Enhancement of Phase order Searching using an Effective Tuning Strategy

Dr.J.ANDREWS
Faculty of Computing
Sathyabama University, Chennai-600119, India andrews_593@yahoo.com

Abstract:- Modern compilers provide a large number of compiler options. Each option has two states called as enable or disable. Enabling on some options may improve or degrade the performance of the program. The objective is to increase the performance of the source program by means of adjusting the compiler options. The selection and ordering of the most efficient compiler options is required to improve the execution time, code and speed up. Some option combination may affect the execution time. The problem of getting the best combination of options is found by using modified genetic algorithm with the help of genetic operators. Finding the better ordering of best options will change the final performance of the program. Ordering of the best combination of options obtained from selection algorithm is fed to phase order search algorithm. The existing algorithms such as combined elimination, batch elimination, branch and bound, push and pop with combined elimination algorithm, optimality random search and iterative random search algorithm are modified and the results are compared with the newly created combined push and pop with modified genetic algorithm. It is found that the combined push and pop with modified genetic algorithm shows better performance when compared to other algorithms. The phase searching algorithm shows increase in the program performance for some benchmark applications than combined push and pop with modified genetic algorithm. The experimental results show that 9% improvement in both tuning time and normalized tuning time. The speedup exhibit 11% increase over the set of benchmark applications. The combination of combined push and pop with modified genetic algorithm and phase order searching provide a 10% overall improvement in the program performance from the existing algorithms.

Index Terms—Optimization, Combined Push and pop with modified Genetic Algorithm, Phase order searching

1 Introduction and related work

Recent compiler has many levels of optimization techniques. Each level contains a set of compiler options for optimizing the source program. Each option has two states enable or disable. If enabling some options may tends to improve the program performance or degrade the program performance. Some options may even change the entire meaning of the program code which in turn affects the expected output. The main goal is to find which option to be enabled or disabled. For example the function in-lining can make a program faster but over use of in-lining function affect the final performance. Recent version of the modern compiler [1, 2] release comes with more flags for optimization. It is unpredictable that how a compiler behaves on applying these optimizations. This work deliberately involves evaluating various benchmark applications with different optimization levels to check how it impacts the program performance. The main aim is to find the best set of optimization techniques provided by GCC compiler with a better order for a given application. Ordering is the process of changing position of the compiler option and achieves better program performance. For example constant propagation replaces some operands by constants, making some variables become unused. A dead code elimination pass may then remove these variables [3, 4] but only if applied after the constant propagation pass.

In recent years various algorithms have been proposed to solve this problem. The existing algorithms like Batch elimination strategy [5, 6, 7] eliminate the negative technique one at a time. This is failed to estimate the combined effects of the techniques used. Branch and bound algorithm also provides better output but the effects on individual code segments are ignored. It only considers overall performance obtained. Combined Elimination [8, 9] out performs all the existing orchestration algorithms giving the best result out of all. But it has high tuning time. It also needs more number of iterations to find out the negative relative improvement percentage given by each technique.

Bashkansky et al. [10] proposed a framework (ESTO) to obtain suboptimal compilations by the help of genetic algorithm. Considering iterative elimination strategy, it helps to predict the best set of techniques to be turned on. However the correct order of these best set of techniques is not predicted. Optimality random search algorithm helps to find out the best set of sequences in random. It is not optimal at all -time which proves that it is not the best strategy being used. The push and pop combined elimination [5] selects the best set of techniques based on the relative improvement percentage. It is similar to combined elimination strategy but it overcomes the problem of eliminating single technique at a time. The existing framework failed to give both static and dynamic program features. Almagor.L [11] et al considers sequences that affect optimization by examining a small fraction of compilation order. Z Pan et al provide an effective orchestration strategy to get the best performance considering only static program features. Grigori Fursin [12, 13, 14] et al presented a novel approach for knowledge reuse to obtain best optimization sequence. J.Andrews [5] et al proposed a push and pop algorithm that eliminate negative relative improvement percentage values sequences to enhance the best set of sequence but it takes more execution time. The proposed combined push and pop with modified genetic algorithm proved 10% efficient than any other considered alternative. It gives the best performance for many benchmark applications with better speed up.

1.1 System Architecture

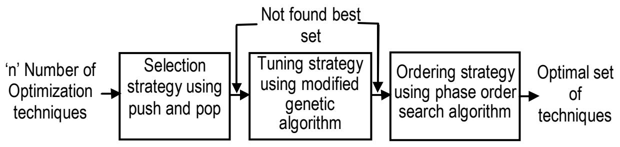
'n' Number of Optimization techniques
Selection strategy using push and pop
Not found best set
Tuning strategy using modified genetic algorithm
Ordering strategy using phase order search algorithm
Optimal set of techniques

Fig.1 System Architecture Diagram

Modern compiler provides the users a large number of options [1]. The problem is how to find the best combination of optimization options provided by compiler for a given source program. This gives raise the requirements for selecting the best set of compiler optimizations for a program has been an open problem in compilation research for decades. Selecting the set of techniques that optimize the code is done by the selection algorithms. Once the selection is made tuning operation is carried out to fine tune all the available techniques to obtain best speed up. Tuning time is also measured in this process. The algorithms which provides lesser tuning time is taken into consideration. The process is repeated until best tuned optimal set is obtained. The tuning process is done by MGA. The tuned optimal set is then ordered and compiled for a given application to check the impact of the ordering of those techniques in a sequence. Phase order search algorithm is used to order the tuned techniques. The optimal set of techniques is obtained from phase order search which is greater than or equal to the current speed up.

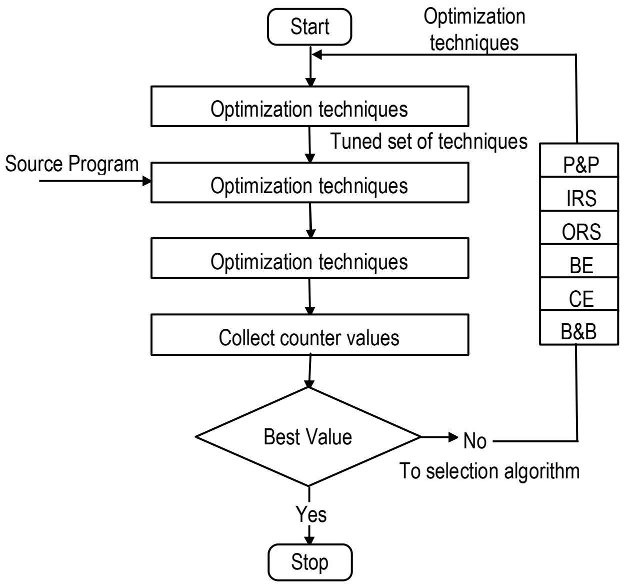

Fig.2 Selection frame work with PAPI Interface

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, code size, 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 were selected. This will ultimately improve the program's performance. The performance analyzer tool like PAPI [15] can be used to find out the performance counter values. This will help us to predict the best sets for the programs with the similar features without consuming much of time. The feedback from the PAPI tool will help 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. That is the best set of techniques is generated. The process of finding the best set of techniques requires more iteration. An algorithm which gives better performance for most of the programs tested is the reliable algorithm that can preferred for the optimization.

1.2 Proposed Algorithms

- Posh and Pop with Combined Elimination
- Combined Push and Pop with Modified Genetic Algorithm.
- Phase Order Searching Algorithm

1. ***Push and pop with combined elimination***

- Input: n number of optimization techniques
- Output: best set of techniques Steps:

1. Set base time, $B_t = F_0=1$, $F_1=1$, $F_2 = 1$, …, $F_n=1$ and sequence $S = \{F_1, F_2, ..., Fn\}$.
2. Calculate the RIP value of each set of two optimization options Fi in S with respect to the base time B_t
3. Store the sequence to an array with the set of techniques and their execution time.
4. $R = \{R_1, R_2, ...,R_n\}$ be the set of optimization options with negative rips. R contains list of values stored in an ascending order. Remove R_1 from s and set $R_1$to 0 in R.
5. Repeat the following from 1 to n-1. 5.1) Measure the RIP of R_i relative to the base time Bt. 5.2) If the RIP R_i is negative then remove Ri from S. 5.3) Set R_i to 0 in R.
6. Repeat steps 2 and 3 until all options in sequence s have non negative RIPs.
7. Sort out the stored sequences in the ascending with the help of execution time.
8. Display the best set of sequences.

For a given set of 'n' optimization techniques create a baseline and calculate base time using that baseline. Calculate relative improvement percentage for each option using the obtained base time. Push each option in the sequence to an array with the execution time. Remove the options that produce negative relative improvement percentage in the array. The best sequence is obtained after all the options are pushed and verified in the array.

2. ***Combined Push and Pop with Modified Genetic Algorithm***

- Input : Individual chromosomes
- Output: Best set of chromosomes

Steps:

```
1. Initialize the population randomly
2. Compatibility function
   if check(Chromosomes) then
   For all Gi=1 € chromosome && Gj=1 € chromosome do compatible=true;
       Pij=0
   Construct P_ij matrix where P_ij is i'th gene incompatible with j'th gene
       End for Else if
   Find the pair of genes which causes incompatibility
3. Incompatibility function
   For all (Gii,Gj) € chromosome do For all j!=i ^ Pij>0 do
   if(Gi,Gj==error)
   Pij=0;
       End for
   For all Pij with i<=j do If (Pij==0) then
   Incompatible=true Gij=0 || 1
   end if
   end if
4. Fitness function Speedup=Tb/exe time
   F=min(Speed up)
5. Picking the best chromosome with respect to the fitness function value.
6. Maintain the population size.
7. Remove (worst chromosome)
8. Add(new chromosome)
   8.1 Mutation(Gij)
   Gij=0 || 1
   8.2 Crossover(Gij,G(i+1)(j+1))
   Mix the two chromosomes
9. Repeat the process for 100 iterations to find the best set of chromosome.
```

The solution for finding the best set of technique is stored in the form of 0's and 1's called as chromosome. In Combined Push and Pop with Modified Genetic Algorithm the chromosome of each individual genes (G) represents the compiler option. Every gene (G_i) has two possible values (0 or 1). When the option has 1 in the matrix, then it is enabled for the process otherwise it is taken as disable option (0). Each chromosome consists of N (65 options) genes. Recently some incompatibilities between the compiler options are reported by certain compilers. Compatible genes not produce any error during their execution. But the incompatible genes produce error in the execution time. Detection of these incompatible genes in the compiler options may reduce the search space and increase the efficiency of the program. So there is a need for an efficient algorithm to detect these incompatible genes. Since most of the executions are successful, a large number of iteration is need for the detection of whole incompatible genes. In MGA the Compatible Function and Incompatible Function are made only for the selected set of chromosome. Fitness function value is taken the highest speed up from the large number of chromosomes. This fitness function values is differentiate the chromosomes from best set or worst set of chromosomes. The genetic algorithms performance is largely influenced by the genetic operators such as crossover and mutation.

1.3 *Phase Order Search Algorithm*

- Input : Best set of combination
- Output: Phase ordered set of sequences

Steps:

```
1. Select the best set of techniques using selection
   algorithm $P = \{S_1, S_2, ..., S_n\}$.
2. Measure the speedup for each and every
   techniques selected and eliminate sequences
   which have Speed up <1.
3. Randomly select the best speed up sequence for
   phase ordering.
4. Repeat the following until get the best
   sequence.
   4.1 Select the position of the sequence Si[x]
       where x=random().
   4.2 Swap the technique Si[x] with Si[x+1].
   4.3 Display the sequence with Execution time
       and Speedup.
```

Phase order searching is the process of changing position of the compiler option and achieves better program performance for example Constant propagation replaces some operands by constants, making some variables become unused. A dead code elimination pass may then remove these variables, but only if applied after the constant propagation pass .Then chromosomes are stored in the forms of 0's and 1's.The output from the combined push and pop with Modified Genetic algorithm is given as the input to the phase order search algorithm. This will change the position of the options in the sequence.

2. Experimental Setup

The experiment is conducted in Intel(R) Core(TM) i3 2.13GHZ CPU With 4GB DDR2 RAM,L1 cache 64KB,L2 cache 256KB,L3 cache 3MB using ubuntu12.10 operating system, GCC compiler 4.7.2. The list of performance counter values collected for every benchmark applications [16] using PAPI interface.

Benchmark Application Tested

basicmath: The basic math application process simple mathematical calculations for embedded processors that do not have dedicated hardware support . For example, integer square root, cubic function solving, and degrees to radians angle conversions are used for calculating road speed or vector values. The input data is a set of constant values.

bitcount: The bit count algorithm process the bit manipulation capacity of a processor by calculating the number of bits in different strategies. The input is an array of integer values .It contains nine sub-algorithms. Each algorithms output is the number of bits in input which is 1.The recursive and data decomposition techniques are used to develop the parallel algorithms such as Bitcnt 1, Bitcnts ,and Bitstring.

qsort: The qsort application uses the famous quick sort algorithm for sorting a large number of strings into ascending order. The small data set contains a list of words and the large data set contains set of data with three tuples representing points.

susan: Susan is stands for Smallest Univalve Segment Assimilating Nucleus .It is an image recognition package. It is used in the brain for identifying the corners and edges in Magnetic Resonance Images, vision-based quality assurance, and performs adjustments for brightness, threshold, spatial control, and image smoothness. The small input data contains a black white image of a rectangle but the large input data contains a complex picture.

dijkstra: Dijkstra algorithm is an effective way to find the shortest path problems. The Dijkstra benchmark calculates single source and all pairs shortest paths in an adjacency matrix representation. The single source shortest path has two parallelized strategies such as single and multiple queue implementations. The all pair shortest path problem uses the data decomposition strategy.

FFT/IFFT: FFT benchmark stands for Fast Fourier Transform and IFFT for inverse transform. FFT and IFFT are manipulating the array of input data. These are used in to find the frequencies in digital signal processing. The input data contains polynomial function with frequency sinusoidal components and pseudo random amplitude.

CRC32: CRC is stands for Cyclic Redundancy Check (CRC). This benchmark is used only for the 32 bit data on the file. CRC used to detect errors in data transmission process. The input data is the sound files from the Adaptive Differential Pulse Code Modulation benchmark.

3 Performance Metrics

Tuning Time: It's the time required to run an application selected by an algorithm to find the best set of techniques

Normalized Tuning Time: Normalized Tuning Time is calculated using the following formula:

Normalized Tuning Time =

$$\frac{\text{Tuning time}}{\text{Compilation time} + (3 \times \text{execution time})} \quad (1)$$

Relative Improvement Percentage (RIP)

RIP value is calculated using the following formula

$$\text{RIP}(F_i=0) = \frac{T(F_i=0) - T_b}{T_b} \times 100\% \quad (2)$$

Where,

- ❖ T_b is the base time, time required by an application for executing with o3 level of optimization.
- ❖ $T(F_i=0)$ is the execution time with that particular technique is disabled

Speed Up

- ❖ It is the parameter that helps to find whether the set of optimization is better than the highest level of optimization.
- ❖ It can be determined using the formula:

$$\textbf{Speed Up} = \frac{\text{base time}}{\text{time taken to fine tuning the application}} \quad (3)$$

Cache Miss Rate

Cache miss rate is calculated using the formula

$$\text{Cache miss rate} = 1 - \frac{\text{no.of cache hits}}{\text{total cache references}}$$

4 Results and Discussion

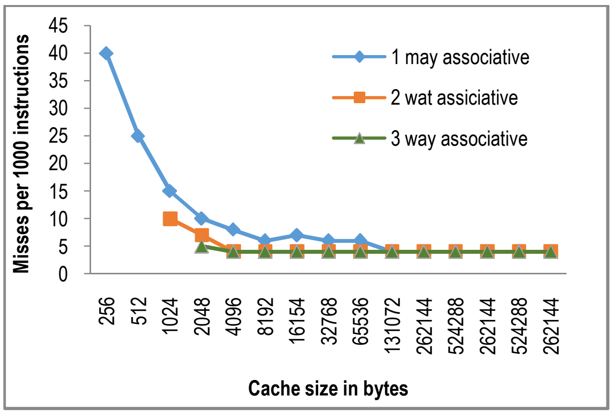

Fig 3 Graph showing the cache miss rate per 1000 instructions for susan application.

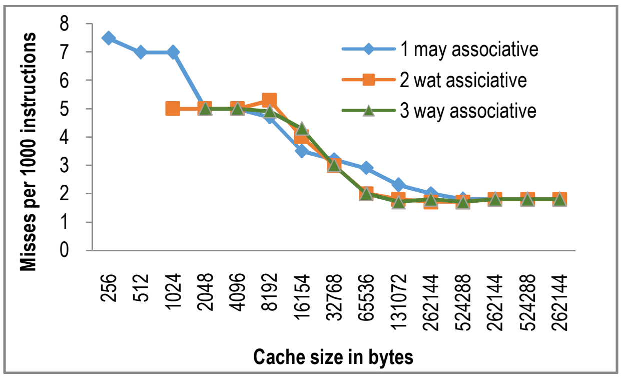

Fig.4 Graph showing cache miss rate per 1000 instructions for dijkstra application

A cache miss rate is defined as the failure attempt to read or write data in the cache. The Figure 2 and 3 shows the cache miss rate per 1000 instruction for the benchmark application dijkstra and susan. This cache miss rate is directly depending on the set of optimization sequences used in the program. Cache miss rate can be reduced by the help of optimal set of techniques.

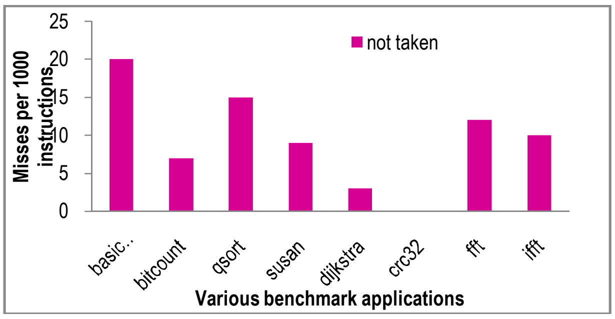

Fig.5 Graph showing branch prediction rates per 1000 instructions for various benchmark applications

Figure 5 shows branch prediction rate for the various benchmark application. It shows that the basicmath having highest level of branch prediction rate obtained by the not-taken prediction scheme. Some benchmark having large prediction rate due randomness of data. But some have few prediction rates due to large number of integer arithmetic logical operations.

Table 1 Tuning time taken by each algorithm to run benchmark applications in seconds

Benchmark	MGA	ODRS	IRS	BE	B&B	CD	P&P CE
SUSAN	2	3	2.36	2.6	2.56	2.68	2.3
BITCOUNT	1.12	2.12	1.59	1.6	1.55	2.1	1.5
DIJKSTRA	0.36	0.55	0.46	0.42	0.43	0.452	0.4
FFT	8	10.1	9.2	8.6	9.11	9	8.5
IFFT	4	5.6	4.36	5.07	4.465	5	4.33
QSORT	10.1	11.16	10.53	10.38	10.5	11	10.3
CRC32	1.1	1.58	1.3	1.4	1.36	1.46	1.2
BASICMATH	1.63	1.9	1.82	1.96	1.81	1.88	1.78

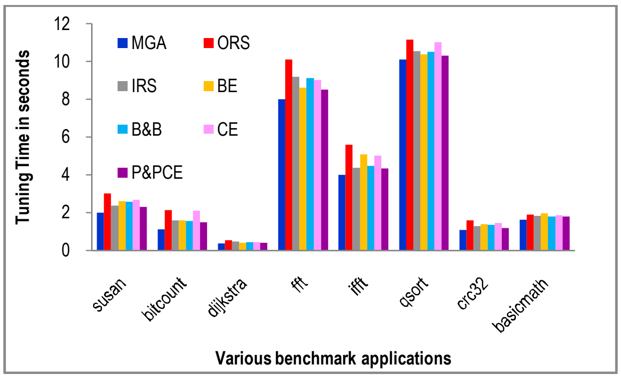

Fig.6 Graph showing the Tuning Time taken by algorithms for each benchmark applications.

From the Table 1 and Figure 6, it can be proved that MGA takes the minimum tuning time for every benchmark application. The MGA also is capable of giving the best set of techniques which requires only less number of iterations than any other algorithm, which makes it more efficient than the other algorithms. Although, P&P CE gives close tuning time compared to MGA, it requires more time to eliminate the negative RIP. This value makes the MGA is more efficient. MGA show a 9% decrease in tuning time than other algorithms.

Table 2 Normalized Tuning Time taken by each algorithm to run benchmark applications in seconds

Benchmark	MGA	ORS	IRS	BE	B&B	CE	P&P CE
SUSAN	4.12	5.21	4.62	4.62	5.01	5.26	4.6
BITCOUNT	1.93	3.47	2.65	2.71	2.62	3.38	2.34
DIJKSTRA	0.78	1.1	0.865	0.833	0.914	0.957	0.893
FFT	1.07	1.357	1.19	1.25	1.19	1.93	1.08
IFFT	0.77	0.996	0.844	0.954	0.822	0.918	0.783
QSORT	0.809	0.9	0.838	0.825	0.826	0.856	0.8135
CRC32	1.119	1.548	1.262	1.359	1.402	1.431	1.212
BASICMATH	1.193	1.37	1.325	1.427	1.315	1.36	1.295

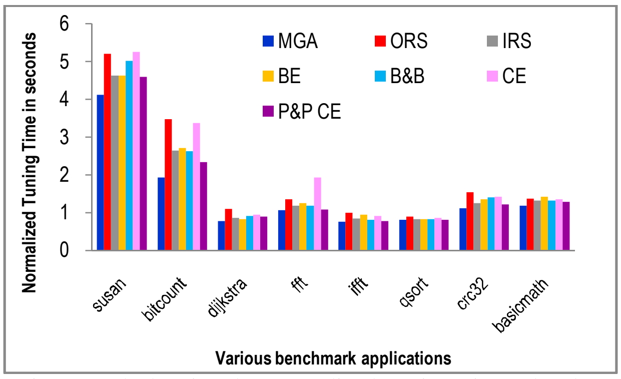

Fig.7 Graph showing the Normalized Tuning Time taken by algorithms for each benchmark applications

The normalized tuning time is calculated using the tuning and the execution time to run the application. From the Table 2 and Figure 7, it is proved that the MGA take less normalized tuning time than the existing algorithms, that is almost 9% overall difference .

Table 3 Speed Up obtained from each algorithm while running benchmark applications in seconds

Benchmark	MGA	ORS	IRS	BE	B&B	Ce	P&P CE	POS
SUSAN	1.33	1.14	1.07	.67	.29	1	1.26	1.33
BITCOUNT	1.03	0.67	1	0.79	1	0.8	0.9	1.04
DIJKSTRA	1.03	1.01	1	1.00	0.67	0.78	1	1.03
FFT	1.37	1	1.03	1.13	1	1.25	1.13	1.37
IFFT	1.26	1	1.12	1.11	1	1	1.24	1.37
QSORT	1.28	1.18	1.2	1	1.01	1	1.19	1.2
CRC32	1.05	1.04	1.04	1	1.03	1.02	1.05	1.05
BASICMATH	1.03	1	1.00	1	1.02	1	1.02	1.03

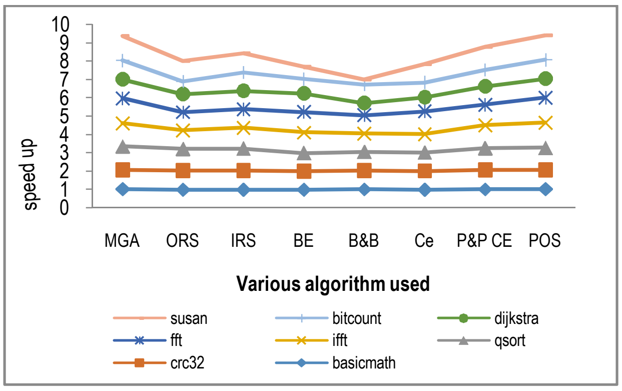

Fig.8 Graph showing the speed up values for algorithms running each benchmark applications.

From Table 3 and Figure 8 it has been inferred that MGA provides better speed up values for all the benchmark applications The POS algorithm however gives same or better speed up values after ordering is done to the best set obtained from MGA. These values are nearly close to the values of MGA. However speed up for CRC application while running in POS gives less speed up value than MGA which explains that ordering degraded the performance of the program but only in a small ratio which can be neglected. The combination of MGA and phase order searching algorithm achieve 11% improvement from the existing algorithms.

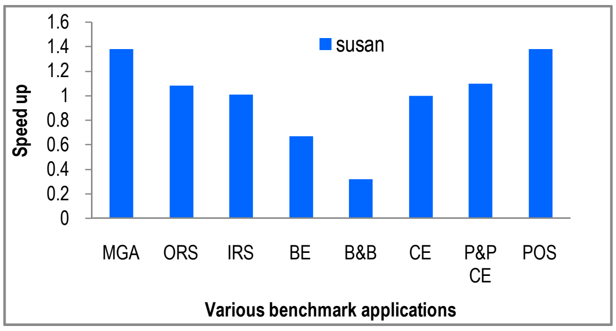

Fig. 9 Graph representing the speed up value provided by each algorithm for Susan application.

From Figure 9 it is proved clearly that the speed up obtained by the MGA and POS are higher or relatively equivalent to the other algorithms specified. MGA improves SUSAN benchmark up to 9.3% better speed up than P&P CE. POS maintains the speed up after the selection strategy.

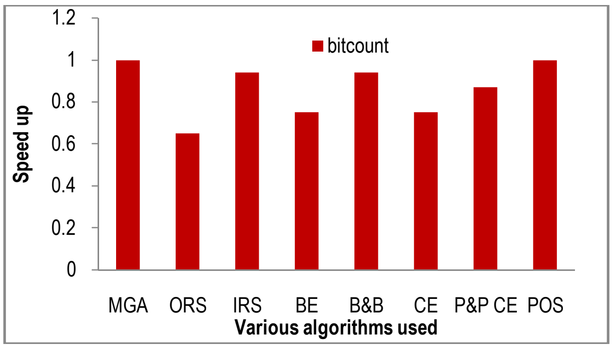

Fig. 10 Graph representing the speed up value provided by each algorithm for Bit Count application

From the Figure 9 MGA provides up to 9 % better speed up than iterative random search and push & pop combined elimination and even better speed up of up to 9.7% for other algorithms for the bitcount benchmark. Phase order search algorithm also gives better speed up of up to 9.7% after ordering applied.

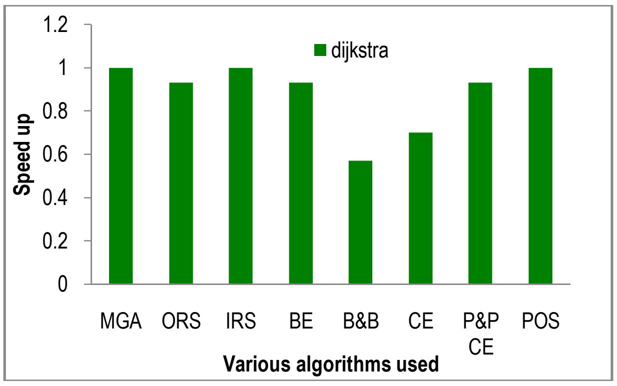

Fig. 11 Graph representing the speed up value provided by each algorithm for Dijkstra application

Figure 10 shows speed up obtained for the Dijkstra application with large data sets. The combined push and pop with modified genetic algorithm provides 2% higher speed up than batch elimination. But MGA improves more than 2% speed up when compared to other algorithms such as optimality random search, iterative random search and push and pop combined elimination. Phase Order Search algorithm provides almost equivalent speed up after ordering MGA.

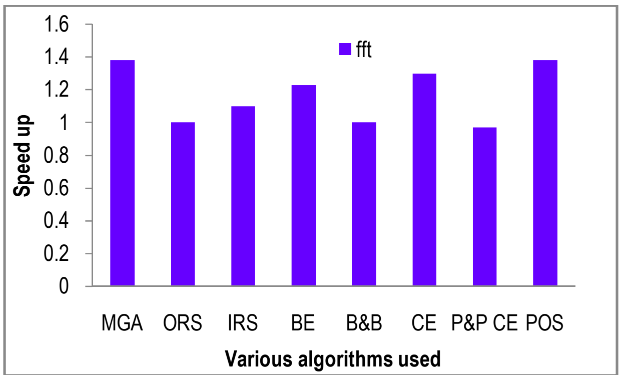

Fig.12 Graph representing the speed up value provided by each algorithm for FFT application.

From Figure 12 it has been inferred that MGA gives 9.1% better speed up than combined elimination and even much better output than other algorithms. Phase order search algorithm gives almost the same output after ordering from MGA.

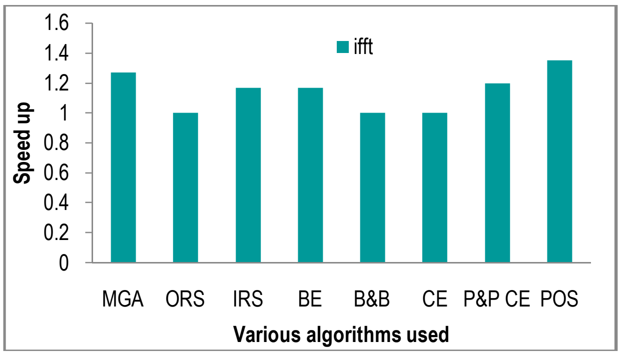

Fig.13 Graph representing the speed up value provided by each algorithm for IFFT application

Figure 13 shows that IFFT application tested with large data sets. MGA provides better speed up than almost all other algorithms. MGA gives 9.8% better speed up than push and pop combined elimination and 8.9% better speed up than iterative random search algorithm. It also gives 8% better speed up than batch elimination and optimality random search algorithm. Phase order search improvises the speed up of up to 5% after ordering from MGA.

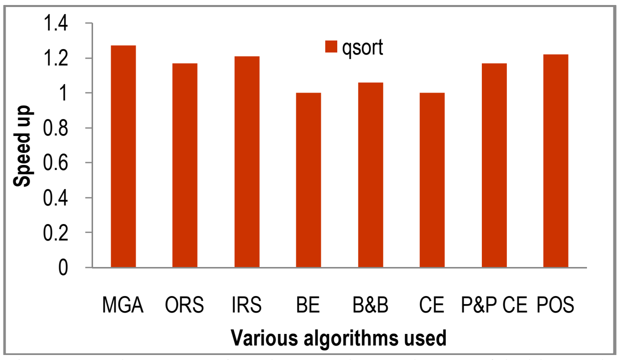

Fig.14 Graph representing the speed up value provided by each algorithm for Quick Sort application.

Figure 14 shows that MGA gives 9.27% better speed up than push and pop combined elimination and 9.38% better speed up than iterative random search algorithm. Phase order search does not provide better result for quick sort application as it provides 3% lesser speed up after ordering from MGA.

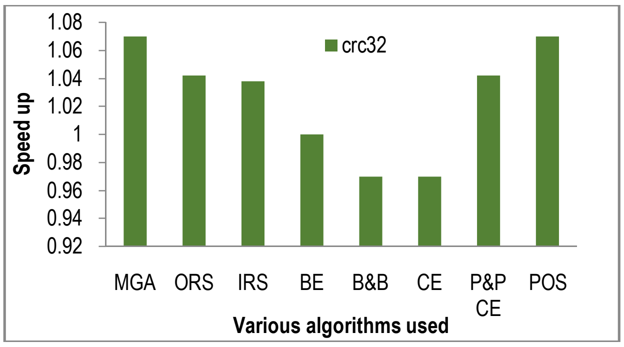

Fig.15 Graph representing the speed up value provided by each algorithm for CRC32 application

From Figure 15 it has been inferred that MGA gives 2% better speed up than push and pop combined elimination and 3% better speed up than optimality random search algorithm. Phase order search algorithm almost the same speed up after ordering from MGA.

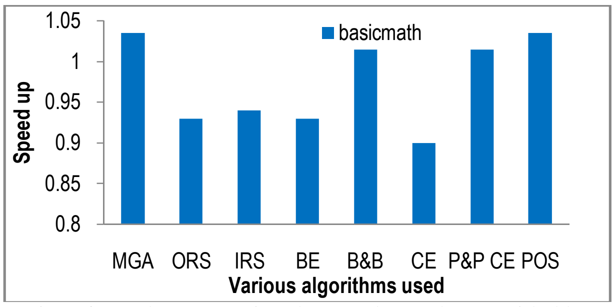

Fig. 16 Graph representing the speed up value provided by each algorithm for Basic Math application

Figure 16 shows that basic math application tested with large data sets for all other algorithms. MGA gives 9.3% better speed up than push and pop combined elimination and 11.3% better speed up than iterative random search algorithm.

The MGA provides better set of optimization techniques than the existing strategies. The MGA achieve overall 11 % improvement in the speed up. Phase order search algorithm gives the same output after ordering from MGA. From the above figures it has been inferred that overall 2 % to 12 % improvement for the speed up has been achieved if ordering performed from MGA.

5 Conclusion

This paper deals with the evaluation of different selection strategy, tuning strategy and phase ordering strategy is integrated in one framework. These different strategies are applied to eight different benchmark applications [16] with large data sets considered. The result proves that better performance is achieved for Combined Push and Pop with Modified Genetic Algorithm with phase order searching. The MGA algorithm proves 10% consistent performance for all of the benchmark programs. The MGA algorithm can be used further to fine tune the program performance which is intend select best set of optimization techniques. Applying various algorithms upon eight different benchmark applications run three time with large data sets concluded that MGA is the best strategy. The resultant sequences obtained from MGA are given to the phase order searching algorithm. The phase order searching algorithm gives 3 % better speed up than the selection algorithms considered. The experimental results show that 9% improvement in both tuning time and normalized tuning time. The speedup exhibit 11% increase over the set of benchmark applications. The combination of MGA and Phase order searching proves that 10% overall improvement in the program performance when compared to other existing algorithms.

This research work will opens several possibility to the future work related to this study of compiler options. In future apply additional machine learning algorithms to develop a prediction model [17] that will predict the good optimization combination. In this framework the algorithms are implemented only for GCC compiler and uses only dynamic program features. In future the framework can also be extended to other application domains such as Clang [18], ICC [19], ROSE Compiler [20] and Open 64 [21]. In future we consider both static and dynamic program features with more benchmark applications.

REFERENCES

[1] GCC Manual available at http://gcc.gnu.org/online docs/gcc-4.7.2/gcc.

[2] Optimization in GCC, Online Linux Journal which can be read from http://www.linux journal.com/article/7269.

[3] Michael R.Jantz, Prasad A.Kulkarni, "Exploiting phase inter-dependencies for faster iterative compiler optimization phase order searches," CASES '13 Proceedings of the 2013 International

Conference on Compilers, Architectures and Synthesis for Embedded Systems, Article No. 7, Year 2013, pp. 1-10.

[4] David Whalley, Gary S.Tyson and Prasad A.Kulkarni," Evaluating Heuristic Optimization Phase Order Search Algorithms," International Symposium on Code Generation and Optimization(07), Year 2007,pp. 157 – 169.

[5] Andrews J and Dr.Sasikala T "Efficient framework architecture for improved tuning time and normalized tuning time," WSEAS transactions on information science and applications, Issue 7, Vol. 10, July 2013, pp.230-240.

[6] Retantyo Wardoyo and Suprapto ,"Algorithms of the combination of compiler optimization options for automatic performance tuning," International Conference, ICT- EurAsia 2013, Vol. 7804,Year 2013 ,pp. 91-100.

[7] Rudolf Eigenmann and Zhelong Pan," Fast and Effective Orchestration of Compiler Optimizations for Automatic Performance Tuning," International Symposium on Code Generation and Optimization (06), Year 2006, pp.319-332.

[8] Andrews J and Dr.Sasikala T, " Evaluation of various compiler optimization techniques related to Mi-bench benchmark applications," Journal of Computer Science, Vol. 9 Issue 6, Year 2013, pp.749-756.

[9] Andrews J and Dr.Sasikala T, "Analysis of Mi-bench Benchmark Applications Using GCC Compiler," 3rd International Conference on Computer Science and Information Technology (ICCSIT'2013) ,Year 2013, pp. 34-37.

[10] G. Bashkansky and Y. Yaari, "Black box approach for selecting optimization options using budget-limited genetic algorithms, "SMART Workshop 2007 (HiPEAC). European Network of Excellence on High Performance and Embedded Architecture and Compilation, Ghent, Belgium,Year 2007, pp. 1–16.

[11] Almagor L, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon and Todd Waterman," Finding Effective Compilation Sequences," ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems(LCTES '04),Vol. 39 Issue 7, Year July 2004 ,pp. 231 – 239.

[12] Grigori Fursin, Olivier Temam and Inria Saclay, "Collective Optimization: A Practical Collaborative Approach", ACM Transactions on Architecture and Code Optimization, Vol. 7, Year 2010, pp. 1-29.

[13] Grigori Fursin, Olivier Temam, Mircea Namolaru, Elad Yom-Tov and Ayal Zaks et al. "*MILEPOST GCC: machine learning based research compiler,"* GCC Developers' Summit, Ottawa, Canada, Year 2008, pp. 1-13.

[14] Edwin Bonilla , Felix Agakov , Grigori Fursin, John Cavazos, , Michael F.P., O'Boyle and Olivier Temam," Rapidly Selecting Good Compiler Optimizations using Performance Counters," In CGO '07: Proceedings of the International Symposium on Code Generation and Optimization ,Year 2007, pp. 185-197.

[15] PAPI: A Portable Interface to Hardware Performance Counters. http://icl.cs.utk.edu/papi.

[16] Jeffrey S. Ringenberg and Matthew R. Guthaus et al.:"MiBench: A free, commercially representative embedded benchmark suite "*IEEE 4th Annual Workshop on Workload Characterization, Year 2001.*

[17] Eunjung Park, Sameer Kulkarni and John Cavazos, " Evaluation of different modeling techniques for iterative compilation," CASES 2011 ACM ,Year 2011 ,pp. 65-74.

[18] CLang, A C Language Family Frontend for LLVM,(http://clang.llvm.org/)

[19] ICC, Intel C++ Compiler,(http://software.intel.com/)

[20] Rose compilers at http://rosecompiler.org/

[21] Open 64 compilers at http://www.open64.net/home.html