Decoding Performance: A Comparative Analysis of Open-Source Compilers

*CS323 2023F Research Report

1st Ruixiang JIANG
SUSTech
Shenzhen, China
12111611@mail.sustech.edu.cn

2nd Liquan WANG
SUSTech
Shenzhen, China
12011619@mail.sustech.edu.cn

3rd Wenhui TAO
SUSTech
Shenzhen, China
12111744@mail.sustech.edu.cn

***Abstract*—The landscape of open-source compilers is diverse and dynamic, with several prominent players contributing significantly to the field. This research delves into a comprehensive analysis and comparison of five prominent open-source compilers: GCC, Clang, LLVM, Codon, and Ark Compiler (also known as FangZhou). The study aims to elucidate the distinctions among these compilers, focusing on aspects such as architecture, optimization techniques, language support, and overall performance. Additionally, a crucial facet of this investigation involves an in-depth examination of the runtime speed differences exhibited by these compilers. By providing a detailed comparison, this research equips developers and enthusiasts with valuable insights to make informed decisions regarding compiler selection for diverse programming needs.**

***Index Terms*—Compiler, GCC, Clang, LLVM, Codon, Ark Compiler**

I. Introduction

Traditional compilers are typically divided into three main components: the frontend, optimizer, and backend. During the compilation process, the frontend is primarily responsible for lexical and syntactical analysis, transforming source code into an abstract syntax tree. The optimizer builds upon the frontend by enhancing the efficiency of the generated intermediate code through various optimization techniques. The backend then translates the optimized intermediate code into machine code tailored for specific platforms.

These three components collaborate to form the complete workflow of a compiler. The frontend understands the structure and syntax of the source code, creating an intermediate representation. The optimizer improves program performance and efficiency through a series of optimization techniques. Finally, the backend translates the optimized intermediate code into machine code relevant to the hardware platform, enabling the computer to execute the program.

As the demand for efficient and high-performance compilers continues to rise, the open-source community has witnessed the emergence and evolution of several notable compiler projects. In this research, we explore and compare five such compilers that have made significant contributions to the field: GCC, Clang, LLVM, Codon, and Ark Compiler.

GCC (GNU Compiler Collection), stands as one of the most venerable and widely-used open-source compilers, supporting an extensive range of programming languages and platforms. Its robust architecture and comprehensive feature set have solidified its position as a cornerstone in the development community.

LLVM (Low-Level Virtual Machine), serves as a standalone compiler infrastructure, providing a foundation for various language front ends. Its innovative design, featuring an intermediate representation (IR) and a wide range of optimization passes, has enabled LLVM to find applications beyond traditional compiler use cases.

Clang, renowned for its emphasis on modularity and user-friendly design, has gained prominence as a compiler front end, often coupled with LLVM as its backend. Its modular architecture and focus on static analysis have made it an attractive choice for developers seeking a versatile and efficient compilation tool.

Codon, as a relatively recent addition to the open-source compiler landscape, brings its own set of features and optimizations. Positioned as a compelling alternative, Codon aims to enhance compilation performance and efficiency, offering a fresh perspective in the realm of open-source compilation.

Ark Compiler, developed by Huawei, specifically targets ARM architectures, with a focus on optimizing performance in the mobile development space. Its unique optimizations and tailored approach make it a noteworthy contender in the context of mobile application compilation.

This research aims to unravel the architectural variances, optimization strategies, language support, and other distinctive features that set these compilers apart. Furthermore, a critical aspect of our investigation involves a meticulous comparison of the runtime speeds exhibited by each compiler. Through this comparative analysis, we seek to empower developers and the broader community with valuable insights, enabling them to make informed decisions when selecting a compiler tailored to their specific requirements.

II. DISTINCTIONOFGCC

The GNU Compiler Collection (GCC) has undergone a remarkable evolution, transforming from a modest C compiler to a versatile multi-language compiler capable of generating code for over 30 architectures. This extensive language and architecture support has propelled GCC to the forefront of compiler usage today. Serving as the default system compiler for every Linux distribution and gaining significant traction in academic circles for compiler research, GCC has earned its status as one of the most widely utilized compilers.

A. Brief Overview

In GCC, there are three main parts: front end, middle end and back end. Source code enters the front end, progressing through the pipeline, and at each stage, it undergoes transformations into progressively lower-level representations until the final stage of code generation, producing assembly code that is subsequently passed to the assembler.

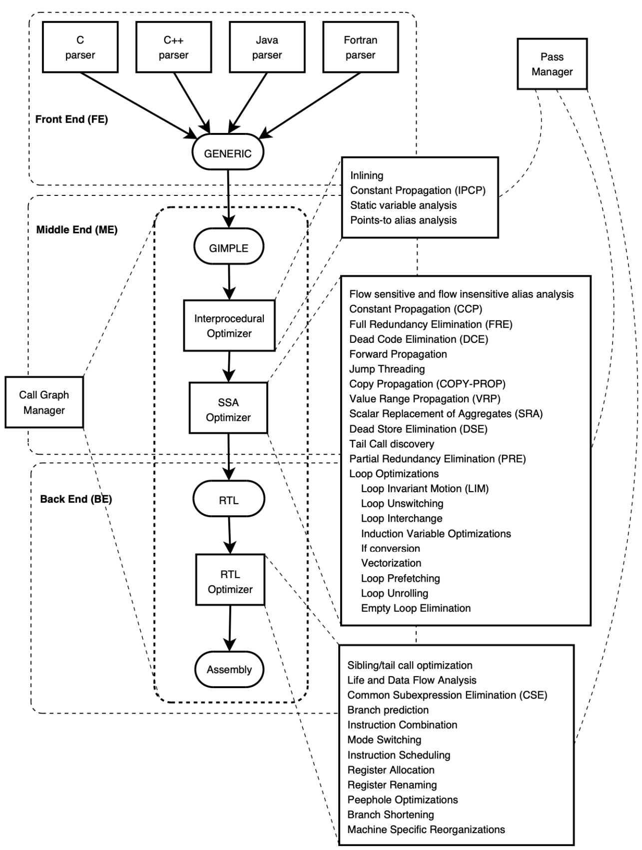

Fig. 1. An Overview of GCC [1]

Figure 1 shows a bird's eye view of the compiler. Notably, the various phases are orchestrated by the Call Graph and Pass managers. The call graph manager constructs a call graph for the compilation unit, determining the order in which each function should be processed. Additionally, it facilitates interprocedural optimizations (IPO), such as inlining. On the other hand, the pass manager oversees the sequencing of individual transformations and manages pre and post cleanup actions required by each pass.

The source code is organized in three major groups: core, runtime and support. In what follows all directory names are assumed to be relative to the root directory where GCC sources live. [1]

B. Optimization

Typically, optimizations provided by GCC can be divided into three degrees. Some optimizations make the assembly code shorter, while others speed up the code, which potentially is enlarged.

The O1 optimization level in GCC represents a moderate level of compiler optimization designed to enhance program performance while maintaining a relatively swift compilation process. The primary focus is on applying fundamental optimizations to the code. The O1 optimization level strikes a balance between improving program performance and minimizing compilation time, making it suitable for scenarios where moderate optimization is desired without significantly impacting build times. Here are some key aspects of O1 optimization:

- Unused Variable Removal: The compiler identifies and eliminates variables that are declared but not used in the program. This helps reduce the size of the generated code.
- Expression Simplification: O1 includes basic expression simplification, where the compiler aims to simplify complex expressions, potentially leading to more efficient code execution.
- Code Layout Optimization: The compiler may perform basic code layout optimizations, reorganizing code sections to improve locality and potentially enhance runtime performance.
- Inlining of Functions: O1 may include basic function inlining, where small functions are substituted directly into the calling code to reduce the overhead of function calls.
- Strength Reduction: Basic strength reduction techniques may be applied to replace expensive operations with cheaper equivalents, optimizing arithmetic expressions for improved performance.
- Control Flow Optimization: Basic control flow optimizations are employed to simplify and streamline conditional statements and loops, potentially reducing branch mispredictions.
- Minimization of Code Size: While not the primary focus, O1 aims to keep the generated code relatively compact, balancing performance improvements with code size considerations.

The O2 optimization level in GCC encompasses a set of advanced compiler optimizations aimed at substantially improving program performance. Building upon the optimizations introduced in O1, O2 introduces more sophisticated techniques. It is characterized by a more aggressive set of optimizations, making it suitable for scenarios where achieving higher performance is a priority, even at the cost of slightly

```
-fauto-inc-dec
-fbranch-count-reg
-fcombine-stack-adjustments
-fcompare-elim
-fcprop-registers
-fdce
-fdefer-pop
-fdelayed-branch
-fdse
-fforward-propagate
-fguess-branch-probability
-fif-conversion2
-fif-conversion
-finline-functions-called-once
-fipa-pure-const
-fipa-profile
-fipa-reference
-fmerge-constants
-fmove-loop-invariants
-freorder-blocks
-fshrink-wrap
-fshrink-wrap-separate
-fsplit-wide-types
-fssa-backprop
-fssa-phiopt
-ftree-bit-ccp
-ftree-ccp
-ftree-ch
-ftree-coalesce-vars
-ftree-copy-prop
-ftree-dce
-ftree-dominator-opts
-ftree-dse
-ftree-forwprop
-ftree-fre
-ftree-phiprop
-ftree-sink
-ftree-slsr
-ftree-sra
-ftree-pta
-ftree-ter
-funit-at-a-time
```

Fig. 2. O1 optimization flags [2]

```
-fthread-jumps
-falign-functions  -falign-jumps
-falign-loops  -falign-labels
-fcaller-saves
-fcrossjumping
-fcse-follow-jumps  -fcse-skip-blocks
-fdelete-null-pointer-checks
-fdevirtualize -fdevirtualize-speculatively
-fexpensive-optimizations
-fgcse  -fgcse-lm
-fhoist-adjacent-loads
-finline-small-functions
-findirect-inlining
-fipa-cp
-fipa-bit-cp
-fipa-vrp
-fipa-sra
-fipa-icf
-fisolate-erroneous-paths-dereference
-flra-remat
-foptimize-sibling-calls
-foptimize-strlen
-fpartial-inlining
-fpeephole2
-freorder-blocks-algorithm=stc
-freorder-blocks-and-partition -freorder-functions
-frerun-cse-after-loop
-fsched-interblock  -fsched-spec
-fschedule-insns  -fschedule-insns2
-fstore-merging
-fstrict-aliasing -fstrict-overflow
-ftree-builtin-call-dce
-ftree-switch-conversion -ftree-tail-merge
-fcode-hoisting
-ftree-pre
-ftree-vrp
-fipa-ra
```

Fig. 3. O2 optimization flags [2]

longer compilation times. Below is a detailed description, combining the objectives and impacts:

- Loop Unrolling: Replicating loop bodies to reduce loop control overhead and enhance instruction-level parallelism, thereby improving execution speed.
- Data Flow Analysis: Analyzing the flow of data through the program facilitates a better understanding of variable relationships, leading to more effective optimizations.
- Cross-Module Inlining: Extending function inlining to functions defined in separate compilation units enhances opportunities for inlining across different parts of the program.
- Strength Reduction: Replacing expensive operations with cheaper equivalents optimizes arithmetic expressions for improved efficiency.
- Loop Fusion: Combining adjacent loops reduces loop overhead, improving cache locality and reducing loop control overhead.
- Loop Distribution: Distributing loop iterations enables better parallelization, improving the potential for parallel execution of loop iterations.
- Vectorization: Converting scalar operations into vector operations leverages SIMD instructions, enhancing parallelism, especially on architectures with SIMD support.

The O3 optimization level in GCC represents the highest degree of compiler optimization, aimed at maximizing program performance, even if it results in longer compilation times. Building upon the optimizations introduced in O2, O3 incorporates more sophisticated and time-consuming techniques.

While the -O3 optimization level is capable of generating high-performance code, it's important to note that the resulting increase in the size of the executable can potentially have detrimental effects on its speed. Specifically, if the size of the executable surpasses the capacity of the available instruction cache, this could lead to significant performance penalties. Consequently, it might be more prudent to opt for compiling at the -O2 optimization level. This decision is driven by the intention to enhance the likelihood that the executable fits within the constraints of the instruction cache, thereby mitigating the risk of severe performance degradation.

To illustrate the impact of GCC compiler optimization levels, we'll use a C code example that performs numerical computations. We'll use a simple numerical integration algorithm as our case study. Below is the code without any optimizations applied: [3]

```c
// integrtion.c
#include <stdio.h>
#include <math.h>

double integrate(double (*func)(double), double a, double b, int n) {
    double h = (b - a) / n;
    double result = 0.0;

    for (int i = 0; i < n; i++) {
        double x = a + i * h;
        result += func(x) * h;
    }

    return result;
}

int main() {
    double result = integrate(sin, 0, M_PI, 1000000);
    printf("Result: %lf\n", result);
    return 0;
}
```

Fig. 4. C Code Example [3]

Running on the same computer, the statistics are shown in Table 1. The "Real" time is the actual wall-clock time it took to execute the program. The "User" time represents the CPU time consumed by the program. The "Sys" time indicates system-

TABLE I
TABLE EXAMPLE

p1cm¡ p1cm¡ p1cm¡

Title
1
2
2
2heightlineA
3
4
2
2lineB
5
6
2
2

related CPU time. [3]

REFERENCES

[1] Novillo, D. (2006, September). GCC an architectural overview, current status, and future directions. In Proceedings of the linux symposium (Vol. 2, p. 185).

[2] GNU Compiler Collection. (2017). Optimize Options. GNU Compiler Collection Documentation. Retrieved from https://gcc.gnu.org/onlinedocs/gcc-7.2.0/gcc/Optimize-Options.html

[3] Bharatiya, P. (2023, November 1). Maximizing C++ Program Performance: A Comprehensive Guide to GCC Compiler Optimization. Data Intelligence. Retrieved from https://data-intelligence.hashnode.dev/maximizing-c-program-performance-a-comprehensive-guide-to-gcc-compiler-optimization