Optimizing Code by Selecting Compiler Flags using Parallel Genetic Algorithm on Multicore CPUs

T Satish Kumar#1, S Sakthivel#2, Sushil Kumar S#3

#1Research Scholar, , Department of Information and Communication Engineering, Anna University, Chennai, Tamilnadu, India

1satish.savvy@gmail.com

#2Professor, Department of Computer Science and Engineering, Sona College of Technology, Salem, Tamilnadu, India

2sakthits@rediffmail.com

#3Research Scholar, Department of Computer Science and Engineering, RNS Institute of Technology, Bangalore, Karnataka, India

3sushil.rnsit@gmail.com

Abstract - The compiler optimization phase ordering not only possesses challenges to compiler developer but also for multithreaded programmer to enhance the performance of Multicore systems. Many compilers have numerous optimization techniques which are applied in predetermined ordering. These ordering of optimization techniques may not always give an optimal code further it is impossible to find a unanimous optimization phase ordering that will produce the best code. Finding an optimal sequence for even a simple code is not easy task since the search space for attempting optimization phase sequences is very huge. The focus of this paper is to implement an optimal compiler phase ordering using parallel genetic algorithm exploiting the advances of parallel programming on Multicore processors. Inherently Genetic algorithms are very well suited for modeling multiple individuals. Three methods were adopted in testing and comparing the performance of the Parallel Benchmark programs, firstly the Benchmarks were compiled without applying optimization flags, next the Benchmarks were compiled by randomly selecting the optimization flags and finally a parallel genetic algorithm was used to set the Compiler optimization flags for compilation. The parallel genetic algorithm outperformed the other two methods in this work.

Keywords: Multicore, Phase Ordering, compiler Optimization, Parallel Genetic Algorithm, Performance**.**

I. INTRODUCTION

In the age of Multicore, as the cores increase aggressive optimization of application code should be evident to enhance its performance. Modern compilers support many different optimization phases and these phases should analyze the code and should produce semantically equivalent performance enhanced code. The three vital parameters defining enhancement of the performance are time, space and energy.

The finally produced code after applying optimization for various applications may be different for different sequence of optimization phases. The variation in optimization phase ordering depends on the application that is compiled, the architecture of the machine on which it runs and the compiler implementation. Many compilers allow optimization flags to be set by the users. The GNU compiler collection (GCC) is a compiler system which provides collection of optimization flags (O, O1, O2, O3, Os, Og, Ofast). Several optimization flags can generate optimized code for several applications but for further enhancing the throughput of Multicore processors these optimization phases are to run parallel on Multicore systems. It is widely accepted that there cannot be a single ordering of optimization phase that produces an optimal code for any application [1, 2, 3, 4, 5, 6]. Multicore systems run code parallel providing very high performance at lower power consumption; therefore each core can handle multiple subtasks for computation.

A straight forward solution to the phase ordering problem is the Exhaustive search method. This solution is infeasible because of the magnitude of the search space. Further, the searching is made more complex when the different optimization ordering gives a unique performance gain every time. The frame work used in this work uses GCC 4.8 Compiler and 12 distinct optimization options shown in table 2. The maximum number of optimization sequence length of n would result in 12^n possible combination sequences. Studies [7] made earlier suggests that not all optimization phase ordering will deliver optimized code.

The Genetic Algorithm (GA) is a adoptive algorithm based on theory of evolution. This algorithm is successfully applied for phase ordering of compiler optimization by several researcher [4, 3, 8, 9]. The approach used in the current work in determining the optimization phase ordering is the application of Parallel Genetic Algorithm (PGA) and by obtaining dynamic feedback information to compute fitness values. Search space was not as huge as Exhaustive search method but did take several hours to find optimization sequence for individual parallel Benchmark programs.

The remainder of the paper is structured as follows. First we discuss the other methods that are existing in determining the optimization phase ordering which are used to tune the application on single core. Second we give an overview of the framework comprising of PGA and the system in which experiments were performed. Third, we describe the PGA used in the frame work that is used to take decisions for better prediction of optimization phases. Fourth, we show the results that proves our techniques performs faster search in predicting better optimization phase ordering. Finally we present the future work and the conclusions of the paper.

II. RELATED WORKS

Compiler Optimization phase ordering is a long standing issue and is quite obligatory for compiler designer to pay more attention towards optimization and to better the performance of the code after the arrival to Multicore processor technology. It is widely accepted that compiler optimization not only needs coding skills but also knowledge about hardware, architecture and other software tools. Many compilers today support hundreds of optimization flags for enhancing the performance of the application run on Multicore machines. These optimization flags not only improve the performance but also should maintain the semantics of the original code. With these many optimizations available researchers have attempted to predict several techniques for phase ordering the compiler flags. Soffa and Whitfield had developed a frame work based on axiomatic specification techniques which included precondition and post conditions that existed before and after applying optimization [10,11]. Kulkarni et al., had developed a framework called VISTA, a interactive compilation system for finding the best sequences of optimization phases.

To further fine tune the code the users get static and dynamic information about the performance. It automatically selected the best optimization sequence among the group of techniques [12]. Most importantly the work had genetic algorithm implemented to search effective sequencing of optimization phases. It considered optimizing of speed and space for the code. Optimization-Space exploration (OSE) was first iterative compilation systems which supported phase ordering for general purpose production compiler [13]. OSE not only used restricted heuristics but brought down the exploration space to a very small extent by using static performance estimators. From the previous configuration tried the system used feedback technique to prune the optimization configuration in the search space.

III. EXPERIMENTAL FRAME WORK

The work in this research uses GCC 4.8 compiler on Ubuntu 12.04 with OpenMP 3.0 library. The Fig. 1 depicts the overview of the Compiler Optimization tuning system. The source program is compiled by setting Compiler optimization flags that is chosen by the PGA. After the compiler, compiles the code and generates the executable file it is sent to the performance evaluation center which in turn evaluates the code and sends the statistics to the PGA, based on which the PGA will generated the consecutive Optimization phases.

The flow chart in the Fig. 2 shows the working of PGA. The PGA used a master-slave model. The algorithm starts by creating an initial population, which is randomly selected and evaluated using fitness function before assigning it to the slave cores. The initial population contains eight chromosomes. The slave core applies Selection, Crossover, Mutation and Replacement operations on this population and generates the new offspring. The old population is replacement with this new offspring which sets a new Compiler Optimization flag list as a collection of the genes in the chromosomes.

These newly set flag options are tested by compiling the code and statistical analysis of code performance is done. The generation is not allowed to cross two hundred, once it reaches this limit, the iteration is stopped for that application.

PGA can run on any number of cores. If it is run on a single core machine one thread is exclusively assign for the master and remaining threads are created to support the slaves of the PGA. This creates a bottle neck in the system by slowing down the overall system performance. Considering this reason the experiment is run on the system with AMD FX 8120 8-core processor with 4 GB DDR RAM. Out of 8-core one acts as a master node and the remaining core support the slaves in the system, depending on how many is required.

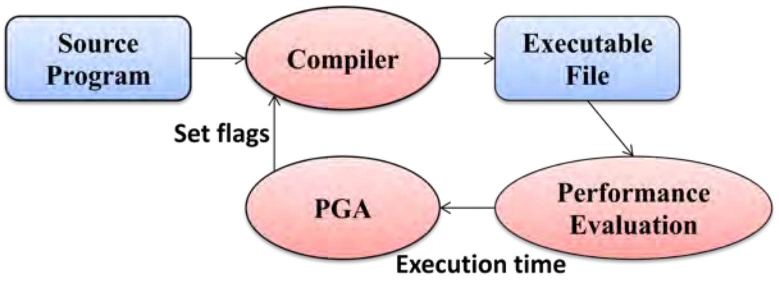

Fig. 1: Overview of the Compiler Optimization tuning System

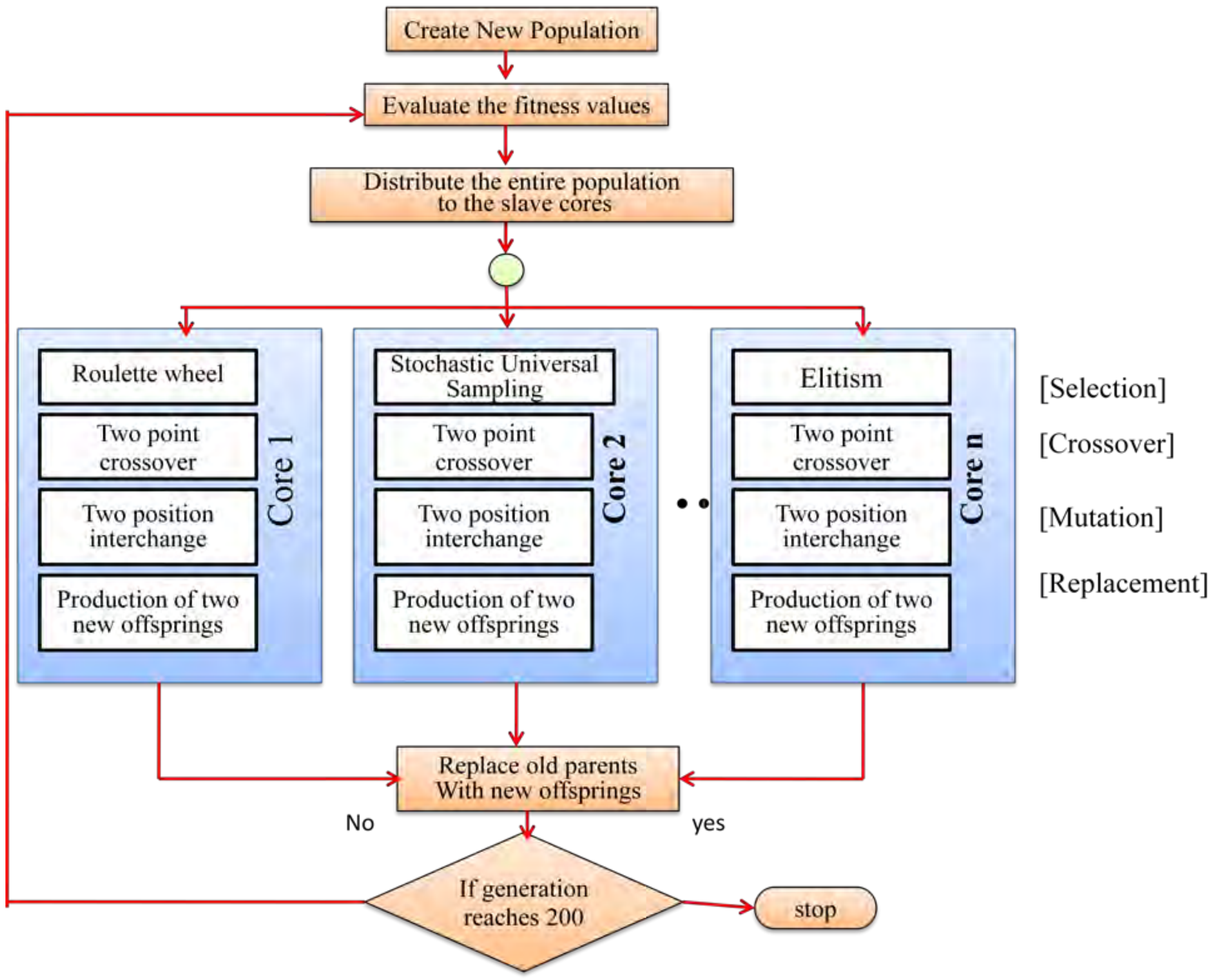
Create New Population
Evaluate the fitness values
Distribute the entire population to the slave cores
Roulette wheel
Two point crossover
Two position interchange
Production of two new offsprings
Core 1
Stochastic Universal Sampling
Two point crossover
Two position interchange
Production of two new offsprings
Core 2
Elitism
Two point crossover
Two position interchange
Production of two new offsprings
Core n
[Selection]
[Crossover]
[Mutation]
[Replacement]
Replace old parents With new offsprings
No
yes
If generation reaches 200
stop

Fig. 2: The Flow chart representing the Master-Slave Parallel Genetic Algorithm

Initially deciding on the Compiler optimization flags to be considered in the experiment that are supported by GCC was a time consuming task. Especially, optimization flag like register assignment can be applied only once. The optimization flags selected for the experiment in the current work can be applied more than once. The PGA finds the best optimization flags that can be applied between phases. The Table 2 lists the twelve optimization flags which are considered as distinct gene in the chromosomes. The first digit in the Sl no. of TABLE 2 represents, to which optimization class does the flag belongs: O1, O2 or O3.

Table 1: Parallel Benchmarks used in the Experiment

Benchmark	Description
LUD	LU decomposition method factors a matrix as the product of a lower and upper triangular matrix.
MM	Multiplies two Matrices.
QS	Sorts Array of elements using Quick sort technique.
PA	Prism's algorithm is used to find the minimum spanning tree for a connected weighted undirected graph.
DES	Data Encryption Standard algorithm for encryption of data.

The Parallel bench marks used in the experiment are listed in Table1. It begins with LU decomposition (LUD) which factors a matrix into the product of a lower triangular matrix and an upper triangular matrix. Matrix Multiplication (MM) is used to multiply two matrices. Quick sort (QS) sorts the elements. Prim's Algorithm (PA) is used to find the minimum spanning tree for a connected undirected graph. Data Encryption Standard (DES) algorithm is used for encrypting data. All the Bench mark programs are parallelized using OpenMP library to reap the benefits of PGA.

TABLE 2: Candidate optimization phases in Parallel Genetic Algorithm

Sl no:	flags	Description
11	-ftree-dce	Performs dead code elimination (DCE) on SSA trees(GCC 4.x only).
12	-ftree-dse	Performs dead store elimination (DSE) on SSA trees (GCC 4.x only).
13	-fcprop-registers	Attempts to reduce the number of register copy operations performed.
14	-fdelayed-branch	Utilizes instruction slots available after delayed branch instructions.
15	-fguess-branch-probability	Uses a randomized predictor to guess branch possibilities.
16	-floop-optimize	Applies several loop-specific optimizations.
17	-fomit-frame-pointer	Omits storing function frame pointers in a register. Only activated on systems where this does not interfere with debugging.
21	-fcse-follow-jumps	Follows jumps whose targets are not otherwise reached.
22	-fcse-skip-blocks	Follows jumps that conditionally skip code blocks.
23	-fgcse	Perform a global common subexpression elimination pass.
24	-fpeephole2	Performs machine-specific peephole optimizations.
31	-funswitch-loops	Moves branches with loop invariant conditions out of loops

IV. PARALLEL GENETIC ALGORITHM

A. *Parallel Genetic algorithm for finding the optimizing Sequence.*

Genetic Algorithm (GA) is applied to the problem that can be formulated as function optimization programs. GA and heuristic searching algorithms are based on the evolutionary ideas of natural selection and hereditary coined by Charles Darwin to reproduces populations over successive generation. GAs operate on entire population of points, this improves the chance of reaching the global optimum and avoids local stationary point. In nature, evolution is highly parallel process. The parallel Genetic Algorithm will not only take the fullest advantage of Multicores on the system but it also reduces the solution time by adding computational power. The choice of the Parallel Genetic Algorithm (PGA) is made to explore the quantities of robustness (basically independent of problem), reaches multiple local minima and simultaneously yielding better results without parallel hardware. Compared to sequential GAs, PGAs is highly efficient. The PGA used in this work is the Master-Slave model (MSM). The MSM executes the main loop of the PGA and allows the Slaves to perform Selection, Crossover and Mutations. In the master-slave model the master runs the evolutionary algorithm, controls the slaves and distributes the work. The Slaves take batches of individuals from the master and evaluate them. Finally send the calculated fitness value back to master. The Pseudo code of the described Parallel genetic algorithm is presented below.

The twelve Optimization flags shown in TABLE 2are encoded as twelve genes in the chromosomes as shown in the Fig. 3.

1	0	1	1	1	0	0	1	0	0	1	1

Fig. 3: Genes in the chromosome representing Compiler Optimization flags.

For evaluating the species the PGA uses the fitness function. The fitness function computes a fitness value for each chromosomes in the population. Crafting a fitness function is not easy task especially because it has a tremendous effect on the new generation. In the proposed system the PGA works with a population of six Chromosomes on eight core machine. The fitness function is computed at the Master core. The difference between the execution time of the code with optimization and execution time without optimization is used as the fitness function.

$$\begin{cases} F(exec\_with\_flag_i - exec\_with\_out\_flag_i) > 0 \\ i \,\varepsilon\, \{1,2,3, .. ,12\} \end{cases} \quad (1)$$

Where exec_with_flag$_i$ indicates ith flag set and exec_with_out_flag$_i$ indicates ith flag is not set. The function evaluates the difference in timing after compiling a benchmark with and without applying the flag.

Algorithm for Master node

```
Step 1: The master-node initially generates a random population and successively collects the new population
        from the
        slave nodes
Step 2: Slave nodes Algorithm takes over
Step 3: The master node evaluates all the individuals and again chooses the best population.
        go to Step 2 until 200 generations.
Step 4: Stop.
```

Algorithm for Slave Nodes (Every Slave node Executes these steps)

```
Step 1: Receives all the chromosomes from the master node with the fitness values.
Step 2 : The slave cores apply the roulette wheel, Stochastic Universal Sampling and Elitism methods
        respectively for
        selection process in parallel.
Step 3: Create next generation applying two point crossover.
Step 4: Applies mutations using method, two position interchange.
Step 5: Produce two new offspring/chromosomes.
Step 6: Sends both the chromosomes back to the master-node. (The master collects chromosomes from all
        slaves.)
```

B. *Details of Parallel Genetic Algorithm Operators*

1. Selection

The selection methods adopted it the slave cores are Roulette-Wheel selection (RWS), Stochastic Universal selection (SUS) and Elitism selection (ES). In RWS the individuals of the populations are assigned slots on the Roulette-Wheel proportional to their fitness values. The random candidates are selected where the pointer stops in the Roulette-Wheel. The probability of the member selected from the population is given by:

$$p_i = f_i / \sum_{j=1}^{m} f_j \quad (2)$$

where m is the population size and f_i represents the fitness of the individual.

The second method used for selection process is the SUS. This selection is allowed to choose optimization flags by not allowing the best fitness chromosomes always, thus showing the characteristic of zero bias with minimal spread. This method ensures that the weakest chromosome will also get a chance to be selected for mutation.

The third selection method is Elitism in which the best fit individual are selected and used as new offspring. The method used in this work uses a unique technique where one of the chromosome selected by Elitism is retained without applying mutation operation and carrying forward as the offspring in the new generation but allows mutation on the other chromosome in the pair of newly generated offspring. This ensures that the new generation does not always contain hundred percent new offspring which may totally divert the entire population to be unfit.

2. Crossover

Cross over us a process of considering two parent and producing the offspring form them. The purpose of crossover is to create better offspring from the mating pool. Out of several methods the current work adopts circular two point cross over to balance between complete diverse offspring generation and retaining the parent to the new generation.

Parent 1	**1**	**1**	**0**	**0**	**1**	**0**	**1**	**1**	**0**	**0**	**1**	**1**
Parent 2	**1**	**0**	**1**	**0**	**1**	**1**	**1**	**0**	**1**	**1**	**1**	**1**

L C R

Child 1	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**1**	**0**	**0**	**1**	**1**
Child 2	**1**	**1**	**0**	**0**	**1**	**1**	**1**	**0**	**1**	**1**	**1**	**1**

L C R

Fig. 4: circular two point crossover.

The Fig. 4 shows the over view of two point crossover. In the first generation the Left (L) part of the chromosome is allowed to swap while generating the children. During the second generation the centre (C) part of the chromosome is allowed to swap while generating the children. Finally in the third generation the Right (R) part of the chromosome is allowed to swap while generating the children. This process will continue in a circular fashion until the final generation is reached.

3. Mutation

The last operation is the mutation where two random genes are interchanged living out the just crossover part. The Fig. 5 shows the mutation of a child after first generation where the L part is untouched, the 6th bit is swapped with the 11th bit. The swapping will take place between the bits of a random nibble (four bits) with a bit in another nibble.

Child 1	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**1**	**0**	**0**	**1**	**1**
Child 1	**1**	**0**	**1**	**0**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**1**

Fig. 5: Mutation of the offspring

V. RESULTS

The results of the work on the five Parallel Benchmarks are given in this section. As one sees the Figures in this section, the results after applying PGA (WGAO) presents a major improvement with respect to the random optimization (WRO) and compiling code without applying optimization (WOO). As expected, PGA behaves very satisfactorily, the best solution is obtained because of three selection methods and picking the best solution out of these methods.

The Figures (6,7,8,9,10) represents the performance graphs of LUD, MM, QS, DES and PRIMS respectively. The X-axis represents time and Y-axis represents number of inputs to the parallel benchmarks. Out of twelve curves in each graph, three represents core-1 performances, consecutive three represents core-2, core-4 and core-8 performances respectively. From these graphs it is clearly evident that the PGA betters random choice and compilation without optimization for all the Benchmark cases. Figures (11, 12, 13, 14, 15) shows the performance enhancement graphs for the same Benchmark programs. Again the graphs clearly show the upward surge of PGA methods.

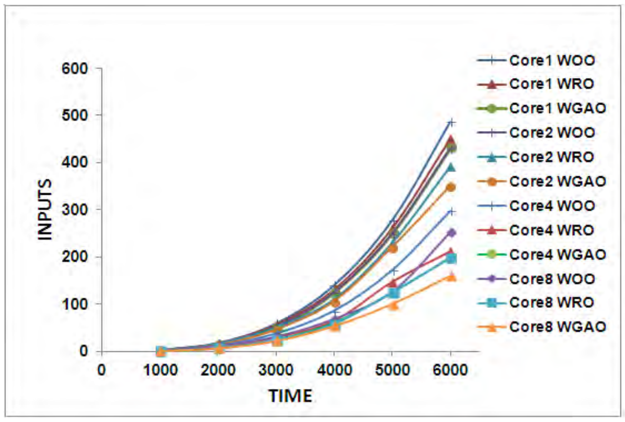

Fig. 6: LUD performance

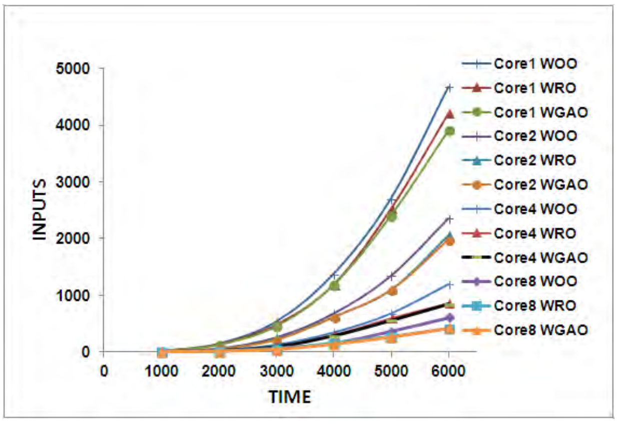

Fig. 7: MM performance

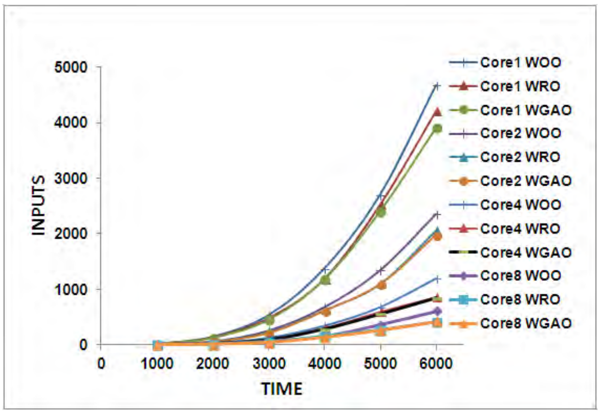

Fig. 8: QS performance

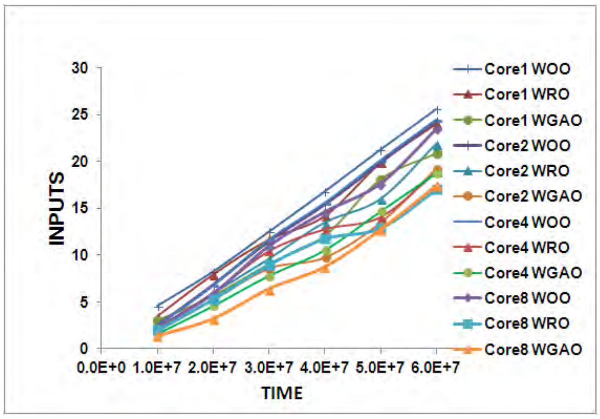

Fig. 10: PRIMS performance

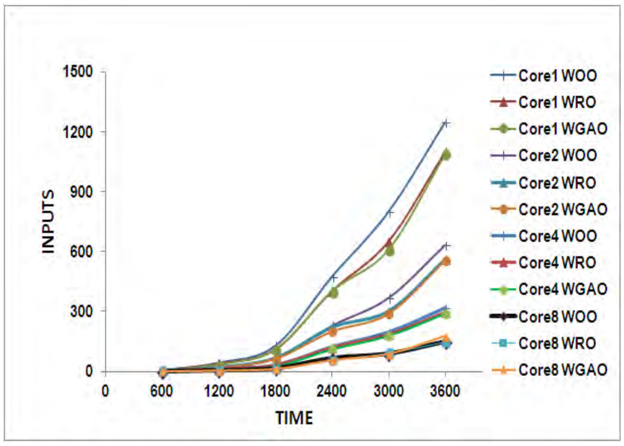

Fig. 9: DES performance

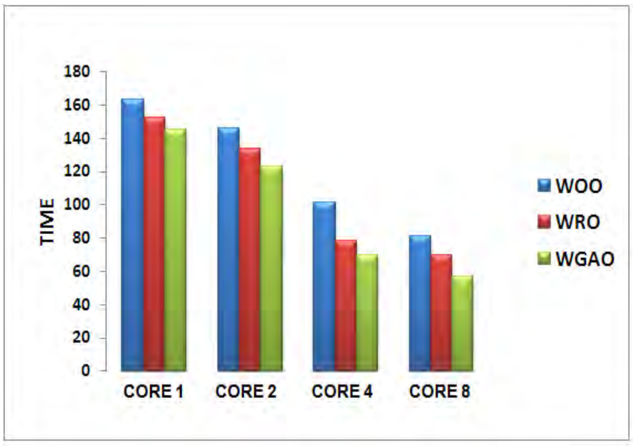

Fig.11: LUD: Core performance enhancement

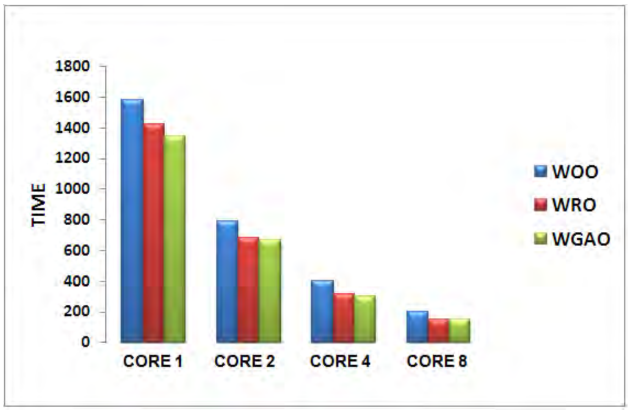

Fig. 12: MM: Core performance enhancement

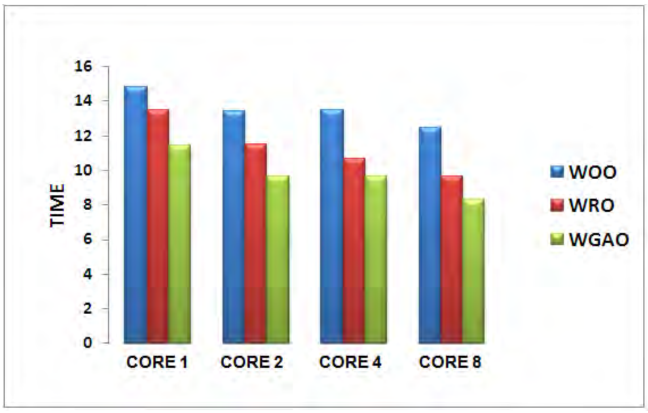

Fig. 13: QS: Core performance enhancement

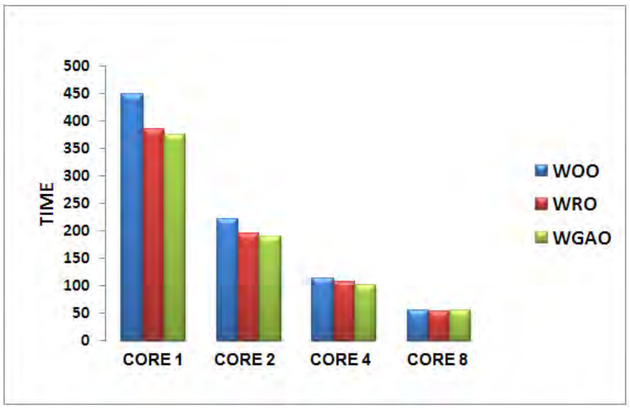

Fig.14: PRIMS: Core performance enhancement

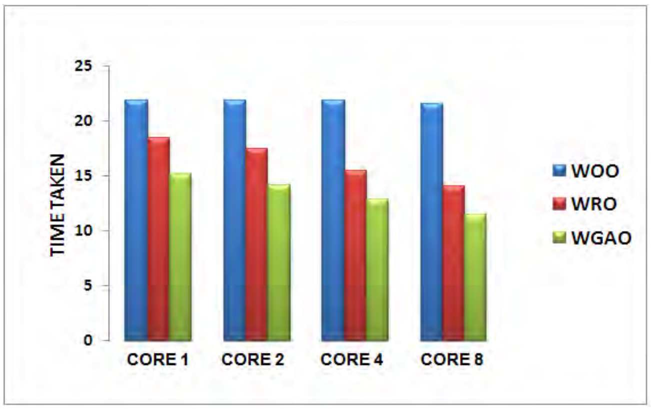

Fig.15: DES: Core performance enhancement

VI. CONCLUSIONS

In Compiler Optimization research the phase ordering is an important performance enhancement problem. This paper discusses the study on comparing performance of various benchmarks on different core machines. We conclude that as the number of cores increase the performance of the benchmark program increases along with the usage of PGA, but the major concern in the experiment was that the master core has to wait until the slave cores returned the values to it. This was primarily due to the usage of Synchronized communication between the Master-Slave cores in the system. Further it may be explicitly noted that apart from PRIMS algorithm for core-8 system all other Bench marks exhibit better average performance.

VII. FUTURE WORK

There are a variety of enhancements that may be planned for future. we would not only like to examine the usage of other selection techniques like rank selection, tournament selection and other methods in the PGA, but also randomly change these methods to see the enhancement on the performance of the code. The second change that we are planning is to remove the cap on the PGA generations depending on the requirement of the application it may be increased or decrease. The Third change is to use multi point cross over technique to gradually bring down the number of generations so that the solution converges quickly. Fourthly, we plan to study various other algorithms like hill climbing, simulating annealing to decide next phase ordering. Finally, we propose to combine Elitism and SUS for better performance and also like to enhance the experimental system to clusters allowing Message passing interface to communicate between the master node and the slave nodes.

REFERENCES

[1] Steven R. Vegdahl. Phase coupling and constant generation in an optimizing microcode compiler. In Proceedings of the 15th annual workshop on Microprogramming, pages 125. 133. IEEE Press, 1982.
[2] D. Whitfield and M. L. Soffa. An approach to ordering optimizing transformations. In Proceedings of the second ACM SIGPLAN symposium on Principles & Practice of Parallel Programming, pages 137.146. ACM Press, 1990.
[3] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. In Workshop on Languages, Compilers, and Tools for Embedded Systems, pages 1.9, May 1999.
[4] Prasad Kulkarni, Wankang Zhao, Hwashin Moon, Kyunghwan Cho, David Whalley, Jack Davidson, Mark Bailey, YunheungPaek, and Kyle Gallivan. Finding effective optimization phase sequences. In Proceedings of the 2003 ACM SIGPLAN conference on Language, Compiler, and Tool for Embedded Systems, pages 12.23. ACM Press, 2003.
[5] T. Kisuki, P. Knijnenburg, and M.F.P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In Proc. PACT, pages 237.246, 2000.
[6] SpyridonTriantafyllis, Manish Vachharajani, Neil Vachharajani, and David I. August. Compiler optimization-space exploration. In Proceedings of the international symposium on Code Generation and Optimization, pages 204.215. IEEE Computer Society, 2003.
[7] Prasad A. Kulkarni, Stephen R. Hines, David B. Whalley, Jason D. Hiser, Jack W. Davidson, and Douglas L. Jones. Fast and effcient searches for effective optimization-phase sequences. ACM Trans. Archit. Code Optim., 2(2):165.198, 2005.
[8] A. Nisbet. Genetic algorithm optimized parallelization. Workshop on Problem and Feedback Directed Compilation, 1998.
[9] Mark Stephenson, SamanAmarasinghe, Martin Martin, and Una-May O'Reilly. Meta optimization: improving compiler heuristics with machine learning. In Proceedings of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation, pages 77.90. ACM Press, 2003.
[10] D. Whitfield and M. L. Soffa. An approach to ordering optimizing transformations. In Proceedings of the second ACM SIGPLAN symposium on Principles & Practice of Parallel Programming, pages 137.146. ACM Press, 1990.
[11] Deborah L. Whitfield and Mary Lou Soffa. An approach for exploring code improving transformations. ACM Trans. Program. Lang. Syst., 19(6):1053.1084, 1997.
[12] Prasad Kulkarni, Wankang Zhao, Hwashin Moon, Kyunghwan Cho, David Whalley, Jack Davidson, Mark Bailey, YunheungPaek, and Kyle Gallivan. Finding effective optimization phase sequences. In Proceedings of the 2003 ACM SIGPLAN conference on Language, Compiler, and Tool for Embedded Systems, pages 12–23. ACM Press, 2003.
[13] SpyridonTriantafyllis, Manish Vachharajani, Neil Vachharajani, and David I. August. Compiler optimization-space exploration. In Proceedings of the international symposium on Code Generation and Optimization, pages 204.215. IEEE Computer Society, 2003.