Visilence: An Interactive Visualization Tool for Error Resilience Analysis

Shaolun Ruan*
Kent State University

Yong Wang†
Singapore Management University

Qiang Guan‡
Kent State University

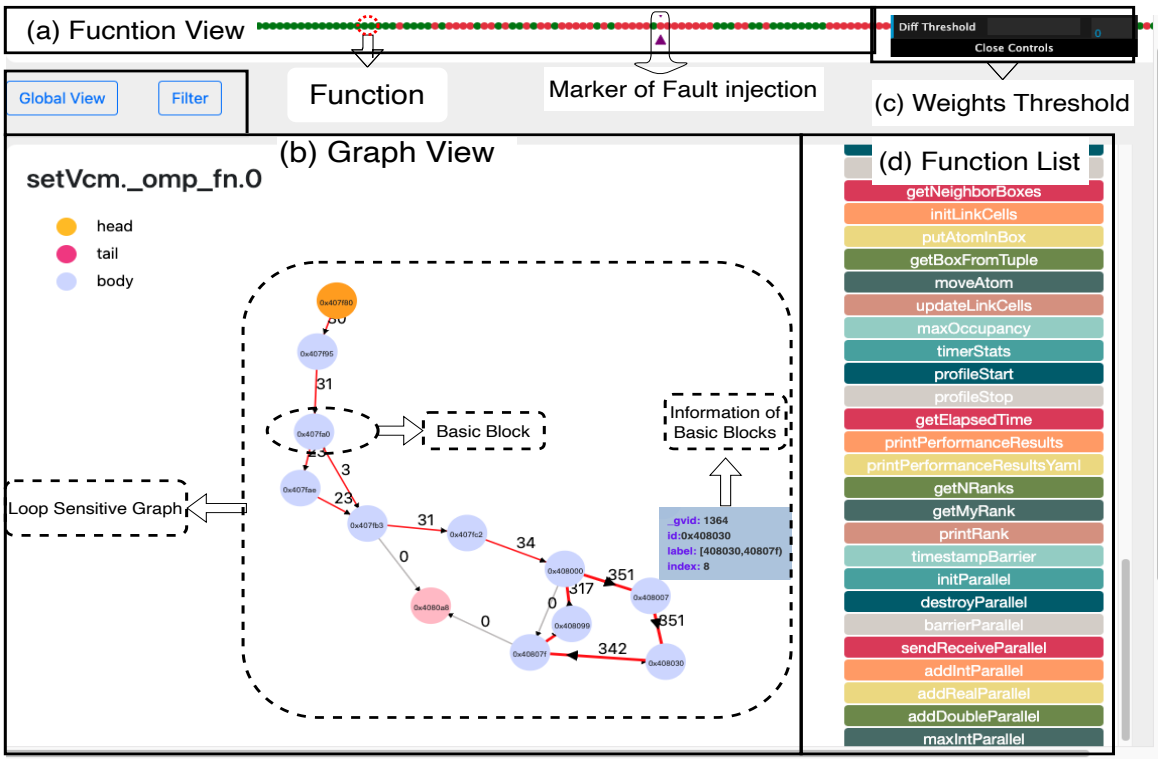

Figure 1: The interface of Visualization Engine. (a) Function view is a series of dots at top represent the functions. (b) The graph is shown in the Graph view and the nodes are basic blocks. (c) Weight threshold is used to set the weight threshold (d) The functions with specific names are listed in Function List.

Abstract

Soft errors have become one of the major concerns for HPC applications, as those errors can result in seriously corrupted outcomes, such as silent data corruptions (SDCs). Prior studies on error resilience have studied the robustness of HPC applications. However, it is still difficult for program developers to identify potential vulnerability to soft errors. In this paper, we present *Visilence*, a novel visualization tool to visually analyze error vulnerability based on the control-flow graph generated from HPC applications. *Visilence* efficiently visualizes the affected program states under injected errors and presents the visual analysis of the most vulnerable parts of an application. We demonstrate the effectiveness of *Visilence* through a case study.

Index Terms: Human-centered computing—Visualization—Visualization techniques—Treemaps; Human-centered computing—Visualization—Visualization design and evaluation methods

*e-mail: haywardryan@foxmail.com
†e-mail: yongwang@smu.edu.sg
‡e-mail: qguan@kent.edu

1 Introduction

As the HPC systems keep scaling up, the chance of the systems encountering soft errors also increases [5]. 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 [2], which can lead to a failure of applications and even serious outcomes such as silent data corruptions (SDCs). Prior studies [6, 7] have shown that the impact of errors is not uniformly distributed, and the likelihood of causing SDC is also different. Soft errors affect different states of an application. It indicates that understanding how an error occurring on a particular program state affects the outcome of an application, which may help developers add extra protection in development, e.g., duplication of variables or instructions.

However, resilience analysis for HPC applications is often known as a "Black Box" analysis: the user can estimate the resilience characteristics via fault injection [3] to an application, which usually lacks explainability on a case-by-case basis. For example, SpotSDC [4] visualizes error propagation in programs through fault injection methods, whereas the targeted vulnerable code regions and functions in SpotSDC require the user's expertise, which is not appropriate for general developers. 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.

2 *Visilence*

We roughly categorize the introduction of *Visilence* into two categories: overall workflow of *Visilence* and generation pipeline of visualization.

2.1 Overall workflow of *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.

2.2 Generation Pipeline of Visualization

Visilence has two modules, namely the function selecting module and the graph module, to support the collaborative design of basic-block like visualization. The pipeline is shown in Fig. 2. The visualization system has two separated stages for resilience graph generation, namely the layout simulation and the anomaly mapping.

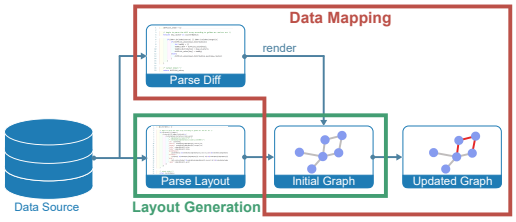

Figure 2: The workflow of our visualization system

We implement a user-friendly interface to visualize error propagation and functions interactively (see Fig. 1). The interface consists of four parts:

- **Function View (a)** 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 (b)** 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 of the different executed times 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 (c)** 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 (d)** 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**.

3 Case Study

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. 1 shows the error propagation pattern along with the basic blocks of an example faulty run of CoMD [1]. The series of dots at the top represents all the 157 functions of CoMD. The green dot indicates that the LSG generated for that function is consistent with the golden run, while the red dot indicates that they are inconsistent. The "marker of fault injection" indicates that the fault was injected in that function.

Fig. 1 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.

4 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.

References

[1] Comd. `http://www.exmatex.org/comd.html`.

[2] H. Cho, S. Mirkhani, C.-Y. Cher, J. A. Abraham, and S. Mitra. Quantitative evaluation of soft error injection techniques for robust system design. In *Proceedings of the 50th Annual Design Automation Conference*, pp. 1–10, 2013.

[3] Q. Guan, N. DeBardeleben, S. Blanchard, and S. Fu. F-SEFI: A fine-grained soft error fault injection tool for profiling application vulnerability. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*, pp. 1245–1254. IEEE Computer Society, 2014. doi: 10.1109/IPDPS.2014.128

[4] 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.

[5] 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*, 28(2):129–173, 2014.

[6] V. Sridharan and D. R. Kaeli. Eliminating microarchitectural dependency from architectural vulnerability. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, pp. 117–128. IEEE, 2009.

[7] B. Wibowo, A. Agrawal, and J. Tuck. Characterizing the impact of soft errors across microarchitectural structures and implications for predictability. In *2017 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 250–260. IEEE, 2017.