Automatic Instrumentation and PGO Optimization of HPC Compute Dwarfs

Anja Gerbes[1], Panagiotis Adamidis[2], Julian Kunkel[1]

[1]*Georg-August-Universität Göttingen* [2]*Deutsches Klimarechenzentrum (DKRZ)*

Introduction

Optimizing code for performance is essential in high-performance computing (HPC) to fully utilize modern processors. LLVM, an open-source compiler, facilitates Profile-Guided Optimization (PGO) and enables integration with profiling tools like LIKWID, which measures hardware performance metrics such as cache usage and CPU cycles. By combining LLVM's optimization capabilities with LIKWID's profiling, developers can automatically instrument source code to collect runtime data, allowing for optimizations that align with real-world application behavior. This approach enhances the performance of computationally intensive applications, though challenges remain in scaling it for large applications and reducing the overhead of automatic instrumentation.

Abstract

Profile-guided optimization (PGO) is an effective technique for enhancing program performance by utilizing runtime profiling data to guide optimizations. This poster explores the integration of PGO with the LLVM compiler infrastructure, focusing on the use of LLVM and Clang for source code instrumentation and optimization. Through source-level instrumentation, we capture detailed profiling information that is later used to optimize execution paths, enhance branch prediction, and refine function inlining decisions. The process begins with the compilation of the source code using Clang with appropriate instrumentation flags, which produce profiling data during program execution. This data is subsequently fed back into the LLVM compilation pipeline for further optimization during subsequent builds. We discuss the steps involved in this workflow, challenges encountered, and the impact of various optimizations on application performance. The study demonstrates the potential of LLVM's toolchain in efficiently applying profile-guided transformations, as well as the advantages of fine-tuned performance through data-driven decisions enabled by Clang's instrumentation capabilities.

Today's challenges in HPC

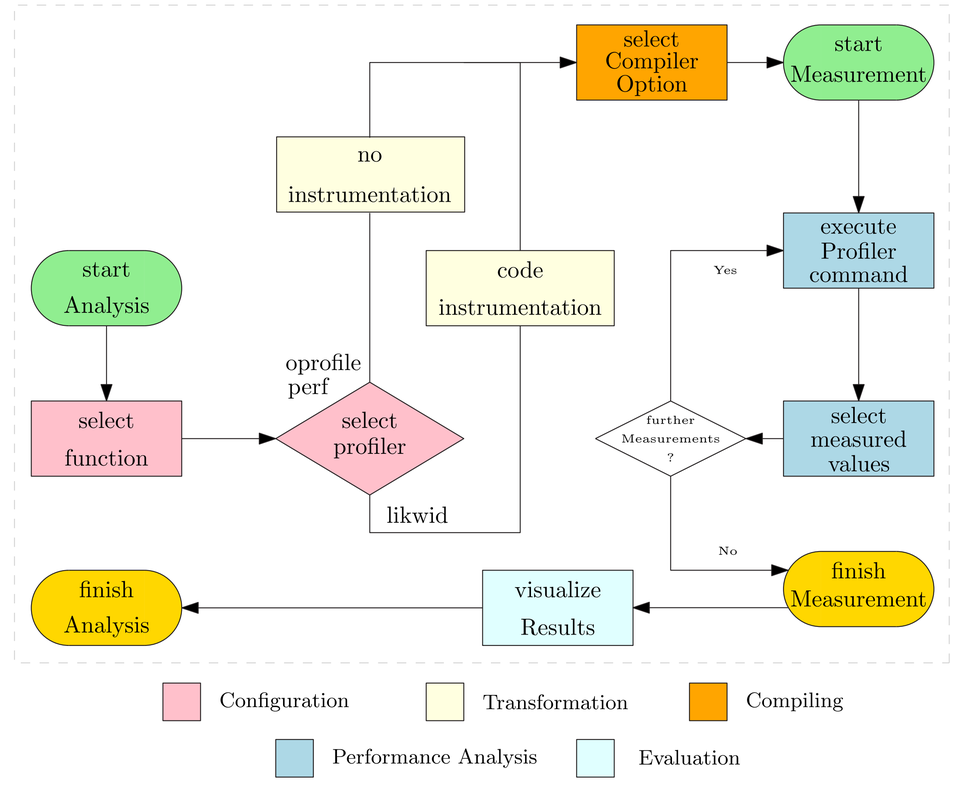
High-Performance Computing (HPC) is an integral part of today's science in order to solve increasingly complex mathematical calculations. The use of supercomputers has become much more widespread. Gone are these days of niche experimentation, as we are in the realm of utilizing HPC for immediate needs. Out from a niche to a useful and important tool to extract insight from big data. Challenges and today's requirements in modern HPC are the ever-increasing complexity of computer architectures towards the exascale computing era. Gordon Moore's law predicted that the number of transistors would double in less than two years and has already reached its fundamental physical limit.

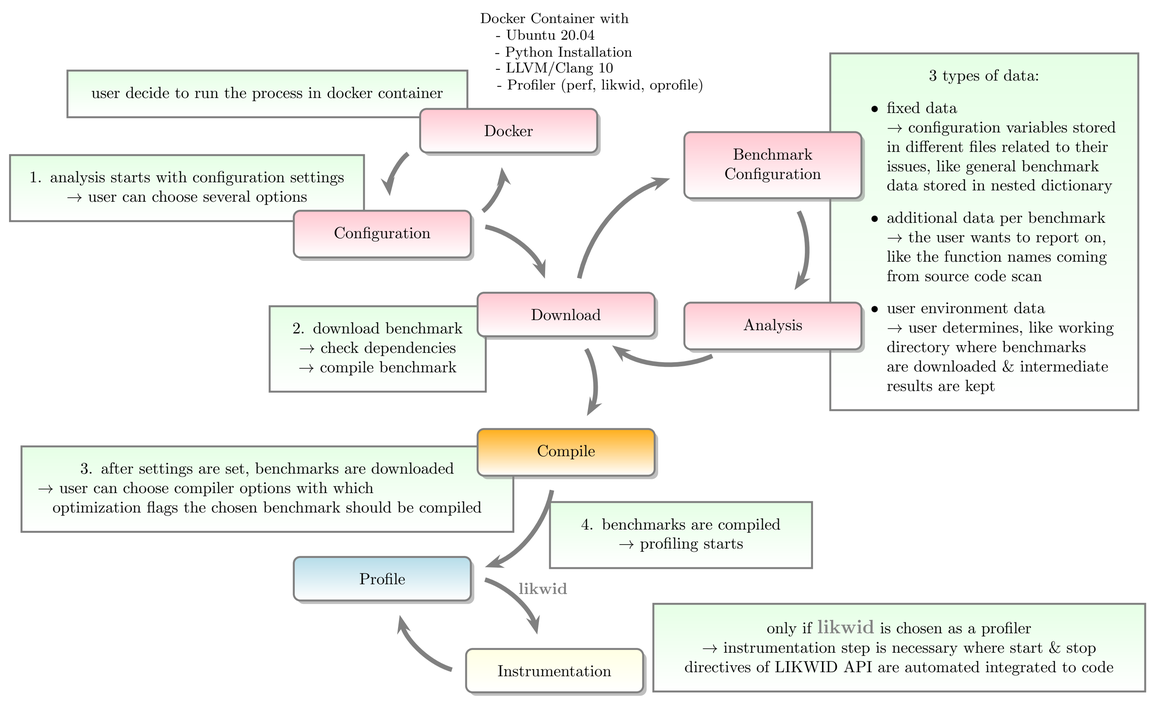
Motivation and goals

Studying the compiler's deficits in terms of performance for relevant HPC motifs in order to identify optimization potential is a major task to identify why compilers can not make necessary optimizations when compiling HPC applications. The key aspects are:

1. Study the compiler's deficits by utilizing HPC motifs
2. Theoretical analysis by utilizing CPU counters
3. Identification of optimization potential by manual code optimization
4. Automatisation of optimization process

PASCIT Workflow





Concept & Methodology

Automatic Source-Code Instrumentation via LLVM/Clang

The green boxes are the ones that should be automatically added to the source code.

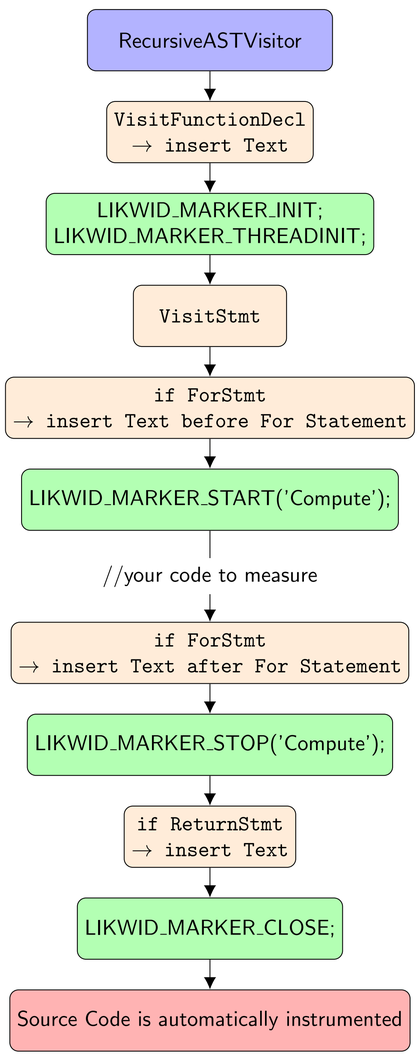


Fig. 1: Clang libraries workflow

Applying the automatic source code profiling

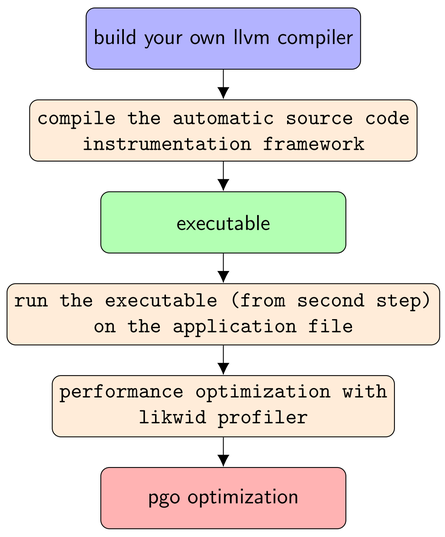


Fig. 2: Steps

Profile-Guided Optimization via LLVM/Clang

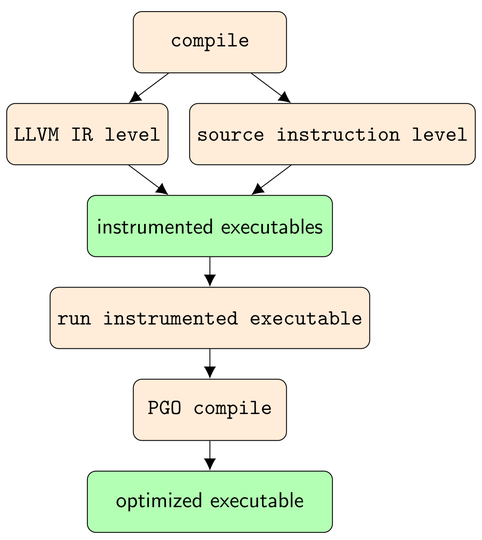


Fig. 3: PGO workflow

LLVM challenges

The LLVM compiler faces several challenges in its design and implementation, including:

- Complexity of Language Support
- Maintaining Portability
- Performance Tuning
- Optimizing for Modern Heterogeneous Architectures
- Optimization Across Multiple Computer Architectures

LLVM insights

Abstract Syntax Tree (AST) LLVM uses AST to analyze and optimize the source code.

Intermediate Representation (IR) LLVM uses IR to translate the source code into an intermediate representation that can be used by various optimizers and generators.

Compiler is used to read the source code and translate it to machine level instructions.

Optimizer LLVM uses a variety of optimizers, eg. constant factorization, loop shortening and register optimization, to optimize the source code.

Linker is used by LLVM to connect the objects and generate the binary code.

Profile-guided optimization (PGO) - compiler technique for improving the compilation process, where ones collects a profile, through instrumentation or sampling, then uses that information to guide the compilation process.

Instrumentation and PGO insights

Clang is a powerful and library-friendly C++-compiler, which makes a source-to-source instrumentation of C/C++ code possible. The goal is to set up the Clang libraries to parse some source code into an AST and then traverse the AST and modify the source code. The aim is to instrument the likwid profiler with the help of clang.

LibAstMatchers match interesting nodes of the AST and execute code that uses the matched nodes

RecursiveASTVisitor enables you to traverse the entire Clang AST and visits each node in a depth-first manner. (Figure 1)

- **TraverseDecl(Decl *x)** traverse the AST and going to each node
- **WalkUpFrom_ (_ *x))** will walk up the class hierarchy, starting from the node's dynamic type, until the top-most class (e.g. Stmt, Decl or Type) is reached.
- **Visit_ (_ *x)** will visit the node.
- _ can be statements (Stmt), declarations (Decl) or types (Type)

Rewriter manage the code rewriting task

- **InsertText()** insert the specified string at the specified location

ASTConsumer gives you fine grained control over the parsing, read ASTs and will call the RecursiveASTVisitor

LLVM IR-level instructions enriched by the profiling hooks are embedded into the program to track hotspots, execution paths, and other performance metrics during runtime. (Figure 3)