Analyzing the Effect of Compiler Optimizations

89th ROOT PPP meeting

Jonas Hahnfeld

07/01/2021

i12 High Performance Computing
RWTH AACHEN UNIVERSITY

Motivation

- HPC applications run on multiple systems
  - Different hardware architectures and configurations
  - Varying software environments
  - → Write portable code and rely on compiler for optimizations

Motivation

- HPC applications run on multiple systems
  - Different hardware architectures and configurations
  - Varying software environments

  → Write portable code and rely on compiler for optimizations

- Reality shows: different compilers have different optimization strategies
  - Due to different implementations, different heuristics, etc.
  - Manifests in varying performance of the generated code
  - No structured approach in literature to understand the effect of compiler optimizations

Motivation

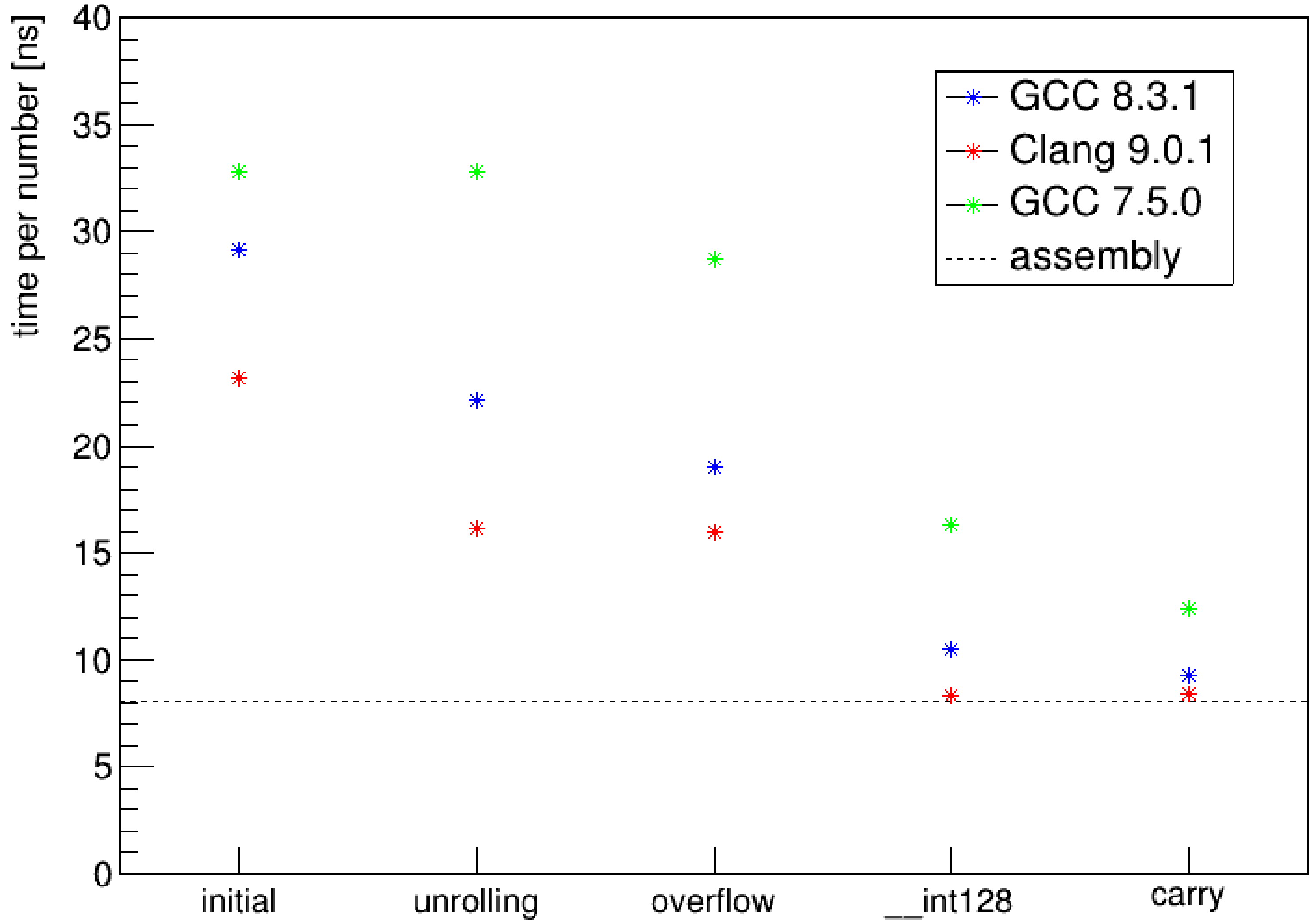
- HPC applications run on multiple systems
  - Different hardware architectures and configurations
  - Varying software environments

  → Write portable code and rely on compiler for optimizations

- Reality shows: different compilers have different optimization strategies
  - Due to different implementations, different heuristics, etc.
  - Manifests in varying performance of the generated code
  - No structured approach in literature to understand the effect of compiler optimizations

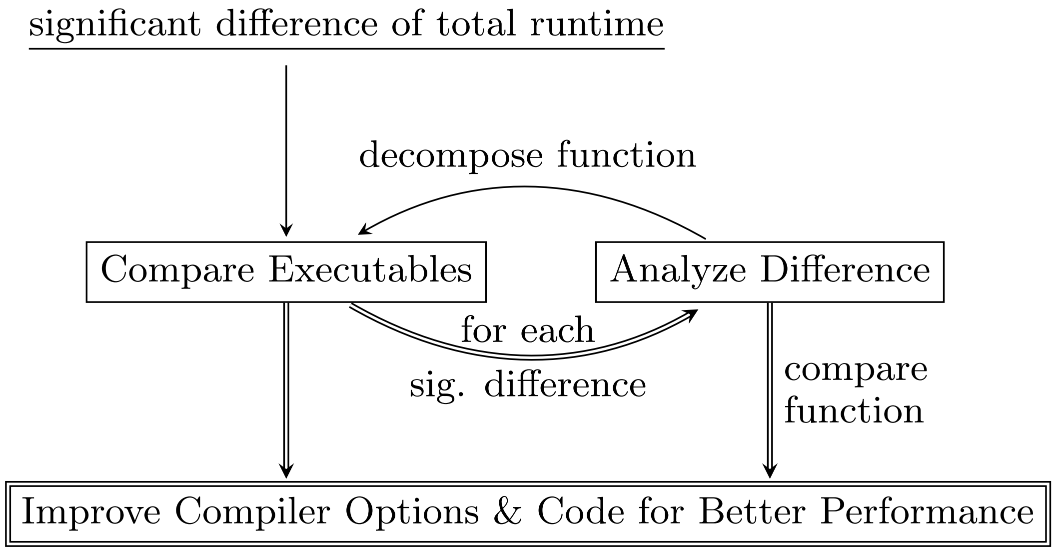
- Workflow to locate and analyze such performance differences

(Part of the) Motivation for ROOT: RANLUX++



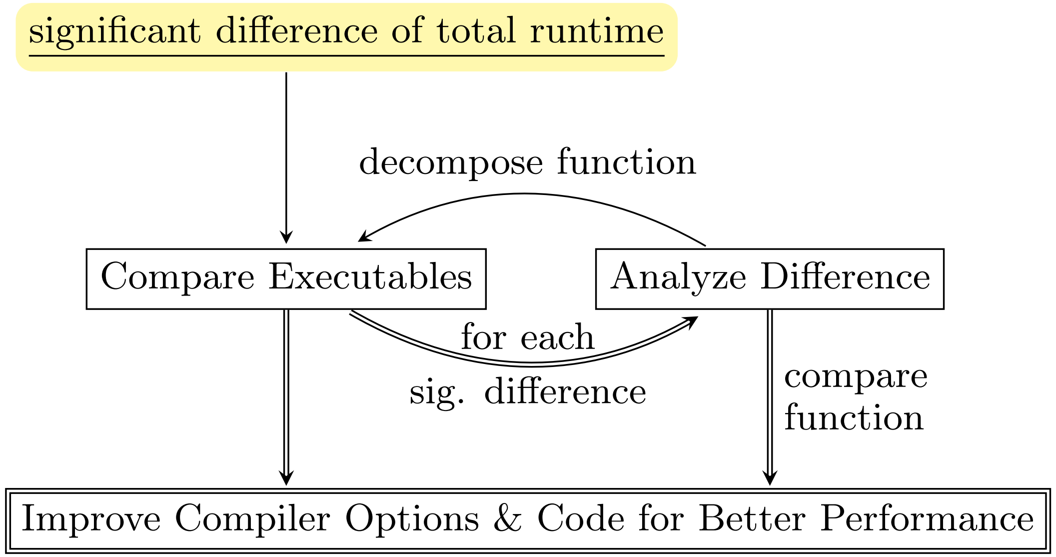
Overview of the Workflow

- Scenario:
  - Two executables from unique build configurations



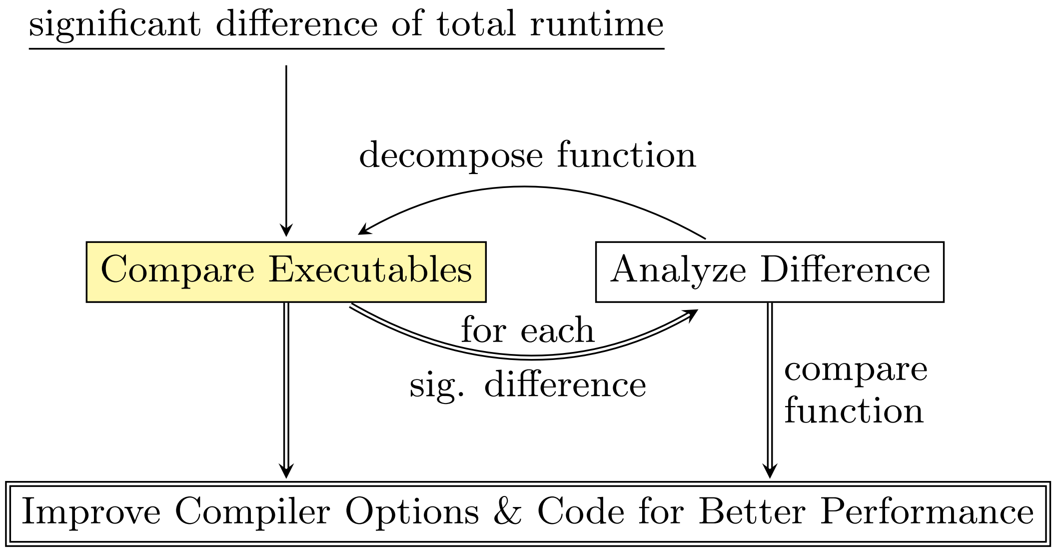
Overview of the Workflow

- Scenario:
  - Two executables from unique build configurations



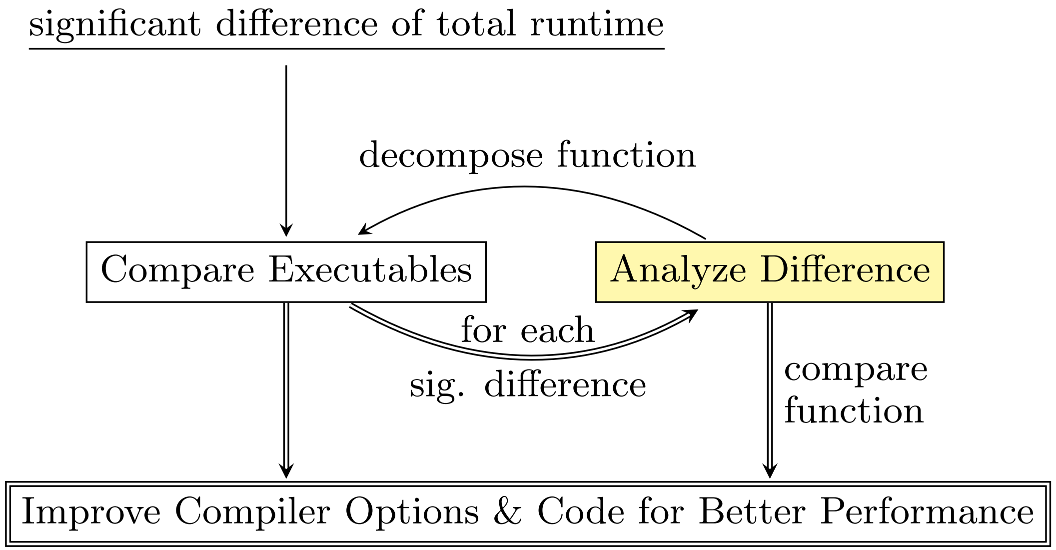
Overview of the Workflow

- Scenario:
  - Two executables from unique build configurations
- Strategy: divide-and-conquer
  - Compare profiling data to find differences in functions

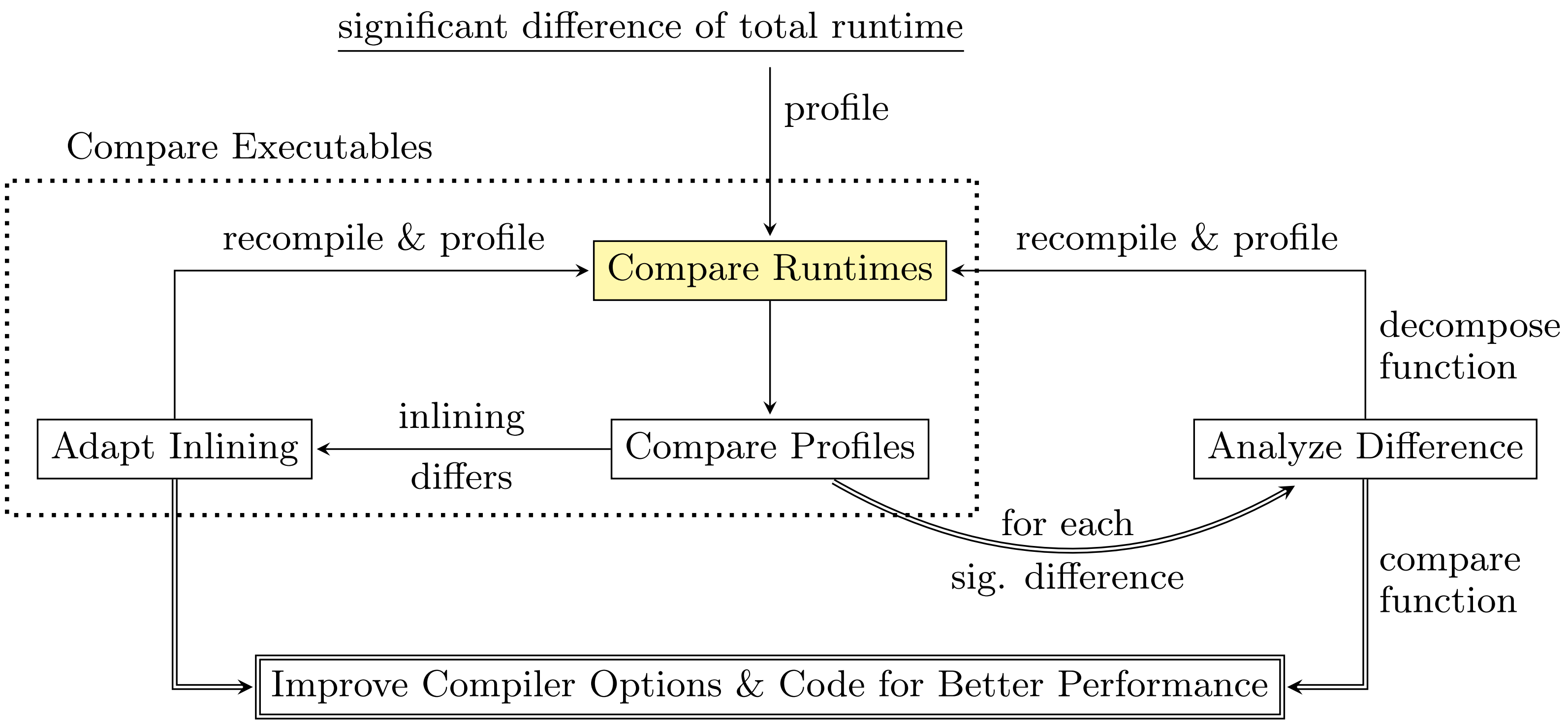


Overview of the Workflow

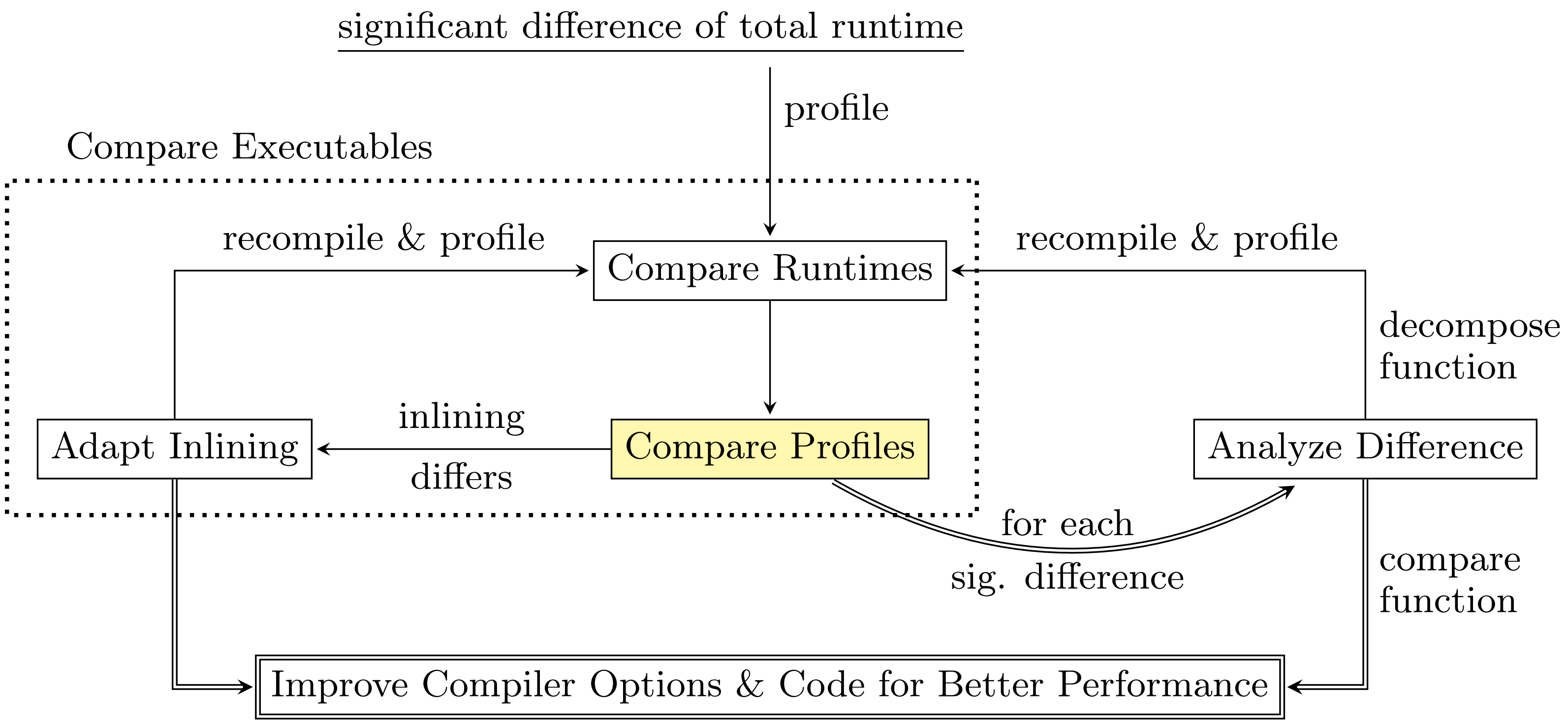
- Scenario:
  - Two executables from unique build configurations
- Strategy: divide-and-conquer
  - Compare profiling data to find differences in functions
  - Decompose functions to increase level of detail



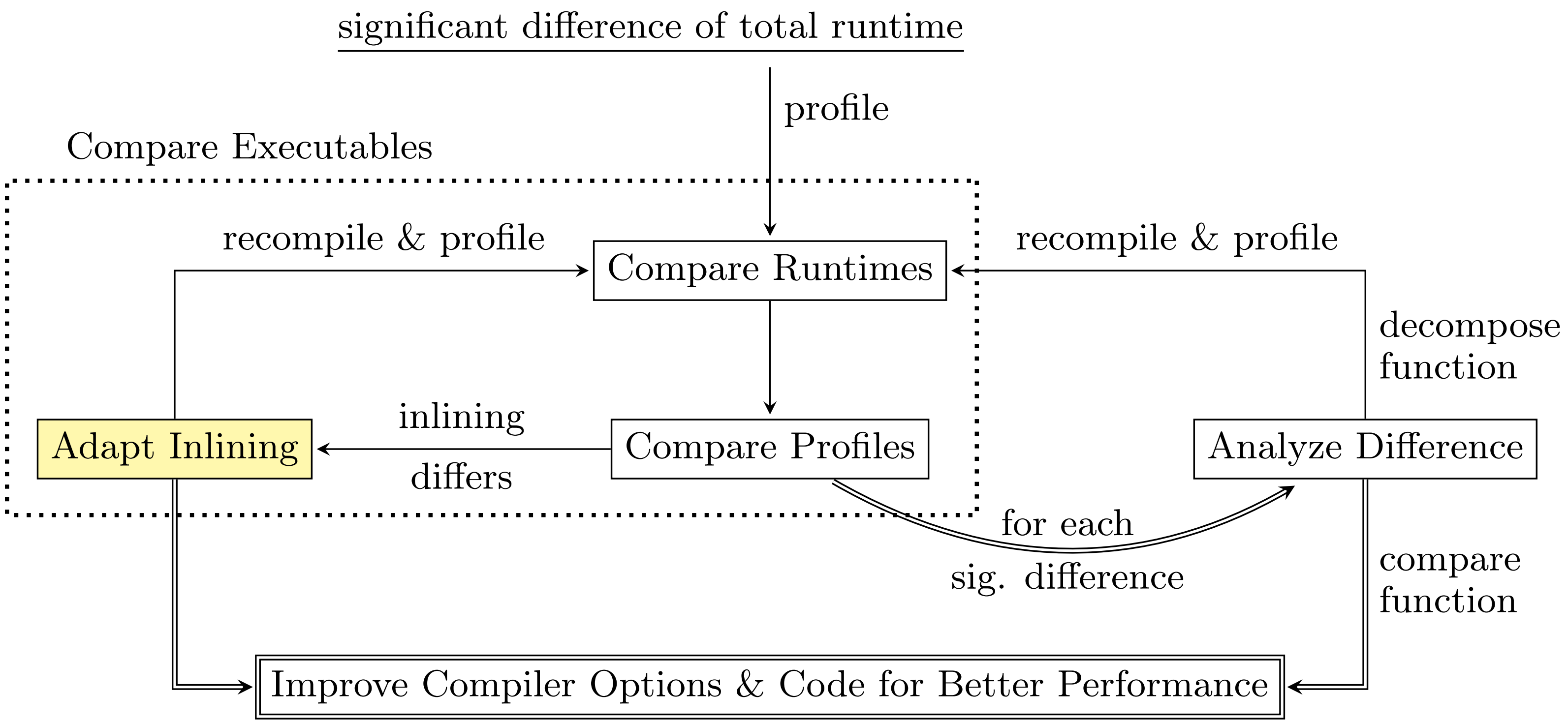
Compare Executables



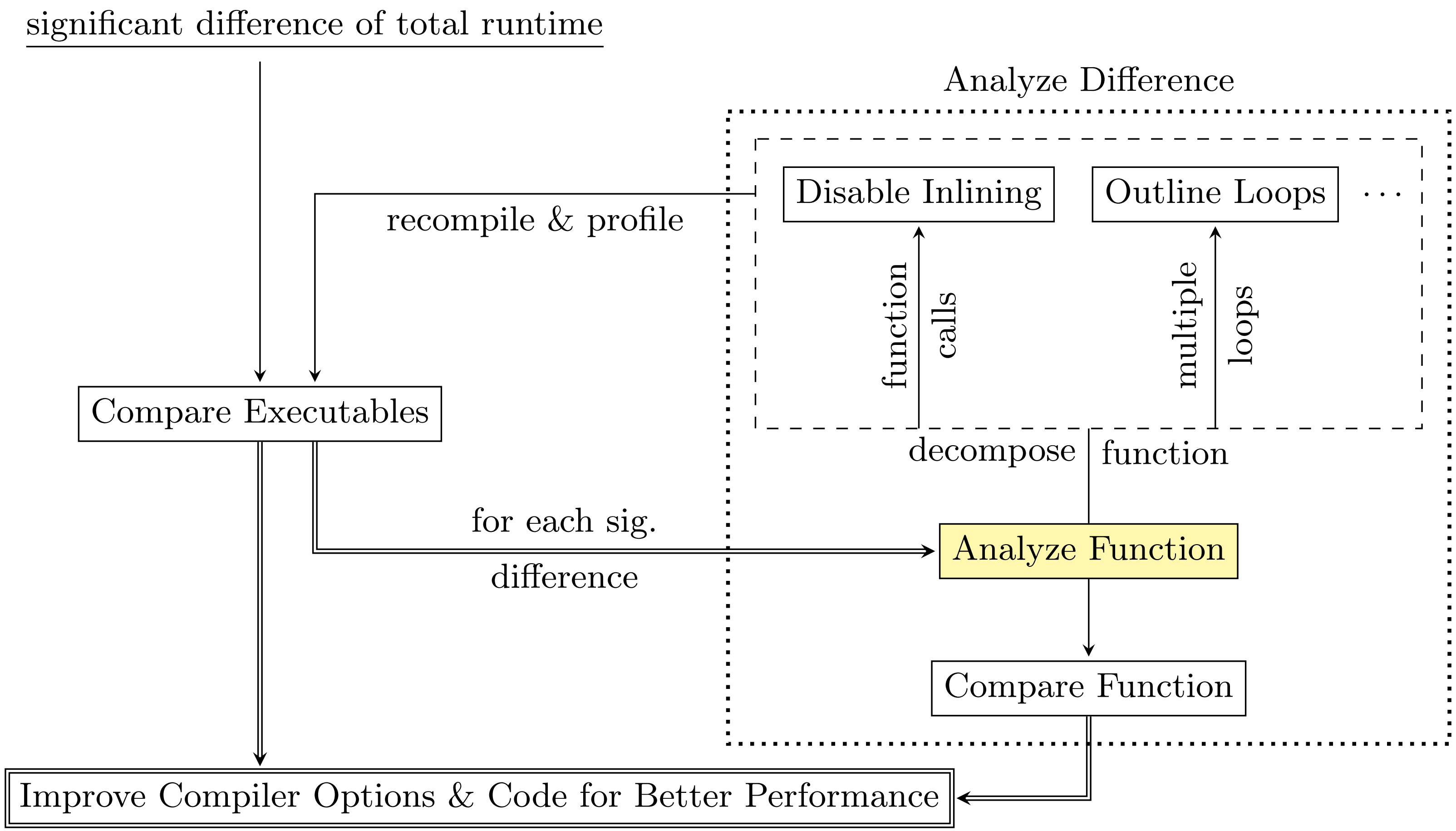
Compare Executables



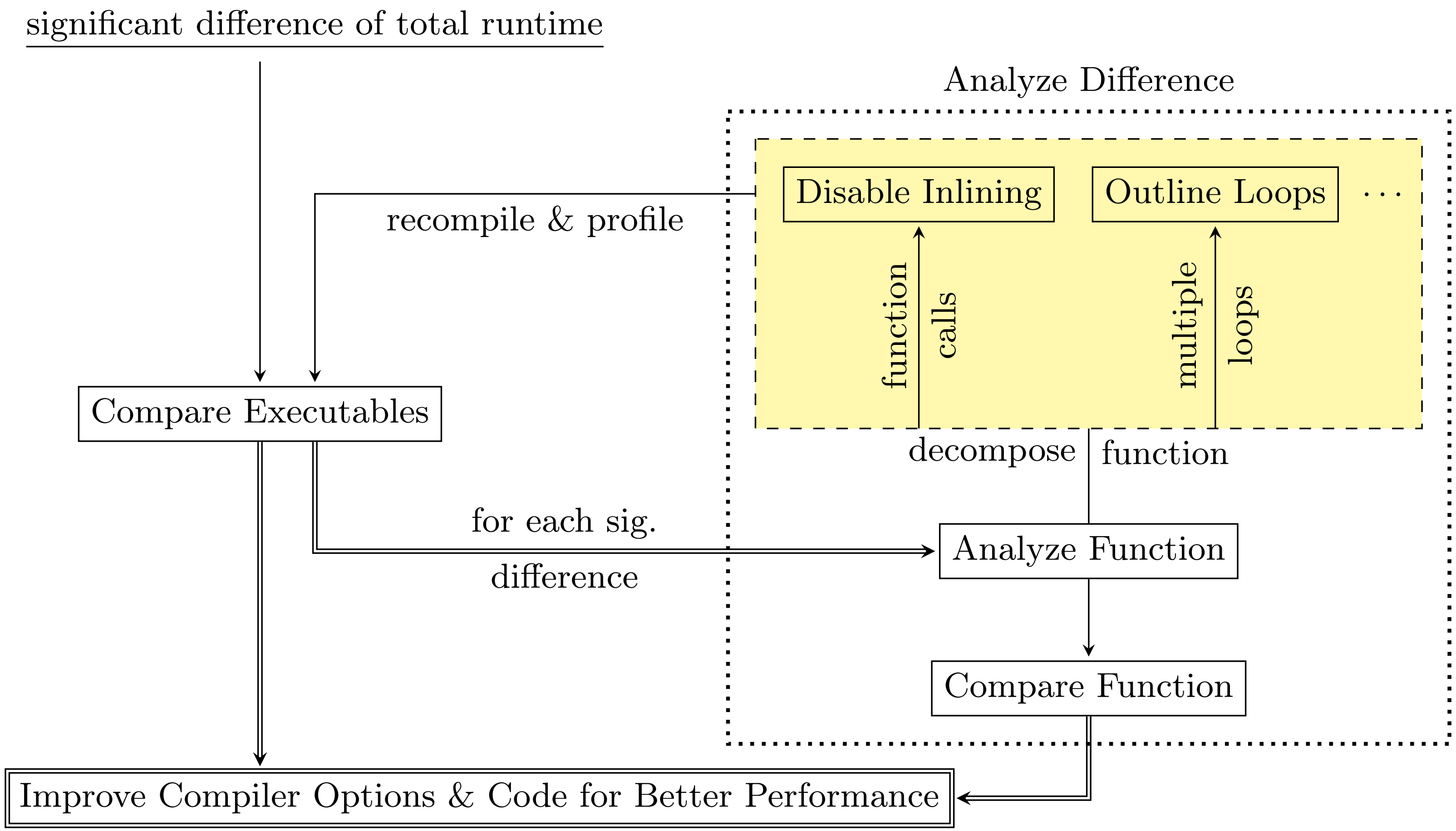
Compare Executables



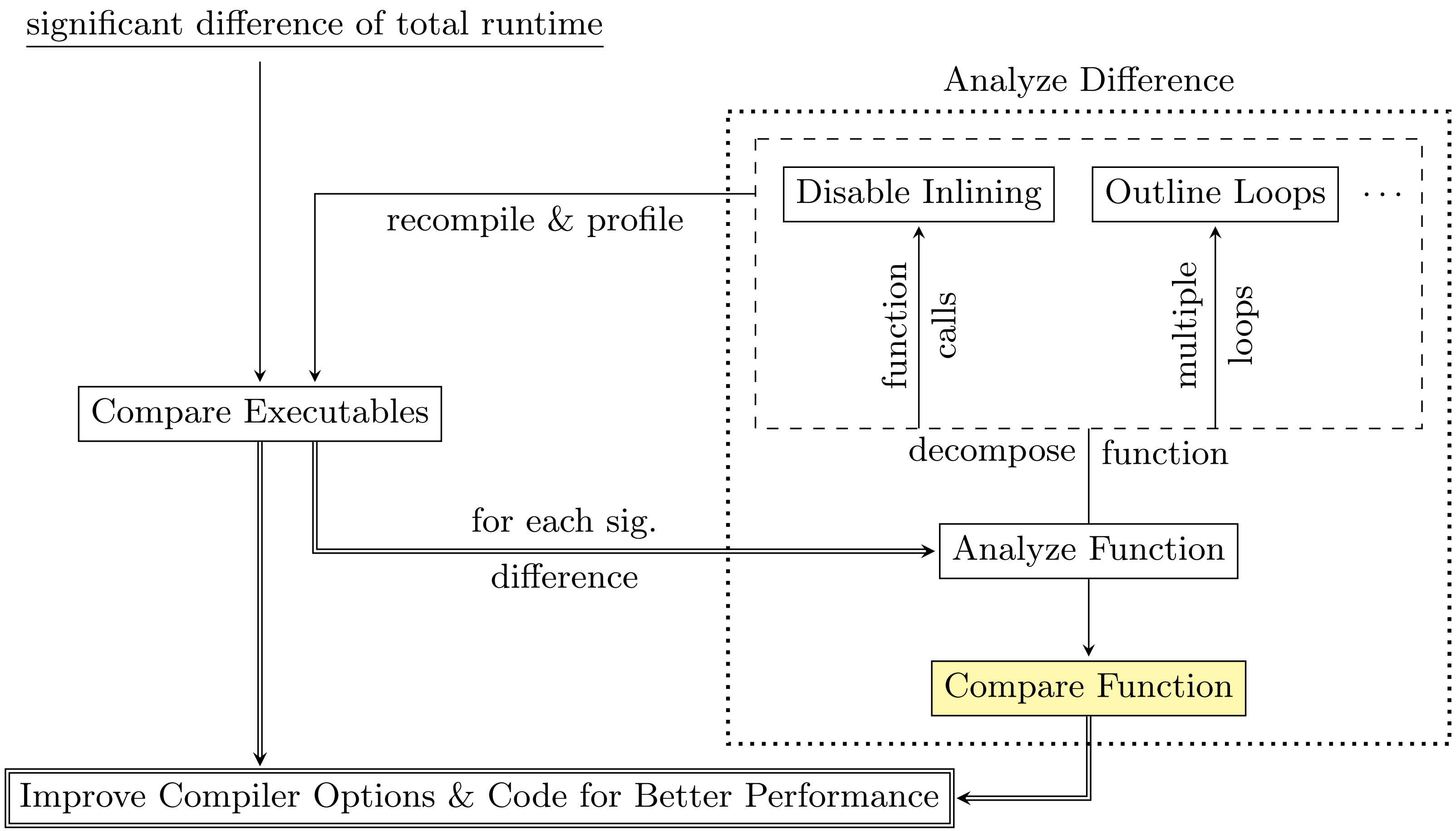
Analyze Difference in Function



Analyze Difference in Function

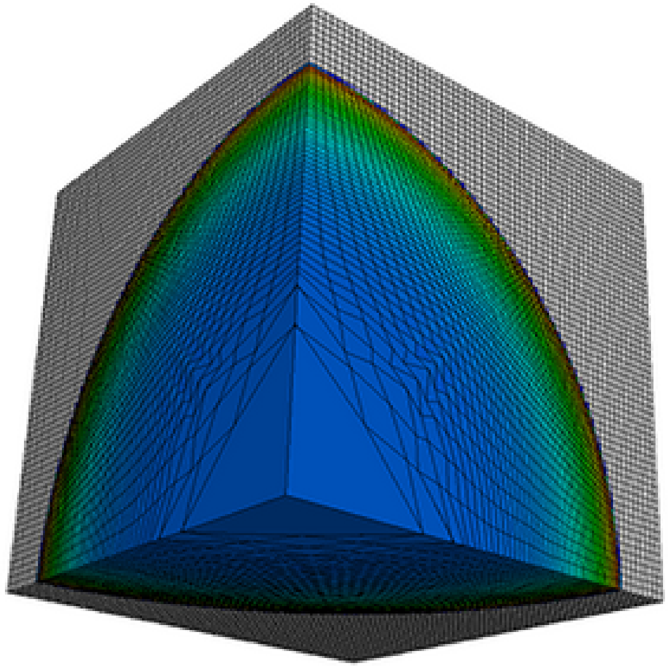


Analyze Difference in Function



Example: LULESH

- **L**ivermore **U**nstructured **L**agrangian **E**xplicit **S**hock **H**ydrodynamics
- Written in C/C++ and representative code structure

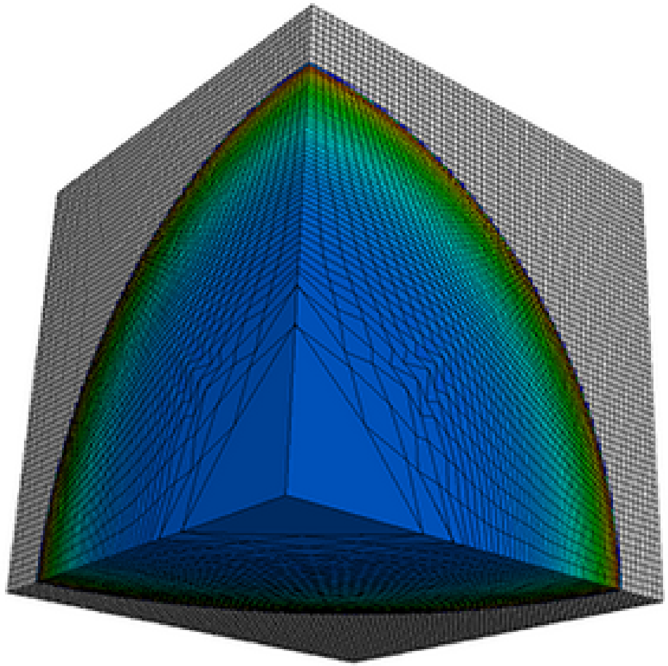


https://computing.llnl.gov/projects/co-design/lulesh

Example: LULESH

- **L**ivermore **U**nstructured **L**agrangian **E**xplicit **S**hock **H**ydrodynamics
- Written in C/C++ and representative code structure

- Setup:
  - Measurements on single core of an Intel Xeon Platinum 8160 (“Skylake”, 2.1 GHz)
  - Profiling with Linux `perf` (cf. presentation by Guilherme)



https://computing.llnl.gov/projects/co-design/lulesh

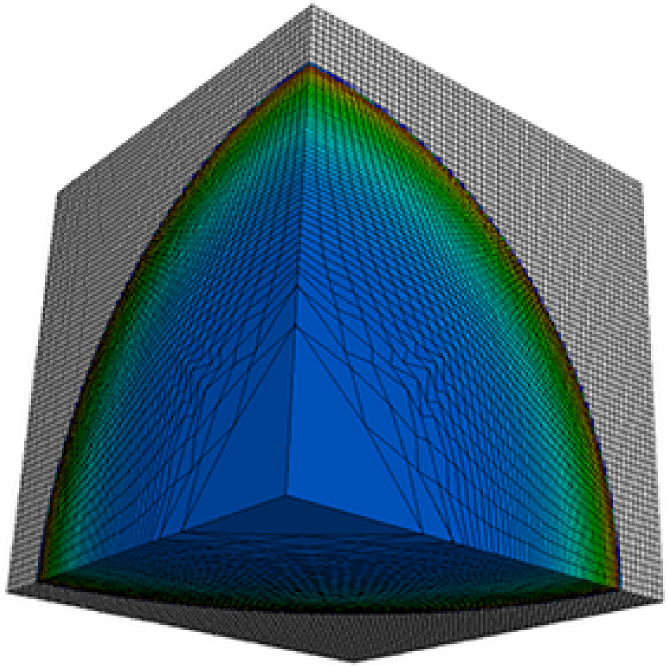
i12 High Performance Computing | RWTH AACHEN UNIVERSITY

Example: LULESH

- **L**ivermore **U**nstructured **L**agrangian **E**xplicit **S**hock **H**ydrodynamics
- Written in C/C++ and representative code structure

- Setup:
  - Measurements on single core of an Intel Xeon Platinum 8160 (“Skylake”, 2.1 GHz)
  - Profiling with Linux `perf` (cf. presentation by Guilherme)

- Measurements:
  - Clang 10.0.0: 93.4 s
  - Intel Compiler 19.1.1.217: 78.6 s
  - Relative difference: $\approx$ 16 %



https://computing.llnl.gov/projects/co-design/lulesh

LULESH: live demo

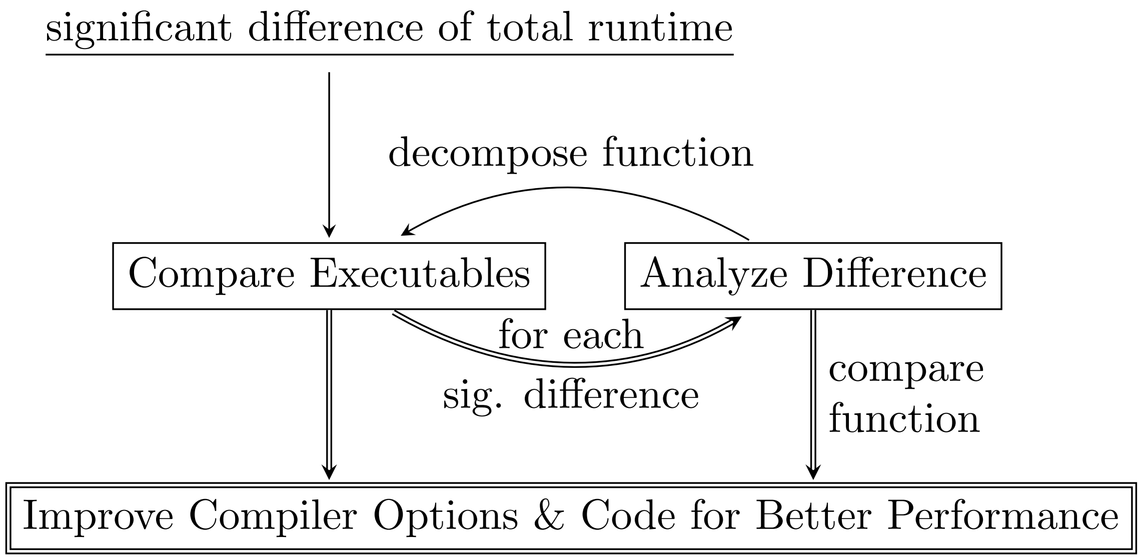
LULESH: Summary

- Original measurement with Clang: 93.4 s
  - Intel Compiler: 78.6 s
- Add attribute `always_inline` to `CalcElemShapeFunctionDerivatives`
  - Runtime improves by around 5 % (88.7 s)
- Implement loop fusion portably in the source code
  - Improves runtime with Clang to 80.4 s (another 9 %)

⇒ Performance improvement of 14 % compared to original version

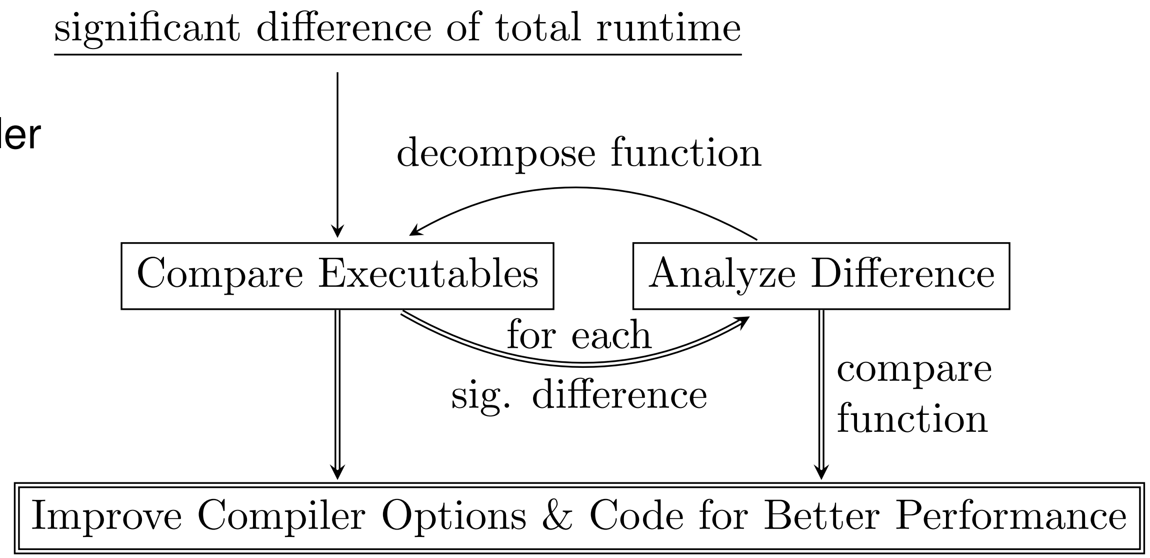
Possible Use Cases

- Main use case: performance tuning of portable applications
  - Directly leads to parts of the code that can be improved



Possible Use Cases

- Main use case: performance tuning of portable applications
  - Directly leads to parts of the code that can be improved
- Also possible: (see thesis for case study of miniMD)
  - Analyze differences between two versions of the same compiler
  - Use case: investigate regressions in optimization pipeline

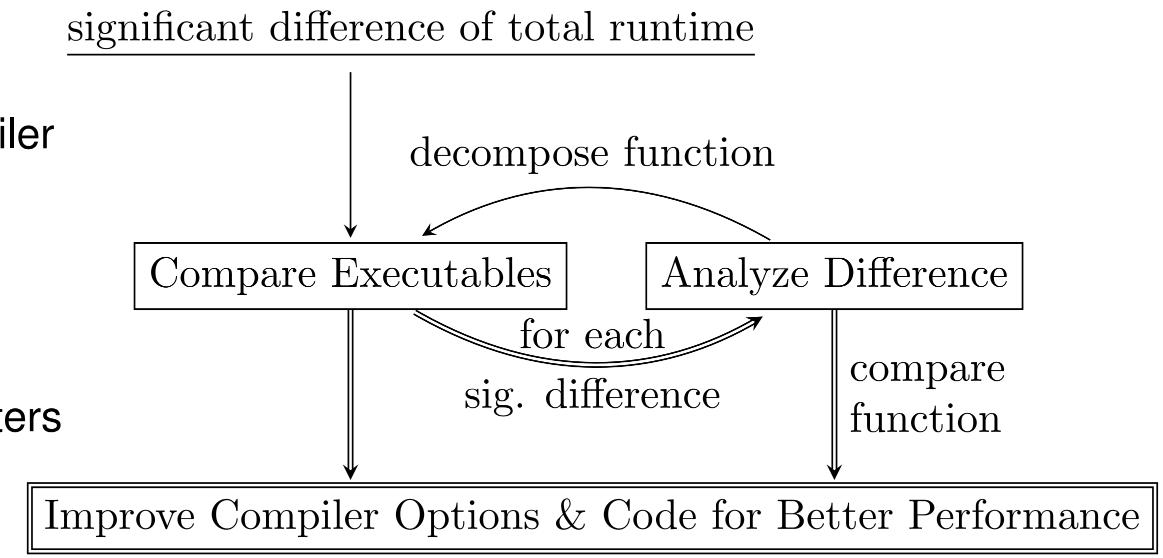


Possible Use Cases

- Main use case: performance tuning of portable applications
  - Directly leads to parts of the code that can be improved

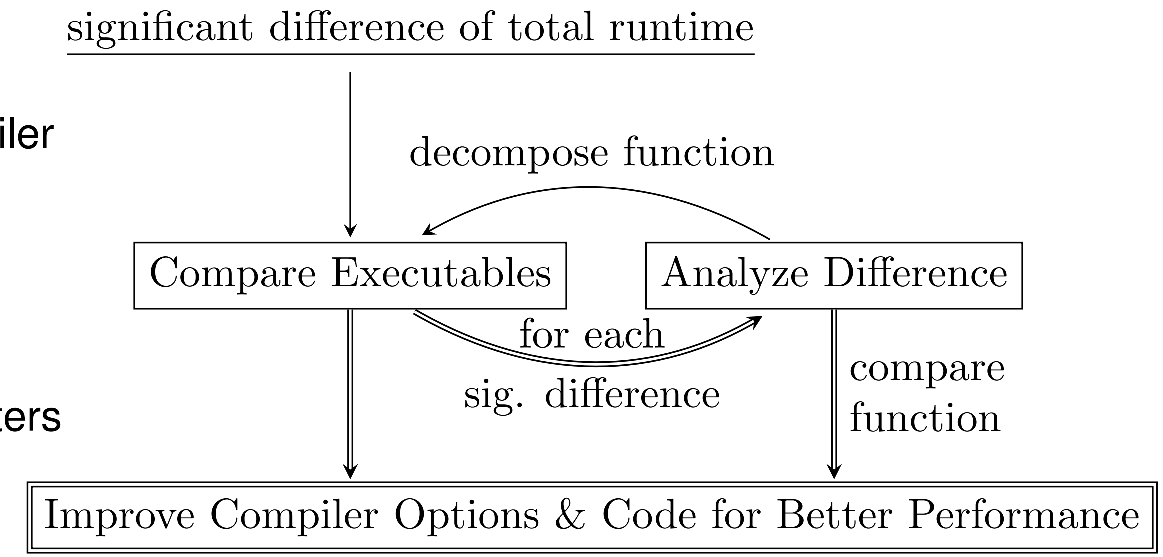
- Also possible: (see thesis for case study of miniMD)
  - Analyze differences between two versions of the same compiler
  - Use case: investigate regressions in optimization pipeline

  - Comparison of different compiler flags
  - Use case: analyze individual optimizations and their parameters



Possible Use Cases

- Main use case: performance tuning of portable applications
  - Directly leads to parts of the code that can be improved
- Also possible: (see thesis for case study of miniMD)
  - Analyze differences between two versions of the same compiler
  - Use case: investigate regressions in optimization pipeline
  - Comparison of different compiler flags
  - Use case: analyze individual optimizations and their parameters



- Not investigated yet: mutually improve binaries
  - Promising for cases where no binary is the fastest for all functions
  - Idea: choose the best of two in each case, improve overall performance