ABSTRACT

Aggressive technology scaling trends are expected to make the hardware of high-performance computation (HPC) systems more susceptible to transient faults. Transient faults are commonly caused by high-energy particles from cosmic radiation or electrical noise and corrupt program computation by causing random bit flips. These faults may be masked by program logic without affecting the program output; cause a program to crash, e.g., a transient fault corrupts a pointer variable; or lead to silent data corruption (SDC), where the transient fault introduces an error into the application that is not detected. The HPC community is concerned with the impact of silent data corruption on computation results, on which critical decisions may rely. Due to the insidious nature of SDC, understanding a fault-corrupted program's behavior is crucial for developing and evaluating software resiliency techniques to ensure correct output results from HPC applications. Classical fault injection studies give an overall statistical resiliency profile for an application. However, summarizing critical information and understanding the behavior of the fault-injected program are challenging. In this work, we introduce SpotSDC, a visualization system for analyzing a program's characteristics of resilience to SDC. SpotSDC provides an overview of the effect of SDC on an application by indicating the impact of a transient fault over the program's different region and different time steps during program execution. It highlights the regions of code that are most susceptible to SDC and that will have a high impact on the program's output. SpotSDC also enables users to visualize the propagation of error through a program execution. The results from SpotSDC allow developers to design selective algorithmic error detection and mitigation techniques to protect their applications from SDC.

Introduction

Aggressive technology trends tend to make the hardware of high-performance computation systems more susceptible to transient faults. Transient faults, which are commonly caused by cosmic radiation, device noise, and other factors, can manifest in applications in three ways. They can cause an application to crash, e.g., the transient fault corrupts a pointer variable and causes a segmentation fault. They can be masked without affecting the application’s output. The most insidious outcome is silent data corruption (SDC), where the transient fault introduces an error in the application that is not detected and will potentially alter the computation results without being noticed. The corrupted computation result will then pollute subsequent researches or lead to incorrect decision.

Due to the insidious nature of SDC, understanding how it affects a particular application is critical for developing and evaluating software resiliency techniques. The classical approach for studying how transient faults affect an application is a fault injection campaign, in which a tool injects an artificial fault into an application to test the program, observes its behavior, records the result and provides a high-level statistical profile.

However, without a detailed examination of the SDC impact over different regions of a program and different time step of a program execution, the conclusion of the analysis may be incomplete and inaccurate. Examining a fault injection case by observing the error propagate through a program is a difficult task for HPC researchers, since they must repeatedly check the states between the fault injection computation and error-free computation. At the same time, studying how a transient error propagates through a program and results in SDC or how it is masked by the program logic will help researchers develop a better understanding of the impact of a transient error on a program.

In this work, we introduce SpotSDC, an interactive visualization system to analyze a program’s characteristics of resilience to SDC. SpotSDC identifies the impact of SDC through space and time. It highlights the regions of code that are most susceptible to SDC and that have a high impact on the program’s output. SpotSDC also provides users with the overall SDC characteristics of the application and enables them to visualize the propagation of error through an application execution. SpotSDC will also enable researchers to design selective algorithmic error detection and mitigation techniques to protect their code from producing erroneous output.

Our key contributions in this paper are as follows:

* An interactive visualization system to study the silent data corruption vulnerability and impact of HPC computation kernels.
* A bit-based visualization to address the error scaling challenges among multiple code regions, a study of the difficulty of visually comparing the sensitivity of various code regions.
* A grand visualization display of the sample space of the fault injection to help domain experts reveal the limits of the data sample.
* A visual fault injection analysis scheme from statistics to annotate source code for reasoning and planning for protection.
* A visual encoding to encompass multiple levels of granularity from program to function, to variable, and finally to individual bits.

The remainder of this paper is structured as follows: Section 2 discusses the related work in this area, Section 3 introduces the domain background, Section 4 talks about the domain tasks, and Section 5 discusses the challenge of comparing multiple code regions. In Section 6, we show how users can use our tool to improve their sampling. Section 7 introduces the interface of the system, Section 8 demos the usability of the system, and finally, we conclude our research in section 9.

Domain Background and Data Description

In this study, we do exhaust fault injection experiments on a HPC computation kernel; record the result of each experiment and the value of each critical data variable during program execution.

# The SDC Ratio of a Program’s Different Bits

The IEEE 745 double-precision binary float-point format has 1 sign bit, 11 exponent bits, and 52 mantissa bits. During the program execution, each dynamic instruction has an equal chance of being corrupted by a transient fault. Each instruction has 64 bits, each with an equal chance of being flipped. Figure 1 shows the 64-bit IEEE 754 double-precision binary floating-point representation.

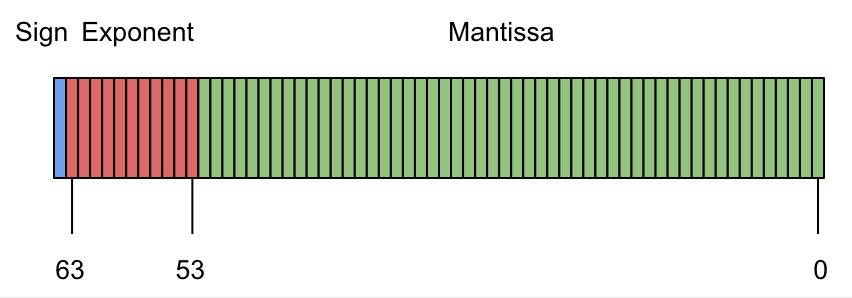


Figure 1 IEEE 754 double-precision binary floating-point format

# Dynamic Instruction

A program is constructed of many dynamic instructions. For this study, we focus on the instruction that will change the value of critical variables. Each instruction is a potential fault injection location.

# Fault Injection

A fault injection experiment is the classical method to understand the dependability of a system. In each experiment, an artificial fault (a bit flip) is injected at a data variable once in a random time step during program execution and records the computation result. A collection of fault injection experiments is called a fault injection campaign.

For each fault injection experiment, the following information will also be recorded:

* The name of the function in which each fault is injected.
* The line number in which the fault is injected.
* The variable name in which the fault is injected.
* The injected variable’s initial value.
* The injected variable’s corrupted value after the fault is injected.
* The bit number of a variable that is flipped.
* The approximate error of the program output.
* The outcome of a program running, masked, SDC, DUE.
* The dynamic instruction number in which the fault is injected.

In each experiment, we observe the behavior of the program by recording the value of the program’s critical variable. The artificial fault can be injected at any time during the program execution, but the observation begins at the start of the program until it exits.

In this research, we study the transient fault caused by a bit flip, in which one bit of a 64-bit float value has been unexpectedly flipped. Here, we use stencil computation as an example to explain what a bit flip is and what kind of result it may cause. As Figure 3 shows, one bit of a matrix variable Anext’s one element, which is a float value, is flipped. If the program continues for execution, this error will propagate through to the other elements of the matrix. Once the program finishes executing, it will produce an error outcome.

# Program Outcome

The outcome of a fault injection experiment may be classified in many different ways, e.g., whether the program needs additional execution time to produce a correct solution or the error in a program propagates to the memory. In this study, we define three categories of a fault injection experiment outcome: Masked, SDC, and DUE based on a threshold value.

In scientific computation, many problems are solved with fault tolerance, in which the program does not give an exact solution but a solution that is close enough to the ground truth. A threshold value, which is defined by a domain expert, is used to define the accuracy of the approximate solution. In a fault injection experiment, we use the threshold value to decide the category of an experiment output.

* **Masked**: A fault is injected into the program, and the program finishes running and gives an output that has the error under the threshold. If the fault does not impact the program’s outcome, the injected error may be too small to impact the program’s behavior. The other case, which may be more interesting, is that the injected error is large, but the program logic dismisses it.
* **Silence Data Corruption (SDC)**: Error is injected into the program, and the program finishes running but the output error is above the threshold. It either gives an incorrect solution to the researchers or pollutes the subsequent computation. However, in neither case will the program report an error or warn the researchers.
* **Crash**: After the error is injected into the program, the outcome of the program is NAN or infinity. We define this outcome as program crash because this type of output can be easily caught by program’s protection mechanism, e.g., throw exception.

# Feature of the Data

Our data set commonly has hundreds of thousands of fault injection records, and the size of the data is about a few MB. Of each fault injection, the value of each critical variable during the program execution is recorded. For all execution records, the size is hundreds of MB.

# How to Trace Error

In order to measure the error in the fault-injected experiment, we align the golden execution instruction with the fault injection execution instruction. For each alignment instruction, we take the absolute value of the difference between two executions. Here, we assume that the golden run and the error run will have a similar computing structure. The computation state of two different run is perfectly matched.

# SDC Ratio and SDC Impact (incomplete section)

The number of times the fault is injected into location A.

# Domain Tasks

We are collaborating with two computer scientists in the fault tolerance research community, including fourteen months of remote meeting and three months of on-site bi-weekly meetings. The SpotSDC design is constructed based on feedback from the domain experts about what they want to see. On an informal basis, we also ask other researchers interested in fault tolerance analysis how they analyze the bit flip problems in program execution. After our many discussions, we have formalized the domain experts’ interest into four goals.

* **G1: Understand the impact of error in different code regions.**

For each program test, different silent data corruption situations have different error outcome scales. Some cases with large-scale error need more attention compared to the case in which the error is only slightly above the error threshold. The large-scale error may severely pollute the next step calculation in the computation pipeline. The study of the error outcome distribution for all tests helps domain experts gain a better understanding of the resilience property of the code region.

* **G2: Understand the silent data corruption vulnerability of different code regions.**

During the diagnostic process, each code section will be tested many times at different time steps. Revealing how many times an error may attack on a location will explain how vulnerable the code region is.

* **G3: Reason about the interesting feature.**

During our discussion with the domain experts, they frequently mention they need to go back to the source code to analyze why some code regions are vulnerable and resilient. Visualization presents abstract information to which the domain experts should pay more attention, but they still need to go back to the source code for advanced analysis.

* **G4: Communicate and presentation.**

Domain experts are interested in using the visualization tool to explain the fault tolerance analysis and error propagation analysis to other experts.

* **T1: Aggregate the code region data based on the program structure.**

The domain experts are interested in understanding the resilience of the program from multiple levels instead of one specific location. They may question how resilient the whole program is to the error or how specific function’s fault tolerance character. An interactive control that allows users to aggregate data from different hierarchical levels of a program is essential to answer these questions.

* **T2: Examine the property of a target region.**

How specific code region resilient to the error and why a specific code region is vulnerable to the error are question domain expert interested to know.

* **T3: Explore a subset of the data.**

Domain experts are interested to see how different types of error impact the outcome of the simulation. How do different mantissa and exponent bits change the outcome? How cans a small error in the program affect the result of a simulation?

* **T4: Connect the source code with the abstract visualization.**

Once the domain experts observe some interesting feature in the data set, they want to check the source code and reason about it. They also annotate their source code to understand the resilience of the program.

System

SpotSDC has server visualization views to help researchers assess the impact of transient fault. The primary views are 1) Overview, 2) SDC analysis of source code regions, and 3) Error propagation view. It loosely follows Schneiderman mantra: “Overview first, zoom, filter, details on demand”, where SpotSDC first provides an overview impact of transient fault on the program output, and the user can conduct further detailed analysis by zooming into a specifics about source code component or analyze the error propagation for a specific case.

In this section, we provide details about the individual components in SpotSDC and highlight the main insights obtained from our visualization system. An exhaustive fault injection campaign was performed on a conjugate gradient (CG) benchmark, where we used a single bit-flip fault model. The transient faults introduced by our fault injection have three outcome, namely crash (also denoted as DUE), masked and SDC. SpotSDC uses the data provided by the fault injection campaign to construct different visualization views to help researchers identify critical resilience properties and behavior of CG in the present of transient faults.

# Overview Visualization

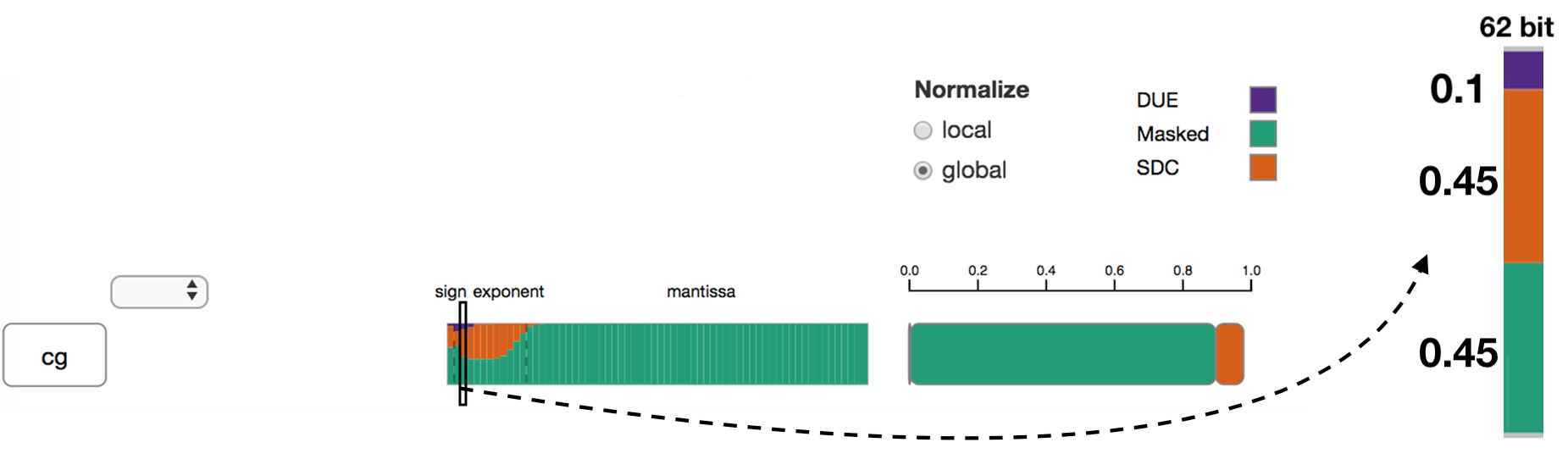


Figure 2. SpotSDC shows the overall transient fault impact on a conjugate gradient benchmark(CG)

The overview visualization component provides a high level view of the entire application’s resilience characteristics. Figure 2 shows the overview component of CG. This view has a hierarchical structure, where the users can select different contexts for the hierarchy. The overview visualization panel provides the following for the selected program components.

* Frequency of transient faults.
* SDC, DUE and Masked rates.
* Distribution of SDC with respect to bit location.

In Figure 2, we show on overview view that gives the overall impact of transient faults on CG. The stacked bar chart on the right shows the total number of fault injections conducted on CG, and the overall SDC rate for this application. The chart on the left is called the BitStackChart, which indicates the distribution of transient fault outcomes with respect to the bit location. Here we use the IEEE Standard 754 floating-point representation, and the x-axis shows the bit location for double precision floating-point format having 52 bits of mantissa, 11 bits of exponent and 1 sign bit. At each bit location in the BitStackChart, a stacked bar chart shows the distribution of SDC, DUE and Masked due to a transient fault (bit flip) at that bit location. Specifically, Figure 2 shows that 45% of the single bit flips at bit location 62 result in SDC (shown in orange), 10% result in DUE (blue), and 45% are masked by the program (green).

# Source Code View

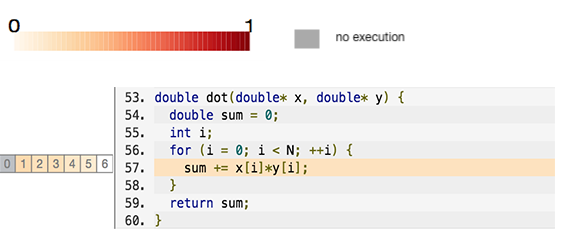


Figure 3. Source code shows critical line's SDC rate and different iterations' SDC ratio

Figure 3 is a subsection of the source code view showing the SDC analysis of line 57 in function dot of CG. The view highlights critical regions of the code, where an error can significantly corrupt the outcome of the application, and also provides information with respect to loop iterations by color coding the iteration number next to the source line. The gray color indicates no execution happen at that iteration.

# Propagation View

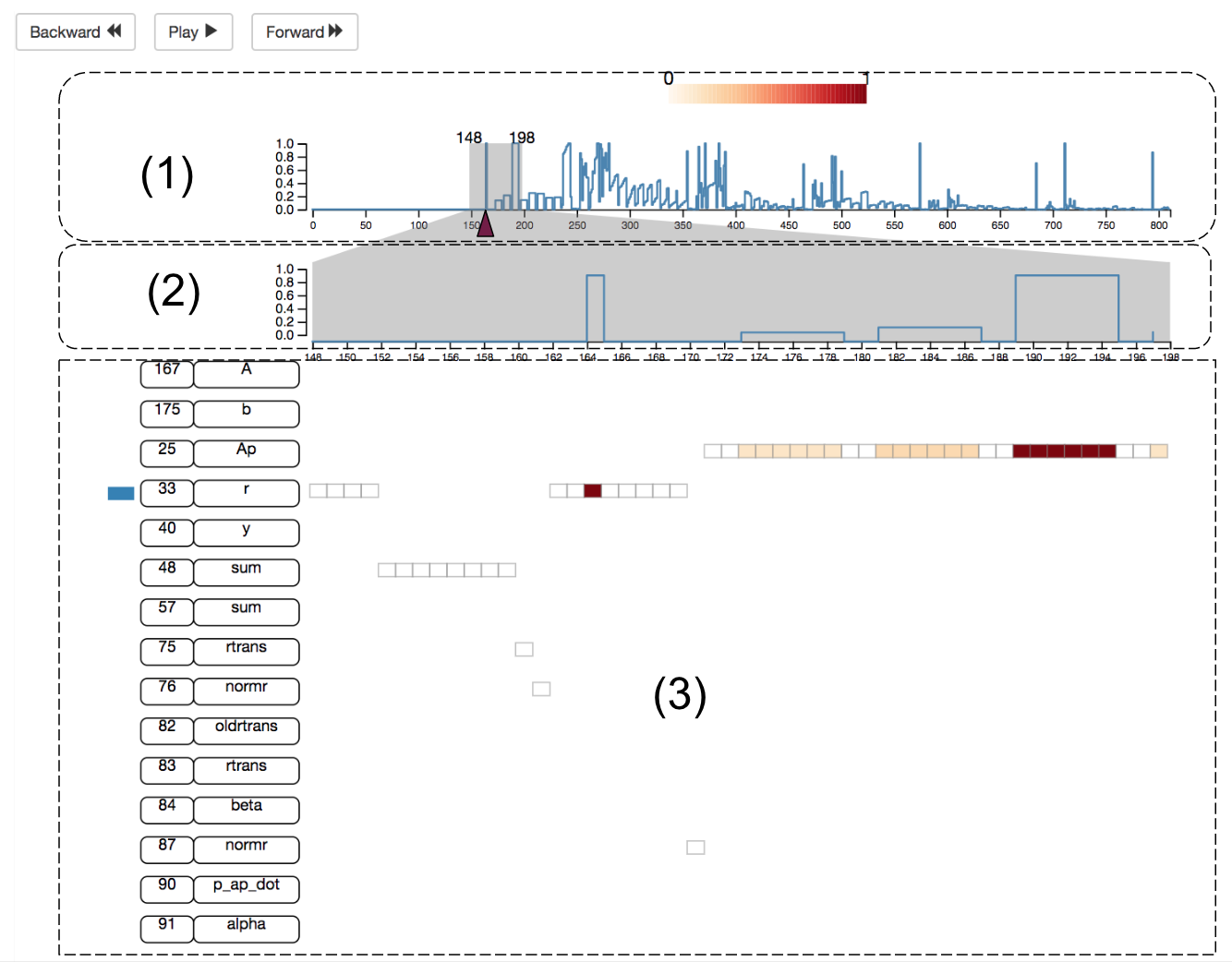


Figure 4. Error propagation view shows a transient fault propagate through a program execution

In Figure 4 of this abstract we show how a transient error propagate through CG. The top panel (1) shows where the error was introduced (depicted by the red triangle), and the relative error in variables as the execution progresses. The middle panel (2) zooms into the period of execution, which is under the zoom lens to show more detail. Finally, the bottom panel (3) shows how the error propagates from one variable to another within that time interval. As the visualization shows, the error starts at line 33 in variable r, and propagates to line 25 in the array variable Ap.

Results

# Impact of Different Bits

Figures 2 and 3 demonstrate that most of the fault injection caused by the bit flips occurs in the exponent bit, and the majority of the mantissa bit does not result in silent data corruption.

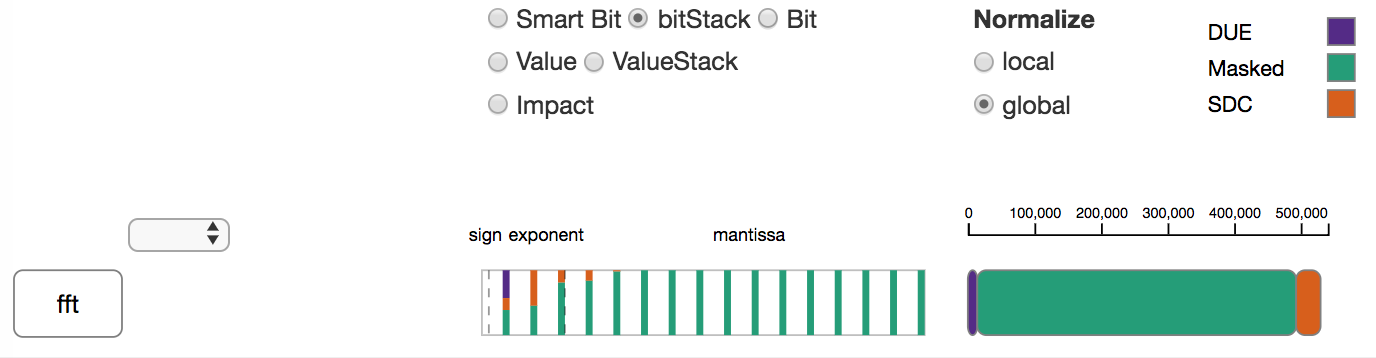


Figure 5. SDC ratio of CG in each bit

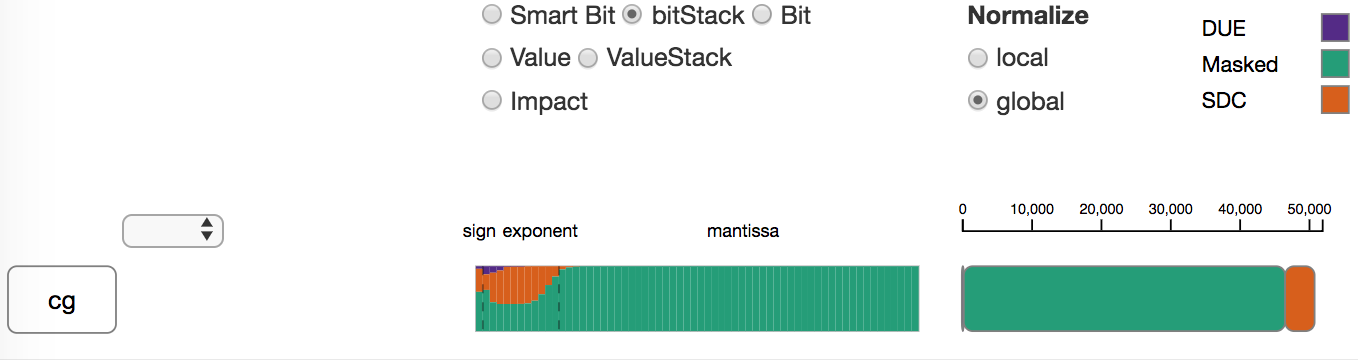


Figure 6. SDC ratio of FFT in every four bits

Figures 2 and 3 show the summary results of data collected by the fault injection campaign in FFT and CG, two different applications. Both visualizations show that the highest bit has a lower SDC ratio than a lower bit in the two applications. In CG, the highest exponent bit has a smaller SDC ratio compared to the second highest exponent bit. In FFT (Figure 2), for the data we have, the highest exponent bit has an SDC ratio that is lower than the second highest exponent bit. The DUE (program crash) ratio is high, but DUE can be detected easily.

# Sample of Data Set

The data is collected by a random fault injection campaign. Therefore, understanding the sample distribution of bit flip in a program is necessary to reach an accurate conclusion about the SDC ratio or SDC impact of a specific code region.

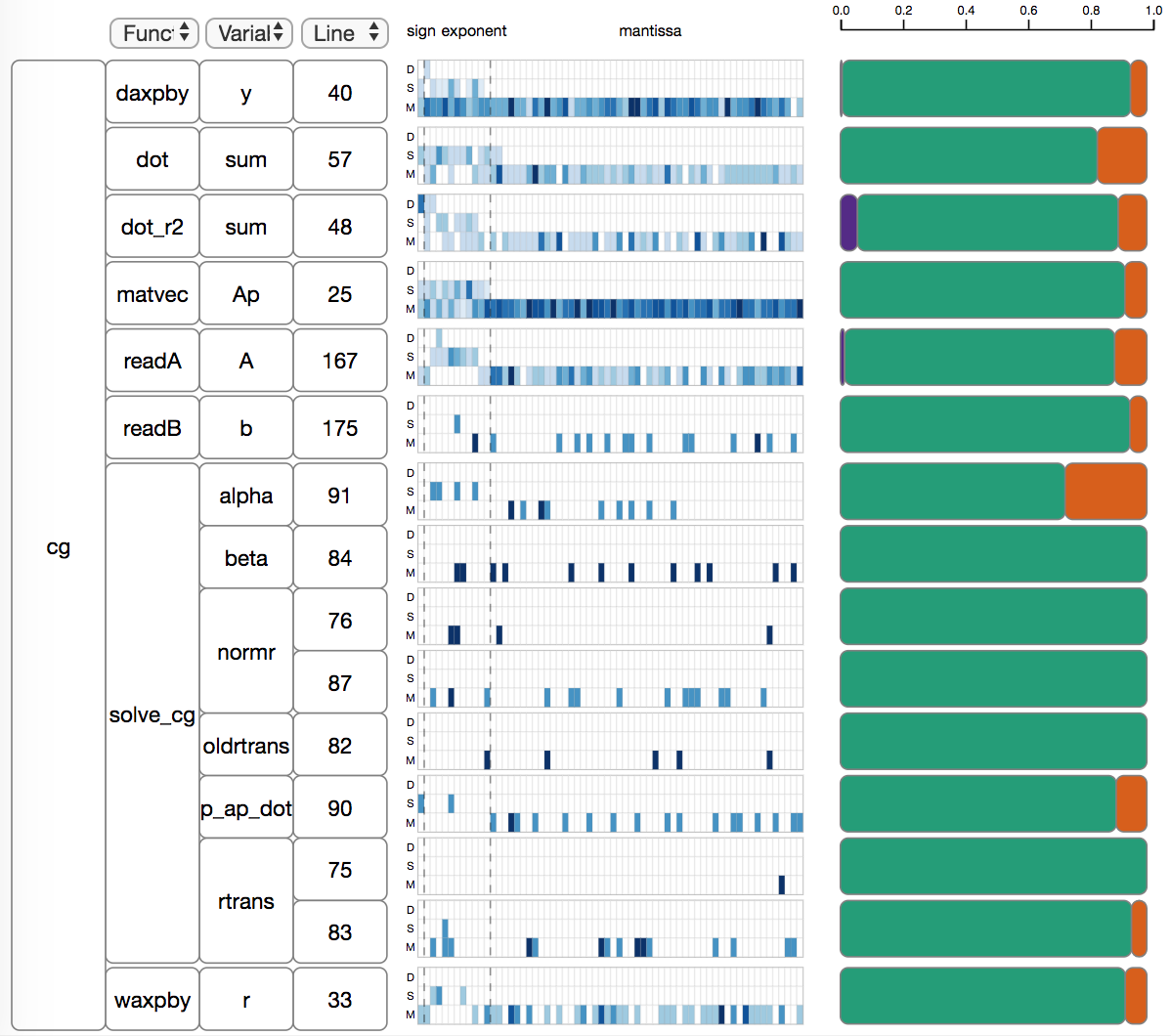


Figure 7. Displaying the bit flip distribution in a different line of code.

In Figure 4, line 75 does not have an SDC outcome for the fault injected in this location, and the SDC ratio in this line is 0. However, the user can check the heat-map on the right to see the distribution of the bit flip that occurs in these locations. For line 75, all the bit flips happen in the low mantissa bit. As we mentioned before, bit flip in the mantissa has much less impact. In order to reach a more concrete conclusion about the impact of this line, the user needs to apply more fault injection experiments in this location with a balance bit flip distribution.

# Vulnerability and Resiliency

As the visualization in Figure 5 shows, **matvec-Ap-25** has been executed most of the time, and the number of SDC outcomes caused by error injected is the largest. The line of code that has been executed most frequently does not imply a high SDC frequency. The user can choose to do local normalization (Figure 6), which will show the SDC ratio in this line of code. **Daxpby-y-40** is executed more frequently than **dot-sum-57** when we try to compare these two lines in the global domain. Once we analyze the SDC ratio in the local domain, we can see that **dot-sum-57** has a higher SDC ratio than **daxpby-y-40**. At the same time, the user can check the heat map in the middle, which displays the output error distribution in a different line of code. In Figure 5, **line 25** has the highest chance of being attacked by the transient fault. However, for all the SDC output caused by fault injected in this region, most of the output error is relatively small.

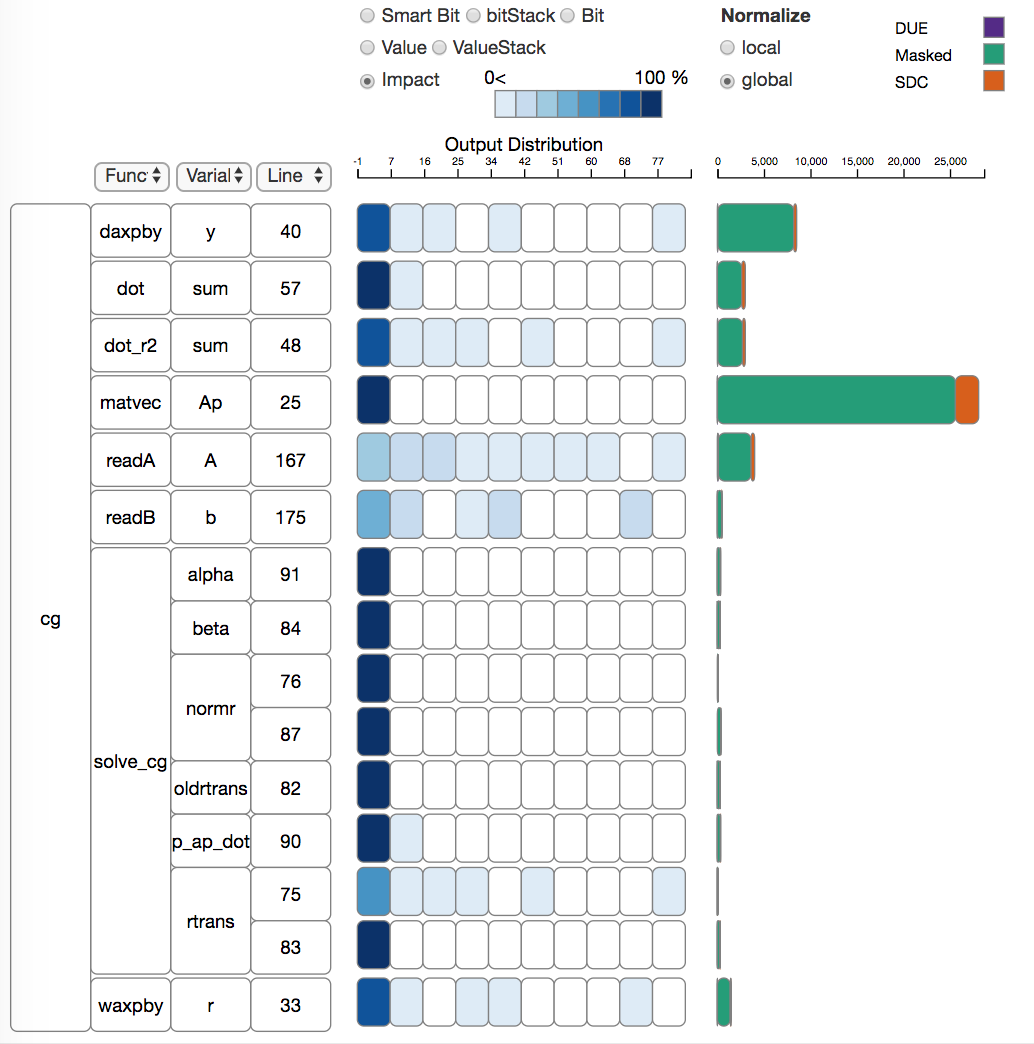


Figure 8 Which code region is more vulnerable to the fault injection.

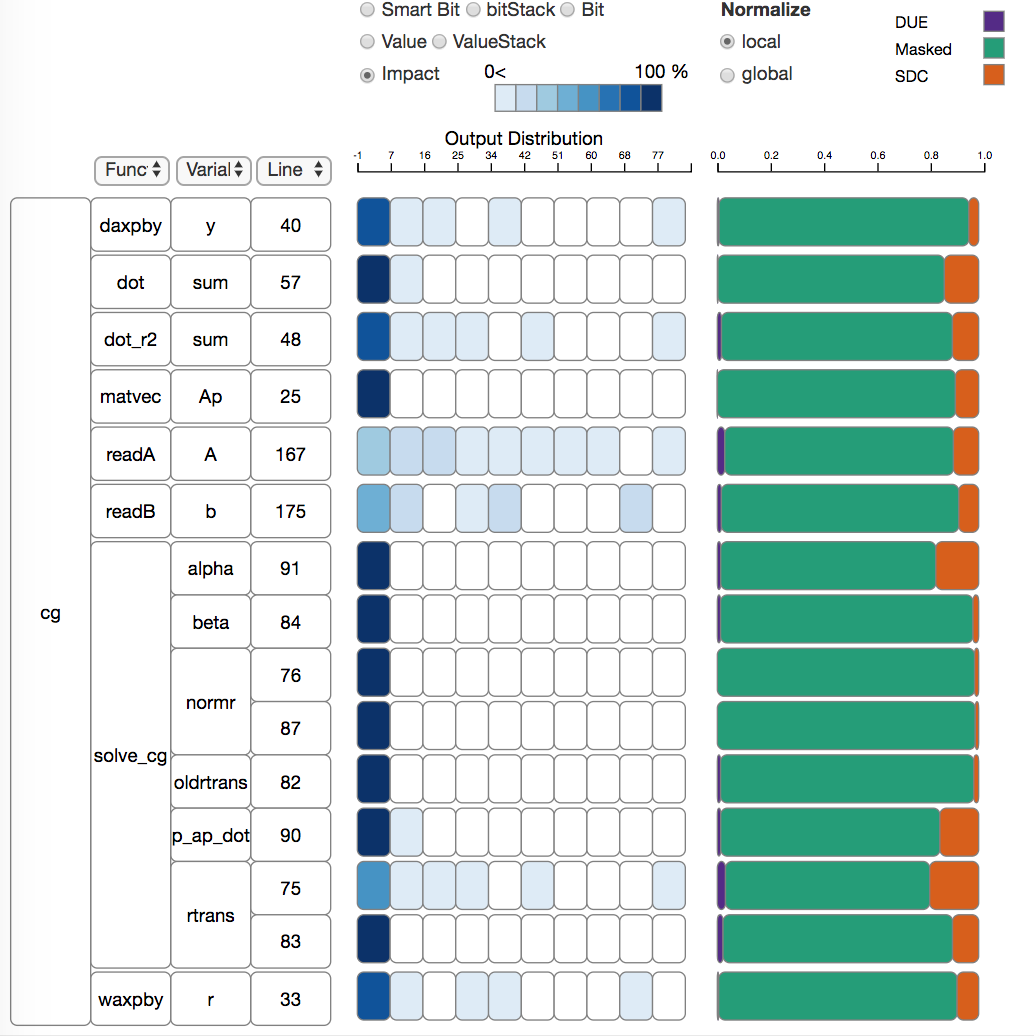


Figure 9. How different iterations will affect the error injection result.

# Error Injection in Different Iterations

During the program execution, when the error is injected into the program may impact the outcome of the simulation differently. A user can choose iteration information as the first level of the location indication tree and check how error that happens earlier or later may affect the outcome of the simulation. As we can see from Figure 7, the number of times error is injected into the earlier iterations 1 to 6 is the same. In zero iteration, the program initializes the value of the variables. However, as we can observe from Figure 8, once we filter out the masker sample, error that happens earlier has a greater impact than that which happens later. Error injected into the first iteration causes the greatest amount of SDC, and as the iteration number increases, the number of SDC caused by error in this iteration decreases.

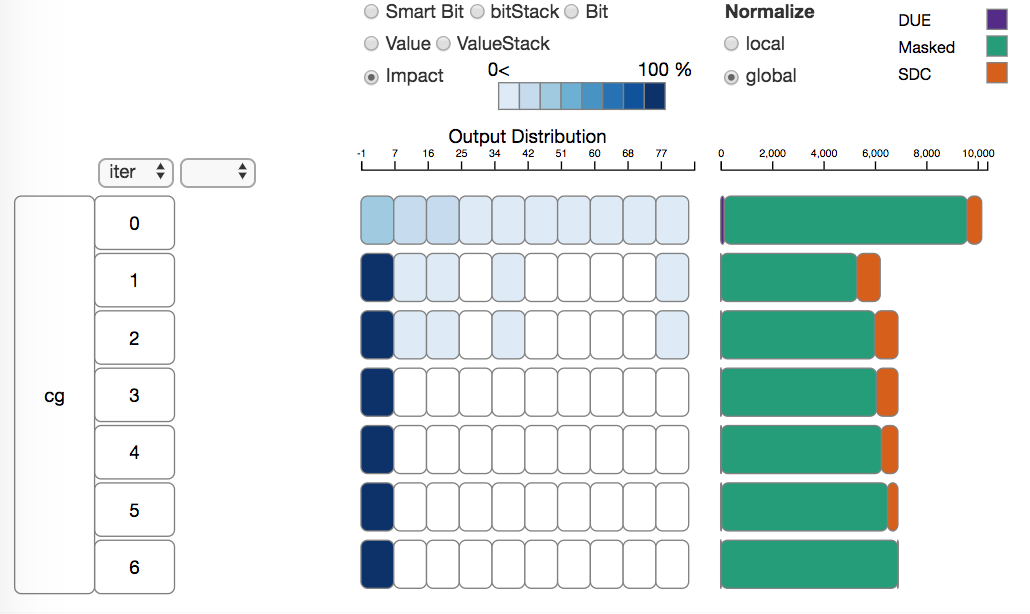


Figure 10. How different iterations will affect the error injection result.

Conclusion

In this research, we design a visualization tool, SpotSDC, to help domain experts understand the impact of silent data corruption on an HPC computation kernel. We demonstrate our tool's usability on two computation kernels, conjugate gradient and fast Fourier transform. Spot SDC helps domain experts understand the impact of a bit flip on the different levels of the program and provides more insight into how different bits of a variable flip may impact the outcome of the program. At the same time, Spot SDC also enables users to study the sample size of the random fault injection and determine the sample space.