VISILIENCE: An Interactive Visualization Framework for Resilience Analysis using Control-Flow Graph

Hailong Jiang*
Kent State University
hjiang13@kent.edu

Shaolun Ruan*
Singapore management university
haywardryan@foxmail.com

Bo Fang
Pacific Northwest National Laboratory
bo.fang@pnnl.gov

Yong Wang
Singapore management university
yongwang@smu.edu.sg

Qiang Guan
Kent State University
qguan@kent.edu

***Abstract*—Soft errors have become one of the main concerns for the resilience of HPC applications, as these errors can cause HPC applications to generate serious outcomes such as silent data corruption (SDC). Many approaches have been proposed to analyze the resilience of HPC applications. However, existing studies rarely address the challenges of analysis result perception. Specifically, resilience analysis techniques often produce a massive volume of unstructured data, making it difficult for programmers to perform resilience analysis due to non-intuitive raw data. Furthermore, different analysis models produce diverse results with multiple levels of detail, which can create obstacles to compare and explore the resilience of the 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. Furthermore, three widely used models for resilience analysis (i.e., Y-Branch, IPAS, and TRIDENT) are seamlessly integrated into the framework for resilience analysis and result comparison. Multiple case studies have been conducted to demonstrate the effectiveness of our proposed framework VISILIENCE.**

***Index Terms*—Error Resilience, Visualization, Visual Analytics, Control Flow Graph**

I. INTRODUCTION

Transient hardware faults, typically caused by particle strikes [1] are significant concerns in High Performance Computing (HPC) systems. As HPC systems continue to scale up, the chance of occurrence of soft errors also increases [2]. Although soft errors can be detected and corrected by hardware and system-level mechanisms, such as ECC or checkpointing/restart schemes, some can escape these mechanisms and propagate to applications [3]. These errors may cause applications to fail and serious outcomes such as silent data corruptions (SDCs) and crashes.

SDCs need more attention due to their no-symptom effect [4]–[10]. Traditional fault tolerance methods protect programs against data corruption through Dual Modular Redundancy (that is, DMR, for error detection) and Triple Modular Redundancy (that is, TRM, for error recovery) [11]. However, a full-state DMR or TMR might not be feasible for HPC applications as such applications are significantly time-consuming and resource-intensive. Therefore, a large body of studies leverage applications' inherent fault-masking abilities to devise cost-effective SDC detection and recovery approaches [4], [7], [12], [13]. For example, Wang et al. [14] use random fault injections to determine whether a branch is outcome-tolerant, called Y-Branch. They randomly choose a branch and force it down the alternate path when the chosen branch is encountered at run-time. They test all branch instances and output a binary result of each branch as Y-Branch or non-Y-Branch. I. Laguna et al. proposed a machine learning model, IPAS [15], to find the instructions that must be protected to avoid SOC (the term SOC is explained in Section III). IPAS classifies an instruction as a SOC-generating instruction or non-SOC-generating instruction based on its extracted features. G. Li et al. [8] proposed a tool named TRIDENT to predict both the overall SDC probability of a given program and the SDC probabilities of individual instructions without fault injection. Trident outputs the SDC probability values of instructions.

However, directly adopting 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. For example, IPAS identifies the SOC-generating instructions but does not explicitly help users scope the instructions;
- The resulting resilience characteristics of a series of program states can be scattered, lacking a holistic view for the users. For example, TRIDENT models the SDC probability of state transitions where the state dependency is not presented in the final output.
- Large-scale HPC applications produce a massive volume of dynamic program states, which could lead to enor-

*These authors contributed equally to this work and should be considered co-first authors.

Thanks to the support partially from NSF #2217104 #2212465

mous efforts to classify and summarize the unstructured resilience-related data generated by those approaches.

- 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.

Therefore, a unified approach allowing an intuitive exploration for efficiently analyzing the results of multiple resilience approaches would be helpful to the dependability community. Visualization has been proven more effective for large-scale quantitative analysis than raw data exploration [16], [17]. A user-friendly interface that tailors widely-used resilience analysis models can provide comprehensive solutions for the users to understand the findings of those techniques and cross-reference the program states/instructions/code pieces suggested by them.

To this end, we propose **VISILIENCE**, an interactive **VIS**ual re**SILIENCE** analysis framework in this paper. VISILIENCE takes Control Flow Graph (CFG) as a visualization layout and visualizes the resilience properties on it. A CFG comprises functions, basic blocks, instructions, and program execution paths and shows the dependency between all these elements. Also, the data size of CFG is hugely smaller than dynamic program states for HPC programs. These properties of CFG narrow the "fundamental gaps" introduced above. Three widely used models for resilience analysis (i.e., Y-Branch, IPAS, and TRIDENT) are seamlessly embedded into our framework for resilience analysis and result comparison. These resilience analyses are on different levels, and the results are in different formats. VISILIENCE encodes the analysis results into a unified data format as an input of the Visualization Engine. The unified data format of our visualization platform allows programmers to use multiple resilience analysis models simultaneously. The Visualization Engine outputs an interactive visual interface showing the resilience analysis results.

To the best of our knowledge, VISILIENCE *is the first visualization tool for the resilience analysis that efficiently supports large-scale HPC applications via control flow graph.*

In summary, the contributions of this paper are as follows:

- We propose a novel interactive visualization framework, VISILIENCE. It 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 Visualization Engine is based on the Control Flow Graph (CFG), which can accommodate information from the instruction level to the 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.
- We propose a unified JSON data format to Visualize Engine in VISILIENCE. The unified data interface can bridge the gap of understanding the differences between resilience analysis models.

The rest of this paper is organized as follows: Section II introduces the background information, including Control Flow Graph and definitions related to resilience; Section III presents VSILIENCE's system design and the implementation of each component in the analysis framework. In Section IV we introduce the design of the visualization system. We showcase the results of visualizations for the benchmark applications in Section V, and the related studies are discussed in Section **??**. We present our conclusion in Section VII.

II. BACKGROUND

In this section, we describe the background of the Control Flow Graph, the fault model, and the terms related to resilience.

A. Fault Model

We consider soft errors, i.e., transient faults that escape from hardware protection and propagate to the application level. These errors manifest themselves as bit-flips in registers and memory locations. The application consumed the corrupted registers and memory cells. We focus on single-bit errors, not multi-bit errors. The reason behind this is that (1) single-bit errors are the most common soft errors—multi-bit errors rarely happen in large-scale systems [18]; (2) in many cases, multi-bit errors have a similar impact on the application as single-bit errors [19].

B. Terms and Definitions

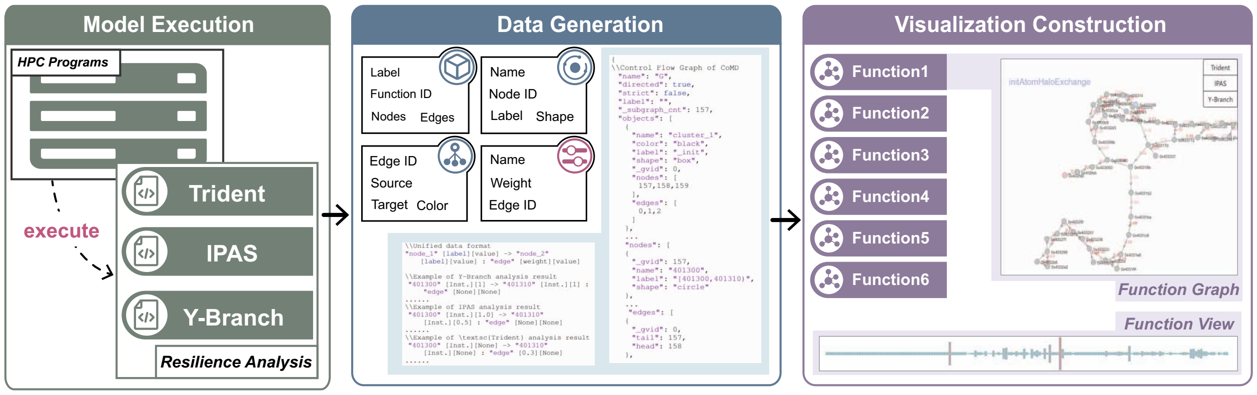
If the program output matches that of the error-free execution even though a fault occurred during its execution, it is called **Benign**; in contrast, if the program output does not match, it is called **Silent Data Corruption (SDC)**. If the OS terminates the program due to the error, it is called **Crash**.

III. SYSTEM DESIGN

This section describes the design of VISILIENCE. We first introduce the overall architecture (Figure 1), then explain each component's design details.

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 a resilience analysis on the application based on three resilience analysis models: Y-Branch [14], IPAS [15] and TRIDENT [8]. 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 III-C) 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.

In the rest of this section, we present the details of each component.



```
\\Unified data format
"node_1" [label][value] -> "node_2"
    [label][value] : "edge" [weight][value]

\\Example of Y-Branch analysis result
 "401300" [Inst.][1] -> "401310" [Inst.][1] :
     "edge" [None][None]
......
\\Example of IPAS analysis result
 "401300" [Inst.][1.0] -> "401310"
     [Inst.][0.5] : "edge" [None][None]
......
\\Example of \textsc{Trident} analysis result
 "401300" [Inst.][None] -> "401310"
     [Inst.][None] : "edge" [0.3][None]
......
```

```
{
\\Control Flow Graph of CoMD
 "name": "G",
 "directed": true,
 "strict": false,
 "label": "",
 "_subgraph_cnt": 157,
 "objects": [
  {
   "name": "cluster_1",
   "color": "black",
   "label": "_init",
   "shape": "box",
   "_gvid": 0,
   "nodes": [
    157,158,159
   ],
   "edges": [
    0,1,2
   ]
  },
  ...
  "nodes": [
  {
   "_gvid": 157,
   "name": "401300",
   "label": "[401300,401310)",
   "shape": "circle"
  },
  ...
  "edges": [
  {
   "_gvid": 0,
   "tail": 157,
   "head": 158
  },
```

Fig. 1. An overall overview of VISILIENCE. *(A)* three resilience analysis models in the Resilience Analysis part (Section III-A); *(B)* Data Generation part generates CFG data (Section III-B), and encodes the resilience analysis results into a unified format (Section III-C); *(C)* Visualization Engine takes the formatted data and CFG data as input and outputs an interactive visual interface of the resilience analysis results (Section IV).

A. Resilience Analysis

For programmers, understanding the resilience of a program on multiple levels is essential: *(1)* it can instruct economical protection of vulnerable sections of the program; *(2)* it also can help the programmers build a resilient program.

We implemented three models proposed by previous researchers to analyze resilience at the basic block and instruction levels. Here, we introduce the implementation of these models in our work:

- **Y-Branch:**, *Y-branches* [14] is built on a characterization of outcome-tolerant branch instances. Using statistical sampling, the Y-Branch can search out the dynamic and mispredicted branches that do not affect correct program behavior when forced down the incorrect path. The Y-Branch model uses values "0" and "1" to represent Y-Branch and non-Y-Branch, respectively.
- **IPAS:** Due to natural error masking, only a subset of SDC errors actually affects the output of scientific codes–these errors **S**ilent **O**utput **C**orruption errors are called SOC errors [15]. IPAS has trained a classifier model to take the instruction features as input and outputs the class of this instruction: *Class 0:* non-SOC-generating instruction or *Class 1*: SOC-generating instruction. Applying the IPAS model to all the instructions in a basic block, we can obtain the number of SOC-generating instructions in this basic block and calculate the SOC-generating instruction rate.
- **TRIDENT:** TRIDENT [8] estimates the SDC probability of individual instruction and the entire program without performing any FIs. TRIDENT takes three inputs:(1) The program code compiled in the LLVM IR, (2)a program input to execute the program and obtain its execution profile, and (3) the target instruction(s). TRIDENT automatically computes (1) the SDC probability of individual instructions and (2) the overall SDC probability of the program based on these inputs. TRIDENT has three submodels to abstract Static-instruction level, Control Flow level, and Memory level. We use the Control Flow level submodel to calculate SDC probabilities of dynamic instructions. Since there is no branch inside the basic blocks, the entry instruction domains all instructions inside the basic blocks, which means that all the instructions in the same basic block should have the same SDC probability except the exit instructions.

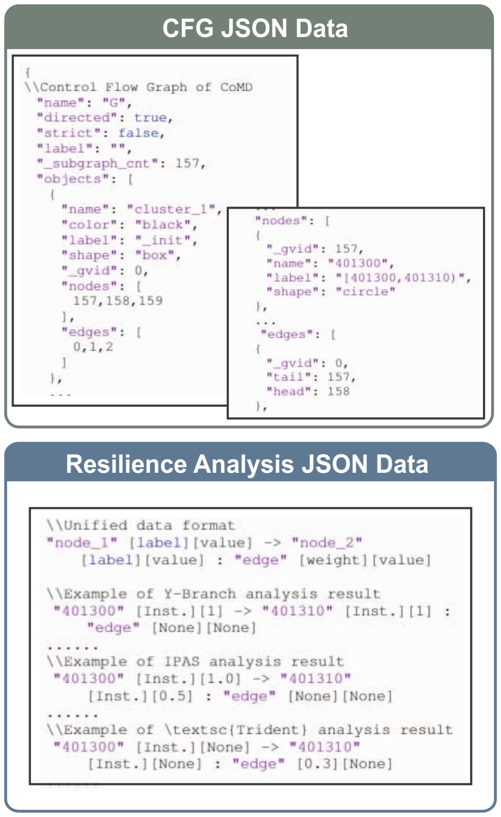
B. Benefits of using CFG representation

CFG is widely used in compiler optimizations and static analysis tools. VISILIENCE uses CFG as a visualization layout due to the following reasons:

- A CFG not only comprises functions, basic blocks, instructions, and execution paths of the program but also shows the dependency between all these elements. Accommodation of multiple-level information and dependency can help programmers understand resilience better.
- The data size of an HPC program CFG is hugely smaller than dynamic program states, which reduces the resilience analysis time and overhead.
- CFG is a graphical representation of a program. It naturally visualizes how execution traverses a program intuitively.
- Weights are added to the edges in the CFG so that one more data dimension could be represented. The dynamic iteration number of each edge in CFG is set as the default weight and labeled on it as shown in Figure 3 "CFG" diagram.

For example, The CFG JSON file of "CoMD" [20] in VISILIENCE is shown in Figure 2 and its diagram is in Figure 3. There are three kinds of elements *"objects", "nodes"* and *"edges"* representing *"functions", "basic blocks"* and *"control flows"* of CoMD, respectively. Each element in this file has a unique *"name"* and *"_gvid"*. Each *"objects"* contains the *"nodes"* inside it and the *"edges"* between them. The *"name"* of the *"nodes"* is the address of the entry instruction. The *"edges"* go from *"tail"* to *"head"*.

The CFG JSON file is generated using DCFG [21]. DCFG is built on "ParseAPI" and "InstructionAPI" of Dyninst [22] to



```
{
\\Control Flow Graph of CoMD
  "name": "G",
  "directed": true,
  "strict": false,
  "label": "",
  "_subgraph_cnt": 157,
  "objects": [
   {
     "name": "cluster_1",
     "color": "black",
     "label": "_init",
     "shape": "box",
     "_gvid": 0,
     "nodes": [
      157,158,159
     ],
     "edges": [
      0,1,2
     ]
   },
   ...
```

```
 ...
 "nodes": [
 {
  "_gvid": 157,
  "name": "401300",
  "label": "[401300,401310)",
  "shape": "circle"
 },
 ...
 "edges": [
 {
  "_gvid": 0,
  "tail": 157,
  "head": 158
 },
```

```
\\Unified data format
"node_1" [label][value] -> "node_2"
    [label][value] : "edge" [weight][value]

\\Example of Y-Branch analysis result
 "401300" [Inst.][1] -> "401310" [Inst.][1] :
     "edge" [None][None]
......
\\Example of IPAS analysis result
 "401300" [Inst.][1.0] -> "401310"
     [Inst.][0.5] : "edge" [None][None]
......
\\Example of \textsc{Trident} analysis result
 "401300" [Inst.][None] -> "401310"
     [Inst.][None] : "edge" [0.3][None]
```

Fig. 2. CFG json data and resilience analysis json data

produce the static control flow graph of a binary file. Dyninst is a binary instrumentation tool, performing static and dynamic analysis on binaries and processes.

C. *Visual Encoding*

The Three analysis models above analyze resilience on different levels and output three data formats. The Data Transformer part encodes the resilience analysis result to a unified format shown in Figure 2 and passes it to the Visualization Engine.

The first line in resilience analysis json data in Figure 3 shows the unified format of data. The *"node_number"* and *"edges"* are the same as those in Figure 2 (a). The *"label"* and *"value"* of each elements are the Data Interface between Data Transformer and Visualization Engine.

Y-Branch asserts a conditional basic block as a Y-Branch or not; the Data Transformer would set *"value"* of the *"node"* "1" when it is not Y-Branch and "0" in contrast. Visualization Engine will mark the node of basic block *"red"* when *"value"* is "1" and *"green"* when *"value"* is "0" as demonstrated in Figure 3. This paper takes the regular basic blocks with no branch as Y-Branch blocks.

IPAS classifies the 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 the basic block, and R_{SOC} is the SOC-generating-instruction rate of the basic block. Then, the Data Transformer sets the R_{SOC} as the *"value"* of each basic block and passes it to the Visualization Engine. VisualDiagram of centralization Engine will adjust the darkness of the node according to the *"value"* as shown in Figure 3.

TRIDENT calculates the probability of propagation of SDC 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 colors the edge black if the *"value"* of it is "0"; otherwise, it is red as shown in Figure 3. More details are introduced in the Use Case (Section V).

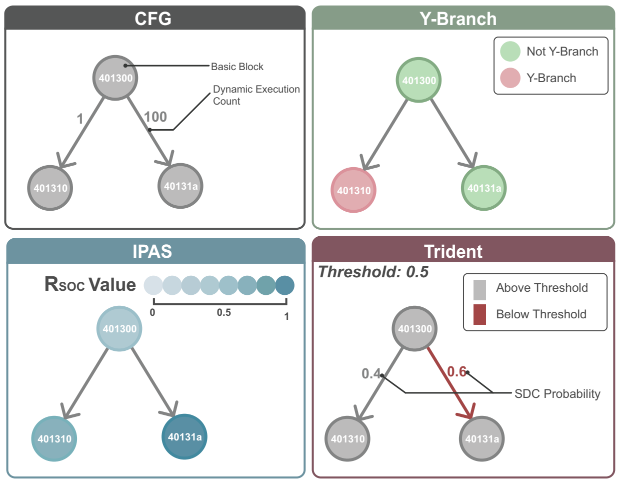


Fig. 3. Diagram of control flow graphs of three models.

D. *Visualization Engine*

The Visualization Engine of 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 and then maps the analysis result onto it. The Visualization Engine is described in more detail in the next section.

IV. VISUALIZATION ENGINE DESIGN

We proposed VISILIENCE, an interactive visualization framework for programmers to facilitate the resilience analysis workflow. The framework applies intuitive visual representation and effective user interactions to portray the error patterns of HPC applications. More specifically, we used Control Flow

Graph to visualize the interaction between basic blocks, taking model characterization (e.g., SDC probability and R_{SOC} value) into account. Furthermore, we leveraged a bar sequence as a system portal, present an overview of all functions in a certain model. The design principles, system workflow, and interface will be discussed in this section.

A. Design Challenges

To guide the design of our visualization engine, we summarize three primary design challenges:

DC1. Imitate the basic block interface to generate the layout. Although the sequence of basic blocks is available, a layout that resembles the basic block interface is still uncertain. That is because the existing graph layout algorithms for representing a divergent execution flow, such as tree-map and hierarchy, do not support the layout that includes ambiguous nodes and cycles within a stratified relationship; otherwise, multiple root nodes will be generated to break the hierarchy structure.

DC2. Enhance the scalability to support real-world data. We implement the layout generation process module in the front-end component, which handles both the generation and rendering tasks. In contrast, many entities will be parsed prior to the graph rendering. It is a tough job to parse and generate the position of each node simultaneously. Specifically, powerful hardware is needed when the velocity force *vx* and *vy* of each node are hard to iterate if the number of entities gets 5000 or more.

DC3. Visualize the connection status between different clusters. After our qualitative evaluation, the anomalous edges exist both within the cluster and the connection between adjacent clusters. The authoring system must handle the diff array between two different clusters prior to the generation of the graph layout and the re-rendering stage.

B. System Workflow

Our Visualization System has two modules, namely the function selecting module and the graph module, to support the collaborative design of basic block-like visualization. The processing pipeline is shown as Fig. 4. the visualization system

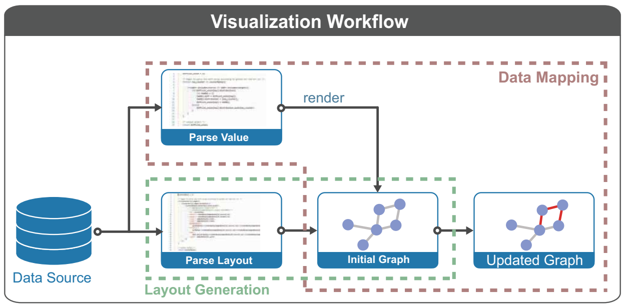


Fig. 4. The workflow of our visualization system incorporates two stages. The layout file will be handled and generated the information of nodes and edges including *vx* and *vy* simulation. Prior to the re-rendering of anomalous mapping.

has two separated stages for resilience graph generation, namely the layout simulation and the anomaly mapping.

Layout generation: by parsing the data source into the graph layout format, a force-directed simulation has been generated to control the zooming of each code block, which meets the requirement of **DC1**. Unlike stratified layouts, this approach produces a separated force simulation for each node's position instead of a global positioning function, which enables flexible interactions for users to explore the graph entities themselves. The weak point is the disorder of the code block's execution flow due to the missing of the root node. This is an interesting point worth exploring and we leave it to future work. To overcome **DC2**, we first handle the visual scalability of the node elements by separating the global code blocks into clusters based on the cluster label, which facilitates the simulation *forceManyBody()* between nodes to be more legible to positioned within the view.

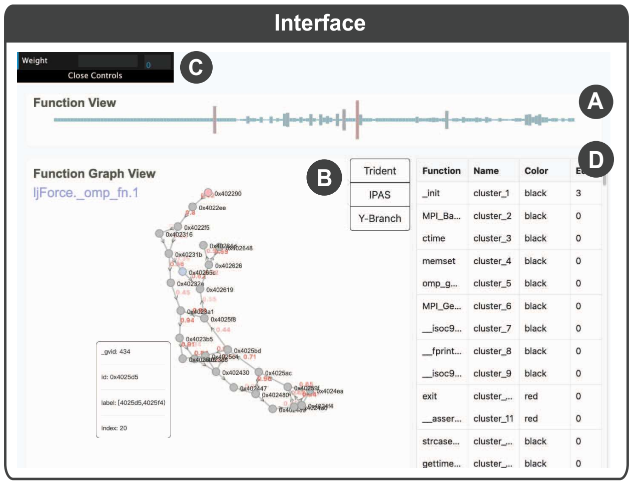
Data mapping: after parsing the diff output, both the cluster overview and graph module will be re-rendered to hold the weight from the trace data.

C. Interface

We implement a user-friendly interface to visualize the resilience analysis results. (see Figure 5). The interface consists of five parts:

- **Function View (A)** consists of a sequence of bars, where each bar depicts a certain function in a model. The number of edges in a certain function is encoded by each bar height, while the bar color represents the node number. Specifically, the more nodes in a function, the redder the bar. Function View is a portal for VISILIENCE, and the user can drill-down to the detailed graph by selecting the preferred function.
- **Graph View (B)** shows the Control Flow 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. There are three graph view options on the right corner: "Y-Branch", "IPAS", and "TRIDENT". Click on any one and the Graph View will display the analysis result of that one. The detailed information of each node would pop up when the mouse hovered on it, including its id, the entry/exit instruction of the basic block, and the order of this basic block in this function.
- **Weight Threshold (C)** is used to filter the edges. The value would be changed if we slide the bar in the Weight Threshold, and the edges with smaller weights below the threshold value would be assigned into gray. On the contrary, the edge will be rendered in red.
- **Function List (D)** lists all the function name in the program with specific name in the same order in **Function View**. The user can browse all functions via Function List before proceeding to the drill-down view. The parameters include the "name" of the function, its cluster sequence, color and edge numbers.

- **User Interaction** allows programmers to interact with the visualization engine and get a deep insight for the resilience analysis. First, the user can adjust the threshold value to launch the graph edge re-rendering. Second, with the hover of a graph node, the user can gain detailed information of a basic block, e.g., the label and shape attributes. Furthermore, flexible switch of the CFG is supported via interaction with the sequence bars.



_gvid: 434

id: 0x4025d5

label: [4025d5,4025f4)

index: 20

Function	Name	Color	E
_init	cluster_1	black	3
MPI_Ba...	cluster_2	black	0
ctime	cluster_3	black	0
memset	cluster_4	black	0
omp_g...	cluster_5	black	0
MPI_Ge...	cluster_6	black	0
__isoc9...	cluster_7	black	0
__fprint...	cluster_8	black	0
__isoc9...	cluster_9	black	0
exit	cluster_...	red	0
__asser...	cluster_11	red	0
strcase...	cluster_...	black	0
gettime...	cluster_...	black	0

Fig. 5. 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.

V. Case Study

In this section, we demonstrate the usage of Visilience on CoMD [20] benchmark.

At the top, there is a sequence of bars that represent the functions of CoMD. 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 6 (A) shows the partial control flow graph of function *ljForce_omp_fn.1*. The weights on the edges are the SDC propagation possibilities between basic blocks. For example, the weights on edge of "$0x40231b$"$->$ "$0x40232a$" is 0.66. That means The error effect in basic block "$0x40231b$ has a 66% possibility to affect the basic block "$0x40232a$". 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 6 (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 6 (B) shows the partial control flow graph of function *ljForce_omp_fn.1*. 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 6 (C) shows the partial control flow graph of function *ljForce_omp_fn.1*. 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 "$0x4023a1$".

VI. Discussion

A. Scalability

In the visualization process, the capacity of the virtualization engine in our tool has a limit that each function can accommodate up to 1000 basic blocks. For our experiments and benchmarks, the number of basic blocks is far less than 1,000. If we encounter any function with more basic blocks than the upper limit, we need to optimize the visualization engine. Specifically, the layout to present basic nodes is generated based on force-directed approach, leading to a bad performance of the visualization rendering if the number of basic block entities exceeds thousands.

B. Applicability

Besides, Visilience can be combined with other performance profile tools, such as HPCtoolkits [23]. HPCtoolkits employs binary-level measurement and analysis and associated with static and dynamic context. Similarly, HPCtoolkits can also profile out and extract the temporal pattern in performance and it profiles performance at the function level granularity. HPCtraceViewer provides layered profile reports and human-computer interaction to enable developers to explore and figure out the limitation of the application in performance.

VII. Conclusion

To protect the applications from soft errors is an essential while challenging task. A profound understanding of resilience on different levels is the building block to develop resilient programs and conduct economical protection. 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

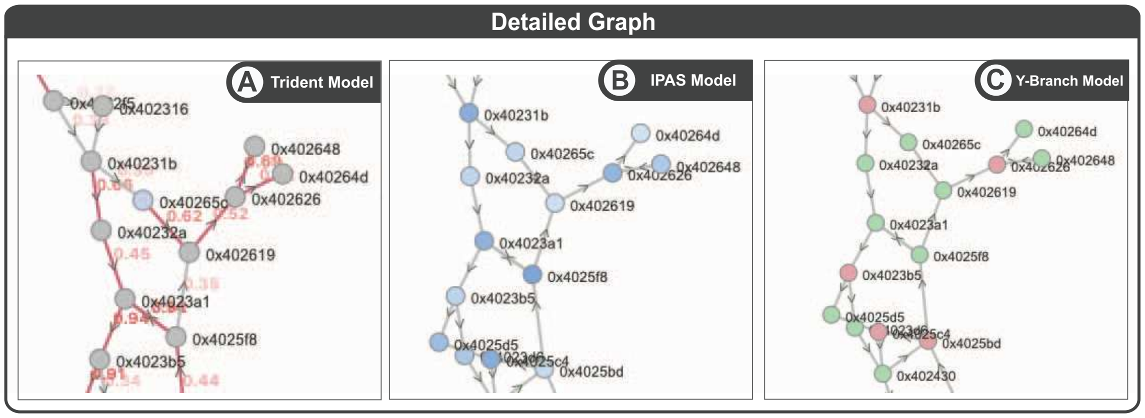


Fig. 6. Detailed graph of (A) TRIDENT, (B) IPAS, and (C) Y-Branch.

the data better. Multiple case studies have been conducted to demonstrate the effectiveness of VISILIENCE.

ACKNOWLEDGMENT

Thanks to the support partially from NSF #2217104 #2212465.

REFERENCES

[1] R. C. Baumann, “Radiation-induced soft errors in advanced semiconductor technologies,” *IEEE Transactions on Device and materials reliability*, vol. 5, no. 3, pp. 305–316, 2005.

[2] 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.

[3] H. Cho, S. Mirkhani, C.-Y. Cher, J. A. Abraham, and S. Mitra, “Quantitative evaluation of soft error injection techniques for robust system design,” *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, pp. 1–10, 2013.

[4] V. Sridharan and D. Kaeli, “Eliminating microarchitectural dependency from architectural vulnerability,” in *HPCA 2009.*, 2009, pp. 117–128.

[5] 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)*, Oct 2017, pp. 250–260.

[6] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin, “Measuring architectural vulnerability factors,” in *IEEE MICRO*, vol. 23, no. 6.

[7] B. Fang, Q. Lu, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, “epvf: An enhanced program vulnerability factor methodology for cross-layer resilience analysis,” in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), acceptance rate = 21%*, June 2016, pp. 168–179.

[8] 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.

[9] Q. Guan, N. BeBardeleben, P. Wu, S. Eidenbenz, S. Blanchard, L. Monroe, E. Baseman, and L. Tan, “Design, use and evaluation of p-fsefi: A parallel soft error fault injection framework for emulating soft errors in parallel applications,” in *Proceedings of the 9th EAI International Conference on Simulation Tools and Techniques*, 2016.

[10] R. Ashraf, R. Gioiosa, G. Kestor, R. F. DeMara, C.-Y. Cher, and P. Bose, “Understanding the propagation of transient errors in HPC applications.” *SC*, pp. 72–12, 2015.

[11] L. Chen and A. Avizienis, “N-version programminc: A fault-tolerance approach to rellablllty of software operatlon,” in *Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'.*, 1995, pp. 113–.

[12] 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*, 2021, pp. 362–373.

[13] 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.

[14] 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.

[15] 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.

[16] N. Gershon, S. G. Eick, and S. Card, “Information visualization,” *interactions*, vol. 5, no. 2, pp. 9–15, 1998.

[17] J. J. Van Wijk, “The value of visualization,” in *VIS 05. IEEE Visualization, 2005.* IEEE, 2005, pp. 79–86.

[18] L. Bautista-Gomez, F. Zyulkyarov, O. Unsal, and S. McIntosh-Smith, “Unprotected computing: A large-scale study of dram raw error rate on a supercomputer,” in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 645–655.

[19] B. Sangchoolie, K. Pattabiraman, and J. Karlsson, “One bit is (not) enough: An empirical study of the impact of single and multiple bit-flip errors,” in *2017 47th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*. IEEE, 2017, pp. 97–108.

[20] 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.

[21] PASSLAB, “Dcfg,” https://github.com/passlab/DCFG, 2016.

[22] W. R. Williams, X. Meng, B. Welton, and B. P. Miller, “Dyninst and mrnet: Foundational infrastructure for parallel tools,” in *Tools for High Performance Computing 2015*. Springer, 2016, pp. 1–16.

[23] X. Liu and J. Mellor-Crummey, “A data-centric profiler for parallel programs,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2503210.2503297