Compiler Optimization: A Genetic Algorithm Approach

Prathibha A. Ballal
Dept. of CSE, NMIT
Bangalore-560064

H. Sarojadevi, Ph.D.
Dept. of CSE, NMAMIT
Nitte-574110

Harsha P S
Dept. of CSE(PG),NMIT
Bangalore-560064

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 generation of fast executables begins at code design phase up until the compilation process is complete. Even though compilers are at the tail end of generating fast executables, the right flag 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. 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). A valid combination of these metrics is called "Pareto optimal" if there is no possibility of further enhancement without degradation of another objective.[2] 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].

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.

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. 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].

We present background and analysis of performance of a quick sort program to understand the impact of optimization in section 2. The methodology used to select the flags for optimization using genetic algorithm is discussed in section 3. A discussion of the experimental setup and performance analysis is presented in section 4 of the paper. Section 5 concludes the paper.

2. BACKGROUND AND RELATED WORK

The quest to optimize all the available resources has always been the goal and at the same time, a challenge to mankind. The same applies to compiler technology also. It is usually of interest to an application developer to make complete use of all the optimizations available. Previously, the user used to specify the optimization switch which in turn sets/resets some flag(s) of the compiler. In order to control the attributes such as compilation-time and memory usage, and the resulting trade-offs between speed of execution and space for the resulting executable, GCC provides a range of optimization levels, each level performing a series of optimizations and also an option to set individual flags to perform a specific optimization as required by the user. There is always a trade-off between the advantage gained due to optimization and the cost incurred to achieve it. The impact of the different optimization levels on the input code is as described below:

-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 or -O

When a program is compiled using this level, the compiler generally ensures that the executable code generated occupies less space and consumes less time, as compared to the code generated at the default compilation level. A lot of simple optimizations are performed at this level, which eliminates redundancy, thereby reducing the quantum of data processing. Hence the code runs faster as opposed to default level compilation.

-O2

At this level, the compiler incorporates additional optimizations, besides the optimization done at level –O1. More complex techniques which schedule instructions for faster execution are performed. It takes longer to compile the source code and also the memory required during compilation is more. However the priority is also, to ensure that when the optimization is done the size of the executable does not increase. This feature makes it the best level for compilation of a code, before deployment. The executable code will be the most optimized for its size.

-O3

Each optimization level is a superset of the previous level, in that, it incorporates all the optimizations done at the previous level with added optimizations. Here complex techniques like function in-lining are applied on the source code, the advantage of which may be a faster executable but the drawback is that the executable becomes bulky. Also, there is no guarantee on the speed of the executable, if the source code does not lend itself well to such type of optimization.

-Os

The main objective of this optimization level is to generate executables for memory constrained systems. All the optimizations applied at this level only aim to reduce the code size without any concern on speed. As observed in level -03, it is possible that the reduction in size of the executable may enable effective use of the cache memory. This may sometimes increase the speed of execution as well. However speed is not a guaranteed feature[5][6][7].

Impact of Gcc Optimization Levels

The optimization levels of GCC are tested on sequential and parallel Quick sort program for the same data sets. In Fig.1, the code executed without any optimization level is observed to be taking the maximum time in seconds.

After using four of the compiler levels the execution time has significantly reduced. Among the levels compared it is observed that -O3 is the best for sequential execution of Quick sort with any size of input data set as –O3 level performs the maximum loop optimization. From Fig.2, there are two main observations:

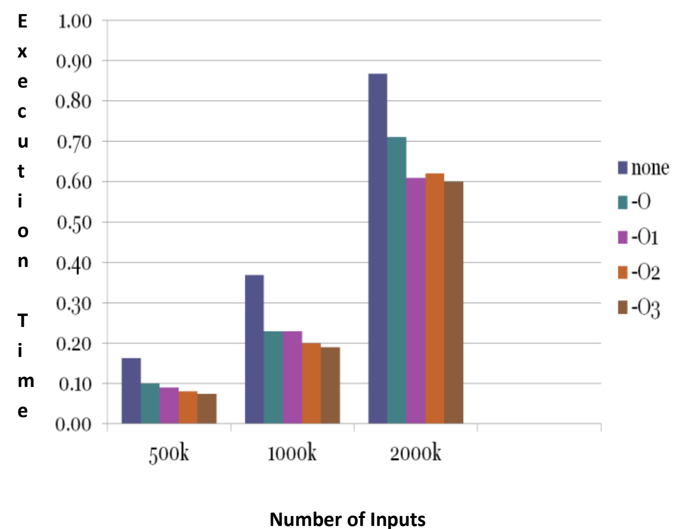

Fig. 1 Performance of GCC optimization levels on sequential quick sort program

(i) The level O3 which performs best optimization of loops exhibited the best result for sequential execution as seen in Fig.1, does not have the same impact for parallel execution, because of the increased overhead of inter-process communication.

(ii) The maximum time taken for execution of the sequential code is 0.88 seconds whereas for the same algorithm and same input size the parallel code takes a maximum execution time of 0.33 seconds.

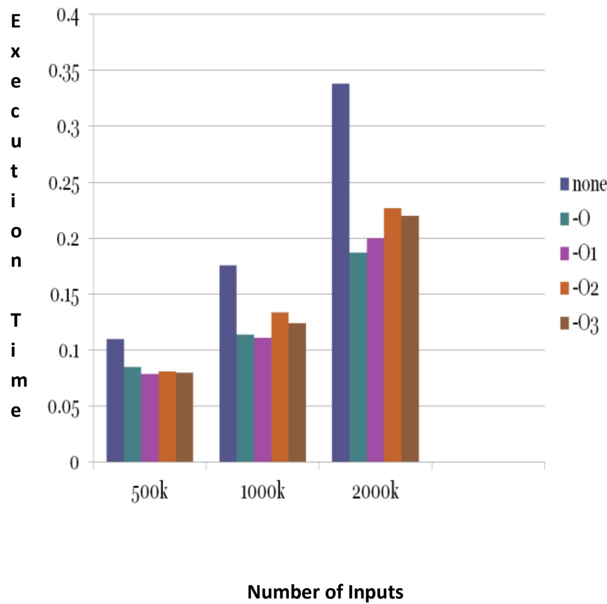

Fig. 2 Performance of GCC optimization levels on parallel quick sort program

For an input size of 2000K the improvement in execution time of parallel code as compared to sequential code is 62.5 percent when the GCC -O switch is used. However, this poses a challenge as to which optimization feature has to be considered for a code to perform the optimization out of a large space of optimization features. 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.

3. 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.

The steps of genetic algorithm are listed below:

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

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

Step 3: Make the cross-over of the chromosomes, being the flags of the compiler, to generate new chromosomes.

Step 4: Compile the sample program using different combination of flags and execute it.

Step 5: The compilation and execution time durations are noted.

Step 6: The Fitness function is derived as a function of compilation and execution times which is as shown below:

$$\text{Fitness} = \frac{1}{Compilation\ \ time + Execution\ \ time}$$

Step 7: Calculate the fitness value for each chromosome in the generation.

Step 8: Find out the chromosome which corresponds to the maximum fitness value and the combination of flags corresponding to that chromosome is said to be the best combination of flags for that particular sample program since it results in least amount of compilation and execution times.

Step 9: The process is repeated and the best chromosome is derived over generations.

Step 10: 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 which in this case is the inverse sum of compilation and execution time.

The GA uses a fitness function to determine the performance of each chromosome. The definition of the fitness function depends on the problem domain. In this, the fitness function is the inverse sum of compilation and execution times. For compilation, the flags in the chromosome from the pool are used to compile the code and this process is timed. The resulting time duration is the compilation time. This represents the amount of time taken to compile that code using different chromosomes. Upon completion of the compilation process, the code is executed and this process is also timed. The resulting time duration is called execution time which represents the amount of time taken to execute that code. Both time durations are summed, its inverse is taken and the resulting value is fed to the algorithm as fitness value. The reason for taking inverse of the sum is that the GA returns the solution corresponding to the chromosome with the highest fitness value as the optimal solution. And the whole aim of the project is to minimize the time taken for compiling the code. Thus, the inverse sum is fed to the algorithm is fitness value. So, lower the sum of compilation and execution times, higher is its fitness value and vice-versa. The design of the working of the algorithm is illustrated in Fig.3.

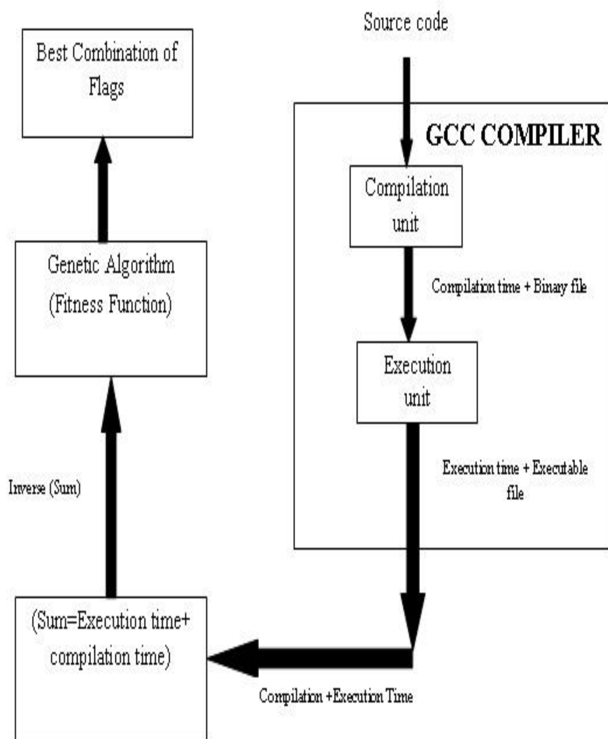

Fig. 3 Genetic Algorithm Usage Flow

4. PERFORMANCE ANALYSIS

The test beds for this are Intel hardware architecture namely, the x86_32 and x86_64. The intention was to determine if this algorithm has a positive impact across various hardware architectures. The source code considered for optimization contained loops, as loops are hotspots for optimization. The recursive(quick sort) and non-recursive (Fibonacci series) programs, was repeatedly compiled and executed.

The gene pool of the flags and the chromosomes used in the genetic algorithm are as listed in Table 1.

Table 1. : Chromosomes of the flags gene pool

Chromosome	Gene Pool	
a	-fexpensive-optimizations	-foptimize-sibling-calls
b	-fcse-follow-jumps	-funroll-all-loops
c	-fforward-propagate	-floop-strip-mine
d	-floop-parallelize-all	-foptimize-sibling-calls
e	-floop-block	-frerun-cse-after-loop
f	-funsafe-loop-optimizations	-floop-interchange

During each of the compilation and execution sessions, the duration was timed. Then, the compiler flags of the next chromosome were used to repeat the above process till the best chromosome was selected. Graphs of chromosomes versus fitness values were plotted for recursive and non-recursive programs to identify the difference in the fitness values for compiling using flags and also without any flags.

The variance in the fitness values for recursive and non-recursive programs is compared, when compiled with and without any flags. The sample code for recursive program is the quick sort program and for non-recursive Fibonacci program was considered. The fitness values for both the sample programs plotted against the chromosomes are as seen in the graphs, Fig. 4, Fig.5, Fig.6 and Fig. 7. The graphs do not start from zero being the basis of the axes value since the obtained results start from 16.27 and 10.53 for Fig.5 and Fig.7 respectively and there is no significant difference between consecutive values plotted to derive a constant interval. The cases of compiling the program with and without having flags were taken in order to see the difference in the fitness values and to determine the best chromosome for that particular kind of program. Fig.4 and Fig.6 depict straight lines in the graphs for recursive and non-recursive programs considered since there is no flag(s) explicitly set/reset for compiling. The default flag(s) are considered for compilation which is set/reset by the default optimization switch “o”. As there is no difference in the compilation times for each program, the fitness value remains same along all the runs resulting in a straight line in the graphs.

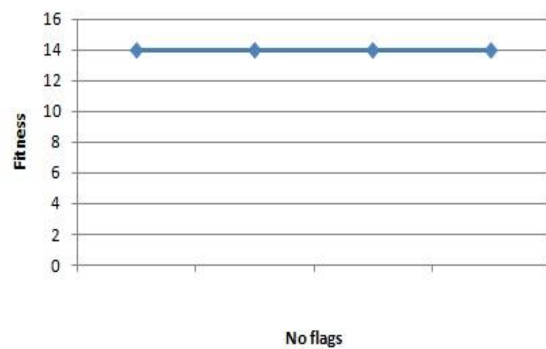

Fig. 4 The comparison of fitness values for non-recursive sample program without any flags.

Fig.5 shows sudden decline in the fitness value and then, it increases as the flags taken to compile the non-recursive program considered change. This shows how flags affect the overall fitness values as some flags increase the compilation time and some decrease it. It always depends on what optimization is being carried out by the flag in the background.

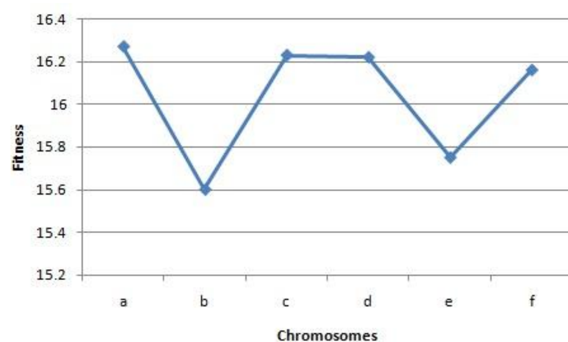

Fig. 5 The comparison of fitness values for non-recursive sample program with flags.

Fig.7 shows gradual decrease in the fitness value indicating that the flags corresponding to the label “a” are the optimal flags for that particular program as they impose optimizations that results in lesser compilation duration. One main point to be noted is that all the optimizations don’t turn out to be useful in all the

cases. So, it is very important to note which optimization is the best for which particular input. It can be seen that Genetic algorithm proves to be very efficient in searching an optimal chromosome, in this case, the best combination of flags from a very large set of compiler flags being the search space for it.

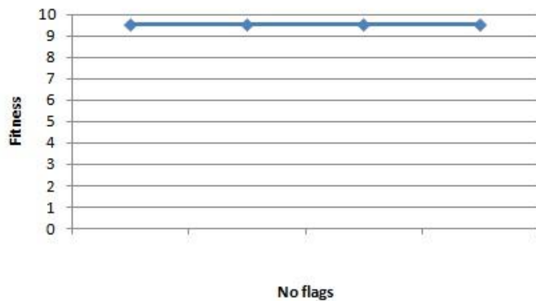

Fig. 6 The comparison of fitness values for recursive sample program without any flags.

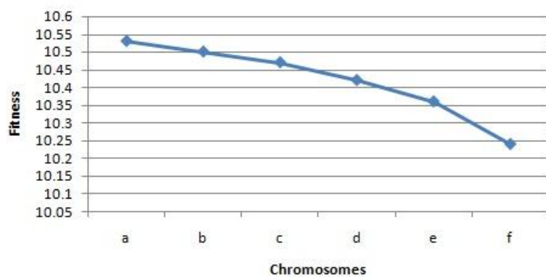

Fig. 7 The comparison of fitness values for recursive sample program with flags.

Note: The labels "a - f" correspond to chromosomes of the genetic algorithm i.e. combinations of compiler flags.

5. CONCLUSION

The experiments conducted on the gene pool considered demonstrates that, different chromosomes have different fitness values for the same program indicating two results

i) The chromosomes generated from the gene pool have a significant impact on the optimization of a program. Hence bigger the gene pool the probability of getting better fitness values for a particular program increases.

ii) The fitness values obtained is program dependent.

The GCC Compiler has about 36 flags for optimization alone. In our future work we will consider more flags in the gene pool. Use of genetic algorithm enables testing of program for many different chromosomes and across several generations easily, which otherwise (manual selection of flags) would have been very cumbersome. The algorithm can also be tested for parallel programs thereby drawing an inference and moving towards supervised machine learning.

6. ACKNOWLEDGEMENTS

We thank Nitte Meenakshi Institute of Technology for providing the support for publishing this paper. Our special thanks to Prof. N.R.Shetty, Director NMIT, Dr. H.C.Nagaraj , Principal NMIT and Dr.Dinesh Anvekar, , HOD department of CSE, NMIT, for their invaluable support.

7. 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.network-theory.co.ukdocs/gccintro/ gccintro_49.html

[7] http://lampwww.epfl.ch /~fsalvi/docs/ gcc/www.network-theory.co.uk/docs/ gccintro /gccintro_42.html

[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", California Language 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.

[11] Wind River Systems, "Advanced compiler optimization techniques" April 2002. Online [December. 2012].

[12] Scott Robert Ladd,, "Acovea: Analysis of Compiler Options via Evolutionary Algorithm" Describing the EvolutionaryAlgorithm", http://stderr.org /doc / acovea /html/acoveaga.html.