Visilence: An Interactive Visualization Tool for Error Resilience Analysis

Shaolun Ruan[1], **Yong Wang**[2], **Qiang Guan**[1]
1. Kent State University, USA
2. Singapore Management University, Singapore

Introduction

As the HPC systems keep scaling up, the chance of the systems encountering soft errors also increases. Though many soft errors can be detected and corrected by hardware-level mechanisms, some errors escape these mechanisms and further propagate to the application-level. However, error resilience analysis for HPC applications is often known as a "Black Box" analysis: the user can estimate the resilience characteristics via fault injection to an application, which usually lacks explainability on a case-by-case basis. The conventional preceding studies rarely analyze error propagation and resilience through visualization methods and symbols turning inflexible HPC programs trace data into graphical representations and providing interactive analysis modes. We propose a novel control-flow based visualization tool to explore the error resilience of HPC applications. Furthermore, we showcase the error propagation pattern along with the basic blocks of an example faulty run of CoMD and demonstrate the usage of *Visilence* to identify the critical sections of the applications.

Visilience

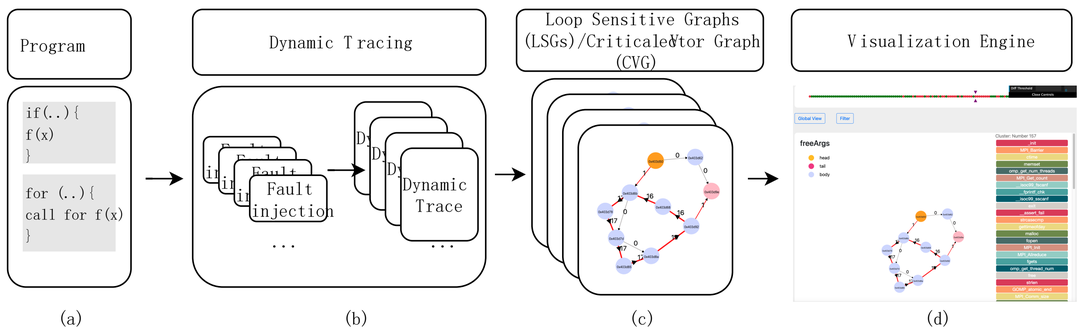

Figure 1: An overall workflow of *Visilence*. (a) Binary code: the input of ResilienceVis; (b) Dynamic tracing part contains both fault injection and tracing; (c) Loop Sensitive graph (LSG) generated from the dynamic traces and Critical Vector Graph (CVG) generated based on the accumulation of multiple LSGs; (d) Visualization engine shows the results to users visually and interactively.

Fig. 1 shows an overall workflow of proposed tool *Visilence*. At a high level, *Visilence* needs three levels of abstractions: (a) a model that can keep the static and dynamic program states, (b) a format to allow systematic analysis of the program states, and (c) a visualization tool that offers a friendly interface to identify the code regions that are sensitive to the errors for the users. We define Loop Sensitive graph (LSG) generated from the dynamic traces and Critical Vector Graph (CVG) generated based on the accumulation of multiple LSGs. The workflow of *Visilence* proceeds as follows: *(i)*, it takes an HPC program as input and conducts a statistic fault injection campaign on the application to generate a set of dynamic execution traces; *(ii)*, it creates LSGs/CVGs based on the obtained dynamic traces of the application, and *(iii)* it implements a novel visualization system that takes the LSGs/CVGs as the data source and provides a fine-grained representation of error propagation and resilience characteristic for the application.

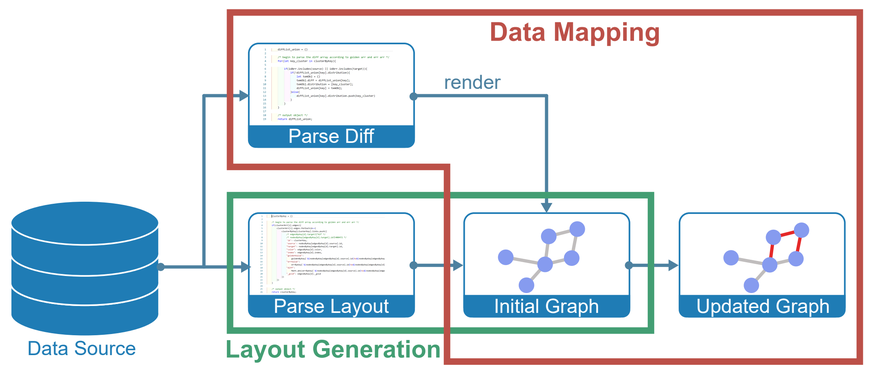

Figure 2: The visualization workflow of *Visilence*. The workflow of our visualization system consists of two stages.

We implement a user-friendly interface to visualize error propagation and functions interactively (Fig. 3). The interface consists of four parts:

- **Function View (Fig. 3a)** is a sequence of functions which are represented by dots. These functions are placed in the order of where they are defined. A green dot means it matches exactly like the golden run's, or it would be rendered in red when they are different in weights. The triangle on the sequence is a marker labeling the function where the fault is injected.
- **Graph View (Fig. 3b)** shows the Loop Sensitive Graph/Critical Vector graph. The vertices of the graph are basic blocks and the *head* (in yellow) and *tail* (in red) nodes are the entry and exit of the function respectively. The edges represent the connections between two basic blocks in the CFG, and the weights are the absolute values between the faulty traces and golden runs. The edge is gray when its weight is zero and is red otherwise. There are two options above: Global view and Filter.
- **Weight Threshold (Fig. 3c)** is used to filter the edges. When we slide the bar in Weight Threshold, the value would be adjusted, and the edges with smaller weights below the threshold would be assigned into gray.
- **Function List (Fig. 3d)** lists all the functions in the program with specific name in the same order in **Function View**. We can click on it to select the function to be shown in **Graph View**.

Case Study

Error Propagation Visualization

When soft errors occur in the running process of the program, this error may affect the subsequent control flow. Our tool can intuitively indicate how this error propagates.

Fig. 3 presents an example of LSG for the function 'setVcm_omp_fn.o' in benchmark program CoMD. The function starts from the 'head' basic block $'0x407f80'$ and ends in the 'tail' basic block $0x4080a8$, in total 12 basic blocks. The weights are the difference in executed times between the golden run and the faulty run. The biggest difference in this function is 351 on the edges from basic block $0x408000$ to $0x408030$. The path from the basic block $'0x408000'$ to $'0x408030'$ maps to the source code of 'initAtoms.c' at Lines 126 to 129 inside a for loop. We observed that 64 functions were affected by the injected fault.

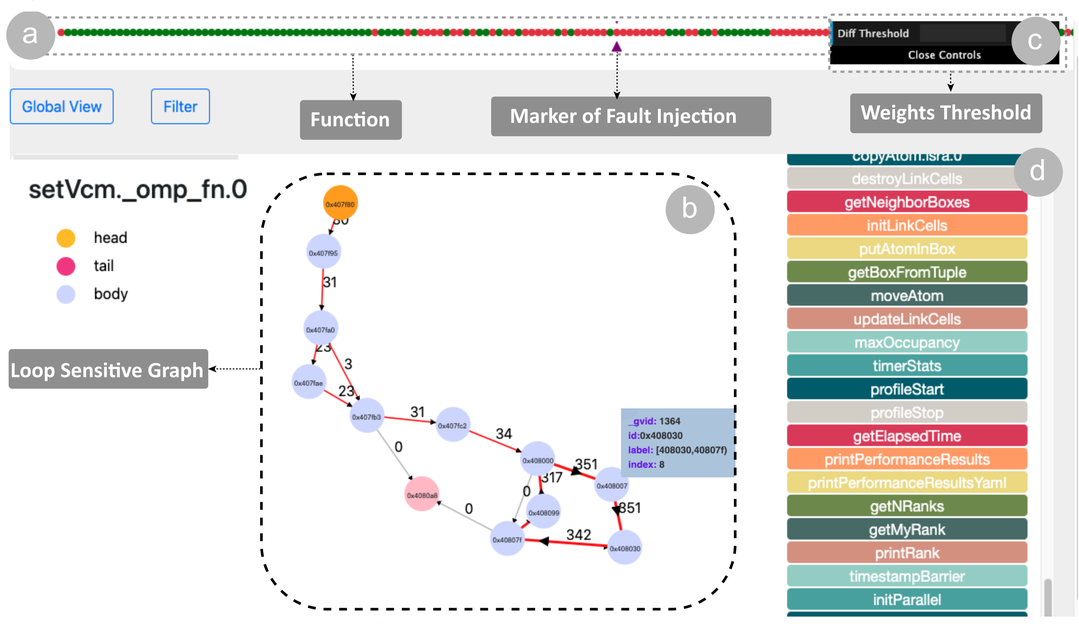

Figure 3: The interface of *Visilence*.

Weight Thresholding

Fig. 4 illustrates an example use of this functionality. In Fig. 4 (a), all the weights on the edges range from 0 to 1000 in this LSG. When we set the weight threshold to 100, as shown in Fig. 4b, the edges with a weight less than 100 automatically become gray and no longer within the scope of our analysis of resilience. We can see this works for different values of the threshold.

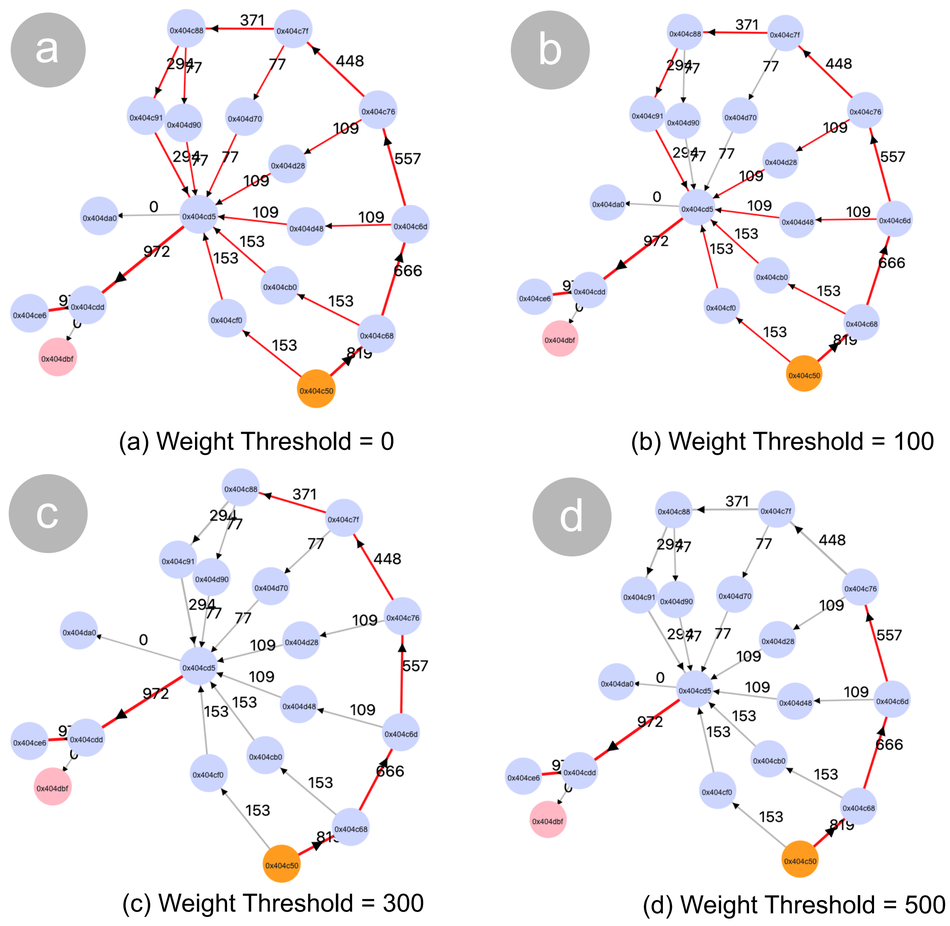

Figure 4: Weight Threshold examples. Weight Threshold is used to eliminate edges whose weight is lower than the threshold. From (a) to (d), the threshold is set to 0, 100, 300, 500; correspondingly, the CVG (21 basic blocks) is split into two disconnected red zones (2 basic blocks and 3 basic blocks).

This functionality brings convenience for the users when there is a constraint in resources to protect the program from being affected by errors. When the resources are limited, our tool can help the user visually prioritize the choices to protect code regions that differ the most from the golden run, i.e. more sensitive to errors.

Conclusion

We proposed *Visilence*, a control-flow graph based visualization tool for error resilience analysis, which provides human analysts with detailed facets of error propagation for further decision making. *Visilence* addresses the issue of understanding how the applications are affected by the errors via a graph-based abstraction to represent the affected program states and the reason for the error propagation across different error scenarios.