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 several prominent open-source compilers: GCC, Clang/LLVM. 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**

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 several such compilers that have made significant contributions to the field: GCC, Clang and LLVM.

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.

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.

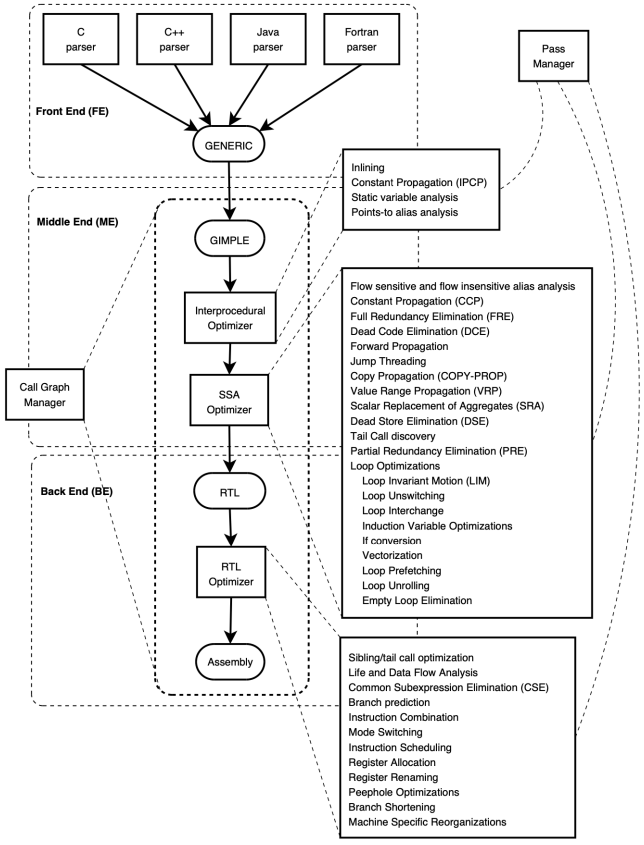

Fig. 1. An Overview of GCC [1]

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 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

```
-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]

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:

```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]

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]

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

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 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.

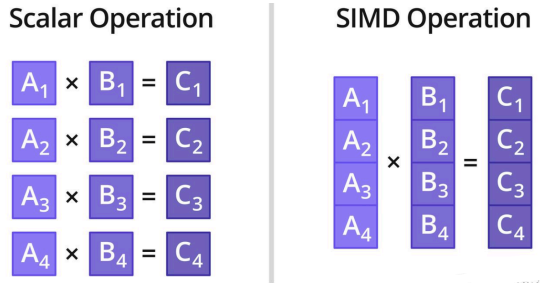

Fig. 7. A comparison between scalar and SIMD operation [5]

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.

```
// Serial Sum
double sum_serial(double a[], int size) {
    double sum = 0;
    for (int i = 0; i < size; i += 1) {
        sum += a[i];
    }
    return sum;
}

// Vectorized Sum using ARM Neon Intrinsics
double sum_vectorized(double a[], int size) {
    // Ensure that 'a' is 128-bit aligned
    float32x4_t vsum = vdupq_n_f32(0.0f);
    for (int i = 0; i < size; i += 4) {
        // Load 4 elements into a vector with correct alignment
        float32x4_t va = vld1q_f32((float32_t*)&a[i]);
        vsum = vaddq_f32(vsum, va);
    }

    // Reduce vector lanes to get the sum
    float32x2_t sum_lane = vpadd_f32(vget_low_f32(vsum), vget_high_f32(vsum));
    float32_t sum = vget_lane_f32(vpadd_f32(sum_lane, sum_lane), 0);
    return sum;
}
```

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.

Now take one more complex example. That is matrix multiplication, which is widely used, forming the foundational logic for numerous algorithms. This paper will compare the time consumption of different approaches in matrix operations, with a focus on exploring the efficacy of O3 optimization and elucidating the methodologies potentially employed by O3 optimization.

In Matrix.h we define a matrix, simultaneously, in Matrix.c we define how it works. Suppose we have two matrices mtx1 and mtx2, and then set ans equal to $mtx1 \times mtx2$. Just follow the principle of matrix multiplication we can write a plain method. Optimizing cache utilization is critical for enhancing computational efficiency. One effective strategy involves manipulating the loop order to improve cache hit rates, specifically by employing a technique known as loop interchange. By employing loop interchange, the order of traversal is rearranged, typically from ijk to ikj.

```
for (int i = 0; i < ans->n; i++)
        for (int j = 0; j < ans->m; j++)
            for (int k = 0; k < mtx1->m; k++)
                ans->num[i * ans->m + j] += mtx1->num[i * mtx1->m + k] * mtx2->num[k * mtx2->m + j];
```

Fig. 9. Matrix Multiplication Plain Method

When computing each element of the matrix $C = A \times B$ using the Plain Method, a vector dot product is performed between the rows of matrix A and the columns of matrix B. As matrices are stored row-wise, accessing individual elements of the corresponding row in matrix A results in contiguous memory access. However, for matrix B, which requires column-wise computation, accessing current and subsequent positions in memory is non-contiguous. Upon completion of the computation, matrix B incurs n jumps in memory access, resulting in n^3 jumps after n^2 computations.

By altering the loop order from ijk to ikj, the program performs row-wise computations for the elements of C. While computing the first element in a row, access is made to the first element in the corresponding row of A and the first element in the corresponding column of B. When calculating the subsequent element, the B pointer increments by one position while the A pointer remains stationary. After completing the calculation for all elements in that row of C, the A pointer moves one position to the right. Consequently, each computation of an element in C incurs only one jump, resulting in a total of n^2 jumps after n^2 computations.

Notably, when transitioning to the next row after accessing the current row, there is no jump if using a one-dimensional array for storage, whereas one jump occurs if using a two-dimensional array.

The relationship between the number of jumps and loop order, as well as storage format, is summarized in Table II. Furthermore, the inner loop can be vectorized, utilizing hardware-provided fused multiply-add (FMA) operations, with subsequent unrolling. If compiler optimizations are enabled, manual unrolling becomes unnecessary.

Additionally, SIMD optimization can be employed. During each computation, a 4×4 small block is calculated by extracting 4 rows from matrix A and 4 columns from matrix B. Each row and column are reused 4 times in this process, with all data reuse occurring in the cache, effectively reducing the number of memory accesses. To ensure contiguous reading for each row, the corresponding data is packed into 4 one-dimensional arrays after computing 4 rows, and these arrays are repacked for the next 4 rows after completing all terms in the product related to these data.

TABLE II
NUMBER OF JUMPS

Order	One-dimensional array	Two-dimensional array
ikj	n^2	$2n^2 + n$
kij	$2n^2$	$3n^2$
jik	$n^3 + n^2 + n$	$n^3 + 2n^2$
ijk	$n^3 + n^2 - n$	$n^3 + n^2 + n$
kji	$2n^3$	$2n^3 + n$
jki	$2n^3 + n^2$	$2n^3 + n^2$

TABLE III
TIME COSTED BY DIFFERENT METHOD

N	Plain	Loop interchange	O3	SIMD
16	0ms	0ms	0ms	0ms
128	3ms	1ms	0ms	0ms
256	1802ms	555ms	627ms	95ms
2000	16049ms	4347ms	4164ms	507ms

If the matrices are sufficiently large, and 4 rows or columns cannot fit into the cache, matrix A can be divided into row vectors, and matrix B into column vectors. Finally, SSE instructions are utilized to accelerate the multiplication operation. If AVX2.0 is employed, processing 4 double-precision floating-point numbers simultaneously is possible, thereby doubling the efficiency.

The time costed by different method is shown in Table III (AMD R5-5500U). In the 'MatrixMultiplication' directory, execute 'Test.cpp' to obtain the corresponding output. Note that the runtime performance during execution may be influenced by factors such as CPU architecture and capabilities, operating system, available system memory, and other related considerations.

From Table III, it can be observed that the runtime after O3 optimization and after swapping loop order are relatively close, and both are significantly higher than the runtime after SIMD optimization. This suggests that O3 optimization may alter the loop order and does not inherently provide SIMD optimization.

In conclusion, GCC provides robust support for automatic vectorization, enabling the compiler to detect and transform segments of code suitable for vectorization at compile time. This automatic vectorization is performed without explicit directives from the programmer, relying on the compiler's analysis of data dependencies and loop structures to identify vectorizable portions. In fields dealing with large datasets and scientific computing, vectorization plays a crucial role within the GCC compiler.

D. Back-end Optimization Analysis

Register Transfer Language (RTL) is a crucial intermediate representation in the context of compiler construction, serving as a pivotal bridge between high-level source code and low-level machine code within the framework of GCC. The RTL generation, optimization, and assembly code generation processes constitute an intricate series of stages aimed at transforming abstract program structures into efficient machine-executable instructions.

The RTL generation process involves the translation of the abstract syntax tree (AST) derived from the high-level source code into a hardware-oriented, register-transfer-based representation. This step captures the essential data and control flow aspects of the program, representing them in terms of RTL operations and their associated operands. This intermediate form serves as a foundation for subsequent analyses and transformations.

RTL optimization encompasses a suite of techniques geared toward enhancing the performance of the generated code. This involves a myriad of low-level optimizations such as register allocation, instruction scheduling, and dead code elimination. Register allocation seeks to judiciously assign program variables to processor registers, minimizing memory accesses and improving overall execution speed. Instruction scheduling aims to reorder operations to exploit parallelism and reduce pipeline stalls. Additionally, dead code elimination identifies and removes code segments that contribute no discernible impact on program output.

Subsequent to RTL optimization, the compiler proceeds to the generation of assembly code tailored to the target architecture. This involves mapping RTL operations and operands to specific machine instructions, considering the intricacies of the underlying hardware. The assembly code generated at this stage serves as the final output that can be further processed by an assembler to produce machine code executable on the target platform.

The processes of RTL generation, optimization and assembly code generation, should be guided by machine description. The machine descriptions are composed of three files: target machine-description macro definitions file (target.h), machine-description functions support file (target.c) and target machine descriptions file (target.md). [7]

These descriptions play a pivotal role in the backend stages of the compiler, particularly during the code generation phase. Compiler utilization of machine descriptions ensures the generation of assembly code congruent with the target machine architecture. These descriptions not only furnish the compiler with the requisite information for the production of correct code but also endow the optimizer with insights, facilitating the adept utilization of the target machine's features to enhance code performance.

It deserves to be mentioned that the efficiency of complier should be taken into full account in the porting process. As the pattern of each instruction is matched in order, the more instruction patterns there are, the more slowly the compiling system will be. It is better to combine the similar instructions to one instruction pattern to obtain higher complier efficiency. The priority of instruction matching should be paid attention to at the same time. Take a simple C language sentence 'i++' as an example and set the target machine as Minisys2. The result of matching can be 'INC $2', as well as the mode of 'add $2, $3, 1'. The process of matching 'target.md' file is carried out through the order from top to bottom in GCC, so

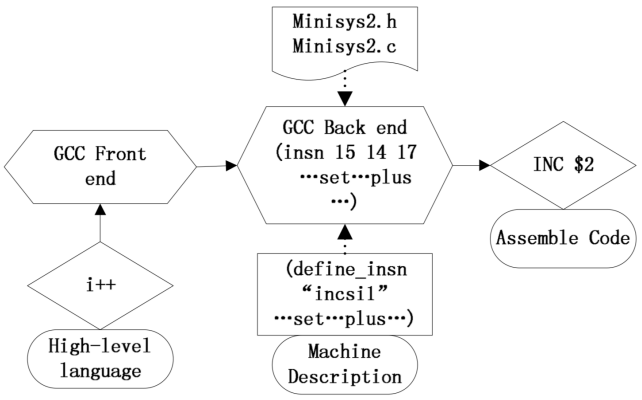

Fig. 10. Example Code 'i++' Compilation Process [7]

'INC' instruction pattern should be put in the front of 'add' instruction pattern in order to test the newly increased INC instruction. [7]

In summation, machine descriptions serve as crucial metadata in the compiler domain, delineating how the compiler interfaces with a specific target architecture, thereby enabling the production of accurate and efficient machine code.

E. Summary of GCC

GCC plays a pivotal role in the entire process of translating source code into target machine executable code. Its impact extends across critical facets of software development.

The primary function of GCC lies in the transformation of source code, authored in high-level programming languages such as C and C++, into executable code for the target machine. It accommodates various target architectures, enabling developers to create and execute programs on diverse platforms.

As an open-source and widely adopted compiler, GCC has exerted a profound influence on the entire software ecosystem. It affords developers the capability for cross-platform development, fostering software portability. Furthermore, the existence of GCC has propelled advancements in compiler technology, serving as a template for numerous other compiler projects.

GCC incorporates a potent optimizer capable of enhancing the performance of the generated target code during the compilation process. Optimization spans various dimensions, encompassing, but not limited to, constant folding, register allocation, instruction scheduling, and loop optimization. Through these optimization techniques, GCC generates machine code that is not only more efficient but also aligns closely with the underlying hardware capabilities.

III. Distinction of Clang/LLVM

LLVM (Low Level Virtual Machine) is a compiler infrastructure project designed to provide a flexible and extensible compiler framework. It comprises a suite of general-purpose compiler tools, including frontends responsible for source code parsing, optimizers tasked with enhancing intermediate code, and backends responsible for generating target code. The design of LLVM emphasizes modularity and reusability, making it a preferred framework for compiler implementations and optimization tools across various programming languages.

Clang is an open-source, cross-platform compiler for the C, C++, Objective-C, and Objective-C++ programming languages. It is part of the LLVM project and is designed to offer a high-performance, modular, and extensible compiler frontend. The design of Clang emphasizes code clarity and readability, making it a preferred compiler for many developers and projects.

A. Introduction

Figure 11 illustrates the basic architecture of Clang/LLVM. Initially, the frontend for LLVM was GCC. Subsequently, Apple aspired to develop its own Clang to replace GCC. However, currently, GCC with Dragon Egg can still generate LLVM IR, serving as a viable alternative to Clang. Moreover, it is possible to develop custom frontends, which, when combined with the LLVM backend, enable the creation of compilers for custom programming languages.

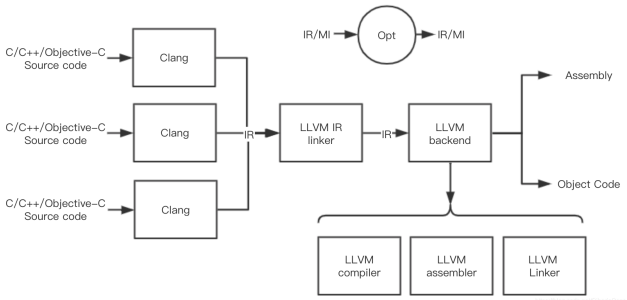

Fig. 11. Architecture of Clang/LLVM [8]

In LLVM, Intermediate Representation (IR) exists in three forms.

The first is a human-readable IR, akin to assembly code, albeit positioned between high-level languages and assembly. This representation is designed for human consumption, and its disk file suffix is .ll.

The second form is an unreadable binary IR, known as bitcode, with a disk file suffix of .bc.

The third representation is a memory format, exclusively stored in memory, devoid of any file format or suffix. This format contributes to LLVM's swift compilation, distinct from GCC, which generates intermediate process files at the conclusion of each stage. LLVM's intermediate data at various compilation stages is in the form of this third representation of IR.

In LLVM IR, a compilation unit (i.e., a .c file) represents a Module. Within a Module, there are Global Values, primarily comprising Global Variables and Functions. A Function encompasses Basic Blocks, and each Basic Block contains instructions, such as "add". Therefore, the hierarchical relationship can be conceptualized as Module - Function - Basic Block - Instructions. Some APIs of LLVM IR might be referred like Figure 12.

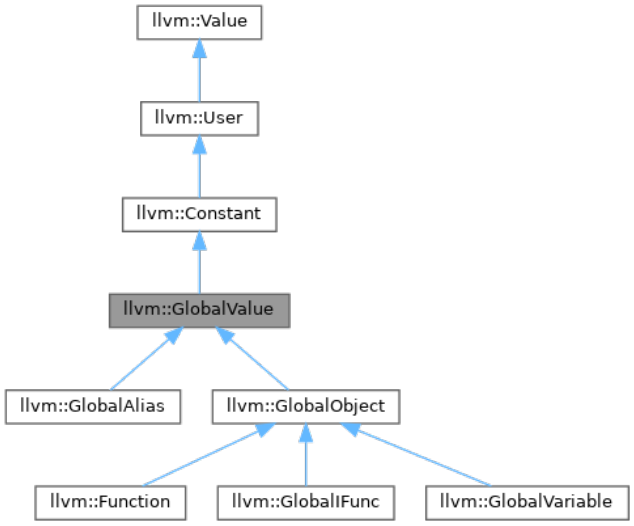

Fig. 12. Inheritance Diagram for llvm::GlobalValue [10]

All three representations are entirely equivalent. While Clang/LLVM tools do not generate these files by default (typically unnecessary for non-compiler developers), they can be specified using parameters in the toolchain. Conversion between the first two file types can be facilitated using llvm-as and llvm-dis.

Notably, there is an LLVM IR linker in the process, distinct from the linker in GCC. To achieve link-time optimization, LLVM, after generating IR for individual code units in the frontend (Clang), links the entire project's IR while concurrently performing link-time optimizations.

The LLVM backend constitutes the genuine backend of LLVM, also referred to as the LLVM core. It encompasses compilation, assembly, and linking processes, ultimately producing assembly files or target code. It is important to differentiate the LLVM compiler here from the compiler in GCC; in LLVM, the LLVM compiler exclusively compiles LLVM IR.

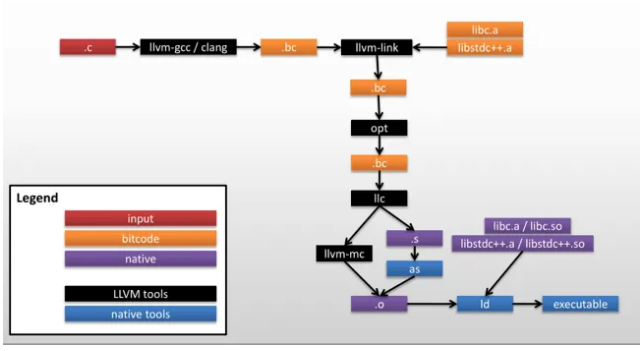

Fig. 13. Procedure of Clang/LLVM [9]

Figure 13 illustrates the procedure a C code will go through in Clang/LLVM. Firstly, we have C source code program files, which then undergo the Clang frontend (or the deprecated llvm-gcc frontend, used prior to the advent of Clang, capable of generating LLVM IR). Subsequently, LLVM IR is produced. In this context, LLVM IR is identified by the .bc extension, representing a serialized data format utilized for storage on disk. However, LLVM IR also boasts another remarkable feature in the form of a human-readable format, typically denoted by the .ll extension. Figure 14 is an example of the human-readable format for a simple C language Hello World program:

```
01.  extern int printf(const char *, ...);
02.
03.  int main()
04.  {
05.    printf("Hello, World!");
06.  }
```

Fig. 14. Hello-world Human-readable Format

Use 'clang a.c -S -emit-llvm' to generate .ll file, which is shown in Figure 15. It's useful to obtain information of more instructions referring to LLVM Language Reference Manual [11].

```
; ModuleID = 'a.c'
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.11.0"

@.str = private unnamed_addr constant [14 x i8] c"Hello, World!\00", align 1

; Function Attrs: nounwind ssp uwtable
define i32 @main() #0 {
  %1 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([14 x i8], [14 x i8]* @.str, i32 0, i32 0))
  ret i32 0
}

declare i32 @printf(i8*, ...) #1

attributes #0 = { nounwind ssp uwtable "disable-tail-calls"="false" "less-precise-fpmad"="false" "no-frame-p
attributes #1 = { "disable-tail-calls"="false" "less-precise-fpmad"="false" "no-frame-pointer-elim"="true" "

!llvm.module.flags = !{!0}
!llvm.ident = !{!1}

!0 = !{i32 1, !"PIC Level", i32 2}
!1 = !{!"Apple LLVM version 7.3.0 (clang-703.0.31)"}
```

Fig. 15. Content of .ll File [9]

After the completion of the generation of the .bc file, if optimizations such as O2 are applied, there is an additional optimization phase known as unit-level optimization. This phase encompasses approximately 40 optimization passes, including dead code elimination, inlining, and others. Despite the optimizations, the output remains in the .bc file format. Subsequently, the process proceeds to the llvm-link stage. During the llvm-link stage, the primary objective is to merge multiple .bc files into a single .bc file while simultaneously performing link-time optimizations, which will be explained later.

After the llvm-link stage, we obtain a new .bc file, prompting another round of optimization. This is necessary because llvm-link might introduce structural changes to the IR, potentially making it more amenable to optimization. Subsequent to this step, the final optimized .bc file is obtained, marking the formal entry into the code generation phase.

In Figure 13, only llc, representing the path for machine code generation, is illustrated. However, LLVM also supports an additional process, namely lli, which involves interpreting the LLVM IR. Although this paper won't delve into this here, the focus remains on machine code generation. The role of llc is to compile LLVM IR into an assembly file (.s). Roughly,

LLVM IR can be treated as a programming language, and after passing through the llc compiler, an assembly file is produced. Analogous to the generation of LLVM IR from a C program through Clang, the generation of an assembly file is achieved.

Following the creation of the assembly file, the system assembler, such as GNU's as, is invoked to generate the object file (.o). Attentive readers might notice another path involving llvm-mc, a direct route to generating the object file. However, this path is not universally followed on all platforms, and the key lies in LLVM's integrated assembler, denoted by '-fintegrated-as'. Using the integrated assembler, one can go from MCLowering to MCInst and then employ MCStreamers to generate the .o directly, or alternatively, generate the .s file. [9]

Once the .o file has been generated, the remaining process is relatively straightforward. It involves the linker linking relevant libraries and combining them with the object file to produce the executable file with extensions like .out or .exe.

In summary, the Clang/LLVM workflow initiates from source code, undergoes multiple stages of transformation, optimization, and linking, ultimately yielding an executable file. The modular and extensible nature of this workflow positions it as a robust tool widely utilized in compiler and toolchain development.

B. Clang Driver

The driver of a compiler is a program that controls and coordinates the entire compilation process. It receives compilation options and source code files provided by the user, and is responsible for invoking various components of the compiler, such as the preprocessor, compiler proper, assembler, and linker, to generate the final executable or object file. The compiler's driver plays an organizational and managerial role during the compilation process, ensuring that each compilation phase is invoked in the correct order with the appropriate parameters.

In practical engineering, the term 'driver' pertains to concrete issues rather than theoretical aspects of compilation. Questions such as how to set compiler options, what interfaces to provide to users, and how to utilize these interfaces are considerations in the development of compiler products, falling outside the typical scope covered in compiler theory textbooks.

However, the 'driver' constitutes a crucial component in actual compilers, serving as a direct interface for programmer-users and significantly influencing user experience. For instance, when utilizing the GCC compiler and configuring numerous compiler options, if there is a need to port it to other platforms or support additional compilers, it often requires adjustments to compiler options rather than modifications to your code. For example, using '-std=c++11' to enable C++11 features, while on IBM's compiler on AIX, the corresponding option is '-qlanglvl=extended0x', and Microsoft's MSVC on Windows does not recognize the '-std=c++11 option.

Figure 16 shows how Clang driver works. The orange portion in this diagram not only symbolizes the mentioned input/output in the top-left corner of the image but also represents the specific data structures within the Clang Driver, such as the ArgList as you observed. The green section not only stands for Driver Functions but also signifies the concrete execution flow within the Driver. The blue-purple segment encompasses some significant auxiliary classes.

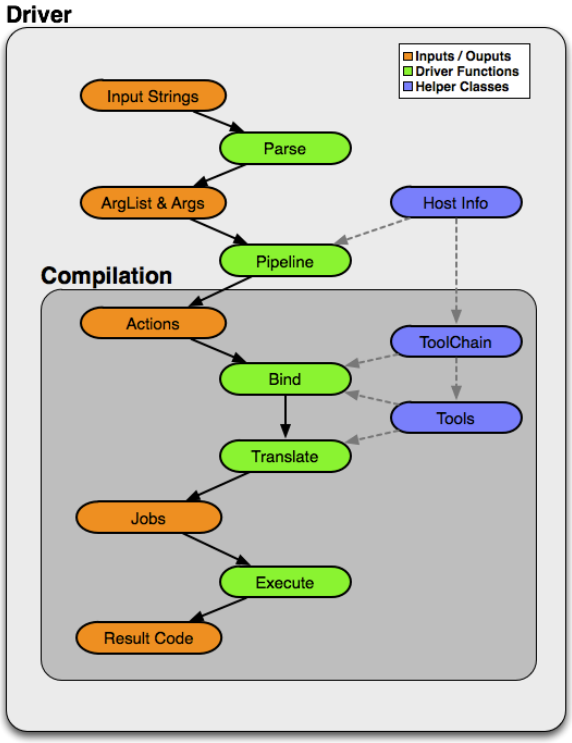

Fig. 16. Clang Driver [12]

The role of Parse is option parsing. It is responsible for parsing the command-line options provided by the user into individual parameters and placing them into instances of the Arg class. Suppose we have a file named 'a.c' containing only one line, which is 'int main(){}'. In terminal, as an example, use 'clang -### a.c -I/CS323/2023F' to compile it. The result is shown in Figure 17.

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

[7] Xiaowei, W., Kuixing, W., Quansheng, Y. (2012). Research and Development of Compiler Based on GCC. Recent Advances in Computer Science and Information Engineering: Volume 3, 809-814.
[8] P2Tree. (2019). https://blog.csdn.net/SiberiaBear/article/details/103111028
[9] Blue. (2016). https://zhuanlan.zhihu.com/p/21889573
[10] LLVM. (2023). GlobalValue Class Reference. Retrieved from https://llvm.org/doxygen/classllvm_1_1GlobalValue.html
[11] LLVM. (2023). LLVM Language Reference. Retrieved from https://llvm.org/docs/LangRef.html
[12] Blue. (2017). https://zhuanlan.zhihu.com/p/22974869