EdgeTrace: Automated Call Graph Comparison for GraalVM Native Image

Milan Vodák*

Abstract
This project focuses on the GraalVM Native Image compiler and its static analysis technique that allows it to transform Java bytecode into optimized native binaries. Our goal was to find a viable method of comparing call graphs from subsequent compilations and develop a tool to help visualize impacts and evaluate the results of this analysis. We implemented an algorithm for call graph comparison and created a web application that allows processing call graphs from Native Image compilation reports and their comparison and visualization.

*xvodak07@stud.fit.vut.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

GraalVM Native Image is a compiler that transforms Java bytecode into native executable binaries. To optimize the resulting programs, Native Image utilizes static analysis techniques, whose results have to be examined. The goal of this project is to develop a reliable method to compare call graphs produced during static analysis by GraalVM Native Image and present the differences in a way that can be used to evaluate the effectiveness of such analysis. I experimented with various algorithms and methods for graph comparison and used one of them to develop the final product: *EdgeTrace*, a web application that allows the import of Native Image reports, viewing them, and comparing them to each other. The result is a portable and intuitive web application that can be used by GraalVM Native Image developers to examine the results of their work and to aid with further development of the compiler's static analysis.

2. Overview of GraalVM Native Image

GraalVM is a Java Development Kit (JDK) developed by Oracle and built on top of Oracle JDK or OpenJDK and the HotSpot Java Virtual Machine [3]. Aside from a modern just-in-time compiler, one of its components is Native Image – technology that allows compilation of applications written in Java ahead-of-time. The output of Native Image is a standalone executable file that contains the user application, all of its dependencies, and the components of the runtime environment, such as the garbage collector and threading support [2]. This means that the application can run independently, without the need for a Java Virtual Machine (JVM).

Native Image works with Java bytecode and the result is a native executable file for a specific operating system and architecture, called *native image* [10]. The compiler reads bytecode from the given Java class file, performs points-to analysis on it, optionally runs user initialization code, creates a snapshot of the heap, then compiles all reachable methods, which have been discovered by the analysis, into machine code, and finally links the compiled code and parts of the runtime into a single executable file. The entire compilation process is shown in Figure 1.

To optimize the resulting programs, Native Image utilizes various static analysis methods, mainly *points-to analysis* (PTA), which is used to discover elements (classes, methods, and variables) that will probably be used during program execution, such that the output, in the ideal case, contains only this so-called reachable code. Java supports polymorphism and Java programs often have a lot of virtual methods and calls, thus there can be multiple different implementations of every method, and at build time, it is not always known which specific implementation will be called; however, sometimes it can be determined by static analysis, such as the PTA. Ignoring the discovery of reachable and unreachable elements would cause compilation of large amounts of code that are not needed at runtime. The absence of such analysis would result not only in unnecessarily large binaries, but also much longer compilation times, because all program methods would have to be compiled, regardless of whether or not they are actually used at runtime [10].

To compare different compilations, we use reports generated by Native Image, in particular, call graphs in the form of CSV tables.

3. Comparing Call Graphs

Comparison of call graphs is a challenging task, mainly because call graphs of real applications are very large to examine, with tens to hundreds of thousands of nodes. We use an approximation algorithm by Lhotak et al. [4], which simulates the flow of a fluid in reverse from the methods to the program's entrypoints. The two graphs being compared are called *supergraph* and *subgraph*. An initial value is assigned to all methods of the supergraph, and in each iteration, the fluid flows in reverse through edges, and its amount slowly decreases until it gets consumed entirely by a method that is reachable in both graphs. The amounts of fluid flowing through each edge are recorded and after the maximum amount among all methods reaches a specific value, the algorithm terminates. Edges that lead into larger areas missing from the subgraph will have a higher score.

Consider the example graphs in Figure 2, where the supergraph has five nodes that are missing from the subgraph. We want to identify the edges that cause the most difference. We would probably expect the edge from node B to node D to have the highest score, as removing it would eliminate three nodes from the call graph. Other important edges are C$\rightarrow$G and C$\rightarrow$H. Those are also a part of the difference boundary; however, their score should be lower than the first edge, as they lead into smaller subgraphs, and removing either would not have such an impact. The fluid flow algorithm correctly identifies these three edges, as seen in the final scores.

Our final implementation is written in C, focusing on optimization to provide maximum performance. For two call graphs with a difference of 25,000 nodes, the algorithm takes around an hour to complete; however, we can significantly reduce the time it takes to run by limiting the number of iterations with minimal loss of precision.

4. EdgeTrace

EdgeTrace is a web application that allows the import of call graphs from Native Image compilation reports, their visualization, and difference calculation.

At the heart of EdgeTrace is Neo4j [6], a graph database that stores the imported call graphs and lets us run queries on them by using its Cypher language [5]. The backend of the web application is written in Python and built using the FastAPI framework [9], and it sends queries to the graph database using the official Neo4j Python driver [7].

The visualization is handled by a SvelteKit frontend [8] that uses Cytoscape.js [1] to render graphs. For better orientation in a call graph, EdgeTrace provides a tree view of packages, classes, and methods that can be filtered (see Figure 5), and toggleable compound nodes that highlight the same hierarchy directly in the graph (see Figure 4). Detailed properties of methods and edges are shown by clicking on them inside the graph, and neighbors of methods (callers and callees) can be easily shown in the graph using the neighbors panel. It is possible to open up to 10 separate views simultaneously.

The difference algorithm is written in C and is controlled by the backend using Python–C bindings. The computation progress is displayed in the application in real time. After computing the difference between two specific graphs, the most important edges can be viewed, and their coloring uses a scale based on their score, as shown in Figure 3.

5. Conclusions

The result of this project is an intuitive web application that processes reports of GraalVM Native Image compilations, compares the resulting call graphs, and displays them in an interactive way. We have found a viable method of call graph comparison and developed a program that allows intuitive visualization of the process. EdgeTrace can already be used by Native Image's developers to evaluate the results of their work on static analysis.

Acknowledgements

I would like to thank my supervisor, Ing. David Kozák, for his valuable help during the work on this project.

References

[1] Franz, M. *Cytoscape.js* online. April 2025. Available at: https://js.cytoscape.org. [cit. 2025-04-27].

[2] GraalVM. *Native Image Basics* online. April 2025. Available at: https://www.graalvm.org/latest/reference-manual/native-image/basics. [cit. 2025-04-03].

[3] GraalVM. *Why GraalVM?* online. April 2025. Available at: https://www.graalvm.org/why-graalvm. [cit. 2025-04-03].

[4] Lhoták, O. Comparing call graphs. In: *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. New York, NY, USA: Association for Computing Machinery, 2007, p. 37–42. PASTE '07. ISBN 9781595935953. Available at: https://doi.org/10.1145/1251535.1251542.

[5] Neo4j. *Cypher Query Language* online. April 2025. Available at: https://neo4j.com/product/cypher-graph-query-language. [cit. 2025-04-27].

[6] Neo4j. *Neo4j Graph Database* online. March 2025. Available at: https://neo4j.com/product/neo4j-graph-database. [cit. 2025-04-27].

[7] Neo4j. *Neo4j Python Driver 5.28 — Neo4j Python Driver 5.28* online. February 2025. Available at: https://neo4j.com/docs/api/python-driver/current. [cit. 2025-04-27].

[8] Svelte contributors. *Svelte • Web development for the rest of us* online. April 2025. Available at: https://svelte.dev. [cit. 2025-04-27].

[9] tiangolo. *FastAPI* online. April 2025. Available at: https://fastapi.tiangolo.com. [cit. 2025-04-27].

[10] Wimmer, C.; Stancu, C.; Hofer, P.; Jovanovic, V.; Wögerer, P. et al. Initialize once, start fast: application initialization at build time. *Proc. ACM Program. Lang.* New York, NY, USA: Association for Computing Machinery, october 2019, vol. 3, OOPSLA. Available at: https://doi.org/10.1145/3360610.