Research Article

Experimental Analysis Of Optimization Flags In Gcc

[1]Ezhil P, [2]Ananthi Sheshasaayee

[1]Research Scholar(Part Time),
PG and Research Department of Computer Science,
Quaid-E-Millath Government College for Women(Autonomous),Chennai.
Email:ezhilsenthil.research@gmail.com

[2]Research Supervisor,
PG and Research Department of Computer Science,
Quaid-E-Millath Government College for Women (Autonomous),
Chennai.

Article History: Received: 11 January 2021; Revised: 12 February 2021; Accepted: 27 March 2021; Published online: 16 April 2021

Abstract: Compiler performance is critical in improving the overall functionality of a system. Evaluating compiler efficiency is a difficult process because it necessitates the completion of many steps in order to obtain the best result. In this paper, the performance of the GCC compiler's standard optimization levels is experimentally analyzed using the Linux perf tool with selected Collective Benchmark (CBench-V1.1) programs. The analysis is carried out with two parameters, execution time and instruction count which contributes majorly towards the parameters considered for satisfying the compilers design's objective of increasing the speed of execution and minimizing utilization of the memory of a program.

Index Terms: performance, compiler optimization, GCC, standard optimization levels

1.Introduction

The compiler is basically a language translator that translates the high-level language to the target language which is generally machine language or assembly language. For any application built the compiler design plays a significant role in impacting the performance of the overall system[1]. Compilers are designed with an objective to enhance the performance of the system by increasing the execution speed and decreasing the utilization of memory without altering the meaning of the program. Recently modern compiler design also concentrates on ensuring optimal usage of power. Enhancing the performance of the compiler enhances the performance of the overall system.

The GCC (GNU Compiler Collection) compiler is a popular compiler that is available for free distributed by Free Software Foundation(FSF) with multi-language and multi-platform support. The GCC provides more than 200 optimization options for the developer. The GCC also provides different optimization levels comprising of these optimization options for the user to choose from the standard optimization level based on the requirement of the application design. Every application is different from one and another and there cannot be a common optimization level that can be chosen in order to deliver optimal performance. The Standard optimization levels -O1, -O2, -O3, -Os, -Ofast provided by the GCC compiler are designed to enhance the performance of the application[2]. There is a big struggle in the design of modern software due to the availability of new resources like memory, processor, and constraints like parallel programming, power-aware systems.

The number of optimization options available in the GCC compiler creates a challenge for the developer to choose the optimization options that can enhance the overall performance of the system based on the objective of the application design. Though the GCC compiler has provided standard optimization levels -O1, -O2, -O3, -Os, -Ofast with the motive to enhance performance. These optimization levels are not always the optimal ones. Sometimes the implementation of compiler optimization can even degrade the performance[3]. So care has to be taken while choosing the right set of optimization options.

This paper discusses the application of standard optimization levels of GCC compiler on CBench programs and consolidates the results based performance with respect to execution time and instruction count.

2.GCC OPTIMIZATION

Optimizations are used to improve the efficiency of a program without modifying its purpose. The key goal of optimization is to improve overall performance by increasing processing speed and decreasing memory consumption of the code.

There are number of compiler optimization techniques that can be applied to the code. Common sub expression elimination, dead code elimination, constant folding, constant propagation, code movement, strength reduction,

loop optimization, peephole optimization etc., are the compiler optimization techniques that can be applied to the code[1].
The GCC allows the application developers to choose a collection of compiler optimization options provided by GCC. The GNU Compiler Collection (GCC) is a flexible compiler designed to support a number of languages across different platforms and instruction set architectures. It is still one of the most widely used compilers today. The GCC compiler also provides standard optimization levels -O1, -O2, -O3, -Os, -Ofast for the developer to choose based on the application design.
The standard optimization levels of GCC are as described below:[4][5]
A. -O0 or no -O alternative (default)
The compiler does not optimise the source code instructions until converting them to object code. At this stage, the program is compiled using the compile command, which has no special optimization switches. This stage has the advantage of allowing for fast error elimination within the application.
B.-O1 or -O
When a program is compiled according to this standard, the compiler almost ensures that the resulting executable code takes up less space and time. A lot of simple optimizations are done at this point to minimise redundancy and hence the amount of data processing.
Therefore the code runs faster than default level.
C.-O2
In addition to the level –O1 optimization achieved, the compiler makes additional changes at this stage. More advanced methods are used, such as scheduling instructions for faster execution. Compiling the source code takes longer, and it uses more memory throughout the process.
However in this option, when optimising the executable, it is often necessary to ensure that the optimal size of the executable is reached.
D.-O3
Each optimization level is a superset of the one before it, adding all of the previous level's optimizations along with some additional optimizations. Complex methods, such as function inline, are added to the source code in this optimization level, with the benefit of a fast executable but with the drawback of a larger executable. There's still no assurance that the program can be executed in a reasonable time.
E. -Os
The key goal of this optimization level is to provide executables for memory-constrained systems. All of the changes made at this point are aimed at reducing the size of the code without sacrificing performance. Reducing the executable size, as seen in level -03, can allow for more efficient cache memory use. This will sometimes speed up the execution process.
F. -Ofast
This includes all -O3 options, as well as -ffast-math, -fallow-store-data-races, and for FORTRAN -fmax-stack-var-size, -fstack-arrays, and -fno-protect-parens is defined. This alternative is normally not recommended for use because it violates strict standards enforcement.

3.BENCHMARK PROGRAMS

Benchmark suites are a set of software programs that are used to assess the performance of a system's hardware and software. In this experimental model, the CBenchbenchmark suite is used for evaluation.
CBench program’s source code was originally derived from Mibenchprograms[6]. To ensure portability, the derived programs are streamlined. The CBench benchmark suite is a collection of open-source programs organized into different domains with 20 separate data sets that enable users to conduct experiments.
The CBench benchmark programs are categorized into the following categories automotive, security, telecom, consumer, network, office, and bzip.

4.EXPERIMENTAL SETUP

The Collective Benchmark (CBench-V1.1) programslisted in Table 1 under each category areused for the analysis.

CBench Programs	
automotive	automotive_bitcount automotive_qsort1 automotive_susan_c automotive_susan_e automotive_susan_s
security	security_blowfish_d security_blowfish_e
telecom	telecom_adpcm_c telecom_adpcm_d

consumer	consumer_jpeg_c consumer_jpeg_d
bzip	bzip2d bzip2e

TABLE I. CBENCH PROGRAMS USED FOR ANALYSIS

The GCC 9.3.0 compiler has been used and the performance is studied for the following levels of optimization listed in Table II.

Optimization Level in GCC
-O1
-O2
-O3
-Ofast

TABLE II. OPTIMIZATION LEVELS OF GCC APPLIED FOR ANALYSIS

The Linux perf tool is used to measure performance in this experimental setup. The Linux perf tool is a profiler that is used to evaluate the performance of an application depending on a variety of parameters. The Linux perf tool is used to measure both hardware and software counters for GNU/Linux programs, such as CPU cycles, instructions, cache misses, and so on. The perf tool has a wide range of commands for assessing and monitoring performance[7][8].

In this experiment the execution time and the instruction count are taken into consideration as performance parameters. The execution time is used to measure the speed of execution of the program and the instruction count used to measure the memory the program occupies based on code size after the implementation of optimization. These two parameters are measured and taken into consideration for analysis as the main objective of the compiler is to maximize the speed of execution and minimize the utilization of memory.

Fig 1. depicts the performance of standard optimization levels of GCC based on execution time and Fig 2. shows the performance of standard optimization levels of GCC based on instruction count. Fig 3. exhibits the overall performance of CBench programs based on execution time and Fig 4. expresses the overall performance of CBench programs based on instruction count.

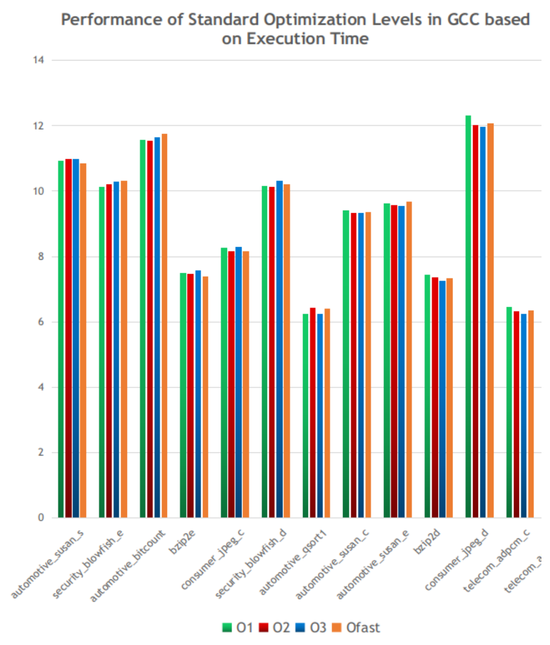

Figure 1. Performance of standard optimization levels of GCC based on execution time

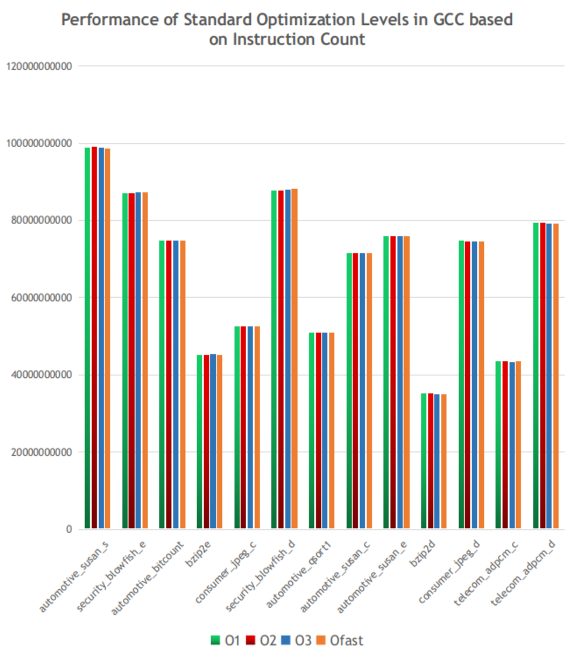

Figure 2. Performance of standard optimization levels of GCC based on instruction count

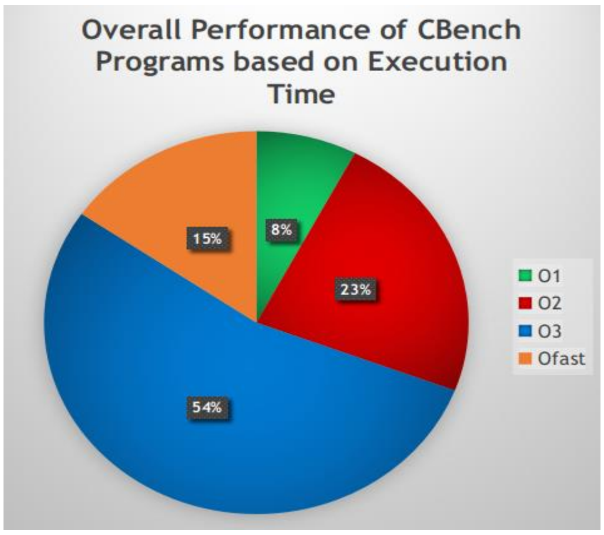

Figure 3. Overall Performance of CBench Programs based on Execution Time

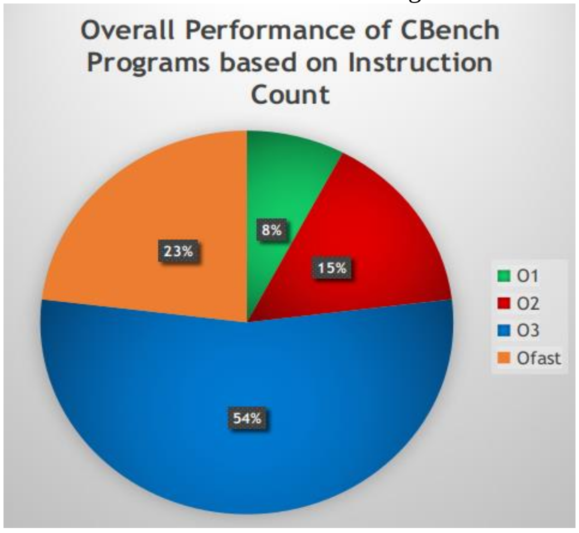

Figure 4. Overall Performance of CBench Programs based on Instruction Count

Figure 5.

From Fig 3. it is evident that the standard optimization level -O3 of GCC has been the overall best performing option as far as execution time is considered. Next overall best option is -O2 then -Ofast and the optimization level -O1 has the least performance with respect to execution time
According to the results of the study in Figure 4, GCC's standard optimization level -O3 has been the overall best performing alternative in terms of instruction count. The next best choice is -Ofast, followed by -O2, with the optimization level -O1 providing the lowest results in terms of instruction count.

5. CONCLUSION

In this paper, the GCC 9.3.0 compiler is analyzed for identifying the performance of standard optimization levels provided by GCC with selected Collective Benchmark (CBench-V1.1) programs. Through the experimental analysis, it is observed that standard optimization level -O3 to be the overall best option as far as performance is considered with respect to execution time and instruction count. The standard optimization level -O1 is recognized to be the least performing option among standard optimization levels with respect to execution time and instruction count.

References

1. Compilers: Principles, Techniques, And Tools"Alfred V. Aho, Monica S. Lam, Ravi Sethi, And Jeffrey D. Ullman
2. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
3. Ashouri, Amir H., William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. "A survey on compiler autotuning using machine learning." ACM Computing Surveys (CSUR) 51, no. 5 (2018): 1-42.
4. "GCC The Complete Reference" Arthur Griffith
5. https://gcc.gnu.org/onlinedocs/gcc-9.3.0/gcc/

6. http://cTuning.org/cbench
7. http://man7.org/linux/man-pages/man1/gcc.1.html
8. Kukunas, Jim. Power and Performance: Software Analysis and Optimization. Netherlands: Elsevier Science, 2015.
9. A.S. Arunachalam, T.Velmurugan. "A Survey on Educational Data Mining Techniques." International Journal of Data Mining Techniques and Applications 5.2 (2016): 167-171.
10. Amir Hossein Ashouri, Gianluca Palermo, and Cristina Silvano. An Evaluation of Autotuning Techniques for the Compiler Optimization Problems. In RES4ANT2016 co-located with DATE 2016. 23 –27. http://ceur-ws.org/Vol-1643/ #paper-05
11. Amir Hossein Ashouri, Vittorio Zaccaria, Sotirios Xydis, Gianluca Palermo, and Cristina Silvano. 2013. A framework for Compiler Level statistical analysis over customized VLIW architecture. In VLSI-SoC. 124–129. DOI:http://dx.doi. org/10.1109/VLSI-SoC.2013.6673262
12. Jose L Ayala, Marisa López-Vallejo, David Atienza, Praveen Raghavan, FranckyCatthoor, and DiederikVerkest. 2007. Energy-aware compilation and hardware design for VLIW embedded systems. International Journal of Embedded Systems 3, 1-2 (2007), 73–82.
13. John Aycock. 2003. A brief history of just-in-time. ACM Computing Surveys (CSUR) 35, 2 (2003), 97–113.
14. R Babuka, PJ Van der Veen, and U Kaymak. 2002. Improved covariance estimation for Gustafson-Kessel clustering. In Fuzzy Systems, 2002. FUZZ-IEEE'02. Proceedings of the 2002 IEEE International Conference on, Vol. 2. IEEE, 1081–1085.
15. David F. Bacon, Susan L. Graham, and Oliver J. Sharp. 1994. Compiler transformations for high-performance computing. Comput. Surveys 26, 4 (dec 1994), 345–420. DOI:http://dx.doi.org/10.1145/197405.197406
16. Victor R Basil and Albert J Turner. 1975. Iterative enhancement: A practical technique for software development. IEEE Transactions on Software Engineering 4 (1975), 390–396.