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.

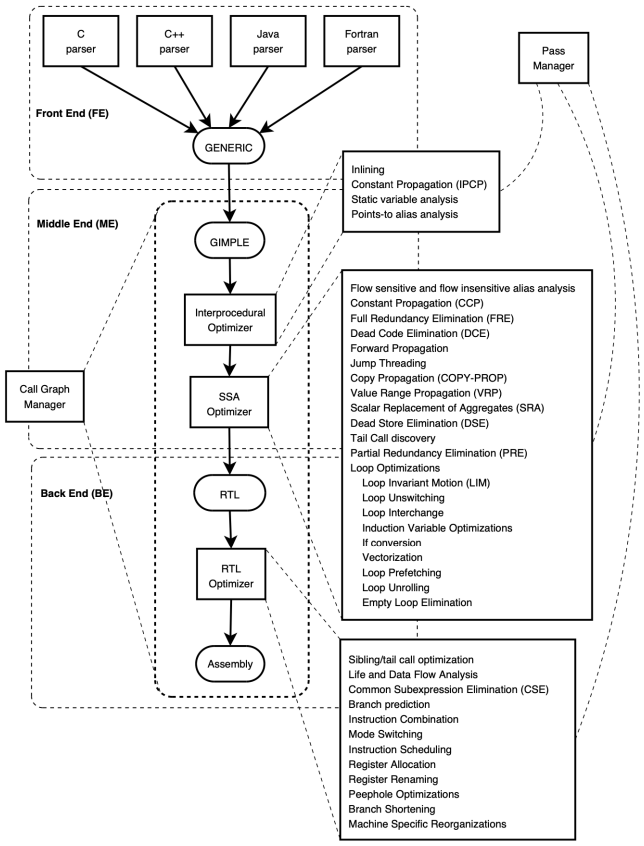

Fig. 1. An Overview of GCC [1]

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 Level

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.

The Os optimization level focuses on minimizing the size of the generated executable. It aims to reduce the overall footprint of the compiled program, making it suitable for environments where compactness is a priority, such as embedded systems with limited storage.

The -Ofast optimization level is similar to -O3 but allows for more aggressive optimizations, including those that may affect mathematical precision. It prioritizes maximizing execution speed and might not be suitable for applications where strict adherence to floating-point precision is required.

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:

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-related CPU time. [3]

Specifying the target architecture also can yield meaningful benefits. The -march option of gcc allows the CPU type to be specified. The default architecture is i386. GCC runs on

```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 that Performs Numerical Computations [3]

TABLE I
RUNNING TIME

Optimization	default	O2	O3	Os	Ofast
Real	0.064s	0.041s	0.058s	0.016s	0.067s
User	0.064s	0.041s	0.058s	0.015s	0.066s
Sys	0.001s	0.001s	0.001s	0.001s	0.001s

all other i386/x86 architectures, but it can result in degraded performance on more recent processors. Let's now look at an example of how performance can be improved by focusing on the actual target. Build a simple test application that performs a bubble sort over 10,000 elements. The elements in the array have been reversed to force the worst-case scenario. [4]

```
[mtj@camus]$ gcc -o sort sort.c -O2
[mtj@camus]$ time ./sort

real    0m1.036s
user    0m1.030s
sys     0m0.000s
[mtj@camus]$ gcc -o sort sort.c -O2 -march=pentium2
[mtj@camus]$ time ./sort

real    0m0.799s
user    0m0.790s
sys     0m0.010s
```

Fig. 5. Effects of Architecture Specification on a Simple Application [4]

By specifying the architecture, in this case a 633MHz Celeron, the compiler can generate instructions for the particular target as well as enable other optimizations available only to that target. As shown in Figure 5, by specifying the architecture we see a time benefit of 237ms (23% improvement). Although it shows an improvement in speed, the drawback is that the image is slightly larger. Using the size command, we can identify the sizes of the various sections of the image. [4]

```
[mtj@camus]$ gcc -o sort sort.c -O2
[mtj@camus]$ size sort
   text    data     bss     dec     hex filename
    842     252       4    1098     44a sort
[mtj@camus]$ gcc -o sort sort.c -O2 -march=pentium2
[mtj@camus]$ size sort
   text    data     bss     dec     hex filename
    870     252       4    1126     466 sort
```

Fig. 6. Size Change of the Application [4]

Here the instruction size (text section) of the image increased by 28 bytes. But in this example, it's a small price to pay for the speed benefit. [4]

In conclusion, optimizing your code with GCC is not just a luxury but a necessity in today's demanding software landscape. A profound understanding of the diverse optimization levels and their judicious application can elevate your code to the status of an efficient, high-performing masterpiece.

However, a word of caution is essential. Optimization is akin to a double-edged sword. While it has the potential to deliver remarkable performance gains, it should not come at the expense of the readability and maintainability of your code. Striking the delicate balance between optimization and code quality is an art that every developer must master.

C. Vectorization

Vectorization in GCC refers to the process of transforming scalar operations within code into vector operations, a technique particularly pertinent to SIMD (Single Instruction, Multiple Data) architectures where a single instruction can concurrently operate on multiple data elements.

The objective of vectorization is to enhance parallelism and harness the capabilities of contemporary processors equipped with vector units. These vector units facilitate the simultaneous execution of a single instruction on multiple data elements, thereby amplifying throughput and overall performance.

In Figure 7, the compiler on the left is capable of computing only one pair of scalar multiplications at a time. Consequently, the multiplication of four pairs of numbers requires four separate operations. Conversely, the compiler on the right possesses the capability to concurrently process four numbers, enabling the completion of the multiplication of four pairs of numbers in a single operation.

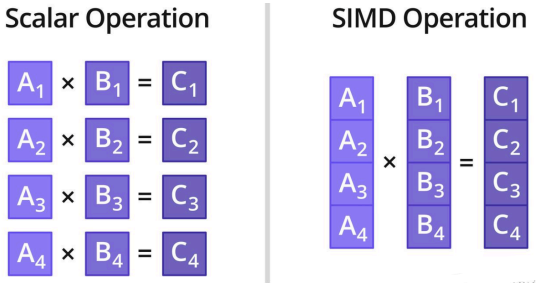

Fig. 7. A comparison between scalar and SIMD operation [5]

In modern computing, SIMD processing units or GPUs are primarily employed for vector processing. High-end CPUs commonly integrate specialized instruction sets for SIMD operations, such as Intel's SSE and AVX. The SIMD processing capability of CPUs operates in a parallel fashion within a single core. The parallel paradigm in multicore processing is referred to as MIMD (multiple instruction multiple data). The distinction lies in SIMD's execution in a lockstep manner, where all ALU units share a single program counter (PC). In contrast, MIMD involves independent execution across cores, each with its own PC.

In GCC, using vector instructions through built-in functions is available. On some targets, the instruction set contains

SIMD vector instructions which operate on multiple values contained in one large register at the same time. [6]

Assuming our SIMD processor can handle 4 double numbers at once, take an addition of two arrays as an example. Inside the loop in line 14, it loads four single-precision floating-point values from the array a into a Neon vector va. Then in adds the loaded vector va element-wise to the cumulative sum vector $vsum$. It accumulates the sum as the loop progresses. After the loop, it uses Neon intrinsics to perform pairwise addition on the lower and higher lanes of the cumulative sum vector $vsum$, resulting in a 2-element vector sum_lane. Finally it completes the reduction by adding the two elements of the 2-element vector sum_lane. The result is a single-precision floating-point value representing the sum of the vectorized elements.

```
01. // Serial Sum
02. double sum_serial(double a[], int size) {
03.     double sum = 0;
04.     for (int i = 0; i < size; i += 1) {
05.         sum += a[i];
06.     }
07.     return sum;
08. }
09.
10. // Vectorized Sum using ARM Neon Intrinsics
11. double sum_vectorized(double a[], int size) {
12.     // Ensure that 'a' is 128-bit aligned
13.     float32x4_t vsum = vdupq_n_f32(0.0f);
14.     for (int i = 0; i < size; i += 4) {
15.         // Load 4 elements into a vector with correct alignment
16.         float32x4_t va = vld1q_f32((float32_t*)&a[i]);
17.         vsum = vaddq_f32(vsum, va);
18.     }
19.
20.     // Reduce vector lanes to get the sum
21.     float32x2_t sum_lane = vpadd_f32(vget_low_f32(vsum), vget_high_f32(vsum));
22.     float32_t sum = vget_lane_f32(vpadd_f32(sum_lane, sum_lane), 0);
23.     return sum;
24. }
```

Fig. 8. SIMD Code Example

When the array size is 100000000, the serial time and vectorized time is 0.3656 seconds and 0.1772 seconds (Apple M2), respectively, which indicates a crucial speedup.

In the context of parallelized numerical accumulation, it is essential to recognize that the order of summation may change due to parallel processing. To achieve parallelism, associativity, the property that allows rearrangement of operands without altering the result (e.g., $a + b + c + d = (a + b) + (c + d)$), becomes crucial. This property ensures that parallel addition remains consistent even with a different summation order.

However, when dealing with floating-point operations involving multiplication or addition, achieving strict associativity can be challenging. Floating-point arithmetic, subject to rounding errors, may not strictly adhere to associativity, especially when parallelized. Consequently, to facilitate parallel floating-point operations, specialized compiler options such as '–ffast-math' in GCC are often employed. These options relax strict adherence to IEEE standard compliance and allow for more aggressive optimizations.

Maintaining numerical stability in the face of reordering operations for parallelism is a fundamental concern. Rounding errors, inherent in floating-point calculations, can accumulate differently depending on the order of operations. Achieving both parallelism and numerical stability requires a delicate balance. The use of appropriate compiler flags, careful algorithm design, and consideration of numerical stability principles are essential in minimizing the impact of rounding errors and ensuring reliable parallel computations.

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

[4] Jones, M. T. (2005). Optimization in GCC. Linux journal, 2005(131), 11.

[5] Qingyang CHEN. (2023, November 9). Retrieved from https://zhuanlan.zhihu.com/p/337756824

[6] GNU Compiler Collection. Vector Extensions. GNU Compiler Collection Documentation. Retrieved from https://gcc.gnu.org/onlinedocs/gcc/Vector-Extensions.html