SpotSDC: Revealing the Silent Data Corruption Propagation in High-Performance Computing Systems

Zhimin Li, Harshitha Menon, Dan Maljovec, Yarden Livnat, *Member, IEEE,* Shusen Liu, Kathryn Mohror, *Member, IEEE,* Peer-Timo Bremer, *Member, IEEE,* and Valerio Pascucci, *Member, IEEE*

Abstract—The trend of rapid technology scaling is expected to make the hardware of high-performance computing (HPC) systems more susceptible to computational errors due to random bit flips. Some bit flips may cause a program to crash or have a minimal effect on the output, but others may lead to silent data corruption (SDC), i.e., undetected yet significant output errors. Classical fault injection analysis methods employ uniform sampling of random bit flips during program execution to derive a statistical resiliency profile. However, summarizing such fault injection result with sufficient detail is difficult, and understanding the behavior of the fault-corrupted program is still a challenge. In this work, we introduce SpotSDC, a visualization system to facilitate the analysis of a program's resilience to SDC. SpotSDC provides multiple perspectives at various levels of detail of the impact on the output relative to where in the source code the flipped bit occurs, which bit is flipped, and when during the execution it happens. SpotSDC also enables users to study the code protection and provide new insights to understand the behavior of a fault-injected program. Based on lessons learned, we demonstrate how what we found can improve the fault injection campaign method.

Index Terms—Fault Injection Sampling, Error Propagation, Information Visualization, Silent Data Corruption

1 Introduction

The growing demands of computation make hardware features (e.g., computer chip size, transistor size, etc.) smaller and the density of hardware components in HPC systems larger [1], which increases the likelihood of transient faults and makes the computation hardware unreliable [2], [3], [4]. Transient faults, caused by device noise, low voltage, cosmic radiation, and other factors, which in turn lead to random bit flips in hardware devices (e.g., memory, register, etc.), can manifest in applications in three ways. 1) They can cause an application to crash, e.g., the transient fault corrupts a pointer variable and causes a segmentation fault; 2) they can be benign and not affect the application's output; or 3) they can alter the computation result without being noticed and cause silent data corruption (SDC). SDC faults are dangerous in that they are hard to detect, and the corrupted output result can have serious consequences in areas such as nuclear reactor design, where corrupted output could lead to regretful circumstances.

Due to the worrisome nature of SDC, studying how it affects an application is critical for developing and evaluating software resiliency techniques against SDC. The classical approach for studying how transient faults affect an application is a fault injection campaign, which is commonly composed of thousands of experiments. Each experiment randomly flips a single bit from a variable during program execution, observes the program's behavior, and records the result. However, a fault injection campaign usually ends with a high-level statistical profile of a program's resiliency (e.g., the percentage of SDC outcome over all fault injection experiments). Without a fine-grained analysis of the impact of the transient fault over different regions and time, the conclusion of the analysis can be incomplete and/or inaccurate.

In the HPC community, researchers often design protection mechanisms, such as process replication [5], component redundancy [6], MPI redundancy [7], or auto instruction duplication [8], [9] in the compiler level, to protect the program from transient faults and reduce the probability of an SDC. However, such protections are expensive and can often cause significant overhead in HPC systems. To mitigate the cost, it is important to identify and understand program locations that are most sensitive to transient errors and the frequency in which they are being reached in order to evaluate the costs and benefits of protecting each location.

Many researchers have investigated the natural resiliency of algorithms [9], [10], [11], [12] (e.g., neural network models, k-means, linear solvers, etc.), which can mask the corruption error during the computation, to design a more efficient protection technique. For example, if a fault corrupts a variable of an iterative program and propagates through the subsequent execution, the program may need to compute more iterations to reach convergence, but in general the program still produces a correct result. However, the behavior of the majority of application domains under SDC is still unknown or known to have a high impact, e.g., SDC can cause large differences in output for high-dimensional partial differential equations (PDEs) [13] and nonsensical output for GPU graphics rendering [14]. Studying the corruption propagation to understand how the error

- *Zhimin Li, Dan Maljovec, Yarden Livnat, and Valerio Pascucci are with the Scientific Computing and Imaging Institute, University of Utah. E-mail: {zhimin, maljovec, yarden, pascucci}@sci.utah.edu*
- *Harshitha Menon, Shusen Liu, and Kathryn Mohor, and Peer-Timo Bremer are with Lawrence Livermore National Laboratory. E-mail: {harshitha, liu42, mohror1, bremer5}@llnl.gov*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2020.2994954, IEEE Transactions on Visualization and Computer Graphics

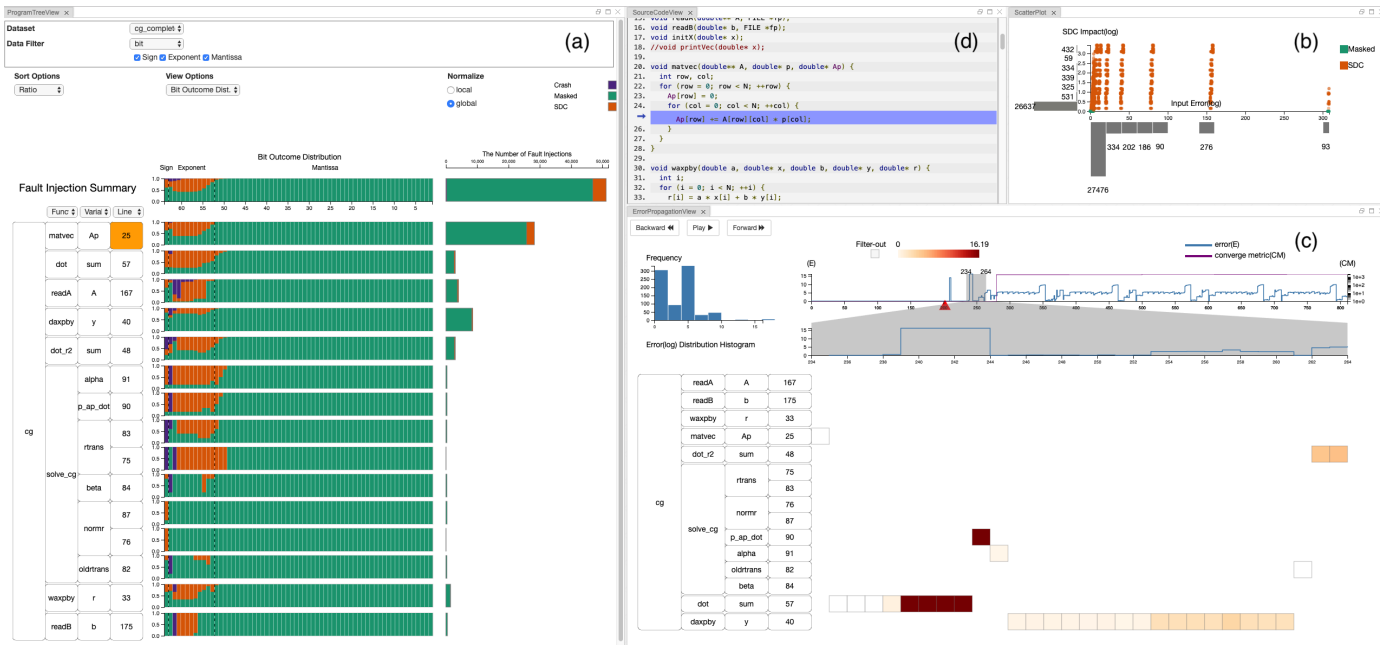


Fig. 1. SpotSDC helps researchers study silent data corruption, such as erroneous bit flips, and how these errors propagate through a program's computation. As illustrated in (a), we show a hierarchical overview of a bit flip's impact on different program regions of the conjugate gradient benchmark and highlight the most vulnerable region. Users can selectively choose a set of fault injection experiments, compare these experiments' fault injection input error vs output error in (b) and use (c) to monitor the propagation of errors during execution of a program. Both views (a) and (c) are linked to (d) for users to jointly reason about the observed error pattern, identified code vulnerability, and study protections.

propagates from one location to another during the program execution will provide insight into why a program can mask errors or whether protecting a specific component may prevent more SDC than protecting the other components. Due to the complicated dynamic of a program's execution, a visual analysis technique is more accessible than a statistical method to study the corruption propagation, but limited visual analysis tools in the domain have been developed to accomplish these purposes.

In this paper, we introduce SpotSDC, an interactive visualization system, to study the impact and propagation of silent data corruption. Our key contributions are as follows:

- We develop a novel interactive visualization system for studying silent data corruption vulnerability and its impact on an HPC computation kernel.
- We design a new visual encoding to show the degree to which silent data corruption happens across multiple levels of granularity in numerical codes ranging from full programs, to functions, variables, or individual bits.
- We introduce new guidelines for sampling a program's fault injection space to improve the fidelity of the user understanding of fault tolerance and demonstrate how what we learn in the visualization can improve the traditional fault injection campaign methods.
- We provide new insights into how error propagates through the program execution and how visualization can improve and accelerate user understanding.

2 Related Work

In this section, we will discuss related work in understanding the impact of SDC faults and their propagation through HPC applications. We will also discuss how to prevent applications from SDC. To the best of our knowledge, applying visual techniques to understand a program's fault tolerance property is not well explored in the literature, and so we connect our work to various prior visualization systems for HPC, visualization of hierarchy, and ensemble visualization.

2.1 Understanding Silent Data Corruptions and Error Propagation

The fault injection campaign [15] is still the most popular approach to understanding how SDC affects a particular application. Menon et al. [16] defined and evaluated two metrics, SDC impact and SDC ratio, to understand the SDC properties of an application. Sangchoolie et al. [17] compared the analysis result of a single bit flip model with that of a multiple bit flip model and found that a single bit flip model causes more SDC than the multiple bit flip model. Di et al. [18] have also shown that a fault injection campaign with different program input may change the resiliency profile of that program. Other tools have been designed for understanding error propagation. Rizwan et al. [19] designed a tool to understand the speed and depth (e.g., the number of processes) of the error propagation in a parallel environment, and Guo et al. [20] developed a framework to extract the resiliency computation pattern that can mask an error. Li et al. [21] have proposed techniques to understand and characterize SDC error propagation in a GPU-based computation application. Skarin et al. [22] presented a visualization to understand the high-level fault injection and error propagation analysis of a Brake-by-Wire system with the assistance of Matlab Simulink. These tools provide only high-level aggregate information regarding the error propagation of the application. In contrast, we build an interactive visualization system for the fault injection data

to enable domain experts to do fine-grained analysis of the impact of SDC on an application.

2.2 Transient Fault Detection and Source Code Protection

Automatic detection methods have been developed to predict the resiliency of an application and protect the highly vulnerable program regions. Most of those methods [23], [24], [25], [26] have designed a probability model using the static analysis data and dynamic information of a program to predict the probability of a region being affected by transient faults. A highly vulnerable region will automatically insert redundant [27], [28] code and validation instructions at the compiler level to test the correctness of the computation. Probabilistic methods are often not suitable for high regret situations, and the transient fault's impact on the program output is not under consideration. Program resiliency can be improved by exploring the fault injection data [29] before implementing the design protection techniques. Our tool presents an exploration environment for the domain to examine multiple SDC properties and the protection cost at the same time. It helps researchers determine the cost and benefits before they design an algorithm to auto-detect the transient faults and protect the source code against them.

2.3 Visualization Software in HPC Programs

Visualization systems for HPC applications mostly focus on analyzing a program's data flow and communication pattern for improving performance. The Boxfish system [30] visualizes a 5D torus network through a flexible multi-view visualization. Ravel [31] has focused on providing a scalable solution to visualize complex trace records during a program's execution. Wongsuphasawat et al. [32] created a visualization tool to display the data-flow of a neural network model in the Tensor-Flow [33] environment. Sabin et al. designed CFGExplore [34] to interactively explore a program's structure and understand its control flow. Xie et al. [35] created a visual framework to understand the call stack trees and proposed stack2vec to detect anomalies. For an overview, please refer to the survey in [36] on performance visualization. To the best of our knowledge, there is no visualization tool dedicated to the task of understanding the effects of transient faults in HPC applications.

2.4 Hierarchical and Ensemble Visualization

SpotSDC is designed based on the hierarchical structure of an application, and a program's resiliency analysis is similar to ensemble analysis. A rich literature in the visualization of hierarchy and ensemble visualization has been developed in the visualization community. Niklas [37] used a hierarchical aggregation model to transform any visualization technique into a multiscale structure. Nobre et al. [38], [39] designed a hierarchical structure-based visualization framework to understand genealogies and coauthor network datasets. Collberg et al. [40] applied a hierarchical technique to visually present the evolution of a software development process. More hierarchical visualization techniques can be found in [41]. Potter et al. [42] created a multiple link view visualization framework, Ensemble-Vis, to analyze the distribution of scientific simulation data. Whitaker et al. [43] introduced innovative contour boxplots, and Mirzargar et al. [44] extended the technique for functions' level set and contour ensembles. For more details about ensemble visualization, refer to Wang's survey work [45].

3 Domain Background

This section provides basic background on how to simulate the appearance of transient faults during a program execution, how data are collected, what heuristics are used to measure the impact of the injected fault, and how we categorize the outcome of a fault injected experiment and quantify error in a fault injected program.

3.1 Fault Injection Model

As the size of computation components (e.g., transistor size) shrinks and the number of components increases, features such as the low-power and high-temperature tolerance are introduced into the computation system. These features will lead to a low threshold of a bit flip event and exacerbate the transient-fault phenomenon. In this work, we consider that bit flips happen only in a processor's functional units (e.g., floating-point unit) or registers. The error corruption in these components often leads to a program's data corruption. Therefore, our work focuses on the error corruption of a program's data variables. Transient faults also can corrupt the cache and main memory. However, researchers in computer architecture have designed solutions such as error-correcting code (ECC) [46] and parity bit, which exist in our current PC computer, to protect these components from the transient fault corruption.

We simulate the transient fault by using the standard single bit flip error model [15], [20], where the fault occurs as a single random bit flip error during the computation. In each fault injection experiment, the fault injection tool selectively introduces a bit flip at a specific bit location of a program's data variable. For each fault injection run, the fault injector records key information regarding the fault injected location, time, and outcome of the experiment in a log file. Transient faults can also affect control and pointer variables; however, they will usually cause the program to crash. A bit flip crash of a program is better than silent data corruption because a program crash is not silent, and domain experts can re-run the application to address the problem. Moreover, faults in control and pointer variables are well understood and can be mitigated using low-cost methods, but faults in data variables are complicated to diagnose and repair. We exclusively consider floating-point variable types for our work. Floating-point arithmetic is the predominant means for computation in HPC applications, and the standard IEEE 32 bits (float) and 64 bits (double) are the most common data types used in a program. The standard IEEE 754 format comprises a sign, an exponent, and mantissa bits.

3.2 Program Outcome

To evaluate the outcome of a fault injection experiment, we first compute the ground truth solution obtained from a run where no fault was injected. The output error is compared

against an acceptable error threshold set by domain experts. This acceptable error threshold indicates how much of an error is tolerable in the output. Based on the output of the program, the outcome of a fault injection experiment is classified as one of the following:

- **Crash**: After the error is injected into the program, the program throws an exception (e.g., segmentation fault, floating-point overflow) and stops running. We also define cases in which the output is NAN or infinity as a program crash because this type of output can be easily caught by the program's protection mechanism.
- **Masked**: A fault is injected into the program and the program finishes running and reports an output with error under a specified threshold defined by the domain expert.
- **Silent Data Corruption (SDC)**: Error is injected into the program and the program finishes running, but the output error is above the specified threshold.

3.3 SDC Properties

In this work, we employ three metrics, SDC frequency, SDC ratio, and SDC impact, to measure the SDC vulnerability of different program components. Each metric examines a program component's SDC feature from different perspectives.

3.3.1 SDC Frequency

The SDC frequency measures the fraction of SDCs caused by a program component in a campaign. Let n_{sdc} be the number of SDCs that occur in a program component and N_{sdc} be the total number of SDCs in the campaign.

$$\text{SDC Frequency} = \frac{n_{sdc}}{N_{sdc}} \tag{1}$$

3.3.2 SDC Ratio

The SDC ratio determines the sensitivity of a program component to a transient fault by measuring the probability that a transient fault in a component will result in an SDC. For a given component, let n_{sdc} be the number of times the result is SDC; n_{masked} be the number of times the result is Masked; and n_{crash} be the number of times the result is Crash.

$$\text{SDC Ratio} = \frac{n_{sdc}}{n_{sdc} + n_{masked} + n_{crash}} \tag{2}$$

3.3.3 SDC Impact

The SDC impact measures the impact of a program component on the final output of the program. Let O_g be the output of an error-free run and O_e be the output of an error-injected run that results in SDC for the same program. We then can define SDC impact as follows:

$$\text{SDC Impact} = |O_e - O_g| \tag{3}$$

3.4 Error Propagation

When an error corrupts a critical variable, the error will propagate to the other variables that depend on the corrupted variable and finally affect the program output. In this study, we monitor the fault injected program's behavior by comparing the value of critical variables in an error-free run and a fault injected run. In the domain, the error is often defined under the assumption that the computation flow of the fault injected run is the same as the error-free run. Therefore, our analysis focuses only on the part of the program execution that has the same execution path between two runs (e.g., if-else statement, iteration). Under this assumption, we define error at dynamic instruction i as $||\text{error\_free}_i - \text{fault\_injected}_i||$.

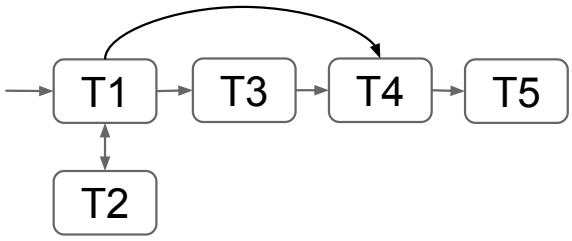


Fig. 2. Domain experts start with a dataset to study SDC properties over different components (T1) and optimize the number of samples in a component (T2) if necessary. Once these experts understand each program component's SDC property, they will either reason the observed feature over the source code (T4) and decide to protect this component (T5) or explore the error propagation start from this component and study the propagation process (T3).

4 Domain-Specific Requirements and Tasks

The development of SpotSDC involves two domain experts who are actively working on fault tolerance research and are also the co-authors of this paper. Furthermore, we have engaged with other researchers who are interested in a program's fault tolerance and have discussed how they analyze a program's resiliency. We designed SpotSDC based on the feedback about what will help them in their efforts. From these discussions, we developed five tasks that follow the task flow in Fig. 2.

T1: **Explore Program's SDC Characteristics.** SDC metrics in subsection 3.3 are commonly used to measure the influence of transient faults on a program component. During the analysis process, domain experts often print out the values of the metrics or plot a chart (e.g., histogram) to compare the values of metrics across different program components. This task is undertaken to help the domain experts understand the characteristics of a program under different granularities in terms of SDC properties and their variance overtime.

T2: **Examine Sample Coverage and Optimize Sample Quality.** A larger number of fault injections performed in the campaign will lead to better coverage of program components, which is critical for the analysis process. Good coverage over the exponent, sign, and mantissa portions of a program component is also important. However, domain experts often sample thousands of samples and start to analyze the program resiliency without understanding the coverage of the sample. With the information of sample distribution in a program component, it is also helpful to allow domain experts to manually add more samples to some of the components to optimize the fidelity of the analysis result.

T3: **Observe corruption Propagation.** Many programs have natural resiliency properties that can mask an error that occurs during the program execution. However, a bit

flip may cause a large error that leads to error explosion during the computation. For a fault injected run, information about how an error is dismissed by a program or where an error is amplified during the execution can help increase users' understanding of a program's resiliency property.

T4: **Source Code Correlation.** Visualization presents the data and displays an abstract SDC characteristic profile of a program. However, such information is usually not sufficient to explain the cause of the associated observations. Visualizing data in the context of the source code to be executed is, therefore, a key aspect in developing proper explanations of why different parts of the code may have different degrees of vulnerability. Moreover, the ability to visualize the data with source code is also useful when documenting the results and presenting research findings to other domain experts.

T5: **Study Code Protection.** For a vulnerable program component, adding protection mechanisms to improve its resilience is an important domain concern. The program component that causes the majority of the SDC may be a code region that is executed frequently. Protecting such regions will reduce the vulnerability of the code but cause significant performance overhead and reduce a program's efficiency. Presenting multiple SDC properties at the same time helps domain experts understand what kind of error is protected, how the overall SDC probability is reduced, and how the overhead of the computation is increased. Providing such insight helps domain experts choose better strategies to improve a program's resilience.

5 System Design and Visual Encoding

As shown in Fig. 1, SpotSDC has four visualization views to explore the fault injected dataset: a) an overview visualization, b) a sensitivity view, c) an error propagation view, and d) a source code view. SpotSDC loosely follows Schneiderman's mantra: "Overview first, zoom, filter, details on demand" [47]. It firstly provides an overview of the impact of a transient fault on a program. The user can then carry out further detailed analysis by zooming in on a program component and observing how an error occurring there will impact program output or how it propagates to other program components.

The system is designed to address the tasks proposed in section 4. To complete a domain task, it requires multiple information sources for joint analysis and online data aggregation for different granularity reasoning. To satisfy these requirements, our system is designed in a multiview layout to present different information at the same time. Each visual component of SpotSDC performs its functionality to answer a part of the domain questions. In Table 1, we show how the composition of the system's visual components can complete each domain task. As a running example, we use the data from an exhaustive fault injection campaign performed on a conjugate gradient (CG) [48] benchmark.

5.1 Overview Visualization

Having an overview of a program's resiliency properties over all program components is often the first step for domain experts. Fig. 3 shows the three panels of overview visualization: **(a)** Data Manipulation, **(b)** Levels of Detail, and **(c)** Statistical Summary.

TABLE 1
Composition of Visual Components to Address Domain Tasks.

Visual Components\Tasks		T1	T2	T3	T4	T5
Overview	Data Manipulation	✓				
	Levels of Detail	✓	✓		✓	✓
	Bit Sample Distribution	✓	✓			
	Bit Outcome Distribution	✓				
	SDC Impact Distribution	✓				✓
	Ratio and Frequency	✓	✓			✓
Sensitivity View		✓				
Source Code View					✓	✓
Propagation View				✓	✓	✓

Fig. 3. Display of a CG benchmark dataset to compare the SDC impact and SDC ratio of across multilevel program components; (a) provides a set of filter and dataset options, (b) enables users to manually decide on a hierarchical structure of data that is presented in (c), and (c) is a composition of visual components to display SDC properties.

5.1.1 Data Manipulation

Domain experts need to understand the SDC properties with respect to different bits, input error, program outcomes, and more. Data manipulation includes a list of data filter options that enable users to expose the visualization in different types of fault injection data and to examine different subsets of the data and gain a more comprehensive understanding of the resilient features of a program.

5.1.2 Levels of Detail

The dataset collected from a program's fault injection campaign will naturally contain the program's hierarchical structure information. To capitalize on such vital information, the overview needs the functionality to present the hierarchy of the data. Levels of Detail is an initialization component of the overview visualization that enables users to decide on the hierarchy of the fault injected data they want to present in the visualization. Instead of fixing the data hierarchy, this view provides additional flexibility that allows users to select and reorder the levels of aggregation for the data displayed in Fig. 3 **(C)**. The data tree could be a single node, which represents a program (e.g., **fft**), or a multilevel tree such as **program-function-variable-line number**, which is a common hierarchy that researchers apply to understand a program's resiliency (e.g., **fft-Scale-x-775**). Fault Injection Summary text with charts on the top of the data tree visually displays the fault injection

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2020.2994954, IEEE Transactions on Visualization and Computer Graphics

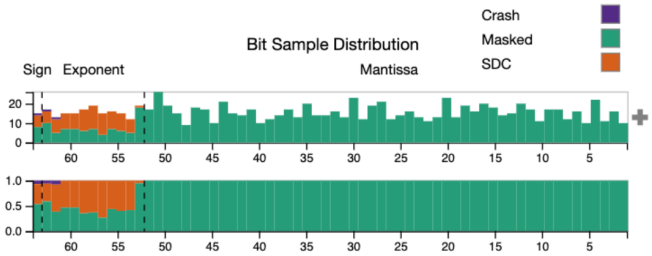


Fig. 4. The visualization on the top displays a program component's sample density distribution across different bit locations. The visualization on the bottom shows the different outcome ratios with respect to different bit locations.

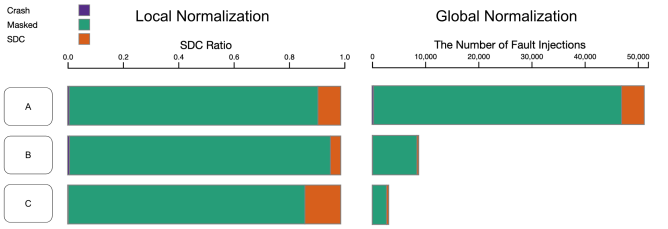


Fig. 5. Component A has the highest number of fault injections and the highest frequency of SDCs. Component C has the largest SDC ratio.

statistic over the entire program. This summary component enables users to maintain the global resiliency profile while exploring the fault tolerance property of each local program component.

During the design process, we also considered other common hierarchical techniques such as TreeMap [49] and pack layout [50] to present the hierarchical data and show the respective SDC ratio, but presenting more information such as bit-level detail and multiple properties at the same time in these techniques is difficult.

5.1.3 Statistical Summary Views

The statistical summary component contains multiple views, and each view aggregates the fault injection data and presents information for users to study the SDC properties (**T1**), examine sample coverage (**T2**), and study code protection (**T5**). In the visualization, the color for each outcome categories is selected from ColorBrewer [51], and all colors are color-blind safe.

Bit Sample Distribution View: Presenting information in bit-level detail provides insight into the impact of the bit flip across different bits. To visualize how such low-level analysis relates to the fault injection data, we have designed our visualization based on the IEEE 754 floating point format. The IEEE 754 floating format is a basic concept in the HPC domain that is easily understandable for the target users.

In Fig. 4, the visualization on the top shows a program component's sample density distribution of a fault injection campaign with respect to the bit location. The y-axis encodes the number of samples, and the x-axis encodes the bit location. At each bit location of the visualization on the top, a stacked column chart shows the accumulated number of Crash, SDC, and Masked outcomes due to transient faults (bit flip) at that bit location. The information presented in the visualization can be used to examine the sample coverage of a program component over sign bit, exponent bits, and mantissa bits. Balanced sample coverage is important to drive an accurate conclusion of a program component's resiliency. If users observe an unbalanced sample distribution, the visualization will enable interactively adding more samples to the current analysis dataset.

Bit Outcome Distribution: In Fig. 4, the visualization on the bottom shows the outcome ratio with respect to a different bit of a program component. The y-axis encodes the ratio value, and the x-axis encodes the bit location. Each bit location has an equal length stacked column chart that shows that the probabilities of a bit flip in different bits lead to three program outcomes. The information presented in the visualization can be used to study the statistical impact of a bit on a program's outcome.

Ratio and Frequency: The stacked row chart in Fig. 5 shows the overall fault injected frequency and outcome ratio. Users can examine local normalization to compare the SDC ratio of different program components and global normalization to target the program component that may cause the greatest amount of SDC and study the protection overhead. Local normalization compares the number of different outcome categories of a program component, which implies a component's SDC ratio. Global normalization shows the number of fault injection experiments in different program components, color encodes the outcome categories, and length represents the number of samples. A fault is injected into a data variable when it is called by a program; therefore, a program component tested by the fault injection campaign often implies the program component is frequently called by the program and adding protections there may cause significant overhead. Users can also selectively choose SDC outcomes data to compare the SDC frequency of different program components.

SDC Impact Distribution: Fig. 6 shows the SDC impact distribution of two program components. For all SDC outputs caused by a fault injection in a program component, output values are presented into the log scale. These data are then aggregated by a 10-bin histogram, and the percentage of elements that fall in each bit is encoded as a different color. The color in Fig. 6 changes from light to dark blue, representing the increasing percentage of SDC cases. Each color in the color map on the top represents a range of percentage values. A white rectangle represents a single value 0%, which is not linked to the continued color map on the right.

In our study, each fault injection experiment resulting in SDC can be an interesting case to study how the transient fault affects the program's output. Therefore, it is critical to distinguish whether a fault injection output value falls into a value range or not. We use a color map to describe error distribution, instead of using a histogram, with which recognizing a small portion frequency of an error-output-density distribution is difficult. The white rectangle is separated from the continued color map because each rectangle of the color map on the right represents a range of values, but the white rectangle represents only a single value. In Fig. 6, most of the SDC fault falls into the smallest error bin of the visualization, which visually attracts users' attention. In the visualization, large errors need more attention, but their colors are often light blue. To mitigate this contradiction, we adjust the size of the rectangles such that large rectangles represent a large error range and small rectangles represent

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2020.2994954, IEEE Transactions on Visualization and Computer Graphics

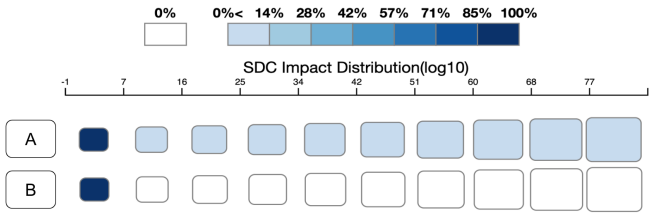


Fig. 6. SDC impact distribution of two program components. A SDC outcome caused by a fault injected in A can lead to a wide range of SDC impact. In B, a SDC outcome will introduce a relatively small SDC impact.

a small error range.

5.2 Source Code View and Sensitivity View

In SpotSDC, the source code is placed on the side of the dynamic visual analysis environment to provide additional context for reasoning and understanding the visual encoding results (**T4**). The source code view (Fig.1 (d)) is linked to the overview visualization and the propagation visualization by interactive operations (e.g., line selection).

The sensitivity analysis helps users identify a program's critical components and interesting propagation cases. SpotSDC contains a sensitivity view with input and output error distribution of fault injection experiments to present the sensitivity information. The sensitivity view is a scatter plot with a density distribution histogram attached on the x-axis and y-axis. Users can choose a component in the overview and display its fault injection experiments on the sensitivity view (Fig.1 (b)). A red dot is a fault injection result in SDC, and a green dot is a masked experiment. The histograms over the x-axis and y-axis indicate the density of different value regions. From the input and output error distribution, users can choose an interesting experiment case and present the propagation process in the propagation view to study the error propagation.

5.3 Propagation View

Studying how an error propagates through a computation will allow users to obtain a comprehensive picture of how the program behaves across its lifetime (**T3**). In Fig. 7, a propagation visualization shows a bit flip error propagating through a conjugate gradient program run. We design component Fig. 7 (a), which shows a summary view of a fault injected program's behavior. The x-axis is the time step, and the y-axis is the error scale. The blue line in the chart presents the variation of error across the program's lifetime. How we measure error is defined in section 3.4. A program computation's convergence or divergence indicates whether the injected error is dismissed or amplified during the computation. In component Fig. 7 (a), the purple line displays a metric (e.g., the residual in a linear solver, such as the conjugate gradient) used to measure such information. An error will start to propagate to the other elements once it is injected. To highlight the initial location of the error, in (b), the red triangle in the figure at the x-axis indicates the time step at which the error is injected.

A computation run has hundreds of time steps, and presenting all of them at the same time is challenging. To reduce the number of time steps, we place a lens in Fig. 7 (a) that enables users to select a time interval, zoom into a period of execution, and display the corruption detail in Fig. 7 (c). To address the problem of displaying many corrupted locations, the y-axis is designed with a program's hierarchical structure. Users can collapse the region they want to ignore or distribute the region they want to examine more closely.

View Fig. 7 (c) is a matrix-based visualization with the y-axis representing a program location and the x-axis representing the time step. The distributed squares in the matrix represent load or store dynamic instructions that are executed at a specific time step that belongs to a program's specific component. The number of continuous squares implies the number of data elements in a program component, which helps to reveal how many elements are corrupted during a period of time. The color assigned to each square represents the error scale at that dynamic instruction, with white indicating no error and dark red indicating large error over the program run. For the meaning of the color, refer to the color map in Fig. 7 (a).

Understanding the error scale across different program components and filtering out error can speed up the process of targeting the interesting error propagation region. The histogram in Fig.7 (e) shows the overall error distribution collected from the fault injected program execution. In the figure, users can selectively choose a different range of errors and display them in Fig. 7 (a) and Fig. 7 (c). The set of playback buttons in Fig. 7 (f) allows users to step line by line through the code's execution or animate the execution of the code, and enables them to see how and where the error increases/diminishes over time. The animation mimics the dynamic execution of a program run and is visually attractive to the domain experts.

During the visual design process, we firstly chose graph-based methods to design a propagation visualization and use the static analysis method to generate a program's data dependencies graph. Each node is a data variable, and the edge between each node is the data dependency between variables. However, static data dependencies do not reflect the dynamic condition of a program execution, and the static dependency information can be misleading or hard to interpret for propagation analysis. Extracting an accurate dynamic dependency of a program is not a trivial task. Once a program's scale grows, extracting an accurate data dependency graph become difficult [26], and sometimes it is worse to extract an accurate data dependency graph in a programming language with pointers such as C++.

5.4 Scalability

Scalability is one of the main concerns in our visualization system. SpotSDC currently focuses on analyzing a program's critical data variables instead of all the lines of code in the program. However, showing the fault injection data over all the variables at once or presenting the propagation data across different program components can still be a scalability challenge. To address this difficulty, both overview and propagation visualizations are designed in a hierarchical structure. During the exploration process, users can interactively collapse or expand the hierarchy to aggregate or distribute the data based on their needs. In the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2020.2994954, IEEE Transactions on Visualization and Computer Graphics

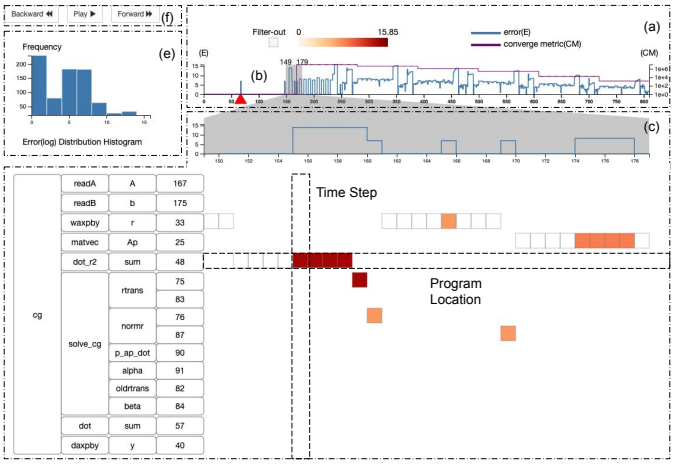


Fig. 7. A bit flip happens at one of the data elements of the variable sum in line 48 and propagates to some of the subsequent variables' data elements during the selected program execution time interval: (a) presents an overview of the error variation during a program's computation, and (b) indicates the initial fault injected location. The lens in (a) can be used to select a time interval and examine the propagation detail in (c). Users can use (e) to selectively filter out a certain range of error and present the rest of the error in (a) and (c), and (f) is a set of play buttons to animate the propagation process.

overview visualization, users can configure the attribute of each tree layer to present the fault injection data. In each layer, the order of nodes can be adjusted such that program components with a large SDC property value are placed on the top, and program components with a small value fall to the bottom. Users can manually decide which property to use to decide the order of the node in each layer. The sort process starts at child nodes belonging to the same parent node, which will be sorted by aggregating the data of the child nodes.

6 Use Cases

In this section, we demonstrate four common use cases, in which HPC experts utilize the proposed tool to address domain challenges and complete the tasks discussed in section 4. Here we focus on datasets collected from the fault injection campaign on two representative computation kernels: conjugate gradient (CG) and fast Fourier transform (FFT). Conjugate gradient [48] is a classical iterative algorithm for solving linear equations, and fast Fourier transform [52] is a widely used numerical algorithm that computes the discrete Fourier transform.

6.1 Study SDC Characteristics in Different Computation Iterations

Understanding the SDC properties **(T1)** with respect to different iterations can provide researchers with an opportunity to selectively protect a certain iteration to improve a program's resilience with less overhead. The visualization at the top of Fig. 8 displays the SDC impact distribution and SDC frequency of a bit flip injected in different iterations of the conjugate gradient (CG) program. As shown in (a), the visualization displays a filtered dataset with an outcome attribute containing only SDC and presents the SDC impact distribution for each iteration of the program. In the filtered dataset, iter 0 means the fault is injected in the initialization stage, which loads the data matrix and presets

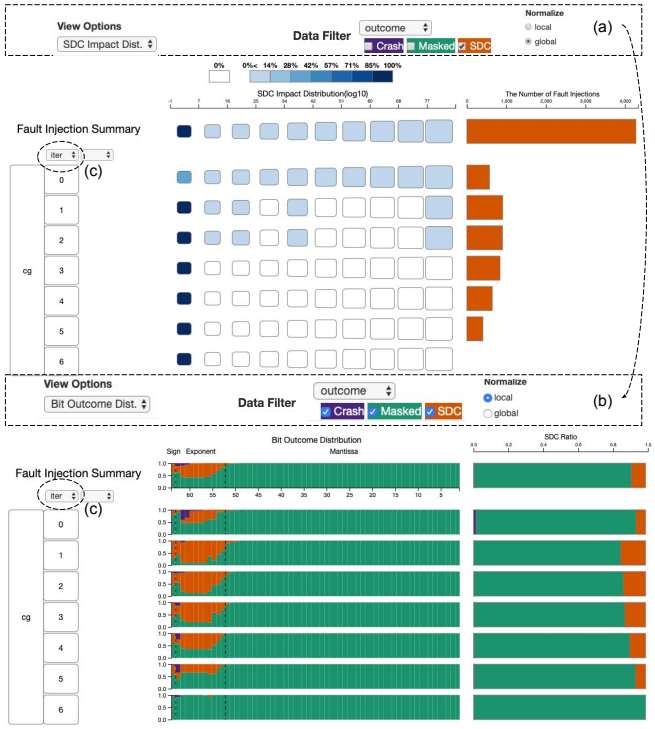


Fig. 8. Both visualizations present different iterations (c) to examine CG's resiliency property. In (a), filter out crash and masked cases and choose global normalization. The top visualization shows that the program's SDC impact and the SDC frequency decreases as the CG's iteration number increases. In (b), choose the full data set and select local normalization. The bottom visualization demonstrates the program's SDC ratio and its related bit location's SDC ratio decrease as the CG's iteration increases.

some parameters. A number i $(i > 0)$ indicates the fault is injected into the i^{th} iteration of the computation. Users are able to observe, in the SDC impact distribution, that errors injected in the initialization stage lead to a wide range of SDC impact. As the computation continues and bit flips are injected in the later iterations, the SDC impact shrinks. At the same time, the bar chart on the right shows that the SDC frequency also decreases as the iteration number increases.

The visualization at the bottom of Fig. 8 shows the SDC ratio with respect to each iteration and each iteration's different bits. Fig. 8 (b) shows an example workflow where users have switched to the bit outcome distribution visualization, selected the complete dataset without filter operations, and chosen local normalization for each computation iteration to compare the SDC ratio. As those row stacked charts show, the overall impact of the high bit decreases as does the SDC ratio. With the observation from the visualization, domain experts can infer that transient faults have a greater impact in CG's earlier iteration than in the later.

6.2 Examine and Improve Sample Coverage of Program's Components

A correct understanding of the SDC characteristics of a program component requires good sample coverage across different bits **(T2)**. In the bottom images of Fig. 8 and Fig. 9, the fault injection summary presents the bit flip outcome distribution across different bits on two programs. The result shows that a bit flip in exponent bits and sign bit has a greater impact than that in mantissa bits. Because

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2020.2994954, IEEE Transactions on Visualization and Computer Graphics

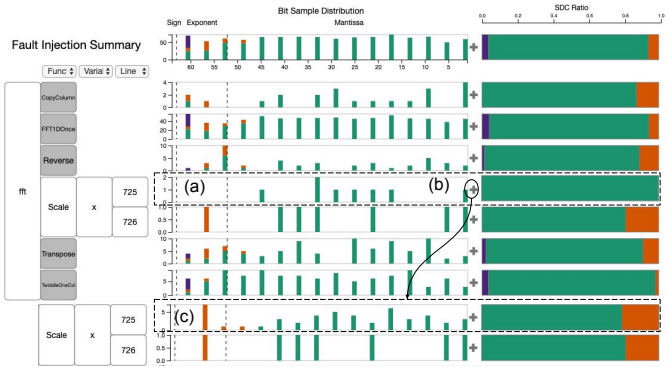


Fig. 9. No SDC outcome is caused by injecting a bit flip in line 725, but (a) shows that all the samples come from mantissa bits. After an interactive fault injection resample (b) in line 725, the visualization shows that line 725 has a relatively high SDC ratio in (c).

of the stochastic process, the bit flip sample distribution in a program component can have unbalanced coverage in mantissa bits, exponent bits, and the sign bit, which leads to inaccurate conclusions being drawn in this location. Here we present a random fault injection dataset of the fast Fourier transform program with which domain experts sample 1 in every 4 bits. The data are sampled in such a way as to reduce the number of samples needed to study a program's fault tolerance property but preserve the relatively high accuracy of the analysis result.

Fig. 9 displays a fault injection sample density distribution over different components of a fast Fourier transform (fft) program. In Fig. 9 (a), variable x at line 725 in the function scale (`fft-Scale-x-725`) does not have an SDC outcome for the fault injection experiments in this location. Without further investigation, users can make the incorrect inference that a bit flip in `fft-Scale-x-725` does not cause SDC. In the visualization, all the samples in `fft-Scale-x-725` are in mantissa bits, and no experiments are sampled in exponent bits. Users can interactively add more samples (Fig. 9 (b)) to line 725. In Fig. 9 (c) shows the resampling result and a few samples come from the exponent bits, and the SDC probability caused by exponent bits in this location is high. Moreover, the overall SDC ratio in `fft-Scale-x-725` is significantly large with the new dataset.

6.3 Identify Vulnerable Region and Study Code Protection

Identifying and reasoning about the program component that causes the greatest amount of SDC **(T4)** is a key task for domain experts to examine a program's resiliency. Studying the code protection in that particular component is a natural follow-up step **(T5)**. Fig. 10 shows the workflow of how users can use SpotSDC to identify the vulnerable code region and study potential protection strategies. In Fig. 10 (a), users select the program component that causes the maximum number of SDCs and the source code highlight `line 25`. In Fig. 10 (b), the source code shows that variable Ap at line 25 in function matvec (`matvec-Ap-25`) nests in a two-level `for` loop in the `matvec` function. If the user traces back to the regions that called the `matvec` function, in Fig. 10 (c) the `matvec` function is called in a `for` loop of the program's main solver function. Due to the three-level nested `for` loops, this line of code will be run many times, which also leads to a high fault injection probability. Even though `matvec-Ap-25` causes the greatest amount of SDC, the error it introduces to the computation output is smaller than the rest of the program components (e.g., `solve_cg`, `readA`). Meanwhile, adding a protection mechanism (e.g., instruction duplication) in `matvec-Ap-25` will cause a significant amount of overhead in this frequently called `for` loop **(T3)**. How to protect such a code region depends on the domain experts' priority: the computation result's accuracy or the computation's performance.

6.4 Visualizing an Error Propagates Through Computations

Observing how an error propagates **(T3)** through a computation and analyzing why the error is masked or explodes during the execution will help domain experts gain a better understanding of the resiliency of a program. The conjugate gradient algorithm [10] has a natural resiliency property that can mitigate a certain amount of errors.

The propagation visualization at the top of Fig. 11 presents a fault injection case where a fault is injected into the variable sum at line 48 in function dot_r2 (`dot_r2-sum-48`), which results in silent data corruption. In this view, the top line chart shows that the initial error is injected around 150th time steps, but the error starts to explode after a program execution around the 250th time step. By moving the lens and zooming in on the time interval at which the error starts to explode in (a), in (b) users can tell that the error starts to explode at variable alpha at line 91 in function solve_cg (`solve_cg-alpha-91`), and the error amplification trend continues in the rest of the program variables.

On the other hand, the bottom visualization in Fig. 11 presents a fault injection case where a bit flip is injected into a variable beta at line 84 in function solve_cg (`solve_cg-beta-84`), but the final outcome is masked. The top line chart shows the initial error is injected around the 230th time step. The converge metric and error explode at the beginning but decrease after a certain number of time step. By moving the lens and zooming in on the time interval in (c) where the error dismisses, users can tell from (d) that the error starts to dismiss at `solve_cg-alpha-91`. As the computation continues, the error continues to decrease.

The explanation from domain experts is that the two case studies provide them with valuable insights about the impact of errors in certain variables. It shows that `solve_cg-alpha-91` is a critical variable, and an error in it can significantly affect the output. This effect has to do with the fact that the conjugate gradient is an iterative algorithm, where the solution at an iteration is a linear combination of the Krylov vectors computed in the previous iterations. Note that the coefficients of the Krylov vectors are given by the alpha, and the alpha in subsequent iteration depends on its value computed in previous iterations, thereby compounding the errors and ultimately corrupting the final output.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2020.2994954, IEEE Transactions on Visualization and Computer Graphics

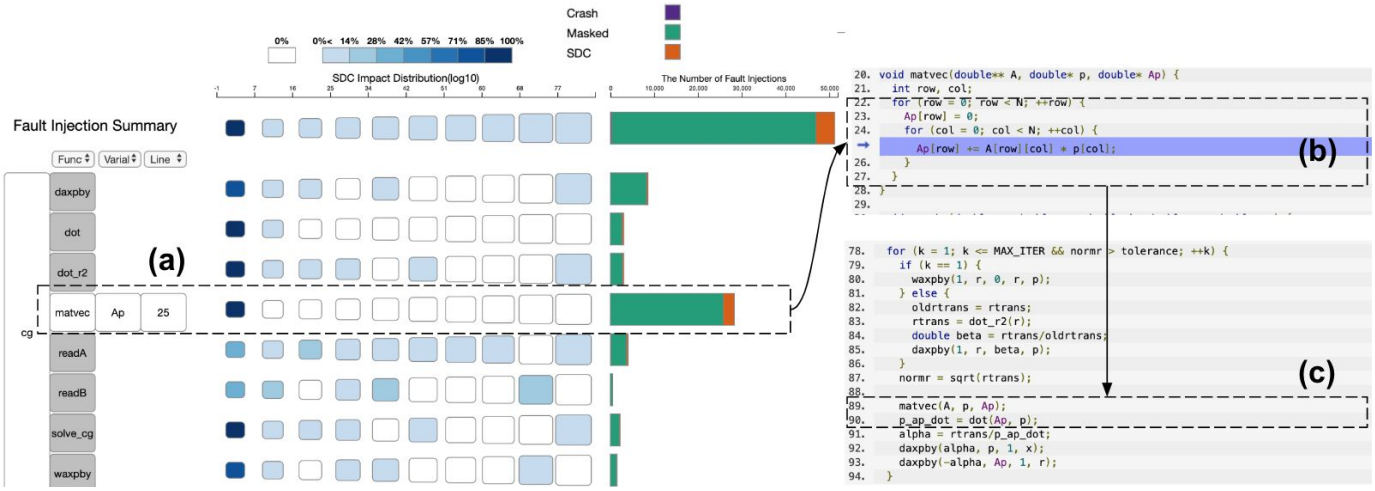


Fig. 10. In (a), line 25 has the maximum number of SDCs compared to the rest of the program components and the maximum number of fault injections, which implies this line of code will be executed by the program most frequently. To interpret the result, link line 25 to the source code (b), which shows that line 25 is called by a two-level for loop in function matvec. In (c), function matevec is also called by a for loop. The program output error scale caused by injecting error in line 25 is relatively smaller compared to the other program components, and adding protection to reduce the probability of SDC in line 25 will introduce significant overhead.

7 Improve Accuracy of Fault Injection Campaign

In Section 6.2, we demonstrated a use case in which unbalanced sample coverage in different bits of a program component leads to an incorrect fault tolerance profile. To improve the fidelity of the analysis, SpotSDC enables users to interactively sample more data for a program component. Often, the number of samples needed will vary over the corpus of code being analyzed. Therefore, using a single number for resampling is insufficient. For the IEEE double floating point representation, in each uniform fault injection campaign, $\frac{52}{64}$ of the samples flip a mantissa bit, which causes fewer SDCs, and only $\frac{12}{64}$ of the samples come from the region that causes the majority of SDCs. Therefore, this uniform sampling strategy wastes a large amount of samples on the region that does not provide useful information to understand the SDC characteristics of a program.

In this section, we will discuss how we can programmatically decide the number of samples that should be generated for a specific program component when the user requests more samples. We also demonstrate two simple sampling methods that change the sampling structure by taking more samples from the sign bit and exponent bits and fewer samples from the mantissa bits. Because the new sampling methods do not follow the uniform assumption, the final SDC metric value needs to be scaled to account for the no longer uniform distribution. How to apply these rescaled processes in our new sampling methods will also be discussed. Our results are comparable to a gold standard ground truth from the exhaustive campaign, and we show the two sampling methods outperform the classic random, uniform sampling.

7.1 Interactive Sample Fault Injection Data

A classical fault injection campaign with uniform sampling does not guarantee that each program component will have balanced coverage over different bits. To address the unbalanced sample problem during exploration, our tool provides an interactive operation that allows domain experts to connect to their fault injection tool, manually sample more data for a specific program component, and analyze its resiliency. Users can check a program component's sample distribution by observing the visualization of the distribution, which is presented in Fig. 4, and manually add samples to the location.

Leveugle et al. [53] formulated an equation to calculate the number of error-injected samples required to bound a program's fault tolerance metrics (e.g., the percentage of failure if faults are injected into a program) within a specified error tolerance, and discussed the sensitivity of the different parameters of the formula. In SpotSDC, we use the same formula but apply it in an interactive context and study a specific program component's SDC properties (e.g., a function or a line of code). The number of samples needed for a program component can be inferred from equation 4 with an initial confidence interval and margin errors. The size of new samples has a statistical significance that the SDC property (e.g., SDC ratio) of the component is in $[SDC_{Ratio} - \epsilon, SDC_{Ratio} + \epsilon]$ with a t confidence interval. The value scale of each variable is often decided by the domain experts.

$$n = \frac{N}{1 + \epsilon^2 \times \frac{N-1}{t^2 \times p \times (1-p)}} \tag{4}$$

In equation 4, N is the total number of the population, ϵ represents the margin error, p is the estimated ratio of a sample dataset having a specific characteristic (e.g., SDC ratio), t represents the confidence interval, and a 95% confidence is usually chosen. The interactive sampling method manually adds more samples to a program component, which does not follow the uniform assumption. Therefore, users often use the interactive fault injection campaign to analyze a program component's SDC property locally. To use the resampled data to analyze the program's global resiliency property, the manually injected data needs to be rescaled to calculate the entire program's resiliency property. For example, consider a particular program where line A has 2% of the total number of samples in a uniformly sampled

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2020.2994954, IEEE Transactions on Visualization and Computer Graphics

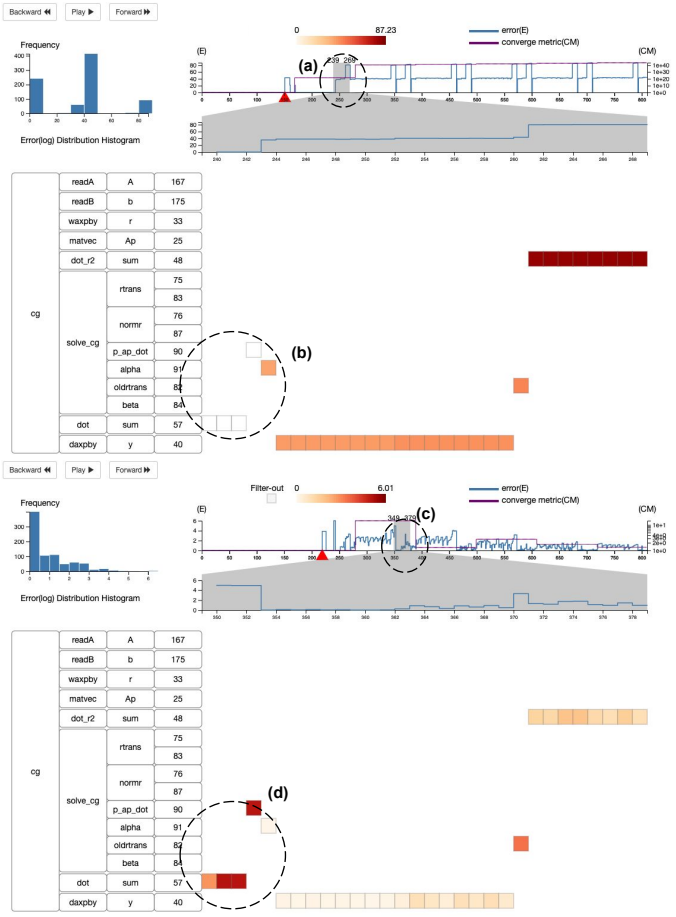


Fig. 11. The propagation visualization at the top displays a bit flip injected into data variable `dot_r2-sum-48` that propagates to the entire program and results in SDC. The propagation visualization at the bottom displays a bit flip injected into data variable `solve_cg-beta-84` that results in Masked. In (a) and (c), the lens is placed on a time interval that the error starts to explode/dismiss. Parts (b) and (d) show where the error starts to explode/dismiss. In both cases, the error starts to explode/dismiss at line `solve_cg-alpha-91`.

fault injection campaign, and a user elects to manually add more samples to line A to analyze the resiliency property. In order to calculate the global SDC ratio, the SDC ratio of line A must contribute only 2% to the final SDC ratio no matter how many new samples are drawn in this location. The above description can be formulated as in equation 7, where SDC_A is the SDC metric value calculated from program component A, and SDC_{Other} is the SDC metric value calculated from the rest of the program components.

$$SDC_{Metric} = SDC_A \times 2\% + 98\% \times SDC_{Other} \quad (5)$$

7.2 Improve Fault Injection Campaign

An exhaust fault injection campaign of a conjugate gradient algorithm with 8x8 matrix takes half of a day and needs 1.49 GB hardware space in a single machine with Intel i7 CPU. In the same machine, the running time increases to 5 days, and the data storage increases to 15.57 GB for solving a 20 x 20 matrix. A uniform random sampling approach can speed up the process, but it is not efficient at capturing the SDC samples as the majority of mantissa bits have a minor impact on a program's output. Previous research [54] has designed a method that generates more data from a high uncertain

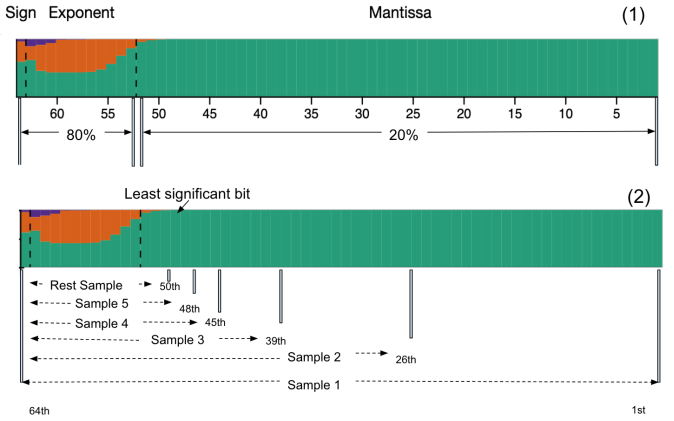


Fig. 12. (1) The 8vs2 sampling strategy with which 80% of the samples come from sign and exponent bits and 20% of the samples draw from mantissa bits. (2) The iterative sampling method, which separates the sampling process into multiple steps. In the initial stage (sample 1), the method samples the entire sample space. In the second round sampling (sample 2), the method ignores half of the mantissa bits as the sampling process continues. The method applies the binary searches strategy during each sample iteration to find the least significant mantissa bit, and then the rest of the sample will be drawn between only the sign bit and the least significant mantissa bit.

time interval of a program computation, but the sample quality of different bits is not considered in the approach. Samples from high bits capture more SDC events than samples in the mantissa bits. To verify the hypothesis, we compare two simple sampling methods, 8vs2 sampling and iterative sampling, with uniform sampling to demonstrate how the observation can help the HPC community improve the fault injection campaign. In the following discussion, we define all exponent bits and the sign bit as **high bits**.

7.2.1 8vs2 Sampling

In Fig. 12 (1), 8vs2 samples 80% of the data from the high bits and 20% from the mantissa bits. Because the sampling process is not uniform, the formula for calculating the SDC ratio, SDC impact, and other metrics needs to be multiplied by a scaling parameter. Under the uniform assumption, the information calculated from high bits will contribute to the final result $\frac{12}{64}$, and that from mantissa bits will contribute to the result by $\frac{52}{64}$ under the IEEE 754 double precision format. In equation 6, H is the SDC metric value calculated from high-bit samples, and M is the SDC metric value calculated from mantissa-bit samples.

$$SDC_{Metric} = H \times \frac{12}{64} + M \times \frac{52}{64} \quad (6)$$

7.2.2 Iterative Sampling

In Fig.12 (2), the adaptive sampling method separates the sampling process into K iterations. In each iteration, the sampling method not only samples the data but also performs a binary search operation to find the lowest impact mantissa bit, which is the index of the lowest mantissa bit resulting in an SDC outcome. Here, we use a 64-bit floating point as an example. The indexes of the mantissa bits are from 1 to 52, the indexes of the exponent bits are from 53 to 63, and the index of the sign bit is 64. In mantissa and exponent bits, the higher the bit index, the greater the impact of the bit on the floating point value.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2020.2994954, IEEE Transactions on Visualization and Computer Graphics

For the first iteration, the method will sample from 1 to 64 bits and find the index of the lowest impact mantissa bit. If the index is larger than 26, which is the index of the middle mantissa bit, then the next round will sample the 26^{th} bit to the 64^{th} bit. The sample space in the mantissa bits will shrink by half in each iteration until the lowest mantissa bit causing SDC is found. After that, the rest of the iteration will sample only the lowest impact mantissa bit to the 64^{th} bit. The worst case is that previous sample iterations do not find a bit flip in the mantissa bits that cause the SDC outcome. A 64-bit floating point variable has 52 mantissa bits, and the binary search method will try a maximum of 6 sample iterations to find the lowest bit that causes SDC. Once $K > 6$, the rest of the iterations will sample only the regions that cause SDC based on the previous sample iterations' information.

Calculating the SDC metric value again requires a rescaled operation. In the second iteration, assume the lowest impact mantissa bit index is larger than 26. The iterative method samples half of the mantissa bits (26^{th} to 52^{th} bit), and all high bits (12 bits) if the lowest impact mantissa bit is in the sample region. The SDC metric value needs to multiply $\frac{12+26}{64}$, and a similar rescaled operation needs to apply to the rest of the iterations. The final SDC metric is the mean of each iteration's result. In equation 7, SDC^i_{Metric} is the SDC metric value calculated in the i^{th} iteration, where n_i is the number of mantissa bits in the sample space of the i^{th} iteration.

$$SDC_{Metric} = \frac{1}{k} \times \sum_{i=1}^{k} SDC^i_{Metric} \times \frac{12 + n_i}{64} \tag{7}$$

7.2.3 Experiment

To evaluate the proposed sampling scheme, we performed uniform sampling, 8vs2 sampling, and iterative sampling on the conjugate gradient and fast Fourier transform programs. In each experiment, each method sampled 1% of the total population. For iterative sampling, the value K is set as 10. We performed 100 experiments and used the mean of the SDC metric value over the program's different lines of code to compare the sample quality of the different methods.

In Fig. 13, we compare the expected SDC ratio of the three sampling methods with the ground truth. The top image shows the expected SDC ratio over different lines of code on the conjugate gradient. The bottom image displays the same information except on the fast Fourier transform dataset. In both datasets, the figure shows the mean SDC ratio of different lines of code is close to the ground truth. Each bar also shows that the 95% confidence interval of the relative mean SDC ratio. As the figure shows, the iterative method has the narrowest interval, and high-bit sampling has a relatively larger confidence interval than the random sampling method. By comparing the SDC ratio, we found that the iterative method generates more stable results than the other methods.

In Fig. 14, we compare the expected SDC impact of the three sampling methods. All values are calculated in the log domain because of the large value scale variation of the SDC impact. As the figure shows, the ground truth is larger than the expected SDC impact in both the conjugate gradient and fast Fourier transform programs. A gap exists between the sampling approximation method and the golden ground truth. However, in both programs, the iterative method and 8vs2 method give better approximation results compared to the random method, and the 8vs2 sampling result is better than the iterative sampling method in general. The random sampling method's expected SDC impact is smaller than that of the other two methods. The new sampling methods are more desirable to approximate SDC impact than the uniform random sampling.

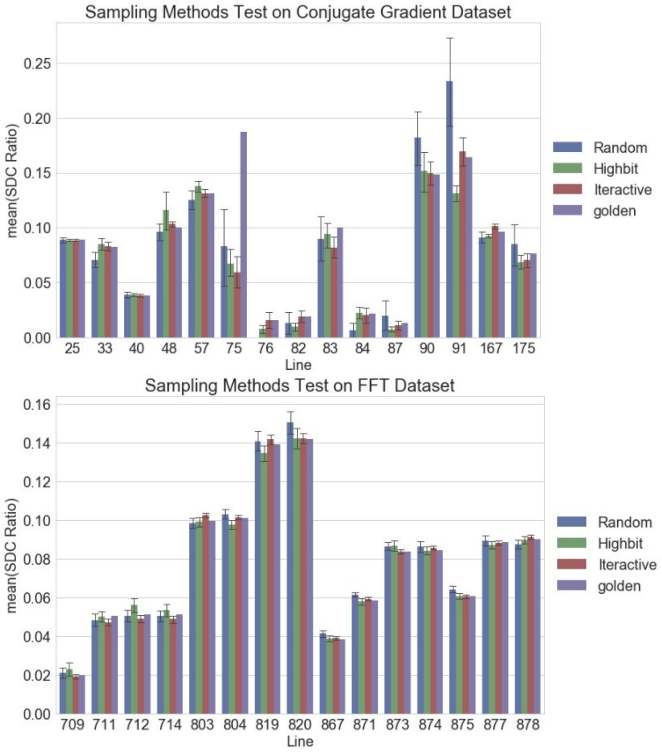


Fig. 13. Testing three sampling methods on conjugate gradient and fast Fourier transform and comparing the expected SDC ratio of different lines of code with the golden ground truth. Most of the mean SDC ratios are close to the ground truth, but the iterative sampling method has the smallest confidence interval compared to the other two sampling methods.

In the above two experiments, with the same number of samples, the new sampling methods perform better statistically than the classical random sampling by comparing the different SDC metrics, which allows us to achieve better sampling efficiency. 8vs2 sampling demonstrates better performance when measuring the program components that cause large output error compared with the other two sampling methods. This method can simply and efficiently be deployed in any fault injection campaign, or it can be used as a prototype method to understand a program's resiliency in the initial state. Iterative sampling methods better capture a more accurate SDC ratio than the other two methods.

The improvement of the analysis fidelity can also be reflected in the visualization. In Fig. 15, we use SpotSDC to compare three 1000 samples datasets with the exhaust fault injection campaign using the CG benchmark. The exhaust dataset shows the fault injection can cause a range of error output. The result of 8vs2 sampling is closer to the exhaust dataset, and the interactive sampling method is slightly better than uniform sampling. The visualization result matches with the SDC impact analysis result in Fig. 14.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2020.2994954, IEEE Transactions on Visualization and Computer Graphics

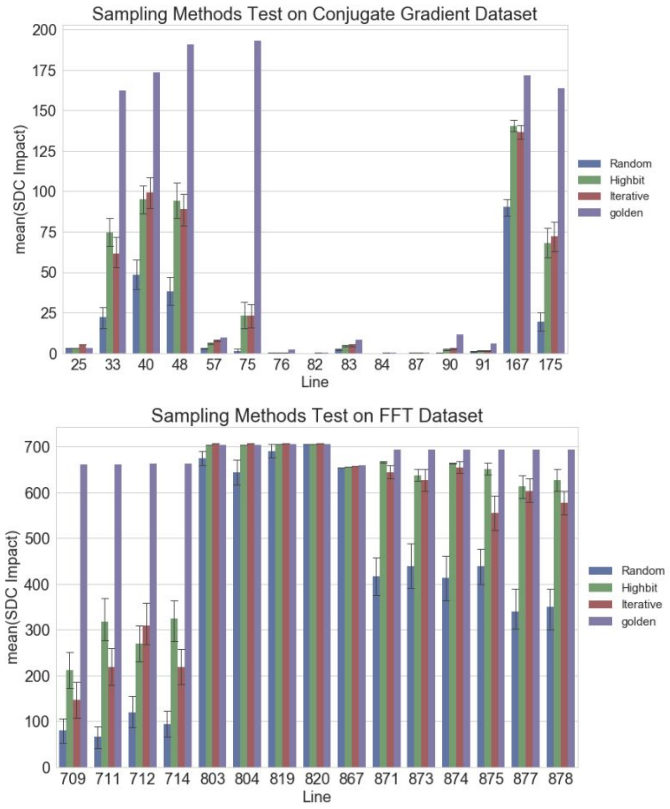


Fig. 14. Comparing the expected SDC impact of different lines of code with the golden ground truth using three sampling methods tested on conjugate gradient and fast Fourier transform. The expected SDC impact of high-bit sampling and iterative sampling is closer to the ground truth than random sampling.

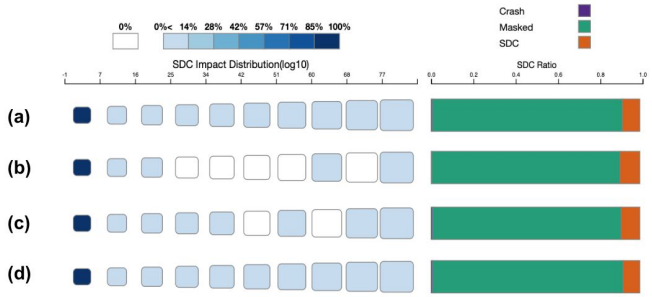


Fig. 15. Each of three sampling methods samples 1000 points and compares the SDC impact and SDC ratio with the exhaust fault injection dataset in the SpotSDC. (a) exhaust fault injection result. (b) uniform random sampling. (c) interactive sampling and (d) 8vs2 sampling.

8 Limitations and Future Work

Tracking how an error propagates through a program computation is a challenging task. It is difficult to define error in the propagation process. A single bit flip error may have an unpredictable impact on the behavior of a program. In this paper, we define error as the absolute difference between a fault injected run and an error-free run under the assumption that a bit flip error will not change the computation flow of a program. However, such an assumption may not hold in some fault injection experiments. For example, a bit flip error may cause an iterative program to run more iterations or change the control flow of a program, such as an if-else statement. Defining a general metric to measure the error between two different computation flows is not trivial, and will be explored in the future.

The scalability challenge is not completely resolved in SpotSDC. In our current system, visualizations are designed to understand the fault tolerance property of HPC computation kernels, which are relatively small compared to the other large programs, not to mention the large parallel computations that run a few days or a couple of weeks. The scale of the computation increases not only the complexity and challenge of the visualization design but also the size of the dataset a visualization system needs to process. A short survey in previous research [55] found that the potential number of fault injection tests of a program can reach a billion. Managing a dataset at such a scale in an interactive visualization is difficult. Our new sampling method can be part of the solution to mitigate this problem, but an innovative data structure is also necessary. On the other hand, how to select a few interesting fault injection cases instead of the exhaust fault injection dataset to understand a program's resiliency will also be an interesting direction to address this challenge.

The design of our visualization is a reference for researchers interested in designing a tool to study tree structure datasets with multiple attributes. Our visualization research not only will help researchers understand the impact of silent data corruption during program computation but also may be extended to help the research fields that are interested in taking advantage of a program's natural resiliency and trade it for other benefits (e.g., performance). An example is the computation's precision fine tuning [56], which uses a low-precision data type to improve performance with a tolerated round error. Another example is lossy compression for a large-scale simulation computation [57] that uses a lossy compression technique to mitigate the I/O bottleneck problem in an HPC system.

9 Conclusion

In this research, we designed a visualization tool, SpotSDC, to help domain experts understand the impact of silent data corruption on an HPC computation kernel. SpotSDC enables users to understand the SDC impact at varying levels of detail, and provides them with additional information to study the protection trade-off. SpotSDC also allows users to observe the behavior of a fault injected program, which gives them additional insight into how an error propagates through a program computation. In the end, we have shown how the observations from SpotSDC can optimize fault injection data and improve the traditional fault injection campaign.

10 Acknowledgements

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-764021-DRAFT).

References

[1] S. S. Mukherjee, J. Emer, and S. K. Reinhardt, "The soft error problem: An architectural perspective," in *11th International Symposium on High-Performance Computer Architecture*. IEEE, 2005, pp. 243–247.

[2] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson *et al.*, "Addressing failures in exascale computing," *The International Journal of High Performance Computing Applications*, vol. 28, no. 2, pp. 129–173, 2014.

[3] A. Geist, "Supercomputing's monster in the closet," *IEEE Spectrum*, vol. 53, no. 3, pp. 30–35, 2016.

[4] D. Tiwari, S. Gupta, J. Rogers, D. Maxwell, P. Rech, S. Vazhkudai, D. Oliveira, D. Londo, N. DeBardeleben, P. Navaux *et al.*, "Understanding gpu errors on large-scale hpc systems and the implications for system design and operation," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. IEEE, 2015, pp. 331–342.

[5] K. Ferreira, J. Stearley, J. H. Laros III, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold, "Evaluating the viability of process replication reliability for exascale systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, p. 44.

[6] C. Engelmann, H. H. Ong, and S. L. Scott, "The case for modular redundancy in large-scale high performance computing systems," in *Proceedings of the 8th IASTED international conference on parallel and distributed computing and networks (PDCN)*, 2009, pp. 189–194.

[7] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell, "Detection and correction of silent data corruption for large-scale high-performance computing," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012, p. 78.

[8] J. S. Hu, F. Li, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, "Compiler-directed instruction duplication for soft error detection," in *Design, Automation and Test in Europe*. IEEE, 2005, pp. 1056–1057.

[9] J. Elliott, M. Hoemmen, and F. Mueller, "Evaluating the impact of sdc on the gmres iterative solver," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 2014, pp. 1193–1202.

[10] M. Casas, B. R. de Supinski, G. Bronevetsky, and M. Schulz, "Fault resilience of the algebraic multi-grid solver," in *Proceedings of the 26th ACM international conference on Supercomputing*. ACM, 2012, pp. 91–100.

[11] P. Sao and R. Vuduc, "Self-stabilizing iterative solvers," in *Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*. ACM, 2013, p. 4.

[12] V. Piuri, "Analysis of fault tolerance in artificial neural networks," *Journal of Parallel and Distributed Computing*, vol. 61, no. 1, pp. 18–48, 2001.

[13] A. S. Nielsen, "Scaling and resilience in numerical algorithms for exascale computing," Ph.D. dissertation, Ecole Polytechnique Fédérale de Lausanne, 2018.

[14] K. S. Yim, C. Pham, M. Saleheen, Z. Kalbarczyk, and R. Iyer, "Hauberk: Lightweight silent data corruption error detector for gpgpu," in *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2011, pp. 287–300.

[15] J. Wei, A. Thomas, G. Li, and K. Pattabiraman, "Quantifying the accuracy of high-level fault injection techniques for hardware faults," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2014, pp. 375–382.

[16] H. Menon and K. Mohror, "Discvar: Discovering critical variables using algorithmic differentiation for transient faults," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '18. New York, NY, USA: ACM, 2018, pp. 195–206. [Online]. Available: http://doi.acm.org/10.1145/3178487.3178502

[17] B. Sangchoolie, K. Pattabiraman, and J. Karlsson, "One bit is (not) enough: An empirical study of the impact of single and multiple bit-flip errors," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2017, pp. 97–108.

[18] D. Di Leo, F. Ayatolahi, B. Sangchoolie, J. Karlsson, and R. Johansson, "On the impact of hardware faults–an investigation of the relationship between workload inputs and failure mode distributions," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2012, pp. 198–209.

[19] R. A. Ashraf, R. Gioiosa, G. Kestor, R. F. DeMara, C.-Y. Cher, and P. Bose, "Understanding the propagation of transient errors in hpc applications," in *High Performance Computing, Networking, Storage and Analysis, 2015 SC-International Conference for*. IEEE, 2015, pp. 1–12.

[20] L. Guo, D. Li, I. Laguna, and M. Schulz, "Fliptracker: Understanding natural error resilience in hpc applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18. Piscataway, NJ, USA: IEEE Press, 2018, pp. 8:1–8:14. [Online]. Available: http://dl.acm.org/citation.cfm?id=3291656.3291667

[21] G. Li, K. Pattabiraman, C.-Y. Cher, and P. Bose, "Understanding error propagation in gpgpu applications," in *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*. IEEE, 2016, pp. 240–251.

[22] D. Skarin, J. Vinter, and R. Svenningsson, "Visualization of model-implemented fault injection experiments," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2014, pp. 219–230.

[23] Q. Lu, K. Pattabiraman, M. S. Gupta, and J. A. Rivers, "Sdctune: a model for predicting the sdc proneness of an application for configurable protection," in *Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. ACM, 2014, p. 23.

[24] I. Laguna, M. Schulz, D. F. Richards, J. Calhoun, and L. Olson, "Ipas: Intelligent protection against silent output corruption in scientific applications," in *Code Generation and Optimization (CGO), 2016 IEEE/ACM International Symposium on*. IEEE, 2016, pp. 227–238.

[25] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: probabilistic soft error reliability on the cheap," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 1. ACM, 2010, pp. 385–396.

[26] G. Li, K. Pattabiraman, S. K. S. Hari, M. Sullivan, and T. Tsai, "Modeling soft-error propagation in programs," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2018, pp. 27–38.

[27] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee, "Software-controlled fault tolerance," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 2, no. 4, pp. 366–396, 2005.

[28] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "Swift: Software implemented fault tolerance," in *Proceedings of the international symposium on Code generation and optimization*. IEEE Computer Society, 2005, pp. 243–254.

[29] S. K. S. Hari, S. V. Adve, and H. Naeimi, "Low-cost program-level detectors for reducing silent data corruptions," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*. IEEE, 2012, pp. 1–12.

[30] A. G. Landge, J. A. Levine, A. Bhatele, K. E. Isaacs, T. Gamblin, M. Schulz, S. H. Langer, P.-T. Bremer, and V. Pascucci, "Visualizing network traffic to understand the performance of massively parallel simulations," *IEEE Transactions on Visualization and Computing Graphics, Proceedings of InfoVis*, vol. 18, no. 12, pp. 2467–2476, 2012.

[31] K. E. Isaacs, P.-T. Bremer, I. Jusufi, T. Gamblin, A. Bhatele, M. Schulz, and B. Hamann, "Combing the communication hairball: Visualizing large-scale parallel execution traces using logical time," *IEEE Transactions on Visualization & Computer Graphics*, no. 1, pp. 1–1, 2014.

[32] F. M. Hohman, M. Kahng, R. Pienta, and D. H. Chau, "Visual analytics in deep learning: An interrogative survey for the next frontiers," *IEEE transactions on visualization and computer graphics*, 2018.

[33] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 265–283.

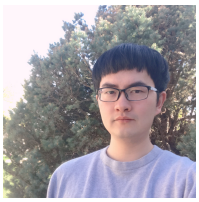
[34] S. Devkota and K. E. Isaacs, "Cfgexplorer: Designing a visual control flow analytics system around basic program analysis operations," *Comput. Graph. Forum*, vol. 37, no. 3, pp. 453–464, 2018. [Online]. Available: https://doi.org/10.1111/cgf.13433

[35] C. Xie, W. Xu, and K. Mueller, "A visual analytics framework for the detection of anomalous call stack trees in high performance computing applications," *IEEE Transactions on Visualization Computer Graphics*, p. 1. [Online]. Available: doi.ieeecomputersociety.org/10.1109/TVCG.2018.2865026

[36] K. E. Isaacs, A. Giménez, I. Jusufi, T. Gamblin, A. Bhatele, M. Schulz, B. Hamann, and P.-T. Bremer, "State of the art of performance visualization."

[37] N. Elmqvist and J.-D. Fekete, "Hierarchical aggregation for information visualization: Overview, techniques, and design guide-

lines," *IEEE Transactions on Visualization and Computer Graphics*, vol. 16, no. 3, pp. 439–454, 2009.
[38] C. Nobre, N. Gehlenborg, H. Coon, and A. Lex, "Lineage: Visualizing multivariate clinical data in genealogy graphs," *IEEE transactions on visualization and computer graphics*, vol. 25, no. 3, pp. 1543–1558, 2018.
[39] C. Nobre, M. Streit, and A. Lex, "Juniper: A tree+ table approach to multivariate graph visualization," *IEEE transactions on visualization and computer graphics*, vol. 25, no. 1, pp. 544–554, 2018.
[40] C. Collberg, S. Kobourov, J. Nagra, J. Pitts, and K. Wampler, "A system for graph-based visualization of the evolution of software," in *Proceedings of the 2003 ACM symposium on Software visualization*. ACM, 2003, pp. 77–ff.
[41] H.-J. Schulz, "Treevis. net: A tree visualization reference," *IEEE Computer Graphics and Applications*, vol. 31, no. 6, pp. 11–15, 2011.
[42] K. Potter, A. Wilson, P.-T. Bremer, D. Williams, C. Doutriaux, V. Pascucci, and C. R. Johnson, "Ensemble-vis: A framework for the statistical visualization of ensemble data," in *2009 IEEE International Conference on Data Mining Workshops*. IEEE, 2009, pp. 233–240.
[43] R. T. Whitaker, M. Mirzargar, and R. M. Kirby, "Contour boxplots: A method for characterizing uncertainty in feature sets from simulation ensembles," *IEEE Transactions on Visualization and Computer Graphics*, vol. 19, no. 12, pp. 2713–2722, 2013.
[44] M. Mirzargar, R. T. Whitaker, and R. M. Kirby, "Curve boxplot: Generalization of boxplot for ensembles of curves," *IEEE transactions on visualization and computer graphics*, vol. 20, no. 12, pp. 2654–2663, 2014.
[45] J. Wang, S. Hazarika, C. Li, and H.-W. Shen, "Visualization and visual analysis of ensemble data: A survey," *IEEE transactions on visualization and computer graphics*, 2018.
[46] R. W. Hamming, "Error detecting and error correcting codes," *The Bell system technical journal*, vol. 29, no. 2, pp. 147–160, 1950.
[47] B. Shneiderman, "The eyes have it: A task by data type taxonomy for information visualizations," in *Visual Languages, 1996. Proceedings., IEEE Symposium on*. IEEE, 1996, pp. 336–343.
[48] J. R. Shewchuk *et al.*, "An introduction to the conjugate gradient method without the agonizing pain," 1994.
[49] B. Shneiderman, "Tree visualization with tree-maps: A 2-d space-filling approach," Tech. Rep., 1998.
[50] W. Wang, H. Wang, G. Dai, and H. Wang, "Visualization of large hierarchical data by circle packing," in *Proceedings of the SIGCHI conference on Human Factors in computing systems*. ACM, 2006, pp. 517–520.
[51] M. Harrower and C. A. Brewer, "Colorbrewer. org: an online tool for selecting colour schemes for maps," *The Cartographic Journal*, vol. 40, no. 1, pp. 27–37, 2003.
[52] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965.
[53] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," in *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 2009, pp. 502–506.
[54] M. Ebrahimi, N. Sayed, M. Rashvand, and M. B. Tahoori, "Fault injection acceleration by architectural importance sampling," in *2015 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*. IEEE, 2015, pp. 212–219.
[55] S. K. S. Hari, S. V. Adve, H. Naeimi, and P. Ramachandran, "Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults," in *ACM SIGPLAN Notices*, vol. 47, no. 4. ACM, 2012, pp. 123–134.
[56] H. Menon, M. O. Lam, D. Osei-Kuffuor, M. Schordan, S. Lloyd, K. Mohror, and J. Hittinger, "Adapt: algorithmic differentiation applied to floating-point precision tuning," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE Press, 2018, p. 48.
[57] P. Lindstrom, "Fixed-rate compressed floating-point arrays," *IEEE transactions on visualization and computer graphics*, vol. 20, no. 12, pp. 2674–2683, 2014.



Zhimin Li is a graduate student working on his PhD from the School of Computing at the University of Utah. Zhimin has been a research assistant at the University of Utah's Scientific Computing and Imaging Institute since 2016. He received his B.S. in computer science and mathematics from the University of Utah in 2016. His research focuses on analyzing silent data corruption in high performance system.

Harshitha Menon is a Computer Scientist in the Center for Applied Scientific Computing (CASC) at Lawrence Livermore National Laboratory. She joined CASC as a postdoctoral research staff in 2016. Her research focuses on floating-point mixed-precision, approximate computing, and fault tolerance of HPC applications. She received her Ph.D. in 2016 and M.S. in 2012, both from the University of Illinois at Urbana-Champaign. She was awarded the ACM/IEEE-CS George Michael Fellowship in 2014, the Anita Borg Scholarship in 2014 and the Siebel Scholarship in 2012.

Dan Maljovec is a recent graduate from the School of Computing at the University of Utah where he focused on the application of topological models to high dimensional data. He received his B.S. in computer science from Gannon University in 2009. Dan has been a research assistant at the University of Utah's Scientific Computing and Imaging Institute since 2012 and has worked at three separate national laboratories during that time. He is now a full-time employee at Recursion Pharmaceuticals in Salt Lake City.

Yarden Livnat is a Research Scientist at the SCI Institute and an Adjunct Research Professor at the School of Medicine at University of Utah. He received a B.Sc. in Computer Science at the Ben Gurion University in Israel, an M.Sc. in Computer Science at the Hebrew University in Israel and a Ph.D. in Computer Science at University of Utah in 1999.

Shusen Liu is a computer scientist at the Center for Applied Scientific Computing (CASC) at the Lawrence Livermore National Laboratory (LLNL). His research interests lie primarily in high-dimensional data visualization and interpretable machine learning. He received a Ph.D. in computing from the University of Utah in 2017.

Kathryn Morhor is the group leader for the Data Analysis Group at the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory (LLNL). Kathryn's research on high-end computing systems is currently focused on scalable fault tolerant computing and I/O for extreme scale systems. Her other research interests include scalable performance analysis and tuning, and parallel programming paradigms. Kathryn is a 2019 recipient of the DOE Early Career Award whose research focuses primarily on user-level file systems for HPC in the Unify project and on scalable I/O with the Scalable Checkpoint/Restart Library (SCR), an RD100 Award-winning multilevel checkpointing library. She is also a Co-Chair of the Administrative Steering Committee for PMIx, a portable interface for tools and applications to interact with system management software.

Peer-Timo Bremer is a member of technical staff and project leader at the Center for Applied Scientific Computing (CASC) at the Lawrence Livermore National Laboratory (LLNL) and Associated Director for Research at the Center for Extreme Data Management, Analysis, and Visualization at the University of Utah. Prior to his tenure at CASC, he earned a Ph.D. in Computer science at the University of California, Davis in 2004 and a Diploma in Mathematics and Computer Science from the Leibniz University in Hannover, Germany in 2000.

Valerio Pascucci the Inaugural John R. Parks Endowed Chair of the University of Utah and the founding Director of the Center for Extreme Data Management Analysis and Visualization (CEDMAV) of the University of Utah. Valerio is also a Faculty of the Scientific Computing and Imaging Institute, a Professor of the School of Computing, University of Utah, and a Laboratory Fellow, of PNNL and a visiting professor in KAUST. Before joining the University of Utah, Valerio was the Data Analysis Group Leader of the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory, and an Adjunct Professor of Computer Science at the University of California Davis. Valerio's research interests include Big Data management and analytics, progressive multi-resolution techniques in scientific visualization, discrete topology, geometric compression, computer graphics, computational geometry, geometric programming, and solid modeling. Valerio is the coauthor of more than two hundred refereed journal and conference papers and is an Associate Editor of the IEEE Transactions on Visualization and Computer Graphics.