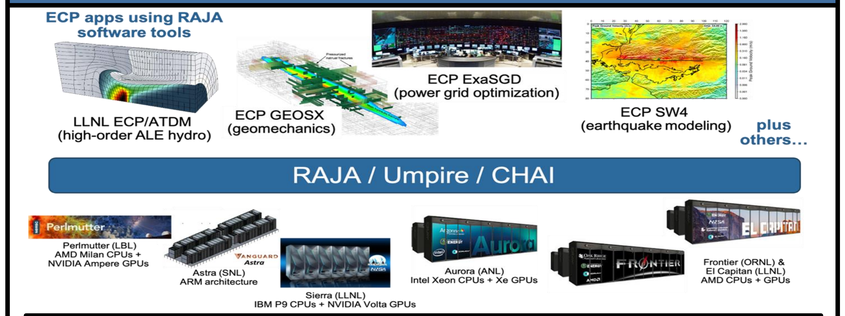
An Approach for Correlating Compiler Optimizations with Runtime Performance

Befikir Bogale[1] (Student)

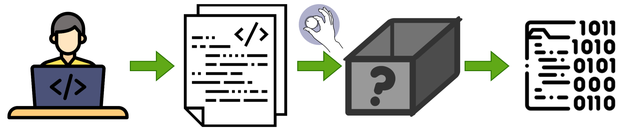
Olga Pearce[2,3], Stephanie Brink[2], David Boehme[2], Ignacio Laguna[2], Jason Burmark[2], Ian Lumsden[1], Tom Scogland[2] (Mentors), Michela Taufer[1] (Advisor)

[1]University of Tennessee, Knoxville, TN, USA [2]Lawrence Livermore National Laboratory, Livermore, CA, USA [3]Texas A&M University College Station, TX, USA

Science Simulations Run On Diverse Hardware



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



Our Solution: Opening the Compiler's Black Box

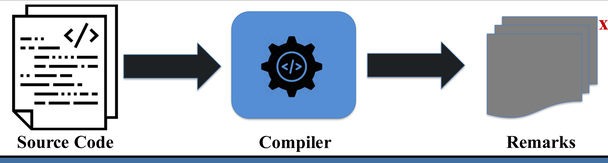
The compiler's middle end emits **remarks** (diagnostic messages) to report:

- Which optimizations were **attempted**
- Which optimizations **succeeded**
- Which optimizations **failed**

Example

```
{ {  {
    Function:   Foo
    Pass Name: Inline
    Remark Name. AlwaysInline
    Location: {
     Absolute Path : /home/foo.cpp
     Line: 10
     Column: 5
    Message:  Bar inlined into Foo
    with cost(always)
    Status: Passed
} }  }
```

The Problem

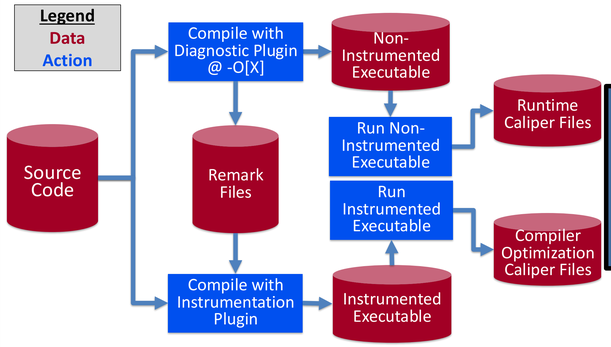


1. Too many remarks
2. No runtime context

Our Contributions

1. We develop a compiler plugin-based methodology that extracts and exposes compiler optimization information at runtime
2. We conduct a case study to determine the compiler optimization requirements of RAJA Performance Suite kernels

LLVM Pass Plugins to Collect Compiler Remarks



We utilize LLVM pass plugins to:

1. Extract remarks at compile time
2. Inject remarks into runtime output

Remark Instrumentation

Original Code

```
void foo() {
  bar() <-- inline occured here

}

int main() {
  foo()
}
```

Instrumented Code

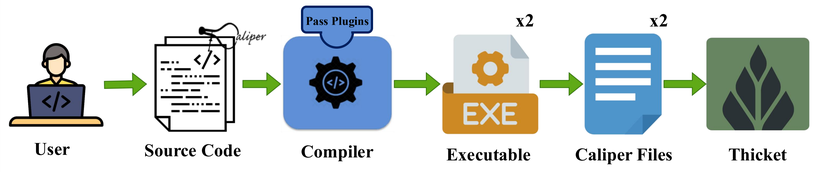
```
void foo() {
  cali_begin_string("Inline")
  bar() <-- inline occured here

}

int main() {
  foo()
}
```

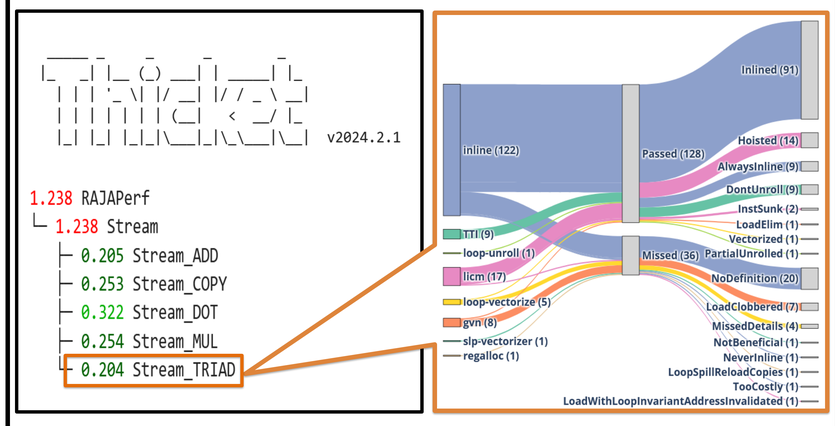
Remark

```
{
Function: Foo
Pass Name: Inline
...
}
```

Details about remarks are instrumented at their respective source code locations

Compiler Optimizations and Runtime Performance

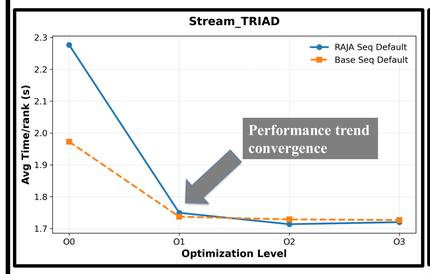




Using Thicket to analyze the optimization information of critical regions

RAJA Performance Suite Case Study

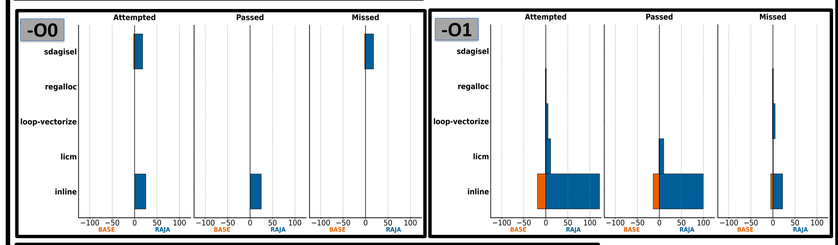
RAJA and Base implementations of Stream_TRIAD at different compiler optimization levels

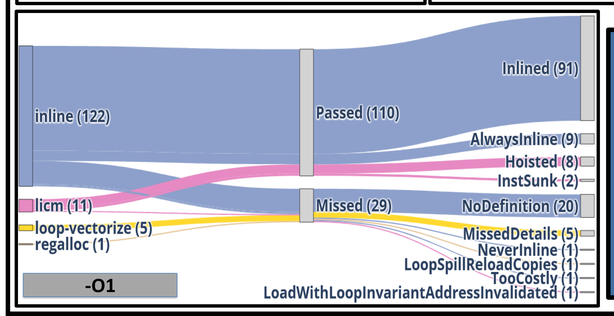


- RAJA implementation of the kernel is significantly slower than Base implementation when using optimization level lower than –O1
- Performance trends of the RAJA and Base implementations converge at –O1 and higher optimization levels

Key Question:

- Which compiler optimizations are critical for the RAJA implementation to achieve performance comparable to the Base implementation?





Inlining is by the far the most active optimization

- The RAJA implementation needs much more inlining
- Auto-inlining accounted for most of the inlining

Key Takeaways

- Our methodology allows us to see compiler optimizations applied to kernels along with their impact on runtime performance
- We observe that RAJA-based kernels, such as Stream_TRIAD, rely heavily on inlining for performance
- RAJA matches Base performance trends beyond –O1, primarily due to inlining

Future Work

- Extend methodology to support GPU applications
- Reduce instrumentation overhead
- Enhance our methodology by assigning importance factors to specific optimizations

References

[1] A. Beckingsale et al., "RAJA: Portable Performance for Large-Scale Scientific Applications", in Proc. IEEE/ACM Int'l Workshop on Performance, Portability and Productivity in HPC (P3HPC), Denver, CO, USA, 2019.

[2] D. Boehme et al., "Caliper: Performance Introspection for HPC Software Stacks", in Proc. Int'l Conf. for High Performance Computing, Networking, Storage and Analysis (SC16), Salt Lake City, UT, USA, 2016.

[3] R. Pearce et al., "RAJA Performance Suite: Performance Portability Analysis with Caliper and Thicket", in Proc. IEEE/ACM Int'l Workshop on Performance, Portability and Productivity in HPC (P3HPC), held with Int'l Conf. for High Performance Computing, Networking, Storage and Analysis (SC-W), Dallas, TX, USA, 2024.

[4] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation", in Proc. Int'l Symp. on Code Generation and Optimization (CGO), Palo Alto, CA, USA, 2004.

[5] Lawrence Livermore National Laboratory, "RAJA: Managing Application Portability for Next-Generation Platforms", [Online]. Available: https://computing.llnl.gov/projects/raja-managing-application-portability-next-generation-platforms.

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-POST-2009376)