International Journal of Research in Engineering, Science and Management
Volume-2, Issue-5, May-2019
www.ijresm.com | ISSN (Online): 2581-5792

Analysis of Compiler Optimization Techniques

Anshul Gupta[1], Ashutosh Pattanaik[2], Atul Rustagi[3], Sini Anna Alex[4]
[1,2,3]*Student, Dept. of Computer Science and Engg., M. S. Ramaiah Institute of Technology, Bangalore, India*
[4]*Assistant Professor, Dept. of Computer Science and Engg., M. S. Ramaiah Inst. of Tech., Bangalore, India*

***Abstract*: Finding the good set of compiler optimisation sequence for a particular program is a difficult task in hand. If the best suited optimisation technique is not chosen, the subsequent compiler settings thus generated can result in the program not executing at its performance peak. This problem can be avoided if there is proper awareness about which compiler optimisation technique to apply given the use case. This paper therefore discusses various compiler optimisation techniques which are known to work well in their domain. The paper helps in providing a clear idea about selecting the appropriate compiler optimisation technique for the given program.**

***Keywords*: Compiler Optimisation, Optimisation Sequence, Compilers, Optimisation Technique Selection, Performance**

1. Introduction

Compiler optimisation deals with managing some of the attributes of the program execution procedure in order to minimise the time taken to execute the program or minimize the total memory used by the program. There are numerous ways to optimise any program. GCC provides some code optimisations each of them changes several optimisation options. Many methods have been proposed to further optimise the optimisations already provided by various compilers. The problem that most of the programmers face is to decide which set of optimisation will make their program most optimised.

Compiler optimisations can be program specific, general purpose or targeting power efficiency. This paper provides an overview of such techniques which are generally used in optimisation. Two of the techniques described in this paper are program specific, one is case based and the other makes use of Genetic Algorithms to find the compiler options [1] [3]. The other two techniques are generic optimization techniques [2] [4].

2. Optimization techniques

A. Optimization of sequences using case based technique

This technique finds the best compiler optimisation by reducing the size of search space greatly. This is done with the help of compiler optimisation sequence (COS) available for other programs. This technique maintains a search space where a performance counter is maintained along with a certain number of COSs for various programs. The performance counter stores the efficiency of each COS when applied to a program. A baseline of efficiency is made for the given program, and all the programs falling above the baseline are stored in reduced search space along with their COS. Then the reduced search space is sorted according to the similarity of stored programs with the given program. The COS of most similar program is applied while compiling the given program and its performance is monitored. If it is not satisfactory then the next most similar program's COS is taken. In case if no program is above baseline then the baseline program's COS is taken as the best optimization sequence.

B. Optimisation of general purpose optimisation

This paper proposes that though it is generally believed that there is no common setting for compiler switches which performs optimally for all the programs, they have derived a compiler setting that gives best results for all programs which is seen by the fact that it increases the efficiency by nearly 20%. But given there are 54 switches, there are 254 possible settings. Therefore, the search space for the optimal setting needs to be trimmed down. This is achieved in two ways:

a) By iteratively adding new switch states to already successful switch settings which are incrementally extended.
b) By creating a representative subset of 254 different settings using an orthogonal array of fractional factorial design.

The algorithm to find a compiler setting:

a) Finding the maximal subsets of positively interacting options. This is done so as to not include the settings which interact negatively together and thus negatively impact the performance improvement.
b) Now the subsets which do not have a negative impact on each other are found. This is done so as to find out which subsets must be combined to give the final set.
c) This step selects the best setting. After getting the set of candidate compiler settings by the two steps (by analysing reduced search space), the search space is still reduced since it is not feasible to execute all settings in S (set of candidate settings) to select the best one. A setting is selected from those settings which have most optimizations turned on. After execution and calculating E(s) (effect of a compiler setting) for each s from S, the compiler setting with maximal E(s) is chosen.

This setting when run through the standard SPECint95 benchmarking suite, the improvement in performance was significantly better. This proves that the proposed systematic

way of finding the optimal compiler setting works the way as designed.

C. Tuning of Compiler Optimisation Options (AcovSA)

GNU compiler collection gives many options while compiling for optimization. GCC provides a built-in set of these options for optimization but user can also choose the set of options. As the number of possible combinations of these set is huge, thus reaching to most optimum is very hard. AcovEA (Analysis of compiler options via evolutionary algorithm) uses genetic algorithms to search for the optimum set of options. It compiles the program with every set and analyse the performance and select the best, but this process is time consuming. This paper introduces a tool AcovSA (Analysis of compiler options via simulated annealing) which reaches to similar result as AcovEA but in less time.

Simulated Annealing is a heuristic algorithm. New benchmarks are used based on multiprocessor scheduling problem with DAG test data. The number of generations is changed which compromises with the problem size and sticks with the original problem definition. DAG scheduling problem is solved using genetic algorithms (GA). First the pre-processing of DAG test data needs to be done and then GA is used to iterate on the solution. To reduce the number of iterations, number of iterations are reduced to 10. Keeping the number of iterations less can be a good compromise for complexity of problem. The proposed alternative approach gives a good optimization option set with better performance.

D. Embedded Softwares - Impact of Compiler Optimisations

In compilers, power and energy optimization can be done by compile time analysis and code reshaping that can be further implemented in hardware along with circuit design. But current compilers are unaware of the energy details of the processor and are only tuned for performance and code size.

In recent times, global optimization levels (-o0 to -o3) are used to study the effects of compilers on power consumption and energy dissipation. In previous works effect of architectural features like dual memory accesses and packing of instructions into pairs, and instruction level power models are used to study energy consumption on programmable processors. Influence of high-level compiler optimization like loop unrolling and fusion were also studied.

When effect of optimization was studied on energy and power consumption it was found that more aggressive optimization (-o3 and -o2) consumes more power when compared to less aggressive one but when software loop pipelining features of -o3 and -o2 were taken consideration then they provide substantial energy savings as compared to -o1, -o0, and no optimization which uses more variables and pre-fetched data for most of the benchmarks.

On analysing the effect of optimization levels on other execution characteristics like cache miss, it was found that more the cache miss more will be the CPU stall cycles but in case of optimization level -o3 which supports pipelining, pipelining miss hides the miss overhead by overlapping the processing of several cache misses. Thus, in case of -o3 level stall cycles decreased but no effect on power consumption. Also, power consumption is inversely proportional to parallelization.

Thus we can conclude that energy usage while running an algorithm with -o3 optimization level is significantly decreased by 95.6%. Also, the aim to decrease power consumption is not much entertained and that is why to target both performance and power consumption it is recommended to use -o3 while disabling the software pipelined loop using -mu in conjunction with -o3.

3. Discussion results

The paper discusses different compiler optimization techniques. These methods are used in different scenarios with a common goal - that is to optimize the performance of the running code or the machine. Depending on the use case, the paper gives an insight about which algorithm to select. For deciding the compiler settings for a program, any technique which focuses on optimising performance of the given program can be used. This results in the choice of two techniques – AcovSA and case based technique. If computation involves executing a program that is similar to that of those already optimised to its best known performance measure, then those optimisation sequences can be used to optimise the given program. This reduces the exploratory state space and causes a performance gain. On the other hand using less or non-similar programs increases number of optimisation sequences, which in turn increases the search time. Hence case-based approach should be used in this scenario.

It can be seen that if execution time of the given program is large, then the process of discovering good compiler options becomes very difficult. For programs with large execution time, AcovSA can be used as it uses GA with reduced number of iterations to find the optimised compiler options. This compromise is acceptable for problems with such large complexity.

For the generic use, the readily available optimisation techniques can be used and/or modified for better performance. These techniques sets the different optimisation options dynamically depending on the program being executed and thus are better for the quick and efficient optimization. Along with these techniques for energy consumption, the pipelining in loop can be introduced. The parallel search for the best optimisation set can greatly reduce the power consumption as well as decrease the execution time.

4. Conclusion

This paper gives an idea of which compiler optimisation technique to be used depending on the use case. This helps greatly in achieving the proper performance and productivity as it eases decision making during testing or development.

Acknowledgment

We would like to acknowledge Management and staff of Ramaiah Institute of Technology in supporting us to do this research work.

References

[1] https://www.embedded.com/electronics-products/electronic-product-reviews/embedded-tools/4086427/Advanced-Compiler-Optimization-Techniques