Performance Comparison between Clang and GCC on RISC-V and ARM Platform

Liu Leqi
12011327@mail.sustech.edu.cn
Southern University of Science and Technology
Shenzhen, China

Jin Yang
12012801@mail.sustech.edu.cn
Southern University of Science and Technology
Shenzhen, China

Huang Zitong
12012710@mail.sustech.edu.cn
Southern University of Science and Technology
Shenzhen, China

ABSTRACT

Computing is now constantly evolving shaped by technology, applications, and markets trends. Mobile, IoT, and cloud computing have been major drivers of innovations recently and are likely to remain so in the next decade. The end of semiconductor scaling and demise of Moore's and Dennard's laws indicate that future performance improvements will have to come from architectural improvements and new computing structures rather than from getting smaller and faster transistors. Emerging new applications in domains such as machine learning, and block chain present a new set of challenges requiring more performance and increased energy-efficiency of modern processors. Market trends continue to shrink time-to-market.

The modern compilers are all of complex architectures, and they contain the procedure from lexical analysis to intermediate code generation, and finally generating the efficient machine code. The faster compilers are crucial to achieve higher productivity of codes. In this research report, we conduct investigations on many perspectives around Clang, mainly by comparison with other popular compilers such as GCC.

KEYWORDS

Investigation, Clang, Performance Comparison, Compilers

1 INTRODUCTION

Modern compliers are extremely complex softwares that translate programs written in high-level languages into binaries that can be executed on the underlying hardware. The translation includes multiple tasks, including preprocessing, lexical analysis, parsing, semantic analysis, code optimization, and finally creating executable files.

Clang is a popular open-source compiler that is primarily associated with the C language family (C, C++, Objective-C, OpenCL, CUDA and RenderScript). It is now maintained by the LLVM project, which stands for Low-Level Virtual Machine. Clang offers several notable characteristics and advantages compared to other compilers. Clang's primary goal is to provide a high-quality compilation toolchain with a focus on speed, efficiency, and user-friendly diagnostics. Clang is known for its fast compilation times and efficient resource utilization. It employs advanced optimization techniques to generate optimized binaries with improved performance. This makes it particularly appealing for large codebases and projects where compilation speed is crucial.[1]

GCC, which stands for GNU Compiler Collection, is a widely used open-source compiler suite that supports multiple programming languages. It is developed and maintained by the GNU Project and has a long history dating back to the early 1980s. GCC is renowned for its robustness, portability, and wide range of supported platforms and architectures. GCC includes front ends for C, C++, Objective-C, Fortran, Ada, Go, and D, as well as libraries for these languages (libstdc++,...). GCC was originally written as the compiler for the GNU operating system. [2]

Both Clang and GCC have a long history and have been widely adopted by a variety of organizations and individuals. Since their initial release, Clang and GCC has undergone numerous updates and improvements, and it has grown to support a wide range of languages and platforms. The competitions between them are also intensive and popular, which in turn promotes the development of these two compilers.

2 METHODOLOGY

The research method used here [3] is an experiment utilizing two benchmarks compiled with multiple compilers and optimization options on two processors. Making an experiment by measuring performance by execution time is a reliable way to make a quantivative evaluation of performance as long as you are aware of the traits and impact of the independent variables. Benchmarks are used to measure performance on hardware and by compiling the benchmarks with different compilers and optimization options, you will be able to see how the compilation affects the performance.

To run this experiment, you need to feed the compiler with the source code from one of the benchmarks for compilation and linking into an executable, which can be uploaded to a platform and executed to obtain a benchmark score. This process will be repeated with different optimization options in order to measure the impact and compare it between different compilers and platforms. In the

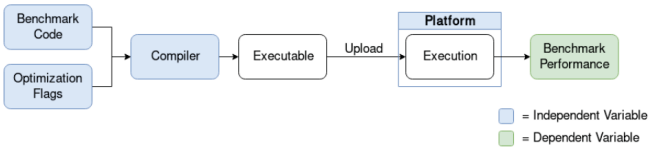

Figure 1: Overview of research methodology

experiment, the researchers have executed 2 different benchmarks on 2 different CPU architectures with multiple compilers using multiple optimization settings.

2.1 Platform and Processor

For RISC-V platform, the researchers chose the Freedom E310 processor on HiFivel board, because it is currently the only easily

available RISC-V chip.
For reference platform, the researchers chose the Freescale ARM Cortex-M4 processor on the Teensy 3.6 board, because it is mature, well supported and has good compiler support.

2.2 Benchmarks

The benchmarks the researchers chose were the popular Dhrystone benchmark in addition to CoreMark. Dhrystone is a synthetic computing benchmark developed in 1984 intended to be representative of processors integer performance. CoreMark is a synthetic computing benchmark developed in 2009 intended to become the successor to Dhrystone and therefore the new industry standard. The reason why chose Dhrystone is because it is a very popular embedded processor benchmark, which makes it easy to compare the results with lots of other processors as well as being multiple scientific analyses on it.

3 EXPERIMENTS

3.1 Experiment 1: Dhrystone

The researchers run Dhrystone on both ARM and RISC-V in addition to run both GCC and LLVM/clang on RISC-V and only GCC on ARM. They also compile the benchmark with the -O3, -O2 and -O1 optimization options focused on execution performance in addition to -Os for code size optimization and -O0 for no optimization. The experiment on Dhrystone is illustrated in Figure 2.

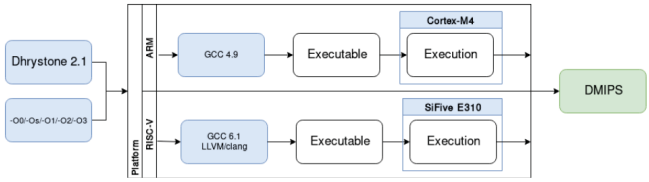

Figure 2: Overview of methodology for Dhrystone experiment

3.2 Experiment 2: CoreMark

The researchers run CoreMark on both ARM and RISC-V in addition to run both GCC and LLVM/clang on RISC-V and only GCC on ARM. The compiler optimization options tested will be the standard -O3, -O2 and -O1 focused on performance, -Os focused on code size and -O0 for no optimization. In this experiment, they also execute all of their tests with a set of extra optimization flags both on and off. The experiment on Dhrystone is illustrated in Figure 3.

4 RESULT

4.1 Dhrystone

From Figure 4 we can see that the Dhrystone performance was overall rather decent on both RISC-V and ARM with GCC. The -O0 and -Os performance on GCC seems to be very low for an unknown reason, but -O1 performance is pretty good. The LLVM/clang on the other hand seems to be broken. All optimization levels above -O0 only performs slightly faster than -O0. And comparing with

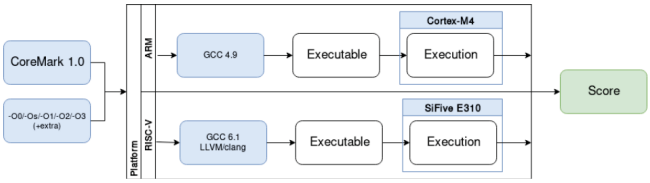

Figure 3: Overview of methodology for CoreMark experiment

GCC on RISC-V, the optimizations of LLVM/clang barely seem to make a difference. The -O0 level on LLVM/clang is also a third of the performance of -O0 on GCC which in turn is a third of the performance of -O0 on ARM which certainly points to some significant issue in its code generation.

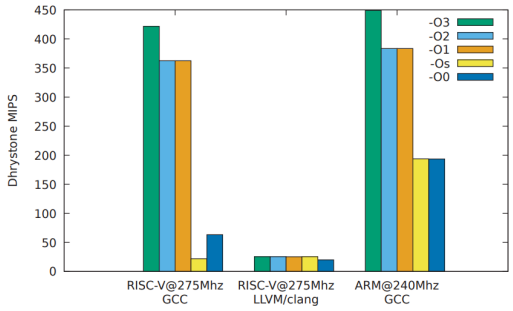

Figure 4: Dhrystone Performance

4.2 CoreMark

From Figure 5 we can see that the performance is rather even between RISC-V and ARM in the CoreMark benchmark. The GCC performance seems to be very good and those extra optimization flags gives a very good performance increase especially in conjunction with -O2.
However, LLVM/clang falied to compile CoreMark for RISC-V.

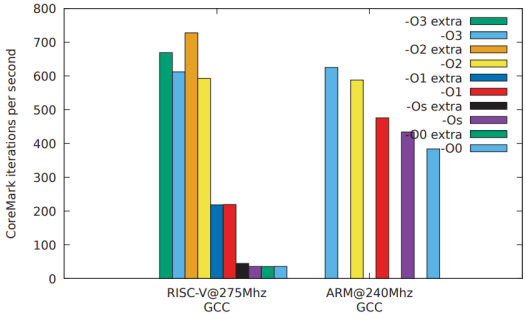

Figure 5: CoreMark Performance

Performance Comparison between Clang and GCC on RISC-V and ARM Platform

5 CONCLUSION

It does perform rather well with GCC on the higher optimization levels, but there seems to be some issues on lower optimization levels, where GCC was significantly slower. On the other hand, LLVM/clang falied to compile CoreMark for RISC-V and the optimization flags on Dhrystone did not make any performance impact at all.

Another finding is that, both GCC and LLVM/clang have RISC-V specific performance issues. GCC performs much worse in lower optimization levels while LLVM/clang seems to be fundamentally broken by not leveraging any performance increase at all, and it even crashed when trying to compile the CoreMark benchmark.

There is a lot of work to be done on LLVM/clang compiler on RISC-V to become competitive with the performance and the stability of GCC.

6 REFERENCES

[1] Clang C language family frontend for LLVM. (n.d.). Llvm.org. Retrieved December 24, 2023, from https://clang.llvm.org/index.html

[2] GCC, the GNU compiler collection. (n.d.). Gnu.org. Retrieved December 24, 2023, from https://gcc.gnu.org/

[3] RISC-V Compiler Performance:A Comparison between GCC and LLVM/clang. (2017).