Ingénierie des Systèmes d'Information
Vol. 29, No. 4, August, 2024, pp. 1611-1617
Journal homepage: http://iieta.org/journals/isi

Candidate Best Optimizations Sequences for Code Size Reduction

Esraa H. Alwan[1*], Ali Kadhum M. Al-Qurabat[2]

[1] Department of Computer Science, College of Science for Women, University of Babylon, Babylon 51002, Iraq
[2] Department of Cyber Security, College of Sciences, Al-Mustaqbal University, Babylon 51001, Iraq

Corresponding Author Email: esraa.hadi@uobabylon.edu.iq

https://doi.org/10.18280/isi.290434

Received: 29 May 2024
Revised: 10 June 2024
Accepted: 30 July 2024
Available online: 21 August 2024

Keywords:
code size reduction, optimization sequence, LLVM

ABSTRACT

Recently, the number of smaller and smarter embedded devices have rapidly increased. This increment puts more pressure on the compiler developer to develop more dedicated application programs for these devices. Modern compilers (like LLVM) offer standard optimization levels (flags) that deal with reducing the code size named Os and Oz flags. The question arise in this paper in is: Is it possible to find sequence that deliver smaller code compare to standard flags? A Sign Table, it is the suggested method that is introduced in this paper. It can suggest an optimization sequence that can reduce the code size for set of unseen program. Initially, two thousand optimization sequences are generated randomly. Each sequence is compiled with 50 programs, where the programs that give smaller code size compared with the Os or Oz flags are extracted. After building the signs table, which contains the sequences that give the average programs sizes smaller than the Os or Oz flags, the process of quantifying similarity between the unseen program and the programs contained within the signs table is performed. The sequences that belong to the most similar programs are selected to compile the unseen program. The proposed methodology is assessed through an empirical investigation, employing three benchmark suites, namely PolyBench, Shootout, and Stanford. The experiments show that the proposed method reduces the unseen program size by about 9% compared standsrd optimization flags.

1. INTRODUCTION

Recent compilers contain various optimizations, which can frequently be manually switched on or off via compiler flags or switches. With so many optimizations available to compilers, it's practically hard to choose which is best for a single program. Traditionally, compiler developers have operated under one of two assumptions: either a predefined optimization sequence is "good enough" for all programs, or supplying users with an extensive collection of optimization flags is sufficient, so putting the burden to the user [1-3].

An optimization sequence is a collection of optimization passes that can be applied on the program's Intermediate Representation (IR). Choosing the right optimization pass and the right order for given application is called phase-ordering problem. The set of all optimization sequences is known as the optimization sequence space, and it is incessantly large. To explain the phase-ordering problem, let us establish a boolean vector o, with components oi representing the various compiler optimizations. Each optimization oi can be activated (oi=1) or disabled (oi=0). The vector o depicts a phase-ordering compiler optimization sequence in the n-dimensional factorial space |Ophases| = n!, where n is the number of optimizations. However, the provided limit is for a made easier phase-ordering problem with a constant vector length and lack of repetitions. Allowing repetition and dynamic duration broadens the design possibilities to:

$$|Ophases\text{-}repetition| = \sum_{i=0}^{m} n^{i} \ldots \quad (1)$$

where, n is the number of optimizations being investigated, and m is the maximum length for the optimization sequence. For instance, with n optimizations and m=10, |Ophases repetition| will result in more than 11 billion various combinations for each application [4]. Many default optimization levels are provided by compilers, e.g., O1, O2, O3, Oz Some of these optimization levels O2, O3 focus on enhancing the program execution time, while the Os, Oz are dedicate to dealing with reducing code size [5-7]. Through the execution of the program, all the optimization flags will be turned off by default, and the expert can turn some or all of them on according to the program's needs. For example, the GCC has more than 200 passes while the LLVM Clang and opt have more than 100 passes [8-10]. The sequence of these passes called the optimization sequence. Choosing the right sequence can help the programmers achieve better performance [11, 12].

Clearly, some optimizations can increase the code size, like loop unrolling or procedure inlining, while others can decrease it, for example, dead code removal or strength reduction [13, 14].There are two reasons that arise to answer the question of "why is there a need to turn off some optimization passes instead of applying all of them ". The first is, the characteristics of some passes might not match with the characteristics of the program. The second reason is that some of these passes have

a negative impact on the quality of the resulting program [15-18]. Thus, instead of speeding up the program or decreasing the program size, the reverse may occur. An illustrative example involves the employment of inlining, a technique that substitutes a function invocation with its corresponding body. This particular transformation has the potential to cause a phenomenon known as code expansion, subsequently increasing the potential for cache misutilization [19, 20]. The execution time of the generated code is beyond the scope of this paper. This paper is primarily focus on the code size, and the resulting code may have a reduced size. The rest of the paper is organized as follows: Section 2 introduces some of the related works. Section 3 details the suggested method and explains it is implementation. In Section 4, the experimental results obtained from the proposed methods are presented. Finally, Section 5 summarizes the findings of suggested method.

2. RELATED WORKS

This section outlines prior research that highlight the efficacy of compilers in minimizing code size. In particular, Kim et al. [19] aimed to address the challenges posed by resource limitations in embedded systems, precisely the overhead attributed to program memory. The authors established that fine-grained function inlining is an effective approach to optimize program memory utilization. Jain et al. [20] divided the Os optimization level into seven logical groups and studied the effect of each group on reducing the code size of a set of benchmark programs. Moreover, the combination of these groups is also examined. This work enables users to customize their own series to get the desired program sizers. Cooper et al. [21] presented GA to find compiler optimization sequences that can reduce the code size. They used 10 optimization passes and they were able to produce a fixed sequence that dramatically reduced static code size compared with no optimization. Foleiss et al. [22] explore how generating a small code size can be affected by choosing a combination of compiler optimization techniques, especially in a resource-poor environment such as a WSN. Through using data mining techniques and a comprehensive compilation process, they found many combinations of compiler optimization that could reduce code size in many cases. Therefore, the optimization technique that may be used in tandem to reduce the code size it's possible to determine. An iterative approach to reducing the number of generated instructions in assembly code is proposed by Haneda et al. [23]. Depending on the Mann-Whitney test, non-parametric inferential statistics, the decision is made about which compiler options should be on or off. The results show that this technique outperforms the Os in reducing the code size in almost all cases.

3. SUGGESTED METHOD

The present study introduces a methodology for identifying the most optimal optimization sequence that yields the most efficient program size. Initially, a dataset is built which contain randomly generated optimization sequences. Subsequently, all of the generated sequences are applied on 50 programs. The block diagram of the proposed method is illustrated in Figure 1. Furthermore, algorithm 1 illustrate the steps of the suggested method, which will be explain in detail later.

3.1 Construction dataset

3.1.1 Generating optimization sequences

Random optimization sequences are generated, where more than 2 thousand optimization sequences are used to compile a set of programs. These optimization sequences have fixed lengths not to exceed a maximum of 60 passes. Table 1 provides the standard optimization levels of Os and Oz.

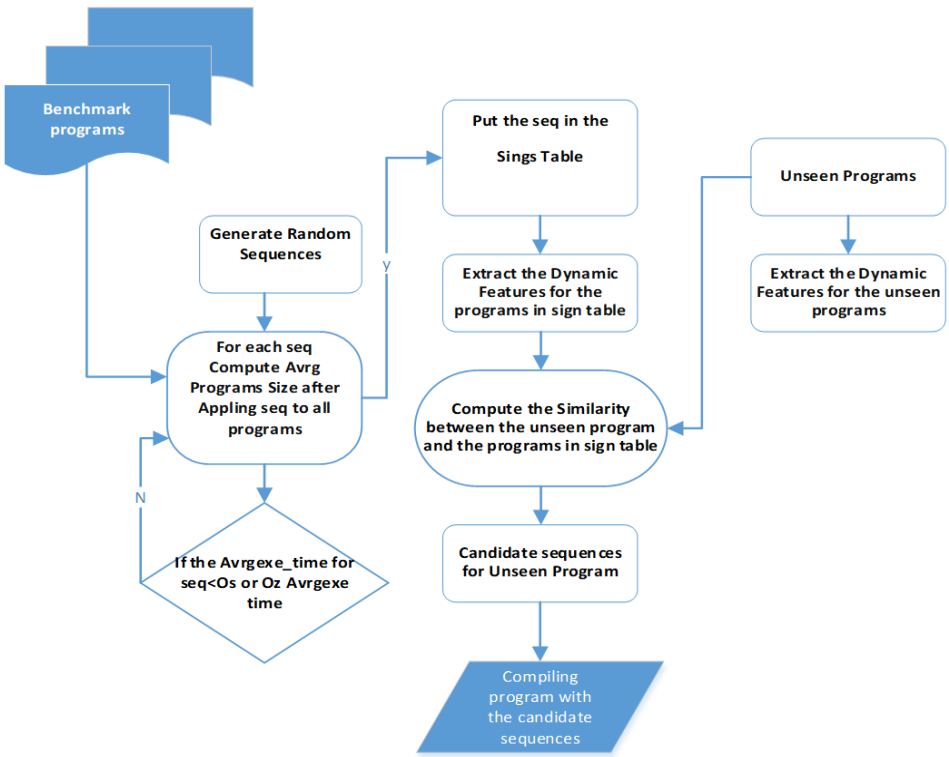

Figure 1. The block diagram of the proposed method

3.1.2 Compute program size

A wide range of different programs, 50 programs, are extracted from three benchmark suites: PolyBench, Shootout, and Stanford. The programs' size is computed in which each program (p) is executed with all optimization sequences that are generated prviously. Moreover, these programs are compiled using Os and Oz flags and their size are computed.

Table 1. Os and Oz optimization passes [24, 25]

List of Optimization Passes		
-domtree	-lcssa-verification	-early-cse-
-inline	-loop-acicesses	memsa
-scalairzer	-globaldce	-rpo-
-calld- value-	-loop-load-elim	funtioniattrs
propagaton	-inferattrs	-prune-eh
-tti	-div-rem-pairs	-ipsccp
-assmption-cache-	-porfile-sumary	-globldce
tracker	-tbaa	-indvars
-opt-remark-emitter	-libcalls-shrinkwrp	-forcattrs
-lazy-block-freq	-jump-threading	-mem2eg
-block-freq	-globals-aa	-globalopt
-instsimplify	-targetlibinfo	-tailcallelim
-loop-unswitch	-scalar-evolution	-pgo-memop-
-licm	-basiccg	opt
-simplifycfg	-loop-simplify	Reassociate
-memoryssa	-speculative-	-basicaa
-loop-rotate	execution	-lazy-block-freq
-callsite-splitting	-loops	-instcombine
-aa	-sroa	
-demended-bits		
-loop-unroll		

3.2 Extracting the selective sequences

According to the previous stage, the average of programs size obtained by compiling these programs with seq (SI) compared to the average programs size obtained by compiling the identical programs with the Os and Oz flags. The sequence with average program sizes less than the one with Os or Oz flag adds to the singed table. This procedure carried out for every sequence.

3.3 Construction sings table

Each program is compiled with the Os and Oz flags, and their execution times are stored in the Sign table. Subsequently each one of the two thousand sequences is used to compile all the programs. The sequences that result in an average size for all programs smaller than those created by Os or Oz are recognized and included into the Sign table as shown in Figure 2.

The first raw of this table represents the extracted sequences and the first column represents the programs' names. The entry to this table is a sign to give us an indicator if the i-th sequence reduces the program size or no. Finally, the files name of dynamic features for these programs are also added to the Sign Table.

3.4 Compute the similarity

For unseen programs (new programs) and the programs in the Signs Table, their dynamic features are extracted. Perf/Linux tool is used to extract these features (36 events). Performance counter events are used to reflect the program behavior. Table 2 illustrates these events. The process of quantifying similarity between the unobserved program and the programs contained within the signs table is undertaken. The sequences corresponding to the most similar programs are selected as potential candidates to optimize the unseen program.

Algorithm 1
Input: number of sequences N, number of programs M, unseen program
Output: Candidate best program size for unseen program
Begin
Step 1: Generate random sequence
In this step we generate more than 2000 random sequences

Step 2: Extract the best sequences for each program M and put these sequences in
the signs table
#Intilise set of varibles
Prog_size=0, Prog_size_Oz
Total 1=0, Total 2 =0
for seqi = 1 to N
for prgj =1 to M
Prog_size_seq= Compile the program progj with seqi and extract size
Total 1=total1+ Prog_size_seq
Prog_size_Oz= Compile the program progj with Oz flag and extract size
Prog_size_Os= Compile the program progj with Os flag and extract size
Total 2=total2+ Prog_size_Oz
Total 3=total3+ Prog_size_Os
end
# computer the average for Total1 and Total 2
if ((average total 1 < average total 2) or (average total 1 < average total 3))
Put the seq in the sing table
end
Step 3: Candidate sequence for unseen program
Extract features for all programs in the sign table using Perf tool.
Extract features for unseen program using Perf Tool.
Compute similarity between the unseen program and all programs in the sign table.
Choose the sequences that belong to most similar Program.
Compile the unseen program with chosen sequences and select the one lower program size.
End

Multiple metrics are available for the quantification of similarity. One commonly employed approach is the utilization of the cosine metric, as illustrated through the equation presented herein.

$$\text{Sim(p, pi)}=\frac{\sum_{w=1}^{m} pw*piw}{\sqrt{\sum_{w=1}^{m} pw^2}*\sqrt{\sum_{w=1}^{m} piw^2}} \quad (2)$$

where, the symbol 'p' denotes the unseen program, which refer to a program yet to be observed, while 'pi' signifies a set of supplementary programs, herein referred to as training programs [26, 27]. The new program that is yet to be observed, its level of similarity is determined by extracting its dynamic

features and evaluating them against those of all other programs, in the signs table.

Table 2. Perf events [1, 28]

List of the Features used in Our Approach	
LLC-loads	L1-dcache-stores
LLC-load-misses	cpu-cycles
L1-dcache-loads	bus-cycles
L1-dcache-load-misses	ref-cycles
cache-misses	page-faults
cache-references	context-switches
dTLB-loads	cpu-migrations
LLC-stores	minor-faults
LLC-store-misses	major-faults
dTLB-loads	alignment-faults
dTLB-load-misses	emulation-faults
iTLB-loads	cpu-clock
iTLB-load-misses	task-clock
dTLB-store	mem-loads
dTLB-store-misses	mem-stores
itlb_misses.walk_completed	branch-instructions
branch-misses	LLC-prefech-miss
instructions	
L1-dcache-stores-misses	

3.5 Candidates sequences for unseen program

After computing the similarity between the unseen program and the programs in the signs table, the most similar programs are selected. Then, the unseen program is compiled using the list of sequences that belong to the selected programs. Finally, sequences that produces lower the code size are chosen.

4. EXPERIMENTAL OUTCOMES

This section discusses the findings of the proposed method. The case study includes programs from various benchmarks site. The proposed technique is evaluated using the LLVM compiler's standard optimization level Os and Oz. In the bgginning, to acquire precise outcomes, it is necessary to compile the programs using Clang's O0 level, signifying that no optimizations are applied, prior to executing any optimization sequences. Subsequently, to activate additional transformation passes, the *"-scalarrepl*" (scalar replacement) is applied in conjunction with each pass.

After converting the program to LLVM Intermediate Representation (IR), which is a machine-readable bit code (.bc) file format, the optimization sequences (generated sequences) are applied to it. The source code programs are converted to the bit code file format using the Clang compiler, which is an LLVM C language frontend. Two useful tools named opt and llc are used to compile the programs. The optimization tool, "opt", performs a sequence of passes to optimize the source code program and subsequently stores the optimized code in a bit file format. The tool llc produces the object code by converting the bit file format. Finally, Clang is used to generate the executable code.

Table 3 shows some of the details that evaluate the machine used in the proposed approach.

4.1 Sings table

The main core of our work is the Sings Table. It represent the reference to candidate the right for unseen program. Table 4 below illustrates a small example of this table. Where the raw represents the sequence name and the column represents the program name. The ✓ sign means these sequences improve the program size.

Table 3. Machine details

Processor Model	Intel (R) Core (TM) i5 CPU
Processor Speed	2.5 GHz
RAM	4 GB
Operating System	Linux Ubuntu 16.4
Compiler	Clang/LLVM

Table 4. Signs table

Program Name	Seq1	Seq2	Seq3	Seq4	Seq5
Heapsort.c	✓		✓		
Matrix.c	✓	✓		✓	
n-body.c					
Sieve.c		✓		✓	
Flops_7.c	✓	✓			✓
Quicksort.c		✓	✓		✓
Perlin.c	✓		✓		✓
Perm.c		✓		✓	
Mvt.c	✓				✓
Oscar.c		✓	✓		✓

4.2 Unseen programs results

Within the scope of these empirical investigations, a set of eight unobserved programs is employed for the purpose of validating the method. Subsequently, the process of computing the similarity between the aforementioned unseen program and the programs contained within the signs table is performed. Then, the sequences that belong to the most similar program are used to compile the unseen program. Table 5 illustrate the best programs sizes that we get with different optimization sequences for eight unseen programs.

Table 5. Programs size

Programs Name	Best Three Sequences That Give Best Program Size (Byte)			Program Size with Os and Oz Flags (Byte)	
				Os	Oz
Sic.c	Seq1 11160	Seq2 11144	Seq3 11080	11208	11208
Mds.c	Seq1 12464	Seq5 12400	Seq6 12400	12568	12568
2mm.c	Seq7 8022	Seq1 7584	Same	8232	8232
Rc4.c	Seq2 8784	Seq6 8752	Same	9928	9928
Corcol.c	Seq10 9992	Seq11 9952	Seq12 9784	10168	10000
Des.c	Seq8 28032	Seq9 27920	Seq1 27824	28328	28328
Symm.c	Seq7 9440	Seq11 9432	Seq4 9392	9496	9496
Pc1cod.c	Seq3 9936	Seq7 9912	Seq5 9848	9960	9960

Figures 2-9 below illustrate the improvement in the program size for 8 unseen programs compared with the program size for the same unseen when compared with Oz flag.

As you can see from the above figures, the average rate of the improvement in the code size is about 9% with different optimization sequences. Furthermore, the obtained findings

reveal that most of the unseen programs their sizes reduce after applying candidate sequence. Particularly, the rc4.c program receives the most significant reduction, which is about 11.8 %. Table 6 illustrates the most powerful extract sequences that have best impact on the unseen programs.

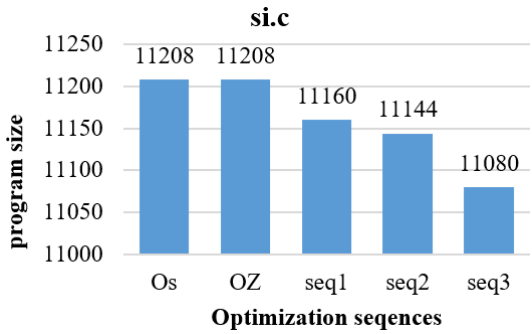

Figure 2. si.c optimized program with different sequences

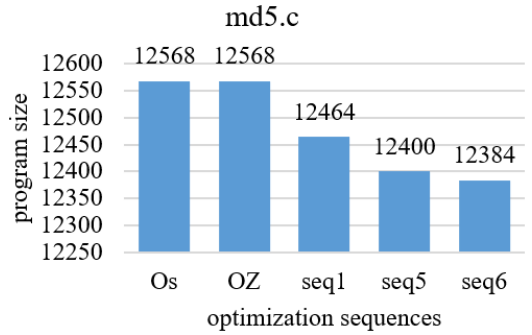

Figure 3. md5.c optimized program with different sequences

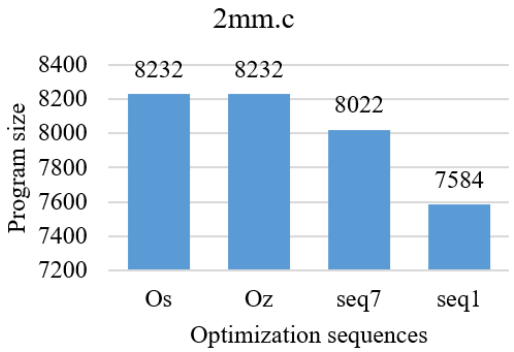

Figure 4. 2mm.c optimized program size with different sequences

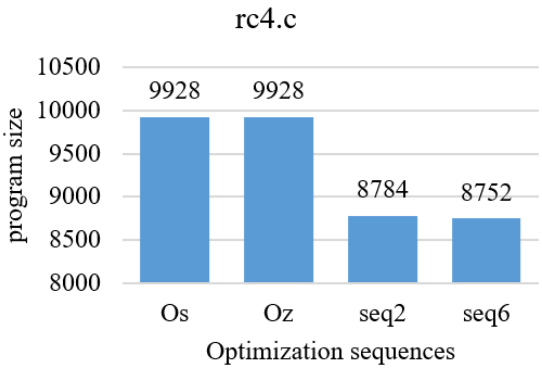

Figure 5. rc4.c optimized program size with different sequences

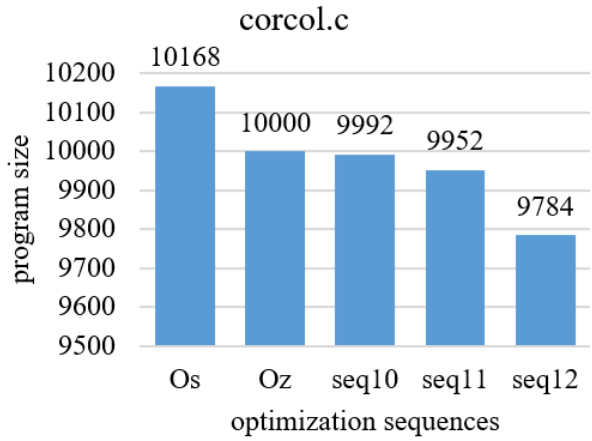

Figure 6. corcol.c optimized program size with different sequences

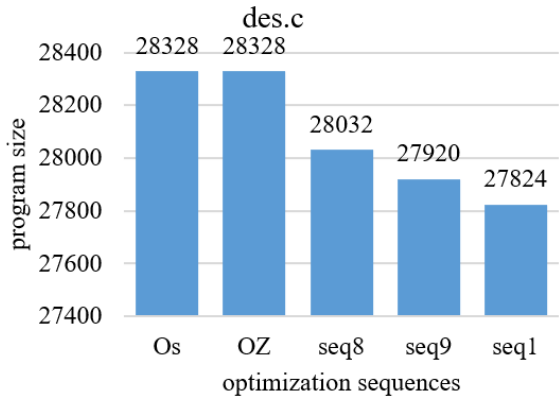

Figure 7. des.c optimized program with different sequences

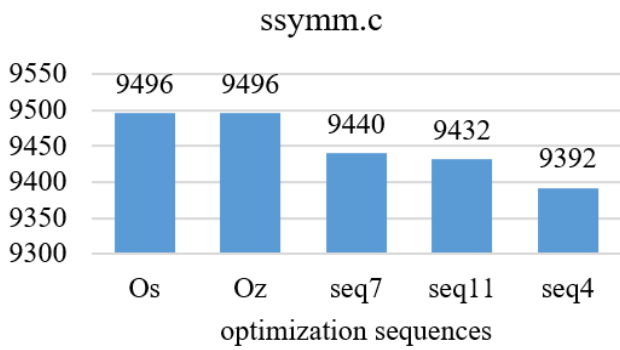

Figure 8. ssymm.c optimized code size with different sequences

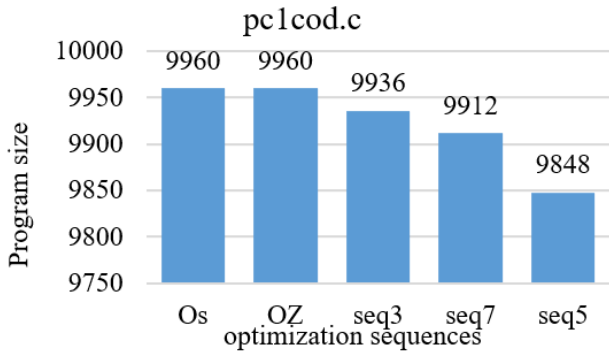

Figure 9. pc1cod.c optimization program with different sequences

Table 6. The extract sequences that have best impact on the unseen programs

Seq1	-reassociate -scalarrepl -loweratomic -dce -strip-dead-prototypes -early-cse -loop-deletion -scalarrepl-ssa -lcssa -reassociate -jump-threading -correlated-propagation -instsimplify -lowerswitch -globaldce -sccp -gvn -scalar-evolution -indvars -strip-dead-prototypes -lazy-value-info -ipsccp -mergefunc -memcpyopt -gvn -basicaa -scalarrepl-ssa
Seq2	-oop-idiom -simplifycfg -loop-unswitch -memcpyopt -functionattrs -no-aa -lowerswitch -lower-expect -loop-rotate -reassociate -correlated-propagation -simplifycfg -mergefunc -indvars –memcpyopt
Seq3	-argpromotion -dse -strip-dead-prototypes -tailcallelim -scalarrepl-ssa -constmerge -early-cse -lazy-value-info -tbaa -mergefunc -loops -domtree -loop-idiom -loweratomic -early-cse -globaldce -globalopt -loop-deletion -ipconstprop -loops -loop-idiom -lcssa -constprop -ipconstprop -domtree -ipsccp -no-aa -early-cse -lowerinvoke -instsimplify -early-cse -lowerswitch -globalopt -targetlibinfo -die -loop-reduce -lowerswitch -lazy-value-info
Seq4	-die -loop-idiom -strip-dead-prototypes -ipsccp -globaldce -loop-deletion -tbaa -adce -early-cse -licm -mergereturn -sccp -dce -lcssa -gvn -die -loop-reduce -lower-expect -correlated-propagation -correlated-propagation -argpromotion -ipsccp -scalarrepl-ssa -sink -dce -always-inline -globaldce -sink -memcpyopt -prune-eh -tailcallelim -loop-unroll -constprop -mergereturn -ipsccp -dse -simplifycfg -loop-idiom -loop-rotate -memcpyopt -constmerge -die -correlated-propagation -early-cse -indvars -always-inline -lazy-value-info -prune-eh -loops -targetlibinfo -strip-dead-prototypes -basiccg -domtree -constprop -constmerge -lazy-value-info -ipsccp -sink -early-cse
Seq5	-argpromotion -dse -strip-dead-prototypes -tailcallelim -scalarrepl-ssa -constmerge -instsimplify -early-cse -lazy-value-info -tbaa -mergefunc -loops -gvn -domtree -loop-idiom -loweratomic -gvn -early-cse -globaldce -globalopt -loop-deletion -ipconstprop -loops -loop-idiom -lcssa -constprop -ipconstprop -domtree -ipsccp -scalar-evolution -no-aa -early-cse -lowerinvoke -instsimplify -early-cse -lowerswitch -globalopt -targetlibinfo -die -loop-reduce -lowerswitch -lazy-value-info -sccp -constmerge -functionattrs -basicaa -ipconstprop -lower-expect -lowerinvoke -targetlibinfo -argpromotion -die -ipconstprop -dce -targetlibinfo -loop-deletion -scalar-evolution –tbaa
Seq6	-dse -scalarrepl -loop-idiom -loop-deletion -loop-instsimplify -memcpyopt -memdep -memcpyopt -deadargelim -loop-simplify -loop-instsimplify -scalarrepl-ssa -constmerge -loop-deletion -basiccg -constprop -argpromotion -argpromotion -mergereturn -lazy-value-info -loops -memcpyopt -lower-expect -ipsccp -dse -tailcallelim -mergereturn -loop-deletion -jump-threading -sink -jump-threading -loop-idiom -loop-deletion -no-aa -mergereturn -argpromotion -lowerinvoke -loop-deletion -ipconstprop -constmerge -no-aa -functionattrs -globalopt -functionattrs -basicaa -die -scalarrepl-ssa -loops -dce -no-aa -ipconstprop -mergereturn -tailcallelim -scalarrepl -adce -basicaa -mergereturn –dse
Seq7	-functionattrs -domtree -loop-rotate -lazy-value-info -instsimplify -lowerinvoke -strip-dead-prototypes -memdep -ipconstprop -adce -always-inline -sccp -functionattrs -memdep -gvn -lazy-value-info -die -globaldce -loweratomic -gvn -loweratomic -mergefunc -die -loop-unroll -basicaa -ipconstprop -loop-instsimplify -loops -scalar-evolution -tbaa -loops -jump-threading -prune-eh -ipsccp -globalopt -instsimplify -loop-idiom -codegenprepare -ipconstprop -loop-simplify -tbaa -lcssa -lowerswitch -constmerge -globaldce -loop-simplify -die -loop-deletion -tbaa -codegenprepare -functionattrs -strip-dead-prototypes -globalopt -sccp -loop-rotate -targetlibinfo -lcssa –die
Seq8	-lowerinvoke -basicaa -dce -loop-instsimplify -constmerge -simplifycfg -prune-eh -instsimplify -dce -inline -no-aa -sink -ipsccp -sink -lowerinvoke -ipconstprop -correlated-propagation -dse -no-aa -mergefunc -tbaa -early-cse -mergereturn -ipsccp -reassociate -inline -instsimplify -loop-unswitch -die -prune-eh -strip-dead-prototypes -adce -indvars -constmerge -jump-threading -indvars -functionattrs -jump-threading -loops -loop-idiom -reassociate -loop-reduce -globaldce -tbaa -loop-unroll -scalarrepl -loop-simplify -jump-threading -die -globalopt -constmerge -deadargelim -reassociate -mergereturn -lower-expect -loops -lowerswitch -loop-reduce -lowerswitch -loop-deletion
Seq9	-ipsccp -basiccg -dce -lcssa -correlated-propagation -constmerge -globaldce -dse -indvars -lower-expect -prune-eh -scalarrepl -basiccg -lowerswitch -memcpyopt -lazy-value-info -globalopt -ipsccp -gvn -lazy-value-info -reassociate -loweratomic -adce -die -loop-instsimplify -memdep -loop-idiom -gvn -mergereturn -sccp -functionattrs -deadargelim -loop-unroll -loop-simplify -dce -ipconstprop -loop-simplify -indvars -globalopt -adce -adce -scalarrepl-ssa -mergereturn -lazy-value-info -licm -loop-rotate -lowerinvoke -lcssa -targetlibinfo -die -tailcallelim -loop-unswitch -domtree -ipsccp -memdep -gvn -indvars -correlated-propagation

5. CONCLUSIONS

In this paper, a method to reduce the code size based on the signs table is proposed. Two thousand sequences are randomly generated. Each sequence compiled with 50 programs. Then the average program size for 50 programs is computed, where the sequence that gives the average program's size better than -Oz flag is extracted and put in the signs table. A set of eight unobserved programs are employed for validating the method. After computing the similarity of the unseen program, the sequences that belong to the most similar program are candidates to optimize it. The results show that for most eight unseen programs, the produced programs are 9% smaller than -Oz. In our future work, we need to address other types of features and study their effect on program size. Furthermore, we plan to investigate the power of machine learning techniques for code size reduction.

REFERENCES

[1] Al Baity, R.M., Alwan, E.H., Fanfakh, A. (2022). A top popular approach for the automatic tuning of compiler optimizations. In AIP Conference Proceedings, Al-Samawah, Iraq, p. 050013. https://doi.org/10.1063/5.0094022

[2] Ashouri, A.H., Bignoli, A., Palermo, G., Silvano, C., Kulkarni, S., Cavazos, J. (2017). Micomp: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning. ACM Transactions on Architecture and Code Optimization (TACO), 14(3): 1-28. https://doi.org/10.1145/3124452

[3] Almagor, L., Cooper, K.D., Grosul, A., Harvey, T.J., Reeves, S.W., Subramanian, D., Waterman, T. (2004). Finding effective compilation sequences. ACM SIGPLAN Notices, 39(7): 231-239. https://doi.org/10.1145/998300.997196

[4] Ashouri, A.H., Bignoli, A., Palermo, G., Silvano, C. (2016). Predictive modeling methodology for compiler phase-ordering. In Proceedings of the 7th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and the 5th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms, Prague Czech Republic, pp. 7-12. https://doi.org/10.1145/2872421.2872424

[5] Debray, S.K., Evans, W., Muth, R., De Sutter, B. (2000). Compiler techniques for code compaction. ACM Transactions on Programming languages and Systems (TOPLAS), 22(2): 378-415. https://doi.org/10.1145/349214.349233

[6] Almohammed, M.H., Alwan, E.H., Fanfakh, A.B. (2020). Programs features clustering to find optimization sequence using genetic algorithm. In Intelligent Computing Paradigm and Cutting-edge Technologies: Proceedings of the First International Conference on Innovative Computing and Cutting-edge Technologies (ICICCT 2019), Istanbul, Turkey, pp. 40-50. https://doi.org/10.1007/978-3-030-38501-9_4

[7] Cooper, K.D., McIntosh, N. (1999). Enhanced code

compression for embedded RISC processors. ACM SIGPLAN Notices, 34(5): 139-149. https://doi.org/10.1145/301631.301655

[8] Beszédes, Á., Ferenc, R., Gyimóthy, T., Dolenc, A., Karsisto, K. (2003). Survey of code-size reduction methods. ACM Computing Surveys (CSUR), 35(3): 223-267. https://doi.org/10.1145/937503.93750

[9] Pinkers, R.P., Knijnenburg, P.M., Haneda, M., Wijshoff, H.A. (2004). Statistical selection of compiler options. In The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, (MASCOTS 2004), Volendam, Netherlands, pp. 494-501. https://doi.org/10.1109/MASCOT.2004.1348305

[10] de Souza Xavier, T.C., da Silva, A.F. (2018). Exploration of compiler optimization sequences using a hybrid approach. Computing & Informatics, 37(1): 165-185. https://doi.org/10.4149/caL2018.1.165

[11] Ashouri, A.H., Mariani, G., Palermo, G., Silvano, C. (2014). A bayesian network approach for compiler auto-tuning for embedded processors. In 2014 IEEE 12th Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia), Greater Noida, India, pp. 90-97. https://doi.org//10.1109/ESTIMedia.2014.6962349

[12] Hoste, K., Eeckhout, L. (2008). Cole: Compiler optimization level exploration. In Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization, Boston, MA, USA, pp. 165-174. https://doi.org/10.1145/1356058.1356080

[13] Ashouri, A.H., Palermo, G., Cavazos, J., Silvano, C., Ashouri, A.H., Palermo, G., Silvano, C. (2018). The phase-ordering problem: A complete sequence prediction approach. Automatic Tuning of Compilers Using Machine Learning, pp. 85-113. https://doi.org/10.1007/978-3-319-71489-9_5

[14] Wang, Z., O'Boyle, M. (2018). Machine learning in compiler optimization. Proceedings of the IEEE, 106(11): 1879-1901. https://doi.org/10.1109/JPROC.2018.2817118

[15] da Silva, A.F., de Souza, L.D. (2019). Understanding the code transformation algorithms' impact. Journal of Computer Science., 15(11): 1678-1693. https://doi.org/10.3844/jcssp.2019.1678.1693

[16] Alhasnawy, L.H., Alwan, E.H., Fanfakh, A.B. (2020). Using machine learning to predict the sequences of optimization passes. In New Trends in Information and Communications Technology Applications: 4th International Conference, NTICT 2020, Baghdad, Iraq, pp. 139-156. https://doi.org/10.1007/978-3-030-55340-1_10

[17] Almohammed, M.H., Fanfakh, A.B., Alwan, E.H. (2020). Parallel genetic algorithm for optimizing compiler sequences ordering. In New Trends in Information and Communications Technology Applications: 4th International Conference, NTICT 2020, Baghdad, Iraq, pp. 128-138. https://doi.org/10.1007/978-3-030-55340-1_9

[18] Lattner, C., Adve, V. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In International Symposium on Code Generation and Optimization, CGO 2004, San Jose, CA, USA, pp. 75-86. https://doi.org/10.1109/CGO.2004.1281665

[19] Kim, B., Cho, Y., Hong, J. (2012). An efficient function inlining scheme for resource-constrained embedded systems. Journal of Information Science and Engineering, 28(5): 859-874.

[20] Jain, S., Bora, U., Kumar, P., Sinha, V.B., Purini, S., Upadrasta, R. (2019). An analysis of executable size reduction by LLVM passes. CSI Transactions on ICT, 7(2): 105-110. https://doi.org/10.1007/s40012-019-00248-5

[21] Cooper, K.D., Schielke, P.J., Subramanian, D. (1999). Optimizing for reduced code space using genetic algorithms. In Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems, Atlanta, Georgia, USA, pp. 1-9. https://doi.org/10.1145/314403.314414

[22] Foleiss, J.H., da Silva, A.F., Ruiz, L.B. (2011). The effect of combining compiler optimizations on code size. In 2011 30th International Conference of the Chilean Computer Science Society, pp. 187-194. https://doi.org/10.1109/SCCC.2011.25

[23] Haneda, M., Knijnenburg, P.M., Wijshoff, H.A. (2006). Code size reduction by compiler tuning. In Embedded Computer Systems: Architectures, Modeling, and Simulation: 6th International Workshop, SAMOS 2006, Samos, Greece, pp. 186-195. https://doi.org/10.1007/11796435_20

[24] Al Baity, R.M., Alwan, E.H., Fanfakh, A.B. (2021). A content based filtering approach for the automatic tuning of compiler optimizations. Turkish Journal of Computer and Mathematics Education (TURCOMAT), 12(6): 3913-3922. https://doi.org/10.17762/turcomat.v12i6.785

[25] Al Baity, R.M., Alwan, E.H., Fanfakh, A. (2022). A top popular approach for the automatic tuning of compiler optimizations. In AIP Conference Proceedings, 2398(1): AIP Publishing. https://doi.org/10.1063/5.0094022

[26] Cavazos, J., Fursin, G., Agakov, F., Bonilla, E., O'Boyle, M.F., Temam, O. (2007). Rapidly selecting good compiler optimizations using performance counters. In International Symposium on Code Generation and Optimization (CGO'07), San Jose, CA, USA, pp. 185-197. https://doi.org/10.1109/CGO.2007.32

[27] Sarwar, B., Karypis, G., Konstan, J., Riedl, J. (2001). Item-based collaborative filtering recommendation algorithms. In Proceedings of the 10th International Conference on World Wide Web, Hong Kong, pp. 285-295. https://doi.org/10.1109/CGO.2007.32

[28] de Souza Xavier, T.C., da Silva, A.F. (2018). Exploration of compiler optimization sequences using a hybrid approach. Computing & Informatics, 37(1): 165-185. http://doi.org/10.4149/cai_2018_1_165

NOMENCLATURE

LLVM	Low Level Virtual Machine
Os, Oz	Standard optimization sequence
IR	Intermedite Code
Clang	LLVM front end