most likely cause Hattrick to produce a very small error in its outputs (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).

On the other hand, replication is more effective in correcting results. The degree of replication, however, does not significantly improve accuracy.

Performance-wise, checkpointing at a long interval is almost free of overhead. Our study suggested that the usage of an overly small checkpointing interval (less than 100 iterations) may hurt performance without a significant advantage in improving correctness. Replication is much more expensive and the degree of replication has a direct impact on performance overhead.



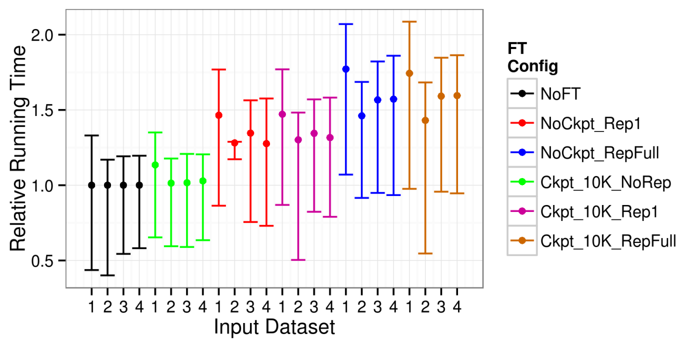


Figure 15: Fault Characteristics and Resilience Overhead of Hattrick. (Inputs 1 to 4 correspond to P2T2090A15, P2T3090A15, P2T4090A15 and P3T2090A11 respectively)

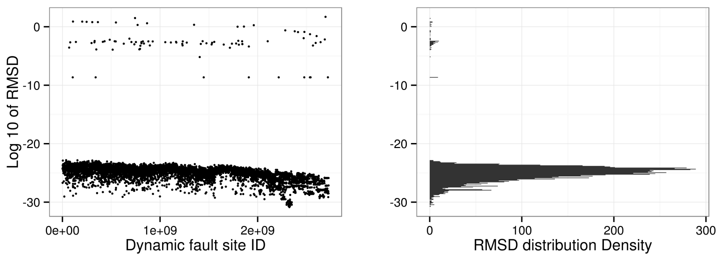


Figure 16: Detailed fault characteristics of Hattrick. The dynamic RMSD plots are almost visually identical for both fault-tolerant and non-fault-tolerant versions and only one figure is shown here for the sake of brevity.

6. Determining Number of Experiments

The effect of errors on applications is an inherently complex process and is difficult to determine the number of fault injection experiments needed to adequately characterize it. In FaultTelescope, we try to characterize an application by relating its dynamic fault site positions to the outcomes. When the outcome is Incorrect Results, the error magnitudes are also considered. The data has been presented in the visualizations in Section 5 and are used to determine the number of experiments. This section shows the process.

FaultTelescope quantifies the amount of relevant information contained in a set of fault injection experiments by modeling its visualizations in terms of a statistical model that takes the available information about a given error injection (e.g. scatterplot x-axis) and predicts the outcome of the error on the application (e.g. scatterplot y-axis). The model uses input in the form of (i) ID of the dynamic fault site, (ii) ID of the static fault site, (iii) index of the flipped bit in the injected instruction's output.

It then categorizes runs into the classes: "Abnormal Termination", "Incorrect Result" and "Correct Result". Finally, for the "Incorrect Result" runs it predicts the RMSD of the result error. Its structure is illustrated in Figure 17.

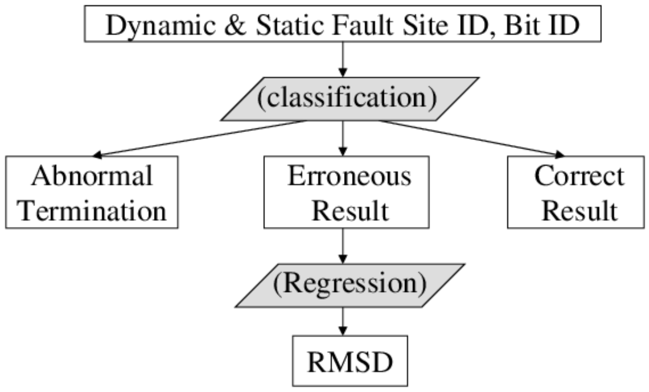


Figure 17: Structure of the FaultTelescope evaluation models. Shaded procedures are where the tree model is applied.

The model's accuracy is evaluated using two metrics:

- 1st-level categorization model: misclassification rate. Since we have 3 categories, a random guess would result in an error rate of 66.7%. With the knowledge of the training set, the tree model should produce a much smaller misclassification rate.

- 2nd-level regression model: R-Square, defined as $1 - \sum_{i=1}^{N} (\hat{y}_i - y_i)^2 \ / \ \sum_{i=1}^{N} (y_i - \bar{y})^2$, which describes 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.)

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-square. 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 characterize the relationship between the injection properties and application outcomes considered by FaultTelescope. Additional improvements in accuracy can only come from adding more features into the analysis, not by running more experiments.

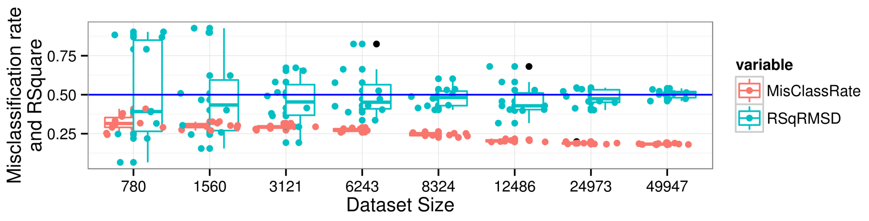


Figure 18: Trend of R-square and misclassification rate as dataset size grows. (A random guess = misclassification rate of 66.7%)

Figure 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 both the misclassification rate and R-square drop steadily 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 and FaultTelescope employs the same procedure for all routines and applications.

7. Conclusion

We present FaultTelescope, a tool that supports application developers in making applications resilient to errors induced by soft faults. FaultTelescope carries out fault injection campaigns, which visualize the relationship between the time a fault occurs and its effect on application results. With statistical analysis on the results, FaultTelescope helps developer draw conclusions on the application'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 applications. The results suggest that an HPC numerical application developer should take the following into consideration when writing fault-resilient programs:

- Algorithm-specific error checkers are effective at detecting incorrect application results, as illustrated in our experiments with MVM, Rank-K update and FFT. During the process the developer should realize that the precision limit of the checker may not be able to correct all results. Example of this is the checker for FFT.
- By recovering applications vulnerable to assertion failures and access violations, the developer can recover many correct runs of Cholesky Decomposition and the Runge-Kutta integrator that would otherwise fail.
- The RK4 Integrator routine demonstrates significantly different characteristics from linear algebra, FFT and FIR routines. It also requires different resilience techniques, 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 applications can be discovered with the FaultTelescope workflow in a similar fashion. We believe FaultTelescope can benefit the production of fault resilient numerical applications.

Acknowledgment

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 to be involved in KULFI and contribute our improvements back to it.

References

[1] R. C. Baumann, "Radiation-Induced Soft Errors in Advanced Semiconductor Technologies," *IEEE Transactions on Device and Materials Reliability*, vol. 5, pp. 305–316, September 2005.

[2] 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*, vol. 5, September 2005.

[3] 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)*, pp. 62–69, 2007.

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

[5] 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)*, pp. 3C.1.1–3C.1.7, 2012.

[6] ITRS, "International Technology Roadmap for Semiconductors," tech. rep., 2013.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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)*, pp. 317–326, 2007.

[12] 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)*, pp. 161–170, 2010.

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

[14] 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*, pp. 1–11, Nov 2010.

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

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

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

[18] 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*, vol. 7156 of *Lecture Notes in Computer Science*, pp. 282–291, Springer Berlin Heidelberg, 2012.

[19] 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*, pp. 1–11, Nov. 2012.

[20] J. Sloan, R. Kumar, and G. Bronevetsky, “Algorithmic approaches to low overhead fault detection for sparse linear algebra,” *Dependable Systems and Networks*, no. Section III, 2012.

[21] 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.

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

[23] M. C. Hsueh, T. K. Tsai, and R. K. Iyer, “Fault Injection Techniques and Tools,” *IEEE Computer*, vol. 30, pp. 75–82, November 1997.

[24] “Kulfi fault injector. https://github.com/quadpixels/kulfi.”

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

[26] 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*, vol. 3, pp. 1–122, June 2011.

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

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

[29] F. S. Foundation, “Gnu scientific library – reference manual,” 2011.