FaultSight: A Fault Analysis Tool for HPC Researchers

Einar Horn
Department of Linguistics
University of Washington
Seattle, Washington, USA
einarh@uw.edu

Dakota Fulp
Jon C. Calhoun
Holcombe Department of Electrical and Computer Engineering
Clemson University
Clemson, South Carolina, USA
{dakotaf,jonccal}@clemson.edu

Luke N. Olson
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA
lukeo@illinois.edu

ABSTRACT

System reliability is expected to be a significant challenge for future extreme-scale systems. Poor reliability results in a higher frequency of interruptions in high-performance computer (HPC) applications due to system/application crashes or data corruption due to soft errors. In response, application level error detection and recovery schemes are devised to mitigate the impact of these interruptions. Evaluating these schemes and the reliability of an application requires the analysis of thousands of fault injection trials, resulting in tedious and time-consuming process. Furthermore, there is no one data analysis tool that can work with all of the fault injection frameworks currently in use. In this paper, we present FaultSight, a fault injection analysis tool capable of efficiently assisting in the analysis of HPC application reliability as well as the effectiveness of resiliency schemes. FaultSight is designed to be flexible and work with data coming from a variety of fault injection frameworks. The effectiveness of FaultSight is demonstrated by exploring the reliability of different versions of the Matrix-Matrix Multiplication kernel using two different fault injection tools. In addition, the detection and recovery schemes are highlighted for the HPCCG mini-app.

KEYWORDS

soft error analysis, fault analysis tool, fault tolerance, resiliency, fault injection, fault analysis

ACM Reference Format:
Einar Horn, Dakota Fulp, Jon C. Calhoun, and Luke N. Olson. 2019. FaultSight: A Fault Analysis Tool for HPC Researchers. In *Proceedings of FTXS'19:Fault Tolerance for HPC at eXtreme Scale (FTXS) Workshop (FTXS'19).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.475/123_4

1 INTRODUCTION

High-performance computing (HPC) systems are crucial to scientific simulation and analysis. As HPC resources continue to play a role in scientific discoveries, simulations are increasing in size and complexity. Equally, in order to meet these workload demands, HPC systems continue to grow in size and computational power. As HPC systems become larger and more powerful, design concerns such as power usage and reliability may inhibit the ability of next generation HPC systems [7, 35, 38].

Reliability is a significant challenge for HPC systems and applications. To overcome fail-stop failures, checkpoint restart routines are integrated into most applications. Checkpoint restart allows applications to save a portion of the computational state and use it to recover from this saved state when a failure occurs [5]. However, as processor feature size continues to shrink and power constraints force hardware to operate at near-threshold voltages, the occurrence of soft errors increases [2, 9, 27]. Soft errors occur from charged particles or hardware degradation thus causing bit-flips in data, computation, and control logic. If left undetected, soft errors can corrupt application state, leading to silent data corruption (SDC) that may alter the results of an application. Additionally, the time-to-solution may also increase even in situations where data is left uncorrupted.

Full protection of HPC systems from soft errors at the hardware level is prohibitively expensive. Therefore, protection is mainly confined to the memory system in the form of parity bits and error correcting codes (ECC) on data paths and storage. General protection from soft errors at the application level relies on replication [8, 20, 28], data analysis [3, 4], or leveraging algorithm modifications [12, 16, 25]. Recovery schemes rely on checkpoint restart [4, 12], replication [20], or forward recovery techniques [8, 26].

Measuring an application's reliability to soft errors requires conducting large numbers of fault injection trials. Efficient evaluation of these trials is difficult, due to the lack of analysis tools for fault injection campaigns. This makes it difficult to determine the vulnerable code regions. Soft error detection and recovery schemes also require thousands of fault injection trials, and likewise do not have any common tools to aid in their analysis.

This paper presents FaultSight, a fault analysis tool designed to aid the development and analysis of application-level soft error detection and recovery schemes. FaultSight provides an API that allows data from a campaign of fault injection trials to be inserted into an extendable database. The database is queried by an accompanying web-app that couples dynamic queries and visualizations to obtain a fine-grained view of the fault injection campaign. Qualitative and statistical analysis are then used to determine the baseline reliability of an application and the effectiveness of integrated detection and recovery schemes.

This paper makes the following contributions:

978-1-7281-6013-9/19/$31.00 ©2019 IEEE
DOI 10.1109/FTXS49593.2019.00008

- an overview of the FaultSight package, a user extendable fault analysis tool that allows for efficient analysis of fault injection data for HPC applications.
- examples of fault injection analysis using FaultSight, including bit-flips in computation and how to use FaultSight with bit-flips in memory.
- an analysis of the resiliency of three Matrix-Matrix Multiplication algorithms through use and comparison of FlipIt [11] and FSEFI [23]. As well as an analysis of detection and recovery scheme for the mini-app HPCCG.

2 BACKGROUND AND MOTIVATION

Determining the reliability of an application and evaluating the improvement in reliability due to a detection or recovery scheme requires thousands of fault injection experiments. Complex HPC applications result in millions of possible fault locations. To obtain statistically significant results, thousands of fault injection trials are conducted [29]. This process can be seen in the left-hand side of Figure 1. Even through parallel execution, the thousands of fault injection trials use significant computational resources and storage.

Upon completion of these trials, aggregation and analysis of each trial can be achieved. Some metrics of interest include: the return code of the application, the acceptability of the application's results, the impact on application runtime, the detection of injected faults, and the recovery from detected faults. These metrics are analyzed generically for the whole application. However, this has a limited view of *when* and *where* a fault is injected and how that impacts these metrics. Deeper analysis of *when* and *where* a fault is injected is complicated by the lack of fault analysis tools. This paper addresses the lack of such tools by presenting FaultSight, a tool designed to conduct detailed analysis of fault injection campaigns.

FaultSight is capable of exploring the importance of *when* and *where* faults occur inside of a fault injection campaign and the associated impact on key application metrics. FaultSight's contributions to the fault injection and analysis workflow are shown in Figure 1. FaultSight is structured as a general tool capable of incorporating data from many different fault injection frameworks and provides a web application to efficiently analyze the whole campaign or a specific subset of the trials. Using FaultSight to analyze a fault injection campaign allows application developers/users to determine the vulnerable regions of their application and quantify the efficacy of an integrated soft error detection and recovery scheme.

3 FAULTSIGHT DESIGN

FaultSight is a tool that enables the analysis of HPC resiliency in applications by quantifying an applications reliability, identifying where to add protection mechanisms, and assisting in the evaluation of application-based detection and recovery schemes. FaultSight aims to integrate seamlessly with most HPC fault injectors while providing a comprehensive set of functionality to assist with the analysis of high-level and fine-grained fault injection campaign results. An interactive web application presents the analysis and allows quick observations about the reliability of the tested application and the effectiveness of integrated detection/recovery schemes.

FaultSight operates first by parsing fault injection campaign data into an extendable database through its built-in API calls. The large range of API endpoints allows FaultSight to support data coming from most fault injectors. Upon construction, FaultSight queries the database to make high-level statistical analysis and visualizations.

Overall, the structure of FaultSight involves three main components: the database, the API for database communication, and the web interface used for analytics.

3.1 Database

The database is designed in a general manner and supports a variety of fault injector types — e.g. instruction-based, memory-based, and fail-stop injectors. Section 4 explores FaultSight's ability to analyze results from two different instruction-based fault injectors. The database consists of five extendable tables: Site, Trials, Injections, Signals, and Detections. Each table is discussed below, including their utility for different types of fault injectors while an example of database creation can be seen in Algorithm 1.

3.1.1 Sites. The Sites table allows FaultSight to support a wide variety of fault injector types. The inclusion of *threadId* and *rank* fields allows FaultSight to support multi-threaded and parallel applications. For instruction-based fault injectors, the *type* field allows information about the site's instruction classification — e.g. floating-point, fixed-point, address calculation — to be stored. In memory-based fault injectors, the *type* field is instead used to denote the data type for the memory allocation or the memory address where the fault occurred. In both instruction and memory based injectors, information on a site's location in the source code can be stored via the *file*, *func*, and *line* fields for for subsequent visualization and analysis. Details on the functionality of the site are stored via the opcode field. For instruction-based injectors this can be used to denote the injection instruction. Not all fields will be used with all fault injectors and are prepopulated with `NULL` values. An example of an insertion into this table can be seen in Algorithm 1.

3.1.2 Trials. The Trials table stores information on each of the fault injection trials. In addition to assigning a unique identifier to each trial, this table holds properties of each trial, including the number of injections per trial, as well as a boolean value to represent the return code of the application. Extending this table allows users to store specific information about the trials such as wall-clock time, power/energy, and resource utilization statistics which results in a more detailed view of each trial.

3.1.3 Injections. The Injections table stores information on each injection that occurs during the fault injection campaign and associates it with a unique trial ID. Information such as an injection's site, process rank, and thread identifier assists in mapping back to the fault injection location. Additionally, instruction and memory based injectors can store information on any corrupted bits, injection probability, and the static instruction of the application during the injection. Through the use of this table, FaultSight can perform fine-grained analysis on this data. An insertion into this table can be seen in Algorithm 1.

3.1.4 Signals. The Signals table stores information on known signals that are raised during trials. In addition to the Unix signal

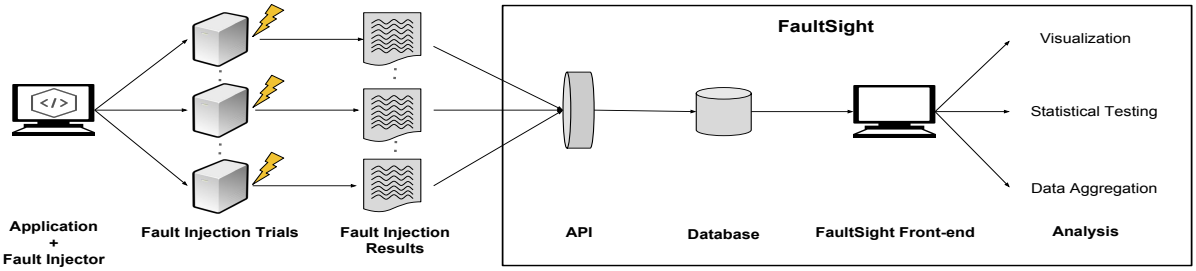


Figure 1: Overview fault injection and the FaultSight tool.

number, information on which process and thread raised the signal can be stored. This table is useful in distinguishing how a program terminates unexpectedly such as through segmentation fault violations or failed assertions. Also knowing if the signals are raised on the same process/thread where an injection occurred can help analyze error propagation.

3.1.5 Detections. The Detections table enables the ability to store information on the performance of implemented algorithm-based fault detectors when testing their effectiveness. By combining the process/thread that detected the error/failure with the process/thread where the injection occurred, a spatial map of the events occurring inside the trial after each injection highlights how the error propagates. If temporal information is known, the latency — e.g., number of instructions since injection, number of iterations since injection, elapsed time since injection — provides another dimension to explore when addressing a detector's effectiveness. The spatial and temporal properties of detection combined with application knowledge allows for estimating the extent at which error due to the fault propagates. Knowledge of error propagation is useful in the design of custom local recovery schemes.

3.2 API

FaultSight's API supports data coming from various injection styles and frameworks. The API is a collection of Python functions that wrap low-level SQL commands. Through the API, the user experience is abstracted and simplified. Thus, allowing users to generate the database more intuitively.

Algorithm 1 Creation of a FaultSight database.

```
Input sites_data : a list of site information
Input injections_data : a list of injection information

# Create a database object and get a reference to it
db ← create_and_connect_database("./database.db")

# Extend injections table to hold number of incorrect elements
extend_injections_table(db, "incorrect_count", "INTEGER")

# Insert site information into the database
for site_information in sites_data do
    insert_site(db, site_information)

# Insert injection information into the database
for injection_information in injections_data do
    start_trial(db)
    insert_injection(db, injection_information)
    end_trial(db)
```

The API facilitates the generation of the database online during the fault injection process as well as offline once the campaign has been completed. For the experiments in Section 4, both FlipIt [11], an LLVM instruction-based fault injection tool, and FSEFI [23], a sequential fault injection tool, are used. However, other HPC fault injection frameworks [31, 36] can be used.

During fault injection, FlipIt and FSEFI each record a variety of data on each fault injection trial, such as the instruction and bit-location in which the fault is injected. A Python script is developed to map each of their output to the appropriate API calls upon completion of each fault injection campaign. Other fault injectors [31, 36] provide similar information, such as memory addresses of the injection for memory-based injectors, which can also be stored in the database through the API. An example of generating a database using the API is shown in Algorithm 1.

To analyze the fault injection trials in more detail, the API allows users to make database schema changes such as extending tables with additional fields to better support their situation. Section 4 exploits this capability to generate databases for the Matrix-Matrix Multiplication and HPCCG applications.

3.3 Web Application

FaultSight interacts with the database through a web application that runs locally and uses Flask[1], a flexible Python web framework. The back-end uses SQLAlchemy[2], a SQL toolkit and object-relational mapper, to query the database. To simplify the selection of data for visualization and analysis, FaultSight uses a dedicated web application over other data analytics tools. There are three main features of the web application: Statistical Analysis, Charts, and Code Analysis.

3.3.1 Statistical Analysis. FaultSight provides the end-user with two forms of statistical analysis: independent analysis and comparative analysis.

Independent analysis allows users to determine if there have been enough injections on a given type or location to justify meaningful results. For example, to accurately evaluate the reliability of function *foo* a sufficient number of injections into *foo*'s fault injection space are required. A key question for instruction-based fault injectors is whether the dynamic/static instruction percentages match the percentage of instruction types that suffer faults. Similarly for memory-based injectors, the injection locations are

[1] http://flask.pocoo.org/
[2] https://www.sqlalchemy.org/

Table 1: Subset of chart types in FaultSight.

Injections	Features
Injected types	Detected trials
Injection type per function	Detected injection bit locations
Injected bit locations	Detection latency
Injected functions/components	Trials unexpectedly terminated
Injected line numbers	

analyzed in comparison to the memory locations used by `foo`. FaultSight supports this analysis by incorporating profiling information from traces or by using static analysis on the code or static memory allocation based on size.

Comparative analysis allows the user to filter the trial space and divide the resulting set of trials into two sets based on given criteria. This partitioning allows the user to conduct A/B style testing for resilience. To partition the sets, the user requests aggregated information about these two trial sets such as injection information, detection occurrences, and signals. For each of these fields, the user requests a statistical comparison between the two trial sets, to determine the p-value in the statistical difference or equality of these sets with full control over the confidence interval to use in these calculations. This functionality is most useful when comparing resiliency algorithms, effectivness of software-based detection and/or recovery schemes, or most other quantitative comparisons between two groups of fault injection trials. Section 4.1 uses this feature to compare the number of elements that are incorrect in the results of three different Matrix-Matrix Multiplication routines.

3.3.2 Charts. FaultSight has the ability to generate charts of quantitative data at various levels of granularity. Charts utilize data from various combinations of the data stored within the tables of the database. For example, FaultSight is able to visualize where the fault injection campaign injects faults — e.g., what functions, what components, what bit-locations, what process/thread. Additionally, FaultSight can generate charts of statistics over all the fault injection trials such as percentage of trials terminated with a segfault and percentage of trials with detection of the injected fault.

Through the use of user defined constraints on the data, applied through a series of drop down menus, FaultSight is able to generate custom fine grain views into the campaign. For example, a user could filter and visualize the flip locations on process rank 4 where injections occurred in function `foo` and generated a segfault. The use of custom constraints when visualizing data through FaultSight provides users with a powerful tool to gain detailed knowledge about each fault and how they impact detection and correctness. Section 4 explores this in more detail, as FaulSight generates all plots and figures referenced. A subset of default charts generated by FaultSight is shown in Table 1. These plots provide a strong foundation before custom constraints are applied, leading to finer-grained views into the campaign results. FaultSight also supports exportation to common image formats and the raw data to JSON files for use in other software and frameworks.

3.3.3 Code Analysis. Code analysis displays syntax highlighted regions of code where fault injections occur. If possible, the source code lines are expanded to display a line's assembly breakdown

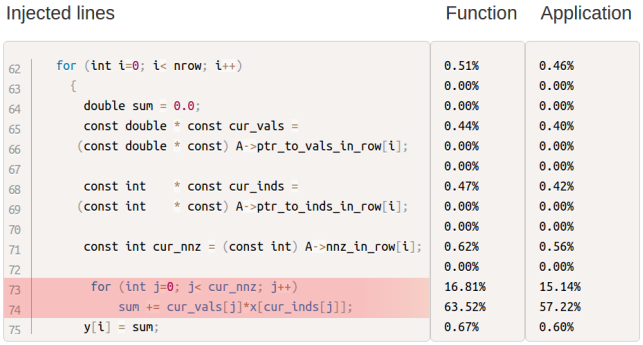


Figure 2: FaultSight code analysis for injections into function `HPC_sparsemv` from HPCCG (Section 4.2). *Note: Not all injected source lines displayed.*

information. This allows for more accurate information and inferences on injection location and fault vulnerability, such as regions where faults are likely or where faults cause the most harm. Figure 2 shows the location and percentage of injections into the function `HPC_sparsemv`, the most commonly injected function in the HPCCG experiments (Section 4.2), for the function itself and the whole application. Most injections (63.52% for the function and 57.22% for the application) into this function occur in the inner-most loop (line 74) where most of the time is spent. Prior work, has shown this routine is a critical function to protect for linear solvers [13, 14]. FaultSights code analysis assists users in understanding where injections are likely and where they are most serious for programmatic symptoms — e.g., unexpected termination, output corruption. With this knowledge, users are able to discover key data structures in need of protection and where to place detectors to minimize detection overhead and error propagation. This type of information can be provided to instruction/data replication techniques to help provide high levels of resiliency with low overheads [28].

4 EXPERIMENTAL RESULTS

In this section, FaultSight is used to analyze fault injection campaign results on a Matrix-Matrix Multiplication (MMM) kernel from the perspective of two different fault injectors and on the High Performance Computing Conjugate Gradients (HPCCG) mini-app from the Mantevo Suite[3].

In our experiments, two different fault injectors are used. The first fault injector is the LLVM based fault injector, FlipIt [11]. Trials with FlipIt were run on Phase 6 of the Palmetto Cluster at Clemson University. The second fault injector is the sequential fault injector, FSEFI [23]. Trials with FSEFI were run on the Gamma Cluster at the New Mexico Consortium located at Los Alamos National Laboratory.

When testing the MMM kernel, FlipIt injects a single bit-flip into a random bit position in the result register of a random LLVM intermediate representation (IR) instruction during the main computation phase. Conversely, FSEFI injects a single bit-flip into a random bit position during the computation of specified floating

[3]https://mantevo.org/packages.php

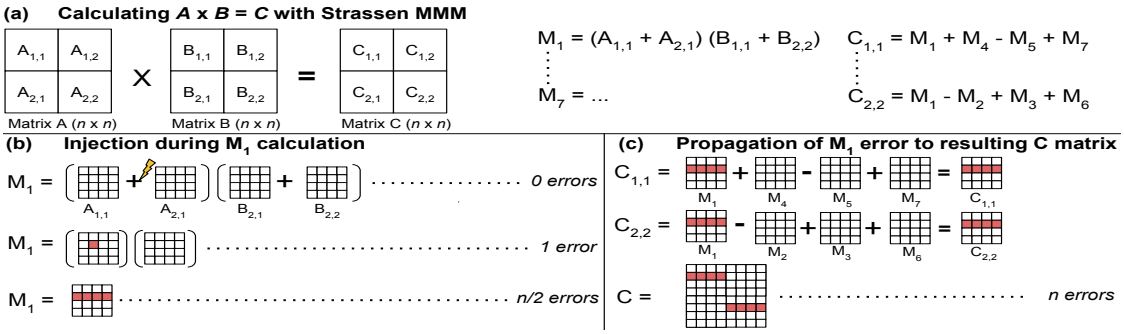


Figure 3: Propagation of error in the Strassen algorithm.

point x86 instructions — e.g. *fmul*, *fadd*, *fsub* — during the main computation phase. When testing the HPCCG mini-app, FlipIt utilizes the same configuration used when testing the MMM kernel.

Both injectors classify each injection based on how that instruction is used in code: floating-point arithmetic (*Arith-FP*), fixed-point integer arithmetic (*Arith-Fix*), pointer and address calculation (*Pointer*), branching and comparisons (*Ctrl-Branch*), and control flow/looping (*Ctrl-Loop*). The injection classification, fault injection location in source code, and result of each trial is recorded and inserted off-line into a FaultSight database using a Python script and the API (see Section 3).

FaultSight directly generates all the plots and statistical analysis in this section without the need for external processing and verifies that fault injection campaign's size allows meaningful statistical analysis (see Section 3.3.1).

4.1 Matrix-Matrix Multiplication (MMM)

In this experiment, FaultSight's ability to perform A/B testing as well as the ability to compare two different fault injection frameworks is explored.

MMM computes $C = A \times B$, where A and B are dense $n \times n$ matrices of order $n = 512$. Three different MMM algorithms are investigated to determine which is the most resilient. The first algorithm is the canonical three loop algorithm `Matmul`, the second is a cache-block tiled version of the standard algorithm `Tiled`, and the third is the Strassen algorithm `Strassen` [39]. Figure 3(a) shows how the Strassen algorithm recursively reduces each matrix into four equally sized blocks and recombines them to calculate each entry, C_{ij}. This reduces the algorithmic complexity by eliminating some operations. The reliability of each algorithm is assessed by analyzing the number of corrupted elements in the final output. Machine epsilon is the error threshold to determine if any two floating-point values differ. Through the use of the `extend` API call, an attribute that represents the number of corrupted elements in the final ouput of each trial was added.

The FlipIt fault injection campaign consists of running 6,000 injection trials. The trials were split into three groups and had the following breakdown: 2,000 `Matmul`, 2,000 `Tiled`, and 2,000 `Strassen`.

When FlipIt considers a function for injection, it considers the base function and all functions it invokes. Conversely, FSEFI requires a specific instruction to be given that it will target. With this in mind, a set of floating point x86 instructions were given to FSEFI to better mirror the results of FlipIt and therefore lead to a better comparison of the two tools.

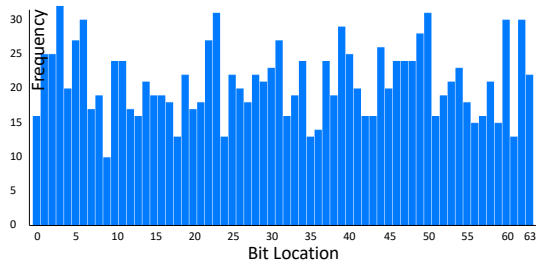
For FSEFI, the fault injection campaign consists of running 8,000 injection trials. The trials are split into five groups with the following breakdown: 2,000 `Matmul` while targeting the main *fmul*, 2,000 `Tiled` while targeting the main *fmul*, 2,000 `Strassen` while targeting the main *fmul*, and 2,000 `Strassen` with 1,000 targeting the main *fadd* and 1,000 targeting the main *fsub*.

Throughout the FlipIt campaign, many different functions were hit with faults. During the `Matmul` trials all 2,000 injections hit within the *matmul* function. During the `Tiled` trials 1,999 injections hit within the *matmul_tiled* function while 1 hit within the *min* function. During the `Strassen` trials 182 injections hit within the *matmul_strassen* function, 1,764 injections hit within the *matmul* function, and 54 injections hit within the *matadd* function. The *matmul_strassen* function utilizes both *matmul* and *matadd* to combine blocks of matrices. These results help derive the set of x86 instructions that FSEFI implemented in its experiments.

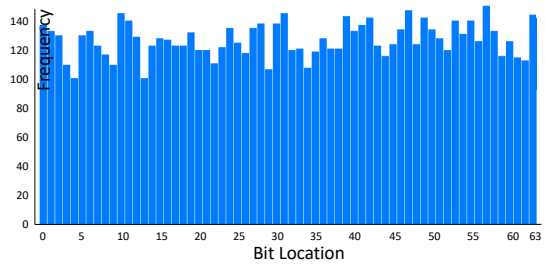
The number of corrupted elements for all FlipIt and FSEFI trials are recorded. Fault injections in the `Matmul` trials result in a total of 1,259 corrupted items in FlipIt and 1,700 corrupted items in FSEFI. On average, a single injection in FlipIt corrupted 0.63 elements and 0.85 elements in FSEFI. Injections in the `Tiled` trials result in a total of 1,216 corrupted elements in FlipIt and 1,707 corrupted items in FSEFI. On average, a single injection in FlipIt corrupted 0.61 elements and 0.853 elements in FSEFI. Lastly, Fault injections in the `Strassen` trials result in 1,156,562 corrupted items in FlipIt and 364,029 corrupted items in FSEFI. On average, a single injection in FlipIt corrupted 578.3 elements and 182 elements in FSEFI.

The difference between FlipIt and FSEFI's results from the `Matmul` and `Tiled` trials are due to variations in how significant the fault was on the injected value. For instance, if the fault occurred in the lower bits then the fault could fall off during floating point computation or due to rounding of by the system.

The high number of elements corrupted in the `Strassen` results, seen from both fault injectors, come from corruption of the sub-matrices that are used to compute C. Corrupting these sub-matrices propagates the errors to even more elements of C. Figure 3(b)–(c) illustrates this propagation pattern when a fault is injected into a *matadd* operation during the calculation of M_1. The fault results



(a) FlipIt Distribution



(b) FSEFI Distribution

Figure 4: Bit Flip Distribution of Arith-FP instruction types for each injector

in a single element (i, k) of the *matadd* being in error. The multiplication operation propagates this error at (i, k) to all entries in row i due to data reuse in the *matmul* function leading to $n/2$ errors. The algorithm uses M_1 in the calculation of $C_{1,1}$ and $C_{2,2}$ which propagates the results to the final output. Depending on how much progress the algorithm has made before the fault occurs, this can result in corruption of up to n elements in the matrix C. Fault injection results show a corruption of 578.3 elements from FlipIt and 182 elements from FSEFI on average. From these results, it can be observed that FSEFI chose a uniform distribution of locations to hit with some bias towards later instructions in the execution. If hit locations were completely unbiased then the average number of corrupted elements when only hitting the Arith-FP instructions, as in the case of FSEFI, would be $n/2$ or 256 in our case. Therefore, FSEFI's 182 denotes a uniform distribution with some bias towards later instructions. FlipIt's results went above n due to different propagation signatures that lead to increase corruption in C — e.g. terminating a loop early and not computing a portion of the matrix.

Fault injectors need to be unbiased in their choice of which bits should be flipped from the range they are given. Through FaultSight's visualizations, we can visualize the distribution of injection locations from each fault injector and compare them. As seen in Figure 4, we can are able to compare the distributions for both fault injectors Arith-FP operations and tell that both have uniform distributions when choosing which bit to flip.

4.1.1 Algorithm A/B Testing. Using the custom partitioning feature of FaultSight, we instruct FaultSight to partition the fault injection trials into the 3 groups for the two different fault injectors, one for each MMM. Performing a Student's T-Test to compare frequency of corrupted elements in the matrix C pairwise between two different MMM functions, it is found that the difference in error rates between the `Matmul` and `Tiled` approaches is not significant, with a p-value of 0.16 with FlipIt and a p-value of 0.76 with FSEFI. However, the difference between the `Matmul` or `Tiled` to the `Strassen` approach is significant, where both comparisons result in a p-value of 0.0097 with FlipIt and a p-value of 1.78×10^{-211}.

Based upon this analysis, if you are running in an environment where the probability of suffering a bit-flip error during MMM operation is likely and employ no additional resiliency techniques, it is beneficial to use a slower performing algorithm — i.e., `Matmul` or `Tiled` — as opposed to `Strassen`. This is due to the fact that the high volume of data reuse in the `Strassen` algorithm leads to a high degree of corruption in the resulting matrix C.

4.1.2 Protecting Matrix-Matrix Multiplication. Prior work [25, 41] explores protecting linear algebra operations with row and column checksums in a technique known as algorithm-based fault tolerance (ABFT). Using checksums, ABFT is able to pinpoint the matrix entry that is incorrect and correct it. Based on FlipIt's MMM injection campaign, on average 0.63 and 0.61 elements are corrupted for `Matmul` and `Tiled`, respectively; therefore, ABFT is an effective scheme that detects and corrects all single entry errors. Even though, in FlipIt's trials, `Strassen` resulted in 578.3 elements on average being corrupted, ABFT is an effective scheme for protecting this algorithm. Since `Strassen` subdivides A and B, appending row and column checksums to these sub-matrices allows for detection of corrupt entries in the intermediate *matadd* and *matmul* before the error propagates via *matmul* operations. Applying checksums to each of the sub-matrices results in a larger memory overhead due to requiring more checksums by a factor of 2 for each level of the recursion. Although ABFT is on average an attractive solution to protect MMM, the faults that result in multiple erroneous entries such as early loop termination may be unrecoverable with ABFT. However, augmenting ABFT protected MMM with code that ensures correct control flow [28, 33] alleviates this shortcoming.

4.2 HPCCG

In this example, FaultSight is used to show that a software-based soft error detection/recovery scheme successfully limits the number of extra iterations until the problem converges compared to runs of an unmodified HPCCG. Here, FaultSight conducts statistical tests of significance in several contexts — comparing the efficacy of a detection/recovery scheme and comparing the impact of injections into different locations/bits/datatypes.

HPCCG[4] is a conjugate gradient (CG) benchmark from the Mantevo Suite that simulates a 3D chimney domain using a 27-point finite difference matrix. In these experiments, HPCCG is run with 16 parallel MPI processes, a local block size of $nz = ny = nz = 48$, and a solver tolerance of $1e{-}10$. For each trial, a single bit-flip is injected using FlipIt during the solve phase of HPCCG on a randomly selected MPI process and instruction. Because a bit-flip can

[4] https://mantevo.org/packages.php

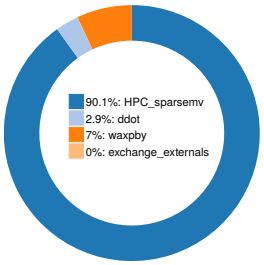


Figure 5: Fault injections mapped to functions for HPCCG.

lead to extra iterations, the number of iterations is capped at 500. Any trial reaching 500 iterations is marked as *not converged*.

The campaign uses 3,000 trials, where the first 1,500 trials have detection but no recovery scheme, and the remaining 1,500 trials have both a detection and recovery scheme implemented. The detection scheme verifies the plausibility of the residual at the end of each iteration by ensuring that the residual does not increase by an order-of-magnitude [12]. At the end of every iteration, the solution vector x, the residual vector r, and the search direction vector p are checkpointed to local memory. When the residual check is violated, the in-memory checkpoint is read and the variables reinitialized to the checkpointed values [40].

4.2.1 Impact on Convergence. FaultSight maps the injections back to the current executing function, Figure 5, and reveals that greater than 90% of injections occur in the function `HPC_sparsemv` that computes sparse matrix-vector multiplication (SpMV), the most time consuming portion of the code. This is consistent with prior studies on other iterative linear solvers [12, 14]. Other functions with injections include the `ddot` (inner-product), `waxpby` (scaled vector addition), and `exchange_internals` (communication of vector values for parallel sparse matrix-vector multiplication)[5]. Exploring the code view for each injected function highlights the vulnerable regions. Typically this is the most frequently executed code segments such as the innermost loop (see Figure 2).

During the database generation phase of the workflow, we use FaultSight's API (`extend_trial_table`) to extend the existing database by adding an attribute to represent the number of iterations for the problem to converge. Combining this with FaultSight's ability to test hypotheses, we determine if the detection/recovery scheme is effective at limiting the number of iterations to converge in a faulty environment.

In the fault-free case, HPCCG requires 90 iterations to converge to our set tolerance. In our fault injection trials, 1.8% of trials require extra iterations. Of those trials requiring extra iterations, 5 trials did not converge. Figure 6 refines Figure 5 to explore only those trials that cause extra iterations. Because the SpMV has a higher degree of data reuse than vector only operations [13], error propagation patterns in the SpMV are similar to *matmul* in the Strassen algorithm.

Figure 7 shows the bit location of the injections broken down by the type of instruction for all trials that require extra iterations to converge. In Figure 7, there are 3 clusters based on bit location and instruction type. The first group (bits 0–2) consists of injections into *Pointer* type instructions, resulting in reads and writes of unaligned data. Reading/writing unaligned data produces garbage values that have a high error between the correct value. The second group consists of *Ctrl-Loop* and *Arith-Fix* operations. Corrupting these operations leads to early termination of loops which result in arrays not being fully updated or correctly initialized. This leads to computing on unknown values, resulting in large deviations from the correct values. The third group consists of injections into *Arith-FP* instructions. These injections take place in the upper bits of the mantissa (bits 48–51), all the exponent bits (bits 52–62), and the sign bit (bit 63). Bit-flips in the most significant bits of the mantissa cause noticeable deviations in the magnitude of floating-point values. Bit-flips in the sign bit results in the negation of a floating-point value, and the deviation from the true value is proportional to its magnitude. However, bit-flips in the exponent cause much larger deviations; some orders of magnitude more than bit-flips in the mantissa or sign-bit. Large deviations result in added error that propagates to the vectors x, p, and r. Corruption in these vectors cause HPCCG to require more iterations to achieve the fixed solver tolerance. In addition, injections into *Arith-FP* instructions is statistically significant to lead to extra iterations (beyond fault-free) with a p-value of $6.8e{-}278$. Other instruction types show non-significant results due to the high rates of unexpected application termination.

Figures 8 and 9 highlight detection. The detection scheme for HPCCG verifies that the new residual is within an order of magnitude of the previous one. Therefore, this detector only triggers when large deviations occur — i.e., bit-flips in the exponent and reading of garbage values. If the detection scheme places tighter guarantees on how much the residual increases, Figure 9 approaches Figure 7 as all injections that lead to extra iterations are detected. Similarly, Figure 8 closely matches Figure 6 signifying that the detection scheme is preventing trials that require extra iterations. Because a tight bound is not used, only trials that require 3+ iterations to converge are detected. Provided that other information such as program wall-clock time or energy is logged, FaultSight can also be used to help analyze and tune the tolerance bound on the residual detector to minimize wall-clock time and/or energy consumption.

Out of the 90 trials that took more iterations to converge than the fault-free case (and successfully converged), 62 trials (70%) have a detection. The 62 trials are split into two trial sets based on whether the trial implements a recovery scheme or not. The average number of iterations to converge of the set of trials with a recovery scheme is 96.44, while the average number of iterations to converge for trials with no recovery scheme is 108.6. Using the Student's T-test for statistical independence, shows that the difference is significant with a p-value of $8.9e{-}8$. Thus, the detection and recovery scheme is effective at reducing the extra iterations needed to converge when transient soft errors corrupt portions of the CG linear solver.

4.2.2 Unexpected Termination. Over the course of the fault injection campaign, 33.8% of fault injection trials experience unexpected application termination. In these experiments, two symptoms cause the unexpected termination events: generation of a segmentation fault and generation of a bus error.

Figure 10 reports the bit locations that lead to a segmentation fault grouped by the injected instruction type. In Figure 10, there

[5]Note there are 2 injections into `exchange_internals`.

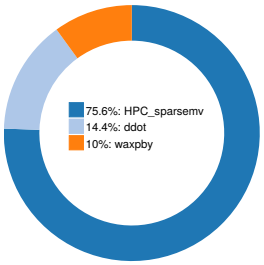


Figure 6: Injections mapped to functions for HPCCG, where the number of iteration are greater than the fault-free case.

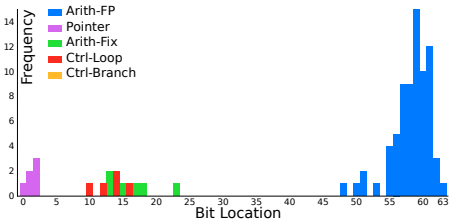


Figure 7: Classification of injections by bit, from trials where the number of iterations to converge is greater than the fault-free case.

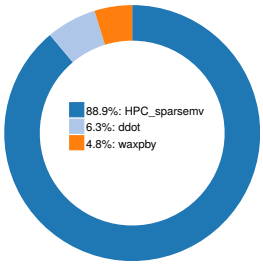


Figure 8: Injections mapped to functions, for trials with detection.

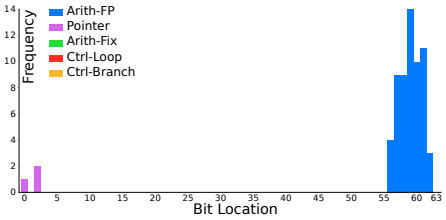


Figure 9: Classification of injections by bit, for trials with detection.

are no segmentation faults due to injections into *Arith-FP* instructions. Segmentation faults from injections into *Pointer* instructions occur in bits that lead to unaligned data access (0–1) yielding incorrect loads/stores or occur in bits that creates a pointer outside the memory allocation (21–63). Fixed-point arithmetic operations, *Arith-Fix*, are performed on 32-bit integers and do not have injections in bits 32–63. Injections into *Arith-Fix* instructions that cause segmentation faults produce results that lead into incorrect address calculation. During compilation, `clang` promotes the 32-bit integers used in loop iteration variables to 64-bit, allowing segfaults generated to be concentrated in bits outside of the size of allocated memory for calculation on loop iteration variables, *Ctrl-Loop*. Finally, injections into branches, *Crtl-Branch* results in an extra iteration, and during the extra iteration the program accesses out-of-bound memory — e.g., accessing off the end of an array.

The second cause of unexpected termination is due to bus errors. An x86 processor raises a bus error when the program accesses an undefined portion of a memory object. Figure 11 shows the bit position and instruction types that lead to a bus error in our fault injection campaign. These are confined to *Pointer* instructions and in particular, the most significant bits of the pointer. Bit-flips in these positions cause a large deviation in the pointer that leads to accessing undefined portions of memory.

Statistical analysis of the bit locations shows that injections in bits 32–63 are statistically more likely to lead to unexpected termination than bits 0–31 with a p-value of $1.08e{-}39$. From this analysis, FaultSight shows that injections into the most significant bits are the most impactful. Adding protection to these bits either in hardware or software can lower an application's likelihood of unexpectedly terminating when soft error corrupts logic and instruction results. For example, [14] uses triple modular redundancy to protect key pointers and avoid segfaults. Additionally instruction replication techniques [28] could add protection for pointers and pointer operations. Using code analysis to pinpoint the code regions that suffer unexpected termination allows for reduced overheads in these resiliency schemes.

5 RELATED WORK

Several types of fault injectors have been used to evaluate the reliability of HPC applications. FSEFI [23] injects faults into an application running on a virtual machine, and has a very high level of control on where it inject faults. At a slightly higher level, FlipIt [11], LLFI [31], KULFI [36], and SAFIRE-[21] uses an LLVM compiler pass to instrument code for fault injection. However, other studies corrupt MPI communication [20, 30]. Yet others, corrupt application level variables [4, 6]. Each fault injector has a unique way of recording data about the location and time of the injection. The tool FaultTelescope built for KULFI identifies vulnerable regions of code [15], but does not support data from other fault injectors nor statistical testing. FaultSight's API allows for data from many different fault injectors to be recorded into the database for analysis.

Tools to analyze system reliability have been developed to analyze reliability, availability, and serviceability of (RAS) logfiles as well as other logfiles generated by HPC systems [17–19, 22, 24, 32, 34]. These tools address similar problems working with large data volumes, but are not designed to analyze and visualize data coming from fault injection campaigns and comparing the effectiveness of integrated application level detection and recovery schemes. FaultSight's web app interface provides more efficient analysis of fault injection data.

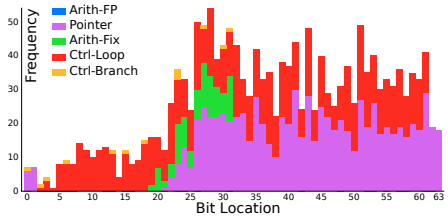


Figure 10: Classification of injections by bit, for trials with a Segmentation Fault.

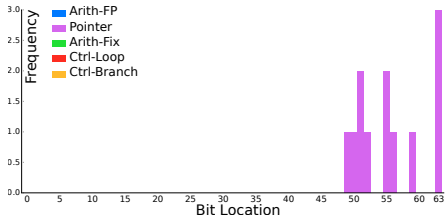


Figure 11: Classification of injections by bit, for trials with a Bus Error.

Debugging tools — e.g., Allinea DDT[6], TotalView[7], GDB[8] — allow deep introspection into a running application in order to identify software bugs and promote correctness. While they can be used for fault injection, their view is limited to a single application's execution and do not easily allow analysis of thousands of application runs at once.

In order to improve performance or efficiency of HPC applications software profilers are commonly used to visualize how time and resources are spent during an application run [1, 10, 37]. Software profilers allow a detailed view of events such as function invocations, message ordering, load imbalance, and their associated runtimes. However, current HPC profilers do not visualize the extra information coming from/about a simulation due to fault injection.

6 CONCLUSION AND FUTURE WORK

To obtain meaningful statistics when assessing an application's resiliency to faults or to test the effectiveness of a software-based fault detection/recovery scheme requires thousands of fault injection trails. Analyzing the large volume of data is laborious as there is no common tool for various fault injectors used in practice. This paper presents FaultSight: a general fault injection analysis tool capable of visualizing and analyzing the results of thousands of fault injection trials from various fault injectors thought its extendable interface.

We use FaultSight to analyze the reliability of three Matrix-Matrix Multiplication algorithms from multiple perspectives and find there is not much difference between the canonical three loop version and the tiled version. However, we find that Strassen algorithm generates hundreds more corrupt entries in both perspectives than the other algorithms due to its higher reuse of sub-computations. We also analyze the effectiveness of a detection and recovery scheme for the mini-app HPCCG. Through FaultSight's statistical analysis feature, we show that the resiliency scheme is effective at limiting the number of extra iterations when transient faults occur.

Future work on FaultSight seeks to improve its interoperability with other common data analysis frameworks such as Spark and provide support for machine learning to detect trends in the fault injection data.

Finally, prior research is replicated and supported through analyzing fault injection campaigns on MMM and HPCCG.

FaultSight is publicly available under the MIT license on github: https://github.com/einarhorn/FaultSight.

[6]https://www.arm.com/products/development-tools/hpc-tools/cross-platform/forge/ddt
[7]https://www.roguewave.com/products-services/totalview
[8]https://www.gnu.org/software/gdb/

ACKNOWLEDGMENT

This work was sponsored by the Air Force Office of Scientific Research under grant FA9550–12–1–0478. This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (NSF) (awards OCI–0725070 and ACI–1238993) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications. This material is also based in part on work supported by the Department of Energy, National Nuclear Security Administration, under Award Number DE-NA0002374. This material is based upon work supported by the NSF under Grant No. SHF-1617488.

We would also like to thank Terry Grové and Nathan Debardeleben at Los Alamos National Laboratory for their assistance and allowing us to utilize the Gamma Cluster.

REFERENCES

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. 2010. HPCTOOLKIT: Tools for Performance Analysis of Optimized Parallel Programs Http://Hpctoolkit.Org. *Concurr. Comput. : Pract. Exper.* 22, 6 (April 2010), 685–701. https://doi.org/10.1002/cpe.v22:6

[2] R. C. Baumann. 2005. Radiation-induced soft errors in advanced semiconductor technologies. *Device and Materials Reliability, IEEE Transactions on* 5, 3 (Sept. 2005), 305–316. https://doi.org/10.1109/TDMR.2005.853449

[3] Leonardo Bautista-Gomez and Franck Cappello. 2015. Detecting Silent Data Corruption for Extreme-Scale MPI Applications. In *Proceedings of the 22Nd European MPI Users' Group Meeting (EuroMPI '15)*. ACM, New York, NY, USA, Article 12, 10 pages. https://doi.org/10.1145/2802658.2802665

[4] Leonardo Bautista-Gomez and Franck Cappello. 2015. Exploiting Spatial Smoothness in HPC Applications to Detect Silent Data Corruption. In *Proceedings of the 2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conf on Embedded Software and Systems (HPCC-CSS-ICESS '15)*. IEEE Computer Society, Washington, DC, USA, 128–133. https://doi.org/10.1109/HPCC-CSS-ICESS.2015.9

[5] Leonardo Bautista-Gomez, Seiji Tsuboi, Dimitri Komatitsch, Franck Cappello, Naoya Maruyama, and Satoshi Matsuoka. 2011. FTI: high performance fault tolerance interface for hybrid systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*. ACM, New York, NY, USA, Article 32, 32 pages. https://doi.org/10.1145/2063384.2063427

[6] Austin R Benson, Sven Schmit, and Robert Schreiber. 2015. Silent error detection in numerical time-stepping schemes. *The International Journal of High Performance Computing Applications* 29, 4 (2015), 403–421. https://doi.org/10.1177/1094342014532297 arXiv:http://dx.doi.org/10.1177/1094342014532297

[7] Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, et al. 2008. *Exascale computing study: Technology challenges in achieving exascale systems*. Technical Report. Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO).

[8] E. Berrocal, L. Bautista-Gomez, S. Di, Z. Lan, and F. Cappello. 2017. Toward General Software Level Silent Data Corruption Detection for Parallel Applications. *IEEE Transactions on Parallel and Distributed Systems* 28, 12 (Dec 2017), 3642–3655. https://doi.org/10.1109/TPDS.2017.2735971

[9] Shekhar Borkar. 2005. Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE Micro* 25, 6 (Nov. 2005), 10–16. https://doi.org/10.1109/MM.2005.110

[10] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. 2000. A Portable Programming Interface for Performance Evaluation on Modern Processors. *Int. J. High Perform. Comput. Appl.* 14, 3 (Aug. 2000), 189–204. https://doi.org/10.1177/109434200001400303

[11] Jon Calhoun, Luke Olson, and Marc Snir. 2014. FlipIt: An LLVM Based Fault Injector for HPC. In *Proceedings of the 20th International Euro-Par Conference on Parallel Processing (Euro-Par '14)*.

[12] Jon Calhoun, Luke Olson, Marc Snir, and William D. Gropp. 2015. Towards a More Fault Resilient Multigrid Solver. In *Proceedings of the Symposium on High Performance Computing (HPC '15)*. Society for Computer Simulation International, San Diego, CA, USA, 1–8. http://dl.acm.org/citation.cfm?id=2872599.2872600

[13] Jon Calhoun, Luke N. Olson, Marc Snir, and William D. Gropp. 2017. Towards a More Complete Understanding of SDC Propagation. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '17)*. ACM, New York, NY, USA.

[14] Marc Casas, Bronis R. de Supinski, Greg Bronevetsky, and Martin Schulz. 2012. Fault resilience of the algebraic multi-grid solver. In *Proceedings of the 26th ACM international conference on Supercomputing (ICS '12)*. ACM, New York, NY, USA, 91–100. https://doi.org/10.1145/2304576.2304590

[15] Sui Chen, Greg Bronevetsky, Bin Li, Marc Casas Guix, and Lu Peng. 2015. A framework for evaluating comprehensive fault resilience mechanisms in numerical programs. *The Journal of Supercomputing* 71, 8 (01 Aug 2015), 2963–2984. https://doi.org/10.1007/s11227-015-1422-z

[16] Zizhong Chen. 2013. Online-ABFT: An Online Algorithm Based Fault Tolerance Scheme for Soft Error Detection in Iterative Methods. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*. ACM, New York, NY, USA, 167–176. https://doi.org/10.1145/2442516.2442533

[17] S. Di, R. Gupta, M. Snir, E. Pershey, and F. Cappello. 2017. LOGAIDER: A Tool for Mining Potential Correlations of HPC Log Events. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 442–451. https://doi.org/10.1109/CCGRID.2017.18

[18] E.Chuah et al. 2019. Towards comprehensive dependability-driven resource use and message log-analysis for HPC systems diagnosis. *J. Parallel and Distrib. Comput.* (May 2019), 95–112. https://doi.org/10.1016/j.jpdc.2019.05.013

[19] M. Hickman et al. 2018. Enhancing HPC System Log Analysis by Identifying Message Origin in Source Code. *2018 IEEE International Symposium on Software Reliability Engineering Workshops* (Aug. 2018), 100–105. https://doi.org/10.1109/ISSREW.2018.00-23

[20] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. 2012. Detection and Correction of Silent Data Corruption for Large-scale High-performance Computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*. IEEE Computer Society Press, Los Alamitos, CA, USA, Article 78, 12 pages. http://dl.acm.org/citation.cfm?id=2388996.2389102

[21] G Georgakoudis, I Laguna, H Vandierendonck, D S Nikolopoulos, and M Schulz. 2019. SAFIRE: Scalable and Accurate Fault Injection ForParallel Multithreaded Applications. *IEEE International Parallel Distributed ProcessingSymposium* (Jan. 2019). https://www.osti.gov/servlets/purl/1518562

[22] Alfredo Giménez, Todd Gamblin, Abhinav Bhatele, Chad Wood, Kathleen Shoga, Aniruddha Marathe, Peer-Timo Bremer, Bernd Hamann, and Martin Schulz. 2017. ScrubJay: Deriving Knowledge from the Disarray of HPC Performance Data. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*. ACM, New York, NY, USA, Article 35, 12 pages. https://doi.org/10.1145/3126908.3126935

[23] Qiang Guan, Nathan BeBardeleben, Panruo Wu, Stephan Eidenbenz, Sean Blanchard, Laura Monroe, Elisabeth Baseman, and Li Tan. 2016. Design, Use and Evaluation of P-FSEFI: A Parallel Soft Error Fault Injection Framework for Emulating Soft Errors in Parallel Applications. In *Proceedings of the 9th EAI International Conference on Simulation Tools and Techniques (SIMUTOOLS'16)*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium, 9–17. http://dl.acm.org/citation.cfm?id=3021426.3021429

[24] Hanqi Guo, Sheng Di, Rinku Gupta, Tom Peterka, and Franck Cappello. 2018. La VALSE: Scalable Log Visualization for Fault Characterization in Supercomputers. In *Eurographics Symposium on Parallel Graphics and Visualization*, Hank Childs and Fernando Cucchietti (Eds.). The Eurographics Association. https://doi.org/10.2312/pgv.20181099

[25] Kuang-Hua Huang and J. A. Abraham. 1984. Algorithm-Based Fault Tolerance for Matrix Operations. *IEEE Trans. Comput.* 33, 6 (June 1984), 518–528. https://doi.org/10.1109/TC.1984.1676475

[26] Luc Jaulmes, Marc Casas, Miquel Moretó, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. 2015. Exploiting Asynchrony from Exact Forward Recovery for DUE in Iterative Solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. ACM, New York, NY, USA, Article 53, 12 pages. https://doi.org/10.1145/2807591.2807599

[27] Himanshu Kaul, Mark Anders, Steven Hsu, Amit Agarwal, Ram Krishnamurthy, and Shekhar Borkar. 2012. Near-threshold Voltage (NTV) Design: Opportunities and Challenges. In *Proceedings of the 49th Annual Design Automation Conference (DAC '12)*. ACM, New York, NY, USA, 1153–1158. https://doi.org/10.1145/2228360.2228572

[28] Ignacio Laguna, Martin Schulz, David F. Richards, Jon Calhoun, and Luke Olson. 2016. IPAS: Intelligent Protection Against Silent Output Corruption in Scientific Applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO 2016)*. ACM, New York, NY, USA, 227–238. https://doi.org/10.1145/2854038.2854059

[29] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert. 2009. Statistical Fault Injection: Quantified Error and Confidence. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '09)*. European Design and Automation Association, 3001 Leuven, Belgium, Belgium, 502–506. http://dl.acm.org/citation.cfm?id=1874620.1874743

[30] Charng-da Lu and Daniel A. Reed. 2004. Assessing Fault Sensitivity in MPI Applications. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC '04)*. IEEE Computer Society, Washington, DC, USA, 37–. https://doi.org/10.1109/SC.2004.12

[31] Qining Lu, Mostafa Farahani, Jiesheng Wei, Anna Thomas, and Karthik Pattabiraman. 2015. LLFI: An Intermediate Code-Level Fault Injection Tool for Hardware Faults. In *Proceedings of the 2015 IEEE International Conference on Software Quality, Reliability and Security (QRS '15)*. IEEE Computer Society, Washington, DC, USA, 11–16. https://doi.org/10.1109/QRS.2015.13

[32] Catello Di Martino, Saurabh Jha, William Kramer, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. 2015. LogDiver: A Tool for Measuring Resilience of Extreme-Scale Systems and Applications. In *Proceedings of the 5th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS '15)*. ACM, New York, NY, USA, 11–18. https://doi.org/10.1145/2751504.2751511

[33] Xiang Ni and Laxmikant V. Kale. 2016. FlipBack: Automatic Targeted Protection Against Silent Data Corruption. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*. IEEE Press, Piscataway, NJ, USA, Article 29, 12 pages. http://dl.acm.org/citation.cfm?id=3014904.3014943

[34] Byung Hoon (Hoony) Park, Yawei Hui, Swen Boehm, Rizwan Ashraf, Christian Engelmann, and Christopher Layton. 2018. A Big Data Analytics Framework for HPC Log Data: Three Case Studies Using the Titan Supercomputer Log. In *Proceedings of the 19^{th} IEEE International Conference on Cluster Computing (Cluster) 2018: 5^{th} Workshop on Monitoring and Analysis for High Performance Systems Plus Applications (HPCMASPA) 2018*. IEEE Computer Society, Los Alamitos, CA, USA, Belfast, UK, 571–579. https://doi.org/10.1109/CLUSTER.2018.00073

[35] John Shalf, Sudip Dosanjh, and John Morrison. 2011. Exascale Computing Technology Challenges. In *Proceedings of the 9th International Conference on High Performance Computing for Computational Science (VECPAR'10)*. Springer-Verlag, Berlin, Heidelberg, 1–25. http://dl.acm.org/citation.cfm?id=1964238.1964240

[36] Vishal Chandra Sharma, Arvind Haran, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. 2013. Towards Formal Approaches to System Resilience. In *Proceedings of the 19th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*.

[37] Sameer S. Shende and Allen D. Malony. 2006. The Tau Parallel Performance System. *Int. J. High Perform. Comput. Appl.* 20, 2 (May 2006), 287–311. https://doi.org/10.1177/1094342006064482

[38] Marc Snir, Robert W Wisniewski, Jacob A Abraham, Sarita V Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, Andrew A Chien, Paul Coteus, Nathan A DeBardeleben, Pedro C Diniz, Christian Engelmann, Mattan Erez, Saverio Fazzari, Al Geist, Rinku Gupta, Fred Johnson, Sriram Krishnamoorthy, Sven Leyffer, Dean Liberty, Subhasish Mitra, Todd Munson, Rob Schreiber, Jon Stearley, and Eric Van Hensbergen. 2014. Addressing Failures in Exascale Computing. *International Journal of High Performance Computing Applications* 28, 2 (May 2014), 127 – 171. https://doi.org/10.1177/1094342014522573

[39] Volker Strassen. 1969. Gaussian Elimination is Not Optimal. *Numer. Math.* 13, 4 (Aug. 1969), 354–356. https://doi.org/10.1007/BF02165411

[40] Dingwen Tao, Sheng Di, Xin Liang, Zizhong Chen, and Franck Cappello. 2018. Improving performance of iterative methods by lossy checkponting. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 52–65.

[41] Dingwen Tao, Shuaiwen Leon Song, Sriram Krishnamoorthy, Panruo Wu, Xin Liang, Eddy Z Zhang, Darren Kerbyson, and Zizhong Chen. 2016. New-sum: A novel online abft scheme for general iterative methods. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 43–55.