JIT Compiler Phase-by-Phase Visualization

Ellora Devulapally, Taft Harrell, and Katy Williams

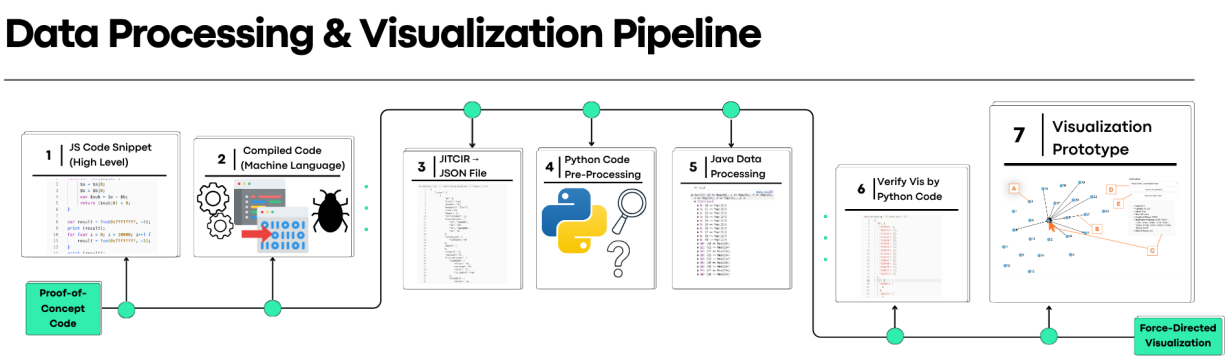

Fig. 1: Data Processing and Visualization Roadmap. JS proof-of-concept code (1) is compiled (2) and necessary information about intermediate representation is collected (3). Python scripts were used to understand data (4). JS was used to visualize and organize data (5). Another Python script was used to verify JS visualization (6) and final prototype was created using a force-directed layout (7).

Abstract— Just-in-time (JIT) compilers are commonly used in web applications today, and are thus utilized by millions of users. Like a traditional compiler, JIT compilers translate high-level programming languages into computer machine code. JIT compilers differ in that they optimize code during execution. However, some optimizations within compilers can produce bugs that affect program performance. We utilized JavaScript and D3 to build a force-directed layout graph representation of the optimization evolution within a compiler. Our prototype supports user inspection of specific node elements, node relationships, and changes by phase. We seek to aid developers in quickly identifying optimization bugs to correct JIT compilers.

1 Introduction

Compilers are used to translate human-readable source code written in high-level languages into low-level machine language for program execution. All compilers produce an intermediate representation (IR), a graph representation of operations and relationships that identifies the optimizations that take place throughout this process. IRs evolve over time in iterative phases as the compiler program executes [1].

From Google's V8 to Apple's JavaScriptCore, Just-in-time (JIT) compilers are frequently used in everyday web applications. Unlike traditional compilers, JIT compilers optimize code concurrently with execution. Thus, due to their broad use across web infrastructure as well as their fast-paced implementation, correct JIT optimization is necessary to provide reliability and security to all web users [4]. The dynamic nature of JIT compilers and IRs makes debugging difficult, as optimization changes are not explicitly tracked by the compiler.

Here, we present an early prototype of a graph that visualizes the IR of a JIT compiler over several optimization phases. By displaying dynamic node information and edges, we hope developers can more easily see the way their compiler makes modifications to source code over time, making for quicker debugging.

2 Related Work

A crucial, unpublished work by Dalbo, Ahmed, and Lim strongly aligns with our research and informs our approach. In their work, these authors introduce JITScope, a conceptual framework for visualizing the IR of a JIT compiler. Although they did not implement a visualization using code, they sketched a paper prototype that includes core necessary features for understanding optimization phases such as a node-link diagram that can be restructured, animation keys for going through optimization phases in order, and a dropdown selector for choosing to observe a specific phase [3].

The broader space of JIT compiler research and debugging visualizations is sparse. Many papers cover relevant concepts such as traditional compiler visualizations, complex graph visualizations, and debugging tools for traditional compilers [5] [6] [12] [10]. The most notable existing tool is Turbolizer, built by Google for its V8 engine, which allows developers to step though different optimization phases of the IR. However, its assistance is limited by inaccessibility, both because of its intricate directions and that it can only be used with JavaScript [11]. Our work aims to bridge this gap through building a more accessible visualization framework.

Lim and Kobourov's work is a recent attempt to visualize JIT compilers. Through the use of their metro map metaphor, they represent an IR graph as a hypergraph and identify specific buggy optimizations. However, their approach is based on multiple iterations of a single input program, whereas we visualize a singular IR through several optimization phases. [9]. However, their work demonstrates the value and challenge of making compiler behavior interpretable.

3 Methodology

3.1 Data Specification and Processing

Our data is generated using Lim, Kang, and Debray's JITCIRModeler. Their code records all relevant IR information during compilation and arranges it in JSON file format [8]. Manual parsing of the JSON file was impractical, as the simplest source code that was only 13 lines long produced a JSON file of over 750,000 lines. Thus, we utilized Python scripts to better understand how the file was formatted.

The data file is arranged by node with attributes such as initial

• *Ellora Devulapally is with Davidson College. E-mail: eldevulapally@davidson.edu.*
• *Taft Harrell is with Davidson College E-mail: taharrell@davidson.edu.*
• *Katy Williams is with Davidson College. E-mail: kawilliams@davidson.edu.*

Manuscript received xx xxx. 202x; accepted xx xxx. 202x. Date of Publication xx xxx. 202x; date of current version xx xxx. 202x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.202x.xxxxxxx

edges, final edges, removed edges, replaced edges, added edges, and all optimization instructions applied to a node. Each instruction is stored within the instAccess field of a node, and has a phaseFnId attribute that represents the phase the instruction was executed. Understanding this data file was paramount towards visualizing our data, and data processing required comparable or greater time than developing the visualization itself.

3.2 Beta Visualization

We chose to develop this visualization in D3.js, along with other HTML and CSS components [2]. Files were separated to support multiple visualization frameworks, including processing the data in a separate main.js file, creating selection buttons that allowed users to alternate between the IR at different phases, and a hover-over tooltip that projected specific information about each node at each individual phase. A key component of our ideation process was developing a phase-based data structure that would collected information about edges between nodes at each phase. From a strictly node-based JSON file, we built a JS map of all optimization phases and their corresponding sets of [source, target] node to node edges.

3.3 Verification Procedures

Verification of the beta visualization was conducted to accomplish two goals: 1) Nodes were appropriately marked as alive or dead during each phase and 2) The edges for each node were visualized correctly through phases.

Initially, the beta vis was verified by comparing the edges of each node in our JS map with that of the final edges of each node, as listed in the JSON file. By comparing the final phase edges of each node within the visualization against this attribute, we were able to identify several bugs within our code and correct them.

However, this method was partially flawed, as edges could only be compared for the last phase. Our second approach was to build a Python script that built a list of edges-per-phase in JSON format based on data collected directly from the IR JSON. This script enabled us to directly compare edges of each phase instead of just the last phase and further improve our code.

3.4 Force-Directed Graph Layout

The beta visualization we produced was mostly used as a first attempt at visualization and a way of verifying edges were constructed correctly. However, its structure was not intuitive nor efficient for identifying changes in nodes by phase. Thus we used a force-directed layout to more efficiently organize nodes. Force-directed code written by Laranjeira was modified to utilize data structures created through our initial JavaScript data processing [7].

Figure 2 identifies all the main components of our final force-directed visualization. Notably, in our static visualization, nodes that are dead or not yet created are rendered with a lower opacity. In the force-directed graph, these nodes are not rendered on the screen, and only live nodes are seen.

4 Limitations and Future Work

This work has been guided by conversations with only one compiler expert within the Davidson College Computer Science department. Future work entails structured interviews with a broader set of real-world developers, including user testing with structured feedback and exploration of developer pain points when debugging JIT compilers. Open problems such as readability issues for larger IRs will inform the next phase of work.

5 Conclusion

We present a working prototype aimed at helping developers identify and correct bugs in JIT compilers. This tool provides specific IR information such as operations through nodes, relationships through edges, and how the IR evolves over time. We hope that this prototype can help users identify specific buggy nodes that represent inappropriate optimization. Through this, developers can more efficiently and quickly solve real-time web application bugs that affect millions of user worldwide.

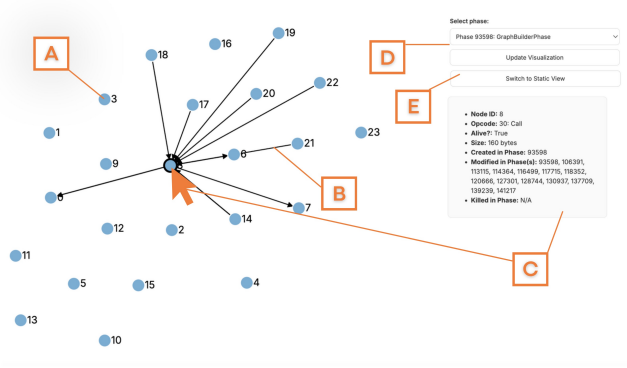

Fig. 2: A labeled image of the force-directed IR graph. A: Nodes of IR. Only alive nodes during each phase are displayed. B: Edges between nodes. Edges are directional to communicate source-to-target flow. C: Vis tooltip. Displays node information on hover over. D: Phase drop-down selection. Allows user to see IR under different optimization phases. E: Toggle button allowing user to switch between static or force-directed IR.

Acknowledgments

The authors would like to thank Dr. Terrence Lim for his expertise in compilers and helpful conversations. They would also like to thank Davidson College's Math and Computer Science Department for their support.

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Boston, MA, 2 ed., 2006. 1

[2] M. "Bostock, V. Ogievetsky, and J. Heer. "d3 data-driven documents". *"IEEE Transactions on Visualization and Computer Graphics"*, 2011. 2

[3] K. Dalbo, Y. Ahmed, and H. Lim. JITScope: Interactive visualization of jit compiler ir transformations. *arXiv preprint arXiv:2505.21599*, 2025. 1

[4] Q. Ducasse, P. Cotret, and L. Lagadec. War on jits: Software-based attacks and hybrid defenses for jit compilers - a comprehensive survey. *ACM Computing Surveys*, 2025. 1

[5] S. Erlandsson and G. Kärrman. TigerShrimp: An understandable tracing JIT compiler. Master's thesis, Chalmers University of Technology, 2021. 1

[6] Y. Jia et al. Detecting JVM JIT compiler bugs via exploring two-dimensional input spaces. In *Proc. International Conference on Software Engineering (ICSE)*. IEEE, 2023. doi: 10.1109/ICSE48619.2023.00016 1

[7] B. Laranjeira. D3 v6 force directed graph with directional straight arrows. https://observablehq.com/@brunolaranjeira/d3-v6-force-directed-graph-with-directional-straight-arrow, 2021. Accessed: 2026-05-09. 2

[8] H. Lim, X. Kang, and S. Debray. Modeling code manipulation in JIT compilers. pp. 9–15, 2022. doi: 10.1145/3520313.3534656 1

[9] H. Lim and S. Kobourov. Visualizing jit compiler graphs. *arXiv preprint arXiv:2108.11426*, 2021. 1

[10] M. Stamenković and J. Jovanović. A web-based educational system for teaching compilers. *IEEE Trans. Learn. Technol.*, 2019. 1

[11] V8, Project. Turbolizer. https://v8.github.io/tools/head/turbolizer/. Accessed: 2026-05-09. 1

[12] H. Zhong. Understanding compiler bugs in real development. In *Proc. International Conference on Software Engineering (ICSE)*, 2012. 1