Intelligent Selection of Compiler Options to Optimize Compile Time and Performance

Anja Gerbes[1], Julian Kunkel[2], Nabeeh Jumah[3]

[1]*Center for Scientific Computing (CSC)* [2]*Deutsches Klimarechenzentrum (DKRZ)* [3]*Universität Hamburg*

Introduction

- Efficiency of the optimization process during compilation is crucial for the later execution behavior of the code.
- Achieved performance depends on hardware architecture and compiler's capabilities to extract this performance.
- Optimization influences the debuggability of the resulting binary; for example, by storing data in registers.
- During development
  - Code optimization can be a CPU- and memory-intensive process which – for large codes – can lead to high compilation times.
  - Compile files individually with appropriate flags enable debugging and provide high performance during the testing but with moderate compile times.

This example shows the dependency of compile time vs. optimization level. Between `-O0` and `-O3` is a compile time 2x.:

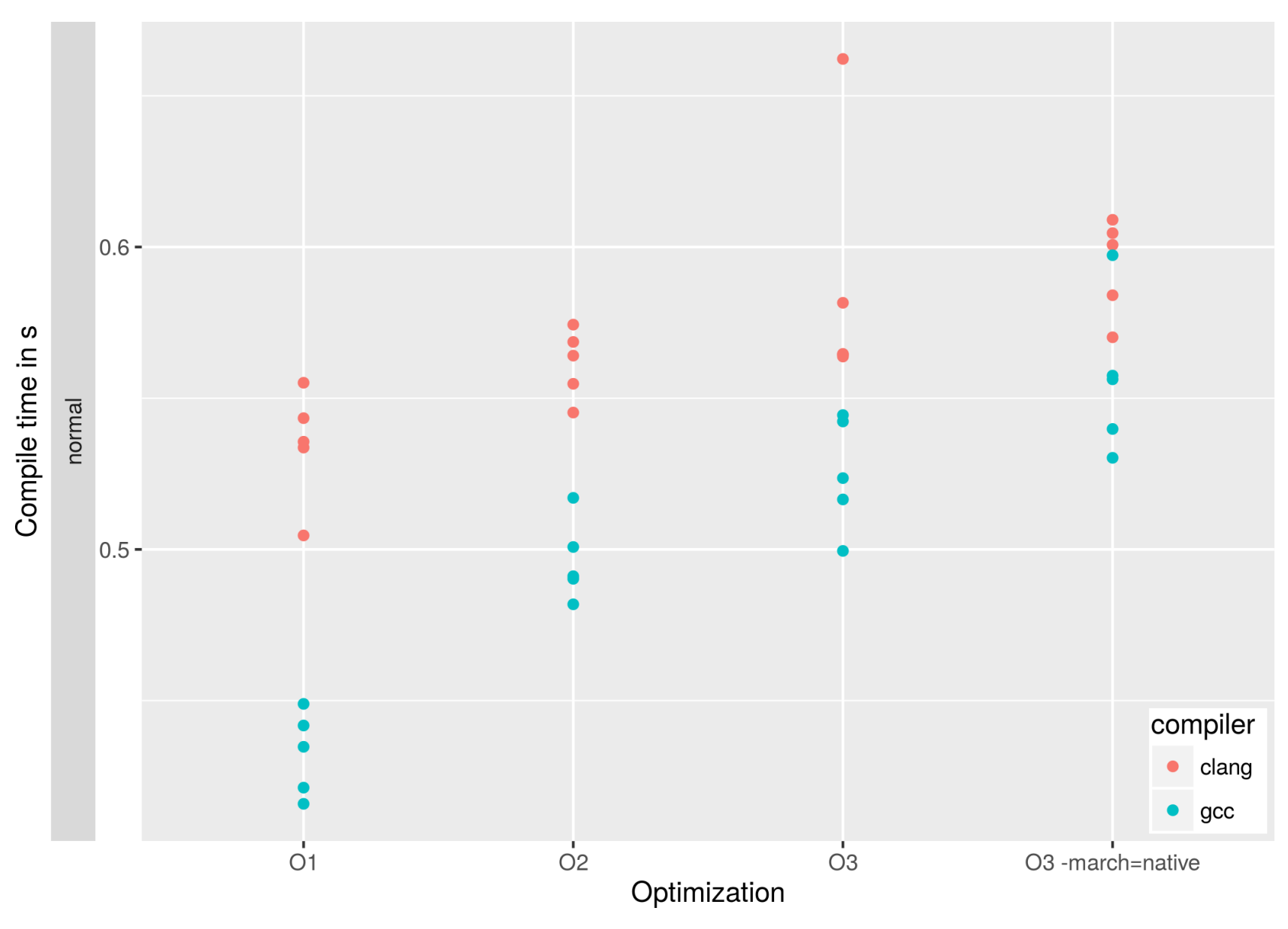

Fig. 1: Compile time for our proxy application

Approach

Create a tool to identify code regions that are candidates for higher optimization levels. We follow two different approaches to identify the most efficient code optimization:

- compiling different files with different options by brute force
- using profilers and code instrumentation to identify the relevant code regions that should be optimized

After using these approaches:

- The relevant files are evaluated with different compiler flags to determine a good compromise of the flags.
- Once the appropriate flags are determined, this information could be shared between users.

Application Domain

- Climate and atmospheric sciences.
- Grid-bound variables.
- Codes carrying out stencil operations.

Experiments Environment

Proxy Application

A simple climate application is used:

- Time-stepped application.
- Three operators in each time step.
- Stencil operations to update grid-bound variables in each operator.
- Different code structures developed.
- Different Memory Layouts for grid-bound variable storage
  - 3D Euclidean space
  - 1D transformation of 3D Euclidean space
  - 3D Hilbert filling curve
  - HEVI 2D Hilbert filling curve with vertical dimension
- Support for different data types
  float double int

Testing machine

Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz

Analysis Toolsuite

Main task of the analysis toolsuite is to identify the relevant codes and the optimization flags.

Brute force strategy:

1. Compile each C file into an object file using `-O0` and `-O3`.
2. Iterate over all files, link one file at a time with `-O3`, all others with `-O0`, or otherwise.
3. Run the application, measure & record performance.
4. Compare performance & compilation time for the different variants to identify relevant files.
5. Test variants of even better compilation options for the relevant files.

One problem of this strategy is, that codes need a number of iterations == # files to identify relevant files.
Profiler are able to identify time consuming regions/files & to allow repeatedly refine the selection.

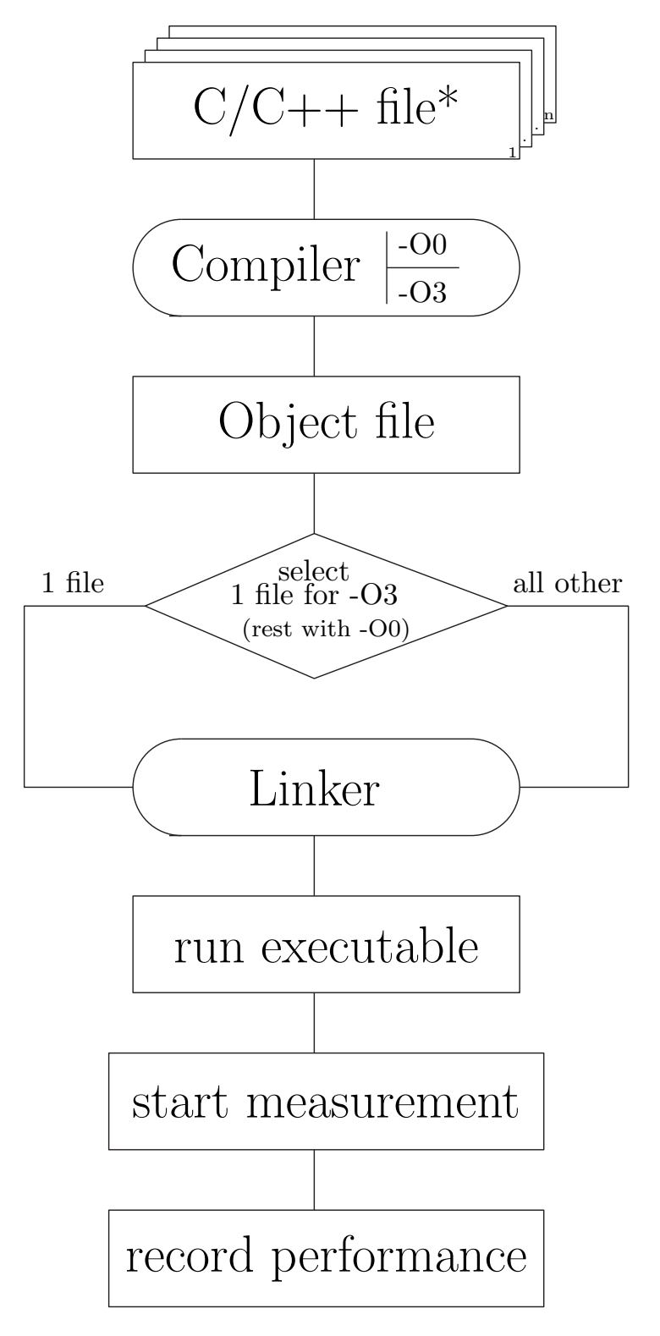

Fig. 2: brute force strategy

The profiler-guided selection of files replaces Step 1 and 2 by running the application to identify the relevant functions and, thus, the files that make up most of the execution time. Consider those files to be relevant in Step 5. This process can be repeated until no more relevant files can be identified.

We consider to explore the usage of LIKWID and Oprofile for this task:

1. Instrument the functions using LIKWID marker API for annotating the file name. The problem in using the LIKWID profiler could be the high overhead of instrumentation for short functions.
2. Using Oprofile or perf allows to explore performance without instrumenting the source code but requires parsing.

When using a profiler that requires code instrumentation such as LIKWID, source-to-source code transformation can be used.

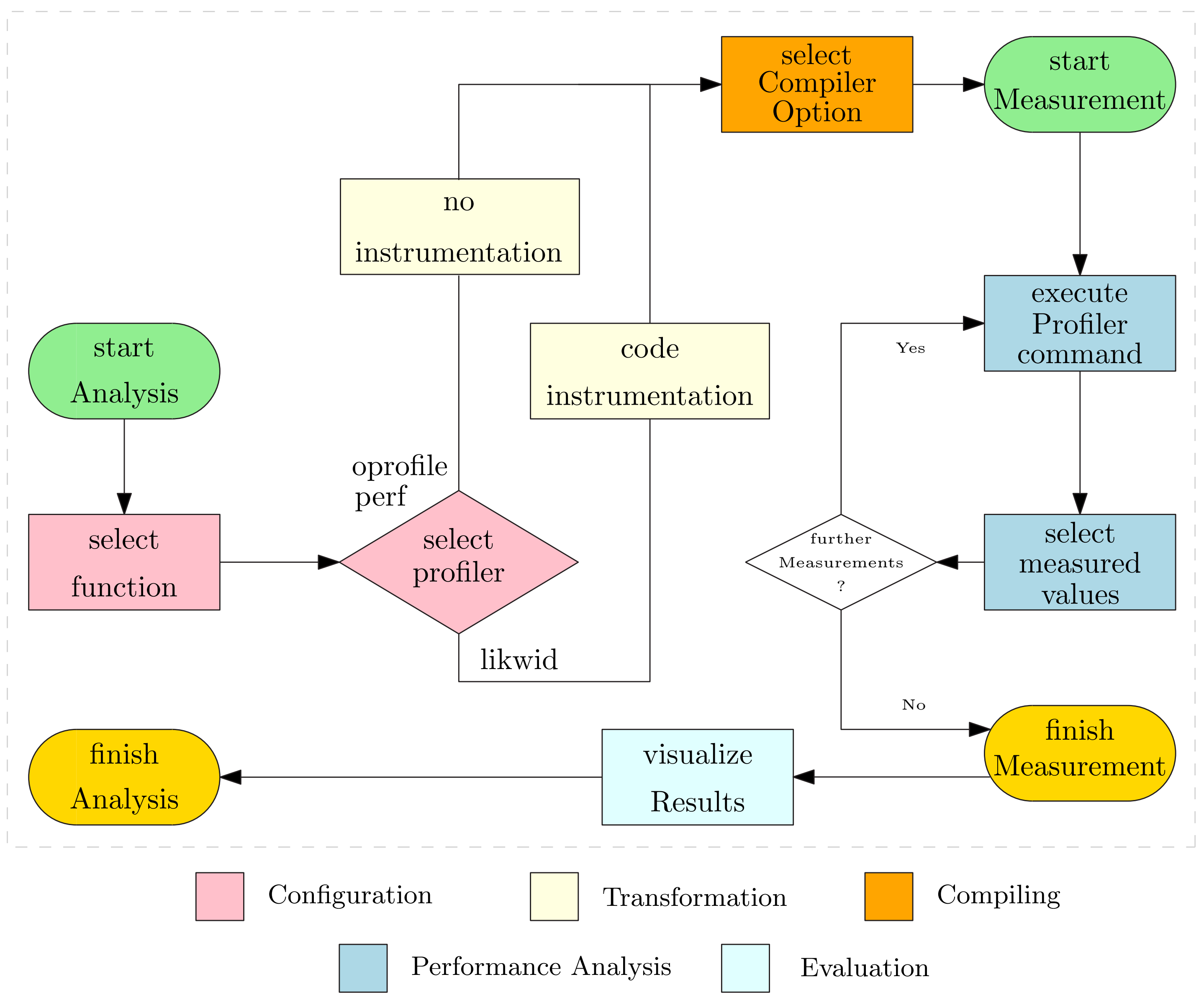

Fig. 3: profiler strategy

Results

The results show that more time (2×) is spent for compiling code using higher optimization levels in general, through gcc takes a little less time in general than clang.

O3	compiler	pgo	compile time	GFLOPs for gridsize mean	32	64	128	256
all	gcc	-	0.548	2.450	2.70	2.95	2.09	2.06
all	gcc	steps	0.426	2.420	2.58	2.80	2.18	2.12
all	gcc	profile	0.492	2.430	2.54	2.89	2.16	2.13
all	clang	-	0.594	2.517	2.76	3.05	2.08	2.18
all	clang	steps	0.490	2.157	2.48	2.57	1.77	1.81
div/grad/step	gcc	-	0.414	2.879	2.86	3.25	2.74	2.66
div/grad/step	gcc	steps	0.325	3.222	2.81	3.95	3.09	3.04
div/grad/step	gcc	profile	0.346	2.939	2.86	3.23	2.71	2.95
div/grad/step	clang	-	0.507	3.473	3.37	4.36	3.04	3.13
div/grad/step	clang	steps	0.387	3.040	3.26	3.57	2.67	2.67

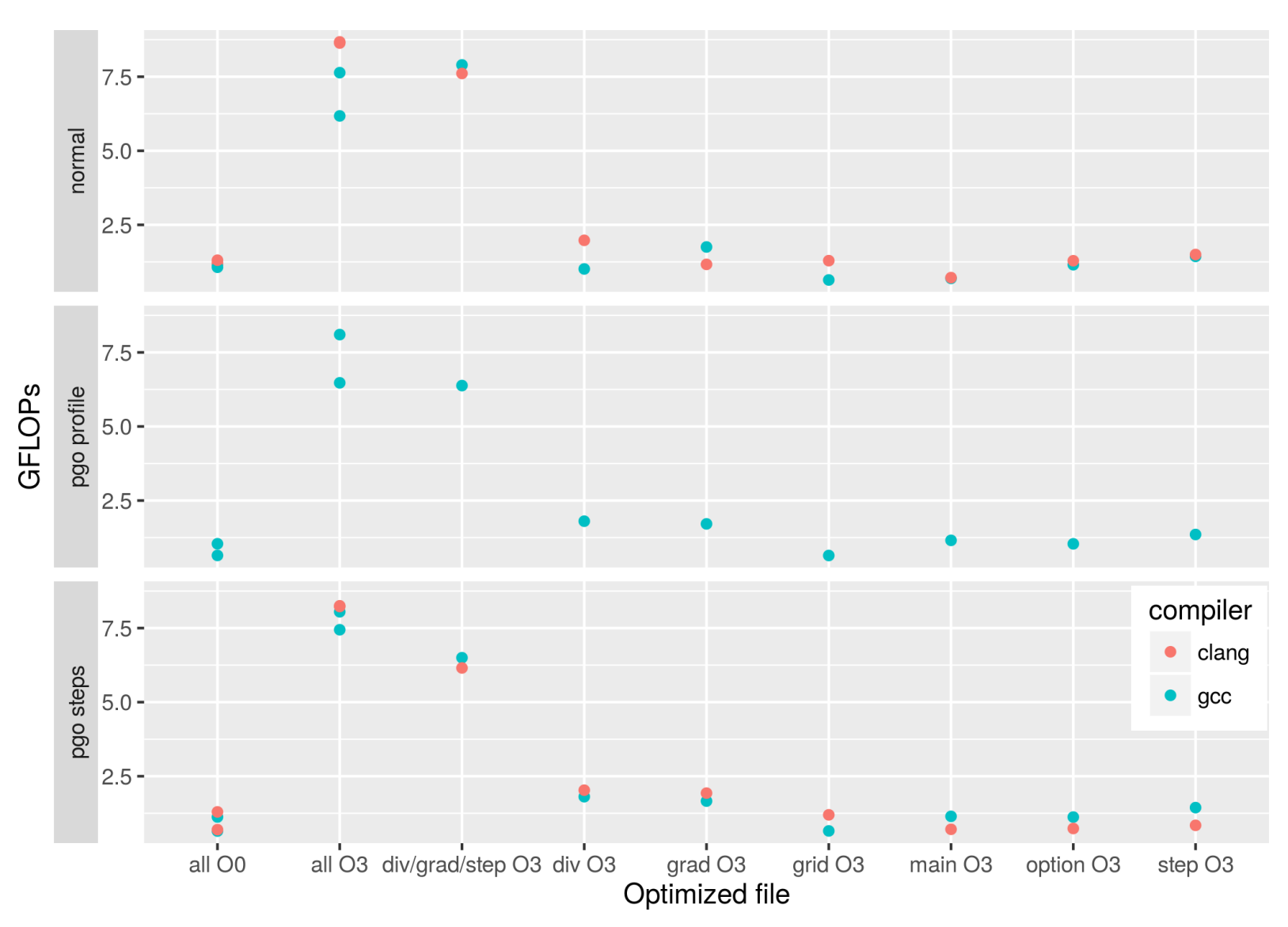

Fig. 4: Performance (for 4 threads, double precision, different grid sizes)

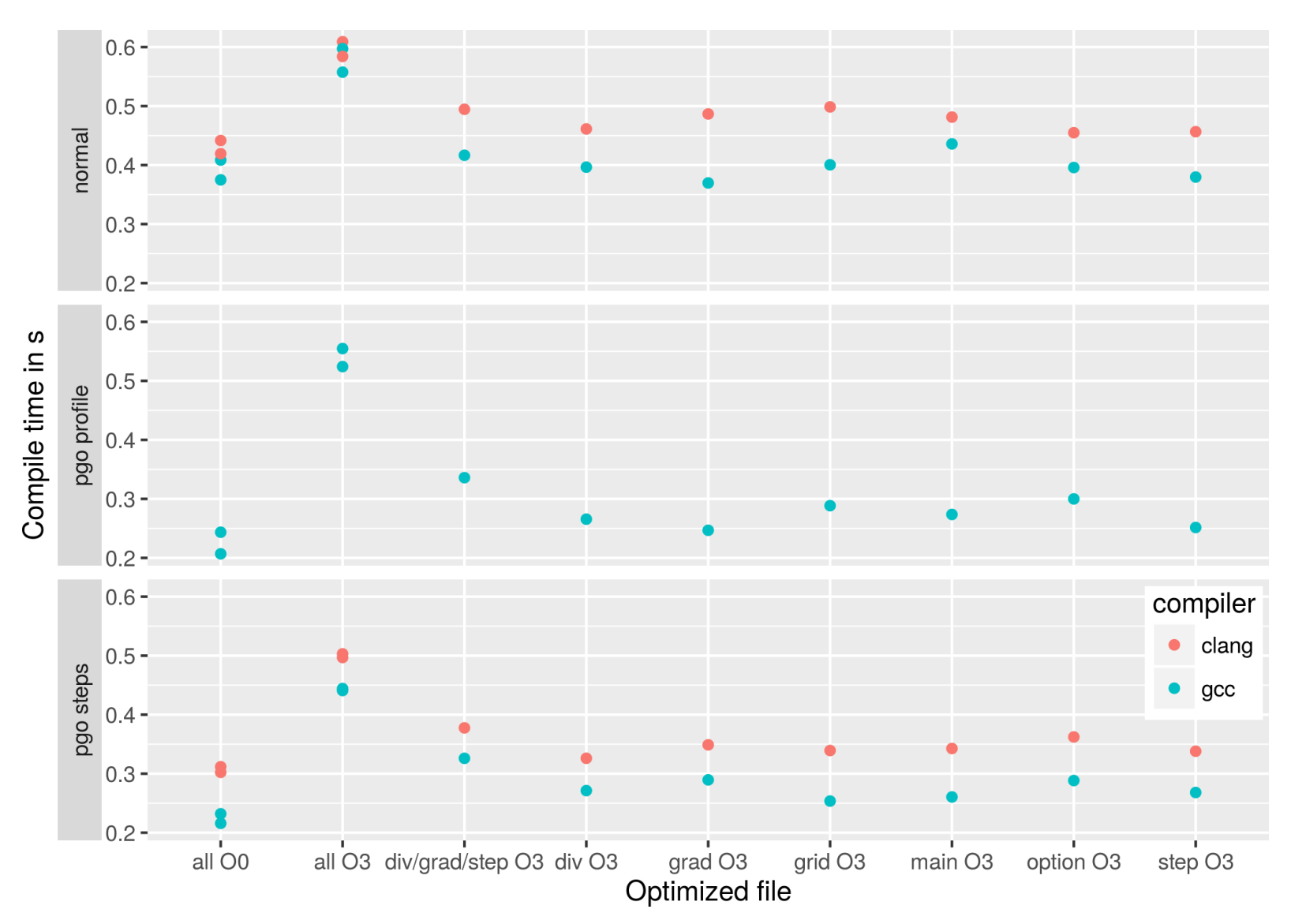

Fig. 5: Compile time when compiling/linking files with different optimization levels

Figures show compiling files individually, also show when all files are compiled with either `-O0` or `-O3 -march`. Compiling a single file with `-O3 -march` is not so expensive, subset (div/grad/step) is still faster. Optimizing this subset of files yields nearly the optimal performance.

LLVM Clang

- LLVM's CLang is not only compilers language
- CLang exposes an internal data structure for analyzing code
- CLang provides mechanisms for traversing the Abstract Syntax Tree (AST) nodes
- Analysis Toolsuite scans a set of source files looking for types of functions, via AST
- AST Matcher API provides a Domain Specific Language (DSL) for matching predicates on CLang's AST.
- CLang's LibTooling transforms applications code to add the LIKWID Marker API start-stop-directives
- Code instrumentation on function level is a preparation of the profiling step

Explored Configurations

- We explored compilation & performance while varying different settings.
- Compilation process & performance are tested under different
  - code structures
  - memory layouts
  - grid sizes
  - numbers of threads to run code
  - data types
  - compilers
    - GCC version 6.3.0
    - CLang version 3.9.1
  - Variety of compiler/linking options:
    - `-O0` - `-O3`
    - `-march`
    - PGO options
- Generated combinations had been run and performance results analyzed.

Summary

- An application is developed to experiment compilation improvement while preserving performance.
- A tool is built to produce a set of optimization options during compilation process for the set of files comprising the code repository.
- Iterative process of applying high optimizing compiler options to a file at a time is done to find the set of files critical for application performance.
- Brute force search within different application configurations is undertaken to drive further compilations.
- PGO capabilities of compilers are exploited in experiments.
- The results show success to achieve comparable performance of applications compared to compilations done with high compiler options for all code components.

Future Work

- Further enhancements will be done.
- Analysis Toolsuite does not remember good optimization choices. Transfer of information learned in a compilation to serve subsequent compilations is subject to future work.
- Compilation time reduction and code updates are considerations to take into account for inter-compilation learning.
- Further code structure details will be studied to analyze impact of code structure on this compilation approach.

Acknowledgements

This work was supported in part by the German Research Foundation (DFG) through the Priority Programme 1648 "Software for Exascale Computing" (SPPEXA) (GZ: LU 1353/11-1).