A Visual Comparison of Silent Error Propagation

Zhimin Li, Harshitha Menon, Kathryn Mohror, *Member, IEEE,* Shusen Liu, Luanzheng Guo, Peer-Timo Bremer, *Member, IEEE,* Valerio Pascucci, *Member, IEEE*

Abstract—High-performance computing (HPC) systems play a critical role in facilitating scientific discoveries. Their scale and complexity (e.g., the number of computational units and software stack) continue to grow as new systems are expected to process increasingly more data and reduce computing time. However, with more processing elements, the probability that these systems will experience a random bit-flip error that corrupts a program's output also increases, which is often recognized as silent data corruption. Analyzing the resiliency of HPC applications in extreme-scale computing to silent data corruption is crucial but difficult. An HPC application often contains a large number of computation units that need to be tested, and error propagation caused by error corruption is complex and difficult to interpret. To accommodate this challenge, we propose an interactive visualization system that helps HPC researchers understand the resiliency of HPC applications and compare their error propagation. Our system models an application's error propagation to study a program's resiliency by constructing and visualizing its fault tolerance boundary. Coordinating with multiple interactive designs, our system enables domain experts to efficiently explore the complicated spatial and temporal correlation between error propagations. At the end, the system integrated a nonmonotonic error propagation analysis with an adjustable graph propagation visualization to help domain experts examine the details of error propagation and answer such questions as why an error is mitigated or amplified by program execution.

Index Terms—Fault Tolerance Boundary, Information Visualization, Graph Visualization, Error Propagation, Silent Data Corruption

1 Introduction

High-performance computation (HPC) systems are critical for advancing science. The demand for higher computation speed and larger data processing will increase the scale of such systems in the near future. However, this scale will make the systems vulnerable to different types of errors. One of the most dangerous errors is the *soft error* [1], which is caused by device noise, low voltage, or cosmic radiation. A soft error is a temporal error that affects computation for only a short period of time, causes a random bit-flip event during a program's computation, and introduces error into the application's computation. Such an event is often recognized as silent data corruption (SDC), in which a program's execution is corrupted and generates an incorrect computation result without notification. Recently, this computation concern has attracted the attention of the HPC community [2], [3], [4] and industry [5] [0].

Understanding the influence of silent data corruption on HPC programs is critical for designing efficient solutions to improve the resiliency of these programs. However, analyzing SDC can be a difficult task because of the number of variables that can be corrupted. The classical solution [6], [7] to study a program's resiliency to SDC is through fault-injection experiments, in which a tool injects an error (e.g., flips a single bit of a variable) during an application's execution and observes the impact of the error on the program's output. One drawback of this approach is that obtaining a full resiliency profile of an application can require a large amount of computation resources [8], and it costs too much time to be practical. Therefore, some researchers have tried to study the resiliency of a program through error propagation [9], [10].

Previously, researchers [11], [12] have demonstrated that using error propagation can significantly reduce the number of fault injection experiments to understand a program's resiliency. Compared with the classical solution, error propagation analysis is able cover more computation components in a single fault injection experiment. Furthermore, understanding the propagation behavior of the errors that result in SDC or are mitigated during computation can provide valuable information for reasoning about the vulnerability of computation units and designing efficient protection or recovery mechanisms (e.g., which checkpoint the HPC application will roll back for recovering once an error is detected). However, most works [13], [14], [15] only design solutions to detect SDC and improve a program's resiliency with a certain heuristic without exploring and understanding these complex error propagation behaviors.

Currently, understanding the error propagation process of a program is still a challenging task. A program's error propagation process often involves a large amount of intermediate variables. Observing error propagation [16] through these variables is tedious and often misses critical information. Meanwhile, how these variables influence each other and lead to different corruption outcomes can be too complex to understand. For example, why does one error corruption lead to error explosion during program computation but another is mitigated? Does an error corrupt the same variable of a program but called by program at a different time may share the same propagation behavior?

- *Zhimin Li, and Valerio Pascucci are with the Scientific Computing and Imaging Institute, University of Utah.E-mail: {zhimin, pascucci}@sci.utah.edu*
- *Harshitha Menon, Shusen Liu, Kathryn Mohror, and Peer-Timo Bremer are with Lawrence Livermore National Laboratory. E-mail: {gopalakrishn1, liu42, mohror1, bremer5}@llnl.gov*
- *Luanzheng Guo is with Pacific Northwest National Laboratory. E-mail: {lenny.guo}@pnnl.gov*

0. The Universe is Hostile to Computers https://www.youtube.com/watch?v=AaZ_RSt0KP8&ab_channel=Veritasium

This article has been accepted for publication in IEEE Transactions on Visualization and Computer Graphics. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2022.3230636

Fig. 1. View (1) on the left shows a fault tolerance boundary visualization of the conjugate gradient algorithm. An execution interval is selected in the boundary view, and the related bit-flip outcome over each bit in the interval is displayed in (3). The error propagation that starts from the selected interval is highlighted in view (2), and similar error propagations are clustered together. Views (4),(5), and (6) on the right coordinate with each other to demonstrate a nonmonotonic error propagation case in which a large error is injected in the middle of the program but mitigated.

To address the above challenges, we design an interactive visualization system to help HPC researchers understand a program's error propagation and study its resiliency. The system engages a program's fault tolerance boundary [11], which gives the maximum error that each variable can tolerate without causing silent data corruption. Coordinating with the boundary visualization, it displays a summarized resiliency profile of a program and an overview of error propagation similarity to jointly study a program's resiliency. For a specific error corruption experiment, we propose a nonmonotonic inference method to locate error mitigation and amplification during propagation and design a graph visualization to highlight the error propagation process. We summarize our main contributions as follows:

- A new interactive visualization system to study HPC applications' resiliency to silent error corruption and propagation (section 6).
- A novel visual design that reveals the complex spatial and temporal correlation between different error propagation (section 6.1).
- An adjustable graph design that is integrated with a nonmonotonic error analysis model to study error propagation and identify error mitigation and amplification (section 5, 6.5).
- Three use cases and two examples of domain feedback to evaluate the usability of the visualization system (section 7).

2 Related Work

In this section, we discuss the relevant literature and compare our work with current state-of-the art studies in SDC analysis.

2.1 Fault Tolerance Analysis

The HPC community has been dedicated to addressing the challenge of silent data corruption for decades [13], [17], [18], [19]. The classical approach for studying the problem is to use a fault injection campaign, which repeatedly tests an application over different locations to get a resiliency statistical profile [20]. This approach requires numerous fault injection tests to achieve full coverage of an application, and that translates to a large amount of computation resources and time. Many researchers [8], [21] have tried to reduce the number of fault injection experiments by using methods such as instruction clustering or selecting representative instructions to approximate a program's resiliency. This approach can significantly reduce the number of tests but sacrifice the accuracy of the resiliency measurement.

An alternative approach uses static and dynamic program analysis to predict a dynamic instruction's resiliency [22], [23]. The approach needs only a few fault injection experiments with the data dependency graph and control flow graph to approximate the program's resiliency. However, achieving high accuracy of the approximation result using this approach is difficult. Researchers have also tried to design algorithms to auto-detect the occurrence of an SDC event during program computation [14]. Berrocal et al. [15] designed an auto-detection method based on temporal information. Huang and Jacob [24] developed an error-correcting matrix multiplication by adding an additional column and row in the matrix for verification. Di et al. [25] suggested a solution to auto-detect the occurrence of soft error based on the assumption that nearby computation

elements have a natural correlation with each other and their value ranges are within a certain threshold. These approaches are often application-specific and cannot apply to general programs. Meanwhile, all the above approaches try to reduce or detect the SDC but few try to understand the error propagation after an error corruption. To fill this gap, we have designed an interactive visualization to reveal the dynamic of the inner state behavior of error-corrupted programs and how the program's variables interact with each other. Such information improves domain experts' understanding of an error-corrupted program and helps them design efficient solutions for better detection and protection.

2.2 Performance Visualization

Due to the complexity and large scale of the data log of an HPC computation, many visualization techniques have been proposed to debug and improve the performance of HPC systems. Guo et al. [26] designed the LaVALSE visualization system to analyze the state log information of the supercomputer Mira. LaVALSE is targeted to help HPC researchers debug the potential source of a failure event in supercomputers but is not specific to a silent data corruption event. Wongsuphasawat et al. [27] designed a visualization to show the data flow of the neural network computation model, which helps machine learning engineers understand and debug the neural network model. Xie et al. [28] proposed stack2vec, a context-based approach to learning a vector representation for a call stack of an HPC application, and used an active learning approach to identify the potential anomalous function executions. Significant efforts in this field have contributed to visualizing the call paths and computation logs of large parallel systems. For more detail, refer to the survey by Isaacs et al. [29].

However, none of the above work has focused on the impact of silent data corruption in the HPC system specifically. In previous research, Li et al. [16] designed a visualization system to present the impact of every bit-flip error in a single view and visualize the propagation of the error without variable-dependency information. Observing the error propagation is time consuming and tedious for domain experts. They need to spend a great deal of time exploring the data and figuring out which fault injection experiments are informative and which potential components can mitigate or amplify error. In comparison, our system coordinates multiple views to compare error propagations from different experiments. The system helps domain experts track down useful fault injection experiments and provides a nonmonotonic error propagation model to locate the computation units that have the properties to mitigate or amplify error.

2.3 Graph Visual Encoding

Visualizing a program's data dependency graph is critical for studying its error propagation. However, visualizing the dynamic of an error propagating through the data dependency graph intuitively is still an on-going challenge. Two classical approaches to visualize graph data are the node-link diagram and adjacency matrices. The authors of previous studies [30], [31] have compared user performance on these visual encodings. Their studies have found that the node-link diagram is better to visualize small-scale and sparse data. The adjacency matrix outperforms the node-link diagram with a dense graph having more than 20 nodes in a few basic user tasks. However, the adjacency matrix demonstrates poor performance with path tracking, which makes tracking the path between nodes in the graph difficult. Different innovative graph encodings have been proposed to address the scale challenge of visualizing large-scale gene graph data, such as BioFabric [32] and Quilts [33]. More visual encodings for graph data can be found in the surveys in [34], [35]. In comparison, we have designed a hybrid graph visual encoding based on the adjacency matrix and node-link diagram to visualize a program's dependency graph. It keeps the semantics of a program's execution and improves the path tracking difficulty in the adjacency matrix visualization.

2.4 Dimension Reduction and Time Series Analysis

Each error propagation dataset records the error in each variable during program execution, and error propagation data can be considered as a multivariate time series. Time series comparison and visualization [36], [37] is an important topic. Van Wijk and Van Selow [38] proposed a cluster and calendar-based visualization that analyzes univariate time series data. Ruta et al. [39] designed a visualization tool that enables users to explore large time series data collections from a global scale to a single observation. Anna et al. [40] compared the line and color encoding of a time series visualization for a similarity search task. Because of the limited screen pixels, visualizing a whole time series with millions of time steps is difficult. Many algorithms [41], [42], [43] have proposed to reduce the amount of information for visualization. To visualize the fault tolerance boundary and an overview of error propagation detail, we use the a dimension reduction technique M4 [44], which uses four selected points of a selected subinterval of the time series data to present the selected interval with only a minor loss of information.

Instead of comparing multiple pairs of time series one by one, we perform the dimension reduction approach that projects all error propagations into a 2D view and coordinates it with the temporal and spatial information to analyze these error propagations. The common dimension reduction techniques include PCA [45] and MDS [46], which preserve the maximum variance or distance information. More techniques can be found in the survey [47]. In this work, we focus on the dimension reduction techniques that preserve the neighbor information, such as T-SNE [48] and UMAP [49].

3 Background

Before getting into the detail of the system, we first introduce the necessary background knowledge and key terminologies used in our study. The workflow overview of this study is summarized in Figure 2.

3.1 Soft Errors

Soft errors are temporal errors that affect a program's computation for only a short period of time. Soft errors are

This article has been accepted for publication in IEEE Transactions on Visualization and Computer Graphics. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2022.3230636

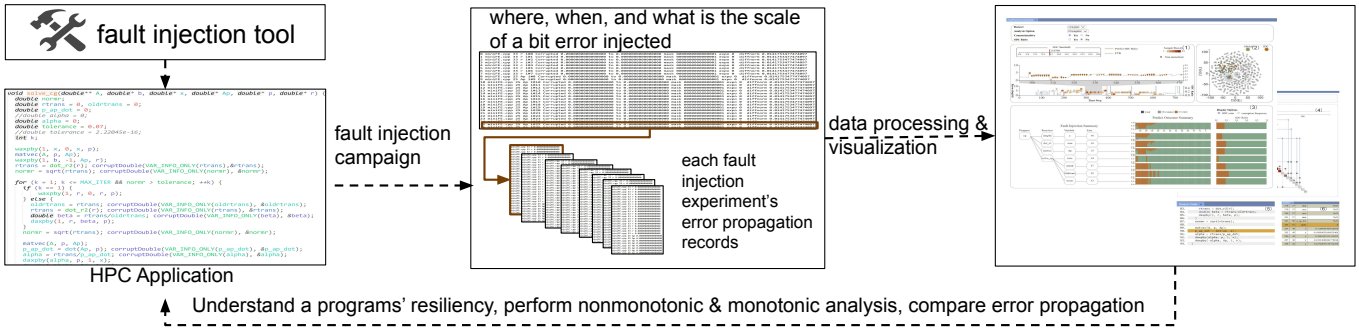


Fig. 2. A workflow of the designed visualization to analyze an HPC application's resiliency. Domain experts use the fault injection tool to perform a fault injection campaign, and collect fault injection data and error propagation data. The design system processes the data and visualizes the results. At the end, domain experts can use the visualization result to perform a program's resiliency analysis and explore efficient application protection mechanisms.

caused by cosmic radiation and device noise, and their occurrence often leads to bit-flip errors in storage, data transmission, and compute units. These bit-flip errors can bypass the hardware protection mechanism, further affecting the application state, and finally corrupting the application outcome. Soft errors can occur in an unpredictable manner and influence an application in many different ways. One of the ways in which they can affect a program is by corrupting pointer and control variables. This kind of corruption often shows up as a program crash, which is easy to detect and can be addressed by rolling back the application to a previous checkpoint state. Soft errors might be masked by the hardware or the application, in which case they do not affect the program's output. The most challenging case is when the soft error occurs in a data variable without obvious symptoms and the error propagates to the final output.

To clarify concepts, Fig. 3 shows a bit-flip error corrupting variable *d* and changing its value without warning. Under this condition, the program continues to execute and produce an incorrect final output *pi*. To verify whether the final output is acceptable despite the errors introduced, we use a **SDC threshold** (e.g., 0.00001), which is often defined by domain experts. If the error in the final output is within the threshold, we consider that the error is masked. However, if the final output error is greater than the threshold, then it is considered as SDC. We summarize the three different outcomes of soft errors below.

- **Silent Data Corruption (SDC)**: The fault injected application execution produces a different outcome from the outcome of fault free runs. Further, the execution does not pass the result verification phase.
- **Masked**: The application execution outcome is found to be the same as the outcome of fault free runs. It can be different, but the fault injected application execution passes the result verification phase.
- **Interruption (Crash)**: The fault-injected application execution does not make it to the end, but is interrupted in the middle of the execution.

3.2 Fault Model

In our analysis, we consider soft errors that occur in registers, logic circuits, and data transmission. We do not consider soft errors found in system memory components, such as on-chip cache, because those memory architectures are typically protected by error correcting code (ECC) or parity bit from the architecture level. These assumptions are common in the current fault tolerance analysis literature [50], [51], [52], [53]. We use the most common single bit-flip error model [8], [52], [54], [55] instead of the multi-bit-flip error model as multi bit-flips are highly unlikely in HPC systems. In a single bit-flip model, a fault injection tool introduces a single-bit flip in a variable. In our study, we perform the fault-injection experiment on variables at the source-code level to provide insights and analysis, which is the most suitable abstraction for designing resilience techniques.

An example of a fault injection is shown in Fig. 3, where a single bit of variable *d* is flipped, which results in an error that gets propagated to subsequent computations. The bottom line plot displays the error in each tracked variable over time. The initial error happens in variable *d*, and it propagates to *pi* and *d* in the next iteration and continues until the end of the execution. In this case, the **error propagation data** can be considered as multidimensional time series data with 99 elements.

In this study, we conduct a fault injection campaign to collect the error propagation information of a program. A **fault injection campaign** is a collection of fault injection runs where a fault is injected at a randomly chosen location, called the fault injection site, for each run. An **exhaustive fault injection campaign** is a collection of fault injection experiments that flip every bit of a program's fault injection sites. In the case of Fig. 3, each of three variables over different times can be considered as a fault injection site. The exhaustive fault injection campaign will test all three variables over the entire execution.

We use the **SDC ratio** to quantify the overall resiliency of the program. The SDC ratio is defined as $\frac{n_{sdc}}{N}$, in which n_{sdc} is the number of times fault injection runs lead to SDC, and N is the total number fault injection runs. The value range of SDC ratio is between 1 and 0. The value close to zero means robust, and close to one vulnerable.

3.3 Data Dependency Graph

An error that corrupts a variable of a computation will propagate to subsequent computations that depend on the variable. In this study, we also track the dependency of critical data variables of a program to study error propagation.

This article has been accepted for publication in IEEE Transactions on Visualization and Computer Graphics. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2022.3230636

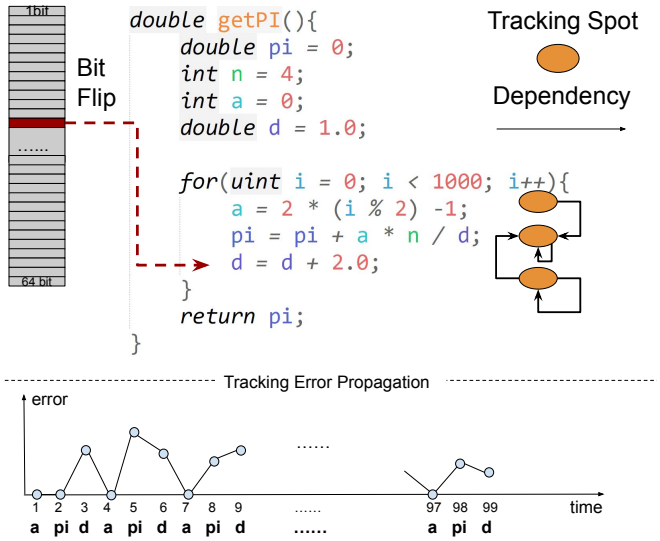


Fig. 3. A simple program for calculating Pi to demonstrate the process of fault injection and error propagation tracking. The fault injection experiment flips a bit of variable *d* of function *getPI* and tracks the function's three critical data variables *a*, *pi*, and *d* to understand how errors propagate through a program's computation. The bottom plot displays a fault injection experiment's error in variables *a*, *pi*, and *d* over time.

For example, in Fig. 3, we track the value of three data variables in the *getPI* function to understand the error propagation process. For example, variable *a* does not depend on variables *pi* and *d*, and an error that corrupts either variable will not affect variable *a* in the subsequent computation. However, *pi* depends on itself, *a*, and *d*. An error corrupting any of them will propagate to *pi*, which leads to final output corruption. A program's dynamic data dependency graph is extracted by using an LLVM tool during the program execution. Extracting an accurate dynamic program dependency graph is not a trivial task. The dependency graph may be incomplete as it might be input-dependent; therefore, the domain expert will still need to be involved in the analysis process to fix the dependency graph if some dependencies are found to be missing.

3.4 Fault Tolerance Boundary

Previous work [11] has defined the fault tolerance boundary of a program as a set that consists of the maximum error that can be tolerated at each program variable. The formal definition is given in **Definition 1**. It is guaranteed that a threshold, δ, exists for each variable of a program. The worst case is that $\delta = 0$, and the variable is sensitive to any level of perturbation (e.g., a pointer variable).

Definition 1. *For a variable of a program at a certain time, a maximum amount of perturbation $\delta \in R^+$ exists, such that with $\forall e \in [-\delta, \delta]$ error in the variable, the program still generates an acceptable output. The fault tolerance boundary is a set that has the δ value of each fault injection site of a program.*

Fig. 4 displays a synthetic fault tolerance boundary of the *getPI* function. The red and green dots are a set of fault injection experiments from a fault injection campaign. The injected error above the threshold will be predicted as SDC and below as Masked. Researchers can use the fault

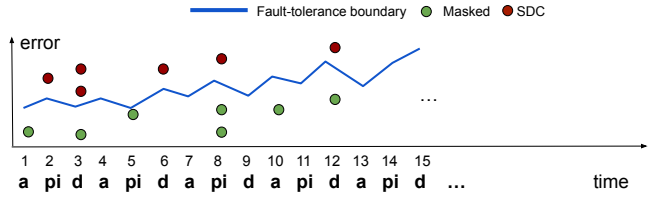


Fig. 4. A synthetic fault tolerance boundary of the *getPI* function. A fault injection experiment with injected error above the fault tolerance boundary will be predicted as SDC, and below will be predicted as Masked.

tolerance boundary to study a program's resiliency to bit-flip errors. The threshold value in each fault injection site reveals its SDC ratio. For example, a double type variable has 64 bits, and if the value of the variable is known, then all possible 64 bit flip errors can also be calculated through the floating point representation.

Calculating a program's fault tolerance boundary is expensive. The brute-force approach needs to test a program's unit many times to find the fault tolerance threshold value. Considering the number of units that need to be tested in a program, calculating a program's fault tolerance boundary is difficult. In this work, we use an **error propagation method (EPM)** [11] to approximate a program's fault tolerance boundary. EMP specifically selects the fault injection experiments with the masked outcome to approximate a program's fault tolerance boundary. For selected experiments, the method tracks the maximum error that propagates through each variable. Once the algorithm goes through all available experiments, it will output the tracked maximum value in each variable as the fault tolerance threshold value. This approach can save up to several orders of magnitude samples to understand a program's resiliency, and the information revealed by the boundary can be used to directly calculate the SDC ratio of each component of a program.

4 Domain Task

It is a common challenge in the HPC domain that having a complete testing covering all units of a program will require a significant amount of computation resources. For example, an exhaustive fault injection campaign, which covers all critical data variables, on the conjugate gradient algorithm with a 200x200 input matrix will take a week on a 16G memory and Intel i7 machine. Instead, domain experts tell us that it is typical for them to test only 1% or less of locations and generate a statistical summary resiliency profile based on that. However, the locations that are not covered by these tests will not have any resiliency information. Under this condition, any resiliency feedback on these regions is valuable.

Meanwhile, the resiliency requirement of applications changes in different applications. For example, the finance application has a much higher standard for a program's resiliency. An application often shows more resilience if domain experts are able to accept a small flaw in the final outcome. On the opposite side, if the final output requires high output quality, then such an application needs more

This article has been accepted for publication in IEEE Transactions on Visualization and Computer Graphics. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2022.3230636

protection. Revealing the resiliency profile update with different error thresholds will help domain experts to properly consider SDC's impact in different application scenarios and to design a customized strategy to improve a program's resiliency.

The other key driving force of this new platform is the difficulty of targeting informative computation units that mitigate or amplify error. Meanwhile, comparing the similarity and difference of error propagation can improve domain experts' understanding of a program's resiliency and give insight into a program's corrupted behaviors. During our separate discussions with three domain experts, we found out that none of them could definitively answer how to identify an experiment that has valuable propagation information. They suggested that instead of analyzing fault injection experiments one by one, having a visualization system that highlights important propagation information can save them considerable time. One potential solution is to compare the similarity of error propagations starting from different computation units and execution times. Also, a program execution involves a large number of intermediate variables; therefore, quickly locating a set of them that can mitigate or amplify error can speed up the error propagation analysis process. Overall, we summarized the following four domain tasks. These tasks are aimed at providing a better understanding of a program's resiliency and improving it by modifying the code, adding protection, or helping design a better protection strategy.

Task 1: Studying a program's resiliency with its fault tolerance boundary. Understanding the fault tolerance boundary that is approximated by these fault injection experiments can provide further understanding of a program's resiliency in addition to the overall statistical summary result. Displaying the relationship between the fault tolerance boundary and the fault tolerance requirement can give more information to improve a program's resiliency.

Task 2: Revealing the temporal and spatial correlation between error propagations. When and where an error is injected will lead to different error corruption behaviors of a program. This information reveals the diverse corrupted behaviors of a program.

Task 3: Locating computation units that mitigate or amplify error for error propagation analysis. Identifying which fault injection experiment has useful information for the domain experiment to understand the mitigation and the explosion behavior is not a trivial task. Showing the propagation process of these examples will provide useful insights for domain experts to design better protection strategies.

Task 4: Correlating source code and analysis result. The data-driven analysis often ends up with a source code's modification to improve a program's resiliency (e.g., instruction duplication). The domain experts often want to go back and forth between the source code and the analysis result to understand the resiliency profile and design strategies that improve the current program's resiliency.

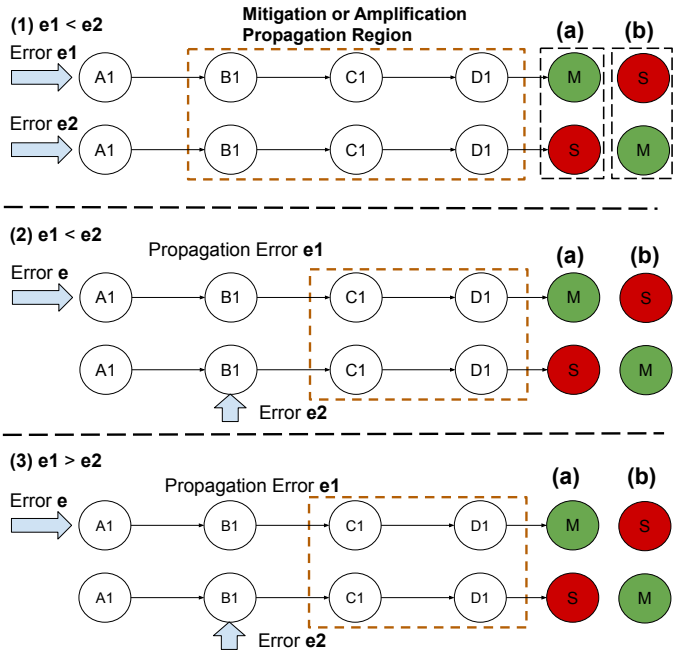


Fig. 5. Enumerating the relationships between a masked experiment and an SDC experiment to infer a program's mitigation and amplification behavior. This enumeration is based on an error monotonic assumption that a large error causes a worse outcome. In the above comparison, case (1) with outcome (b), case (2) with outcome (b), and case (3) with outcome (a) break the monotonic assumption. The experiments that break the monotonic assumption are interesting cases to understand a program's mitigation and amplification phenomena.

5 Monotonic and Nonmonotonic Corruption Analysis

Our regular discussions with domain experts revealed that they are generally interested in analyzing two categories of error propagation processes: error mitigation and error amplification. Error mitigation during propagation indicates that the program computation can eliminate error and recover from the error corruption. Learning from the underlying mechanism of a program that leads to such a behavior helps domain experts design robust HPC applications. Error amplification indicates that the program accelerates the corruption during execution. Locating the computation units that amplify the corrupt error and protecting them can improve a program's resiliency. The challenge in performing such an analysis comes from the complex computation logic of a computation algorithm and the large amount of tracking variables that need to be examined. For example, when and where error is mitigated or amplified is difficult to answer just by observing error propagation without additional assistance. In this section, we describe a method that models the error propagation process to help locate computation units that mitigate or amplify errors and speeds up the propagation analysis.

Before presenting a solution, we discuss an assumption called **error monotonicity**. This assumption can be formulated as *more errors often cause a worse outcome*. For the fault tolerance analysis of a program, this assumption can be rephrased as more errors in a program have a high probability of leading to a worse computation outcome.

The monotonic reaction to error leads to more interpretable program behavior after corruption, since the result always gets worse. Previous works have used the error monotonicity design protection [56] to improve neural network models' resiliency or reduce the number of experiments to understand a program's resiliency [11]. Here we integrate it into our system to speed up the propagation analysis. To explain the monotonic and nonmonotonic analysis in detail, let us look at two fault injection experiments with different amounts of error injection and compare their propagation processes. In Fig. 5, two fault injection experiments are compared for three scenarios.

In Fig. 5, case (1) has errors injected into the same location. For case (1) $e1 < e2$, errors are injected into component A1 and propagate to components B1, C1, D1. If the result is (a), the experiment with **e2** error leads to SDC, and the one with the **e1** error generates a masked outcome. This case follows the monotonic assumption that a location with a large error will cause a worse outcome. If the result is (b), a larger error **e1** leads to a masked outcome, and a smaller error **e2** results in a worse outcome, which is SDC. This case breaks the monotonic assumption, and the follow-up propagation has mitigated/amplified events when they propagate through B1, C1, and D1.

Whereas case (1) has errors injected at the same location, case (2) has errors injected at different locations. Here, we have two experiments. In one experiment, the error **e** is injected into component A1, and the error propagates to component B1, which causes **e1** error. The experiment's final outcome is Masked. The other experiment injects error into B1 with error **e2** and the final outcome is SDC. If the result is (a), it follows the monotonic assumption as $e1 < e2$, and a small error leads to a masked outcome, and a larger error leads to an SDC outcome. However, if the result is (b), A1 is corrupted with error e, and B1 has error **e1**, which leads to an SDC final outcome. In comparison, the other experiment has an error in B1 with **e2**, and A1 is error free but the final experiment's final outcome is masked, which breaks the monotonic assumption. The follow-up propagation process has mitigated and amplified events when errors propagate through C1, D1.

Case (3) also has errors injected into different locations. Case (3) is similar to case (2), but the initial injected error is different where $e1 > e2$. If the final result is (a). This experiment breaks the monotonic assumption, in which a larger error leads to a Masked outcome and a smaller error leads to an SDC outcome. On the other hand, if the final result is (b), then the experiment follows the monotonic assumption.

6 System Design

The designed system coordinates six views to address tasks discussed in the previous section: 1) fault tolerance boundary view, 2) bit-flip summary view, 3) propagation similarity view, 4) propagation tracking view, 5) data log view, and 6) source code view.

The source code (Fig. 1 (5)) and data log view (Fig. 1 (6)) have a relatively simple and straightforward design, as they mostly complement other views for inspecting the correlation to the source code or tracking log **(Task 4)**. Here, we focus the discussion on the design of the other four views.

TABLE 1
Composition of visual components to address each domain task.

Visual Components\Tasks	T1	T2	T3	T4
Fault Tolerance Boundary View	✓	✓	✓	
Bit-Flip Summary View	✓	✓		
Propagation Similarity Summary View		✓	✓	
Error Propagation Tracking View			✓	
Source Code View			✓	✓
Data Log View			✓	✓

6.1 Overview, Interaction, and Tasks

Before getting into each individual view, we discuss the workflow overview of the visualization system, which is briefly summarized in Fig. 6. Users can explore the fault tolerance boundary view and bit-flip summary view to understand a program's resiliency (Fig. 6 ⓣ1). It helps to answer questions such as whether a computation variable appearing during a different time of program computation has a similar resiliency profile or whether the variables called by the program nearby during the execution share a similar resiliency.

Meanwhile, users can coordinate the fault tolerance boundary view, bit-flip summary view, and propagation similarity view to study propagations' spatial and temporal correlations (Fig. 6 ⓣ2). In Fig. 6, a user selects a subset of experiments in the propagation similarity view (Fig. 6 (a)), which shares similar propagation patterns. The fault tolerance boundary view highlights the corresponding temporal regions (Fig. 6 (b)) in which errors are injected. The bit-flip summary view clearly shows which components these errors are injected into and which bit is flipped (Fig. 6 (c)). Such a design can help answer questions such as whether the error injected into the same variable will have similar propagation, whether similar propagation experiments start from the same computation units, and whether error propagations starting from the nearby instructions during the computation are similar. The similar selection operation can be performed in the fault tolerance boundary view, in which users can select a sub-time interval, and the relative error propagation experiments will update in the propagation view and bit-flip summary view. The same operation can also be performed in the bit-flip summary view, in which users can select a component, and the relative temporal information and propagation similarity will be presented in the boundary view and propagation view.

After the exploration, users can select a nonmonotonic sample from the fault tolerance boundary view or an interesting error propagation experiment from the propagation similarity view to examine the detail error propagation (Fig. 6 ⓣ3). The propagation process can be explored in the error propagation tracking view. The last piece of the task is the source code and data correlation (Fig. 6 ⓣ4, which is used to support the previous task. Overall, we summarize how each task is addressed by corresponding designs in Table 1.

This article has been accepted for publication in IEEE Transactions on Visualization and Computer Graphics. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2022.3230636

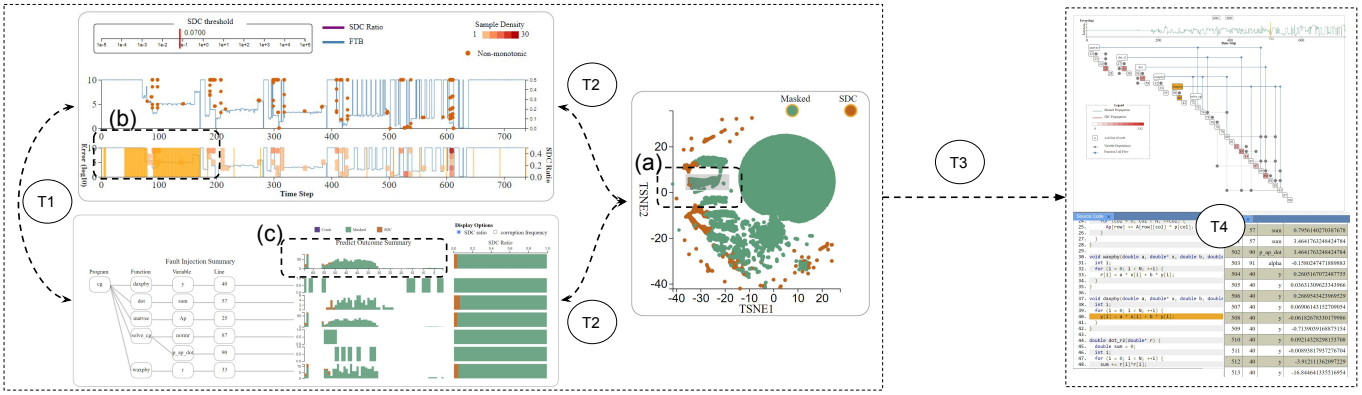


Fig. 6. A subset of samples, which share a similar propagation pattern, are selected in the propagation similarity summary view (a). The corresponding fault injection regions of these error propagation experiments are highlighted in the fault tolerance boundary (b) and indicate when these errors are injected. Meanwhile, the bit-flip summary view shows where these error are injected and which bits are flipped to cause these propagations.

6.2 Fault Tolerance Boundary View

In fault tolerance analysis, domain experts often start the analysis with an overview of a program's resiliency. The fault tolerance boundary view (Fig. 7) is designed as a temporal data visualization that presents domain experts an overview of a program's resiliency and highlights non-monotonic error corruptions. The temporal data visualization component consists of two timeline panels to present a program's fault tolerance boundary (**Task 1**). The bottom panel (Fig. 7 ⓒ) shows the entire boundary (blue color) and SDC ratio (purple color), whereas the top panel shows a zoomed view (Fig. 7 ⓑ) of the selected time duration in the global timeline. Users can brush the bottom panel and select a time interval that will be zoomed in for detail analysis. It reveals a program's resiliency over execution, and indicates execution intervals that are robust or vulnerable. This feedback can help domain experts examine these intervals and perform further analysis.

In the zoomed boundary visualization (Fig. 7 ⓑ), we use red circles to highlight the fault injection experiments, in which nonmonotonic corruption propagation occurs. Each highlighted sample is an error propagation case that breaks the monotonic assumption in section 5. The location of a sample indicates where an error corruption happens and the value scale of a corruption error. Each sample is useful for understanding the error amplification or mitigation during error propagation (**Task 3**) based on the discussion in section 5. Because the highlighted samples in the bottom boundary view (Fig. 7 ⓒ) can be dense and overlap with each other, we use a heat-map with the color bar (Fig. 7 ⓓ) to display the nonmonotonic samples density in different regions.

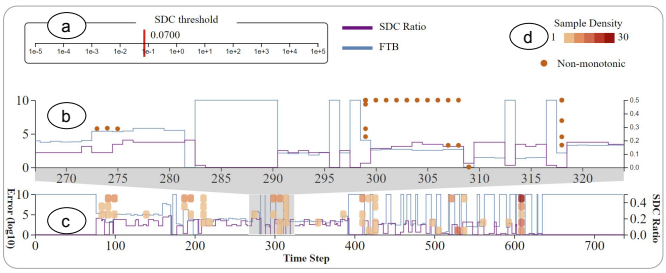


Fig. 7. A fault tolerance boundary visualization presents an overview of a program's fault tolerance boundary and SDC ratio, and highlights nonmonotonic error corruption cases.

Scaling is a common challenge in the fault tolerance analysis, and the scale capability in the current visualization is particularly important when analyzing the error boundary of a program with large amounts of intermediate variables. Our two-layer boundary visualization design can increase scalability and flexibility, i.e., it enables users to examine the global trend as well as to investigate localized concerns. For the bottom boundary and SDC ratio visualization, we apply the dimensionality reduction technique [44] to reduce the number of elements of the fault tolerance boundary.

An important task that the domain experts are interested in is to adjust the SDC threshold of a program and understand how the SDC ratio and fault tolerance boundary change. Interactively adjusting the SDC threshold (Fig. 7 a) will update a program's fault tolerance boundary and SDC ratio, and provide domain experts a comprehensive understanding of a program's resiliency under different fault tolerance requirements. As Fig. 8 shows, a user can adjust the threshold to the maximum and minimum values to locate the most robust or most vulnerable code region. The threshold is set to the maximum (e.g., 100000) to tolerate a large error, but the location (B) still generates an SDC outcome that indicates these locations are vulnerable to a bit-flip corruption. Once a domain expert adjusts the threshold

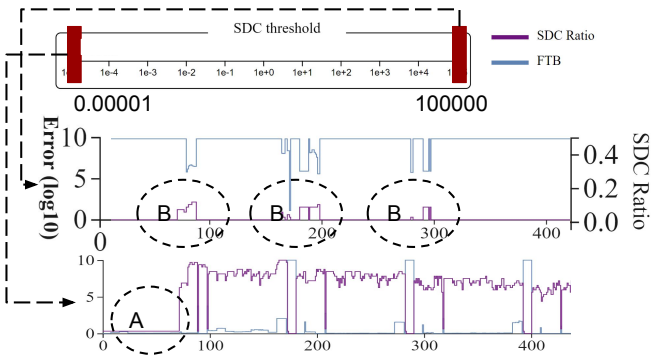


Fig. 8. Users can manually adjust a program's SDC threshold and check the SDC ratio and the fault tolerance boundary to locate spots that are robust or vulnerable to soft errors. With the highest threshold value (10000), region B still has SDC output, which indicates that these regions are vulnerable to soft error. With the lowest SDC threshold (0.00001), the A regions that have the zero SDC ratio are robust to soft error.

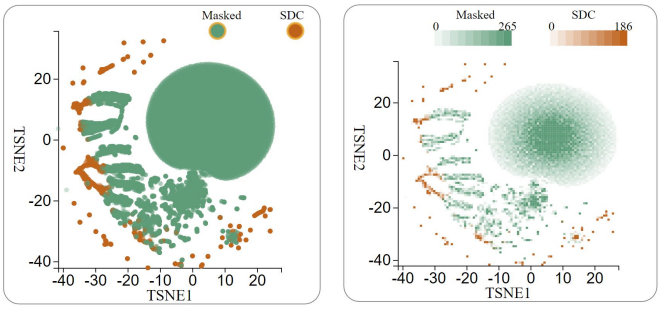


Fig. 9. A propagation summary view displays a 2D t-SNE projection, in which similar error corruption propagation experiments are nearby. In this dataset, the error propagation experiments that lead to SDC outcome are different from the majority of the experiments that end up with Masked outcome. A density heat-map is an option to address the data-scaling problem.

to the minimum value (e.g., 0.00001), but region (A) does not generate any SDC outcome indicating that it is robust to the bit flip corruption. A bit flip in these regions will not cause the final output quality to change significantly. Both cases A and B are interesting regions to understand how an error is amplified or mitigated by the program. An error propagation starting from region A helps clarify why an error is mitigated by the program since most bit-flips here will not lead to an SDC outcome. An error propagation starting from region B helps domain experts understand why an error will explode since a bit flip corrupting this region can lead to a large final error.

6.3 Propagation Similarity View

Each fault injection experiment has its relevant error propagation, and studying the similarity of error propagation is valuable to understand a program's behavior after error corruption. The propagation similarity view is designed to accomplish such a purpose. A propagation similarity visualization (Fig. 9) is a 2D t-SNE dimension reduction visualization that displays corruption experiments' propagation similarities (**Task 2**) in a single view. t-SNE is a dimension reduction algorithm [48] that projects high-dimensional data into a two-dimensional space, and the resulting projection has the feature that a similar propagation experiment will be projected to nearby locations. With the design visualization, domain experts can analyze whether an error that corrupts different variables of a program will affect a program's computation in similar or different ways. Users can also brush a rectangle to select these samples' properties. A point cluster in the visualization indicates that these error propagation experiments share the same propagation pattern. In Fig. 9, the largest point cluster is the fault injection experiments in which a bit-flip error corrupts a low mantissa bit of the variable, and does not cause significant error propagation during computation.

Scaling is also a concern in this visualization since the number of fault injection experiments can reach to millions or billions of experiments. To accommodate this concern, the system provides a heat-map option to display the sample density with different corruption outcomes. The number of variables in each fault injection experiment can also affect the scalability of this view. It takes a very long time for t-SNE algorithm to converge if the number of intermediate variables is thousands, and the number of total experiments is millions or more (e.g., fast Fourier transform in Section 7.1.1). To address this problem, the system performs principal component analysis to reduce the number of dimensions of each propagation experiment to hundreds of dimensions, before performing the t-SNE algorithm to generate the visualization.

6.4 Bit-Flip Summary View

The bit-flip summary view displays current available bit-flip experiments in a single view under different levels of granularities. It is a tree-based visualization that presents a statistical summary of a bit-flip's impact in a program. The view contains two components: a visual tree view that displays the hierarchical structure of a program and a statistical fault injection summary view of each program component. The tree hierarchy (Fig. 11 ⓐ) is organized based on the natural hierarchical structure of a program, which is a program, a program's function, a program's variable, and a specific line. Any of these component can be selected for detail analysis in the fault tolerance boundary view and propagation similarity view.

Each statistical fault injection summary view is a leaf of the tree (Fig. 11 ⓑ) that displays the fault injection summary of a program component. It contains two views. The left view is an IEEE floating-point base stacked bar chart that shows the ratio of different outcomes or the number of fault injection experiments over each bit **(Task 1, 2)**, and the right view is a summary of the corruption experiments' outcome ratio. Above the tree visualization, a visualization (*Fault Injection Summary*) displays a summary of all current selected fault injection experiments' outcome ratio. Domain experts can selectively collapse a tree node that aggregates its child nodes' data and presents a summary view of the tree node (e.g., function). Meanwhile, the collapse operation also helps mitigate the scaling challenge if the diagnosed application has a large amount of variables or functions.

6.5 Error Propagation Tracking View

After exploration, users need to choose a fault injection experiment to study its detail error propagation in the error

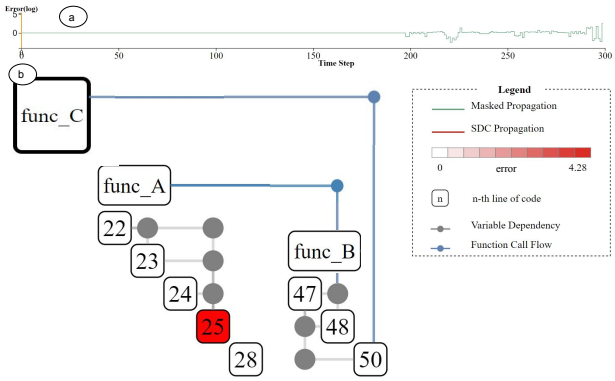


Fig. 10. The time series visualization (ⓐ) displays an overview of error in different time steps during propagation. A graph diagram (ⓑ) displays a program's function call flow and data dependency graph of each function. Each function is visualized as an independent matrix-link visualization and is connected with the function call flow. Each matrix-link visualization (func_C) can also be collapsed to reduce the number of elements presented in the visualization.

This article has been accepted for publication in IEEE Transactions on Visualization and Computer Graphics. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2022.3230636

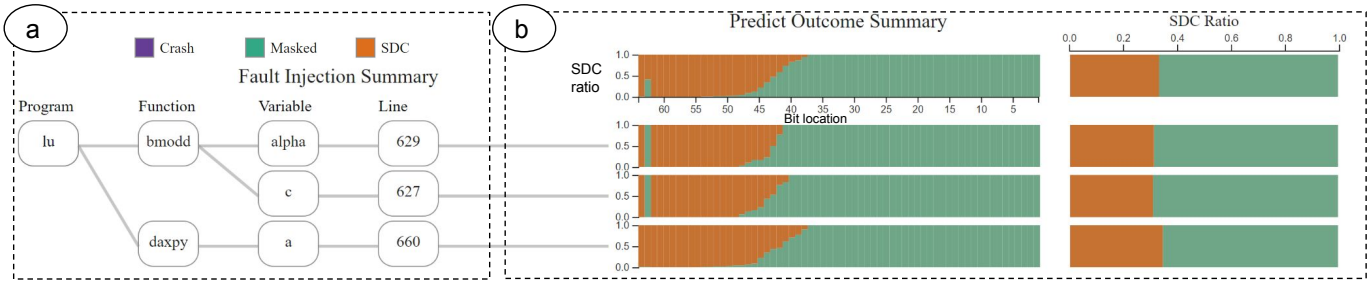


Fig. 11. A bit summary view is constructed as a tree structure ⓐ to display a program's components in a hierarchical manner, and the view ⓑ displays a statistical summary of each component's fault injection outcome. The visualization gives domain experts a clear understanding of the impact of each bit of a variable.

propagation tracking view. This view is a graph-based visualization that shows the dynamic of how an error corruption propagates through a program's critical data elements **(Task 3)** during execution. The view consists of two components: an overview time series visualization of the error of the different data variables at different time steps ((Fig. 10 ⓐ); and a graph visualization of a program's functions and their data dependency and call flow ((Fig. 10 ⓑ). These two components coordinate with each other to show the corrupted state of a program during error propagation.

The critical unit of the visualization is a graph visual encoding (matrix-link) (Fig. 10 func_B) that visualizes the main elements of a function computation. It encodes the execution order, the line number of critical data variables, and the data dependency. A matrix-link is a hybrid of a node-link diagram and adjacency matrix in which the diagonal rectangle represents the function line, and the circle between each rectangle indicates variable dependency. Red indicates the error scale in a specific line. The gray line represents the data dependency. Blue represents the function call and in this case *func_B line 48* calls *func_A*.

Previous works [30], [31] have performed user studies to compare the pros and cons between the adjacency matrix and the node-link diagram for graph visualization. A potential drawback of using the adjacency matrix is that it performs poorly at the path tracking tasks. To mitigate the difficulty of path tracking, the matrix-link diagram adds an additional orthogonal link path between two connected nodes to emphasize the dependency information. The other challenge of using an adjacency matrix to visualize the graph data is the order of nodes. Different orders reveal different data patterns [57] of a graph. However, in our context, the execution order is the default order to visualize a programs' dependency graph. Following the execution order is important to understand the meaning of the code and analyze the corruption propagation through it. Meanwhile, this default execution order is helpful for domain experts to identify the loop of the code, as the latter execution depends on the previous execution, which means a loop exists.

Because of the sparsity of a program's data dependency graph, we have also considered of using a node-link diagram to visualize this data dependency graph. However, the standard force-directed graph layout does not consider the execution order in the layout and will cause difficulties in tracking the error propagation. Our designed matrix-link visual encoding follows the execution order of a program, and is simple and easily implemented. We also perform a pilot user study to compare the performance among, the node-link diagram, adjacency matrix, and matrix-link visualization. The result can be found in supplementary material 1. To mitigate the scaling challenge with a large amount of tracking elements, the visualization also provides a collapse operation over each matrix-link (Figure 10 func_C) graph to reduce the number of displaying elements. Users also can adjust each function graph's location after the initial matrix layout to better customize graph view.

7 Evaluation

In this section, we discuss three use cases and domain experts' feedback to demonstrate the usability of our design system. We use the conjugate gradient and fast Fourier transform benchmarks to evaluate the usability of the system. The conjugate gradient has 50 thousand experiments and the total size of dataset is 1.5 Gigabyte. The fast Fourier transform benchmark has 1.1 million experiments and the total size of the dataset is around 200Gigabyte.

7.1 Use Cases

These three use cases show how domain experts use the design system to understand a program's resiliency, reveal complex correlation patterns between error propagations, and demonstrate how the visualization system helps domain experts understand the amplification and mitigation behavior during the error propagation process. These use cases are performed in an interactive remote meeting environment and constructed with the guidance of domain experts. In the following discussion, we also highlight how each use case addresses domain tasks proposed in section 5.

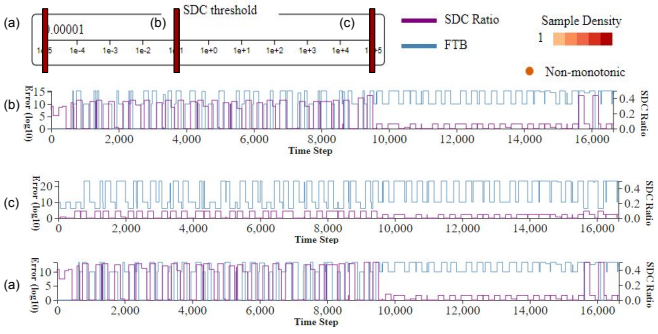


Fig. 12. FFT application shows more resiliency in the later computation than in an earlier computation under different SDC threshold configurations. Even with a large error tolerance, the later computation can still produce an SDC outcome with an error corruption.

This article has been accepted for publication in IEEE Transactions on Visualization and Computer Graphics. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2022.3230636

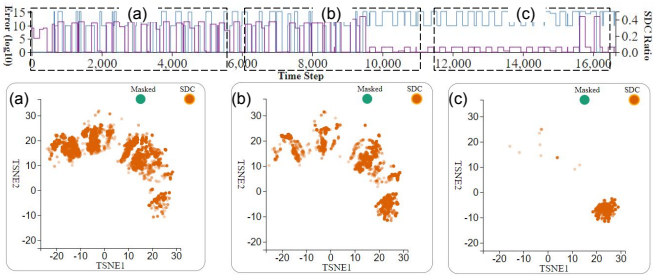


Fig. 13. Comparing the SDC cases where an error corruption propagates from (a) and (b) with a diverse propagation pattern. An error corrupting the later execution has a similar and coherent propagation pattern.

7.1.1 *Exploring a Program's Resiliency Through Fault Tolerance Boundary*

We start our evaluation by analyzing a fast Fourier transform's resiliency with the fault tolerance boundary (**Task 1**). Fig. 12 shows a fast Fourier transform's fault tolerance boundary and SDC ratio with three SDC thresholds. From the visualization, we can tell that the early execution of the fast Fourier transform has a high SDC ratio, and the later execution has a low SDC ratio. The later execution is much less sensitive to the SDC threshold update than the early execution. These observations indicate that the early computation is much more vulnerable to the soft error than the later execution.

The similarity of error propagation that starts from different regions is displayed in Fig 13. The error propagation experiments that start from regions (a) and (b) are more complex than an error propagation starting from region (c). The result also implies that detecting all error corruptions starting at the beginning of the computation can be more difficult than error corruptions that occur in region (c). From the visualization, we also find that nonmonotonic examples **(Task 3)** do not exist in the boundary view. The phenomenon indicates that all fault injection experiments in the FFT benchmark follow the propagation pattern (1)-(a), (2)-(a) and (3)-(b) in Fig. 5, and the error corrupts data variables in FFT following the monotonic assumption that more errors in the variable will cause a worse computation outcome.

7.1.2 *Understanding a Program's Corruption Behavior By Comparing Error Propagation*

Comparing a program's error propagation (**Task 2**) is important to understand a program's behavior after error corruption. Fig. 14 presents overviews of the error propagation similarity of the conjugate gradient (Fig. 14 left) and fast Fourier transform's (Fig. 14 right). Both propagation overviews share a similar pattern: a large of amount fault injection experiments cluster together and the rest of the error propagation experiments are distributed into multiple clusters. By selecting the largest cluster in each view, the bit-flip summary view gives a detailed summary of the relevant bits that are corrupted, leading to these propagations. The visualization reveals that most of these experiments' error propagation is caused by a bit flip corrupting the low bit, and these experiments do not cause significant error propagation. This observation also implies that a bit flip in the low bit often does not lead to the SDC outcome, and

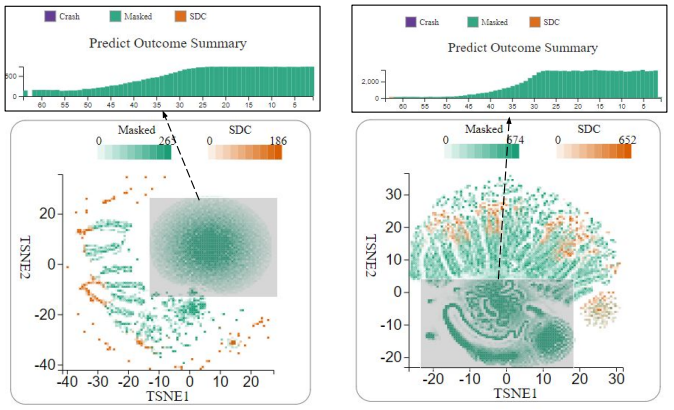


Fig. 14. The conjugate gradient (left view) and FFT (right view) in the visualization shows that a large amount of fault injection experiments that happen in the low bit of a program do not cause significant error propagation, and they share a similar error propagation pattern.

error propagation from these experiments provides only limited value to understand a program's behavior after error corruption.

Furthermore, we choose the conjugate gradient and examine its nearby executions' error propagation similarity. Fig. 15 compares the propagation similarity of error propagation over two regions (Fig. 15 (a) and (b)). Previous discussions have already clarified that error propagations in region (c) are caused by a bit flip in the low bit and do not cause significant error propagation. The visualization shows that an error propagation starting from the nearby execution in (a) has a similar error propagation pattern. The same pattern can also be applied to an error propagation starting from (b).

The above visualization indicates an interesting observation that error propagations collected from different benchmarks contain numerous experiments that are corrupted by a bit-flip error and will not cause significant error propagation. These experiments include error corruption not only in low mantissa bits but also in some of the exponent bits. Furthermore, many error propagations share a similar propagation pattern, and they are redundant to understand a program's corruption behavior.

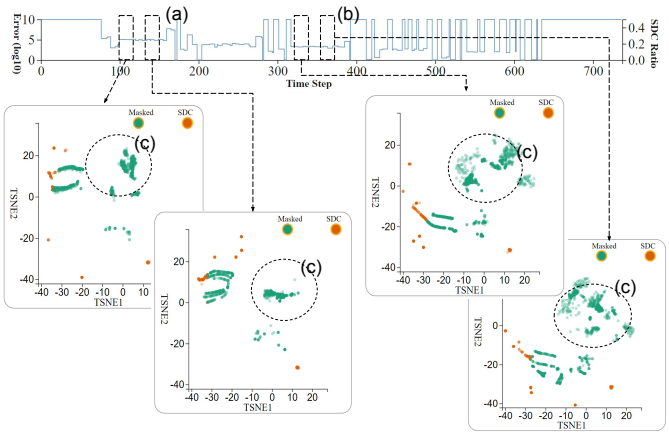


Fig. 15. An error corrupts nearby executions that share a similar error propagation pattern.

This article has been accepted for publication in IEEE Transactions on Visualization and Computer Graphics. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2022.3230636

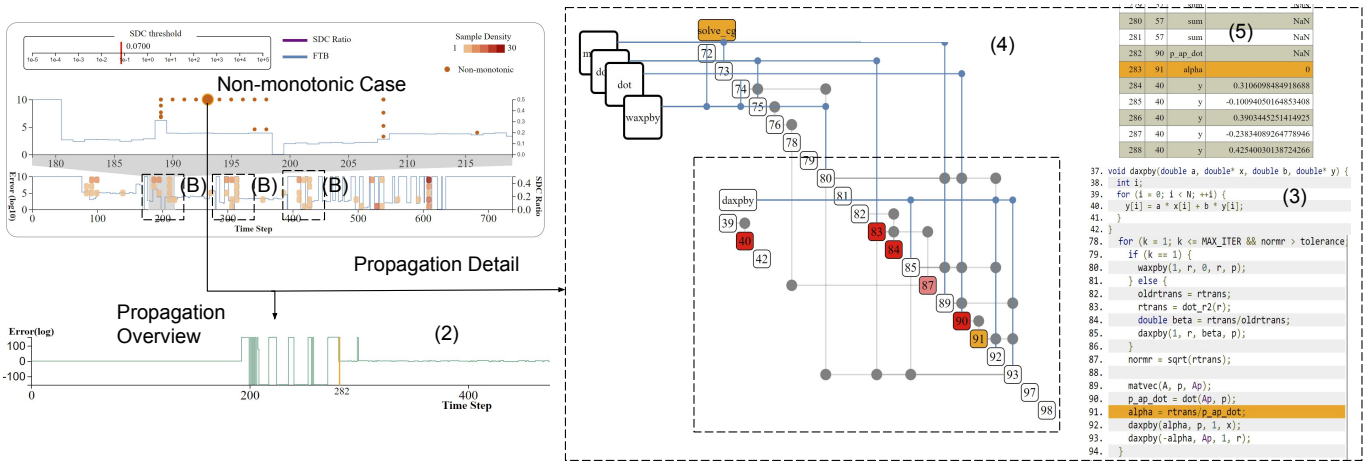


Fig. 16. All locations labeled B have a similar propagation pattern, in which a large error corrupts the computation but the error starts to disappear after the error propagates for a while. The error starts to disappear after the relative error in variable *alpha* in line 91 becomes 0, which causes the computation to discard the current iteration and automatically roll back to the previous iteration's computation state.

7.1.3 Locating Nonmonotonic Propagation Examples and Studying Mitigation/Amplification Propagation

In the last use case, we demonstrate that a fault injection experiment injects a large error into a computation, but the error is mitigated during error propagation (**Task 3**). We explain how the error mitigation happens by coordinating multiple views in the visualization tool (**Task 4**).

In Fig. 16, we select a fault injection experiment from location (B). It this experiment, a large error corrupts the program computation, but the error is mitigated after the error is propagated for a while. As we can see from Fig. 16 (2), the error propagation summary chart shows that a large error is injected around 200 time steps and causes a large vibration in the subsequent execution. These errors are mitigated after 281 time steps. The orange color in the figure highlights the location where the error starts to be mitigated. From the data log (5), we can see that alpha is set to 0. To simplify the dependency graph, we collapse the function graphs, which are not used in this current analysis. The dependency graph (4) on the right shows that line 91 is used by line 92 and line 93, both of which call the *daxpby* function. Previous executions do not depend on line 92, but the result of line 93 variable r is used by the previous execution lines 80, 83, and 85. In the source code (3) (**Task 4**), daxpby(-alpha, Ap, 1, r) is a function that performs the operation $r = -alpha * Ar + r$. The variable *alpha* is zero, which makes the above equation end as $r = r$, meaning the large error causes the program to not do anything in the current iteration. The large error makes the algorithm automatically skip a serious error-corrupted result of the current iteration and automatically roll back to the state of the previous iteration.

One of the surprising insights we obtained using our visualization analysis framework is that for some cases of soft errors, the conjugate gradient algorithm is able to *automatically* fix the corrupted computation by discarding the current iteration result and rolling back to the previous iteration, recomputing it, and generating a correct output. The mitigation mechanism we learned from this use case may be applied to other iterative applications to improve their computation resiliency.

7.2 Domain Feedback

We also interviewed two domain experts individually to collect their feedback regarding the final version of the system. They are also actively involved in the design of the tool and provide valuable feedback for the early iteration of the tool. For the final assessment, we go through a few stereotypical usage scenarios and identify where and how the tool can improve domain experts' daily task of analyzing the fault tolerance of HPC applications.

The summary of the first domain expert response: Overall, the resilience visualization framework lets us see fine-grained application resilience of not only the whole application and specific code regions but also individual instructions. The error propagation probing modules help us identify the cause of fault masking and propagation events behind SDCs at relevant locations, and further determine if a protection mechanism is needed or not at particular locations (such as instruction duplication for individual instructions). It provides us in-depth insights for error propagation that are not found in existing resilience analytical models.

The summary of the second domain expert response: With this tool, we can quickly assess the vulnerable regions and also look at individual fault injection cases to see how the error propagated and locations that resulted in the amplification of the errors and mitigation of the errors. This visualization also captures critical features and enables us to visualize in a compact form how they are related. It clearly shows that cases that result in SDCs are spatially co-located. Previously, identifying vulnerable regions of the code was tedious and error-prone. To obtain detailed information, we would have to examine a large number of fault injection runs. With this tool, we are not only able to instantly obtain an overall resiliency profile of the application, but also to do so with very few fault injection runs. We plan to use this information to apply fault detectors at those locations to detect SDCs. I also anticipate this tool to be useful for other kinds of error propagation analysis, including errors due to the loss of precision or approximations.

8 Limitation and Future Works

In the HPC domain, researchers select a standard input for the diagnostic application for resiliency analysis, but a program's resiliency profile can vary with a different input. The infinite possible inputs of a program with a different scale (e.g, 8x8 or 100x100 matrix) are a general challenge in HPC resiliency analysis [22]. Using the fault tolerance boundary to analyze a program's resiliency also faces a similar problem. In this study, we focused our analysis on small-scale HPC computation kernels (e.g, conjugate gradient, FFT). Although the applications run on the HPC system can be much more complex, their main computation is comprised of core kernels similar to the ones studied in this paper. Therefore, the insights obtained from our study of representative kernels can be applied to a certain extent in the context of the larger application and different inputs. In the HPC community, many researchers [58] have traded precision for performance. For example, lossy compression [59] techniques can significantly reduce the amount of data that needs to be moved and stored in the HPC system by allowing a small amount of precision loss. Mixed-precision tuning [60] techniques selectively reduce certain computation precision to alleviate the memory and energy cost for performance benefit. Both approaches introduce small errors in a trade-off for the performance improvement. However, how to convince researchers to use such techniques for scientific discovery is still a challenge due to the potential for error corruption propagation. Researchers do not know what information is corrupted with the introduced error's propagation and how important this information is. Understanding the error propagation of these techniques and presenting it to domain experts is helpful to address this challenge. In this study, the fault tolerance boundary giving the maximum threshold value of each dynamic instruction can be perturbed individually to assure a program's final output correctness. This concept can also be generalized to numerous dynamic instructions for lossy compression and pruning tuning, in which errors are introduced in several locations. In this study, the error propagation to approximate the fault tolerance boundary single dynamic instruction has shown a promising result. Understanding how this approach helps to bound the error in multiple locations such as lossy compression or precision tuning is interesting and will be explored in future research.

9 Conclusion

In this work, we perform a study of a program's resiliency through error propagation and design a visualization system to explore the error propagation behavior of a program. The result of our study reveals that using a small portion of samples to study a program's resiliency specific to each bit of the computation is possible. We demonstrate how our system can coordinate multiple views to present a program's fault tolerance boundary, a detail bit-flip summary, and a propagation summary in a united interface. This design enables domain experts to examine a program's resiliency from multiple perspective and provides valuable feedback for the domain experts in designing better protection mechanisms to improve programs' resiliency. During the exploration, our visualization revealed interesting insights, such as many error propagation experiments are redundant, or an error in these experiments does not lead to significant error propagation. At the same time, the error propagation visualization with nonmonotonic analysis shows that the conjugate gradient algorithm can automatically roll back to the previous iteration to fix the error in the computation. At the end, we evaluate the performance of our tool with three use cases and domain experts' feedback.

References

[1] 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.

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

[3] T. Benacchio, L. Bonaventura, M. Altenbernd, C. D. Cantwell, P. D. Düben, M. Gillard, L. Giraud, D. Göddeke, E. Raffin, K. Teranishi *et al.*, "Resilience and fault-tolerance in high-performance computing for numerical weather and climate prediction," *International Journal of High Performance Computing Applications*, 2020.

[4] 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.

[5] H. D. Dixit, S. Pendharkar, M. Beadon, C. Mason, T. Chakravarthy, B. Muthiah, and S. Sankar, "Silent data corruptions at scale," *arXiv preprint arXiv:2102.11245*, 2021.

[6] R. Venkatagiri, A. Mahmoud, S. K. S. Hari, and S. V. Adve, "Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–14.

[7] R. Venkatagiri, K. Ahmed, A. Mahmoud, S. Misailovic, D. Marinov, C. W. Fletcher, and S. V. Adve, "Gem5-approxilyzer: An open-source tool for application-level soft error analysis," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2019, pp. 214–221.

[8] 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," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 1, pp. 123–134, 2012.

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

[10] Q. Lu, G. Li, K. Pattabiraman, M. S. Gupta, and J. A. Rivers, "Configurable detection of sdc-causing errors in programs," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 3, pp. 1–25, 2017.

[11] Z. Li, H. Menon, K. Mohror, P.-T. Bremer, Y. Livant, and V. Pascucci, "Understanding a program's resiliency through error propagation," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 362–373. [Online]. Available: https://doi.org/10.1145/3437801.3441589

[12] S. K. Sastry Hari, R. Venkatagiri, S. V. Adve, and H. Naeimi, "Ganges: Gang error simulation for hardware resiliency evaluation," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 61–72, 2014.

[13] L. Bautista-Gomez and F. Cappello, "Detecting silent data corruption for extreme-scale mpi applications," in *Proceedings of the 22nd European MPI Users' Group Meeting*, 2015, pp. 1–10.

[14] P.-L. Guhur, E. Constantinescu, D. Ghosh, T. Peterka, and F. Cappello, "Detection of silent data corruption in adaptive numerical integration solvers," in *2017 IEEE international conference on cluster computing (CLUSTER)*. IEEE, 2017, pp. 592–602.

[15] E. Berrocal, L. Bautista-Gomez, S. Di, Z. Lan, and F. Cappello, "Lightweight silent data corruption detection based on runtime data analysis for hpc applications," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, 2015, pp. 275–278.

[16] Z. Li, H. Menon, D. Maljovec, Y. Livnat, S. Liu, K. Mohror, P.-T. Bremer, and V. Pascucci, "Spotsdc: Revealing the silent data corruption propagation in high-performance computing systems," *IEEE Transactions on Visualization and Computer Graphics*, 2020.

[17] T. C. May and M. H. Woods, "Alpha-particle-induced soft errors in dynamic memories," *IEEE transactions on Electron devices*, vol. 26, no. 1, pp. 2–9, 1979.

[18] F. Cappello, R. Gupta, S. Di, E. Constantinescu, T. Peterka, and S. M. Wild, "Understanding and improving the trust in results of numerical simulations and scientific data analytics," in *European Conference on Parallel Processing*. Springer, 2017, pp. 545–556.

[19] L. Guo, D. Li, and I. Laguna, "Paris: Predicting application resilience using machine learning," *Journal of Parallel and Distributed Computing*, vol. 152, pp. 111–124, 2021.

[20] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," in *2009 Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 2009, pp. 502–506.

[21] S. K. S. Hari, R. Venkatagiri, S. V. Adve, and H. Naeimi, "Ganges: Gang error simulation for hardware resiliency evaluation," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 2014, pp. 61–72.

[22] 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.

[23] A. R. Anwer, G. Li, K. Pattabiraman, M. B. Sullivan, T. Tsai, and S. K. S. Hari, "Gpu-trident: efficient modeling of error propagation in GPU programs." IEEE/ACM, 2020, p. 88.

[24] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE transactions on computers*, vol. 100, no. 6, pp. 518–528, 1984.

[25] S. Di, E. Berrocal, and F. Cappello, "An efficient silent data corruption detection method with error-feedback control and even sampling for hpc applications," in *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 2015, pp. 271–280.

[26] H. Guo, S. Di, R. Gupta, T. Peterka, and F. Cappello, "La valse: Scalable log visualization for fault characterization in supercomputers," in *Proceedings of the Symposium on Parallel Graphics and Visualization*, ser. EGPGV '18. Goslar, DEU: Eurographics Association, 2018, p. 91–100.

[27] K. Wongsuphasawat, D. Smilkov, J. Wexler, J. Wilson, D. Mane, D. Fritz, D. Krishnan, F. B. Viégas, and M. Wattenberg, "Visualizing dataflow graphs of deep learning models in tensorflow," *IEEE transactions on visualization and computer graphics*, vol. 24, no. 1, pp. 1–12, 2017.

[28] 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 and Computer Graphics*, vol. 25, no. 1, pp. 215–224, 2019.

[29] 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." in *EuroVis (STARs)*, 2014.

[30] M. Ghoniem, J.-D. Fekete, and P. Castagliola, "A comparison of the readability of graphs using node-link and matrix-based representations," in *IEEE Symposium on Information Visualization*. Ieee, 2004, pp. 17–24.

[31] R. Keller, C. M. Eckert, and P. J. Clarkson, "Matrices or node-link diagrams: which visual representation is better for visualising connectivity models?" *Information Visualization*, vol. 5, no. 1, pp. 62–76, 2006.

[32] W. J. R. Longabaugh, "Combing the hairball with biofabric: a new approach for visualization of large networks," *BMC Bioinformatics*, vol. 13, pp. 275 – 275, 2012.

[33] B. Watson, D. Brink, T. Lograsso, D. Devajaran, T. Rhyne, and H. A. Patel, "Visualizing very large layered graphs with quilts," 2008.

[34] C. Nobre, D. Wootton, L. Harrison, and A. Lex, "Evaluating multivariate network visualization techniques using a validated design and crowdsourcing approach," ser. CHI '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1–12. [Online]. Available: https://doi.org/10.1145/3313831.3376381

[35] C. Vehlow, F. Beck, and D. Weiskopf, "Visualizing group structures in graphs: A survey," *Computer Graphics Forum*, vol. 36, 2017.

[36] T. Siddiqui, P. Luh, Z. Wang, K. Karahalios, and A. Parameswaran, "Shapesearch: A flexible and efficient system for shape-based exploration of trendlines," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 51–65.

[37] T. Rakthanmanon, B. Campana, A. Mueen, G. Batista, B. Westover, Q. Zhu, J. Zakaria, and E. Keogh, "Searching and mining trillions of time series subsequences under dynamic time warping," in *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2012, pp. 262–270.

[38] J. J. Van Wijk and E. R. Van Selow, "Cluster and calendar based visualization of time series data," in *Proceedings 1999 IEEE Symposium on Information Visualization (InfoVis' 99)*. IEEE, 1999, pp. 4–9.

[39] N. Ruta, N. Sawada, K. McKeough, M. Behrisch, and J. Beyer, "Sax navigator: Time series exploration through hierarchical clustering," in *2019 IEEE Visualization Conference (VIS)*. IEEE, 2019, pp. 236–240.

[40] A. Gogolou, T. Tsandilas, T. Palpanas, and A. Bezerianos, "Comparing similarity perception in time series visualizations," *IEEE transactions on visualization and computer graphics*, vol. 25, no. 1, pp. 523–533, 2018.

[41] U. Jugel, Z. Jerzak, G. Hackenbroich, and V. Markl, "Faster visual analytics through pixel-perfect aggregation," *Proceedings of the VLDB Endowment*, vol. 7, no. 13, pp. 1705–1708, 2014.

[42] G. Burtini, S. Fazackerley, and R. Lawrence, "Time series compression for adaptive chart generation," in *2013 26th IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*. IEEE, 2013, pp. 1–6.

[43] A. Kim, E. Blais, A. Parameswaran, P. Indyk, S. Madden, and R. Rubinfeld, "Rapid sampling for visualizations with ordering guarantees," in *Proceedings of the vldb endowment international conference on very large data bases*, vol. 8, no. 5. NIH Public Access, 2015, p. 521.

[44] U. Jugel, Z. Jerzak, G. Hackenbroich, and V. Markl, "M4: A visualization-oriented time series data aggregation," *Proc. VLDB Endow.*, vol. 7, no. 10, p. 797–808, Jun. 2014. [Online]. Available: https://doi.org/10.14778/2732951.2732953

[45] I. T. Jolliffe and J. Cadima, "Principal component analysis: a review and recent developments," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 374, no. 2065, p. 20150202, 2016.

[46] A. Buja, D. F. Swayne, M. L. Littman, N. Dean, H. Hofmann, and L. Chen, "Data visualization with multidimensional scaling," *Journal of computational and graphical statistics*, vol. 17, no. 2, pp. 444–472, 2008.

[47] C. O. S. Sorzano, J. Vargas, and A. P. Montano, "A survey of dimensionality reduction techniques," *arXiv preprint arXiv:1403.2877*, 2014.

[48] L. Van der Maaten and G. Hinton, "Visualizing data using t-sne." *Journal of machine learning research*, vol. 9, no. 11, 2008.

[49] L. McInnes, J. Healy, and J. Melville, "Umap: Uniform manifold approximation and projection for dimension reduction," *arXiv preprint arXiv:1802.03426*, 2018.

[50] J. Wei, A. Thomas, G. Li, and K. Pattabiraman, "Quantifying the Accuracy of High-Level Fault Injection Techniques for Hardware Faults," in *DSN*, 2014.

[51] J. Calhoun, M. Snir, L. N. Olson, and W. D. Gropp, "Towards a more complete understanding of sdc propagation," in *HPDC*, 2017.

[52] H. Menon and K. Mohror, "Discvar: Discovering critical variables using algorithmic differentiation for transient faults," *ACM SIGPLAN Notices*, vol. 53, no. 1, pp. 195–206, 2018.

[53] L. Guo, D. Li, I. Laguna, and M. Schulz, "Fliptracker: Understanding natural error resilience in hpc applications," in *SC*, 2018.

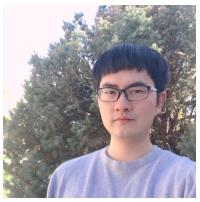
[54] I. Laguna, M. Schulz, D. F. Richards, J. Calhoun, and L. Olson, "IPAS: Intelligent Protection Against Silent Output Corruption in Scientific Applications," in *CGO*, 2016.

[55] L. Guo and D. Li, "MOARD: Modeling Application Resilience to Transient Faults on Data Objects," in *International Parallel and Distributed Processing Symposium*, 2019.

[56] Z. Chen, G. Li, and K. Pattabiraman, "A low-cost fault corrector for deep neural networks through range restriction," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2021, pp. 1–13.

[57] M. Behrisch, B. Bach, N. Henry Riche, T. Schreck, and J.-D. Fekete, "Matrix reordering methods for table and network visualization," in *Computer Graphics Forum*, vol. 35, no. 3. Wiley Online Library, 2016, pp. 693–716.

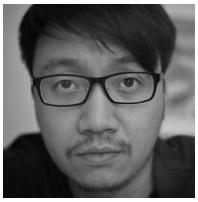
[58] P. Lindstrom, "Fixed-rate compressed floating-point arrays," *IEEE transactions on visualization and computer graphics*, vol. 20, no. 12, pp. 2674–2683, 2014.
[59] S. Di and F. Cappello, "Fast error-bounded lossy hpc data compression with sz," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 730–739.
[60] W.-F. Chiang, M. Baranowski, I. Briggs, A. Solovyev, G. Gopalakrishnan, and Z. Rakamarić, "Rigorous floating-point mixed-precision tuning," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 300–315. [Online]. Available: https://doi.org/10.1145/3009837.3009846



Zhimin Li is a graduate student working on his PhD from the School of Computing at the University of Utah. Zhimin is also a research assistant at the University of Utah's Scientific Computing and Imaging Institute. He received his B.S. in computer science and mathematics from the University of Utah in 2016. His research interests include visualization and interpretable machine learning.

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.

Kathryn Morhor is a computer scientist in the Center for Applied Scientific Computing (CASC) at Lawrence Livermore National Laboratory (LLNL). Kathryn serves as the Deputy Director for the Laboratory Directed Research & Development (LDRD) program at LLNL, Lead for the NNSA Software Technologies Portfolio for the U.S. Exascale Computing Project (ECP), and as the ASCR Point of Contact for Computer Science at LLNL. Kathryn's research on high-end computing systems is currently focused on I/O for extreme scale systems. Kathryn has been working at LLNL since 2010 and is a 2022 fellow in the Oppenheimer Science and Energy Leadership Program and a 2019 recipient of the DOE Early Career Award.



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.

Luanzheng Guo is a postdoctoral researcher at the Pacific Northwest National Laboratory, working with the HPC Group. His current research focuses on AI-centric compiler optimization and system runtime for heterogenous systems. He obtained his Ph.D. degree in Electrical Engineering and Computer Science from the University of California-Merced in 2020. His Ph.D. research focused on system resilience and reliability in large-scale parallel HPC systems. His research on HPC system fault tolerance has been highlighted by HPCwire in its "What's new in HPC research" in 2018 and 2021. He is an NSF Trusted CI Fellow of Class 2020.

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.