Journal of Computer Science 9 (6): 749-756, 2013
ISSN: 1549-3636

doi:10.3844/jcssp.2013.749.756 Published Online 9 (6) 2013 (http://www.thescipub.com/jcs.toc)

EVALUATION OF VARIOUS COMPILER OPTIMIZATION TECHNIQUES RELATED TO MIBENCH BENCHMARK APPLICATIONS

[1]Jeyaraj Andrews and [2]Thangappan Sasikala

[1]Department of Computer Science and Engineering, Sathyabama University, Chennai, India
[2]SRR Engineering College, Chennai, India

Received 2013-01-22, Revised 2013-05-16; Accepted 2013-06-06

ABSTRACT

Tuning compiler optimization for a given application of particular computer architecture is not an easy task, because modern computer architecture reaches higher levels of compiler optimization. These modern compilers usually provide a larger number of optimization techniques. By applying all these techniques to a given application degrade the program performance as well as more time consuming. The performance of the program measured by time and space depends on the machine architecture, problem domain and the settings of the compiler. The brute-force method of trying all possible combinations would be infeasible, as it's complexity $O(2^n)$ even for "n" on-off optimizations. Even though many existing techniques are available to search the space of compiler options to find optimal settings, most of those approaches can be expensive and time consuming. In this study, machine learning algorithm has been modified and used to reduce the complexity of selecting suitable compiler options for programs running on a specific hardware platform. This machine learning algorithm is compared with advanced combined elimination strategy to determine tuning time and normalized tuning time. The experiment is conducted on core i7 processor. These algorithms are tested with different mibench benchmark applications. It has been observed that performance achieved by a machine learning algorithm is better than advanced combined elimination strategy algorithm.

Keywords: Machine Learning, Program Features, Compiler Optimization, Mibench

1. INTRODUCTION

Modern architecture designer strives to bring satisfactory system level performance by applying minimal power across a wide range of applications. But many compilers fail to deliver its performance because of rate of change in hardware evolution. A compiler usually provides a larger number of optimization options, from which users has to pick up best set of available options for a given application. Those who do not have in depth understanding of the compiler options and interactions among compiler options, then it is really difficult to pickup best set of options. Compilers usually provide three levels of optimization techniques. They are -O1, -O2 and -O3. As compiler optimization interacts in unpredictable manner in different architecture, finding an effective orchestration algorithm to search for the best combinations of optimization options is desired. Automatically selecting the best set of compiler optimizations for a particular program is a difficult task. Many existing framework available to select best set of optimal compiler setting from larger set of options. Andrews and Sasikala (2012) used a new algorithm called as Advanced combined elimination which is a modified combination of batch elimination and combined elimination which provides the complexity of $O(n^2)$. Fursin and Temam (2011) used an algorithm called as Random search strategy which picks up best set of combinations in quick time. Park and Cavazos (2011) used different modeling techniques to find best set of combinations for a given benchmark applications.Combined elimination gives better results than other algorithms and only fewer

Corresponding Author: Jeyaraj Andrews, Department of Computer Science and Engineering, Sathyabama University, Chennai, India

evaluations required to find optimal settings. However, these pure search or "orchestration" approaches do not use prior knowledge of the hardware, compiler, or program and instead attempt to obtain this knowledge online. Every time a new program is optimized, the system starts with no prior knowledge. In this study machine learning has been used in a modified form, which has the potential of reusing knowledge across iterative compilation runs, gaining the benefits of iterative compilation while reducing the number of executions needed. In this study we have selected GCC as the compiler infrastructure. GCC is currently the only production compiler that supports different architectures and has multiple aggressive optimizations making it a natural vehicle for our research. GCC provides three levels of optimization techniques. To obtain the best performance a user usually applies the highest optimization level-O3. In this level the compiler perform the most extensive code analysis and expects the compiler generated code to deliver the highest performance. In this study we have proposed an automated framework to select the compiler options for a particular problem from large set options. Many previous works consider only limited set of options. For this framework, we have implemented compiler optimization selection algorithm advanced combined elimination strategy. This algorithm is compared with machine learning algorithm. Efficiency was evaluated to improve its tuning time and normalized tuning time. The study is organized as follows.

1.1. Framework Architecture

The optimization selection algorithm picks up best set of optimization techniques from 'n' available number of optimization techniques. These techniques applied for a given benchmark applications and compiled with GCC compiler. The target code is then analyzed with performance tools using Intel Vtune performance analyzer to collect set of program features. The performance was analyzed for improving the execution speed up and compilation time. This information is then given feedback option selector algorithm to pick up another best set. **Figure 1** shows detailed description of framework architecture.

1.2. Selection Algorithms

Given a set of "n" ON-OFF optimization options {F_1, F_2…Fn), find the best combination of flags that minimizes application execution time and compilation time.

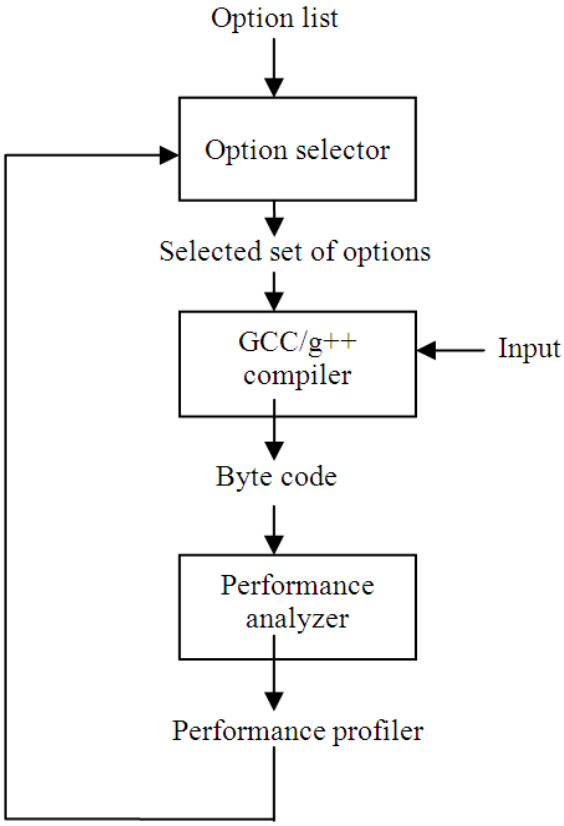

Fig. 1. Optimization selection framework

In this study a novel performance tuning algorithm advanced combined elimination algorithm is compared with a machine learning algorithm which picks up best set of options to improve tuning time and normalized tuning time.

1.3. Advanced Combined Elimination Strategy

Let S be the set of available optimization options:

- Let B represents selected compiler options set
- Find T_B, by applying all flags are on
- Compile the program with T_B configuration and measure the program performance
- Calculate Relative Improvement Percentage (RIP) for each and every optimization options. Relative improvement percentage is calculated based on finding the time required by applying particular flag ON and OFF with respect to T_B
- Store all the values in an array based on ascending order. i.e., the most negative RIP is stored in first position of the array
- Remove the first two most negative RIP's from an array instead of one. Now the value of T_B is changed in this step

- Remaining values in an array i.e., i vary from 3 to n, Calculate RIP and store the negative RIP's in array
- If all values in an array represent positive values then set of flags in B represents best set
- Else
- Repeat steps ii until B contains only positive values
- Stop

1.4. Machine Learning Algorithm

The logistic regression model is a machine learning (Hung *et al*., 2009) technique used to pick up set of options from a trained dataset. For a larger benchmark applications, finding the best set of compiler options will take more amount of time. To find a best set with less number of evaluations, we proposed a machine learning strategy. Collect set of program features for a given benchmark applications for a specific hardware during the training stage itself.

For training we have collected more than 1000 set of combinations. These combinations compiled with gcc or g++ compiler and record the execution speed up. For collecting program features Intel Vtune performance profiler used. For collecting static program features Milepost GCC machine compiler used (Fursin and Temam, 2011). The model is evaluated based on leave one out cross validation procedure. i.e., if we have consider N = 10 (Where N is number of benchmark applications), i.e., the models are trained on N-1 benchmarks and tested on the N^{th} benchmark. The models were trained with 9000 points. The programs were compiled with 1000 sets of compiler setting and the performance is measured for a specific hardware platform. The various information such as compilation and execution time is stored on the repository. After training stage if a similar kind of program arrives by looking database one can who quickly searches best set of optimal settings.

1.5. Experimental Procedure

In this study we have considered recent version of GCC compiler. GCC provides different levels of optimization techniques. Previous work considered only limited set of optimization techniques. In this study more number of optimization techniques considered. **Table 1** show different levels of optimization techniques from -0 to -o3.o3 is the highest level techniques. Level 1 consists of important techniques such as floop-optimize, dead code elimination, ftree-dce, dead store elimination, ftree-dse and scalar replacement of aggregates. Level 2 consists of important techniques such as global common sub expression elimination, gcse, peephole optimization, fpeephole2 and various basic block optimization techniques and scheduling optimization techniques. Level 3 consists of inline functions and unrolling. Although optimization level 3 (-O3) can produce faster code, the increase in the size of the binary image can have adverse effects on its speed.

1.6. Mibenchmark Programs

The Mibench benchmark suite programs are used to experiment the proposed algorithm. These benchmark suites are comparable with SPEC benchmark suite.

1.7. Bzip2

Bzip2 is a free and open source implementation of the Burrows-Wheeler algorithm. Bzip2 compresses most files more effectively than the older LZW (.Z) and Deflate (.zip and .gz) compression algorithms, but is considerably slower. Bzip2 compresses data in blocks of size between 100 and 900 kB and uses the Burrows-Wheeler transform to convert frequently-recurring character sequences into strings of identical letters.

1.8. Consumer_jpeg_c

The JPEG standard allows "Comment" (COM) blocks to occur within a JPEG file. Although the standard doesn't actually define what COM blocks are for, they are widely used to hold user-supplied text strings. This lets add annotations, titles, index terms, in JPEG files and later retrieve them as text. COM blocks do not interfere with the image stored in the JPEG file. Maximum size of a COM block is 64K.

Consumer_tiff2bw Tiff2bw converts an RGB or Palette color TIFF image to a grayscale image by combining percentages of the red, green and blue channels. By default, output samples are created by taking 28% of the red channel, 59% of the green channel and 11% of the blue channel. To alter these percentages, the -R, -G and -B options may be used.

1.9. Qsort

The sort test sorts a large array of strings into ascending order using the well known quick sort algorithm. The small data set is a list of words; the large data set is a set of three-tuples representing points of data.

1.10. Dijkstra

The Dijkstra benchmark constructs a large graph in an adjacency matrix representation and then calculates the shortest path between every pair of nodes using repeated applications of Dijkstra's algorithm.

1.11. Patricia

A Patricia tries is a data structure used in place of full trees with very sparse leaf nodes. Branches with only a single leaf are collapsed upwards in the tries to reduce traversal time at the expense of code complexity. Often,

Patricia tries are used to represent routing tables in network applications. The input data for this benchmark is a list of IP traffic from a highly active web server for a 2 h period. The IP numbers are disguised.

1.12. Security Blowfish

Blowfish is a keyed, symmetric block cipher, included in a large number of cipher suites and encryption products. Blowfish provides a good encryption rate in software and no effective cryptanalysis of it has been found to date.

1.13. Susan

SUSAN is an acronym standing for Smallest Univalve Segment Assimilating Nucleus. For feature detection, SUSAN places a circular mask over the pixel to be tested (the nucleus). For corner detection, two further steps are used. Firstly, the centroid of the SUSAN is found.

1.14. Metrics used for Evaluation

Relative Improvement Percentage (RIP), RIP (Fi), which is the relative difference of the execution times of the two versions with and without Fi Equation 1:

$$RIP(Fi) = T(Fi = 0) - T(Fi = 1) \div T(Fi = 1) \times 100 \quad (1)$$

If Fi = 1 then Fi is ON, else OFF.

The baseline of this approach switches on all optimizations.

T_B = T(Fi = 1) = T(F1 = 1, F2 = 1,…Fn = 1),Where TB represents base time Equation 2:

$$RIP(Fi = 0) = T(Fi = 0) - TB \div TB \times 100\% \quad (2)$$

If RIP (Fi = 0) <0, the optimization of Fi has a negative effect, so it is better to turn off the function.

1.15. Tuning Time

It is the time taken by each probe, to determine the effect of individual options in a set of candidate options.

1.16. Normalized Tuning Time

It is the time taken for computing time needed to check the effects of individual options. It is calculated using the following equation.

NTT = tuning time for entire probe/(number of re executions*total candidates).

2. MATERIALS AND METHODS

Advanced combined elimination algorithm and machine learning algorithm is implemented. Then the normalized tuning time is calculated using the above equation. Architecture used for testing was Intel Corei7 - 2630 QM CPU 2.2 Ghz. With 8GB RAM, using ubuntu operating system and the compiler was GCC 4.3.2.

3. RESULTS

Table 1 shows list of chosen optimization techniques for a GCC compiler. Results obtained from the experiment are tabulated in **Table 2**. **Table 2** represents Normalized tuning time.

Table 1. List of optimization techniques

Level-o1 techniques	Level-o2 techniques	Level-o3 techniques
fcprop-registers	falign-functions	fgcse-after-reload
fdefer-pop	falign-jumps	finline-functions
fdelayed-branh	falign-loops	funswitch-loops
fguess-branch-probability	falign-labels	
fip-conversion	fcaller-saves	
fip-conversion2	fcross-jumping	
floop-optimize	fdelete-null -pointer-checks	
fmerge-constants	fexpesive -optimizations	
fomit-frame-pointer	fforce-mem	
ftree-ccp	fgcse	
ftree-ch	fgcse-lm	
ftree-copy-rename	fgcse-sm	
ftree-dce	foptimize-sibling-calls	
ftree-dominator-opts	fpeephole2	
ftree-dse	fregmove	
ftree-fre	freorder-blocks	
ftree-lrs	freorder-functions	
ftree-sra	frerun-cse-after-loop	
ftree-ter	frerun-loop-opt	
	fsched-interblock	
	fsched-spec	
	fschedule-insns	
	fschedule-insns2	
	fstrength-reduce	
	fstrict-aliasing	
	fthread-jumps	
	ftree-pre	
	fweb	

Table 2. Normalized tuning time in seconds

Benchmark applications	Ace	Machine learning algorithm
Bzip2	0.000130	0.000020
Consumer_jpeg.c	0.000170	0.000030
Consumer_tiff2bw	0.000190	0.000030
Network_dijkstra	0.000025	0.000010
Network_Patricia	0.000080	0.000020
Qsort	0.000140	0.000030
Security_blowfish	0.000230	0.000035
Susan	0.000170	0.000025

4. DISCUSSION

In this stydy, we compare advanced combined elimination algorithm with a machine learning algorithm to find tuning time and normalized tuning time. **Figure 2** shows comparison of ACE and a machine learning algorithm. For most of the benchmark applications LRM gives least tuning time when compared to advanced combined elimination algorithm, because program features can be extracted and stored in a database. So if a similar program arrives with help of database one can quickly select best set of techniques. For some of the benchmark applications especially bzip2, dijkstra and qsort advanced combined elimination gives more or least tuning time when compared to LRM. **Figure 3** shows comparison of execution time of different levels of GCC compiler optimization techniques with advanced combined elimination algorithm. GCC compiler consists of three different levels of optimization techniques.

They are -o1, -o2 and highest level optimization techniques -o3. If by applying only the set of techniques from -o1 may reduce the compilation time but not so much of an performance on the execution time. So by considering set of techniques from -o2 improves execution time for most of the benchmark applications. By applying -o3 techniques for a given application may increase compilation and code size, but improves the program performance.

A good optimization algorithm should achieve both program performance and short tuning and short normalized tuning time. **Figure 3** shows when compared to different levels of optimization techniques, ACE gives least execution time for all the benchmark applications. **Figure 4** shows comparison of normalized tuning time between ACE over LRM. Normalized tuning time is calculated by finding tuning time for each probe divided by number of re executions multiplied by total candidates. **Table 2** shows normalized tuning time for every benchmark applications. **Figure 5** shows execution time speed up between different levels of optimization techniques over LRM. **Figure 6** shows combined execution time between ACE and LRM over different levels. From **Fig. 6** we can conclude that LRM achieves both program performance and short normalized tuning time for most of the benchmark applications.

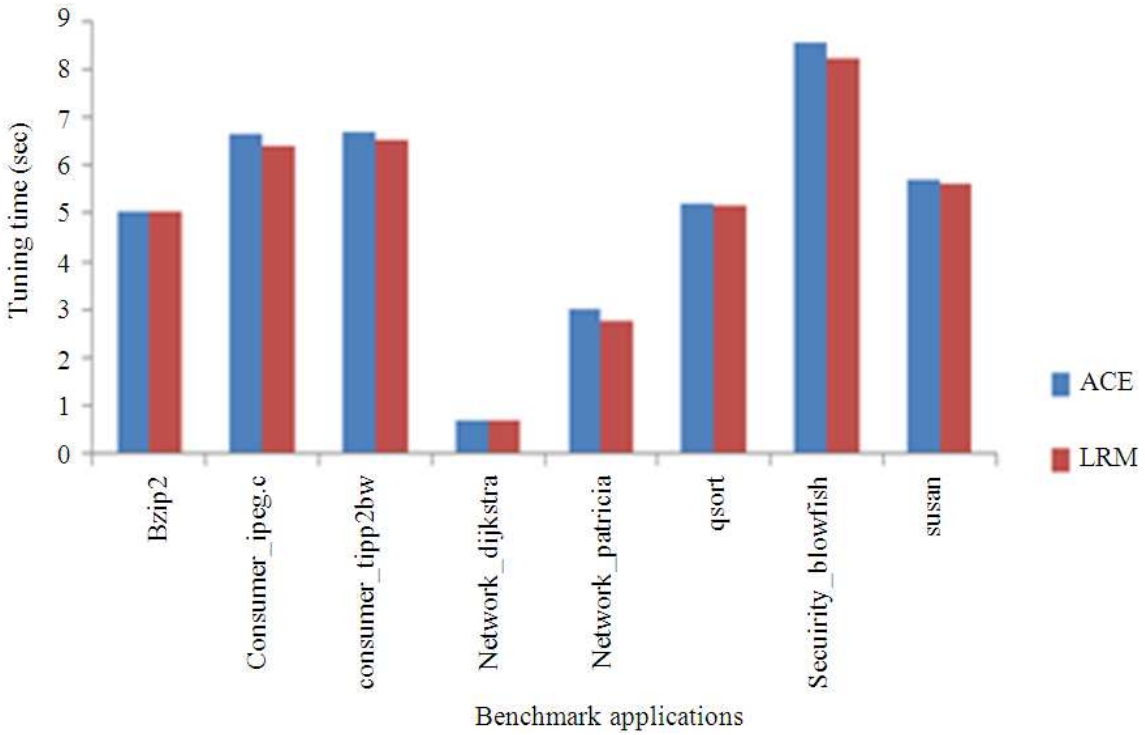

Fig. 2. Comparison of tuning time

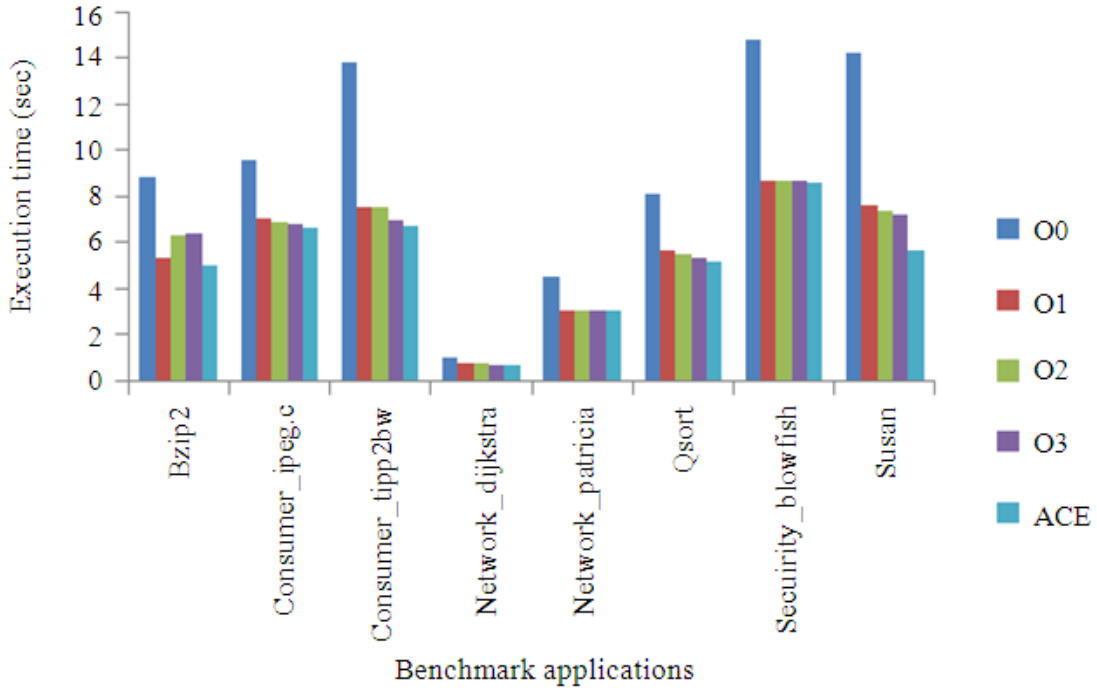

Fig. 3. Execution time over ACE

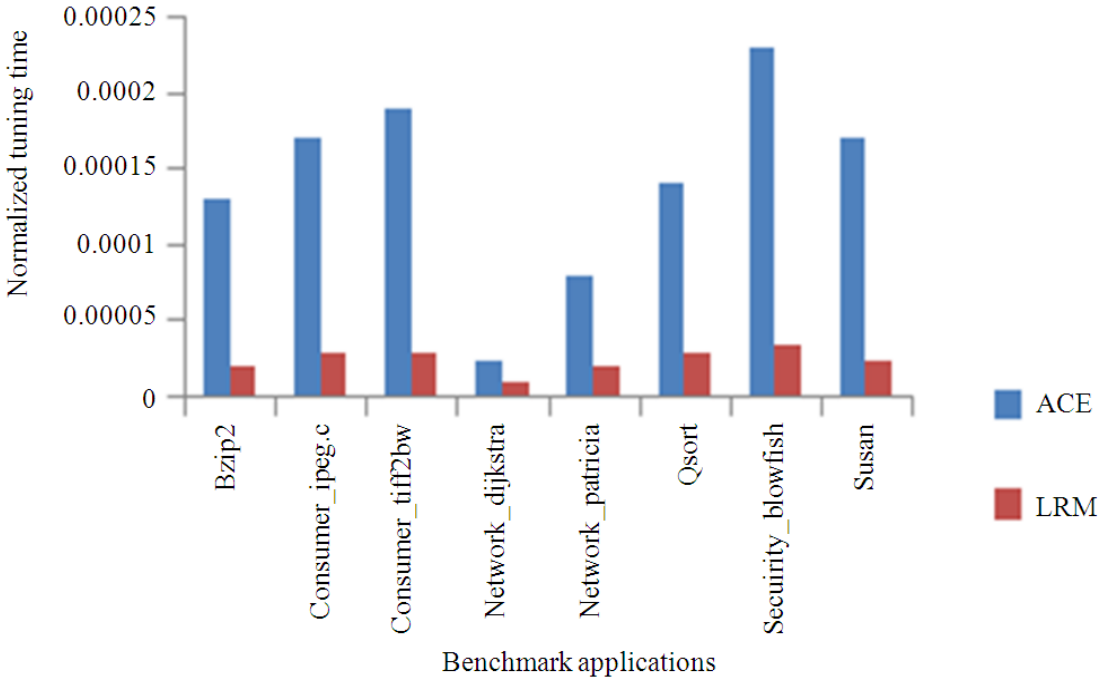

Fig. 4. Comparison of normalized tuning time

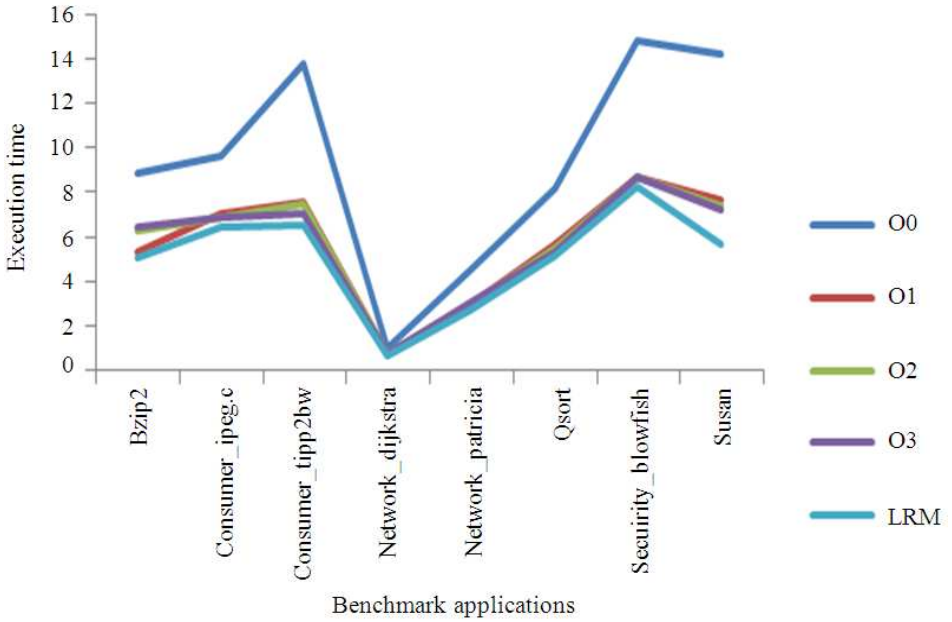

Fig. 5. Execution time over LRM

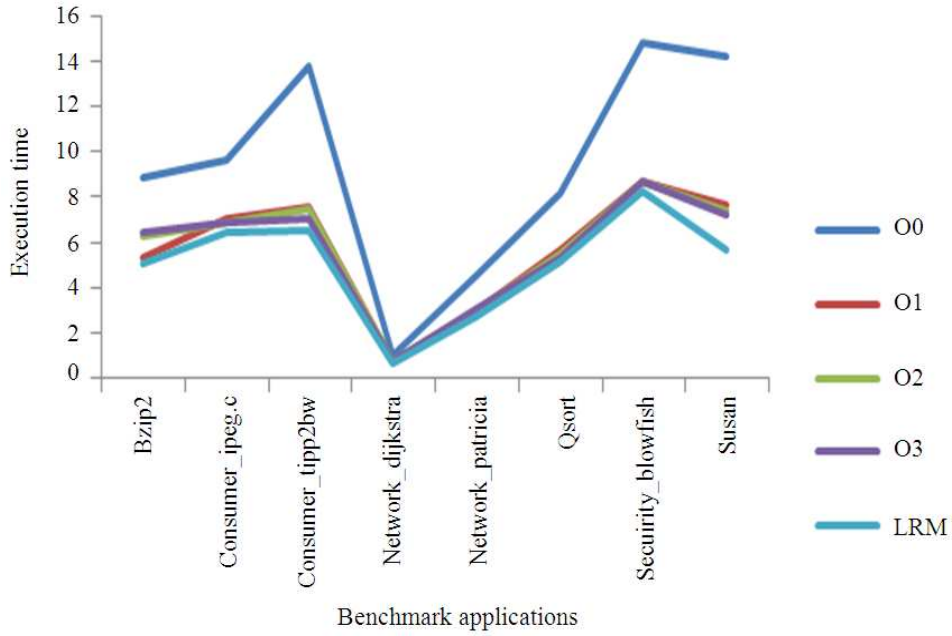

Fig. 6. Comparison between different levels over ACE and LRM

5. CONCLUSION

In this study, an alternative framework proposed for finding tuning time and normalized tuning time for Mibench benchmark applications. In this framework we have integrated Milepost GCC v2.1 and Intel Vtune performance analyzer for extracting program features upon training stage. These in formations are stored in a

repository. So with the help of this information one can find best set of optimization techniques if a similar kind of program arrives. For this frame work we have implemented advanced combined elimination strategy. The results are compared with a machine learning algorithm. The results show that machine learning algorithm which improves the program performance, tuning time and normalized tuning time.

In the future, we incorporate more compiler option selection algorithms to improve tuning time and normalized tuning time. In future we incorporate LLVM, ROSE and path64 and other compilers in our framework. In future we include simulators in our framework to enable software and hardware optimization.

6. REFERENCES

Andrews, J. and T. Sasikala, 2012. Performance enhancement of tuning time in GCC compiler optimizations using benchmark applications. Int. J. Artf. Intell. Syst. Mach. Learn., 4: 276-281.

Fursin, G. and O. Temam, 2011. Milepost GCC: machine learning enabled self-tuning compiler. Int. J. Parallel Programm., 39: 296-327. DOI: 10.1007/S10766-010-0161

Hung, S.H., C.H. Tu, H.S. Lin and C.M. Chen, 2009. An automatic compiler optimizations selection framework for embedded applications. Proceedings of the International Conference on Embedded Software and Systems, May 25-27, IEEE Xplore Press, pp: 381-387. DOI: 10.1109/ICESS.2009.86

Park, E. and J. Cavazos, 2011. An evaluation of different modeling techniques for iterative compilation. Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems, (ES' 11), ACM Press, New York, USA., pp: 65-74. DOI: 10.1145/2038698.2038711