Imperial Journal of Interdisciplinary Research (IJIR)
Vol-3, Issue-3, 2017
ISSN: 2454-1362, http://www.onlinejournal.in

Review of Techniques for Making Efficient Executable in GCC Compiler.

Sharang Kulkarni[1], Prof. Shafali Gupta[2], Rameez Tamboli[3] & Anil Dake[4]
[1,2,3,4]Dept. of computer engineering, RMDSSOE, Pune, India

Abstract—*Compiler optimization is the technique of minimizing or maximizing some features of an executable code by tuning the output of a compiler. Minimizing the execution time of the code generated is a priority in optimization; other attributes include minimizing the size of the executable code. The right flags used during compilation, would provide substantial performance gain. Though, compilers provide a large number of flags (GNU compiler) to control optimization, often the programmer opts for the simpler method, which is to merely choose the optimization level. The choice of optimization level automatically dictates the flags chosen by the compiler. In this paper, we access at the gain provided by using optimization levels, also we propose a genetic algorithm to determine the combination of flags, that could be used, to generate efficient executable in terms of time, space or both. The input population to the genetic algorithm is the set of compiler flags that can be used to compile a program and the best chromosome corresponding to the best combination of flags is derived over generations, based on the time taken to compile and execute, as the fitness function. The experimental analysis shows that genetic algorithm is a suitable candidate to find an optimal solution if the solution space is large, which otherwise would have been very difficult to identify, due to the large set of flags available in the GCC compiler for optimization alone. Also the best combination of flags is application dependent.*

Keywords*— Compiler Flags; Optimization; Fitness function; Population; Generation.*

1. Introduction

Compiler optimization is the technique of tuning the output of a compiler to minimize or maximize some features of an executable computer program. The goal of optimization is to find the best value for each attribute, in order to achieve satisfactory performance. The measure of the compiler optimization is performance in terms of execution time and the size of the code generated, power-awareness etc. The compiler also determines the type and number of instructions executed for any application, which in turn impacts the overall execution time. Incorporating compiler optimizations improves the performance of the executable code, however the compiler becomes more complex [1].

The most common metric is to minimize the time taken to execute a program; a less common one is to minimize the amount of memory occupied by the code (code size). Compiler optimization is a heuristic approach, since there is no general algorithm for optimization, as the amount and type of optimization possible varies depending upon the application being compiled. However, it is observed that during program execution (i) a program spends large percentage of the time in executing a small part of the code (ii) The part that consumes the most time usually consists of loops (iii) Also, there are parts of the code that are not very frequently executed for e.g. the code that tests for errors in input. From the observations it is clear that the highest percentage benefit in performance can be obtained, if loops are optimized in the generated object code [3].

With that said let's take review of loop optimizations and other optimizations that significantly improve performance.

Loop optimizations - Loop optimization is the process of increasing execution speed and reducing the overheads associated with loops. It plays an important role in improving cache performance and making effective use of parallel processing capabilities. Most execution time of a scientific program is spent on loops; as such, many compiler optimization techniques have been developed to make them faster.
Let's take loot at common loop transformations.

1. **Loop fission/distribution -** Loop fission attempts to break a loop into multiple loops over the same index range but each new loop takes only a part of the original loop's body. This can improve locality of reference, both of the data being accessed in the loop and the code in the loop's body.

2. **Loop fusion/combining** - Another technique which attempts to reduce loop overhead, loop fusion combines the body of two adjacent loops that would iterate the same number of times (whether or not that number is known at compile time), as long as they make no reference to each other's data.

3. **Loop interchange/permutation -** These optimizations exchange inner loops with outer loops. When the loop variables index into an array, such a transformation can improve locality of reference, depending on the array's layout.

4. **Loop-invariant code motion** - If a quantity is computed inside a loop during every iteration, and its value is the same for each iteration (a loop-invariant quantity), it can vastly improve efficiency to hoist it outside the loop and compute its value just once before the loop begins. This is particularly important with the address-calculation expressions generated by loops over arrays. For correct implementation, this technique must be used with loop inversion, because not all code is safe to be hoisted outside the loop.

Peephole optimizations - Usually performed late in the compilation process after machine code has been generated. This form of optimization examines a few adjacent instructions (like "looking through a peephole" at the code) to see whether they can be replaced by a single instruction or a shorter sequence of instructions. For instance, a multiplication of a value by 2 might be more efficiently executed by left-shifting the value or by adding the value to itself. (This example is also an instance of strength reduction.)

Local optimizations -These only consider information local to a basic block .Since basic blocks have no control flow, these optimizations need very little analysis (saving time and reducing storage requirements), but this also means that no information is preserved across jumps.

Global optimizations-These are also called "intra-procedural methods" and act on whole functions. This gives them more information to work with but often makes expensive computations necessary. Worst case assumptions have to be made when function calls occur or global variables are accessed (because little information about them is available).

2.GCC COMPILER OPTIMIZATION LEVELS

Compiler optimization is generally implemented using a sequence of optimizing transformations. The implementation of the algorithms take a program as input and converts it to produce an output with the same functionality as the un-optimized code with improved metrics such as speed of execution and minimal usage of resources. In the GNU C compiler there are a large number of optimization flags and several optimization levels(switches) that control the type of optimization during the compilation process. A compiler flag optimizes a particular feature in a program whereas an optimization level which is a combination of several flags may optimize more than one feature, ensuring a tradeoff between the various metrics. The compiler provides an option of turning on and off either the flags or the optimizations levels (-O1, -O2, - O3, and -O4). In the absence of knowledge about optimization, programmers often merely dictate the optimization level and the compiler imposes the default set of flags associated with that optimization level, accordingly. To be able to achieve good optimization, programmers need to know which exact flag to choose during compilation.
With that being said let's discuss impact of different optimization levels on the input code

-O0 or no -O option (default)

This is the level at which the compiler just converts the source code instructions into object code without any optimization. A compile command with no specific switches enabled, compiles the program at this level. The advantage of this level is it enables easy bug elimination in the program.

-O1

The purpose of the first level of optimization is to produce an optimized image in a short amount of time. These optimizations typically don't require significant amounts of compile time to complete. Level 1 also has two sometimes conflicting goals. These goals are to reduce the size of the compiled code while increasing its performance. The set of optimizations provided in -O1 support these goals, in most cases. These are shown in Table 1 in the column labeled -O1. The first level of optimization is enabled as:

gcc –O1 –o test test.c -O2

The second level of optimization performs all other supported optimizations within the given architecture that do not involve a space-speed trade-off, a balance between the two objectives. For example, loop unrolling and function inlining, which have the effect of increasing code size while also potentially making the code faster, are not performed. The second level is enabled as:

gcc -O2 –o test test.c -O3

The third and highest level enables even more optimizations by putting emphasis on speed over size. This includes optimizations enabled at -O2 and rename-register. The optimization inline-functions also is enabled here, which can increase performance but also can drastically increase the size of the object, depending upon the functions that are inlined. The third level is enabled as:

gcc –O3 –o test test.c -Os

The special optimization level (-Os or size) enables all -O2 optimizations that do not increase code size; it puts the emphasis on size over speed. This includes all second-level optimizations, except for the alignment optimizations. The alignment optimizations skip space to align functions, loops, jumps and labels to an address that is a multiple of a power of two, in an architecture-dependent manner. Skipping to these boundaries can increase performance as well as the size of the resulting code and data spaces; therefore, these particular optimizations are disabled. The size optimization level is enabled as:

gcc –Os –o test test.c

Table 1 will clarify the differences between optimization levels.

3. Impact of Gcc optimization Levels

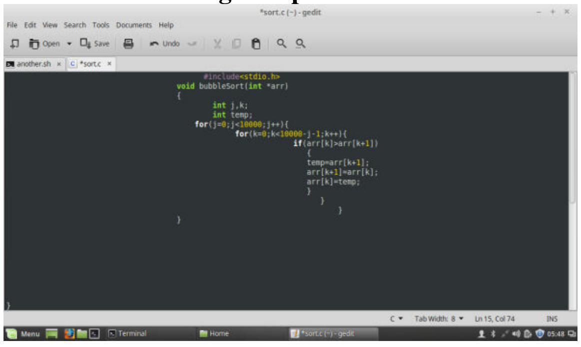
Let's first know the mechanism to evaluate the performance of code compiled with specific optimization levels. /user/nin/time command let evaluate the time feature of executable

The optimization levels of Gcc are tested on sequential bubble sort program for the same data sets. Data set i.e numbers to be sorted are arranged in descending order to enforce worse case scenario.

Fig 1 shows the code for bubble sort

	Included in Level			
Optimization	-O1	-O2	-Os	-O3
defer-pop	●	●	●	●
thread-jumps	●	●	●	●
branch-probabilities	●	●	●	●
cprop-registers	●	●	●	●
guess-branch-probability	●	●	●	●
omit-frame-pointer	●	●	●	●
align-loops	○	●	○	●
align-jumps	○	●	○	●
align-labels	○	●	○	●
align-functions	○	●	○	●
optimize-sibling-calls	○	●	●	●
cse-follow-jumps	○	●	●	●
cse-skip-blocks	○	●	●	●
gcse	○	●	●	●
expensive-optimizations	○	●	●	●
strength-reduce	○	●	●	●
rerun-cse-after-loop	○	●	●	●
rerun-loop-opt	○	●	●	●
caller-saves	○	●	●	●
force-mem	○	●	●	●
peephole2	○	●	●	●
regmove	○	●	●	●
strict-aliasing	○	●	●	●
delete-null-pointer-checks	○	●	●	●
reorder-blocks	○	●	●	●
schedule-insns	○	●	●	●
schedule-insns2	○	●	●	●
inline-functions	○	○	○	●
rename-registers	○	○	○	●

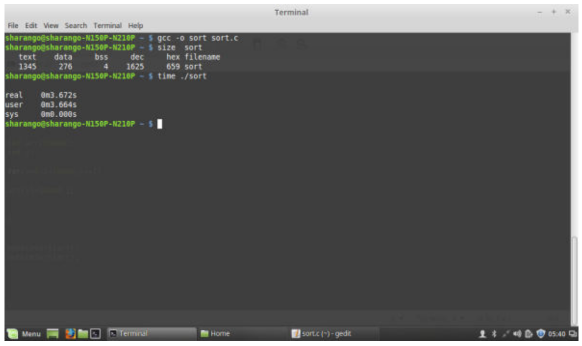
Table. 1 flags in optimization levels


*sort.c (~) - gedit
File Edit View Search Tools Documents Help
Open Save Undo
another.sh *sort.c
C Tab Width: 8 Ln 15, Col 74 INS
Menu Terminal Home *sort.c (~) - gedit 05:48

```
      #include<stdio.h>
    void bubbleSort(int *arr)
    {
            int j,k;
        int temp;
        for(j=0;j<10000;j++){
                for(k=0;k<10000-j-1;k++){
                        if(arr[k]>arr[k+1])
                        {
                        temp=arr[k+1];
                        arr[k+1]=arr[k];
                        arr[k]=temp;
                        }
                          }
                             }
    }
}
```

Fig.1 code for bubble sort

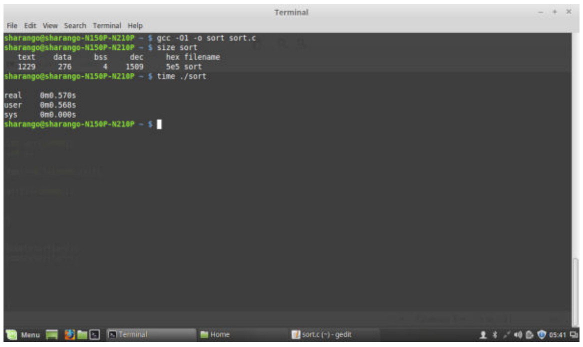
Fig 2,3,4,5,6 shows the executable size and time for – O0 , -O1 –O2 , -O3 ,-Os repectively



```
sharango@sharango-N150P-N210P ~ $ gcc -o sort sort.c
sharango@sharango-N150P-N210P ~ $ size  sort
   text    data     bss     dec     hex filename
   1345     276       4    1625     659 sort
sharango@sharango-N150P-N210P ~ $ time ./sort

real    0m3.672s
user    0m3.664s
sys     0m0.000s
sharango@sharango-N150P-N210P ~ $ 
```

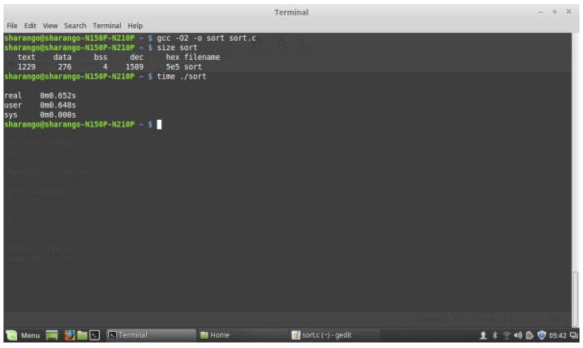
Fig2. Size and time for executable compiled with – O0 level.



```
sharango@sharango-N150P-N210P ~ $ gcc -O1 -o sort sort.c
sharango@sharango-N150P-N210P ~ $ size sort
   text    data     bss     dec     hex filename
   1229     276       4    1509     5e5 sort
sharango@sharango-N150P-N210P ~ $ time ./sort

real    0m0.570s
user    0m0.568s
sys     0m0.000s
sharango@sharango-N150P-N210P ~ $ 
```

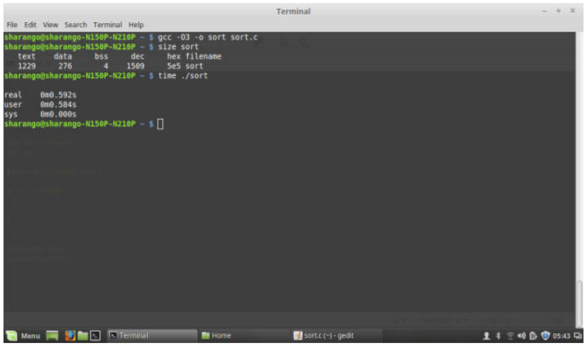
Fig3. Size and time for executable compiled with – O1



```
sharango@sharango-N150P-N210P ~ $ gcc -O2 -o sort sort.c
sharango@sharango-N150P-N210P ~ $ size sort
   text    data     bss     dec     hex filename
   1229     276       4    1509     5e5 sort
sharango@sharango-N150P-N210P ~ $ time ./sort

real    0m0.652s
user    0m0.648s
sys     0m0.000s
sharango@sharango-N150P-N210P ~ $ 
```

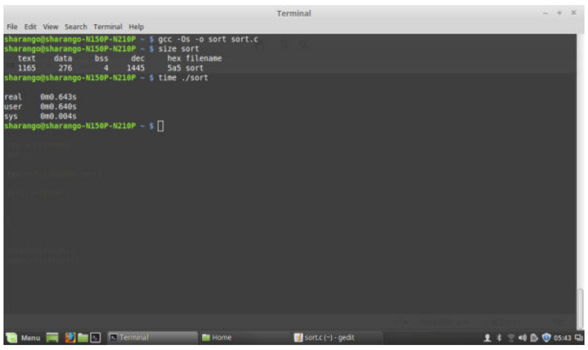
Fig4. Size and time for executable compiled with – O2 level.



```
sharango@sharango-N150P-N210P ~ $ gcc -O3 -o sort sort.c
sharango@sharango-N150P-N210P ~ $ size sort
   text    data     bss     dec     hex filename
   1229     276       4    1509     5e5 sort
sharango@sharango-N150P-N210P ~ $ time ./sort

real    0m0.592s
user    0m0.584s
sys     0m0.000s
sharango@sharango-N150P-N210P ~ $ 
```

Fig5. Size and time for executable compiled with – O3 level.



```
sharango@sharango-N150P-N210P ~ $ gcc -Os -o sort sort.c
sharango@sharango-N150P-N210P ~ $ size sort
   text    data     bss     dec     hex filename
   1165     276       4    1445     5a5 sort
sharango@sharango-N150P-N210P ~ $ time ./sort

real    0m0.643s
user    0m0.640s
sys     0m0.004s
sharango@sharango-N150P-N210P ~ $ 
```

Fig6. Size and time for executable compiled with –Os level.

From above –O1 performs well than -O3 with respect to time .This shows that optimization is dependent on underlying code and optimization levels are not enough.

4. Genetic Algorithm

From the past, up until now, much of the work in the field of optimization has considered evolutionary algorithms as the solution. Evolutionary algorithms or EAs are nature inspired and observe gradual change in characteristics of a particular population or subject. Many previous works on compiler flag selection focused on reducing the search time instead of increasing the performance itself. This approach poses a setback as it assumes that there is no interaction between flags and interaction between flags and underlying source code. Interaction may bring forth improvement in performance. This motivated another evolutionary algorithm approach, Genetic Algorithm, which overcomes all the above mentioned drawbacks by giving out an optimal solution from a large search space of solutions and the programmer need not check for the effectiveness of every possible solution which would be very difficult and time consuming.

Genetic algorithms are search based evolutionary algorithms that imitate the process occurring naturally for selection. They are used to find out an optimal solution among many in a very large search space of solutions. Just like in human body, where the characteristics are determined by genes and the combination of genes becoming chromosomes, genes and chromosomes exist here too. The main steps of genetic algorithm are Selection, Cross-over, Mutation and Termination. Selection is the process of determining which chromosomes are taken into the next generation. The fitness is the value of an objective function which is calculated for every chromosome in the population. It is the measure of how desirable it is to have that chromosome in the population. Based on the fitness value of chromosomes upon cross-over and mutation, the

Selection is done to determine the optimal chromosome. Cross-over is the process of combining two or more chromosomes to derive a new one. Mutation is a process in which a genes change randomly. The evolution starts from population of chromosomes and is an iterative process, with the population in each iteration called a generation. Finally, the GA terminates giving out the optimal solution. In the context of compiler optimization, we consider a flag of the compiler to be a gene and two or more genes i.e. flags combine to form a chromosome[4].

5. Literature Survey And Related Work Done In Past

Before discussing genetic algorithm lets review a work done earlier.
So, from past, till today, a lot of work on this type of a challenge has been carried out having used Evolutionary algorithms as one of the solutions. Evolutionary algorithms or EAs are nature inspired and observe gradual change in characteristics of a particular population or subject. Many evolutionary algorithms like Genetic Algorithm, Simulated Annealing etc. have been considered in the past which determines an optimal solution given a huge search space with no deterministic outcome. Work on compiler optimization was done previously using different approaches by many such as Rodrigo et al. who did an extensive work on evaluation of optimization parameters of GCC compiler [8] Elana Granston et al. gave a framework called "Dr.Options"[9] which automatically recommends the best optimization options for a program. Jeyaraj Andrews et al. focused on evaluating various optimization techniques related to MiBench benchmark applications [10]. The documentation by Wind River Systems gives out the importance of compilers and also the advanced optimization techniques [11].

Further, Scott Robert Ladd synthesized an application called 'Acovea' which is an acronym for Analysis of Compiler Option via Evolutionary Algorithm for compiler flag selection [12]. Many previous works on compiler flag selection focused on reducing the search time instead of increasing the performance itself.

6. Genetic Algorithm Methodology

Evolutionary algorithms or EAs are nature inspired and observe gradual change in characteristics of a particular population or subject. One of the most commonly used evolutionary algorithms is Genetic Algorithm, GA, which selects an optimal solution from a large search space. Just like in human body, how the characteristics are determined by genes and the combination of genes becoming chromosomes, genes and chromosomes exist here also. Here, the compiler flags are the genes. Different combinations of these flags are chromosomes. As in the nature of evolution, chromosomes evolve.
Let's take look at **roulette wheel** selection which is used as selection method for our problem. Basically roulette wheel selection is fitness proportionate selection, This could be imagined similar to a Roulette wheel in a casino. Usually a proportion of the wheel is assigned to each of the possible selections based on their fitness value. Chromosome with
highest fitness will occupy more 'space' on roulette wheel and thus their chance to get selected is increased.

Let's take overview different crossover methods, because this methods influences performance of algorithm on large scale.

Single Point Crossover - A single point is selected on the parents. All the existing content beyond that point in both the individuals is swapped between the two parent chromosomes and the result is the child chromosomes with those features.

Two Point Crossover - As the name suggests, two points are selected on the parent chromosomes. All information contained between the points is swapped, resulting in two child chromosomes or child organisms.

Uniform Crossover - In Uniform crossover the bits are copied in a random fashion either from the first parent or the second parent.

Before going to take loot at genetic algorithm in action, fitness function/metric of chromosome must be defined. In our scenario Fitness metric depends on a feature of executable that user wants to enhance. If user is optimizing for performance in terms of time then the fitness metric will be inverse of execution time or if user is optimizing for performance in terms of space then it will be inverse of code size. If user is going for both time and space then fitness metric will be half of the sum of the inverses of execution time and code size respectively.

Now let's take look at the steps of **genetic algorithm**

Step 1: Consider an initial population of chromosomes i.e. set of compiler flags of the GCC compiler.

Step 2: Consider a sample program of user interest as an input to the Genetic Algorithm.

Step 3: Evaluate performance of population.

Step 4: Use roulette wheel selection and select two chromosomes.

Step 5: Make the cross-over of the chromosomes, being the **set** of compiler optimization flags of the compiler, to generate new chromosomes.

Step 6: mutate each of the chromosome.

Step 7: Evaluate and replace chromosomes generated in previous two steps according their fitness metric.

Step 8: The process is repeated from step 3 and the best chromosome is derived over generations.

Step 9: The termination of the algorithm occurs when the best combination of a set of flags is identified for compiling a particular code.
The final outcome of this process is the chromosome which corresponds to the best fitness value.

Fig 7 illustrates the workflow of system.

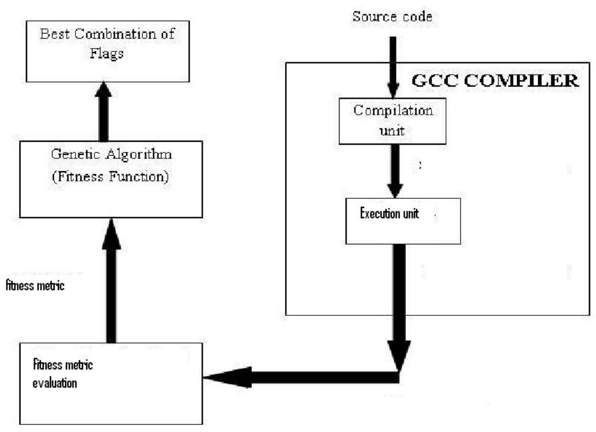

Source code
Best Combination of Flags
GCC COMPILER
Compilation unit
Genetic Algorithm (Fitness Function)
Execution unit
fitness metric
Fitness metric evaluation

Fig 7. Workflow Of System.

7. Acknowledgements

Wet thank RMDSSOE (sinhgad) for providing the support for publishing this paper. Our special thanks to Prof. Veena Lomte, HOD department of computer engineering, for their invaluable support.

8. References

[1] Han Lee1,, Daniel Von Dincklage,1 Amer Diwan,1,_And J. Eliot B. Moss, "Understanding The Behavior Of Compiler Optimizations" Software Practice And Experience, 2004; 01:1–2

[2] Kenneth Hoste Lieven Eeckhout,,COLE: Compiler Optimization Level Exploration,CGO'08, April 5–10, 2008, Boston, Massachusetts, USA.,Copyright 2008 ACM 978-1-59593-978-4/08/04

[3] "*Compilers: Principles, Techniques, And Tools"*Alfred V. Aho, Monica S. Lam, Ravi Sethi, And Jeffrey D. Ullman

[4] http://en.wikipedia.org/wiki/Genetic_algorithm

[5] http://gcc.gnu.org/onlinedocs/gcc/Optimize-ptions.html.

[6] http://www.linuxjournal.com/article/7269

[7] https://en.wikipedia.org/wiki/Optimizing_compiler

[8] Rodrigo D. Escobar, Alekya R. Angula, Mark Corsi, "Evaluation of GCC Optimization Parameters", Ing. USBMed, Vol.3, No.2, pp.31-39, December, 2012.

[9] Elana Granston, Anne Holler, "Automatic Recommendation of Compiler Options", CaliforniaLanguage Lab, Hewlett-Packard Industry, U.S patents 5,960,202 and 5,966,538.

[10] Jeyaraj Andrews, Thangappan Sasikala, "Evaluation of various Compiler Optimization Techniques Related to Mibench Benchmark Applications", Journal of Computer Science 9 (6): 749-756, 2013.