# Better Compiler Optimization Options Generation via Genetic Algorithm and Simulated Annealing

Mitali Gupta
Computer Science and Engineering
MS Ramaiah Institute of Technology
Bangalore, India

Saheel Nizam
Computer Science and Engineering
MS Ramaiah Institute of Technology
Bangalore, India

Sibi Akash
Computer Science and Engineering
MS Ramaiah Institute of Technology
Bangalore, India

Dr. Parkavi A
Assisstant Professor
Computer Science and Engineering
MS Ramaiah Institute of Technology
Bangalore, India

**Abstract:- Finding a good complier optimization is particularly a difficult task. Even though provided with various optimizations set levels it requires a good understanding of the various optimizations provided by the complier, proving difficult for most experts in this field. This paper provides case-based approach integrating genetic algorithm and simulated annealing to find better and efficient complier optimization option set (COOS). Implementing the purposed system will improve various standard benchmark parameters.**

***Keywords:-*** *Compiler optimization option set; Genetic Algorithm; Simulated Annealing.*

## I. INTRODUCTION

There exists more than 60 options for optimization but the problem with this is that, same COOS is used for most the programs without keeping in mind the various benchmark constraints different for different programs. Also it requires a great deal of understanding and years of experience to choose for COOS, which is difficult even for an expert.

GNU compiler collection defines four levels which are -O1, -O2, -O3 and –O4. Each implements various Optimization options. User can also specify the COOS but remembering all 60 options with more than 2^60 combinations is impossible and not necessary.

Mitigating the COOS is a challenge, due to the complications with how the optimization interacts with various codes. Thus, explaining why same COOS is used for almost all the source code.

This paper explores the use of Genetic algorithm and simulated annealing to find a good COOS. Genetic algorithm will the source code to find the best COOS while simulated annealing helps reduce the time to find optimizations.

This paper is organized as follows. Section II presents some related work. Section III describes the proposed approach. Section IV describes possible benchmarks optimization. Conclusion and future work.

## II. RELATED WORKS

In our quest to understand this topic and its many implications, we stumbled on a variety of papers.

Valluri and john have provided results of evaluation of the effects of GCC optimization level (superscalar processor). Tullen and seng validated the above work using Intel complier.

Our work doest evaluate the levels but finds the best COOS for different source code which can outperform compiler optimization level.

Cavazos *et al* in his paper uses machine learning to develop a prediction model to mitigate COSP. Park also in his paper proposed three prediction model using machine learning. Almagor *et al* used genetic algorithm to mitigate COSP. Eigemann and Pan proposed batch elimination, combined elimination and iterative elimination algorithm to mitigate COSP.

Raham *et al* proposed a method to tune optimization configurations to reduce power consumption. Hardware performance counter provides feedbacks to calculate and estimate power consumption, using the results to tune configurations.

Our approach implements genetic algorithm along with hardware counters to find the best optimization configuration.

## III. PROPOSED APPROACH WITH ANALYSIS OF COMPLIER OPTION

### *A. Principle Of Compiler*

Compiler is a computer program which converts or translates code written in high level language to code understandable by computer i.e., source code is converted into object code.

A compiler performs all or most of the following functions: lexical analysis, parsing, preprocessing, semantic analysis, code generation and code optimization. This paper targets code optimization.

Figure 1. Clearly explains the main operations of the compiler.

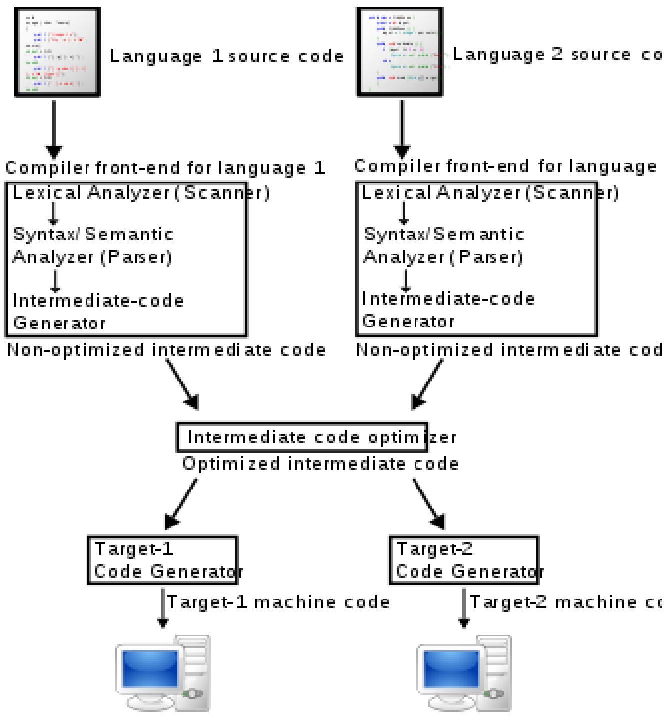
Language 1 source code
Language 2 source co
Compiler front-end for language 1
Compiler front-end for language
Lexical Analyzer (Scanner)
Syntax/ Semantic Analyzer (Parser)
Intermediate-code Generator
Non-optimized intermediate code
Non-optimized intermediate cod
Intermediate code optimizer
Optimized intermediate code
Target-1 Code Generator
Target-2 Code Generator
Target-1 machine code
Target-2 machine cc

Fig 1:- functions of complier

*B. Optimization of Compiler*

Compiler optimization is to minimize certain benchmarks (or maximize efficiency) i.e. tuning the output of the compiler. Command ways are to minimize the execution time, minimize memory occupied etc.

Goal of optimization is to provide best output code (machine code) from the input code (source code). The word best depends on the program and not for the compiler itself. Hence, there arrives a necessity to optimize each program independently requiring algorithms and other hardware techniques.

A simple optimization algorithm might include simply removing the loop invariant while a complex one might include going through the entire source code to remove global sub-expression (common). Optimization might change the code but it shouldn't change the meaning of the code i.e. I what the code is supposed to accomplish must not change. Other optimizations might produce code which uses specific hardware characteristics.

Four levels of GCC for optimization are –O1, -O2, -O3 and –O4. Where each level performs a specific benchmark optimization like –O1 for debugging, -O2 for development along with deploying the code. Higher the number better the optimization will be but the compiler will be slower and take more time to generate the optimization. Example for basic optimization of a program (Prgm.c):

```
gcc –O Prgm.c –o Prgm
```

*C. Compiler optimization with genetic algorithm and simulated annealing*

The data used are DAG test data. The proposed system contains three main parts: space generator, COOS mitigator and COOS validator. Where each performs a specific task to give a best COOS as required by the program.

Space Generator: The sample space S store COOS which acts as a baseline which is generated using simulated annealing algorithm (heuristic algorithm) depending on the program or the source code after code generations. This space S also uses Performance counter set from the hardware to provide better results of COOS at the baseline itself.

COOS Mitigator and validator: From the sample space S which has multiple COOS, a new sample space S' is created using genetic algorithm. S' is created by passing to sample space S to the genetic algorithm and the algorithm runs to find the best COOS which provides maximum benchmark benefits for the given program. Further, S' contains only the COOS which outperform the others. Finally, COOS Mitigator selects the top COOS and gives it to the validator. The validator verifies which COOS is the best to option to compile the program. This COOS is compared to the baseline. Acting as conservative step because if the COOS is similar to baseline. Baseline is returned as the best COOS.

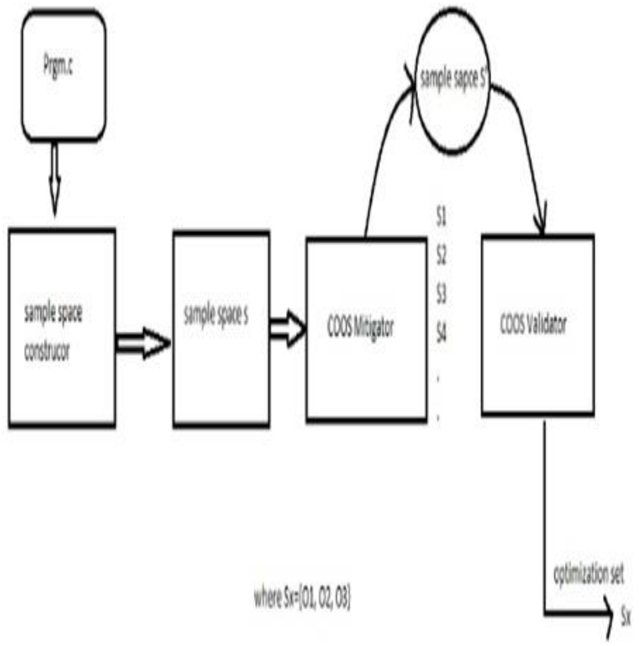

Fig 2:- decribes the processes on how the best COOS is achieved

## IV. BENCHMARK, CONCLUSION AND FUTURE WORK

Compilers are a very important and useful component whose implementation is fairly easy and can help shorten the time of execution of a program.

Implementing the above proposed system will increase the effectiveness, reduce time taken to optimize a code as mentioned in previous session, conservation power consumption etc.

Future work includes developing the proposed system. Using real time DAG test data to produce results and obtaining the expected results.

## REFERENCES

[1.] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. P. O'Boyle, and O. Temam, "Rapidly Selecting Good Compiler Optimizations. using Performance Counters,"
[2.] S. Purini and L. Jain, "Finding Good Optimization Sequence Coverin g Program Space"
[3.] Z. Pan and R. Eigenmann, "Fast and Effective Orchestration of Compiler Optimizations for Automatic Performance Tuning".
[4.] M. Valluri and John L, "Is Compiling for Performance Compilingfor Power?".
[5.] J. S. Seng and D. M. Tullsen, "The Effect of Compiler Optimizations on Pentium four Power Consumptio".
[6.] M. Kandimir, N. Vijay krishnan, M. J. Irwin, and W. Ye, "Influence of Compiler Optimizations on System Power".
[7.] Yang Shen: "Tuning Compiler Optimization Options via Simu lated Annealing".
[8.] Park, S. Kulkkarni, and J. Cavazos, "An Evaluation of Different Modeling Techniques for Iterative Compilation."
[9.] Afonso Ferreira, Pascal Rebreyed, Ricardo C. Correa: "Scheduling multiprocessor tasks with Genetic algorithm".
[10.] Scott Robert Ladd, "GCC 4.0: A Review for AMD and Intel Processors "
[11.] R. Ma and C.-L.Wang "Lightweight application-level task migration for mobile cloud computing".
[12.] Vouk ,"Cloud computing –Issues,Research and implementions".
[13.] Jens Wagner, Rainer Leupers "C Compiler Design for an Industrial Network Processor"
[14.] Hank Shiffman ,"Boosting Java Performance :Native Code and Compiler".
[15.] S. S Muchnick,"Advanced Complier Design and implemention".