Visual Analysis on the Resilience of HPC Applications

Hailong Jiang[1], Shaolun Ruan[2], Bo Fang[3], Kevin Barker[3], Yong Wang[2], Ang Li[3], and Qiang Guan[1]

[1]Kent State University
[2]Singapore Management University
[3]Pacific Northwest National Laboratory

***Abstract*—A large body of approaches has been proposed to analyze the resilience of HPC applications. However, existing studies rarely address the challenges of the analysis result perception. Specifically, resilience analysis techniques often produce a massive volume of unstructured data, making it difficult for programmers to conduct the resilience analysis due to non-intuitive raw data. Furthermore, different analysis models produce diverse results with multiple levels of details, which may create hurdles to compare and explore the resilience of HPC program execution. To this end, we present VISILIENCE, an interactive VISual resILIENCE analysis framework to allow programmers to facilitate the resilience analysis of HPC applications. In particular, VISILIENCE leverages an effective visualization approach Control Flow Graph (CFG) to present a function execution. In addition, three widely-used models for resilience analysis (i.e., Y-Branch, IPAS, and TRIDENT) are seamlessly embedded into the framework for resilience analysis and result comparison. Case studies have been conducted to demonstrate the effectiveness of our proposed framework VISILIENCE.**

I. INTRODUCTION

As the HPC systems keep scaling up, the chance of soft error occurrence also increases [1]. These errors may cause applications to fail and serious outcomes such as silent data corruptions (SDCs) and crashes. A profound understanding of resilience could help programmers build a more resilient program and tell the end-user how to protect the vulnerable sections. A large amount of studies leverage applications' inherent fault-masking abilities to devise cost-effective SDC detection and recovery approaches, such as: TRIDENT [2], IPAS [3] and Y-Branch [4].

However, to directly adopt the results of the approaches for efficiently conducting application-specific error detection and correction remains challenging. Some fundamental gaps exist:

- Instruction level information is hard to follow.
- The resulting resilience characteristics of a series of program states can be scattered, lacking a holistic view for the users.
- It needs sizable extra effort to post-analyze the results from different resilience frameworks if one wants to lift the limitation of a single framework and makes a comprehensive assessment of an application's resilience using multiple frameworks simultaneously.

To this end, we propose **VISILIENCE**, an interactive **VIS**ual re**SILIENCE** analysis framework in this paper.

- VISILIENCE supports three resilience analysis models and enables the interpretable resilience analysis results between different analysis models through several human-computer interactions that help users understand the resilience data.
- The VISILIENCE is built on Control Flow Graph (CFG), which can accommodate the information from instruction level to function level. VISILIENCE can be combined with other static analysis tools, which use CFGs. The resilience analysis outcomes based on CFG can directly guide the compiler optimizations.

II. SYSTEM DESIGN

Figure 1 shows the main components and the workflow of VISILIENCE. VISILIENCE proceeds as follows: *(A)*, first, it takes an application as input and conducts resilience analysis on the application based on three resilience analysis models: TRIDENT [2], IPAS [3] and Y-Branch [4]. These resilience analyses are on different levels, and the results are in different formats; *(B)*, Then VISILIENCE encodes the result into a unified format (in Section II-B) as an input of the Visualization Engine; *(C)* Visualization Engine take these data as input and visualize these data on control flow graph of this program.

A. Resilience Analysis

We implemented three models proposed by previous researchers to analyze resilience on the basic-block level and instruction level:TRIDENT [2], IPAS [3] and Y-Branch [4].

B. Visual Encoding

The three analysis models above analyze resilience on different levels and output three data formats.

TRIDENT calculates the probability of SDC propagation from one basic block to another. VISILIENCE encodes the *"label"* of the *"edge"* be the probability of SDC from the output of TRIDENT. Here in TRIDENT mode, the default weights of edges would be covered by the probability of SDC. Visualization Engine adheres the value to each edge and colours the edge black if the *"value"* of it is "0"; otherwise, it is red as shown in Figure 2 (A). More details are introduced in Use Case (Section III).

IPAS classifies instruction as non-SOC-generating instruction or SOC-generating instruction. Data Transformer calculates the SOC-generating-instruction rate of each basic block using:

$$R_{SOC} = \frac{N_{SOC}}{N_{Inst.}} \tag{1}$$

where N_{SOC} is the number of SOC-generating instructions in the basic block, $N_{Inst.}$ is the number of instructions in

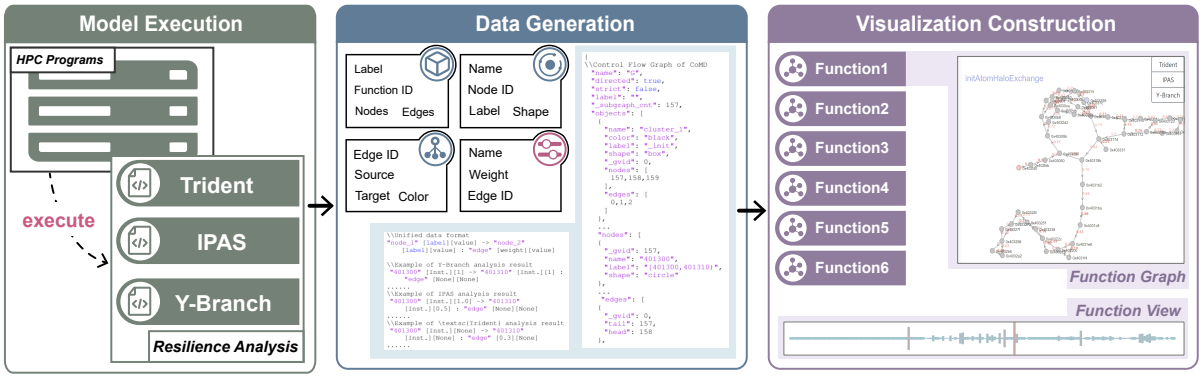

Figure 1: An overall overview of VISILIENCE.

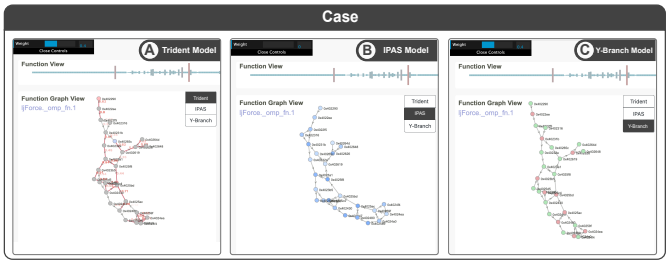

Figure 2: Use cases of VISILIENCE under three models: (A) TRIDENT, (B) IPAS, and (C) Y-Branch.

the basic block, and R_{SOC} is the SOC-generating-instruction rate of the basic block. Then Data Transformer sets the R_{SOC} as the "$value$" of each basic block and passes it to the Visualization Engine. The darkness of the node will be adjusted by according the "$value$" as shown in Figure 2 (B).

Y-Branch asserts a conditional basic block as a Y-Branch or not; the *"value"* of the *"node"* would be set as "1" when it is not Y-Branch and "0" in contrast. The node of basic block would be marked *"red"* when "$value$" is "1" and *"green"* when "$value$" is "0" as demonstrated in Figure 2 (C).

C. Visualization Construction

The VISILIENCE takes the CFG json file and the unified resilience analysis result as inputs and shows us an interactive visual interface. It first draws a layout of CFG, then maps the analysis result onto it.

III. CASE STUDY

In this section, we demonstrate the usage of Visilience on CoMD [5] benchmark.

As shown in Figure 2, there is a sequence of bars representing the functions in CoMD at the top. These functions are placed in the order of where they are defined. The heights of the bars are positively related to the number of edges.The more nodes in a function, the darker the bar will be. Click on a bar, and its CFG will be displayed in Function Graph View. The vertices of the graph are basic blocks. The hexadecimal number next to the node is the basic block's entry instruction address. The edges represent the connection between two basic blocks in the CFG, and the program executes in the arrow's direction.

Figure 2 (A) shows interface of TRIDENT mode. The weights on the edges are the SDC propagation possibilities between basic blocks. The weight threshold bar on the very left top can be slid from 0 to 1. The edges with weights smaller than the threshold are assigned into gray; in contrast, the edges with weights larger than the threshold are highlighted in red. As shown in Figure 2 (A), the weight threshold is set to 0.4. This function can help the user visually prioritize the choices to protect code regions with the highest SDC probability when the protection resources are limited.

Figure 2 (B) shows the interface of IPAS mode. The darkness of the nodes represents the SOC-generating-instruction rate calculated by Function 1: the darker the colour, the higher the rate. One can easily tell the nodes with higher SOC-generating-instruction rate so that one can determine the minimal set of instructions that require duplication to avoid SOC saving runtime overhead. For example, the basic block "$0x40231b$" is darker than "$0x40232a$", which means the basic block "$0x40231b$" has a higher SOC-generating instruction rate.

Figure 2 (C) shows interface of Y-Branch mode. Basic blocks in Y-Branch node are green or red, representing Y-branch and non-Y-Branch. If an error occurs at a red node, the final result will be affected by this error, such as "$0x40231b$"; in contrast, if this error occurs in a green node, it would be masked, such as "$0x4023a1$".

IV. CONCLUSION

In this paper, we present VISILIENCE, a visual resilience analysis framework to show the resilience analysis results to programmers in an intuitive way. VISILIENCE takes the Control Flow Graph as a layout and maps the resilience analysis data on it. VISILIENCE conducts three resilience analysis models and encodes these data into a unified data format, and visualizes the data into an interactive interface. The Visualization Engine provides several human-computer interactions, which help the users understand the data better. Multiple case studies have been conducted to demonstrate the effectiveness of VISILIENCE.

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] 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)*, June 2018, pp. 27–38.

[3] I. Laguna, M. Schulz, D. F. Richards, J. Calhoun, and L. Olson, "Ipas: Intelligent protection against silent output corruption in scientific applications," ser. CGO 2016, 2016.

[4] N. Wang, M. Fertig, and S. Patel, "Y-branches: when you come to a fork in the road, take it," *Parallel Architectures and Compilation Techniques, 2003. PACT 2003. Proceedings. 12th International Conference on*, 2003.

[5] P. Cicotti, S. M. Mniszewski, and L. Carrington, "An evaluation of threaded models for a classical md proxy application," in *Hardware-Software Co-Design for High Performance Computing (Co-HPC), 2014*, Nov.