Efficient framework architecture for improved tuning time and normalized tuning time

J.ANDREWS, T.SASIKALA
Research Scholar, Department of CSE
Sathyabama University
Rajiv Gandhi salai, Chennai-600119
INDIA
andrews_593@yahoo.com,sasi_madhu2k2@yahoo.co.in

Abstract: **-** To improve the performance of the program by finding and applying the best set of techniques from the set of available techniques for the GCC Compiler is a non-trivial task. GCC Compiler provides three different levels of optimization techniques. Some techniques are turned ON or OFF each time. Turning on all the techniques for an application may degrade the program performance and increase the compilation time. The selection is dependent on the program domain, the compiler setting and the system architecture. The objective is to find the best set of techniques from the various options provided by the GCC compiler. The framework should handle all the new set of techniques that get added with the new release. The framework should be capable of finding the best set of optimization techniques for any program that is provided as input. Since there are more number of techniques, finding the best set is not feasible manually. The process of selection is automated in order to minimize the execution time, compilation time, tuning time and normalized tuning time. The existing algorithms such as the Combined Elimination, Batch Elimination, Branch and Bound and Optimality Random Search are modified and the results are compared with the newly developed Push and Pop Optimal Search Algorithm. We have implemented the algorithms on both a Pentium IV machine and a Core 2 Duo machine, by measuring the performance of MiBench benchmark programs under a set of 65 GCC compiler options. It is found that the Push and Pop algorithm shows better performance when compared with other algorithms.

Key-Words: **-** Optimization; Execution Time; Orchestration Algorithms; GCC Compiler Options; Benchmark Applications.

1 Introduction and Related work

The behavior of the compiler is dependent on the program domain, the compiler setting and the system architecture. Therefore it can not be determined how a compiler responds to a program each time. GCC compiler provides three different levels of optimizations. The optimizations are to be applied upon the program without losing any of its features. Turning all the techniques on for an application potentially degrade the program performance and increases the compilation time. Every time when a recent version of the GCC compiler is released, it comes with more flags for optimization.

The Iterative Elimination [1] failed to predict the correct order of optimizations, but only the set of optimizations that are to be turned on are predicted. The concept of parallel programming can be used to test upon distributed memory architectures. The Milepost [6] is a compiler technology that can automatically learn how best it is to optimize programs for heterogeneous embedded processors using machine learning. The paper claims that it improves the execution time of MiBench benchmark by 11%. But Milepost GCC [3] uses only static program features for extracting the features of the given application. It uses GCC [18] , [19] as the compiler infrastructure for Milepost and it is not implemented on any of other architectures. Fine-grain run time adaptation for multiple program inputs on different multi-core architectures as proposed in are yet to be included in this architecture. The Branch and Bound algorithm served better than many of the existing algorithms but its impact is measured on overall performance and not on individual code segments. The process has to be repeated for each set of program since the program features are not extracted using any performance like PAPI which can provide more sophisticated functionality of user callbacks on counter overflow and event multiplexing. A Random Search Strategy can be implemented using the dynamic program features to extract the characteristics of an application as proposed in [6]. But the random set of techniques selected may not

be always the best and hence the results cannot be stated as the best. The Combined Elimination [4] strategy can be used to find the best set of techniques but it appears to be less effective when the process of finding the relative improvement percentage consumes many iterations. One another strategy proposed in called as the Batch Elimination [4] which eliminates the techniques one by one. But it does not test the combined effects of the optimization techniques and hence it cannot produce best results. J.Andrews et al [2],[7],[9] uses machine learning algorithm for selecting best set of techniques, but not used dynamic program features. Grigori Fursin *et al.* [10] have presented a novel probabilistic approach based on competition among pairs of optimizations (program reaction to optimizations) for enabling optimization knowledge reuse and for achieving the best possible iterative optimization performance. Malik Khan *et al.* [11] have presented a novel compiler framework for CUDA code generation. Qingping Wang *et al.* [12] have introduced methods for constructing policies to dynamically select the most appropriate TM algorithm based on static and dynamic information. John R. Wernsing et al.[13] have developed a framework for hybrid computers. Christophe Dubach et al. [14]. L. Almagor et al. [15] have evaluated how optimization sequences which affects optimization. Basilio B. Fraguela et al. [16] have developed an efficient algorithm for HTA programs. M. Burtscher et al. [17] have designed a tool for high performance application.

There arises a need to design a framework that could handle the new set of techniques that get added each time. Research has been going on for a long time on designing the framework for automation. The Batch Elimination algorithm immediately eliminates the techniques that give a negative effect. The problem with this algorithm is that the effects that these techniques have with other techniques upon combination with them could not be analyzed. The Optimality Random Search Strategy picks a set of predefined sequence for a program that has to be tested. But the challenge with this algorithm is that the algorithm requires predefined sequences for its operation. To equip the algorithm with the capabilities to handle any program that is provided as input became a non-trivial task. The evolution of the combined elimination strategy is helpful in finding a set of techniques that suits for any program. The Combined elimination eliminates one single technique at a time and so the need to speed up the process of elimination arises. So the advanced combined elimination eliminates two techniques at a single time. It is found that there could be better sets than the sets that resulted after the implementation of the advanced combined elimination algorithm. The set of outputs generated in each of the iterations of the advanced combined elimination gets stored into a stack along with the execution time required for the selected set. The set of techniques that has got the least execution time is popped out of the stack and it is selected as the resultant set. This proved to be efficient than any other technique and proved 20% more efficient than the closest alternative. The Push and Pop Optimality Search Algorithm is the most efficient technique and it has showed better speed up than any of the other algorithms tested for the MiBench Benchmark applications.

2 Detailed Framework

GCC provides a list of optimization techniques. Some techniques can improve the performance of the program while some of the techniques will degrade the performance of the program. The objective is to find the best set of techniques. The six algorithms serve the same purpose of selecting the optimization techniques. But the fast and efficient algorithm is to be preferred for finding the best set with minimal execution time, compilation time, tuning time and normalized tuning time. The best set of techniques selected by an algorithm can be used to compile a program with improved performance.

The GCC compiler can be used to compile the program with the best set of techniques that are selected. This ultimately improves the program's performance. The performance analyzer tool like PAPI can be used to analyze the features of the program. This helps us to predict the best sets for the programs with the similar features without consuming much of time. The feedback from the performance analyzer tool helps us to see whether any more iteration is to be made. The process repeats until a set of techniques which offers the minimum compilation time, execution time, tuning time and normalized time is found out. The process of finding the best set of techniques requires many iterations and it is a time consuming task. An algorithm which offers better performance for most of the programs tested is the reliable algorithm that we can be preferred.

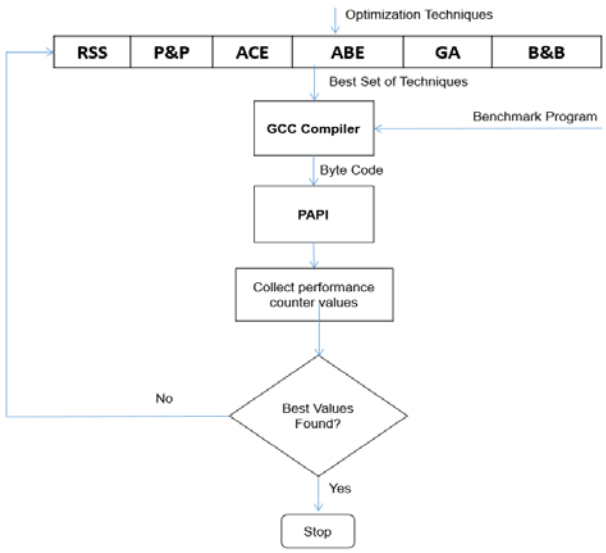

Fig.1 Automated Framework with PAPI Interface

The Figure 1 shows automated framework integrated with PAPI. PAPI [8], [12] is useful for collecting dynamic program features during run time. The recent research shows that program using dynamic program features works well than program using static program features. With the help of PAPI interface set of performance counter events collected with respect to system architecture. The performance counter event varies with respect to system architecture. Using PAPI query interface we can collect various events such as hardware interrupts, floating point instructions, level 1 cache related operations and level 2 cache related operations. PAPI implements the features in software where possible. Also, processors do not support the same metrics, thus we can monitor different events depending on the processor in use. Therefore, the interface remains constant, but how it is implemented can vary. In addition to raw counter access, PAPI provides sophisticated functionality of user callbacks on counter overflow and event multiplexing.

3 Proposed Algorithms

For finding best set of optimization techniques from 65 available optimization techniques the following strategies are applied. They are

1. Optimality Random search
2. Push and Pop algorithm
3. Advanced combined elimination
4. Advanced batch elimination
5. Genetic algorithm
6. Branch and bound algorithm.

3.1 Optimality random search

Input: n number of optimization techniques
Output: Best set of combination.
Steps:

1. Calculate T_b with F_0=1, F_i=1.. F_n=1.
2. Pick Sequence from S[n] where S[n]={F_x=0, F_y=1, F_z=0} and n=random(x).
3. Compile each every benchmark application with S[n] and run with large datasets to measure the execution time.
4. Calculate Speed Up = T_b/execution time.
5. Repeat step 2 to 4 until the best set is found.

This is similar to the algorithm proposed in [6]. The algorithm searches for the best set of techniques from the predefined sets of techniques which are randomly generated during the initial phase of execution of the algorithm. The set of techniques are sorted upon the basis of speed up and the list of techniques with the best speed up is selected as the best set.

3.2 Advanced combined elimination

Input: Set of "n" optimization techniques
Output: Best set of techniques (ON - OFF Combination)
1. Calculate Base Time, Tb = T(Fi=1, Fi+1=1, Fi+2=1,…,Fn-1=1, Fn=1).
2. For every benchmark application, select a sequence of techniques
 2.1. Initialize Seq = {F_1, F_2, ..., F_n}
 2.2. Compile and execute the application.
 2.3. Measure the RIP value of each set of two optimization options F_i in Seq relative to the base time T_b.
 2.4. If F_i=1 gives negative RIP, then set F_i=0.
3. Repeat steps 2 until all options in Seq have nonnegative RIPs.
4. When RIP (Fn >=0), that sequence Seq, represents the final solution.

This algorithm is the modified version of the combined elimination algorithm [4]. The RIP of each techniques are calculated and are sorted in the descending order. Each time, two most negative values are eliminated and the process of elimination continues until there exists any non-negative RIP values for any of the techniques. It was found that the best set of techniques found using this algorithm does not give the best performance all of the time.

3.3 Push and Pop elimination

Input: Set of "n" optimization techniques
Output: Best set of techniques
1. Set Seq = {F_1, F_2, ..., F_n} and Baseline = {F_1 = 1, F_2 = 1, ..., F_n = 1}.

2. Compile and execute the application under the baseline setting to get the base time T_B. Measure the RIP(Fi = 0) of each set of two optimization options Fi in S relative to the Baseline.
3. Push the sequence to an array with the set of techniques and their execution time.
4. Let Y = $\{y_1, y_2, ..., y_n\}$ be the set of optimization options with negative RIPs. Y contatains list of values stored in an ascending order. Remove y_1 from S and set y_1 to 0 in B. For i from 2 to n,
 4.1 Measure the RIP of y_i relative to the basetime B.
 4.2 If the RIP of y_i is negative, remove y_i from S.
 4.3 Set y_i to 0 in B.
5. Repeat Steps 2 and 3 until all options in S have nonnegative RIPs.
6. Sort the stored sequences in the ascending order of their execution time.
7. Pop out the best sequence which is the optimal solution.

The Push and Pop Algorithm is proposed in order to rectify the defects of the advanced combined elimination algorithm. In this proposed algorithm, the set of techniques which are generated in each iteration of the advanced combined elimination algorithm are stored into a stack along with their execution time. Finally, when the algorithm has finished its execution, the stack element, with the least execution time is popped out. This algorithm proved to be the best for all of the six programs tested except for the *dijkstra* program.

3.4 Advanced batch elimination

Input: Set of "n" optimization techniques
Output: Best set of techniques
1. Compile the application under the baseline B= {F1 = 1, F2 = 1, ..., Fn = 1}. Compile and execute the application to get the basetime T_B.
2. For each optimization Fi, switch it off from B and compile the application. Execute the generated version to get T(Fi = 0).
3. Set Fi=0, when execTime(Fi=1) > execTime(Fi=0).
4. Compile the program using the techniques with Fi=1 after the execution of steps 1 to 3 to generate the optimal result.

This algorithm is a modified version of the Batch Elimination Algorithm. The techniques that has a greater execution time when they are ON as compared to the execution time with those techniques turned OFF are eliminated.

3.4 Genetic algorithm

Input: Set of "n" optimization techniques
Output: Best set of techniques
1. Select a Chromosome sequence C1 ={fi=1, fi+1=1, fi+2=1, fi+3=1, fi+4=1}
2. Perform mutation: Generate x, where x<=4. Make fi+x=1, if fi+x=0 and fi+x=0, if fi+x=1.
3. Perform the 2nd level of mutation by repeating step 2.
4. Increment i=i+5. Repeat step 1 to generate Chromosome sequence C2.
5. Exchange chromosomes in both of the sequences tested where C3={C1. f0 , C1. f1 , C2. f0 , C2. f1, C2. f2}.
6. Perform this for all the 65 chromosomes available and store each chromosome sequence in an array.
7. Compile each and every benchmark application with C[n] and run with large datasets to measure the execution time (execTime).
8. Calculate Speed Up = Tb/execTime.
9. Repeat step 7 to 8 until the best set is found.

The Genetic Algorithm initially generates a chromosome sequence with five sets of chromosomes which are optimization techniques. Then a mutation is performed upon this chromosome sequence in order to turn one technique in the selected sequence OFF. A second mutation is also performed upon the selected sequence. The process is repeated with another set of chromosome. Then the crossover is done in order to combine the properties of both of the chromosome sequences. The chromosome sequences are tested upon the programs and they are sorted upon the basis of their speed ups. The set of chromosomes with the best speed up is the best set of techniques for the given program.

3.5 Branch and Bound Algorithm

Input: Set of "n" optimization techniques
Output: Best set of techniques
1. Initialize all falgs are on in set B.
2. Initialize relative improvement percentage upper bound.
3. Repeat the following until the stack is not empty.
 3.1. For each F_x in S, x=1 to n
 {Compute RIP (Fx=0)} Find the option(T) in S with most negative RIP.
 3.2 .Find the set of options(S') such that it gives, i.e. RIP<(60%of RIP_UB)
 3.3 .If S' set is empty then return;
 3.4. Else, {For all the elements in S'
 Set all the flags are OFF;}
 3.5. If RIP(F_T=0)<RIP_UB
 Then, RIP_UB=RIP(F_T=0)

4. Result contains RIP values contains positive value

Return

4 Experimental set up

We have conducted experiments using Intel core 2 duo T6600 processor with speed 2.2Ghz.With 3GB DDR2 RAM,L1 cache 128KB,L2 cache 2 MB,using ubuntu11.10 operating system,GCC compiler 4.5.2.Performance counter events collected using PAPI library. The following Table1 shows some of the important optimization techniques from GCC compiler [10]. All 65 optimization techniques which includes ON or OFF applied to each application.

Table 1 Levels 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	

4.1 Program used for experiments

The algorithms consists of atomic components such as arrays, structures etc. which are helpful in its implementation. The execution of a script file is also required at times in order to perform the execution of benchmark applications used in the algorithm and to find their execution time. The algorithms are tested upon six benchmark applications [5] which are as follows:

basicmath

The basic math test performs simple mathematical calculations that often do not have dedicated hardware support in embedded processors. This benchmark application is used for solving many mathematical computations such as square root calculations, function solving and angle conversions from degrees to radians.. The input data is a fixed set of constants.

bitcount

The bit count algorithm tests the bit manipulation abilities of a processor by counting the number of bits in an array of integers. It does this using five methods including an optimized 1-bit per loop counter, recursive bit count by nibbles, non-recursive bit count by nibbles using a table look-up, non recursive bit count by bytes using a table look-up and shift and count bits. The input data is an array of integers with equal numbers of 1's and 0's.

susan

Susan is an image recognition package. It was developed for recognizing corners and edges in Magnetic Resonance Images of the brain. It is typical of a real world program that would be employed for a vision based quality assurance application. It can smooth an image and has adjustments for threshold, brightness, and spatial control. The small input data is a black and white image of a rectangle while the large input data is a complex picture.

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. Dijkstra's algorithm is a well known solution to the shortest path problem and completes in $O(n^2)$ time.

Patricia

A Patricia tree is a data structure used in place of full trees with very sparse leaf nodes. 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 hour period.

String search

This benchmark searches for given words in phrases using a case insensitive comparison algorithm.

4.2 Script File Format

gcc -static -o3 program_name.c -o output;

This is used to compile the C program with all the optimization flags ON and to generate an output file called 'output'.

time -f \"%e\" -o output.log ./output

This is used to store the time required to execute the output file to an external file called output.log.

Compilation and Execution Procedure

The algorithms were implemented in C++ and so they can be compiled using the G++ compiler.

g++ program_name.cc

Compiles the C++ program and generates an output file called as 'a.out'.

./a.out

Runs the program and prints the output of the program onto the screen.

gcc -static -o3 basicmath_small.c rad2deg.c cubic.c isqrt.c -o output –lm

A Benchmark Application can be compiled using the GCC compiler using the above command.

. /output

The program can be run by specifying the name of the output file.

4.3 Performance Metrics

There are a number of factors that we consider in order to access the performance of the programs which are listed as follows:

Relative Improvement Percentage (RIP)

It's calculated using the following formula

$$\text{RIP } (F_i=0) = \frac{T(Fi=0) - Tb}{Tb} \text{ X } 100\% \qquad (1)$$

Where,

- Tb is the Base Time with all the program compiled with the highest level of optimization.
- $T(F_i=0)$ is the execution time with the particular technique turned OFF.

It is upon the basis of the Relative Improvement Percentage that many of the algorithms such as the Advanced Combined Elimination and Push and Pop Combined Elimination work.

Tuning Time

It is the time required to run an application with the best set of techniques selected by an algorithm.

Normalized Tuning Time

It is calculated using the following formula:

$$\frac{\text{Tuning Time}}{\text{Compilation Time} + (3 \times \text{Execution Time})} \qquad (2)$$

Speed up is the ratio between base time and time required to fine tune the code. Base time is measured by compiling and running the applications by initializing all flags are ON.

5 Results and Discussions

Table.2 Number of Iterations required to obtain best sets of values

Number of Iterations						
Benchmark	ORSA	ACE	P&P	ABE	GA	B&B
basic_math	55	22	23	65	24	25
bit_cnt	55	15	16	65	18	22
dijkstra	55	14	15	65	25	27
patricia	55	21	22	65	21	31
stringsearch	55	12	13	65	18	19
susan	55	18	19	65	24	26

Table.3 Tuning time for different algorithms with respect to speed up

Benchmark	ORSA	ACE	P&P	ABE	GA	B&B
basic_math	1.42	1.39	1.32	1.42	1.39	1.39
bit_cnt	0.293	0.288	0.221	0.313	0.301	0.299
dijkstra	0.14	0.17	0.15	0.16	0.15	0.16
patricia	0.522	0.412	0.401	0.422	0.422	0.482
stringsearch	0.027	0.032	0.025	0.032	0.028	0.031
susan	0.219	0.119	0.109	0.129	0.116	0.118

Table.4 Maximum Speed up between different algorithms

Benchmark	ORSA	ACE	P&P	ABE	GA	B&B
basic_math	1.0347	1.0058	1.0469	1.0275	1.006	1.006
bit_cnt	1	1.0174	1.3258	1.0683	1.0273	1.0204
dijkstra	1.357	1.1176	1.2667	1.875	1.2667	1.175
patricia	1.0524	1.024	1.23597	1	1	1.1422
stringsearch	1	1.08	1.185	1.185	1.037	1.1481
susan	0.5433	1	1.0917	1.084	1.0259	1.0085

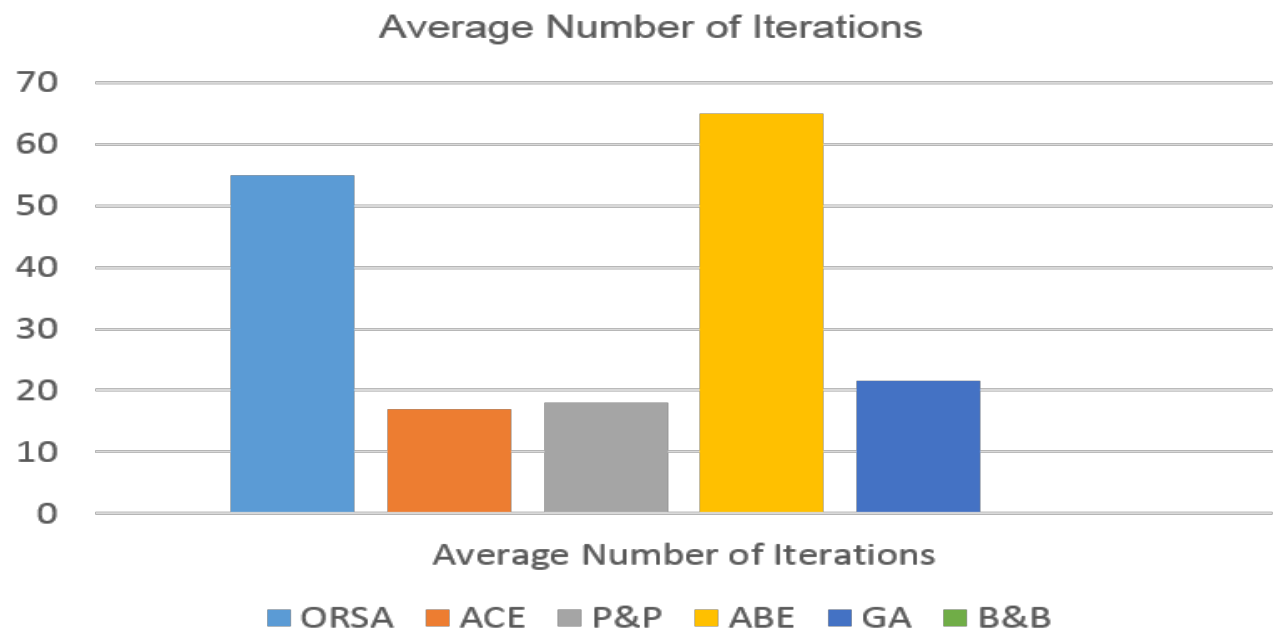

Fig. 2 Number of iterations required to find an optimal set

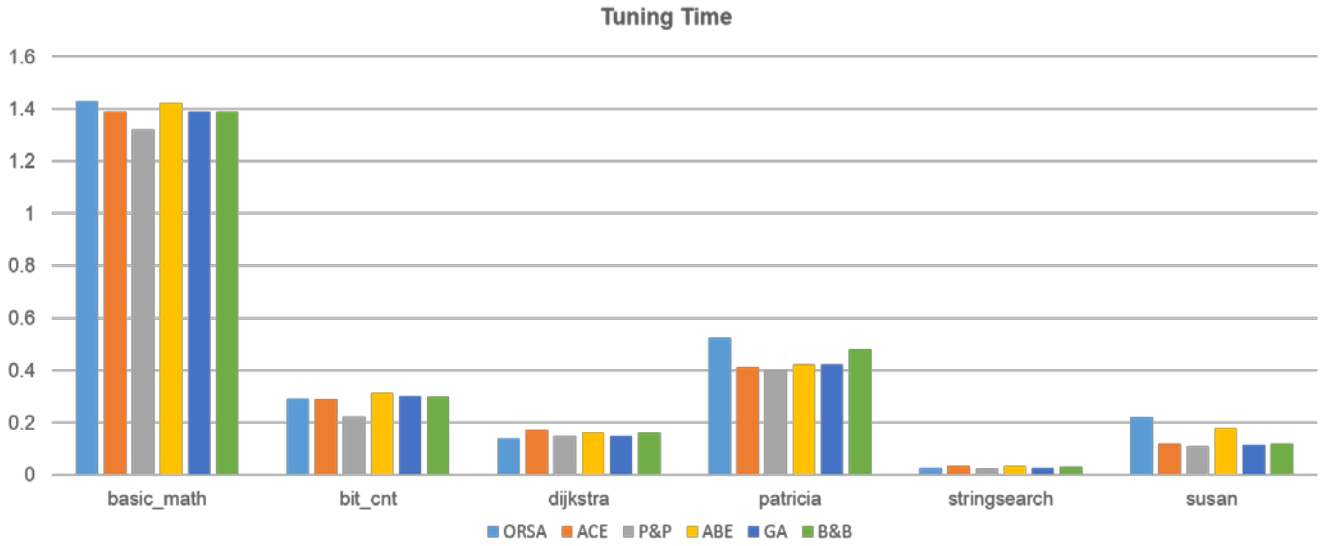

Fig.3 Comparison of different Tuning time

Table.5 Comparison of Normalized tuning time

	ORSA	ACE	P&P	ABE	GA	B&B
basic_math	0.2094	0.2048	0.2016	0.2089	0.2054	0.2051
bit_cnt	0.0898	0.0891	0.0724	0.0942	0.0909	0.0917
dijkstra	0.1286	0.1467	0.1329	0.1417	0.1305	0.1417
patricia	0.1791	0.1582	0.1557	0.1608	0.1712	0.1608
stringsearch	0.0194	0.0228	0.0181	0.0227	0.0201	0.0221
susan	0.0496	0.0286	0.0264	0.0308	0.028	0.0285

The automated framework from Fig.1 is implemented. The framework which accommodates different optimization selection algorithms. All the selection algorithms are explained in section 3. The selection algorithm selects the best combination of optimization sequences. The best combination is compiled with gcc compiler for a benchmark application. The various parameters such as compilation time and execution time are measured. During run time various performance counter values are collected using PAPI [1], [20] library. The above steps are repeated until best combination is found which reduces compilation time and execution time. Table 1 shows set of important optimization techniques used in this research. Table 2 lists numbers of iterations are required to find an optimal set. While testing the benchmark applications with large data sets, it is observed that ORSA required 55 iterations since it has to check all the random sequences generated for the programs that are tested. BE also requires 65 iterations since it checks all the available sets of techniques in order to analyze which techniques should be turned ON and which are to be turned OFF. ACE, B&B and GA has got varying values for the number of iterations depending on the sequences that are generated. P&P algorithm iterates one step more than CE since it has to check all of the sequences that are generated by the CE algorithm.Figure 2 shows number of compa -risons required to find an optimal set for every optimization algorithm. Table 2 lists tuning time of different optimization algorithms with respect to speed up. Tuning time is measured once an optimal set is found. Push and pop elimination gives better tuning for most of the bench mark applications that are tested except for the *dijkstra* program. For the program *dijkstra*, optimality random search algorithm gives the better results. This could not be always the best solution, because the sequences generated by this algorithm are randomly generated and they are not reliable.
Push and pop eliminations improves tuning time by 20%. Figure 3 shows comparison of tuning time for every benchmark applications.

The Normalized Tuning Time is a very efficient metric since the execution time at three different times is taken into consideration for its computation. Normalized tuning time is measured using equation 2. We Normalize the tuning time because execution time for different bench mark applications are not same. Table 5 lists normalized tuning time for different benchmark applications.

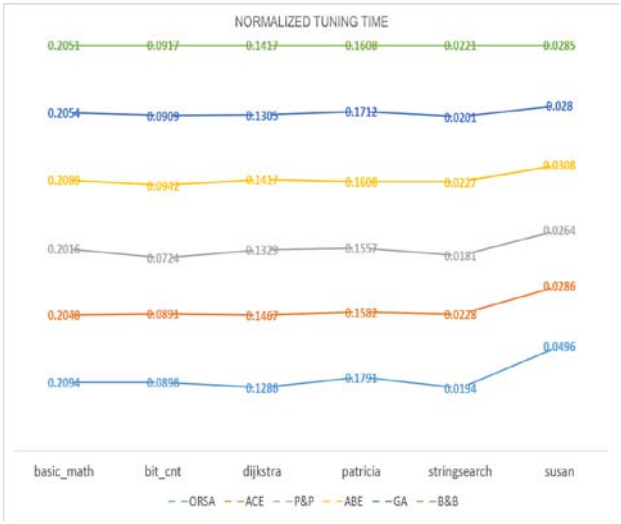

Fig.4 Comparison of normalized tuning time

This graph and results also proves that the Push and Pop Combined Elimination Algorithm is the reliable algorithm. Figure 4 shows comparison of normalized tuning time.

6 Conclusion

The algorithms tested with six different benchmark applications providing large data sets show better performance for Push and Pop Algorithm. It is a modified version of the Combined Elimination [7]. Each program is run three times in order to reach the conclusion. The Push and Pop algorithm shows consistent performance for all of the benchmark programs. The Push and Pop algorithm can be used to improve the programming performance of any program and to pick the right set of optimization techniques. The best set that is picked by the Push and Pop algorithm can be used to enhance the execution time, compilation time, tuning time, normalized tuning time and speed up. Another important fact is that the results could be found in a few numbers of iterations. Performance analyzer tools can be used to study static and dynamic program features and incorporating it with the algorithm can be helpful in finding the results even faster. This research covers an analysis and study of all the optimization techniques available in the GCC Compiler v2.6 [18]. Applying various algorithms upon six different benchmark applications run with large data sets give the conclusion that the Push and Pop algorithm is better than all other algorithms.

Future research can be done upon this by enhancing the existing framework by combining static and dynamic program features. In future the framework can also be extended by including more benchmark applications with various optimization algorithms. Other open source compilers such as LLVM, ROSE, Open Path, etc. can also be considered.

References

[1] Eunjung Park, Sameer Kulkarni and John Cavazos, An Evaluation of Different Modeling Techniques for Iterative Compilation, Volume:2012 Publisher: Proceeding CASES'11 Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems, Pages 65-74.

[2] J.Andrews,Performance enhancement of tuning time in gcc compiler optimizations using benchmark applications,CiiT International journal of Artificial Intelligent Systems and Machine Learning,Vol.4,No.5,2012,pp.276-281.

[3] Grigori et al .Milepost GCC :machine learning enabled self tuning compiler, International Journal on parallel programming,vol.39,2011,pp.296-327.

[4] Z.Pan and R.Eigenmann,Fast and effective orchestration of compiler optimization for automatic performance tuning, In proceedings of the international symposium on code generation and optimization,2006,pp.381-387.

[5] Mathew R.Guthaus et al.,Mibench:A free commercially representative embedded benchmark suite,2001,pp.3-14

[6] Ayal Zaks, Bilha Mendelson, Chris Williams, Cupertino Miranda, Edwin Bonilla, Elad Yom-Tov, Elton Ashton, Eric Courtois, François Bodin, Grigori Fursin, Hugh Leather, John Thomson, Michael O'Boyle, Mircea Namolaru, Olivier Temam, Phil Barnard, MILEPOST GCC: Machine Learning Based Research Compiler,2008.

[7] J.Andrews, Dr.T.Sasikala, Enhancement of orchestration algorithms for compiler optimization, Communications in computer and Information science,Springer,Vol.No.269,2011,pp.617-627.

[8] Agakov, Boyle, Cavazos, Edwin V, Felix, Grigori, John Fursin, Michael F P O', Olivier, Temam, Rapidly Selecting Good Compiler Optimizations using Performance Counters, International symposium on code generation and optimization, 2007,pp.185-197.

[9] J.Andrews, Dr.T.Sasikala, Evaluation of various compiler optimization techniques related to mibench benchmark applications,Vol.9,Issue.6,2013,pp.749-756.

[10] Grigori Fursin, Olivier Temam and Inria Saclay, "Collective Optimization: A Practical Collaborative Approach", ACM Transactions on Architecture and Code Optimization, Vol. 7, No. 4, 2010, pp. 1-29.

[11] Malik Khan, Protonu Basu, Gabe Rudy, Mary Hall, Chun Chen and Jacqueline Chame, "A Script-Based Autotuning Compiler System to Generate High-Performance CUDA Code", ACM Transactions on Architecture and Code Optimization, Vol. 9, No. 4, 2013, pp. 1-25.

[12] Qingping Wang, Sameer Kulkarni, Michael Spear and John Cavazos, "A Transactional Memory with Automatic Performance Tuning", ACM Transactions on Architecture and Code Optimization

[13] John R. Wernsing and Greg Stitt, Vol. 8, No. 4, 2012, pp. 1-23., "Elastic computing: A portable optimization framework for hybrid computers", Elsevier journal on Parallel Computing, 2012, pp. 438–464.

[14] Christophe Dubach, Timothy M. Jones and Michael F. P. O'boyle," Exploring and Predicting the Effects of Micro architectural Parameters and Compiler Optimizations on Performance and Energy", ACM Transactions on Embedded Computing Systems, Vol. 11S, No. 1, 2012, pp. 1-24.

[15] L. Almagor, K. D. Cooper, A. Grosul, T. J. Harvey, S. W. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Finding effective compilation sequences. In Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems, 2004.

[16] Basilio B. Fraguela, Ganesh Bikshandi, Jia Guo, María J. Garzarán, David Padua and Christoph von Praun, "Optimization techniques for efficient HTA programs", Elsevier journal on Parallel Computing , 2012, pp. 465–484.

[17] M. Burtscher, B.-D. Kim, J. Diamond, J. McCalpin, L. Koesterke, and J. Browne, "Perfexpert: An Easy-to-Use Performance Diagnosis Tool for Hpc Applications," Proc. ACM/IEEE Int'l Conf. High Performance Computing, Networking, Storage and Analysis (SC '10), 2010, pp. 1-11.

[18] GCC Manual available at http://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/

[19]Optimization in GCC, Online Linux Journal
which can be read from
http://www.linuxjournal.com/article/7269
[20] A portable interface to hardware
Performance counters
http://icl.cs.utk.edu/papi