Visualizing JIT Compiler Graphs

HeuiChan Lim and Stephen Kobourov

Department of Computer Science, University of Arizona

Abstract. Just-in-time (JIT) compilers are used by many modern programming systems in order to improve performance. Bugs in JIT compilers provide exploitable security vulnerabilities and debugging them is difficult as they are large, complex, and dynamic. Current debugging and visualization tools deal with static code and are not suitable in this domain. We describe a new approach for simplifying the large and complex intermediate representation, generated by a JIT compiler and visualize it with a metro map metaphor to aid developers in debugging.

1 Introduction

Many modern programming systems, such as JavaScript engines that are running our web browsers, use just-in-time (JIT) compilers to improve performance. Examples include Google Chrome, Microsoft Edge, Apple Safari, and Mozilla Firefox, which are used by 2.65 billion, 600 million, 446 million, and 220 million, respectively [4]. JIT compiler bugs can lead to exploitable security vulnerabilities [1,6–9]. Such a bug in Google Chrome could be used to hijack passwords and to navigate to other sites and execute malicious programs, as reported by the Microsoft Offensive Security Research team (CVE-2017-5121 [1]). Thus, the ability to quickly analyze, localize and fix JIT compiler problems is important. However, existing work and available tools focus on static code [15,16,23], and so they are not suitable for developers in debugging JIT compilers, which generates code at run-time. Additionally, the size and complexity of JIT-based systems [12] combined with the dynamic nature of JIT compiler optimizations, make it challenging to analyze and locate bugs quickly. For example, Google V8 has more than 2,000 source files and more than 1 million lines of code.

Traditional debuggers rely on text even though the main feature of a JIT compiler is building a graph-like structure to translate bytecode into optimized machine code. With this in mind, we propose a new debugging tool, which visualizes the JIT compiler's intermediate representation (IR). Our approach uses IR identification and generation techniques described by Lim and Debray [26], where the compiler-related half of the visualization tool's pipeline are described in detail. In this paper we focus on the visualization half, which includes: merging multiple IR graphs into a single graph, simplifying the merged graph, converting the simplified graph into a hypergraph, simplifying the hypergraph, and visualizing the hypergraph using a metro map metaphor. Visualizing the JIT compiler's IR allows us to answer questions such as:

1. What optimizations took place to generate the machine code?

2. What is the relationship among the optimization phases?
3. Which optimization phase was most active?
4. What optimizations affected a specific node?
5. Which optimization phases are likely to be buggy?

Related Work: There are many methods and tools for debugging static code compilers and optimized code, but little on using the intermediate representation and visualizing it to show the explicit information about the compilation and optimization processes. Google V8's Turbolizer [5,13] is one of very few IR visualization tools. It shows the final IR graph after each optimization process and provides interactive features to view the control-flow graphs for each optimization phase. Although Turbolizer provides some information about the IR nodes and their relationships, it does not provide enough information about the optimization process and cannot answer several of our initial set of questions.

Dux *et al.* [21] visualize dynamically modified code at run-time with call graphs and control-flow graphs by showing the graph changes with animation, allowing end-to-end play, pause, and forward/backward step-by-step animation. CFGExplorer [20] visualizes the control-flow graph of a program to represent the program structure for dynamic binary analysis. It provides interactive features allowing developers to find specific memory addresses, loops, and functions to analyze the system. CcNav [19] analyzes and visualizes a C++ compiler's optimization process with a call graph, control-flow graph, and loop hierarchies.

Control-flow graphs and call graphs are popular in program analysis, especially for analyzing static code. However, they are different from dynamically generated IR graphs. Tools for visualizing and interacting with control-flow graphs and call graphs (such as those above) are not sufficient for visualizing the IR graph as, e.g., they cannot capture the optimization phases.

Background: We briefly introduce several concepts relevant to JIT compilers.

Interpreter: a computer program that converts input source code into bytecode and executes it without compiling it into a machine code [22].

Bytecode: instructions generated from input source code by an interpreter; bytecode is portable, unlike compiled programs, and used in many modern languages and systems, such as JavaScript, Python, and Java [17].

Instruction-level Trace: a file that holds all the instructions that a programming system, such as a JIT compiler, has generated and executed at runtime. The instructions are in a machine-level code with symbol information (e.g., function names) and are used for performance analysis and debugging.

Just-in-Time (JIT) compiler: a program that turns bytecode into instructions that are sent to a computer's processor, to improve performance [24]; see Fig. 1(a) for an example of JIT compiler in Google's V8 pipeline.

Optimized code: machine code generated from bytecode by a JIT compiler that can be directly executed by a processor.

Intermediate Representation (IR): a type of graph also known as sea-of-nodes [11,14,18]. Unlike other graphs used in program analysis, such as control-flow or data-flow graphs which have specific types of nodes, nodes in the sea-of-nodes graph represent different types: from scalar values and arithmetic op-

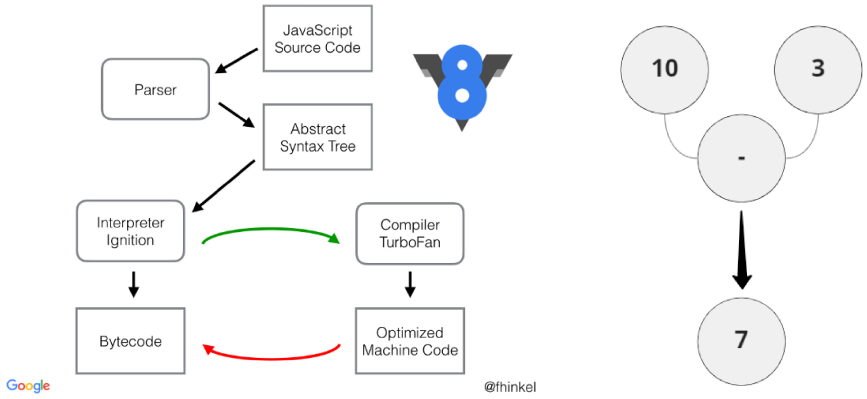

Fig. 1. (a) V8 Pipeline [12] (b) Example of constant folding optimization.

erators to variables and control-flow nodes and function entry nodes. Similarly, edges represent different relationships (e.g., semantic and syntax relationships).

Optimization: adding, removing, and merging nodes and edges in the graph during execution. In a single JIT compilation, the compiler executes several different optimization phases (inlining, loop peeling, constant propagation) to generate efficient machine code, which modify the IR graph and correspond to new hyperedges (the set of all nodes generated or optimized in this phase); see Fig. 1(b) for an example of constant propagation.

Proof-of-Concept Program: an input program that is used to trigger the buggy behavior in the JIT compiler, i.e., a valid program (without any bugs) which when run can reveal bugs in the JIT compiler. In our experiment, we are targeting JavaScript engine V8, so the PoC is a JavaScript program.

2 Visualizing the Intermediate Representation

Our approach for capturing and visualizing the IR of a JIT compiler below uses compiler-related steps 1-4 [26], and steps 5-9 are described in brief below.

1. Modify the input program, P_0, to create similar programs, $\{P_1, ...P_N\}$, by generating the abstract syntax tree for P_0 and then randomly modifying nodes in the tree with allowable edits (passing semantic/syntactic checks). The newly created programs either still contain the code that triggers a bug in the JIT compiler, or the buggy code is replaced and no bug is triggered. In the first case, the execution output of the optimized code is different from the interpreted code (as with P_0).
2. Run each program P_i and collect the instruction-level traces.
3. Analyze traces to check if P_i triggers a bug in the JIT compiler and to identify P_i's IR and the optimization phases executed while optimizing P_i.

4. Select candidate hyperedges, suspected to be buggy, from the information gathered in step 3.
5. Merge all selected candidate hyperedges into the original IR from P_0.
6. Simplify the merged IR by reducing the number of nodes and edges.
7. Convert the simplified graph into a hypergraph by extracting the hyperedges from step 4 and analyzing each node's optimization status.
8. Simplify the hypergraph by reducing the number of hyperedges and nodes.
9. Visualize the simplified hypergraph with MetroSets [25].

2.1 Intermediate Representation

Recall that the intermediate representation (IR) of a JIT compiler is a sea-of-nodes graph that the compiler generates at the beginning of its execution by parsing the bytecode and optimizing it with several optimization phases. Formally, the IR is a simple, undirected graph $G = (V, E)$, where V represents the nodes optimized by the JIT compiler and E contains pairs of nodes connected by different relationships (e.g., semantic and syntax relationships, such as math expressions). By keeping track of the optimization information for each node we construct the hypergraph $H = (V, S)$ from G, where V is a set of nodes optimized by the JIT compiler and each hyperedge in S represents an optimization phase.

Two important node features are phases and opcodes. Phases are the optimization phases where a node was generated and optimized (and which later correspond to hyperedges). Opcodes represent node operations (e.g., add, sub, return). A node also has two different attribute groups: (1) *basic*, such as a node id, address, list of neighbors, opcode, and IR ID; and (2) *optimization*, such as hyperedge (phase) ID, generated hyperedge name, and optimized hyperedge names. Note that a node is generated at one hyperedge, but can be present in multiple different hyperedges, due to different optimization phases.

Recall that given one JavaScript code we generate N similar versions to see if any of them trigger bugs. We generate the IRs for all of these versions (typically about 20). In the real-world examples we work with, each such IR graph has about 300-500 nodes and 30-40 optimization phase executions.

2.2 Merging Intermediate Representation Hyperedges

We now merge the N similar but different intermediate representations into one single graph. There are two main reasons to do this. First, we want to see the differences among the graphs in one single view. Second, by comparing hyperedges from a buggy program IR to hyperedges from a non-buggy program IR, we can find differences in some hyperedges due to different optimizations, and thus find the bug. Consider, for example, a hyperedge α in both buggy and non-buggy program IRs and suppose that an additional node (the result of incorrect optimization) makes a buggy program's α different from the non-buggy program's α. A merged hyperedge will show this additional node, and its attributes will identify the buggy IR. A developer can now see that there was an optimization difference in α and find the bug.

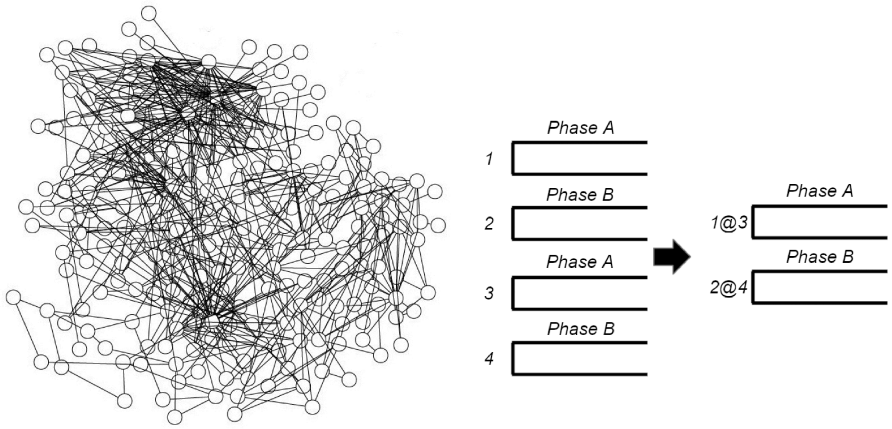

Fig. 2. (a) Example of an IR graph; (b) example of hypergraph simplification.

Let R_0 be the IR from the original program and $\{R'_1, ..., R'_N\}$ the IRs from the modified programs. Let $\{r'_1, ..., r'_n\}$ be sub-IRs, where r'_i is a subgraph of R'_i when $R'_i \neq R_0$, i.e., $r'_i \subseteq R'_i$, and n is the number of IRs different from R_0 ($n \leq N$). Each r'_i holds buggy candidate hyperedges: R'_i hyperedges are different from R_0's hyperedges. We traverse all sub-IRs, comparing each to R_0, and update the merged IR; see Algorithm 1 in [27] for detail.

2.3 Intermediate Representation Simplification

Although the resulting merged graph may be useful for debugging, its complexity makes it difficult for developers to use; see Fig. 2(a). Therefore, we simplify the graph, convert it into a hypergraph, and simplify the hypergraph (hopefully without losing much information in these simplifications). The main goal is to end up with an interactive visualization that allows developers to debug.

Reducing the IR Graph: We remove dead nodes (nodes with no adjacent edges) as they are not translated into machine code and do not affect other nodes. We then identify nodes that can be merged without losing important information. A pair of nodes is merged if they have the same opcode, the same optimization information, belong to the same IR (which can be identified by the IR id attribute), and share the same neighbors; see Algorithm 2 in [27] for detail.

Reducing the IR Hypergraph: We convert the simplified graph $G = (V, E)$ into a hypergraph $H = (V, S)$, by extracting hyperedges based on the optimization phases; see Algorithm 3 in [27]. Recall that a node v generated in phase/hyperedge α and optimized in phases/hyperedges ϕ and γ now belongs to all three hyperedges. We reduce hypergraph H by merging suitable pairs of hyperedges. Different nodes can have the same hyperedge names as attributes, but different hyperedge IDs, as IDs are assigned based on the execution order. Therefore, we merge hyperedges with the same name into a single hyperedge while assigning a new unique identifier generated from the original IDs. We use ID concatenation to obtain unique identifiers. Consider two hyperedges A and

B executed twice in the order shown in Fig. 2(b). We use the order to create unique IDs by merging the 4 hyperedges into 2 hyperedges and assigning new IDs, generated by concatenating two IDs delimited with a special character '@'; see Algorithm 4 in [27].

This reduces the number of hyperedges but increases the number of nodes in each hyperedge. Next, we traverse each hyperedge $s \in S$, and we use node opcodes to see if they can be merged; see Algorithm 5 and Table 1 in [27] for more details and results.

2.4 Visualizing the Hypergraph with MetroSets

MetroSets [25] uses the metro map metaphor to visualize medium-size hypergraphs. It clearly shows the relationships between hyperedges, which in our case captures the relationships among the optimizations. MetroSets provides simple and intuitive interactions that make it possible to quickly identify hyperedges (metro lines) that contain suspicious nodes (metro stations), or hyperedges that intersect with a particular suspicious hyperedge. Each node in the MetroSet map is labeled with its unique ID (representing the node generation timeline). The attributes shown when hovering over a node are phase, opcode, address, graph ID, and phase ID. A phase attribute tells the user where the node was generated and it is useful when nodes belong to multiple sets. A developer can distinguish the phase that generated a node and phases where it was optimized.

3 Evaluation

We work with Google's JavaScript engine and its JIT compiler, using a dynamic analysis tool built on top of Intel's Pin software [28] to collect instruction-level traces, XED [3] for instruction decoding [3], esprima-python [10] to generate the syntax-tree from JavaScript code, and escodegen [2] to regenerate JavaScript from the syntax-tree. Our data comes from the Chromium bug report site; see [26] for details. We can identify the bugs in all listed bug reports, including Chromium bug report 5129. This version of the compiler has a bug in the *EarlyOptimization* phase. We generate 19 additional modified JavaScript programs from the original and run all 20. The instruction traces are used to generate the IR graph shown in Fig. 2(a) and our visualization is shown in Fig. 3. We can now attempt to answer some of the questions from Sec. 1.

"What optimizations took place to generate the machine code?" The map and the "Key to Lines" legend show all optimization phases.

"What is the relationship among the optimization phases?" We can examine the corresponding lines and use the interactive exploration modes (intersection, union, complement, etc.) to see the relationships among the phases.

"Which optimization phase was most active?" We can visually identify the longest line, or hover over each line and see the number of nodes in it; see Figure 9 in [27] for an example of the most active optimization phase.

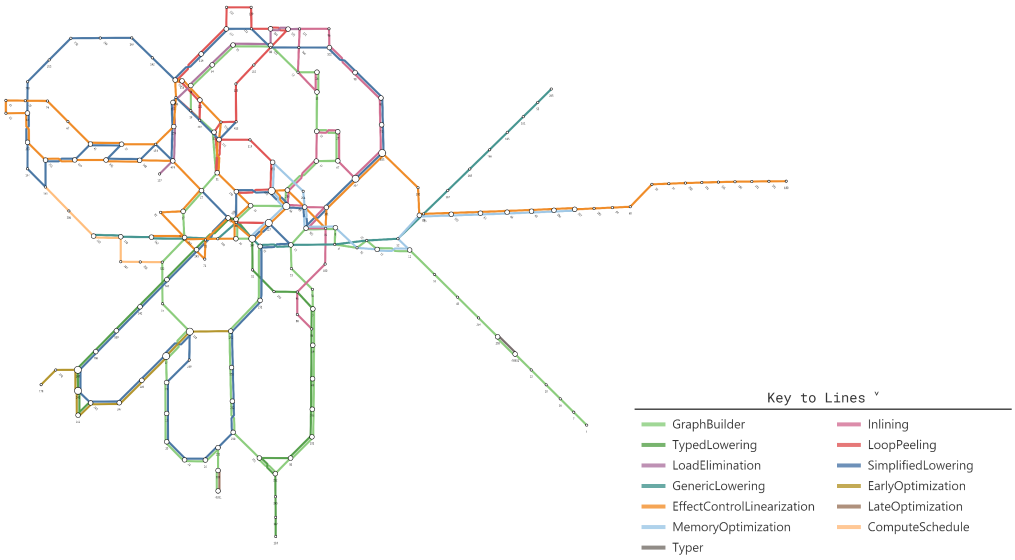

Fig. 3. Metro map of the IR graph from bug report 5129.

"What optimizations affected a specific node" We can hover over the node of interest, which grays out the lines that don't contain the node. We can then examine each of the corresponding lines and look at the displayed node attributes.

"Which optimization phases are likely to be buggy?" One natural way to do this is to find parts that differ in the IR graphs with the bug and those without. In other words, a program is buggy because either it has additional optimizations or missing optimizations, and this information is captured in the IRs. Any line that has many non-original IRs represents a significant difference between buggy and non-buggy programs. In this case study, the majority of nodes (9 out of 11) in the EarlyOptimization line are from different IRs, indicating a difference in optimization between buggy and non-buggy programs; see the full paper [27] for more examples.

Our prototype is available at https://hlim1.github.io/JITCompilerIRViz/.

Acknowledgements This research was supported in part by the National Science Foundation under grants CNS-1908313 and DMS-1839274.

References

1. Browser security beyond sandboxing (2017). https://www.microsoft.com/security/blog/2017/10/18/browser-security-beyond-sandboxing/, accessed 2021-01-22
2. Estools/escodegen (2012). https://github.com/estools/escodegen, accessed 2021-02-03

3. Intel xed (2019). https://intelxed.github.io/, accessed 2021-02-03
4. Internet browser market share 2012-2021 (2021). https://www.statista.com/statistics/268254/market-share-of-internet-browsers-worldwide-since-2009/, accessed 2021-05-23
5. Intro to chrome's v8 from an exploit development angle (2020). https://sensepost.com/blog/2020/intro-to-chromes-v8-from-an-exploit-development-angle/, accessed 2021-02-21
6. Issue 1072171: Security: missing the -0 case when intersecting and computing the type::range in numbermax (2020). https://bugs.chromium.org/p/chromium/issues/detail?id=1072171, accessed 2021-02-01
7. Issue 5129: Turbofan changes $x - y < 0$ to $x < y$ which is not equivalent when (x - y) overflows (2016). https://bugs.chromium.org/p/v8/issues/detail?id=5129, accessed 2021-02-01
8. Issue 8056: [turbofan] optimized array indexof and array includes ignore a prototype that is not initial (2018). https://bugs.chromium.org/p/v8/issues/detail?id=8056, accessed 2021-02-01
9. Issue 961237: Security: jit difference on comparison in d8 (2019). https://bugs.chromium.org/p/chromium/issues/detail?id=961237, accessed 2021-02-01
10. Kronuz/esprima-python (2017). https://github.com/Kronuz/esprima-python, accessed 2021-02-03
11. Turbofan ir (2016). https://docs.google.com/presentation/d/1Z9iIHojKDrXvZ27gRX51UxHD-bKf1QcPzSijntpMJBM/edit?usp=embed_facebook, accessed 2021-01-21
12. Understanding v8's bytecode (2017). https://medium.com/dailyjs/understanding-v8s-bytecode-317d46c94775, accessed 2021-01-21
13. Using turbolizer to inspect the v8 jit compiler (2019). https://lukeolney.me/posts/v8-turbolier/, accessed 2021-02-22
14. V8: Behind the scenes (2016). https://benediktmeurer.de/2016/11/25/v8-behind-the-scenes-november-edition, accessed 2021-01-21
15. Adl-Tabatabai, A., Gross, T.R.: Source-level debugging of scalar optimized code. In: Fischer, C.N. (ed.) Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI), Philadephia, Pennsylvania, USA, May 21-24, 1996. pp. 33–43. ACM (1996). https://doi.org/10.1145/231379.231388
16. Brooks, G., Hansen, G.J., Simmons, S.: A new approach to debugging optimized code. In: Feldman, S.I., Wexelblat, R.L. (eds.) Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI), San Francisco, California, USA, June 17-19, 1992. pp. 1–11. ACM (1992). https://doi.org/10.1145/143095.143108
17. Dahm, M.: Byte code engineering. In: Cap, C.H. (ed.) JIT '99, Java-Informations-Tage 1999, Düsseldorf 20./21. September 1999. pp. 267–277. Informatik Aktuell, Springer (1999). https://doi.org/10.1007/978-3-642-60247-4_25

18. Demange, D., de Retana, Y.F., Pichardie, D.: Semantic reasoning about the sea of nodes. In: Dubach, C., Xue, J. (eds.) Proceedings of the 27th International Conference on Compiler Construction, CC 2018, February 24-25, 2018, Vienna, Austria. pp. 163–173. ACM (2018). https://doi.org/10.1145/3178372.3179503
19. Devkota, S., Aschwanden, P., Kunen, A., LeGendre, M.P., Isaacs, K.E.: Ccnav: Understanding compiler optimizations in binary code. IEEE Trans. Vis. Comput. Graph. **27**(2), 667–677 (2021). https://doi.org/10.1109/TVCG.2020.3030357
20. Devkota, S., Isaacs, K.E.: Cfgexplorer: Designing a visual control flow analytics system around basic program analysis operations. Comput. Graph. Forum **37**(3), 453–464 (2018). https://doi.org/10.1111/cgf.13433
21. Dux, B., Iyer, A., Debray, S.K., Forrester, D., Kobourov, S.G.: Visualizing the behavior of dynamically modifiable code. In: 13th International Workshop on Program Comprehension (IWPC 2005), 15-16 May 2005, St. Louis, MO, USA. pp. 337–340. IEEE Computer Society (2005). https://doi.org/10.1109/WPC.2005.45
22. Gregg, D., Ertl, M.A., Krall, A.: Implementing an efficient java interpreter. In: Hertzberger, L.O., Hoekstra, A.G., Williams, R. (eds.) High-Performance Computing and Networking, 9th International Conference, HPCN Europe 2001, Amsterdam, The Netherlands, June 25-27, 2001, Proceedings. Lecture Notes in Computer Science, vol. 2110, pp. 613–620. Springer (2001). https://doi.org/10.1007/3-540-48228-8_70
23. Hölzle, U., Chambers, C., Ungar, D.M.: Debugging optimized code with dynamic deoptimization. In: Feldman, S.I., Wexelblat, R.L. (eds.) Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI), San Francisco, California, USA, June 17-19, 1992. pp. 32–43. ACM (1992). https://doi.org/10.1145/143095.143114
24. Ishizaki, K., Kawahito, M., Yasue, T., Takeuchi, M., Ogasawara, T., Suganuma, T., Onodera, T., Komatsu, H., Nakatani, T.: Design, implementation, and evaluation of optimizations in a javatm just-in-time compiler. Concurr. Pract. Exp. **12**(6), 457–475 (2000). https://doi.org/10.1002/1096-9128(200005)12:6<457::AID-CPE485>3.0.CO;2-0
25. Jacobsen, B., Wallinger, M., Kobourov, S.G., Nöllenburg, M.: Metrosets: Visualizing sets as metro maps. IEEE Trans. Vis. Comput. Graph. **27**(2), 1257–1267 (2021). https://doi.org/10.1109/TVCG.2020.3030475
26. Lim, H., Debray, S.: Automated bug localization in JIT compilers. In: Titzer, B.L., Xu, H., Zhang, I. (eds.) VEE '21: 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, Virtual USA, April 16, 2021. pp. 153–164. ACM (2021). https://doi.org/10.1145/3453933.3454021
27. Lim, H., Kobourov, S.: Visualizing the intermediate representation of just-in-time compilers. `https://arxiv.org/abs/2107.00063` (2021)
28. Luk, C., Cohn, R.S., Muth, R., Patil, H., Klauser, A., Lowney, P.G., Wallace, S., Reddi, V.J., Hazelwood, K.M.: Pin: building customized program analysis tools with dynamic instrumentation. In: Sarkar, V., Hall, M.W. (eds.) Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005. pp. 190–200. ACM (2005). https://doi.org/10.1145/1065010.1065034