Using Machine Learning to Predict the Sequences of Optimization Passes

Laith H. Alhasnawy, Esraa H. Alwan, and Ahmed B. M. Fanfakh(✉)

Department of Computer Science, College of Science for Women, University of Babylon, Hillah, Iraq
lythhamd071@gmail.com,
{wsci.israa.hadi,wsci.ahmed.badri}@uobabylon.edu.iq

Abstract. The manual tuning for the sequence of optimization passes in modern compilers was impractical, where this sequence was not general to all benchmark programs in achieving optimal performance. Therefore, the process of selecting a set of passes manually them over a particular program to achieve optimal performance is a very difficult problem. Moreover, choosing the order for these passes will add another problem called phase order problem.

In this paper, the proposed approach provides auto tuning optimization sequences instead of manual tuning by building a prediction scheme. The proposed framework used machine learning to find a sequence for each program by collecting these passes based on the features of program. K-Nearest Neighbors classifier algorithm (KNN) is used in the prediction process and improved by that the reduction algorithm that work after it. The reduction algorithm eliminated passes that have a negative impact on program execution time.

The proposed approach was evaluated using the LLVM (Low Level Virtual Machine) compiler under Linux Ubuntu. The obtained results showed that this approach outperforms standard optimization level-O2 of LLVM compiler in improving the execution time byan average of 21% through building a prediction scheme by using KNN algorithm. Consequently, the execution time is improved byan average of 23% due to the application of the reduction algorithm on the sequences of optimization passes resulting from KNN algorithm.

Keywords: LLVM · KNN algorithm · Reduction algorithm · Optimization sequence · Optimization passes

1 Introduction

Modern compilers are portioned into three parts: the front end, middle end and back end. The frontend creates an intermediate representation (IR) for the source program after testing its syntactic and semantic rightness. The middle end which represents the optimizer is a critical part. It applies a sequence of conversions on the IR in order to optimize it to get better execution time, code size, and power consumption. The backend converts the optimized IR to target assembly code. The optimizer part arranged as a sequence of optimization passes that represents analysis passes followed by transformation passes.

A. M. Al-Bakry et al. (Eds.): NTICT 2020, CCIS 1183, pp. 139–156, 2020.
https://doi.org/10.1007/978-3-030-55340-1_10

Analysis passes analyses the program and fetches important information needed through the transformation passes [1].

Compiler developers work on the optimization passes to produce an optimized version from the original version. A code segment may be a function, a basic block, a source file, and the whole program. However, code optimizations are based on application, programming language, and architecture. Generally, the number of optimizations passes in the LLVM compiler is more than 100 passes, while the GCC compiler has more than 200 passes where they are organized at standard optimization levels (e.g. -O1, -O2, -O3) [2].

Every set of optimization passes that can work on the program IR is named an optimization sequence. The set of all optimization sequences is called the optimization sequence space where it is infinitely large because the probability to producing optimization sequences has been increased with growing the number of optimization passes. Thus, the number of the optimization sequences equals $(k)^L$, where k represents the number of optimizations passes, and L represents the optimization sequence length. However, there is a problem in finding a good optimization sequences because there are a large number of optimization sequences that contain many passes that interact with each other in complex ways [1] and [5].

These optimization sequences are applied in a fixed order. This order is tuned manually for a specific group of programs impractically. It also requires to tuned it again at every time the compiler is modified to a new version or when a new optimization passes has been added [8]. Therefore, the word "optimization" is an inaccurate name because these compilers may have a good performance for particular programs, but its performance is bad for other programs, see [2] and [12]. Moreover, there is a problem in understanding the effect of the optimization passes behavior on the code segment to be optimized. These due to these passes are interacted with each other in an unpredictable manner during the optimization process. Therefore, there are many problems in the choosing the passes that achieve the optimal performance for a particular program and in the choosing the order for these passes [2] and [11].

To solve these problems, this paper introduces a method for auto tuning optimization sequences by building a prediction scheme using machine learning. This method predicts the sequences of optimization passes for each program and order these passes based on the features of that program through applying K-Nearest Neighbors classifier algorithm (KNN).

KNN is one of the machine learning algorithms that work under supervised learning. This algorithm deals with classification tasks in the wide applications of various fields such as pattern recognition, cluster analysis for image databases, internet marketing, and etc. It is proposed by Fix & Hodges, and it is called KNN algorithm because it relies on the use of the nearest neighbors to predict the class of unknown data belonging to those neighbors. It is classified the unknown data based on its features to the nearest neighbor, nearest neighbor is calculated based on the k value that determines the number of nearest neighbors that have been observed after applying the similarity measures [3] and [10].

Then, another algorithm called a reduction algorithm is applied to eliminate optimization passes that have a negative impact on program execution time from the resulting optimization sequence by using KNN algorithm.

This paper is organized as follow: Sect. (2) describes the related works to the proposed approach. Sect. (3) illustrates the proposed approach and the steps of building a prediction scheme. Sect. (4) shows the experimental results of the proposed approach. Finally, Sect. (5) describes the conclusion.

2 Literature Review

In this section, some of the related works that has been done in this field are listed. Matrins et al. [4] presented a Design Space Exploration scheme that uses a clustering approach to find groups of good optimization sequences. This approach reduces search space for the number of optimization passes used during the exploration of optimization sequences, where it combines the previously suggested optimization passes into each group to reduce the exploration time and improve the performance. The unseen program is compiled with different groups of optimization sequences, and the best optimization sequence is chosen for it.

Cavazos et al. [6] proposed a machine learning approach depend on a logistic regression scheme to find good optimization sequence for a particular program. This approach used performance counters which based on the dynamic features for each program when executed, where a similarity metric is found between any two programs depending on the similarities between these features. A machine learning approach predicted mechanism depending on the behavior of many of the trained programs in a training phase to create several optimization sequences. The obtained sequences used for training programs, and the prediction mechanism in the test phase to predict the optimization sequence that will be allowed to compile the unseen program by a training phase.

Purini and Jain [1] proposed an approach to find good optimization sequences sets, which are able to cover several programs in each set. This approach uses random and genetic algorithms to create multiple effective sets of optimization sequences and eliminate passes that cause a negative impact from each set.

Ashouri et al. [7] presented a machine learning approach using Bayesian Network to handle the problem of finding the best optimization sequence. This Network provides a probability distribution for the optimization sequences based on the dynamic features extracted for programs. These features feed the network to produce a program-specific which bias to the best optimization sequence from the probability distribution.

Kulkarni and Cavazos [8] applied a Neuro-Evolution depends on a machine learning approach to build an artificial neural network to predict best optimization sequence for program to be improved. This network uses input features that describe the current state of the program, where the best optimization sequence is found by applying network on all passes to determine the probability of these passes. Then the features are extracted again after applying this network one more time on the pass which represent the highest probability because it represents the best pass prediction in the network. Thus, this process continues for several iterations until the best optimization sequence is found from the successful passes for previous attempts which have been applied to the network.

Junior et al. [15] proposed a hybrid approach to solve the optimization sequence selection problem, which combines machine learning using Support Vector Machine and iterative compilation using Genetic Algorithm. This approach is initially used machine learning to identify potential optimization sequences depending on programs characteristics. Then, it uses iterative compilation to adapt these potential optimization sequences to explore potential search optimization sequences and returns the best optimization sequence by reducing the search space for these potential optimization sequences, this procedure is done through a so-called solution adapter which uses a strategy depends on the iterative compilation to improve the solution found in machine learning. Authors in [16], used multi selection genetic algorithm that work over a cluster of programs to find the best optimization sequence for each cluster.

3 Proposed Approach

The main objective of the proposed approach is to provide auto tuning optimization sequences by building a prediction scheme. This scheme uses machine learning to find a sequence for each program instead of the manual tuning. The outline of the proposed approach is as follows:

- Extract static features for each program before and after applying passes, where each pass is applied to all programs.
- Calculate the execution time for each program before and after applying passes.
- Using KNN algorithm to build a prediction scheme.
- Applying reduction algorithm to improve the resulting sequences from KNN.

The LLVM Clanguage frontend (Clang) converts the source program code to machine readable bit code file format (.bc). Moreover, it is providing two useful tools, the first one is opt, and the second is llc. The first tool optimizes the program by applying the specific sequence of optimization passes and stores the result in bit code format file. The second tool converts an LLVM IR from a bit code file to targetcode. After compiling the program using Clang -O0 level (i.e. no optimization applied) then the optimization pass called–scalarrepl (scalar replacement) is applied alongside with each pass. This pass converts the LLVM IR to Static Single Assignment (SSA) to enable other passes to perform their optimizations. Now, the specific optimization sequence for each program can be applied depending on its features by using the opt tool as well. Finally, llc tool converts the LLVM IR produced from the optimizer to target code. Table 1 shows the optimization passes used in -O2 level of LLVM compiler [1].

Table 1. Optimization passes of –O2 standard optimization level

List of optimizations passes			
-scalarrepl	-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	-adce
-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		

Figure 1 shows a summary of the compilation process for each program in our approach by using the optimization passes shown in the Table 1 when building a prediction scheme.

3.1 Features Extraction Using Instruction Count

To extract the static features, the programs must be stored in bit code file format (.bc). Then LLVM InstCount pass is applied to extract them. Overall, these features describe the static behavior for program. Table 2 illustrates the LLVM static features. The total number of them equals 39 features distributed among different programs. Each one has its own numerical value, which it is varying from one program to another as shown in [13] and [14].

In our approach these features are used as shown below:

1. In the training set, the features for each program are extracted before and after applying each pass to each program.
2. When using KNN algorithm, the similarity for each program is calculated based on its features as shown in the scheme of finding optimization sequence.
3. In the testing set, each program is rounded based on its features to the closest program in the training set by using KNN algorithm.

There are many similarity measures to calculate the similarity for features such as Cosine, Simple Matching Coefficient, Jaccard Coefficient and etc. The cosine similarity shown in Eq. (1) was used as a similarity measure in our approach because it is one of

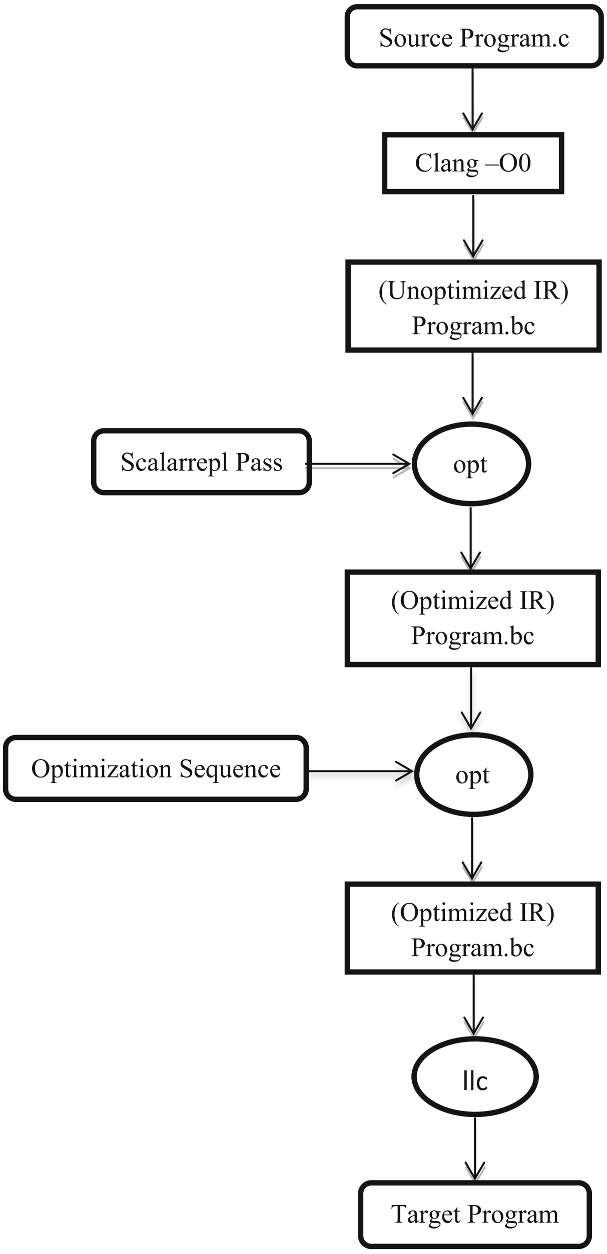

Fig. 1. Program compilation process.

Table 2. The static features for programs

List of static features					
Add instructions	FAdd instructions	GetElementPtr instructions	Ret instructions	SRem instructions	ZExt instructions basic blocks
Alloca instructions	FCmp instructions	ICmp instructions	SDiv instructions	Shl instructions	
And instructions	FDiv instructions	Load instructions	Sub instructions	Store instructions	Memory instructions non-external functions
AShr instructions	FMul instructions	Mul instructions	Switch instructions	Trunc instructions	
BitCast instructions	FPExt instructions	or instructions	SExt instructions	URem instructions	
Br instructions	FPToSI instructions	PHI instructions	Select instructions	Unreachable instructions	
Call instructions	FSub instructions	PtrToInt instructions	SIToFP instructions	Xor instructions	

the most useful measures when using sparse data which contain a few non-zero values (i.e. asymmetric). This measure ignores the zero values matching and handles other non-zero values [9].

$$\text{Sim(p, pi)} = \frac{\sum_{w=1}^{m} Pw \times Piw}{\sqrt{\sum_{w=1}^{m}(Pw)^2} \times \sqrt{\sum_{w=1}^{m}(Piw)^2}} \tag{1}$$

Where *p* represents the basic program, and *pi* represents other programs. In our approach we observe this sparse because "for example" there is a program that has 15 features from the total number that equals 39 features, and another program has 25 features from the total number. This difference in the features number between these two programs, and also the difference for these two programs with the total number of features are processed by placing zero value for features not available in these programs in order to be equal to the total number of features. Thus, each one of the programs contains 39 values mostly zero. As a result, most feature values are zero for the benchmark programs which have been processed through this metric.

3.2 Build a Prediction Scheme

In our approach, the prediction scheme is built to find the sequence of optimization passes for each program through the data set as shown below.

3.2.1 Data Set

The data set consists of programs taken from several benchmarks, see [12] for more information [12]. Examples about these benchmarks are polybench, shootout, standford, and etc. Benchmark programs include various topics like image processing, data structures, data mining, tail recursive, linear algebra. The data set is divided into 70% of the training set and 30% of the testing set.

3.2.1.1 Training Set

In the training set, the features for each training program are extracted before and after applying each pass, and the execution time is also calculated before and after applying each pass. Where - scalarrepl pass was applied together with each pass.

After, KNN algorithm is applied to find a sequence of optimization passes for each program. Then the order of these passes is determined according to the program features.

As a result, these programs are represented as a tree consisting of 62 programs divided into 5 levels, where each level contains twice the level that precedes it (i.e. the first level contains 2 programs, the second level on 4 programs, the third level on 8 programs, the fourth level on 16 programs, and the fifth level on 32 programs) as shown in Fig. 2.

To find the sequence of optimization passes for each program in training set, the similarity is calculated between its feature (before applying any pass) and other program features. Then, two of the most similar programs (K = 2) are chosen to the original one and add their passes to the optimization sequence (SequencArray as shown in Algorithm 1). Each one of chosen program is considered as a new program, then it classified again to the nearest two neighbor programs in the tree by using its features (after applying the pass). Another two similar programs are chosen where their execution time after applying optimization passes represents the lowest one. This process will be repeated until all the programs in the training set are chosen.

3.2.1.2 Testing Set

In the testing set, another 20 benchmark programs are fetched as unseen programs. Then, classify each program depending on its features to the closest program in the training set by using KNN algorithm to predict the good optimization sequence for this program, which represents the same optimization sequence for the closest program in the training set.

3.3 Interaction Between Optimization Passes

After finding the optimization sequences for all the benchmark programs in prediction scheme, we noticed that there are some the optimization passes in the generated sequences have a negative impact on program execution time due to the wrong interactions between these passes.

To eliminate the optimization passes that cause the wrong interactions, the proposed approach uses an algorithm called the reduction algorithm. This algorithm inspects the effect of optimization passes on program execution time and check their impact to determine whether it good or bad [1]. Therefore, a good optimization pass that improves program execution time remains in the resulting optimization sequence, but the bad pass that does not improve the program execution time is removed from that sequence.

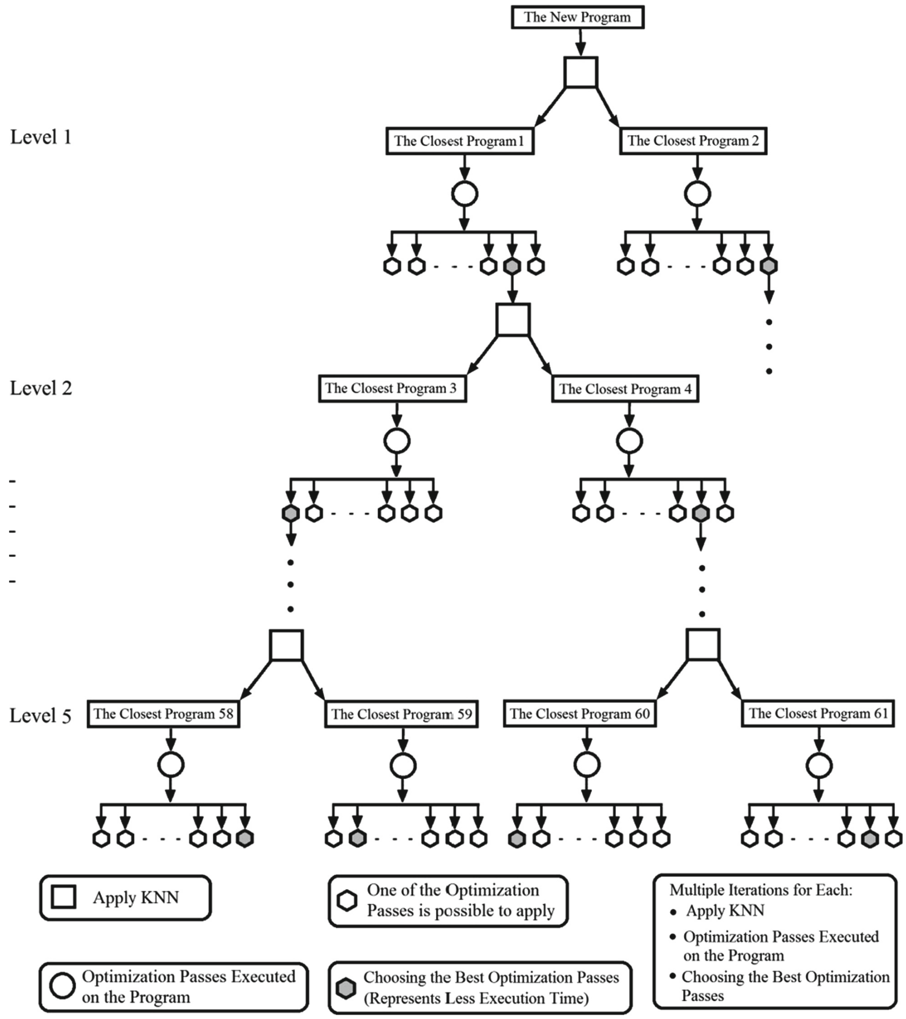
The New Program
Level 1
The Closest Program 1
The Closest Program 2
Level 2
The Closest Program 3
The Closest Program 4
Level 5
The Closest Program 58
The Closest Program 59
The Closest Program 60
The Closest Program 61
Apply KNN
One of the Optimization Passes is possible to apply
Multiple Iterations for Each:
• Apply KNN
• Optimization Passes Executed on the Program
• Choosing the Best Optimization Passes
Optimization Passes Executed on the Program
Choosing the Best Optimization Passes (Represents Less Execution Time)

Fig. 2. The scheme of finding optimization sequence

Algorithm 1: Finding Optimization Sequence

Input: benchmark programs, optimization passes.
Output: optimization sequence of passes for each program (*collectionSequencArray)*.

```
1: For i=1 to end of benchmark programs
2:       Features extraction (program before executing optimization passes)
3: End For i
4: For i=1 to end of benchmark programs
5:For j=1 to end of optimization passes
6:               Apply (–scalarrepl pass)
7:               Features extraction (program after executing each optimization pass)
8:      End For j
9: End For i
10: For i=1 to end of benchmark programs
11:New_program←benchmark program
12:     For j  = 1 to length  New_program
13:                 Use KNN to classify New_program[j] for two closest programs by its
                      features.
14:                 Choose best two optimization passesfrom two closest programs.
15:Save two chosenoptimizationpasses in SequencArray.
16:           Save two optimization passes as new programs in Temp_new_programs.
17:End For j
18          New_program = Temp_new_programs
19:While (!End optimization passes) goto step 12
20:  Save SequencArray in collectionSequencArray
21: End For i
22:      Return collectionSequencArray
```

3.3.1 The Reduction Algorithm

After calculating the program execution time for each optimization sequence in the prediction scheme, the reduction algorithm works on each sequence alone. It starts by deleting the first optimization pass of that sequence and calculates the execution time of optimization sequence again. If the current execution time (i.e. Resulting from the use for the reduction algorithm) is less than the previous execution time (i.e. from the original sequence), this algorithm continues in eliminating the other optimization passes from the original one. Otherwise, this algorithm returns this optimization pass to the original optimization sequence and then eliminates the next pass that come after it. This process is repeated for all optimization passes in the optimization sequences. Algorithm 2 illustrates the steps of reduction algorithm.

Algorithm 2:Reduction Optimization Sequence

```
Input:prog_Opt_seq by using KNN.
Output: Best_Opt_seq.
1: Execution time ← execution (prog_Opt_seq)
2: I← 1
3: While I< length of  prog_Opt_seq
5:          Temp_prog_Opt_seq← Delete (prog_Opt_seq, I)
6:          Best_Execution time ← execution(Temp_prog_Opt_seq)
7:     If Best_Execution time < Execution time
8:          Execution time ← Best_Execution time
9:Best_Opt_seq←Temp_prog_Opt_seq
10:I←0
11:End if
12:         I←I+1
13: End  while
14:         Return  Best Opt_seq
```

4 Experimental Results

Our approach was evaluated on Intel (R) Core (TM) i5 processor with 3.7 GB RAM running a Linux - Ubuntu 16.4 operating system.

Each program from the benchmark programs was executed with its own optimization sequence by using the script file. Each program has been executed three times and the average is calculated for them to increase the accuracy of the results.

In this section, the results will be displayed for 10 benchmark programs as a sample of the training set and 10 benchmark programs as a sample of the testing set. Tables 3 and 4 show the optimization sequences obtained for the programs of these samples in our approach.

Figures 3 and 4 show the execution time obtained for the programs shown in Tables 3 and 4.

Table 3. The optimization sequence before and after the reduction process in the training set.

Programs	Optimization sequence before the reduction process	Number of passes	Optimization sequence after the reduction process	Number of passes
Training program1.c	-instcombine -inline -loop-rotate -early-cse -indvars -gvn -basicaa -basiccg -prune-eh -loop-idiom -loop-reduce -jump-threading -tailcallelim -loop-deletion -licm	15	-instcombine -inline -early-cse -indvars -gvn -basicaa -basiccg -prune-eh -loop-idiom -loop-reduce -jump-threading -tailcallelim -loop-deletion -licm	14

(*continued*)

Table 3. (*continued*)

Programs	Optimization sequence before the reduction process	Number of passes	Optimization sequence after the reduction process	Number of passes
Training program2.c	-early-cse -instcombine -basicaa -basiccg -prune-eh -loop-rotate -loop-idiom -gvn -loop-reduce -inline -jump-threading -tailcallelim -indvars -loop-deletion -licm	15	-early-cse -instcombine -basicaa -basiccg -prune-eh -loop-rotate -loop-idiom -gvn -loop-reduce -jump-threading -tailcallelim -indvars -loop-deletion -licm	14
Training program3.c	-prune-eh -loop-rotate -tailcallelim -indvars -early-cse -loop-deletion -loop-idiom -gvn -jump-threading -inline -basicaa -loop-reduce -basiccg -instcombine -licm	15	-prune-eh -tailcallelim -indvars -early-cse -loop-deletion -loop-idiom -gvn -jump-threading -inline -basicaa -loop-reduce -basiccg -instcombine -licm	14
Training program4.c	-basiccg -gvn -early-cse -loop-rotate -prune-eh -inline -basicaa -instcombine -loop-idiom -jump-threading -loop-reduce -tailcallelim -indvars -licm -loop-deletion	15	-basiccg -gvn -early-cse -loop-rotate -prune-eh -basicaa -instcombine -loop-idiom -jump-threading -loop-reduce -tailcallelim -indvars -licm -loop-deletion	14
Training program5.c	-loop-idiom -loop-rotate -basicaa -gvn -early-cse -basiccg -prune-eh -inline -jump-threading -loop-reduce -instcombine -tailcallelim -indvars -licm -loop-deletion	15	-loop-idiom -loop-rotate -basicaa -gvn -early-cse -basiccg -prune-eh -inline -jump-threading -loop-reduce -instcombine -indvars -licm -loop-deletion	14
Training program6.c	-tailcallelim -inline -prune-eh -indvars -gvn -jump-threading -early-cse -loop-rotate -loop-deletion -loop-idiom -basicaa -loop-reduce -basiccg -instcombine -licm	15	-tailcallelim -inline -prune-eh -indvars -gvn -jump-threading -early-cse -loop-rotate -loop-deletion -loop-idiom -basicaa -loop-reduce -basiccg -instcombine -licm	15

(*continued*)

Table 3. (*continued*)

Programs	Optimization sequence before the reduction process	Number of passes	Optimization sequence after the reduction process	Number of passes
Training program7.c	-loop-rotate -gvn -prune-eh -early-cse -jump-threading -loop-idiom -basicaa -inline -basiccg -tailcallelim -indvars -loop-deletion -loop-reduce -instcombine -licm	15	-loop-rotate -gvn -prune-eh -basicaa -inline -basiccg -tailcallelim -indvars -loop-deletion -loop-reduce -instcombine -licm	12
Training program8.c	-inline -tailcallelim -loop-rotate -basicaa -gvn -loop-reduce -instcombine -early-cse -loop-idiom -prune-eh -basiccg -jump-threading -indvars -loop-deletion -licm	15	-inline -tailcallelim -loop-rotate -basicaa -gvn -loop-reduce -early-cse -loop-idiom -basiccg -indvars -loop-deletion -licm	12
Training program9.c	-loop-idiom -loop-rotate -early-cse -gvn -inline -basiccg -prune-eh -jump-threading -loop-reduce -tailcallelim -basicaa -instcombine -indvars -loop-deletion -licm	15	-loop-idiom -loop-rotate -early-cse -gvn -inline -basiccg -prune-eh -jump-threading -loop-reduce -tailcallelim -basicaa -instcombine -indvars -loop-deletion -licm	15
Training program10.c	-loop-reduce -gvn -early-cse -loop-idiom -basicaa -loop-rotate -inline -prune-eh -basiccg -jump-threading -tailcallelim -instcombine -indvars -loop-deletion -licm	15	-loop-reduce -gvn -early-cse -loop-idiom -basicaa -loop-rotate -inline -prune-eh -basiccg -jump-threading -tailcallelim -instcombine -indvars -loop-deletion –licm	15

According to the Tables 3 and 4, there are some programs that do not apply the reduction process on their optimization sequences because there are no wrong interactions in passes of those optimization sequences when applying the reduction algorithm. Therefore, these optimization sequences remain the same after the reduction process for those programs.

The results mentioned in Figs. 3 and 4 showed the comparison at the execution time when using both the KNN algorithm and the KNN algorithm with the reduction algorithm in the prediction to the standard optimization level -O2.

These results indicate that our approach outperforms standard optimization level-O2 by improving the execution time. This though building a prediction scheme to find a good sequence of optimization passes for each program from the benchmark programs

Table 4. The optimization sequence before and after the reduction process in the testing set.

Programs	Optimization sequence before the reduction process	Number of passes	Optimization sequence after the reduction process	Number of passes
Testing program1.c	-prune-eh -early-cse -loop-rotate -loop-idiom -basicaa -basiccg -gvn -inline -jump-threading -loop-reduce -tailcallelim -instcombine -indvars -loop-deletion -licm	15	-prune-eh -early-cse -loop-rotate -loop-idiom -basicaa -basiccg -gvn -inline -jump-threading -loop-reduce -tailcallelim -instcombine -indvars -loop-deletion -licm	15
Testing program2.c	-early-cse -inline -prune-eh -loop-idiom -loop-rotate -basicaa -gvn -jump-threading -basiccg -loop-reduce -instcombine -tailcallelim -indvars -loop-deletion -licm	15	-early-cse -inline -prune-eh -loop-idiom -loop-rotate -basicaa -gvn -basiccg -instcombine -indvars -loop-deletion	11
Testing program3.c	-loop-rotate -basicaa -early-cse -gvn -loop-idiom -inline -prune-eh -basiccg -jump-threading -loop-reduce -tailcallelim -instcombine -indvars -loop-deletion -licm	15	-loop-rotate -basicaa -early-cse -gvn -loop-idiom -inline -prune-eh -basiccg -jump-threading -loop-reduce -tailcallelim -instcombine -indvars -loop-deletion -licm	15
Testing program4.c	-loop-reduce -indvars -early-cse -loop-idiom -loop-rotate -inline -prune-eh -basicaa -basiccg -gvn -instcombine -tailcallelim -jump-threading -licm -loop-deletion	15	-loop-reduce -indvars -early-cse -loop-idiom -loop-rotate -inline -prune-eh -basicaa -basiccg -gvn -instcombine -tailcallelim -jump-threading -licm -loop-deletion	15

(*continued*)

Table 4. (*continued*)

Programs	Optimization sequence before the reduction process	Number of passes	Optimization sequence after the reduction process	Number of passes
Testing program5.c	-loop-idiom -loop-rotate -early-cse -gvn -inline -basiccg -prune-eh -jump-threading -loop-reduce -tailcallelim -basicaa -instcombine -indvars -loop-deletion -licm	15	-loop-idiom -loop-rotate -early-cse -gvn -inline -basiccg -prune-eh -jump-threading -loop-reduce -tailcallelim -basicaa -instcombine -indvars -loop-deletion -licm	15
Testing program6.c	-prune-eh -loop-rotate -basicaa -inline -gvn -early-cse -loop-idiom -basiccg -jump-threading -loop-reduce -tailcallelim -instcombine -indvars -loop-deletion -licm	15	-prune-eh -loop-rotate -basicaa -inline -gvn -early-cse -loop-idiom -basiccg -jump-threading -loop-reduce -tailcallelim -instcombine -loop-deletion -licm	14
Testing program7.c	-loop-reduce -loop-rotate -early-cse -loop-idiom -basicaa -inline -prune-eh -basiccg -jump-threading -gvn -tailcallelim -instcombine -indvars -loop-deletion -licm	15	-loop-reduce -loop-rotate -early-cse -loop-idiom -basicaa -prune-eh -basiccg -gvn -tailcallelim -instcombine -indvars -loop-deletion -licm	13
Testing program8.c	-loop-rotate -basicaa -early-cse -gvn -loop-idiom -inline -prune-eh -basiccg -jump-threading -loop-reduce -tailcallelim -instcombine -indvars -loop-deletion -licm	15	-loop-rotate -basicaa -early-cse -gvn -loop-idiom -inline -prune-eh -basiccg -jump-threading -loop-reduce -tailcallelim -instcombine -indvars -loop-deletion -licm	15

(*continued*)

Table 4. (*continued*)

Programs	Optimization sequence before the reduction process	Number of passes	Optimization sequence after the reduction process	Number of passes
Testing program9.c	-loop-reduce -loop-rotate -early-cse -loop-idiom -basicaa -inline -prune-eh -basiccg -jump-threading -gvn –tailcallelim -instcombine -indvars -loop-deletion -licm	15	-loop-reduce -loop-rotate -early-cse -loop-idiom -basicaa -prune-eh -basiccg -gvn -tailcallelim -instcombine -indvars -loop-deletion -licm	13
Testing program10.c	-gvn -early-cse -loop-rotate -basicaa -loop-idiom -inline -prune-eh -basiccg -jump-threading -tailcallelim -instcombine -loop-reduce -indvars -loop-deletion -licm	15	-early-cse -loop-rotate -basicaa -loop-idiom -inline -prune-eh -basiccg -jump-threading -tailcallelim -instcombine -loop-reduce -indvars -loop-deletion -licm	14

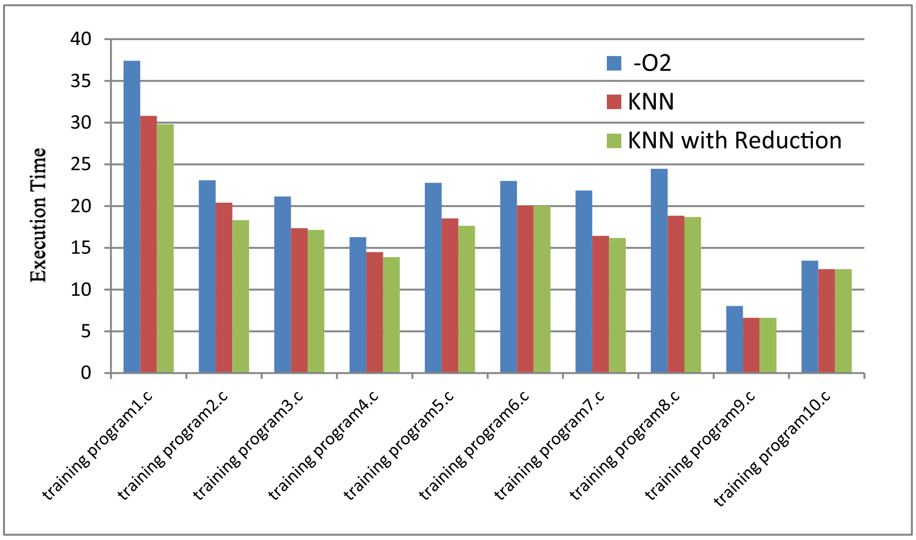

Fig. 3. The performance comparison between our approach and -O2 for the training set.

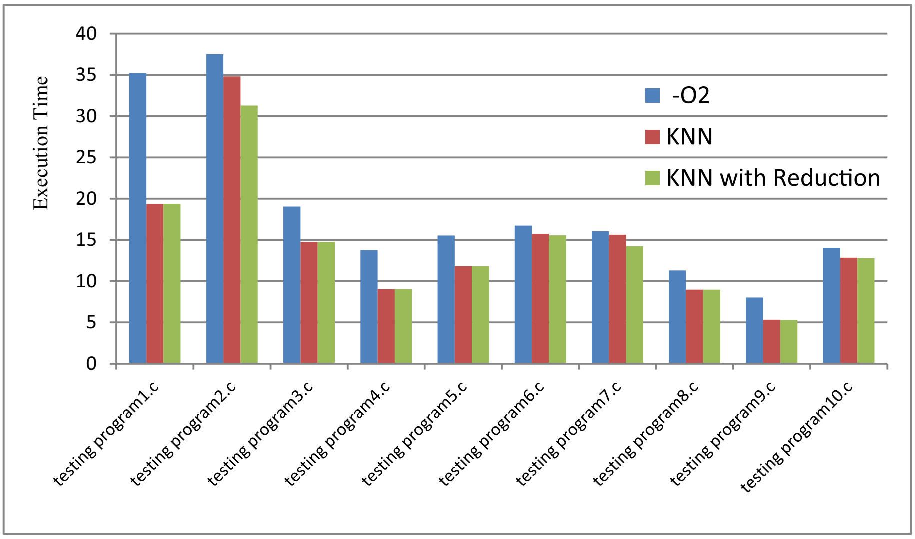

Fig. 4. The performance comparison between our approach and -O2 for the testing set.

based on its features using KNN algorithm. The results of application only KNN show an improvement ratio of 21% on average in the execution time. There is also an increase in improving the execution time when applying the reduction algorithm on the sequences of optimization passes by 23% on average of the execution time compared with KNN without reduction.

5 Conclusion

This work presented machine learning to build a framework prediction scheme for compiler optimization sequence. This scheme provided auto tuning to solve the problems resulting from manual tuning. The KNN algorithm was used in the prediction scheme to find the sequence of optimization passes for each program from the benchmark programs, and then order these passes based on the features of that program. The reduction algorithm was also applied to eliminate passes that have a negative impact on program execution time for the resulting sequence by using KNN algorithm.

LLVM infrastructure has been used to validate the proposed method. The experiment results showed that the proposed approach outperforms LLVM -O2 by improving the execution time by average of 21% when building a prediction scheme using KNN algorithm. Moreover, it improved the execution time byan average of 23% after applying reduction algorithm a long side to the KNN algorithm.

References

1. Purini, S., Jain, L.: Finding good optimization sequences covering program space. ACM Trans. Archit. Code Opt. (TACO) **9**(4), 56 (2013)

2. Ashouri, A.H., Bignoli, A., Palermo, G., Silvano, C., Kulkarni, S., Cavazos, J.: MiCOMP: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning. ACM Trans. Archit. Code Opt. (TACO) **14**(3), 29 (2017)
3. Amra, I.A.A., Maghari, A.Y.: Students performance prediction using KNN and Naïve Bayesian. In: 2017 8th International Conference on Information Technology (ICIT), pp. 909-913. IEEE, May 2017
4. Martins, L.G., Nobre, R., Cardoso, J.M., Delbem, A.C., Marques, E.: Clustering-based selection for the exploration of compiler optimization sequences. ACM Trans. Archit. Code Opt. (TACO) **13**(1), 8 (2016)
5. de Souza Xavier, T.C., da Silva, A.F.: Exploration of compiler optimization sequences using a hybrid approach. Comput. Inform. **37**(1), 165–185 (2018)
6. Cavazos, J., Fursin, G., Agakov, F., Bonilla, E., O'Boyle, M.F., Temam, O.: Rapidly selecting good compiler optimizations using performance counters. In: International Symposium on Code Generation and Optimization (CGO 2007), pp. 185–197. IEEE, March 2007
7. Ashouri, A.H., Mariani, G., Palermo, G., Silvano, C.: A Bayesian network approach for compiler auto-tuning for embedded processors. In: 2014 IEEE 12th Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia), pp. 90–97. IEEE, October 2014
8. Kulkarni, S., Cavazos, J.: Mitigating the compiler optimization phase-ordering problem using machine learning. In: ACM SIGPLAN Notices, vol. 47, no. 10, pp. 147–162. ACM, October 2012
9. Tan, P.N., Steinbach, M., Kumar, V., et al.: Introduction to Data Mining (2006)
10. Suguna, N., Thanushkodi, K.: An improved k-nearest neighbor classification using genetic algorithm. Int. J. Comput. Sci. Issues **7**(2), 18–21 (2010)
11. Pan, Z., Eigenmann, R.: Fast and effective orchestration of compiler optimizations for automatic performance tuning. In: Proceedings of the International Symposium on Code Generation and Optimization, pp. 319–332. IEEE Computer Society, March 2006
12. Alkaaby, Z.S., Alwan, E.H., Fanfakh, A.B.M.: Finding a good global sequence using multi-level genetic algorithm. J. Eng. Appl. Sci. **13**, 9777–9783 (2018)
13. kiran_c: Counting Instructions with LLVM. https://kiran-c.livejournal.com/7956.html
14. Criswell, J.: Need some feed back. https://marc.info/?l=cfe-dev&m=130935922030121&w=2
15. Junior, N.L.Q., Rodriguez, L.G.A., da Silva, A.F.: Combining machine learning with a genetic algorithm to find good complier optimizations sequences. In: ICEIS, vol. 1, pp. pp. 397–404 (2017)
16. Almohammed, M.H., Alwan, E.H., Fanfakh, A.B.M.: Programs features clustering to find optimization sequence using genetic algorithm. In: Jain, Lakhmi C., Peng, S.-L., Alhadidi, B., Pal, S. (eds.) ICICCT 2019. LAIS, vol. 9, pp. 40–50. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-38501-9_4