Framework for Comparing Optimizations of LLVM and GCC

Emory Ducote, Jingping Wan
etd4sv@virginia.edu, jht3hc@virginia.edu

***Abstract*—GCC and LLVM have a large number of optimizations. In order to compare these optimizations in a quantitative and structured way, we propose a framework called Compiler-Benchmarker. It first categorize optimizations based on their usages and then run benchmark programs for each group. This framework is able to compare GCC and LLVM optimizations systematically and is open for extensions.**

I. Introduction

Low Level Virtual Machine (LLVM) and GNU Compiler Collection (GCC) are two commonly used compilers. Both of these compilers have a large number of optimizations that can improve the efficiency of compilation and execution. As users of GCC and LLVM, we may wonder which compiler is better at optimizing programs. There have been high-level and theoretical comparisons between LLVM and GCC [5] [3], which compares the general performance of these two compilers and their fitness on specific platforms. However, we want to further compare GCC and LLVM on the optimization level, so that users can better choose which compiler and corresponding optimization sets to choose based on their specific program.

There are three main challenges in comparing the optimizations of LLVM and GCC. First, the design of LLVM and GCC are not identical, which means that there doesn't exist a one-to-one relationship between LLVM and GCC optimizations. Second, a single optimization may be dependent on multiple prerequisite optimizations. To address these two problems, we categorize the optimizations into different groups, where optimizations in the same group may either be dependent on each other or have similar functionalities so that we can compare them in an apple-to-apple manner. Here we take loop optimizations as an example. The third challenge is how to compare GCC and LLVM optimizations comprehensively. We select three benchmark programs (NPB3.0, TSVC2 and BEEBS), covering a wide range of fields while having a focus on loop structures, to make our benchmark suite comprehensive.

In this project, we propose a framework to compare the optimizations of LLVM and GCC. It is composed of the front-end optimization groupings and the back-end benchmark programs. Users can select a pair of optimization group and a benchmark program, then our framework will give a report about the execution time and the size of binary file after compilation respectively. This project is available at https://github.com/chandlerping/CompilerBenchmarker. While we focus on comparing the loop optimizations in this project, and this framework is open to more optimization groups and benchmark programs. At last, we give a report of comparing LLVM and GCC loop optimizations based on our framework as an illustration of usage, which can also serve as a reference for the GCC and LLVM community.

II. Optimization Grouping

While GCC and LLVM optimizations are designed differently, they may have similar functionalities on a higher level. Here we focus on loop optimizations which can be divided into four categories: loop invariants (li), loop unrolling (lu), handling dead loops (dl) and structure & memory (sm). An empty optimization group can serve as the baseline.

A. LLVM loop optimizations

Loop invariants:

- -licm

Loop unrolling:

- -loop-unroll
- -loop-unroll-and-jam

Handling dead loops:

- -loop-deletion

Structure & memory:

- -loop-reduce
- -loop-simplify

B. GCC loop optimizations

Loop invariants:

- -fira-loop-pressure (default by O3)
- -ftree-loop-im
- -fmove-loop-invariants (default by O1 except for Og)
- -fmove-loop-stores (default by O1 except for Og)

Loop unrolling:

- -ftree-loop-ivcanon (preprocess)
- -faggressive-loop-optimizations (default) (preprocess)
- -floop-unroll-and-jam (default by O3)
- -funroll-loop
- -funroll-all-loops

Handling dead loops:

- -ffinite-loops (default by O2)
- -fsplit-paths (default by O3)

Structure & memory:

- -floop-nest-optimize (experimental)
- -ftree-loop-optimize (default by O1)
- -ftree-loop-distribution (default by O3)

III. Benchmark Selection

For choosing which benchmarks to demonstrate for the framework, it was decided that a broad array of program types should be represented to show diversity of test. This diversity of test is important as showing results on one specific application biases the results to that particular application. There exist several benchmarks including the [1] SPEC 2006 that try to achieve diversity of test in itself. However, such benchmarks as [1] SPEC 2006 are proprietary and require significant financial support to gain access to. It was then decided to achieve this diversity of test through an assortment of publicly available benchmarking sets. Thus, when choosing from the broad selection of benchmarks available, tests from scientific computing, vector calculations, and embedded applications were gathered for this demonstration. These three areas cover a large part of the possible programs that can be optimized. In correlation with these three categories [**?**] NPB3.0, [2] TSVC2, and [4] BEEBS were chosen as sample benchmarks for the framework demonstration.

A. NPB3.0

NPB3.0 is a C implementation of the NAS parallel benchmarking dateset from NASA. This dataset is composed of five computational fluid dynamics based applications as well as three equation solvers. This dataset predefines problem sizes and is well documented in relation to its various implementations of algorithms. While the details of the various benchmarks will not be described here, it represents the scientific computing component of the testing and more information can be found when investigating the reference described [**?**] here.

B. TSVC2

TSVC2 is a newer implementation of the C benchmark for vectorizing compliers. This benchmark focuses on the results of compiler optimization on vector operations, and this newer version allows for more elimination of code that previous code formatting prevented. While the details of the various benchmarks will not be described here, it represents the vectorized code component of the testing and more information can be found when investigating the reference described [2] here.

C. BEEBS

BEEBS is a set of benchmarks for embedded programming applications, implementing algorithms common to embedded applications. This benchmark pulls from a variety of different embedded benchmarks, focusing on power consumption and providing a diverse set of embedded applications. While the details of the various benchmarks will not be described here, it represents the embedded code component of the testing and more information can be found when investigating the reference described [4] here.

IV. Framework Design

This framework can be divided into two parts: the front-end which is the optimization grouping and the back-end which is running the selected benchmark program. These two steps are decoupled and users can select any pair of ¡opt group, benchmark¿ set and our tool will give a report, comparing the file size and execution time.

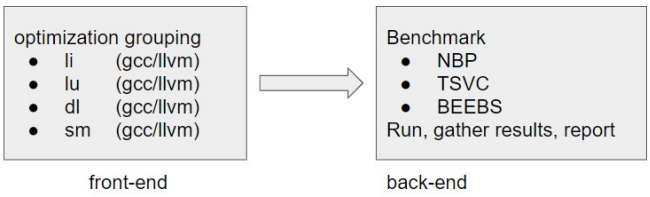

Fig. 1. General framework

The front-end and back-end can be fully decoupled and are both open to extensions. We use a file to store optimization grouping information which can be easily viewed and modified. For the back-end, we use the abstract factory design pattern by defining a general workflow that can be applied to any optimization group and benchmark. The back-end takes the selected optimization group as the input and generate a Makefile containing the corresponding optimization flags. Then it will run the benchmark and report the file size and execution time of the program. This process will be run twice, one for GCC and one for LLVM. Based on this general workflow, we can extend it to any specific benchmark programs by overriding the functions case by case. There is a class with format "¡benchmarkname¿Benchmarker" for each benchmark program, which provides a method called "set_test(group, ...)" that encapsulates the whole benchmarking process, as you can see in the example below.

```
# usage example, main.py
from src.benchmarker.NPB_benchmarker \
   import *
from src.benchmarker.tsvc_benchmarker \
   import *
from src.benchmarker.beebs_benchmarker \
   import *

NPB_b = NPBBenchmarker()
NPB_b.set_test('ft', 'S', 'li')

tsvc_b = TSVCBenchmarker()
tsvc_b.set_test('novec', 'default', 'lu')

beebs = BEEBSBenchmarker()
beebs.set_test('dijkstra', 'sm')
```

V. Test Setup and Results

With the given framework described, setting up tests and performing benchmark comparison's becomes trivial. In the following subsections, an example test design will be described using the loop optimization groupings that have been

defined in the framework. These groupings will be measured with a diverse set of programs being tested.

A. Test Setup

The aim of the test design is to cover a wide range of programs with the predefined looping groups as described. To achieve this, four specific benchmarks from each dataset will be chosen and the results will be measured with each respective loop optimization group. The chosen benchmarks from NPB3.0 are FT, CG, BT, MG. The chosen benchmarks from TSVC2 are 000, 111, 1111, and 112. The chosen bechmarks from BEEBS are dijkstra, nettle-sha256, matmult-int, and bubblesort. These benchmarks with the following loop groups identified: loop invariants (li), loop unroll (lu), remove dead loops (dl), loop structure and memory optimizing (sm). The -O0 grouping will also be run to provide a baseline for the later optimizations.

B. Results

For each of the chosen benchmarks, tests were run with the given optimization grouping in both GCC and LLVM. In the following tables, the BM column identifies the benchmark, the T(llvm) and T(gcc) identify program runtime in llvm and gcc respectively, and S(llvm) and S(gcc) identifies size in bytes of the binary in LLVM and GCC respectively.

For additional context, these benchmarks were run on an AMD Ryzen 9 5900X 12 core processor.

As a baseline for reference the first group run was with the -O0 optimization in both compilers:

-O0 Opt				
BM	T(llvm)	S(llvm)	T(gcc)	S(gcc)
NPB1-FT	7.82	30568	9.3	34344
NPB2-CG	1.91	25928	1.93	25768
NPB3-BT	114.8	168680	133.16	193080
NPB4-MG	3.43	38480	3.53	46488
TSVC1	4.912	104056	5.598	121752
TSVC2	3.02	104056	3.278	121752
TSVC3	8.014	104056	8.328	121752
TSVC4	9.694	104056	9.8	121752
B-dijkstra	0.2698	17432	0.267	17352
B-nettle-sha256	0.00308	21232	0.00208	21200
B-matmult-int	0.07696	20744	0.07018	20648
B-bubblesort	0.05152	16488	0.05524	16392

The optimization grouping of Loop Invariants (li) was run:

-li Opt				
BM	T(llvm)	S(llvm)	T(gcc)	S(gcc)
NPB1-FT	7.72	30568	9.26	34344
NPB2-CG	1.95	25928	1.92	25768
NPB3-BT	116.93	168680	131.43	193080
NPB4-MG	3.41	38480	3.59	46488
TSVC1	4.872	104056	5.467	121752
TSVC2	2.942	104056	3.335	121752
TSVC3	7.891	104056	8.107	121752
TSVC4	8.536	104056	9.99	121752
B-dijkstra	0.27094	17432	0.26592	17352
B-nettle-sha256	0.0031	21232	0.00204	21200
B-matmult-int	0.07642	20744	0.0714	20648
B-bubblesort	0.05022	16488	0.05504	16392

The optimization grouping of Loop Unrolling (lu) was run:

-lu Opt				
BM	T(llvm)	S(llvm)	T(gcc)	S(gcc)
NPB1-FT	7.81	30568	9.25	34344
NPB2-CG	1.98	25928	1.92	25768
NPB3-BT	115.66	168680	134.02	193080
NPB4-MG	3.4	38480	3.53	46488
TSVC1	4.95	104056	5.553	121752
TSVC2	2.913	104056	3.284	121752
TSVC3	8.027	104056	8.12	121752
TSVC4	8.499	104056	9.73	121752
B-dijkstra	0.2764	17432	0.26478	17352
B-nettle-sha256	0.00306	21232	0.00208	21200
B-matmult-int	0.075	20744	0.07078	20648
B-bubblesort	0.05004	16488	0.05508	16392

The optimization grouping of Loop Deletion (dl) was run:

-dl Opt				
BM	T(llvm)	S(llvm)	T(gcc)	S(gcc)
NPB1-FT	7.6	30568	9.06	34344
NPB2-CG	1.97	25928	1.87	25768
NPB3-BT	114.23	168680	129.77	193080
NPB4-MG	3.42	38480	3.54	46488
TSVC1	4.945	104056	5.667	121752
TSVC2	3.001	104056	3.368	121752
TSVC3	7.917	104056	8.428	121752
TSVC4	9.051	104056	9.875	121752
B-dijkstra	0.27686	17432	0.26864	17352
B-nettle-sha256	0.00308	21232	0.00208	21200
B-matmult-int	0.07664	20744	0.0713	20648
B-bubblesort	0.05172	16488	0.05548	16392

The optimization grouping of Loop Structure and Memory (sm) was run:

-sm Opt				
BM	T(llvm)	S(llvm)	T(gcc)	S(gcc)
NPB1-FT	7.8	30568	9.05	34344
NPB2-CG	1.89	25928	1.94	25768
NPB3-BT	114.75	168680	131.41	193080
NPB4-MG	3.37	38480	3.59	46488
TSVC1	4.918	104056	5.601	121752
TSVC2	2.957	104056	3.243	121752
TSVC3	8.108	104056	8.227	121752
TSVC4	8.784	104056	9.784	121752
B-dijkstra	0.27788	17432	0.26542	17352
B-nettle-sha256	0.00302	21232	0.0021	21200
B-matmult-int	0.07684	20744	0.06944	20648
B-bubblesort	0.05232	16488	0.05658	16392

These results were found with comparatively trivial setup time when compared to editing each benchmark individually. Thus, comparison's regarding how each of these optimizations can improve certain types of programs can be made effectively using this data output.

VI. Discussion

While the goal of creating this framework is not to identify exactly how the loop optimizations in LLVM and GCC differ, it is clear that it is easy to do so using this tool. Looking at the data described in the results, one can see that the given optimizations affect the given benchmarks in varingly different ways. There is no consistent optimization grouping that consistently outperforms on all benchmarks, showing that having this diversity of benchmark is key. This fact reveals that benchmarking the particular optimization on application specific benchmarks is paramount when deciding which compiler and optimizations to use. From the data it is also clear that some of the optimizations did not improve performance, or made negligible gains as well. Generally, GCC seems to create smaller binary sizes but LLVM is more effective at cutting down execution time. All of these observations could be made as gathering the data was so straightforward due to the design and implementation of the framework.

VII. Conclusion

In conclusion this report provides a solution to the challenge of comparing specific optimizations accross the LLVM and GCC compilers. By using modular groupings, specific optimization application can be applied at a granular level. This low level approach allows users of the framework to test optimizations in a novel way.

Through using these benchmark groupings, users can save on setup time when analyzing optimizations for their given application. It was demonstrated that biasing these optimizations towards a specific benchmark is easy to do, and if using a benchmark when testing it should be in line with the ultimate intended application being tested. Further, this framework showed that even with this granular level of testing, results may not be entirely clear if the user does not know how the benchmark program can be optimized.

However, the most useful item to come from this endeavor is the framework as described. Adding optimization groupings to this framework is trivial, and the benchmarking class has clear and repeatable functions that can be broken out for whichever benchmark is desired to be used. Thus, this framework allows users to build out a test suite tuned to their chosen application. The majority of the challenge of using this framework is choosing a grouping and benchmark, once those are implemented the use of the framework is trivial as intended.

References

[1] Credits for CPU2006 — spec.org. https://www.spec.org/cpu2006/docs/credits.html. [Accessed 30-Apr-2023].

[2] David Callahan, Jack J. Dongarra, and David Levine. Vectorizing compilers: a test suite and results. *Proceedings. SUPERCOMPUTING '88*, pages 98–105, 1988.

[3] Jae-Jin Kim, Seok-Young Lee, Soo-Mook Moon, and Suhyun Kim. Comparison of llvm and gcc on the arm platform. In *2010 5th International Conference on Embedded and Multimedia Computing*, pages 1–6, 2010.

[4] James Pallister, Simon Hollis, and Jeremy Bennett. Beebs: Open benchmarks for energy measurements on embedded platforms, 2013.

[5] Chanhyun Park, Miseon Han, Hokyoon Lee, Myeongjin Cho, and Seon Kim. Performance comparison between llvm and gcc compilers for the ae32000 embedded processor. *IEIE Transactions on Smart Processing Computing*, 3:96–102, 04 2014.