An Approach for Correlating Compiler Optimizations with Runtime Performance

Befikir Bogale[1] (Student), Olga Pearce[2], Stephanie Brink[2], David Boehme[2], Ignacio Laguna[2], Jason Burmark[2], Ian Lumsden[1], Tom Scogland[2] (Mentors), Michela Taufer[1] (Advisor)
[1]University of Tennessee Knoxville [2]Lawrence Livermore National Laboratory

ABSTRACT

Performance-portability libraries such as RAJA enable single-source applications to run on diverse architectures, but performance often depends on compiler decisions that are hard to observe. Existing tools either show compiler activity without runtime context or runtime performance without compiler provenance. We present an approach that integrates compiler optimization data into runtime profiles, allowing developers to link specific optimizations to their performance impact. We demonstrate this approach through a case study where we determine the compiler requirements of kernels from the RAJA Performance Suite.

KEYWORDS

High-Performance Computing, LLVM, Compiler Remarks

1 INTRODUCTION

Perfomance portability libraries such as RAJA [1] enable single-source applications to run across diverse hardware and programming models through architecture abstractions. These abstractions shift performance responsibility to the compiler. Developers often struggle to connect compiler optimization decisions (*e.g.,* apply inlining, do not vectorization) to runtime performance. The missing link between compiler decisions and runtime behavior makes performance hard to understand.

Solving this challenge requires a two-fold approach: first, capturing the compiler's optimization decisions, and second, relating them to runtime behavior. The optimization compiler remarks from LLVM [4] provide the former by reporting which optimizations were attempted, applied, or skipped during compilation. Runtime profiling tools such as Caliper [2] provide the latter by exposing runtime information. Yet, because these sources remain disconnected, the compiler's influence on performance persists as a black box—remarks lack runtime context, and profiles lack compiler provenance—leaving developers without a clear link between compiler decisions and observed performance.

Our goal is to relate compiler decisions with runtime behavior so developers can understand which optimizations matter and why. Our contributions are as follows: (1) We develop a lightweight, general approach exposing compiler optimization provenance at runtime, which integrates with existing profiling infrastructure such as Caliper, and programmatic analysis with Thicket; and (2) We validate this approach on the RAJA Performance Suite across optimization levels, demonstrating how fusing compiler and runtime perspectives enables evidence-guided optimization for performance portability.

SC '25, November 16–21, 2025, Saint Louis, MO, USA
2025. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 RELEVANT TOOLING

Our approach makes use of the following three software projects, which are developed and maintained at Lawrence Livermore National Laboratory (LLNL), and are used extensively in performance portability and HPC application development workflows.

RAJAPerf. The RAJA Performance Suite (RAJAPerf) [5] is a collection of loop-based computational kernels implemented in multiple programming models, including platform-native (Base) and RAJA variants. It is designed to evaluate the performance and portability of RAJA abstractions across diverse architectures by providing comparable implementations of common HPC kernel patterns.

Thicket. Thicket [3] is a performance analysis tool for ensemble runs. It enables users to load, merge, and query performance data from multiple executions, supporting aggregated statistical analysis and visualization. Thicket's data model allows filtering and grouping by user-defined criteria, making it well suited for analyzing compiler–runtime relations at scale.

Caliper. Caliper [2] is an annotation-based profiling library that allows developers to mark code regions and collect fine-grained performance data. It supports integration with hardware counters, timers, and other performance metrics, producing structured profiles that can be related with user annotations.

3 THE BLACKBOX OF THE COMPILER

The primary mechanism to influence compiler optimizations is through the `-O` family of compiler flags, which have optimization levels ranging from minimal (`-O0`) to aggressive (`-O3`). Each level enables a predefined set of optimization strategies. While these flags broadly guide the compiler, they are coarse-grained controls: developers cannot directly see which specific optimizations were applied or skipped, nor assess their runtime impact trivially. This lack of visibility creates a "black box" around the compiler's behavior.

One partial way to shed light into this "black box" is through compiler remarks, that is diagnostics emitted by the compiler that report which optimizations were attempted, applied, or missed. While remarks can be valuable for performance tuning, they have two significant drawbacks. First, large projects may generate hundreds of thousands to millions of remarks, making them difficult to analyze at scale. Second, because remarks describe static compiler decisions without runtime context, it is challenging to relate them with runtime performance.

Conversely, profiling tools such as Caliper can provide rich runtime measurements for user-defined code regions but no insight into compiler behavior. At present, these two valuable data sources—compiler remarks and runtime profiles—exist in isolation. Our work addresses this gap by introducing a compiler plugin based methodology that integrates compiler optimization information directly into runtime profiles.

4 LLVM PASS PLUGINS TO COLLECT COMPILER REMARKS

Our approach bridges the gap between compiler optimization data and runtime performance measurements through a compiler plugin-based methodology. The methodology requires two requirements of the application: (1) it must be compiled with an LLVM-based compiler (*e.g.,* Clang) and (2) it must have regions of interest marked with Caliper annotations.

Our methods consist of two LLVM pass plugins: a remark extraction plugin and a remark instrumentation plugin. The remark extraction plugin is a module pass that runs early in the LLVM pipeline and installs a custom diagnostic handler to intercept all emitted optimization remarks. For each translation unit, the plugin records attempted, successful, and missed optimizations into an in-memory data structure, which is then written to a machine-readable JSON file at the end of compilation.

The remark instrumentation plugin is also a module pass, but it reads the generated remark JSON files from the remark extraction plugin for each translation unit. It then inserts Caliper API calls at the corresponding instructions for each remark, encoding the optimization metadata as Caliper annotations and preserving this provenance in the executable. This process produces two executables: an un-instrumented version for collecting baseline runtime profiles, and an instrumented version for collecting profiles augmented with compiler optimization data. At runtime, Caliper produces output files containing both performance metrics and associated compiler optimization information. These annotated profiles can then be visualized using performance analysis tools such as Thicket.

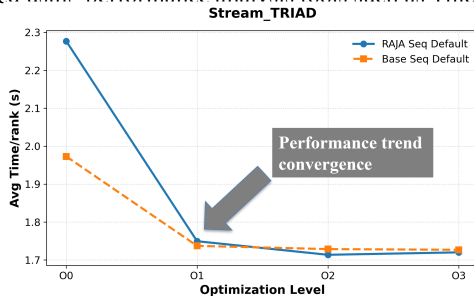

Figure 1: Performance of RAJA vs. Base implementation for Stream_TRIAD kernel across optimization levels.

5 RAJA PERFORMANCE SUITE CASE STUDY

We evaluate our methodology using the RAJA Performance Suite (RAJAPerf) [5]. In this case study, we focus on the platform-native (Base) and RAJA variants in determining which compiler optimizations are necessary for RAJA kernels to achieve performance comparable to their Base counterparts. Fig. 1 shows performance for the Base and RAJA implementations of the Stream_TRIAD kernel across optimization levels. At `-O0`, the RAJA version performs significantly worse than Base. However, performance converges by `-O1` and remains comparable between both implementations at higher levels. We want to understand what changes in compiler optimizations occurred between `-O0` and `-O1` that closes this gap.

To answer this, we compare differences in compiler optimizations between the two levels for both implementations. We find a substantial increase in inlining activity for the RAJA implementation at `-O1` that far exceeds that of the Base version. The Sankey diagram in Fig. 2 shows that inlining dominates optimization activity for RAJA at this level, with 122 inlining attempts, 100 of which succeed. Of these, 91 are auto-inlined based on compiler heuristics rather than explicit hints. These results indicate that RAJA's abstractions require more aggressive inlining than Base to generate efficient code. Auto-inlining plays a central role in closing the performance gap, suggesting that compiler heuristics largely determine whether RAJA implementations reach parity with native implementations.

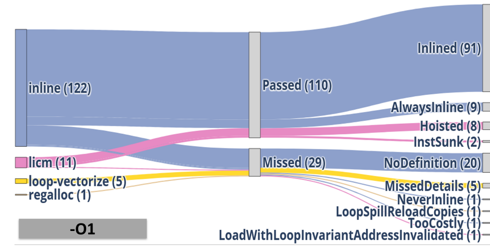

Figure 2: Optimimization breakdown of RAJA implementation of the Stream_TRIAD kernel at `-O1`.

6 CONCLUSIONS AND FUTURE WORK

We introduce a compiler plugin-based approach to bridge the gap between compiler optimization data and runtime profiles. This approach enables the relation compiler decisions with application performance, providing new insight into optimization requirements for HPC applications. For future work, we intend to extend support of our approach to GPU applications. We also intend to conduct research on the assignment of quantitative importance scores to optimization to further guide developers.

ACKNOWLEDGMENTS

This work was supported by the U.S. National Science Foundation (NSF) under grant numbers #2331152, #2334945, and #2103845. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 and was supported by the LLNL-LDRD Program under Project No. 24-SI-005 (LLNL-ABS-2010058).

REFERENCES

[1] David A. Beckingsale et al. 2019. RAJA: Portable Performance for Large-Scale Scientific Applications. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 71–81.

[2] David Boehme et al. 2016. Caliper: Performance Introspection for HPC Software Stacks. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 550–560.

[3] Stephanie Brink et al. 2023. Thicket: Seeing the Performance Experiment Forest for the Individual Run Trees. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing* (Orlando, FL, USA) *(HPDC '23)*. Association for Computing Machinery, New York, NY, USA, 281–293.

[4] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, USA, 75.

[5] Olga Pearce et al. 2025. RAJA Performance Suite: Performance Portability Analysis with Caliper and Thicket. In *Proceedings of the SC '24 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis* (Atlanta, GA, USA) *(SC-W '24)*. IEEE Press, 1206–1218.