J Supercomput (2015) 71:2963–2984
DOI 10.1007/s11227-015-1422-z

A framework for evaluating comprehensive fault resilience mechanisms in numerical programs

Sui Chen[1] · **Greg Bronevetsky**[2] · **Bin Li**[3] · **Marc Casas Guix**[4] · **Lu Peng**[1]

Published online: 24 April 2015

Abstract As HPC systems approach Exascale, their circuit features 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, causing them to occasionally crash or worse, silently return incorrect results. This is motivating extensive work on program resilience to such faults, ranging from generic mechanisms such as replication or checkpoint/restart to algorithm-specific error detection and resilience mechanisms. Effective use of such mechanisms requires a detailed understanding of (1) which vulnerable parts of the program are most worth protecting and (2) the performance and resilience impact of fault resilience mechanisms on the program. This paper presents FaultTelescope, a tool that combines these two and generates actionable insights by presenting program vulnerabilities and impact of fault resilience mechanisms in an intuitive way.

Keywords Soft faults · High-performance computing · Numerical errors · Fault resilience

1 Introduction

The increasing size and complexity of HPC systems are making them increasingly vulnerable to soft faults, which are transient corruptions of the states of electronic

✉ Lu Peng
lpeng@lsu.edu

1 Division of Electrical and Computer Engineering, Louisiana State University, Baton Rouge, USA

2 Lawrence Livermore National Laboratory, Livermore, USA

3 Department of Experimental Statistics, Louisiana State University, Baton Rouge, USA

4 Barcelona Supercomputing Center, Barcelona, Spain

circuits caused by physical phenomena such as strikes by neutrons, alpha particles [3,24] or thermal electrical noise [19]. They can affect processor latches and registers, which could cause the program to crash or silently return incorrect results [22]. As the feature sizes of electronic circuits shrink, technology scaling will exacerbate soft errors [23] 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 ½ Pitch) of approximately 14 nm [16], which is approximately 28 silicon atoms (5 Å per atom) across. These phenomena make it imperative to develop mechanisms to make HPC systems resilient to soft faults.

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 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 [21]. However, as total system memories are expected to grow by 100$\times$ to 350$\times$ to reach Exascale [9], 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 [27] 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 [12,17], 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 [15,29] 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 [25] and pointer replication [6].

To design and deploy software-level resilience schemes, developers need tools to quantify the effect of faults on their programs and support for choosing the most appropriate resilience mechanism for each type of fault. This paper 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.

The importance of analyzing and quantifying the impact of errors on program behavior is demonstrated in various studies. As Du et al. have shown [10,11], 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. [8] document how the numeric error caused by an injected fault evolves over time. Probabilistic modeling has been used by Chung et al. [7] to help compute the expected recovery time, which cannot be measured easily for very large scale programs. Sloan et al. [30] 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.

The aforementioned resilience studies are enabled by 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 [1,14,20].

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 [18]. 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.

Finally, 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 noticeable 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 [5]. 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 [4] solver for the linear solvers domain, the DRC [28] HiFi audio filter for the signal processing domain, and the Hattrick [26] gravity simulator for the differential equation solvers domain. This paper demonstrates the utility of this comprehensive resilience toolchain for helping developers explore the vulnerability properties of their programs.

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 programs. Section 4 describes the fault resilience mechanisms used. Section 5 presents how FaultTelescope finds fault characteristics and performance/resilience tradeoffs. Section 6 presents the algorithm used for selecting the number of fault injection experiments needed for statistically-grounded analysis. We conclude this paper in Sect. 7.

2 Design overview

The workflow of FaultTelescope is described in Fig. 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 program being terminated.
- Incorrect result: Program runs to completion, but produces results that exceed the user-defined error bound and is considered incorrect. In our paper we quantify the magnitude of errors using the root-mean-squared deviation (RMSD) between the incorrect result and the correct result.

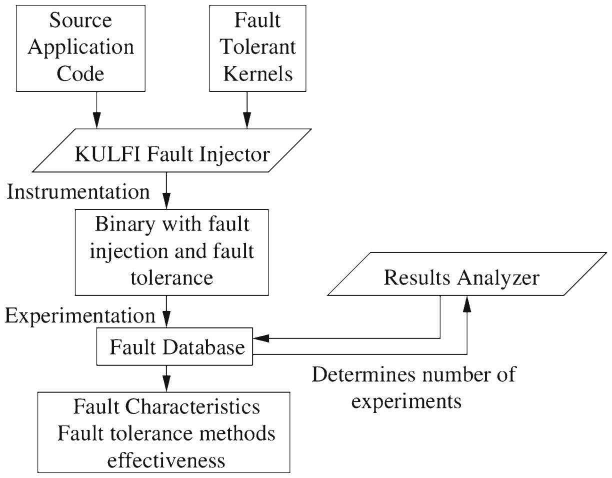

Source Application Code
Fault Tolerant Kernels
KULFI Fault Injector
Instrumentation
Binary with fault injection and fault tolerance
Results Analyzer
Experimentation
Fault Database
Determines number of experiments
Fault Characteristics Fault tolerance methods effectiveness

Fig. 1 Overall workflow of FaultTelescope

Table 1 The resilience mechanisms applied to each major routine of each program

Routine		Algorithmic detector	Checkpointing and pointer replication	
ADDR	MM	Linear encoding Thresholds: 1e−5 to 1e−8	Inputs	No
	SYRK			
	MVM			
	CD			
FRC	FFT	Parseval's theorem. Sum conservation. Thresholds: 1e−6 to 1e−8	Inputs	No
	FIR			
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

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 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 1. The details of the mechanisms will be discussed in the Sect. 4.

3.1 LASSO

The LASSO [4] 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) [13]: 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.

3.2 DRC

Dynamic range compression (DRC) [28] 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 paper is a PCM audio file of size 768 KB, which is internally resampled at 30, 40, 50, 60 or 70 KHz during computation.

3.3 Hattrick

Hattrick [26] 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.

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 1 protect the programs from soft errors. The fault characteristics are quantified by the probability of outcomes which are correct result, abnormal termination and incorrect result.

In Sects. 4 and 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 $\mathrm{RMSD}(x, x_{\mathrm{gold}}) = \sqrt{\frac{\sum_{t=1}^{n}(x_t - x_{\mathrm{gold},t})^2}{n}}$, 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

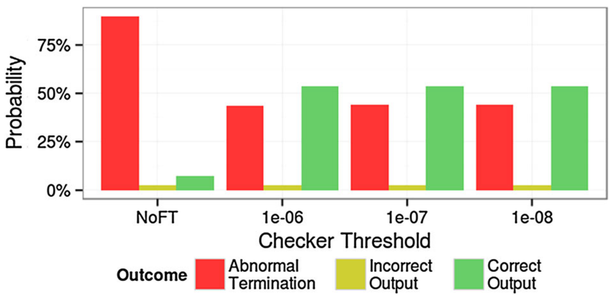


Fig. 2 Fault characteristics of Cholesky decomposition given input size 500 × 500

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.

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.

4.2 Algorithmic error detection

4.2.1 Cholesky decomposition (CD)

The Cholesky decomposition is a decomposition of the form $A = LL^{\mathrm{T}}$, where L is lower-triangular with a positive diagonal. This operation must maintain the identity $Ax = L(L^{\mathrm{T}}x)$ [15], 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.

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 Fig. 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

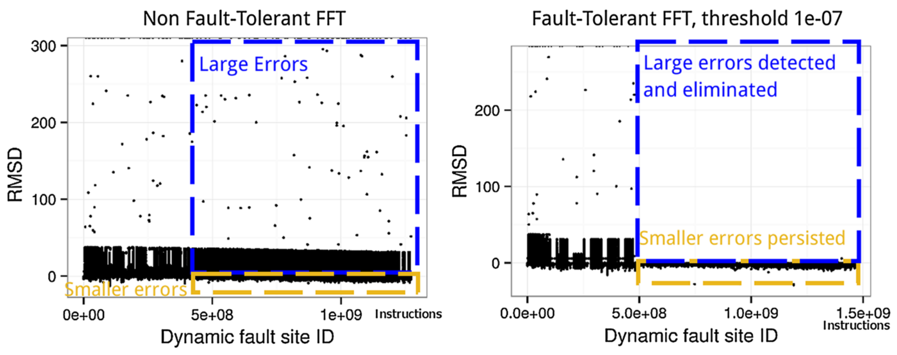


Fig. 3 Detailed characteristics of fault resilient FFT with checker threshold 1e−07

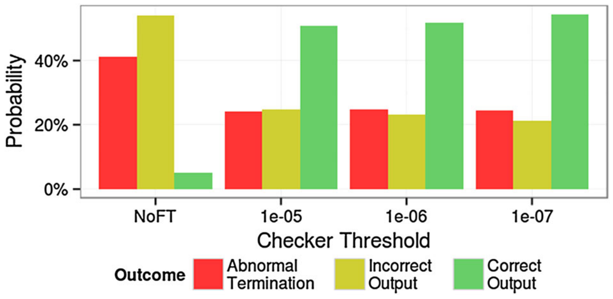


Fig. 4 Fault characteristics of FFT given input size 4 m

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 Sect. 5.

4.2.2 Fast Fourier transform (FFT)

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.

This check takes $O(n)$ operations, which is smaller than $O(n\log(n))$ or $O(n^2)$ for the FFT algorithm, depending on the FFT radix (For example, for a radix $n = 2 \cdot 3 \cdot 19{,}999$ transform, the $O(n^2)$ scaling would dominate).

Figure 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 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).

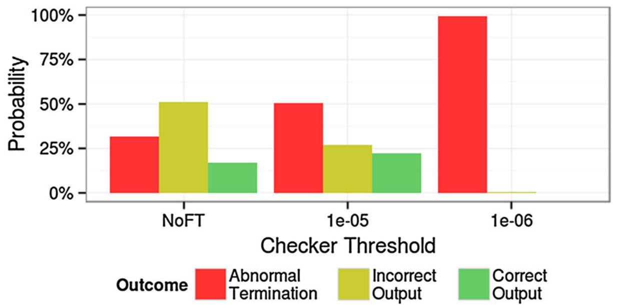


Fig. 5 Fault characteristics of FIR of input parameter 512 K

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 Sect. 5.

4.2.3 Finite impulse response filter generation (FIR)

This algorithm generates a sample of the function $\text{sinc}(x) = \frac{\sin(x)}{x}$ and modulates it with a Blackman window. The result is checked using the invariant $\int_{-\infty}^{\infty} \text{sinc}(x)\text{d}x = 1$, 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.

The checker threshold 1e−06 is too tight and causes many false alarms, resulting in many runs terminated, as can be seen in Fig. 5. We choose 1e−05 as the error checker threshold for FIR for it increases the probability of correct outputs.

4.2.4 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 1 s). The checker takes $O(n^2)$ operations and is asymptotically faster than MMM which takes $O(n^3)$ operations.

Figure 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.

4.2.5 Symmetric Rank-K update

Symmetric Rank-K update (SYRK) computes $\alpha AA^{\text{T}} + \beta B$, where A and B are matrices (Fig. 7).

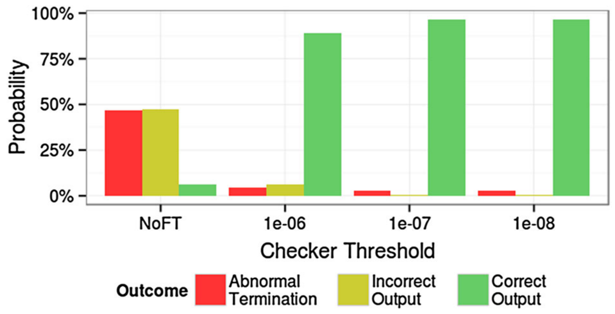


Fig. 6 Fault characteristics of matrix–matrix multiplication with input size 500 × 500

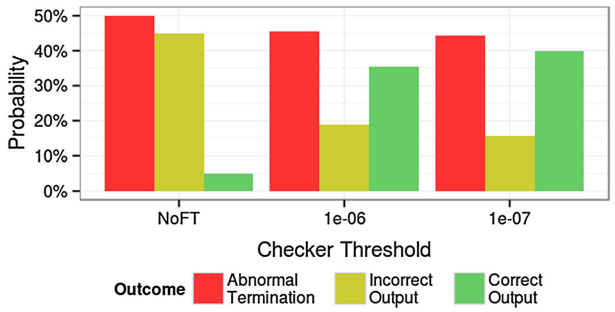


Fig. 7 Fault characteristics of symmetric Rank-K update with input size 500 × 500

The result is checked via the identity $(AA^{\mathrm{T}})x = A(A^{\mathrm{T}}x)$, where x is an error-checking vector. We use a vector of all 1 s. 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 Fig. 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.

We use 1e−06, 1e−07 and 1e−08 as the error checker thresholds for RK in the experiments in Sect. 5.

4.2.6 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 $(x^{\mathrm{T}}A)x = x^{\mathrm{T}}(Ax)$. The complexity of computing $x^{\mathrm{T}}A$ 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 $x^{\mathrm{T}}A$ can be reused, amortizing the cost.

We use 1e−06, 1e−07 and 1e−08 as the error detector thresholds for MV in the experiments. Figure 8 summarizes the fault characteristics of matrix–vector

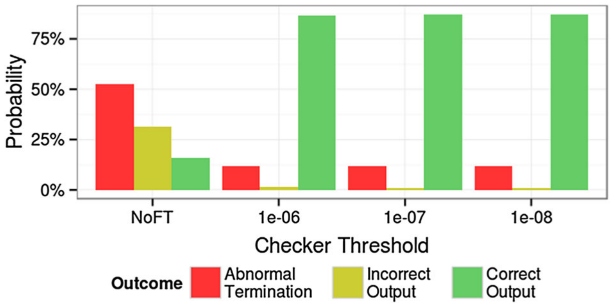


Fig. 8 Fault characteristics of matrix–vector multiplication with input size 500 × 500

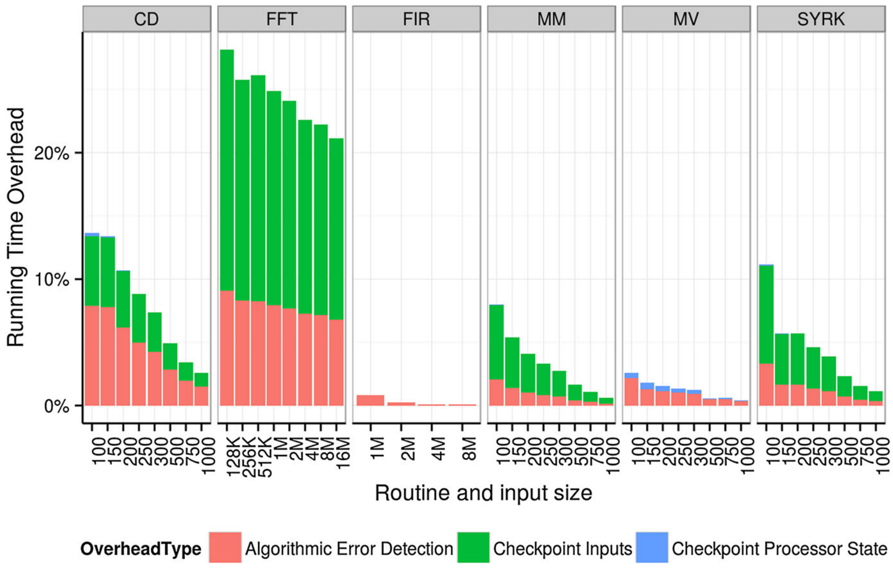


Fig. 9 Overhead of fault resilience mechanisms for linear algebra kernels, FFT and FIR

multiplication. The performance overhead of the algorithmic checks for linear algebra, FFT and FIR routines are listed in Fig. 9. The Runge-Kutta integrator will be discussed separately since its resilience is not achieved through algorithmic invariants.

4.2.7 Runge–Kutta integrator

The 4th order Runge–Kutta method (RK4) is a method for numerically solving an ordinary differential equations of the form $\frac{\mathrm{d}y}{\mathrm{d}x} = f(y, x)$. It advances the variable x by steps of size h and computes the value y at the next point $x + h$ using the derivative $\frac{\mathrm{d}y}{\mathrm{d}x}$ 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

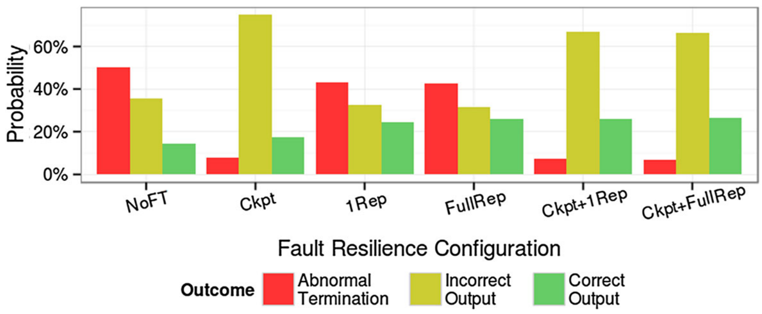


Fig. 10 Fault characteristics of the RK4 integrator

Table 2 Overhead of different versions of the RK4 integrator

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

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 10,000 makes an optimal balance. This routine is tested with the second-order nonlinear Van der Pol oscillator equation in the GSL documentation [13]. Its resilience properties are shown in Fig. 10 and Table 2 respectively.

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.

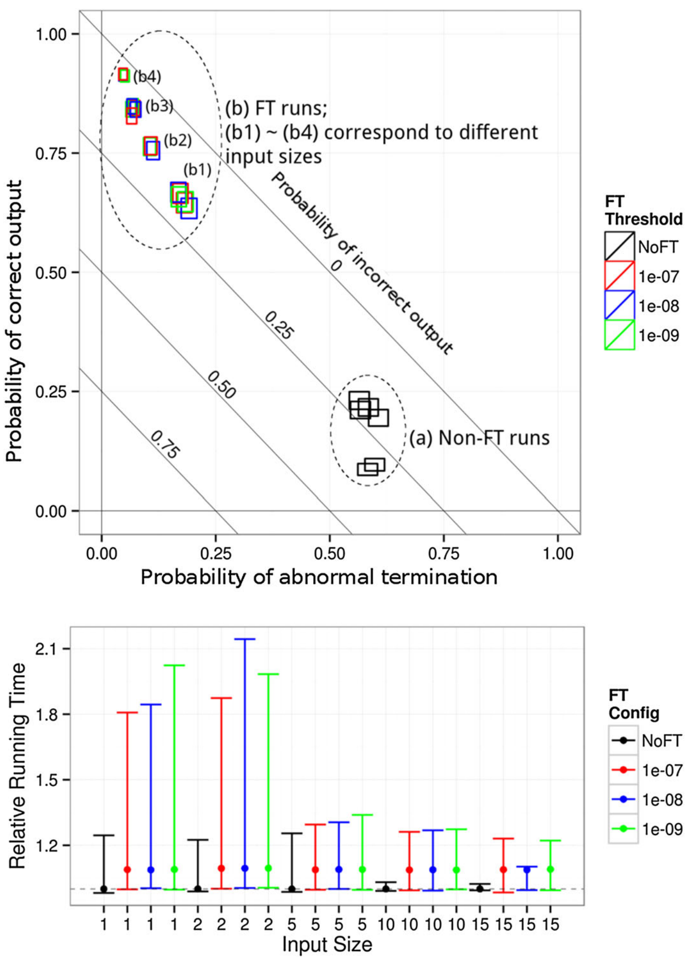


Fig. 11 Fault characteristics and resilience overhead of Lasso

5.1 Lasso

Figure 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 effectiveness of the fault resilience techniques:

- The rectangles around the bottom-right cluster represent runs of non-fault-tolerant (Non-FT) Lasso ((a) in Fig. 11). For these runs, the probability of abnormal termination is high and the probability of producing correct results is low.
- The rectangles around the top-left cluster represent fault-tolerant Lasso ((b) in Fig. 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 ($b1 = \{40, 80\} \cdot 500$, $b2 = 200 \cdot 500$, $b3 =$

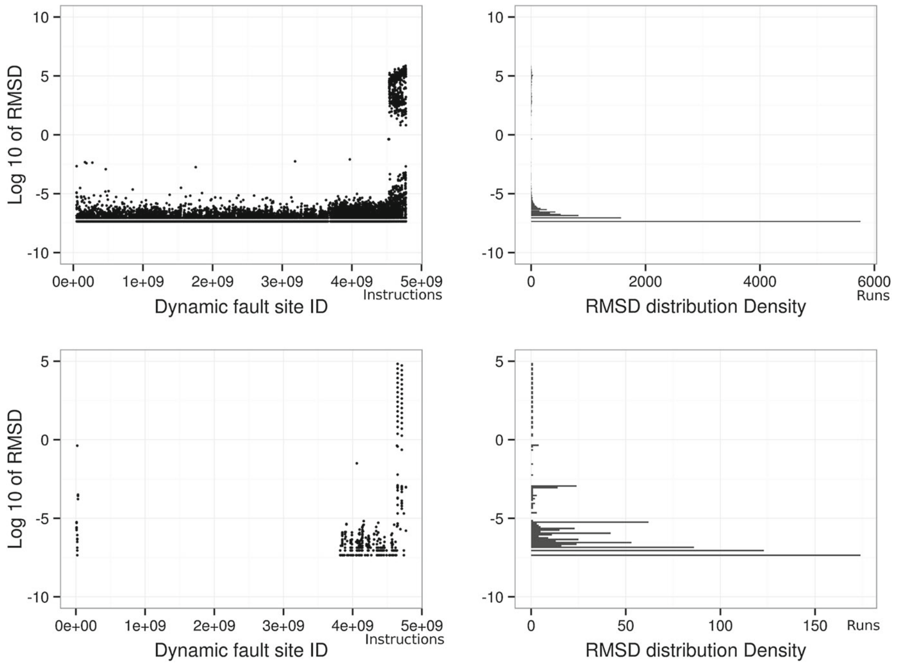


Fig. 12 Detailed fault characteristics of LASSO, without (*top*) and with (*bottom*) fault resilience (error checker threshold set to 1e−07)

$\{400, 600\} \cdot 500$, $b4 = 800 \cdot 500$). The temporal error graph of the input size $\{20, 500\}$ are shown in Fig. 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.

5.2 DRC

Figure 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:

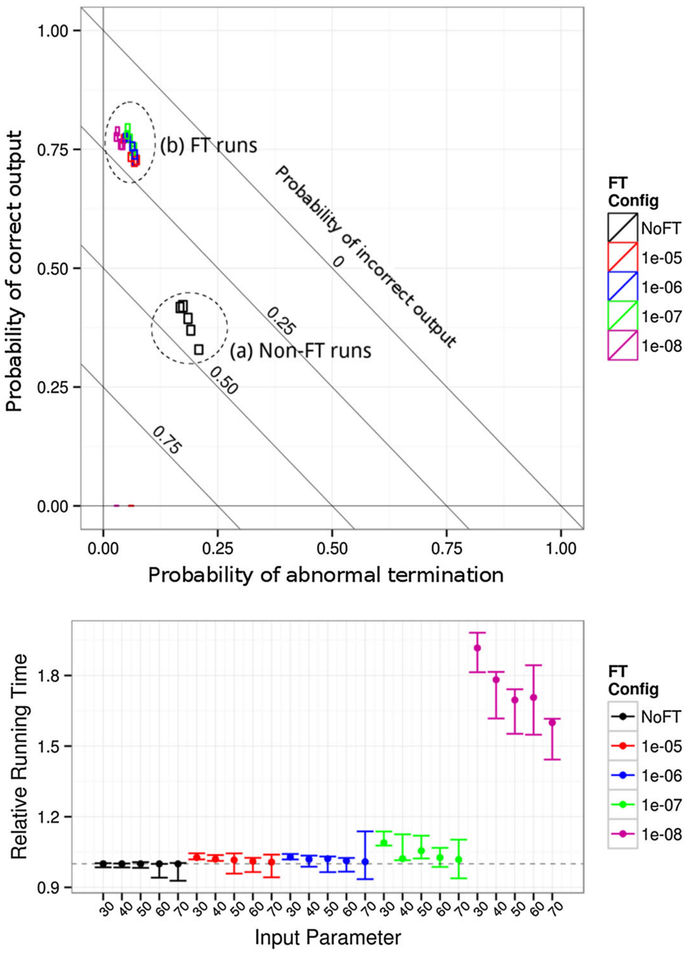


Fig. 13 Fault characteristics and resilience overhead of DRC

- Runs of the non-fault-tolerant DRC are clustered around the center-left region ((a) in Fig. 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 Fig. 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

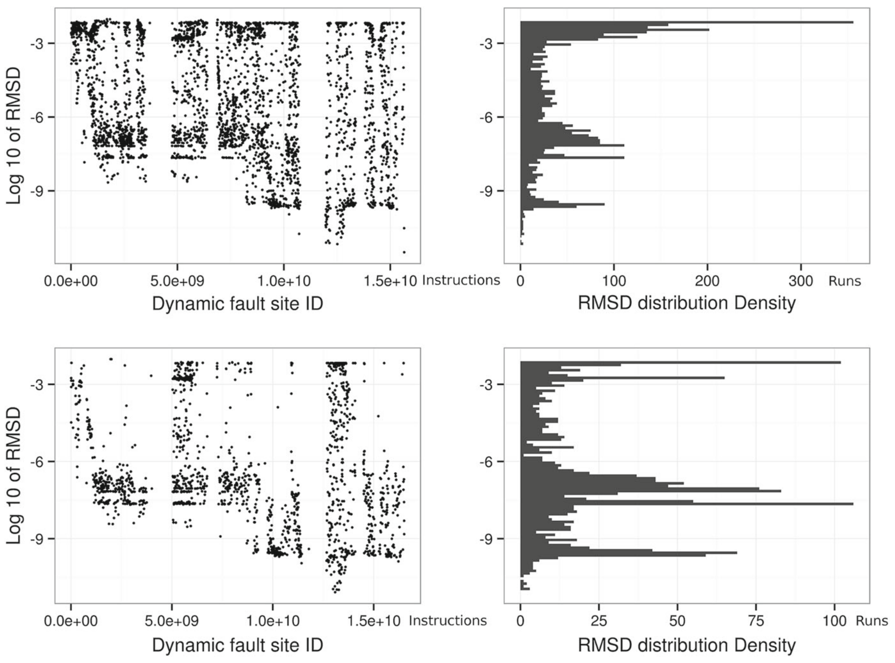


Fig. 14 Detailed fault characteristics of the original (*top*) and fault-tolerant (*bottom*) DRC (error checker threshold set to 1e−06)

detailed temporal error graph in Fig. 14, many incorrect runs with RMSDs greater than 1e−06 are corrected.

5.3 Hattrick

The Hattrick program 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 (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.
- 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).

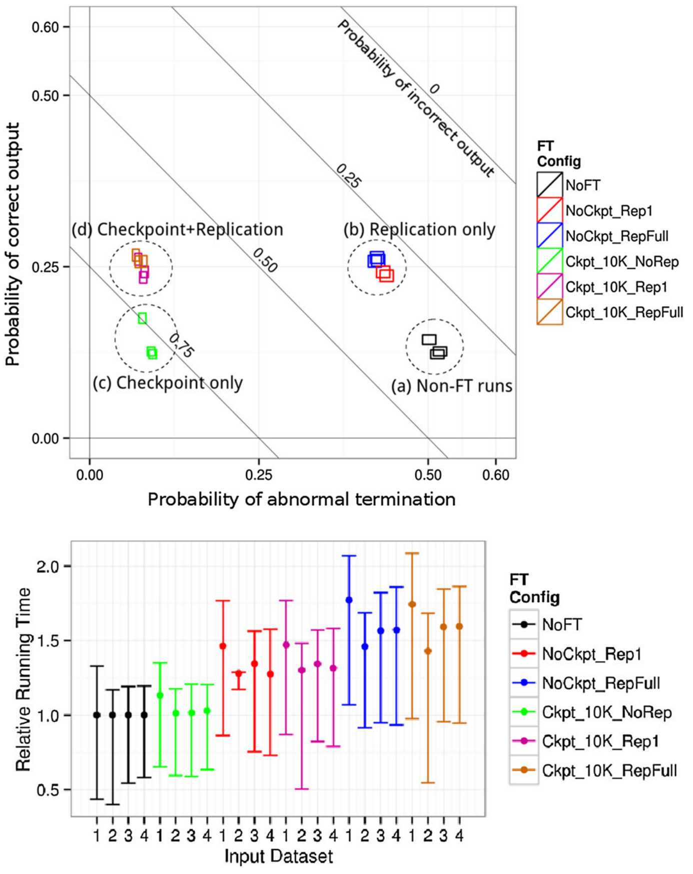


Fig. 15 Fault characteristics and resilience overhead of Hattrick. (Inputs 1–4 correspond to P2T2090A15, P2T3090A15, P2T4090A15 and P3T2090A11 respectively)

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 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 10,000 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

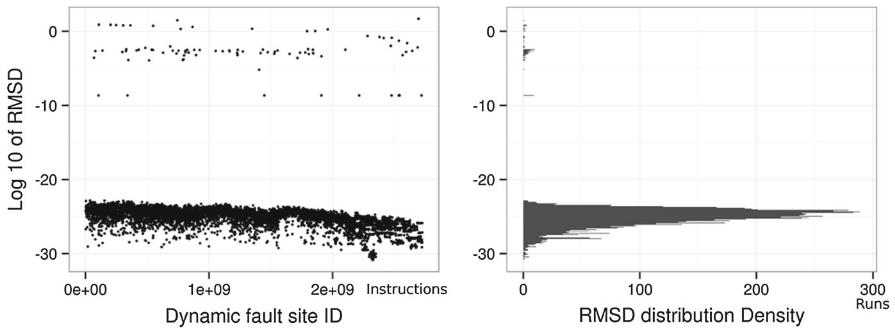


Fig. 16 Detailed fault characteristics of Hattrick. The dynamic RMSD plots are almost visually identical for both fault-tolerant and non-fault-tolerant versions and only one figure is shown here for the sake of brevity

the other hand, replication incurs greater performance overhead than checkpointing. In case of a high degree of replication, the overhead is even higher.

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.

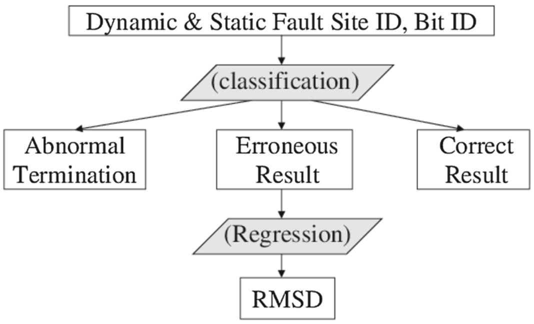
We start by observing the correlation between the Dynamic Fault Site ID and the error magnitude in incorrect results. To illustrate the correlation, the temporal error graphs in Figs. 3, 12, 14 and 16 show that faults with Dynamic Fault Site IDs close to each other are likely to incur errors of similar 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 Fig. 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

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



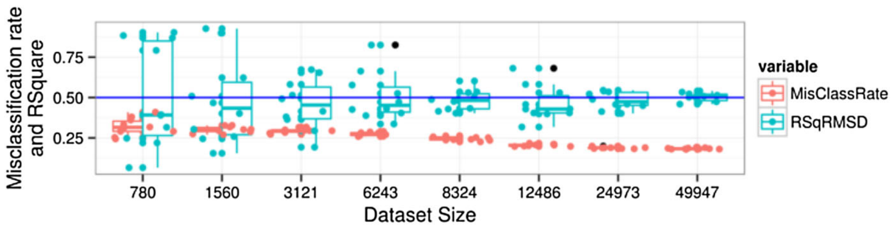


Fig. 18 Trend of R-Square and misclassification rate as dataset size grows. (A random guess = misclassification rate of 66.7 %)

a misclassification rate smaller than 66.67 %. A lower misclassification means a more accurate model.

- *2nd-level regression model* R-Squared, or $1 - \sum_{i=1}^{N} (\hat{y}_i - y_i)^2 / \sum_{i=1}^{N} (y_i - \bar{y})^2$, 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). 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 18 illustrates the procedure using experiments on the matrix vector multiplication routine, executed on 500×500 matrices. As the number of fault injection experiments increases, we see that the misclassification rate drops while the R-square converges steadily until they stabilize at a sample size of 49,947 experiments. As the data shows, this sample size is sufficient for the purposes of FaultTelescope's visualization and is much smaller than the ~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.

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.

Acknowledgments 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.

References

1. Gi D, Martinez R, Busquets JV, Baraza JC, Gil PJ (1999) Fault Injection into VHDL models: experimental validation of a fault-tolerant microcomputer system. In: European Dependable Computing Conference, pp 191–208
2. (2013) Kulfi fault injector. https://github.com/quadpixels/KULFI
3. Baumann RC (2005) Radiation-induced soft errors in advanced semiconductor technologies. IEEE Trans. Device Mater. Reliab. 5(3):305–316

4. Boyd S, Parikh N, Chu E, Peleato B, Eckstein J (2011) Distributed optimization and statistical learning via the alternating direction method of multipliers. Found. Trends Mach. Learn. 3(1):1–122
5. Bronevetsky G, de Supinski B (2008) Soft error vulnerability of iterative linear algebra methods. In: International Conference on Supercomputing
6. Casas M, de Supinski BR, Bronevetsky G, Schulz M (2012) Fault resilience of the algebraic multi-grid solver. In: International Conference on Supercomputing, pp 91–100
7. Chung J, Lee I, Sullivan M, Ryoo JH, Kim DW, Yoon DH, Kaplan L, Erez M (2012) Containment domains: a scalable, efficient, and flexible resilience scheme for exascale systems. In: 2012 international conference for high performance computing, networking, storage and analysis, pp 1–11. doi:10.1109/SC.2012.36, http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6468537
8. DeBardeleben N, Blanchard S, Guan Q, Zhang Z, Fu S (2012) Experimental framework for injecting logic errors in a virtual machine to profile applications for soft error resilience. In: Euro-Par 2011: Parallel Processing Workshops, Lecture Notes in Computer Science, vol 7156. Springer, Berlin, pp 282–291. doi:10.1007/978-3-642-29740-3_32
9. DeBardeleben N, Blanchard S, Sridharan V, Gurumurthi S, Stearley J, Ferreira K (2014) Extra bits on sram and dram errors—more data from the field. In: IEEE Workshop on Silicon Errors in Logic-System Effects (SELSE)
10. Du P, Luszczek P, Dongarra J (2011) High performance dense linear system solver with soft error resilience. In: 2011 IEEE International Conference on Cluster Computing, pp 272–280. doi:10.1109/CLUSTER.2011.38. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6061145
11. Du P, Luszczek P, Dongarra J (2012) High performance dense linear system solver with resilience to multiple soft errors. In: Procedia Computer Science, vol. 9, pp. 216–225. doi:10.1016/j.procs.2012.04.023. http://www.sciencedirect.com/science/article/pii/S1877050912001445
12. Ferreira K, Stearley J, James H, Laros I, Oldfield R, Pedretti K, Brightwell R, Riesen R, Bridges PG, Arnold D (2011) Evaluating the viability of process replication reliability for exascale systems. In: Supercomputing
13. Foundation FS (2011) Gnu scientific library—reference manual
14. Hsueh MC, Tsai TK, Iyer RK (1997) Fault injection techniques and tools. IEEE Comput 30(4):75–82
15. Huang KH, Abraham JA (2010) Algorithm-based fault tolerance for matrix operations. In: International Conference on Dependable Systems and Networks (DSN), pp 161–170
16. ITRS (2013) International technology roadmap for semiconductors. Tech. rep
17. LaFrieda C, Ipek E, Martinez JF, Manohar R (2007) Utilizing dynamically coupled cores to form a resilient chip multiprocessor. In: International conference on dependable systems and networks (DSN), pp 317–326
18. Lattner C, Adve V (2004) Llvm: a compilation framework for lifelong program analysis and transformation. San Jose, CA, pp 75–88
19. Li H, Mundy J, Patterson W, Kazazis D, Zaslavsky A, Bahar RI (2007) Thermally-induced soft errors in nanoscale CMOS circuits. In: IEEE international symposium on nanoscale architectures (NANOARCH), pp 62–69
20. Li ML, Ramachandran P, Karpuzcu UR, Kumar S, Hari S, Adve SV (2009) Accurate microarchitecture-level fault modeling for studying hardware faults. In: International symposiumn on high-performance computer architecture
21. Li X, Huang MC, Shen K, Chu L (2010) A realistic evaluation of memory hardware errors and software system susceptibility. In: USENIX annual technical conference
22. da Lu C, Reed DA (2004) Assessing fault sensitivity in MPI applications. In: Supercomputing
23. Massengill LW, Bhuva BL, Holman WT, Alles ML, Loveless TD (2012) Technology scaling and soft error reliability. In: IEEE Reliability Physics Symposium (IRPS), pp 3C.1.1–3C.1.7
24. Michalak S, Harris KW, Hengartner NW, Takala BE, Wender SA (2005) Predicting the Number of Fatal Soft Errors in Los Alamos National Laboratory's ASC Q Supercomputer. IEEE Trans Device Mater Reliab 5(3):329–335
25. Moody A, Bronevetsky G, Mohror K, De Supinski B (2010) Design, modeling, and evaluation of a scalable multi-level checkpointing system. In: 2010 international conference for high performance computing, networking, storage and analysis (SC), pp 1–11. doi:10.1109/SC.2010.18
26. Olson WT (2014) Hattrick n-body simulator. http://code.google.com/p/hattrick-nbody
27. Reinhardt SK, Mukherjee SS (2000) Transient fault detection via simultaneous multithreading. In: International symposium on computer architecture (ISCA), pp 25–36
28. Sbragion D (2014) Drc: digital room correction. http://drc-fir.sourceforge.net/

29. Sloan J, Kesler D, Kumar R, Rahimi A (2010) 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
30. Sloan J, Kumar R, Bronevetsky G (2012) Algorithmic approaches to low overhead fault detection for sparse linear algebra. In: Dependable systems and networks (Section III). http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6263938