Journal of Engineering and Applied Sciences 13 (22): 9777-9783, 2018
ISSN: 1816-949X
© Medwell Journals, 2018

# Finding a Good Global Sequence Using Multi-Level Genetic Algorithm

Zeyd S. Alkaaby, Esraa H. Alwan and Ahmed B.M. Fanfakh
Department of Computer Science, College of Science for Women,
University of Babylon, Hillah, Iraq

**Abstract:** Trying all the optimization sequences manually to find out a one that give the best performance is not practical solution. Therefore, it is essential to layout a schema which is able to introduce an optimization sequence with better performance for a given function. In this research, multi-levels genetic algorithm has been used to find a good optimal sequence. Our method has three levels. In the first level, the programs search space is divided into three groups and try to find a good sequence for each program in group. These good sequences for each program will be used as initial seed to find good sequence for all programs in that group. This process will be repeated for all three groups to find good sequence for each one. Then, these good sequences from three groups will be used as a seed for initial population to the third level. Genetic algorithm will use the resulting sequences to find out one good optimal sequence for all these groups. LLVM compiler framework has been used to validate the proposed method. Experiments that have been implemented on the generated good sequence for different benchmarks show the effectiveness of the proposed method. Overall, it achieves better performance compare with the -O2 flag.

**Key words:** LLVM, population, sequences, experiments, -O2 flag, performance

## INTRODUCTION

Modern compilers introduce a massive number of optimization passes targeting different code segments of an application. These optimization passes can transform the code segment which might be a basic block, a function or the whole program to optimize one (Purini and Jain, 2013). The optimization can be applied through the whole life of the program in another word, at different compilation stages (Almagor *et al*., 2004). Although, the purpose of optimization is producing better code (speed or code size), however, there is no grantee that the resulting code is doing better than the original one. Typical optimizer is made up of a set of analysis passes followed by transformation passes. The analysis passes responsible for collecting information about the program and the transformation passes are transforming the code segment to a new version. There is standard optimization level provided by compiler developer such as -O1, -O2, -Os. Through the execution of the program all the optimization flags will be turn off by default and the expert can turn some or all of them on according to the program needs. For example, the GCC has more than 200 passes while the LLVM Clang and Opt have more than 100 passes (Ashouri *et al*., 2017). The sequence of these passes called optimization sequence. Choosing the right sequence can make the program give better performance according to it is execution time or code size.

Genetic Algorithm (GA) is a well-known algorithm that is adopted depends on theory of evolution. Several researchers are successfully applied this algorithm in a phase ordering of compiler optimization (Cooper *et al*., 1999; Purini and Jain, 2013). In this research, Multi-Level Genetic Algorithm (MLGA) is introduced. In this method, the programs search space is divided into three groups. GA work on each group alone to find a good sequence for that group. Then, these good sequences that are resulted from these three groups will be used as a seed for initial population in the next level. Genetic algorithm will use the resulting passes in these good sequences to find out one optimal sequence for all groups. The proposed method is implemented using the Low Level Virtual Machine (LLVM) compiler framework (Ashouri *et al*., 2017; Lattner and Adve, 2004). LLVM uses a combination of a low level virtual instruction set combined with high level type information. An important part of the LLVM design is its Intermediate Representation (IR). This has been carefully designed to allow for many traditional analyses and optimizations to be applied to LLVM code and many of them are provided as part of the LLVM framework.

**Corresponding Author:** Zeyd S. Alkaaby, Department of Computer Science, College of Science for Women, University of Babylon, Hillah, Iraq

**Literature review:** Finding the right set of optimizations for each application is easy task, since, the search space is extremely large. Recent research has focused on intelligent search space exploration in order to effciently search for the right optimization sequence. In Kulkarni *et al*. (2006) exhaustive search strategy has been introduced to find optimal compilation sequences for each of the functions in a program. This technique used to reduce the number of sequences to be tried. This method may be not practical, however, it is gave good information about optimization sequence. A highly innovative strategy for iterative compilation was proposed by Parello *et al*. (2004). They worked to employ performance counter at per stage of the tuning process to introduce further optimization sequence. These sequences are evaluated and based on the new performance counter they would prefer new optimization to investigate. Pan and Eigenmann (2006) proposed an algorithm called combined elimination which remove the optimization that have poor interact among each other and in turn have a bad impact on the execution time of a program. Multiple subsequences may falsely interact with each other and affect the potential program speedup achieved. The algorithm tries to find a near optimal solution. Cavazos *et al*. (2007) applied supervised learning to predicting good compiler optimizations. Compiler heuristics have been automatically generated to predict good compiler optimization by using performance counters. In this method, there is no need to do manual experimentation. Moreover, they can generalize the resulted heuristic for unseen program. The researches by Sanchez *et al*. (2011), Jantz and Kulkarni (2013) introduce a novel approach. They use heuristic techniques to design quickly feature-vector suitable for individual function through JIT compilation.

**Multi-Level Genetic Algorithm (MLGA):** The modern compilers provide many optimizations techniques applied in predetermined ordering to increase program performance. Although, these sequences look globally optimal with respect to the program space, sometimes they are producing corrupted codes or their performance are bad for an individual program. Moreover, it can contain optimization that does not contribute in the program speedup or have a poor impact on the program speedup. To achieve significant improvement an approach to find a good global optimization sequence is proposed. The good sequence which covers all programs characteristics in the program space that has a better performance than O2 is constructed. The selected program space which contains a large set of programs is chosen from many benchmarks. This set provides a wide range of scientific applications that cover different program's characteristics.

In this study, Multi-Levels Genetic Algorithm (MLGA) which is composed of three levels is proposed. In the first level, the search space is divided into three groups. The proposed algorithm starts with randomly chosen initial population. It iteratively applied selection, crossover and mutation for a given number of generations. The fitness function computes the execution time of the program that results from the optimization sequence. Finally, disable some optimizations that did not seem to improve the running time of the optimization sequence by using reduction sequence as shown in algorithm 2.

An optimal sequence recorded that is corresponding to fitness value with minimum execution time compared to other sequences. The proposed genetic algorithm is applied to all other programs in the benchmark to hold the best sequence for each one. The given sequence of this level is called Sub-Optimization Sequence (SOS).

To optimize a sequence for a sub-group of programs which belong to one benchmark, the proposed genetic algorithm is rerun again. Consequently, the fitness function is calculated by taking the average for the sub-group of programs. Moreover, the evaluation process will continue until reaching to the stopping criteria as in SOS algorithm. One optimal sequence has been recorded per sub-group. The sequence that output of this level is called Sub-Global Optimization Sequences (SGOS's).

In the third level, the SGOS's that resulted from the previous level is used as a seed of initial population to MLGA. Periodically, the genetic operators, selection, crossover and mutationare applied for each iteration. Offspring fitness's function is calculating by taking the average of the execution time of three benchmarks from running the optimization sequence over them. Each program is executed three times to get accurate value and the average is calculated for these executions. The algorithm stops its generations when reaching to a specific condition. The sequence that results from this level is a general optimal optimization sequence which is called Global Optimization Sequence (GOS) (Fig. 1).

**Details of the multi-level Genetic algorithm:** The initial population is generated randomly by selecting a set of optimization passes from 63 passes available in the LLVM as in Table 1. The size of each chromosome set optimization passes) is variable with maximum length of 60 passes. Proposed Genetic algorithm starts its search from initial population of size 100 chromosome. Each

Table 1: List of LLVM 63 machine-independent optimizations

| List of optimizations | | | |
|---|---|---|---|
| -adce | -always-inline | -argpromotion | -codegenprepare |
| -constmerge | -constprop | -correlated-propagation | -dce |
| -deadargelim | -die | -dse | -early-cse |
| -globaldce | -globalopt | -gvn | -indvars |
| -inline | -instcombine | -instsimplify | -internalize |
| -ipconstprop | -ipsccp | -jump-threading | -licm |
| -loop-deletion | -loop-idiom | -loop-instsimplify | -loop-reduce |
| -loop-rotate | -loop-simplify | -loop-unroll | -loop-unswitch |
| -loops | -lower-expect | -loweratomic | -lowerinvoke |
| -lowerswitch | -memcpyopt | -mergefunc | -mergereturn |
| -partial-inliner | -prune-eh | -reassociate | -scalarrepl |
| -sccp | -simplify-libcalls | -simplifycfg | -sink |
| -tailcallelim | -targetlibinfo | -no-aa | -tbaa |
| -basicaa | -basiccg | -functionattrs | -scalarrepl-ssa |
| -domtree | -lazy-value-info | -lcssa | -scalar-evolution |
| -memdep | -strip-dead-prototypes | | |

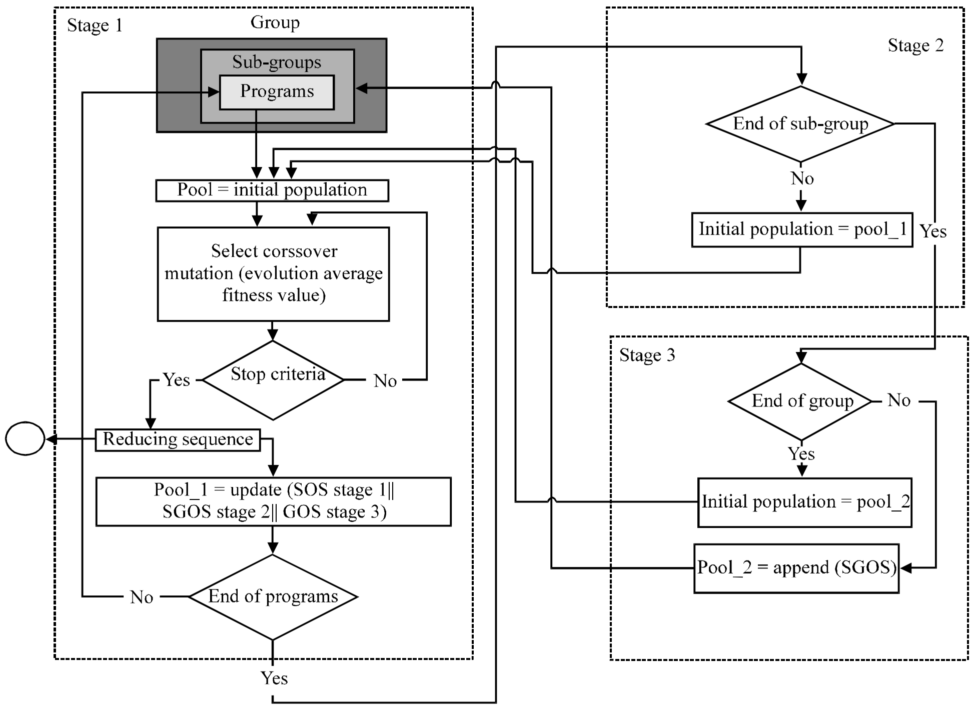
Stage 1
Group
Sub-groups
Programs
Pool = initial population
Select corssover mutation (evolution average fitness value)
Stop criteria
Yes
No
Reducing sequence
Pool_1 = update (SOS stage 1|| SGOS stage 2|| GOS stage 3)
End of programs
No
Yes
Stage 2
End of sub-group
No
Initial population = pool_1
Yes
Stage 3
End of group
No
Yes
Initial population = pool_2
Pool_2 = append (SGOS)

Fig. 1: Overview of the multi-level Genetic algorithm

chromosome is encoded by using integer values from 0-60. Each value corresponds real optimization passes while the zero gene used to implement a null gene. However, variable chromosome length is obtained by using the null genes. Moreover, each chromosome represent a sequence of optimization passes which is used to compile a program and calculates the execution time as shown in Fig. 2 and 3. The fitness function (execution time) is attached with each chromosome.

In order to keep high diversity of population and prevents early convergence on poor solutions. Multi-level genetic algorithm is implemented with a rank selection for more detail about this selection method see reference

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 11 | 00 | 23 | 60 | 34 | 00 | 12 | 10 | 06 | 00 |
| inline | | dic | instsimplify | -gvn | | basicaa | no-aa | licm | |

Fig. 2: Execution time

(Reeves and Rowe, 2002). The algorithm stops its evaluation when the standard deviation of the overall chromosomes is <0.01, otherwise, it continues its evaluation for the second round.

The uniform crossover operator (UX) uses to generate two different children from two selected parents. The UX operator scans all the genes of the two parents with probability of 0.4 to swaps genes. The probability of

| pg | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|---|
| Parent 1 | 34 | 14 | 00 | 61 | 00 | 12 | 54 | 00 | 01 |
| Parent 2 | 00 | 56 | 30 | 28 | 17 | 40 | | | |
| Child 1 | 00 | 14 | 30 | 28 | 17 | 12 | 54 | 00 | |
| Child 2 | 34 | 56 | 00 | 61 | 00 | 40 | | | 01 |

Fig. 3: Fitness function

applying this operator is equal to 0.9. Consequently, one-point mutation of probability 0.02 is applied over the two children that results from the UX operator

In general, after reaching the stop condition, the algorithm retrieves the sequence with minimum fitness value. Then the reduction algorithm is applied to remove harmful passes. The steps form 2-4 are repeating on the rest of the programs until the sub-group is completed. SOS's are assigned to pool_1 which are in turn used as new seeds to initialization population in the second level of the MLGA.

In the second level, the SOSs that are resulted from the first level as initial see to find good optimization sequence for all the programs in that group. The output of this stage will be one good sequence for each sub-group of programs which are called SGOSs.

The best individual of SGOS algorithm from each sub-group is assigned to seed pool, this pool is referred pool_2. MLGM algorithm in its turn depends on the individuals of pool_2 as initial population. At each generation in the MLGA algorithm, selection, crossover and mutation are produced new individuals where each one represents a candidate solution for the best global sequence. Fitness function of each new child is calculated by taking the average of the execution time when the child sequence is applied over all the programs in the group. The best individual is selected with minimum average execution time. The output of this level is GOS. Algorithm 1 shows the details of MLGA.

**Algorithm 1; Multi-level Genetic algorithm:**

```
Input: a set of sub-group, set of optimization sequence
Output: general global optimization sequence
Group → [ ]
Sub-group → [ ]

  While not end of the group
      While not end of sub-group
          Size of.pop →  [ ]
          Prog →  [ ]
        For I = 1 to prog
            Create new pop
            Fitness  →  time(execution-prog)
            L: selection.
               Crossover
               Mutation
              If stop condition is not met go to L
              Reduction SOS
          Pool = SOS
     Init-pop = pool
           L: selection
              Crossover
              Mutation
              For I = I to prog
              Fitness          average time(execution -progs)
              If stop condition is not met go to L
              Pool 1 = SGOS
         Else
      Init → pop = pool 1
      L: Selection
         Crossover
         Mutation
         For I = 1 to sub-group*progs
              Fitness  →     average time(execution-progs)
              If stop condition is not met go to L
               Return GOS
```

**Algorithm 2; Sequence reduction:**

```
Input: SOS and program
Output :reduction SOS
 opt-seq →       SOS
Best-time →      execution (prog,SOS)
   While not end of  opt-seq
     opt-seq -1   →      execution (prog, opt-seq)
     If (opt-seq)time < = best-time
     SOS = Opt-seq
     Else do not change
     End  of while
   Return  SOS
```

## MATERIALS AND METHODS

**The experimental setup:** The experiments in this study are carried out over Core I7 processor model and LLVM-3.2 compiler infrastructure is used to validate the proposed method. LLVM Intermediate Representation (IR) is the most important aspect of its design. The C language frontend Clang can convert the c source code program into IR code which can be stored in machine-readable bitcode format. We applied machine independent optimization option (like -O1, -O2, -O3) chosen by the user on this middle end (IR) representation. Opt and llc are tools provided by LLVM infrastructure. When we get the optimize sequences, opt tool will use this sequence as an input on the command line and applied on the bitcode file format (LLVM IR bitcode file format). After we get the optimize version of LLVM bitcode format, this file will be converted to target assembly code using the llc translator. In this study, we focus on the interactions between machine-independent optimizations acting on the LLVM IR

Our experiments use a set of LLVM (63) optimizations passes in order to find a sequence that will give near optimal execution time for each group. For our proposed method (MLGA), we set the initial parameters to the following values, population size = 100, probability of crossover is 0.9, mutation 0.02, rate of occurrence gene 0.4, stop criteria of the standard deviation 0.01. Finally, we used one metric to evaluate the performance of our algorithm which is:

$$\text{speedup} = \frac{\text{baseline\_runtime}}{\text{new\_runtime}}$$

where, the baseline_runtime represents the execution time of program without optimization:

$$\text{improvement} = \left(\left(\text{speedup}\right)\text{-}1\right)\times 100$$

## RESULTS AND DISCUSSION

**The experimental evaluation:** This section discusses the results obtained from the implementation of the proposed method that we presented in this study. Several programs from three benchmarks have been used as case studies. These benchmarks are polybench from polybench 3.0 suit, shootout and standford from LLVM suit. Moreover, they cover different kind of programs which are involving floating point arithmetic operation programs, programs containing tail recursive, data mining, image processing, linear algebra, etc.

As mention previously, MLGA is used that composed of three levels. Script file has been used to extract programs from a benchmark and carried out on MLGA to construct optimization sequence. Figure 4-6 illustrate the output of the second level for the proposed method where they clarify a comparison between O2 and the proposed method (MLGA) for three benchmarks polybench, shootout and Standford respectively. Figure demonstrate that the MLGA outperformed O2 for all benchmarks. Whereas O2 ignores the interaction between optimization passes, MLGA is able to detect the relation between them and pruning the non-affective ones.

Sequence reduction algorithm is used to eliminate the unnecessary optimization passes. Tables 2 and 3 illustrate the optimization sequence before and after reduction process, respectively.

Table 4 shows the SOS's best optimization sequences for each benchmark separately. These sequences represent the output of the first level and will be the seed of initial population for the second level. The final sequence that will be produced in final step is Global Optimization Sequence (GOS) where it is resulted after run it on all the benchmarks. Figure 7 shows the comparison between the O2 and GOS performance. The comparison illustrate that the resulted sequence of GOS overcomes the O2 with ratio of 10% in the execution time. Table 5 shows passes of the discovered global optimization sequence.

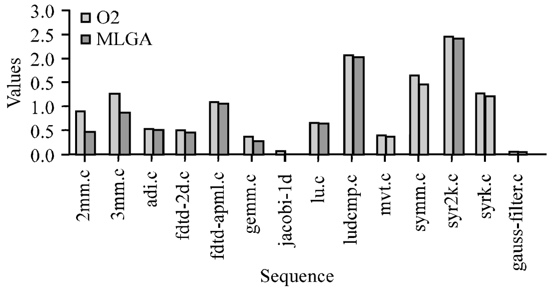

Fig. 4: Optimization sequence before reduced

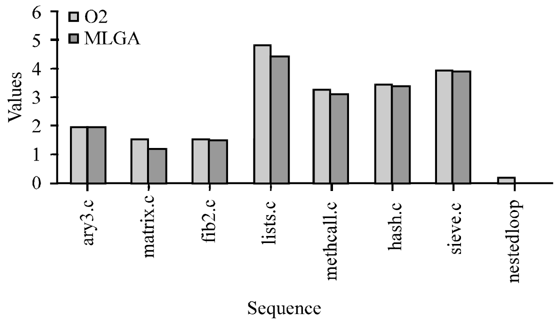

Fig. 5: Optimization sequence aftere reduced

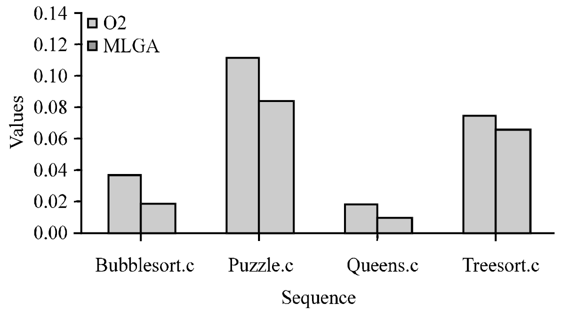

Fig. 6: Optimization sequence for bench mark

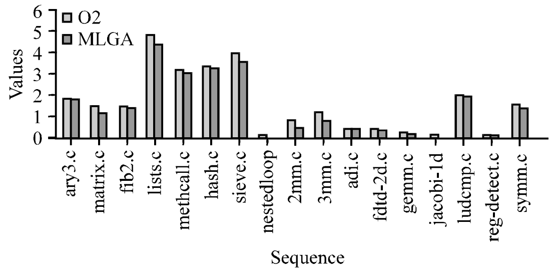

Fig. 7: Comparison between O2 and GOS

**Passes of the discovered GOS:** Functionattrs-simplifycfg-inline-codegenprepare-early-cse-indvars-loop-deletion-instcombine-functionattrs -basicaa -inline -loop-rotate-indvars-loop-unroll-instcombine-licm-gvn-loop-rotate-loop-idiom-loop-deletion-inline-loop-rotate-inline-memcpyopt -tailcallelim -loop-rotate s-inline-instcombine

Table 2: the optimization sequence before reduced it

| Programs | Optimization sequence |
|---|---|
| 2mm.c | -basicaa-argpromotion-simplifycfg-memcpyopt-inline-gvn-globaldce -adce-loop-rotate-loop-unswitch-globalopt-licm |
| 3mm.c | -loop-rotate-licm-basicaa-globaldce-basicaa-die-loop-rotate-lower-expect-lazy-value-info-gvn |
| adi.c | -instcombine-loop-rotate-loweratomic-indvars-loops-reassociate-functionattrs-gvn-targetlibinfo |
| atax.c | -basicaa-lazy-value-info-correlated-propagation-loop-instsimplify-gvn-indvars-dce-sccp-lazy-value-info |
| doitgen.c | -instsimplify-loop-rotate-basicaa-instcombine-indvars-indvars-instcombine-licm-gvn-gvn |
| jacobi-1d-imper.c | -basicaa-instcombine-inline-loop-rotate-indvars-loop-idiom-loop-deletion-loop-unroll-gvn-instcombine-indvars-memdep |
| seidel.c | -simplifycfg-loop-rotate-instcombine-gvn-loop-simplify-constprop-die-loweratomic-loop-idiom |
| syr2k.c | -basicaa-loop-rotate-indvars-loop-rotate-loop-unswitch-lowerswitch-gvn-licm-correlated-propagation-targetlibinfo-instcombine |

Table 3: the optimization sequence after reduced it

| Program | Optimization sequence |
|---|---|
| 2mm.c | -basicaa -simplifycfg-inline-loop-rotate-licm |
| 3mm.c | -loop-rotate-licm-basicaa |
| adi.c | -instcombine-loop-rotate-indvars-gvn |
| atax.c | -basicaa-gvn |
| doitgen.c | -loop-rotate-basicaa-instcombine-indvars-licm |
| jacobi-1d-imper.c | -basicaa-instcombine-inline-loop-rotate-indvars-loop-idiom-loop-deletion-loop-unroll-gvn-instcombine |
| seidel.c | -loop-rotate-instcombine-gvn |
| syr2k.c | -basicaa-loop-rotate-licm |

Table 4: SOS's sequences for each benchmark

| Benchmark | SGOS |
|---|---|
| Polybench | -basicaa-simplifycfg-inline-loop-rotate-licm-loop-rotate-licm-basicaa -instcombine -indvars-gvn-loop-idiom-loop-deletion-loop-unroll |
| Shootout | -functionattrs-simplifycfg-inline-codegenprepare-early-cse-indvars-loop-deletion-instcombine-functionattrs-basicaa-inline-loop-rotate- -indvars-loop-unroll-instcombine-licm-gvn-loop-rotate-loop-idiom-loop-deletion-inline-loop-rotate-inline-memcpyopt-tailcallelim-loop-rotate-inline-instcombine-loop-deletion-tailcallelim |
| Standford | -early-cse -jump-threading -inline-instcombine-targetlibinfo-tailcallelim-indvars-sink-functionattrs-always-inline-loop-otate-basicaa-scalarrepl-ssa- -loop-idiom-loop-rotate-loop-unroll-lcssa-lowerinvoke-always-inline-scalarrepl-gvn-scalarrepl-ailcallelim-dce-licm-prune-eh-instcombine-loops-ipsccp-loops-basicaa-scalarrepl-ssa-globalopt-argpromotion-mergefunc -loop-deletion -correlated-propagation-gvn |

loop-deletion-tailcallelim-basicaa-simplifycfg-inline-loop-rotate-licm-loop-rotate-licm-basicaa-instcombine-indvars-gvn-loop-idiom-loop-deletion-loop-unroll.

## CONCLUSION

In this study, MLGA has been introduced to make a right choice of compilation optimization passes. MLGA have been evaluated on different instances of real-world benchmarks. Our MLGA used rank selector, UX crossover and LLVM infrastructure has been used to validate the proposed method. Many experiments have been implemented on generated sequence and they show the effectiveness of this sequence. Overall, it achieves better performance compare with the -O2 flag.

## REFERENCES

Almagor, L., K.D. Cooper, A. Grosul, T.J. Harvey and S.W. Reeves *et al.*, 2004. Finding effective compilation sequences. Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES '04), June 11-13, 2004, ACM Press, Washington, USA., ISBN:1-58113-806-7, pp: 231-239.

Ashouri, A.H., A. Bignoli, G. Palermo, C. Silvano and S. Kulkarni *et al.*, 2017. MiCOMP: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning. ACM. Trans. Archit. Code Optim., 14: 1-28.

Cavazos, J., G. Fursin, F. Agakov, E. Bonilla and M.F. O'Boyle *et al.*, 2007. Rapidly selecting good compiler optimizations using performance counters. Proceedings of the International Symposium on Code Generation and Optimization (CGO '07), March 11-14, 2007, IEEE, San Jose, California, USA., ISBN:0-7695-2764-7, pp: 185-197.

Cooper, K.D., P.J. Schielke and D. Subramanian, 1999. Optimizing for reduced code space using genetic algorithms. Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES '99), May 05, 1999, ACM, Atlanta, Georgia, USA., ISBN:1-58113-136-4, pp: 1-9.

Jantz, M.R. and P.A. Kulkarni, 2013. Performance potential of optimization phase selection during dynamic JIT compilation. Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'13), March 16-17, 2013, ACM, Houston, Texas, USA., ISBN:978-1-4503-1266-0, pp:131-142.

Kulkarni, P.A., D.B. Whalley, G.S. Tyson and J.W. Davidson, 2006. Exhaustive optimization phase order space exploration. Proceedings of the International Symposium on Code Generation and Optimization (CGO '06), March 26-29, 2006, IEEE, New York, USA., pp: 306-318.

Lattner, C. and V. Adve, 2004. LLVM: A compilation framework for lifelong program analysis and transformation. Proceedings of the International Symposium on Code Generation and Optimization, March 20-24, Chicago, IL, USA., pp: 75-86.

Pan, Z. and R. Eigenmann, 2006. Fast, automatic, procedure-level performance tuning. Proceedings of the 2006 International Conference on Parallel Architectures and Compilation Techniques (PACT), September 16-20, 2006, IEEE, Seattle, Washington, USA., ISBN:978-1-5090-3022-4, pp: 173-181.

Parello, D., O. Temam, A. Cohen and J.M. Verdun, 2004. Towards a systematic, pragmatic and architecture-aware program optimization process for complex processors. Proceedings of the 2004 ACM/IEEE International Conference on Supercomputing (SC'04), November 6-12, 2004, IEEE, Pittsburgh, Pennsylvania, USA., ISBN:0-7695-2153-3, pp: 15-15.

Purini, S. and L. Jain, 2013. Finding good optimization sequences covering program space. ACM Trans. Archit. Code Optim., 9: 1-23.

Reeves, C.R. and J.E. Rowe, 2002. Genetic Algorithm-Principles and Perspectives: A Guide to GA Theory. Kluwer Academic Publishers, Boston, USA.

Sanchez, R.N., J.N. Amaral, D. Szafron, M. Pirvu and M. Stoodley, 2011. Using machines to learn method-specific compilation strategies. Proceedings of the 9th Annual International Symposium on Code Generation and Optimization (CGO), April 2-6, 2011, IEEE, Chamonix, France, ISBN:978-1-61284-356-8, pp: 257-266.