Characteristics of GNU Compiler Collection (GCC) and Comparison with Other Compilers

Mingqian Liao
Xin Li
12012919@mail.sustech.edu.cn
12012138@mail.sustech.edu.cn
Southern University of Science and Technology
Shenzhen, China

ABSTRACT

The modern compilers are all of complex architecture, and it contains the procedure from lexical analysis to intermediate code generation, finally generate the efficient machine code. And faster compilers are crucial to achieve higher productivity of software developers.

And the GNU Compiler Collection(GCC) is a collection of compiler front ends to a common back-end compilation engine, we choose GCC to research.

In this research report, we conduct research on many perspectives around the GCC, mainly by conduct comparison with other popular compilers such as LLVM and IPS.

Our research shows that the GCC is good at registering allocation and jumping optimization. However, GCC did not perform well in loop unrolling and inlining arithmetic function. What is more, compared with the LLVM and IPS, GCC has least build time on average. Also, the performance of GCC is good in higher optimization levels, however, it will be significantly slower on lower optimization levels when running on RISC-V. In addition, GCC has a longer bug lifetime compared to LLVM.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

KEYWORDS

Research, GCC, Performance Comparison, LLVM

ACM Reference Format:
Mingqian Liao and Xin Li. 2023. Characteristics of GNU Compiler Collection (GCC) and Comparison with Other Compilers. In *Proceedings of (SUSTech CS323 Report).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

SUSTech CS323 Report, Characteristics of GNU Compiler Collection (GCC) and Comparison with Other Compilers, January, 2023

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Modern compliers are extremely complex software that translate programs written in high-level languages into binaries that execute on the underlying hardware. The translations include a multitude of tasks, including preprocessing, lexical analysis, parsing, semantic analysis, and code optimization, and finally creating executable files. Compilers are not only expected to produce small executables that achieve the best possible performance, but also to do so in the least amount of time. Large software projects might involve multiple subsystems, code written in multiple programming languages, may incorporate matured libraries, and may span millions of lines of code [3]. Faster compilers are crucial to achieve higher productivity of software developers.

The selection of a compiler depends on parameters such as accessibility, support for hardware, support for different programming languages, efficiency of compiler, and backward compatibility. One of the most widely used modern open source compilers for software development is GCC.

GCC, the GNU Compiler Collection, formerly known as the GNU Compiler Suite, and also known as GNU CC and the GNU C Compiler, is a collection of compiler front ends to a common back-end compilation engine. It was developed by the Free Software Foundation as part of the GNU project, with the goal of providing a free, high-quality compiler system for the GNU operating system. The list of compilers includes C, C++, Objective C, Fortran, and Java.

GCC is an optimizing and cross-platform compiler. It is known for its high level of optimization, which can produce fast and efficient code. It includes general optimizations that can be applied to any language or target CPU, as well as options specific to particular CPU families and even specific models within a family of CPUs. GCC has been successfully ported to a wide range of hardware platforms and supports platform and target submodels to generate executable code that can run on all members of a given platform [7] .

GCC has a long history and has been widely adopted by a variety of organizations and individuals. It was first developed in the 1980s as part of the GNU Project, a collaborative effort to create a free and open source operating system. Since its initial release, GCC has undergone numerous updates and improvements, and it has grown to support a wide range of languages and platforms. Today, GCC is considered one of the most widely-used compilers in the world, and it is an essential tool for many developers and organizations.

The goal of this research report is to evaluate effectiveness of GCC and compare its performance with other open source compilers.

2 LITERATURE REVIEW

Several studies have been conducted on the performance and optimization capabilities of GCC. One study [6] compared the performance of GCC with LLVM in perspective of the code size and the dynamic instruction count for the the EISC Processor. The analysis of the results demonstrated that LLVM excelled at optimizing calculations, while GCC performed better at managing register allocation and optimizing jumps. Overall, GCC performed better in the majority of EEMBC benchmarks, with an average improvement of 18% in terms of dynamic instructions. Additionally, the compiled code produced by GCC was, on average, smaller in size compared to that of LLVM. Another research [2] on GCC evaluated the performance of the GCC and LLVM/clang compilers support for the RISC-V target and their ability to optimize for the architecture. The performance was evaluated from executing CoreMark and Dhrystone which are both popular industry standard benchmarks for evaluating performance on embedded processors. The experiment included running the benchmarks at different optimization levels and comparing the performance with ARM architecture. The results presented that the -O2 and -O3 optimization levels on GCC for RISC-V performed very well in comparison to their ARM Architecture Test Platform. On lower optimization levels, the performance GCC failed to meet the expectations compared to ARM but was much better than clang/LLVM. Aldea et al. compares the sequential and parallel code generated by IPS, GNU, and Sun compiler for SPEC CPU 2006v1.1 benchmarks [1]. Lattner and Adve designed LLVM compilation framework. They compared the performance of LLVM-based C/C++ compiler and GNU on SPEC 2000 [4, 5], showing that LLVM performed better than GNU. Hebbar et al. evaluated Intel Parallel Studio XE-19 (IPS), the LLVM Compiler Infrastructure project, and the GNU Compiler Collection using the SPEC CPU2017 benchmark suite. The results showed that LLVM created the smallest executables, GNU had the lowest build times, and IPS had the best performance [3]. Halbiniak et al. assessed the impact of various state-of-the-art C/C++ compilers available for novel AMD EPYC Rome and Milan multi-core processors on the performance of real-world scientific codes corresponding to the solidification modeling application using the PF method and generalized finite difference scheme for solving governing PDEs. Among the studied compilers are AOCC, Clang, GCC, Intel compiler, and PGI. The results showed that in the case of the static intensity, GCC lagged noticeably behind other compilers and for dynamic intensity, GCC also had a poor performance.

3 PERSPECTIVE

3.1 Performance between GCC and LLVM on RISC-V and ARM platform

3.1.1 Experiment Methodology. The research method [6] used here will be an experiment utilizing two benchmarks compiled with multiple compilers and optimization options on two processors. If we are aware of the traits and impact of the independent variables, it is a reliable way to make a experiment by measuring performance by execution time.

In the experiment, firstly the researchers feed the compiler with the source code from one of the benchmarks for compilation and linking into a executable which they can upload to a platform and execute to obtain a benchmark score. Then the process will be repeated with different optimization options, then they can measure the impact and compare it between our different compilers and platforms. The process can be better understood through Figure 1

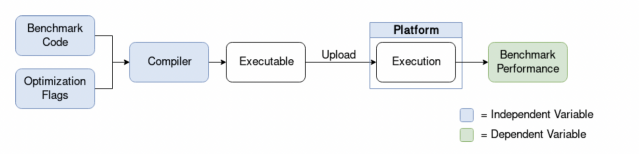

Figure 1: Overview of research methodology

In our first research experiment, the researchers have executed 2 different benchmarks on 2 different CPU architectures with multiple compilers using multiple optimization settings. And this can produce a lot of possible combinations, and we will explore it further.

For the RISC-V platform, the researchers used the Freedom E310 processor on the HiFive1 board because it is the only easily available RISC-V chip, and for the reference platform they used the Freescale ARM Cortex-M4 processor on the Teensy 3.6 board, the reason why they used the ARM architecture is that the architecture is being mature and has good compiler support.

For the benchmarks, they chose 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[19] is a synthetic computing benchmark developed in 2009 intended to become the successor to Dhrystone and therefore the new industry standard.

In the first experiment, the Dhrystone will be run on both ARM and RISC-V in addition to be run both GCC and LLVM/clang on RISC-V and only GCC on ARM, and 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 optimizations. The experiment on Dhrystone is illustrated clearly in Figure 2

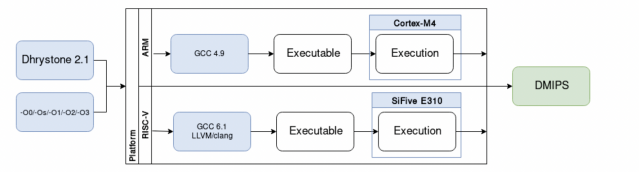

Figure 2: Overview of research methodology on Dhrystone experiment

And the Dhrystone was executed 100 000 000 iterations, which could be more than enough to get a good average of the performance per second. The result was also convert to DMIPS/Mhz when comparing RISC-V with ARM.

In the second experiment, the researchers ran the CoreMark on both ARM and RISC-V in addition to run both GCC and LLVM/clang on RISC-V and only GCC on ARM. And the compilers optimization option is the same as the first experiment. In constrast to the first experiment, they also executed all of the tests with a set extra optimization flags both on and off. They set the amount of CoreMark iterations to 400 000 iterations. The experiment on CoreMark is illustrated clearly in Figure 3

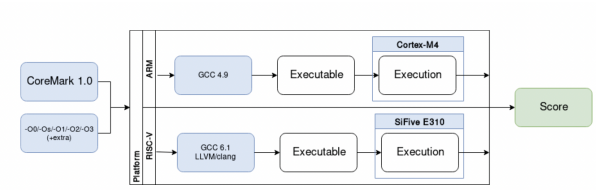

Figure 3: Overview of research methodology on CoreMark experiment

And the result was also converted to score/Mhz when comparing RISC-V with ARM.

3.1.2 Experiment Result. The result of the first experiment conducted on Dhrystone is shown as Figure 4

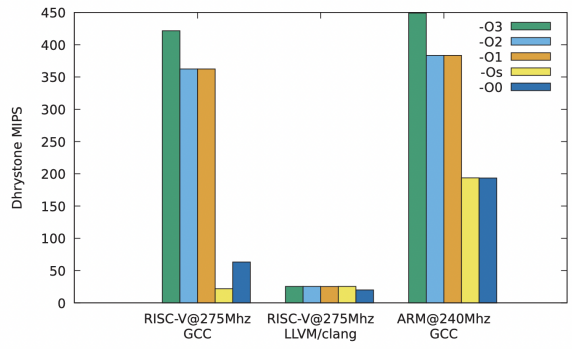

Figure 4: Dhrystone Performance

From figure 4, we can see that the Dhrystone performance was overall rather decent on both RISC-V and ARM with GCC. The performance of those optimization levels above -O1 is significant better than the -O0 and -Os. What interests me is that in the result, the -Os is significantly slower than -O0.

From the result, we can also see that, for the LLVM/clang, On all optimization levels above -O0 only performs slightly faster than -O0 on GCC. The -O0 level on LLVM is approximately a third of the performance of -O0 on GCC, which in turn is a third of the performance of -O0 on ARM. As for the experiment on ARM on GCC, what confused me is that, the performance between -O1 and -O2 was insignificant in Dhrystone, but -O3 gave a larger performance boost on ARM than on RISC-V.

Overall, for the experiment in Dhrystone, we can see that the performance was 6% faster on our ARM processor than on our RISC-V processor, what made RISC-V so competitive in total performance in figure 4 was due to its higher clock speed.

The result of the first experiment conducted on CoreMark is shown as Figure 5

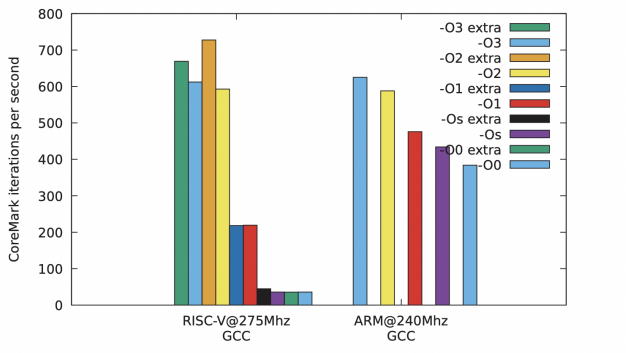

Figure 5: CoreMark Performance

In figure 5 we can see that, the performance is rather even between RISC-V and ARM in the CoreMark benchmark. And the GCC performance is overall better with the extra optimization flags, especially for -O2. But there is little improvement for -O0, -O1. Also, we can see that the -O2 extra has the fastest execution time.

The performance with GCC on RISC-V compared to ARM is great and in fact almost equal on -O2 and -O3 without the extra optimization flags. The lower -O1, -O0 and -Os optimization levels is good on ARM but definitely lacking with GCC on RISC-V though.

3.1.3 Experiment Conclusion. Combining this two experiments, we find that, for the Dhrystone benchmark, GCC on RISC-V performed very well on the higher optimization levels, but was significantly slower on lower optimizaion levels, while for the LLVM, none of the the optimization levels barely made a performance improvement over -O0, and the performance was very slow, especially considering that -O0 seems to be significantly slower on than both GCC och ARM and on RISC-V.

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 our CoreMark benchmark, to sum up, there is a lot of work to be done to gain competitive with GCC in performance and the stability.

3.2 Compiler Bugs Comparison with GCC and LLVM

Compilers are critical, widely-used complex software. Bugs in them have significant impact, and can cause serious damage when they silently miscompile a safety-critical application. We are also interested in that field, so we conduct a research in it, in a research[7] ,

the researchers conduct the first empirical study on the characteristics of the bugs in two main- stream compilers, GCC and LLVM.

3.2.1 Experiment Methodology. The study focuses on fixed bugs. We say a bug is fixed if its resolution field is set to fixed and the status field is set to resolved, verified or closed in the bug repositories of GCC and LLVM. Then they identify the revisions that correspond to these fixed bugs. The researchers collect the entire revision log from the code repositories, and for each revision, and they scan its commit message to check whether this revision is a fix to a bug.

3.2.2 Experiment Statics. Figure 6 shows the overall evolution of GCC and LLVM's bug repositories. In particular, it shows the number of bugs reported, rejected or fixed each month.

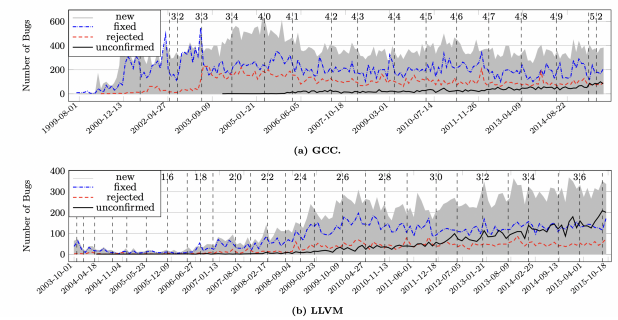

Figure 6: The overall evolution history of the bug repositories (in months)

The trends of the plots for GCC are relatively stable in these years compared to those of LLVM. After gaining its popularity recently, LLVM has drawn much more attention than before and more bug reports are being submitted monthly.

Table 1 shows the information on the bugs which are resolved as invalid, worksforme or wontfix. An invalid bug report is one in which the associated test program is invalid, or the reported "anomalous" behavior is in fact deliberate. If a bug described in a report cannot be reproduced, this report is labeled as worksforme. A bug report is resolved as wontfix if the affected version, component or library of the compiler is not maintained although the reported bug can be confirmed.

	invalid	workforme	wontfix
GCC	7,072/10.4%	1,151/1.7%	1,463/2.2%
LLVM	1,639/6.7%	717/2.9%	593/2.4%

Table 1: The number and the percentage of bug reports revolved as invalid, worksforme or wontfix

Figure 7 shows the information on duplicate bug reports in the GCC and LLVM bug repositories. 9.9% of GCC bugs and 5.3% of LLVM bugs have duplicates.

Apart from this, the research also analyses the distribution of bugs in components and that in files. The components touch all critical parts of compilers: front end (e.g. syntactic and semantic parsing), middle end (e.g. optimizations) and back end (e.g. code generation). And find that firstly, C++ is the most buggy component in GCC, accounting for around 22% of the bugs. We think that it is because C++ has many more features than the other programming languages, supporting multiple paradigms (i.e. procedural, object-oriented, and generic programming). The second find is that half of the source files only contain one bug (60% of GCC and 53% of LLVM), and quite few files have a large number of bugs.

(a) Duplicate bugs of GCC.

#Duplicate	0	1	2	3	4	5	≥6
#Report	35,933	2,924	596	215	98	41	83

(b) Duplicate bugs of LLVM.

#Duplicate	0	1	2	3	4	5	≥6
#Report	12,157	570	72	17	13	5	8

Figure 7: Distribution of duplicate bug reports

3.2.3 Research Conclusion. From the research, we can conclude that most of the bug fixes touch a single source file with small modifications (43 lines for GCC and 38 for LLVM on average). What's more, the average lifetime of GCC bugs is 200 days, and the average lifetime of LLVM bugs is 111 days. and for the GCC, high priority tends to be assigned to optimizer bugs, most notably 30% of the bugs in GCC's inter-procedural analysis component are labeled as the highest priority level. And for both compilers, the bug revealing test cases are typically small, with 80% having fewer than 45 lines of code.

This research deepens our understanding of compiler bugs. What we can learn from the research is that, the study sheds light on interesting characteristics of compiler bugs, at the meantime, highlights challenges and opportunities to more effectively test and debug compilers.

3.3 Performance Comparison among GCC, IPS and LLVM using SPEC CPU2017 benchmark

3.3.1 Research Basics. We also concentrate on the overall evaluation among GCC, IPS, and LLVM. The IPS is the Intel Parallel Studio XE-19 compiler. The study[2] is based on measurement, which evaluates three most prevalent compilers used in industry and academia. The researchers compare the effectiveness of IPS, the LLVM, and the GCC using the SPEC CPU2017 benchmark suite. Standardized Performance Evaluation Corporation (SPEC) is one of the most successful efforts in standardizing benchmark suites. SPEC CPU benchmarks are a great tool to stress compiler. And the SPEC CPU2017 is the most recent incarnation of the SPEC CPU. In the research, we can quantitatively evaluate the compilers with respect to metrics such as benchmark build times, executable code sizes, and execution times. SPEC CPU2017 contains 43 benchmarks, organized into four groups, namely: (a) SPECspeed2017 floating-point (fp_speed), (b) SPECspeed2017 integer (int_speed), (c) SPECrate2017 floating-point (fp_rate), and (d) SPECrate2017 integer (int_rate), and in the study, the researchers use those benchmark to test the performance of the compilers.

3.3.2 Research Result. For the size of the executable code, when test on the fp_speed benchmark, overall the IPS executables are $\tilde{1}$.35x larger than the LLVM executables, whereas the GNU executables are $\tilde{2}$.7x larger than the LLVM executables. GNU being a cross-platform compiler focuses on portability, hence the code generated by GNU is significantly larger. And the other benchmarks are similar, the LLVM creates the smallest executables.

When considering the build time, though LLVM has smaller executable size, it has significantly longer build times in comparison with GNU and IPS. This is especially true for floating-point benchmarks. Though the GNU compilers produce executables that are significantly larger in size, the build times are shorter than the build times of LLVM.

For performance, they run the benchmark with 1, 2, 4, 6 threads, respectively. The figure 8, 9, 10 11 shows the results of the experiment.

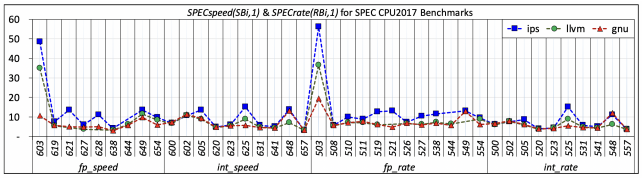

Figure 8: SPEC RATIO FOR SINGLE THREAD/COPY (HIGHER IS BETTER)

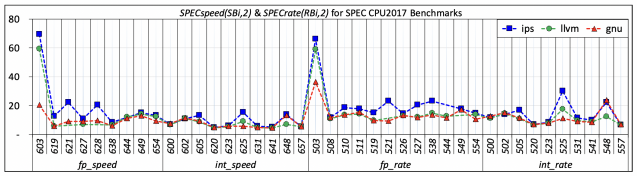

Figure 9: SPEC RATIO FOR 2-THREADS/COPIES (HIGHER IS BETTER)

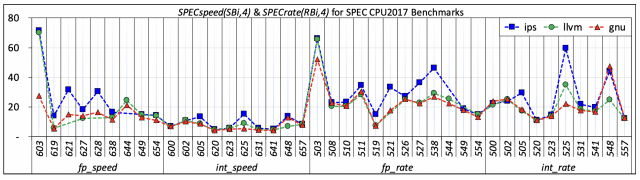

Figure 10: SPEC RATIO FOR 4-THREADS/COPIES (HIGHER IS BETTER)

Overall, IPS executables perform better than LLVM and GNU ones. The LLVM executables are the slowest for int_speed benchmarks. And the IPS executables outperform those created by LLVM and GNU for all benchmarks. The performance of LLVM and GNU are comparable with LLVM doing better for floating-point benchmarks and GNU showing slightly better performance for the integer benchmarks.

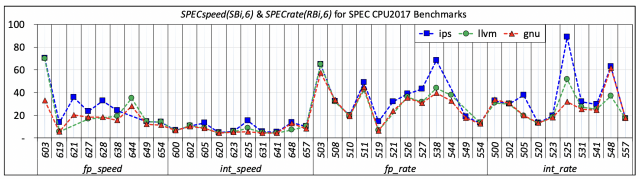

Figure 11: SPEC RATIO FOR 6-THREADS/COPIES (HIGHER IS BETTER)

3.3.3 Research Conclusion. In this measurement-based evaluation, the research look at three widely used compilers (IPS, LLVM, and GNU) using build time, code size, and SPEC Ratio as metrics. And they use the SPEC CPU2017 benchmark suite.

When Considering the size of executable size, the LLVM is the optimal choice for it will produce least code compared to the other compilers, and when build time is considered, GCC is preferred. However, when performance is considered as the metric, IPS outperforms LLVM and GNU due to better exploitation of hardware resources (eg. vectorization, prefetching and cache optimization).

3.4 Performance Comparison of GCC and LLVM on the EISC Processor

In the third research[4] , we research the performance the compilers using code size, build time and SPEC Ratio. And the code quality, especially in terms of and dynamic instructions is also very important in embedded systems. The dynamic instruction count is an indicator of the execution time, and the code size determines memory size of the systems, which is one of the most expensive components. In this study, the researchers compare the code quality of the EEMBC benchmarks from two major compilers, GCC and LLVM on the EISC processor. EISC is an instruction set architecture that takes advantages of simplicity of RISC and small code size of CISC.

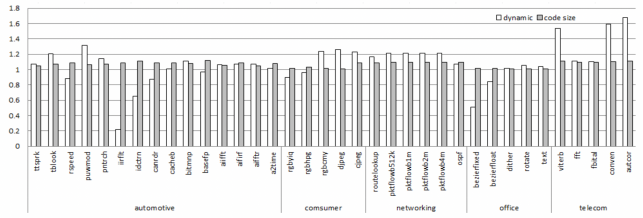

Figure 12: The dynamic instructions and code size by LLVM normalized to GCC.

3.4.1 Research result. Figure 12 shows two performance comparisons: dynamic instruction counts (white) and code size (gray) of EEMBC codes that were compiled by LLVM normalized to those by GCC.

Since GCC has better register allocation, it outperforms LLVM. The GCC compiler tends to use more registers than LLVM, which therefore results in less spills. Due to the reason, most benchmarks, especially in memory intensive codes, which need more registers

to hold base addresses, show better performance with the GCC compiler.

In addition, GCC performs jump optimization better than LLVM. GCC rearranges basic blocks aggressively, and therefore removes unconditional jump instructions and may reduce the code size. In idctrn, an inner-most loop is completely unrolled even if it has a loop unrolling factor of 8. The result indicates that LLVM unrolls loops very actively, and this results in complete removal of loop control overhead. However, loop unrolling increases code size. Owing to that, The static code size of EEMBC benchmarks compiled by GCC is smaller than that of LLVM about 4% on average.

3.4.2 Research Conclusion. In our experiment with EEMBC, when comparing with GCC we found that GCC was good at register allocation and jump optimization. However, GCC did not perform well in loop unrolling and inlining arithmetic function. Especially, GCC performed better than LLVM in memory intensive benchmarks due to register allocation. Overall, the GCC makes dynamic instruction count fewer than LLVM by 18% and reduces code sizes by 4% on average.

4 CONCLUSION

In this research report, we have investigated many perspectives around the GNU Compiler Collection and research many ways of comparison to LLVm/clang and the IPS.

In the research on the performance comparison between GCC and LLVM on RISC-V and ARM platform, we have discovered that when running in Dhrystone benchmark GCC on RISC-V will gain a much better performance on higher optimization levels, however, it will be significantly slower on lower optimization levels, and it has stability as well.

In the interesting research on compiler bugs Comparison with GCC and LLVM, we discovered that most of the bug fixes touch a single source file with small modifications, and the average lifetime of GCC bugs is longer than the LLVM bugs. For both compilers, the bug revealing test cases are typically small.

In the research on the performance comparison among GCC, IPS and LLVM using SPEC CPU2017 benchmark, we find that the LLVM produce least code, and the build time of GCC is least while the IPS has the best performance because it can exploit the hardware resources well.

In the research on the performance comparison of GCC and LLVM on the EISC processor, we can conclude that GCC does well in registering allocation and jumpping optimization. However, GCC did not perform well in loop unrolling and inlining arithmetic function. What's more, GCC generate less static code on average because that the LLVM use much loop unrolling, which increase the code size significantly.

5 ACKNOWLEDGMENTS

Thanks to Professor Liu for the knowledge he taught us this semester. Not only did the theoretical courses prepare and teach very carefully, but he also carefully prepared substantial projects, allowing us to experience and learn the design of compilers from practice, and through this research report, we also learn a lot and gain a deeper understanding of the performance and characteristics of the open source compiler GCC and LLVM. Generally speaking, we have benefited a lot from the course of compiling principles this semester. Thanks again to Prof.Liu and the teaching assistants.

REFERENCES

[1] Sergio Aldea, Diego R. Llanos Ferraris, and Arturo González-Escribano. 2010. Using SPEC CPU2006 to evaluate the sequential and parallel code generated by commercial and open-source compilers. *The Journal of Supercomputing* 59 (2010), 486–498.

[2] Johan Bjäreholt. 2017. RISC-V Compiler Performance:A Comparison between GCC and LLVM/clang.

[3] Ranjan Hebbar S R, Mounika Ponugoti, and Aleksandar Milenković. 2019. Battle of Compilers: An Experimental Evaluation Using SPEC CPU2017. In *2019 SoutheastCon*. 1–8. https://doi.org/10.1109/SoutheastCon42311.2019.9020474

[4] Chris Lattner. 2008. Introduction to the LLVM Compiler System.

[5] Chris Lattner and Vikram S. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* (2004), 75–86.

[6] Chanhyun Park, Miseon Han, Hokyoon Lee, and Seon Wook Kim. 2014. Performance comparison of GCC and LLVM on the EISC processor. In *2014 International Conference on Electronics, Information and Communications (ICEIC)*. 1–2. https://doi.org/10.1109/ELINFOCOM.2014.6914394

[7] Kurt Wall and William von Hagen. 2004. The Definitive Guide to GCC. In *Apress*.